Sun MPI 4.0 User's Guide: With CRE

901 San Antonio Road
Palo Alto, , CA 94303-4900
USA 650 960-1300 Fax 650 969-9131

Part No: 806-0296-10
June 1999, Revision A

Contents

Preface

This manual describes how to use Sun HPC ClusterTools 3.0 software to develop, execute, and debug programs on a Sun HPC cluster that is using the Sun Cluster Runtime Environment (CRE) 1.0 for job managment.

Note - If your cluster uses the LSF 3.2.3 workload management suite *instead* of the CRE, read the *Sun MPI 4.0 User's Guide: With LSF* instead of this manual.

Before You Read This Book

For information about writing MPI programs, refer to the *Sun MPI 4.0 Programming and Reference Guide*. Sun MPI 4.0 is part of the Sun HPC ClusterTools 3.0 suite of software. Product notes for Sun MPI are included in *Sun HPC ClusterTools 3.0 Product Notes*.

Using UNIX® Commands

This document may not contain information on basic UNIX commands and procedures such as creating directories and copying and deleting files.

See one or more of the following for this information:

- AnswerBook™ online documentation for the Solaris™ 2.6 or Solaris 7 software environment
- Other software documentation that you received with your system

Typographic Conventions

TABLE P–1 Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls --a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P–2 Shell Prompts

Shell	Prompt
C shell	*machine_name*%
C shell superuser	*machine_name*#
Bourne shell and Korn shell	$
Bourne shell and Korn shell superuser	#

Related Documentation

TABLE P–3 Related Documentation

Application	Title	Part Number
Sun HPC ClusterTools software	*Sun HPC ClusterTools 3.0 Product Notes*	805-6262-10
Sun HPC ClusterTools software	*Sun HPC ClusterTools 3.0 Administrator's Guide: With CRE*	806-0295-10
SCI	*Sun HPC 3.0 SCI Guide*	805-6263-10
Installation	*Sun HPC ClusterTools 3.0 Installation Guide*	805-6264-10
Sun MPI Programming	*Sun MPI 4.0 Programming and Reference Guide*	805-6269-10
Prism	*Prism 6.0 User's Guide*	805-6277-10
Prism	*Prism 6.0 Reference Manual*	805-6278-10
Sun S3L	*Sun S3L 3.0 Programming and Reference Guide*	805-6275-10

Sun Documentation on the Web

The `docs.sun.com` web site enables you to access Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at:

`http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

`docfeedback@sun.com`

Please include the part number of your document in the subject line of your email.

CHAPTER 1

Introduction

This manual explains how to execute Sun MPI applications on a Sun HPC cluster that is using the Sun Cluster Runtime Environment (CRE) 1.0 for job management.

Sun HPC ClusterTools 3.0 Software

Sun HPC ClusterTools 3.0 software is an integrated ensemble of parallel development tools that extend Sun's network computing solutions to high-end distributed-memory applications. Sun HPC ClusterTools products can be used either with the CRE or with LSF Suite 3.2.3, Platform Computing Corporation's resource-management software.

Note - If you are using LSF Suite instead of the CRE for workload management, you should be reading the *Sun MPI 4.0 User's Guide: With LSF* instead of this document.

The principal components of Sun HPC ClusterTools Software are described in "Sun Cluster Runtime Environment" on page 1 through "Sun S3L" on page 3.

Sun Cluster Runtime Environment

The CRE is a cluster administration and job launching facility. It provides users with an interactive command-line interface for executing jobs on the cluster and for obtaining information about job activity and cluster resources.

The CRE also performs load-balancing for programs running in *shared partitions*.

Note - Load balancing, partitions, and other related Sun HPC cluster concepts are discussed in “Fundamental CRE Concepts” on page 5.

Sun MPI and MPI I/O

Sun MPI is a highly optimized version of the Message-Passing Interface (MPI) communications library. Sun MPI implements all of the MPI 1.2 standard as well as a significant subset of the MPI 2.0 feature list. For example, Sun MPI provides the following features:

- Support for multithreaded programming.
- Seamless use of different network protocols; for example, code compiled on a Sun HPC cluster that has a Scalable Coherent Interface (SCI) network, can be run without change on a cluster that has an ATM network.
- Multiprotocol support such that MPI picks the fastest available medium for each type of connection (such as shared memory, SCI, or ATM).
- Communication via shared memory for fast performance on clusters of SMPs.
- Finely tunable shared memory communication.
- Optimized collectives for symmetric multiprocessors (SMPs).
- Prism support – Users can develop, run, and debug programs in the Prism programming environment.
- MPI I/O support for parallel file I/O.
- Sun MPI is a dynamic library.

Sun MPI and MPI I/O provide full F77, C, and C++ support and Basic F90 support.

Parallel File System

The Sun Parallel File System (PFS) component of the Sun HPC ClusterTools suite of software provides high-performance file I/O for multiprocess applications running in a cluster-based, distributed-memory environment.

PFS file systems closely resemble UFS file systems, but provide significantly higher file I/O performance by striping files across multiple PFS I/O server nodes. This means the time required to read or write a PFS file can be reduced by an amount roughly proportional to the number of file server nodes in the PFS file system.

PFS is optimized for the large files and complex data access patterns that are characteristic of parallel scientific applications.

Prism

Prism is the Sun HPC graphical programming environment. It allows you to develop, execute, debug, and visualize data in message-passing programs. With Prism you can

- Control various aspects of program execution, such as:
  - Starting and stopping execution.
  - Setting breakpoints and traces.
  - Printing values of variables and expressions.
  - Displaying the call stack.
- Visualize data in various formats.
- Analyze performance of MPI programs.
- Aggregate processes across multiprocess parallel jobs into meaningful groups, called process sets or *psets*.

Prism can be used with applications written in F77, F90, C, and C++.

Sun S3L

The Sun Scalable Scientific Subroutine Library (Sun S3L) provides a set of parallel and scalable functions and tools that are used widely in scientific and engineering computing. It is built on top of MPI and provides the following functionality for Sun MPI programmers:

- Vector and dense matrix operations (level 1, 2, 3 Parallel BLAS).
- Iterative solvers for sparse systems.
- Matrix-vector multiply for sparse systems.
- FFT
- LU factor and solve.
- Autocorrelation.
- Convolution / deconvolution.
- Tridiagonal solvers.
- Banded solvers.
- Eigensolvers.
- Singular value decomposition.
- Least squares.
- One-dimensional sort.

- Multidimensional sort.
- Selected ScaLAPACK and BLACS application program interface.
- Conversion between ScaLAPACK and S3L.
- Matrix transpose.
- Random number generators (linear congruential and lagged Fibonacci).
- Random number generator and I/O for sparse systems.
- Matrix inverse.
- Array copy.
- Safety mechanism.
- An array syntax interface callable from message-passing programs.
- Toolkit functions for operations on distributed data.
- Support for the multiple instance paradigm (allowing an operation to be applied concurrently to multiple, disjoint data sets in a single call).
- Thread safety.
- Detailed programming examples and support documentation provided online.

Sun S3L routines can be called from applications written in F77, F90, C, and C++.

Sun Compilers

Sun HPC ClusterTools 3.0 sofware supports the following Sun compilers:

- Sun WorkShop Compilers C/C^{++} v4.2 and v5.0
- Sun WorkShop Compilers Fortran v4.2 and v5.0

Solaris Operating Environment

Sun HPC ClusterTools software uses the Solaris 2.6 or Solaris 7 (32-bit or 64-bit) operating environment. All programs that execute under Solaris 2.6 or Solaris 7 execute in the Sun HPC ClusterTools environment.

Fundamental CRE Concepts

This section introduces some important concepts that you should understand in order to use the Sun HPC ClusterTools software in the CRE effectively.

Cluster of Nodes

As its name implies, the CRE is intended to operate in a Sun HPC cluster—that is, in a collection of Sun SMP (symmetric multiprocessor) servers that are interconnected by any Sun-supported, TCP/IP-capable interconnect. An SMP attached to the cluster network is referred to as a *node*.

The CRE manages the launching and execution of both serial and parallel jobs on the cluster nodes. For serial jobs, its chief contribution is to perform load balancing in shared partitions, where multiple processes can be competing for the same node resources. For parallel jobs, the CRE provides:

- A single job monitoring and control point.
- Load balancing for shared partitions.
- Information about node connectivity.
- Support for spawning of MPI processes.
- Support for Prism interaction with parallel jobs.

Note - A cluster can consist of a single Sun SMP server. However, executing MPI jobs on even a single-node cluster requires the CRE to be running on that cluster.

The CRE supports parallel jobs running on clusters of up to 64 nodes containing up to 256 CPUs.

Partitions

The system administrator can configure the nodes in a Sun HPC cluster into one or more logical sets, called *partitions*.

Note - The CPUs in a Sun HPC 10000 server can be configured into logical nodes. These domains can be logically grouped to form partitions, which the CRE uses in the same way it deals with partitions containing other types of Sun HPC nodes.

Any job launched on a partition will run on one or more nodes in that partition, but not on nodes in any other partition. Partitioning a cluster allows multiple jobs to be

executed on the partitions concurrently, without any risk of jobs on different partitions interfering with each other. This ability to isolate jobs can be beneficial in various ways: For example:

- If one job requires exclusive use of a set of nodes, but other jobs also need to execute at the same time, the availability of two partitions in a cluster would allow both needs to be satisfied.
- If a cluster contains a mix of nodes whose characteristics differ—such as having different memory sizes, CPU counts, or levels of I/O support—the nodes can be grouped into partitions that have similar resources. This would allow jobs that require particular resources to be run on suitable partitions, while jobs that are less resource-dependent could be relegated to less specialized partitions.

If you want your job to execute on a specific partition, the CRE provides you with the following methods for selecting the partition:

- Log in to a node that is a member of the partition.
- Set the environment variable `SUNHPC_PART` to the name of the partition.
- Use the `--p` option to the job-launching command, `mprun`, to specify the partition.

Note - These methods are listed in order of increasing priority. That is, setting the `SUNHPC_PART` environment variable overrides whichever partition you may be logged into. Likewise, specifying the `mprun --p` option overrides either of the other methods for selecting a partition.

It is possible for cluster nodes to not belong to any cluster. If you log in to one of these independent nodes and do not request a particular partition, the CRE will launch your job on the cluster's *default partition*. This is a partition whose name is specified by the `SUNHPC_PART` environment variable or is defined by an internal attribute that the system administrator is able to set.

The system administrator can also selectively enable and disable partitions. Jobs can only be executed on enabled partitions. This restriction makes it possible to define many partitions in a cluster, but have only a few active at any one time.

Note - It is also possible for a node to belong to more than one partition, so long as only one is enabled at a time.

In addition to enabling and disabling partitions, the system administrator can set and unset other partition attributes that influence various aspects of how the partition functions. For example, if you have an MPI job that requires dedicated use of a set of nodes, you could run it on a partition that the system administrator has configured to accept only one job at a time.

The administrator could configure a different partition to allow multiple jobs to execute concurrently. This shared paritition would be used for code development or other jobs that do not require exclusive use of their nodes.

Note - Although a job cannot be run across partition boundaries, it can be run on a partition plus independent nodes.

Load Balancing

The CRE load-balances programs that execute in shared partitions—that is, in partitions that allow multiple jobs to run concurrently.

When you issue the `mprun` command in a shared partition, the CRE first determines what criteria (if any) you have specified for the node or nodes on which the program is to run. It then determines which nodes within the partition meet these criteria. If more nodes meet the criteria than are required to run your program, the CRE starts the program on the node or nodes that are least loaded. It examines the one-minute load averages of the nodes and ranks them accordingly.

This load-balancing mechanism ensures that your program's execution will not be unnecessarily delayed because it happened to be placed on a heavily loaded node. It also ensures that some nodes won't sit idle while other nodes are heavily loaded, thereby keeping overall throughput of the partition as high as possible.

Jobs and Processes

When a serial program executes on a Sun HPC cluster, it becomes a Solaris process with a Solaris *process ID*, or *pid*.

When the CRE executes a distributed message-passing program it spawns multiple Solaris processes, each with its own pid.

The CRE also assigns a *job ID*, or *jid*, to the program. If it is an MPI job, the jid applies to the overall job. Job IDs always begin with a `j` to distinguish them from pids. Many CRE commands take jids as arguments. For example, you can issue an `mpkill` command with a signal number or name and a jid argument to send the specified signal to all processes that make up the job specified by the jid.

Parallel File System

From the user's perspective, PFS file systems closely resemble UNIX file systems. PFS uses a conventional inverted-tree hierarchy, with a `root` directory at the top and subdirectories and files branching down from there. The fact that individual PFS files are distributed across multiple disks managed by multiple I/O servers is transparent to the programmer. The way that PFS files are actually mapped to the physical storage facilities is based on file system configuration entries in the CRE database.

Using the Sun CRE

The balance of this manual discusses the following aspects to using the CRE:

- Choosing a partition and logging in – see Chapter 2.
- Executing programs – see Chapter 3.
- Obtaining information – see Chapter 4.
- Debugging programs – see Chapter 5.
- Performance Tuning and Profiling – see Chapter 6.

CHAPTER 2

Starting Sun MPI Programs

This chapter explains the basic steps for starting up message-passing programs on a Sun HPC cluster using the services provided by the CRE.

Logging In

Logging in to a Sun HPC 3.0 cluster is the same as logging in to any Sun server. That is, to log in on your local machine, just supply your user name at the login prompt and, if a password is required, the password. For remote logins, use `rlogin`.

Note - This differs from the login model found in the Sun HPC RTE 2.0 environment, which was based on two special login commands `tmlogin` and `tmtelnet`. In that model, you were expected to specify the name of the cluster that you wanted to be on.

You receive the standard Solaris login information, followed by a Solaris prompt:

```
Sun Microsystems Inc.   SunOS 5.6       Generic March 1999
Users: wmitty jthurb
node0%
```

You are now logged in to a node whose hostname is `node0`.

After Logging In

Once you are logged in to a Sun HPC cluster, you can issue any Solaris or Sun HPC commands, and you can execute any programs that will execute under Solaris 2.6 or Solaris 7 operating environments. See Chapter 3 for more information about job execution.

Writing Programs

You can perform program development on a Sun HPC cluster node or you can do it on any computer running a compatible Solaris operating environment.

Compiling and Linking Programs

If your program uses Sun HPC ClusterTools components, you must compile and link your program on a cluster that contains the ClusterTools software.

If you plan to use Prism to debug your program, include the `-g` option when you compile your program.

See the *Sun S3L 3.0 Programming and Reference Guide* and the *Sun MPI 4.0 Programming and Reference Guide* for information on linking in the Sun S3L and the Sun MPI libraries.

Issuing CRE Commands

The CRE provides commands that allow you to execute programs and obtain information about cluster resources and job activity. This section provides general information about issuing these commands. The commands are discussed in detail in the next two chapters.

The CRE commands are typically in the directory `/opt/SUNWhpc/bin`. If you are unable to execute them, you may need to add this directory to your path; check with your system administrator. The man pages for Sun HPC commands are in

`/opt/SUNWhpc/man`

If you cannot display these man pages, you may need to add this directory to your manpath.

CRE commands take options that consist of a dash followed by one or two letters. You can combine single-letter options that don't take arguments so long as they don't create ambiguity with multiletter options. For example, the command

```
% mprun --B --J
```

can also be written as

```
% mprun --BJ
```

Logging Out

To log out of the Sun HPC cluster, issue the command

```
% logout
```

CHAPTER 3

Executing Programs

This chapter describes how to issue commands to execute programs on a Sun HPC cluster. You can execute programs on any node or nodes in any partitions to which you have access. A major difference between the Sun HPC cluster and a collection of workstations is that the Sun Cluster Runtime Environment (CRE) provides you with a simple, interactive interface for specifying where and how your program should run.

All programs written for Solaris 2.6 or Solaris 7 can run without recompilation on a Sun HPC cluster.

Note - Running parallel jobs with the CRE is supported on up to 256 processors and up to 64 nodes.

Choosing Where to Execute

The Sun CRE provides you with considerable flexibility in choosing where you want your program to execute. For example, you can specify

- The partition in which you want to execute your program.
- The number of processes you want to start, and how you want to map them to nodes.
- The characteristics of the node or nodes on which you want to run — for example, the minimum amount of memory required or the maximum acceptable load.

See “Specifying Where a Program Is to Run ” on page 21 for additional information on specifying where a program is to run.

You can specify default execution criteria via the MPRUN_FLAGS environment variable; see “Specifying Default Execution Options” on page 14. You can also override these criteria via options to the mprun command.

Authentication Methods

Sun HPC Software includes two optional forms of user authentication that require the execution of user-level commands. The two methods are Kerberos Version 4 and DES. If one of these authentication methods is enforced on your Sun HPC cluster, use the commands listed in Table 3–1.

TABLE 3–1 User Commands Required by Authentication Methods

Authentication Method	Required Command
DES	While DES authentication is in use, you must issue the keylogin command before issuing any commands beginning with mp, such as mprun or mpps.
Kerberos 4	While Kerberos Version 4 authentication is in use, you must issue a kinit command before running any command beginning with mp, such as mprun or mpps.

See your system administrator for details.

Specifying Default Execution Options

You can use the environment variable MPRUN_FLAGS to specify one or more default options to the program execution command, mprun. Then, you need not specify any option contained in MPRUN_FLAGS. mprun will be interpreted as if the options contained in MPRUN_FLAGS were included on the command line (preceding any options that are on the command line).

You can override any default option by including a new value for the option on the mprun command line.

Note - For the `--R` option, the interaction between `MPRUN_FLAGS` and `mprun` is somewhat more complicated. This special relationship is “Expressing More Complex Resource Requirements” on page 23 and “Running on the Same Node(s) as a Another Specified Job” on page 30.

The setting of the environment variable can be any number of valid `mprun` options. If you use more than one word, enclose the list in quotation marks. These options are described in more detail in the remainder of the chapter and are listed in “`mprun` Options ” on page 19.

For example, the following makes `part2` the default partition to be used for `mprun`.

C shell

```
% setenv MPRUN_FLAGS "--p part2"
```

Bourne shell

```
# MPRUN_FLAGS = "--p part2"; export MPRUN_FLAGS
```

You can check the current setting of `MPRUN_FLAGS` by issuing the command `printenv`.

C shell

```
% printenv MPRUN_FLAGS
```

Bourne shell

```
# printenv MPRUN_FLAGS
```

All `MPRUN_FLAGS` settings can be overridden by specifying the corresponding option on the `mprun` command line.

In addition, the default partition setting can be determined in two other ways. If `-p` is not specified on the `mprun` command line and `MPRUN_FLAGS` is not set to a default partition, the default partition is

- First, the one where you are logged in.
- Failing that, the one specified by the system administrator via the cluster administration command `mpadmin`.

Executing Programs via mprun

This section provides general information about executing programs via mprun.

Execution via mprun is similar to standard Solaris program execution. For example,

- Your environment is used as if you executed the program from a traditional shell.
- Signals are treated as they are in standard Solaris; for multiprocess programs, if one process is killed via a signal, all processes are killed.
- You can run a program in the background:

```
% mprun a.out &
```

CRE commands do differ slightly from standard Solaris execution. These differences are discussed in “Moving mprun Processes to the Background” on page 16 through “SMP Characteristics of Sun HPC clusters” on page 18.

Moving mprun Processes to the Background

When you move either a process started with mprun or a script that issues mprun commands to the background, you must do one of the following:

- Redirect stdin away from the terminal.
- Specify the --n option to mprun so that standard in will be read from /dev/null. See “Specifying the Behavior of I/O Streams” on page 33 for a detailed discussion of standard I/O issues.

If you do not take one of these steps, the mprun process will contend with your shell for characters typed at the shell, leading to unexpected results.

Shell-Specific Actions

If you want to perform actions that are shell specific, such as executing compound commands, you must first invoke the appropriate shell as part of the mprun command. For example,

```
% mprun csh --c ‘echo $USER‘
```

or

```
% mprun csh --c 'cd /foo ; bar'
```

Core Files

Core files are produced as they normally are in Solaris. However, if more than one process dumps core in a multiprocess program, the resulting core file may be invalid.

Standard Output and Standard Error

By default, `mprun` handles standard output and standard error the way `rsh` does: The output and error streams are merged and are displayed on your terminal screen. Note that this is slightly different from the standard Solaris behavior when you are not executing remotely; in that case, the `stdout` and `stderr` streams are separate. You can obtain this behavior with `mprun` via the `--D` option. You can also specify other methods for handling I/O streams, including the three standard ones. See "Specifying the Behavior of I/O Streams" on page 33 for additional information.

File Descriptors

If your job consists of a large number of processes, you may need to consider the number of file descriptors the job is using and, if necessary, increase the default number available to you.

For merged standard I/O, each process in a job requires two descriptors. For separate `stderr` and `stdout` streams, each process requires three descriptors. You also need three file descriptors for interacting with your terminal.

You can find out the default number of file descriptors available in your shell by issuing the command

C shell

```
% limit descriptors
```

Bourne shell

```
# ulimit --n
```

The default for most shells is 64. This limits you to about 30 processes for merged standard I/O and about 20 processes for separate standard I/O. If this isn't sufficient, you can increase your limit by issuing the command

C shell

```
% limit descriptors 128
```

Bourne shell

```
# ulimit --n 128
```

Or you can set it to the maximum value

C shell

```
% unlimit descriptors
```

Bourne shell

```
# ulimit --n 'ulimit --Hn'
```

The file descriptor maximum in Solaris 2.*6* and Solaris 7 is 1024.

SMP Characteristics of Sun HPC clusters

Since your Sun HPC cluster consists of symmetric multiprocessors (SMPs), the CRE takes into consideration the number of CPUs per node by default. In general, `mprun` will assign more processes to larger SMPs. For information about how the CRE allocates processes to CPUs, see "When Number of Processes Exceeds Number of CPUs" on page 22 and "Default Process Spawning" on page 30.

Executing Programs

The basic format of the `mprun` command is

```
% mprun [options] [--] executable [args ... ]
```

Note - When the name of your program conflicts with the name of an `mprun` option, use the `--` (dash) symbol to separate the program name from the option list.

`mprun` Options

The following table lists and briefly describes the mprun options. Their use is described more fully in “Specifying Where a Program Is to Run ” on page 21 through “Specifying the Behavior of I/O Streams” on page 33.

TABLE 3–2 Options for `mprun`

Option	Meaning
–A *aout*	Execute `aout` and use a different argument as the *argv[0]* argument to the program. See “Specifying a Different Argument Vector” on page 41.
–B	Send `stderr` and `stdout` output streams to files. See “Specifying the Behavior of I/O Streams” on page 33.
–c *cluster_name*	Run on the specified cluster. See “Specifying the Cluster” on page 21.
–C *path*	Use the specified directory as the current working directory for the job. See “Changing the Working Directory” on page 40.
–D	Provide separate *stdout* and *stderr* streams. See “Specifying the Behavior of I/O Streams” on page 33.
–G *group*	Execute with the specified group ID or group name. See “Executing with a Different User or Group Name” on page 40.
–h	Display help. See “Getting Information” on page 40.
–i	Standard input to mprun is sent only to rank 0, and not to all other ranks.
–I *file_descr_string*	Use the specified I/O file descriptor string to control I/O stream handling. See “Specifying the Behavior of I/O Streams” on page 33.

TABLE 3–2 Options for `mprun` *(continued)*

Option	Meaning
–j *jid*	Run on the same node(s) as the job with job ID `jid`. See “Running on the Same Node(s) as a Another Specified Job” on page 30.
–J	Show the jid, cluster name, and number of processes after executing. See “Getting Information” on page 40.
–n	Read *stdin* from . See “Specifying the Behavior of I/O Streams” on page 33.
–N	Do not open any standard I/O connections. See “Specifying the Behavior of I/O Streams” on page 33.
`--`np *number*	Request the specified number of processes. See “Controlling Process Spawning” on page 22.
–Ns	Disable spawning of multiple processes from a job on SMPs; see “Default Process Spawning” on page 30.
–o	Prefix each output line with the rank of the process that wrote it.
`--`p *partition*	Run in the specified partition. See “Specifying the Partition” on page 21.
–R "*resource_string*"	Specify conditions for choosing nodes. See “Expressing More Complex Resource Requirements” on page 23.
–S	Settle for the available number of nodes (used with –np). See “Controlling Process Spawning” on page 22.
–U *user*	Execute with the specified user ID or user name. See “Executing with a Different User or Group Name” on page 40.
–V	Display version information. See “Getting Information” on page 40.
–W	Wrap the requested processes on the available CPUs (used with *–np*). See “Controlling Process Spawning” on page 22.
–Ys	Allow spawning on SMPs. See “Default Process Spawning” on page 30.
–Z *rank*	Run processes, by groups of size *rank*, together on the same node. (incompatible with –S and –W) See “Mapping MPI Ranks to Nodes” on page 31.

Specifying Where a Program Is to Run

The `mprun` command provides you with considerable flexibility in specifying where you want your job to run.

- "Specifying the Partition" on page 21 describes how to choose the partition in which a program is to run.
- "Specifying the Cluster" on page 21 describes how to choose the cluster on which you want your program to run.
- "Controlling Process Spawning" on page 22 describes how to specify how many processes are to be started and how they should be mapped to nodes.
- "Expressing More Complex Resource Requirements" on page 23 describes a syntax for specifying complex requirements that can't be encapsulated in the basic command-line options.

In cases where your specified requirements can be met by more than one node, the cluster chooses the least-loaded node, unless you have specified other sorting criteria.

Specifying the Partition

Use `mprun --p` to specify the partition in which you want your program to run. The partition must be in the enabled state. For example,

```
% mprun --p part2 a.out
```

specifies that `a.out` is to be run in the partition `part2`.

The `mpinfo` command will tell you the names of enabled partitions in the cluster, along with other useful information about cluster resources. See "`mpinfo`: Configuration and Status" on page 46 for a description of `mpinfo`.

Specifying the Cluster

By default, your job will run on the cluster where you are logged in.

If you are logged in on a machine that is connected to the Sun HPC cluster on which you want to run your job, but is not part of the cluster, use `mprun --c` *cluster_name* to specify the cluster.

Note - Use the hostname of the cluster's master node as the cluster name. You can find the cluster's master node by running `mpinfo --C` on any node in the cluster. See "Specifying the Partition" on page 44 for additional details.

Controlling Process Spawning

Specify the Number of Processes

Use the `--np` option to specify the number of processes you want to start; the default is 1. This option is typically used with a Sun MPI program.

For example,

```
% mprun --p part2 --np 4 a.out
```

specifies that you want four copies of `a.out` to start on the nodes of the partition named `part2`.

You can also specify 0 as the `--np` value. The CRE will start one process per CPU on each available CPU. Thus, if the partition `part2` has six available CPUs, the command

```
% mprun --p part2 --np 0 a.out
```

will start six copies of `a.out`.

Limit to One Process Per Node

Use the `--Ns` option to limit the number of processes to one per node. This prevents nodes from spawning more processes regardless of the number of CPUs they have.

When Number of Processes Exceeds Number of CPUs

When you request multiple processes (via the `--np` option), the CRE attempts to start one process per CPU. If you request more processes than the number of available CPUs, you must include either the `--W` or `--S` option. Otherwise, `mprun` will fail.

Use the `--W` option if you want the processes to *wrap*—that is, to allocate multiple processes to each CPU, which will execute their respective sets of processes one by one. For example, if the partition `part2` has six available CPUs and you specify

```
% mprun --p part2 --np 10 --W a.out
```

the CRE will start 10 processes on the six CPUs.

Note - When the CRE wraps processes, it distributes them according to load-balancing rules. Therefore, you will not be able to predict where they will execute.

If you prefer to have a certain number of processes started, but are willing to settle for however many CPUs are available, use the `--S` option. The CRE will start one process on each available CPU. Thus, if you issue the same command as above, but substitute `--S` for `--W`:

```
% mprun --p part2 --np 10 --S a.out
```

and six CPUs are available on `part2`, then six copies of `a.out` will start, one per CPU.

Note - If you specify –`np` *number*, but not –`np 0`, –`S`, or –`W`, and there are not enough nodes within the partition, the CRE will look for nodes outside the partition to make up the difference. To be elegible, an external node must be both enabled and independent. That is, the node must not be a member of another partition that is enabled. If you specify –`np 0`, –`S`, or –`W`, the search will be restricted to the partition you are in.

Expressing More Complex Resource Requirements

Use the `--R` option to express complex node requirements that are not accessible via the basic options discussed above.

The `--R` option takes a *resource requirement specifier* (RRS) as an argument. The RRS is enclosed in quotation marks and provides the settings for any number of attributes that you want to use to control the selection of nodes. You combine multiple attribute settings using the logical & (AND) and | (OR) operators.

The CRE parses the attribute settings in the order in which they are listed in the RRS, along with other options you specify. The CRE merges these results with the results of an internally specified RRS that controls load balancing.

Note - One option is an exception to this merging behavior, –j. This exception is discussed later.

The result is an ordered list of CPUs that meet the specified criteria. If you are starting a single process, the CRE starts the process on the CPU that's first in the list. If you are starting *n* processes, the CRE starts them on the first *n* CPUs, wrapping if necessary.

Note - Unless `--Ns` is specified, the RRS specifies node resources but generates a list of CPUs. If `--Ns` is specified, the list refers only to nodes.

Specifying Resource Attributes

Table 3–3 lists predefined attributes you can include in an RRS. Your system administrator may also have defined attributes specific to your Sun HPC cluster. You can see what settings these administrator-defined attributes have with the `mpinfo` command.

TABLE 3–3 Standard RRS Attributes

Attribute	Meaning
`cpu_idle`	Percent of time that the CPU is idle.
cpu_iowait	Percent of time that the CPU spends waiting for I/O.
cpu_kernel	Percent of time that the CPU spends in the kernel.
cpu_scale	Performance rating of the CPU.
cpu_swap	Percent of time that the CPU spends waiting for swap.
cpu_type	CPU architecture.
cpu_user	Percent of time that the CPU spends running user's program.
load1	Node's load average for the past minute.
load5	Node's load average for the past 5 minutes.

TABLE 3–3 Standard RRS Attributes *(continued)*

Attribute	Meaning
load15	Node's load average for the past 15 minutes.
manufacturer	Hardware manufacturer.
mem_free	Nodes's available memory, in Mbytes.
mem_total	Node's total physical memory, in Mbytes.
name	Node's hostname.
os_max_proc	Maximum number of processes allowed on the node, including cluster daemons.
os_arch_kernel	Node's kernel architecture.
os_name	Operating system's name.
os_release	Operating system's release number.
os_release_maj	The major number of the operating system's release number.
os_release_min	The minor number of the operating system's release number.
os_version	Operating system's version.
serial_number	Node's serial number.
swap_free	Node's available swap space, in Mbytes.
swap_total	Node's total swap space, in Mbytes.

The CRE recognizes two types of attributes, value and boolean.

Value-Based Attributes

Value attributes can take a literal value or a numeric value. Or, depending on the operator used, they may take no value.

- Attributes with a literal value take a name as a setting. Use an equal sign and the name after the attribute to show the setting. For example,

```
% mprun --R "name = hpc-demo" a.out
```

- Attributes with a numeric value include an operator and a value. For example,

```
% mprun -R "load5 < 4" a.out
```

specifies that you only want nodes whose individual load averages over the previous 5 minutes were less than 4.

- Attributes that use either << or >> take no value. For example,

```
% mprun -R "mem_total>>" a.out
```

specifies that you prefer nodes with the largest physical memory available.

Table 3–4 identifies the operators that can be used in RRS expressions.

TABLE 3–4 Operators Valid for Use in RRS

Operator	Meaning
<	Select all nodes where the value of the specified attribute is less than the specified value.
<=	Select all nodes where the value of the specified attribute is less than or equal to the specified value.
=	Select all nodes where the value of the specified attribute is equal to the specified value.
>=	Select all nodes where the value of the specified attribute is greater than or equal to the specified value.
>	Select all nodes where the value of the specified attribute is greater than the specified value.
!=	Attribute must not be equal to the specified value. (Precede with a backslash in the C shell.)
<<	Select the node(s) that have the lowest value for this attribute.
>>	Select the node(s) that have the highest value for this attribute.

The operators have the following precedence, from strongest to weakest:

```
unary --
*, /
+, binary --
```

```
=, !=, >=, <=, >, <, <<, >>
!
&, |
?
```

If you use the **<<** or **>>** operator, the CRE does not provide load balancing. In the previous example, the CRE would choose the node with the most free swap space, regardless of its load. If you use << or >> more than once, only the last use has any effect—it overrides the previous uses. For example,

```
% mprun --R "mem_free>> swap_free>>" a.out
```

initially selects the nodes that have the most free memory, but then selects nodes that have the largest amount of available swap space. The second selection may yield a different set of nodes than were selected initially.

You can also use arithmetic expressions for numeric attributes anywhere. For example,

```
% mprun --R "load1 / load5 < 2" a.out
```

specifies that the ratio between the one-minute load average and the five-minute load average must be less than 2. In other words, the load average on the node must not be growing too fast.

You can use standard arithmetic operators as well as the C `?:` conditional operator.

Note - Because some shell programs interpret characters used in RRS arguments, you may need to protect your RRS entries from undesired interpretation by your shell program. For example, if you use `csh`, write "`-R \!private`" instead of "`-R !private`".

Boolean Attributes

Boolean attributes are either true or false. If you want the attribute to be true, simply list the attribute in the RRS. For example, if your system administrator has defined an attribute called `ionode`, you can request a node with that attribute:

```
% mprun --R "ionode" a.out
```

If you want the attribute to be false (that is, you do not want a resource with that attribute), precede the attribute's name with !. (Precede this with a backslash in the C shell; the backslash is an escape character to prevent the shell from interpreting the exclamation point as a "history" escape.) For example,

```
% mprun --R "\!ionode" a.out
```

For example,

```
% mprun --R "mem_free > 256" a.out
```

specifies that the node must have over 256 megabytes of available RAM.

```
% mprun --R "swap_free >>" a.out
```

specifies that the node picked must have the highest available swap space.

Examples

Here are some examples of the --R option in use.

The following example specifies that the program must run on a node in the partition with 512 Mbytes of memory:

```
% mprun --p part2 --R "mem_total=512" a.out
```

The following example specifies that you want to run on any of the three nodes listed:

```
% mprun --R "name=node1 | name=node2 | name=node3" a.out
```

The following example chooses nodes with over 300 Mbytes of free swap space. Of these nodes, it then chooses the one with the most total physical memory:

```
% mprun --R "swap_free > 300 & mem_total>>" a.out
```

The following example assumes that your system administrator has defined an attribute called `framebuffer`, which is set (TRUE) on any node that has a frame buffer attached to it. You could then request such a node via the command

```
% mprun --R "framebuffer" a.out
```

The –R Option and MPRUN_FLAGS

With the exception of the `--j` option, specifying `--R` on the command line as well as in the `MPRUN_FLAGS` environment variable *combines* the two sets of values—that is, the command line does not override the environment variable settings. For example, if you have

```
% setenv MPRUN_FLAGS "--R "load1 < 1""
```

and issue the command

```
% mprun --R "load5 < 1" --R "load15 < 1" a.out
```

this would be the same as issuing the command

```
% mprun --R "(load1<1) & (load5<1) & (load15<1)" a.out
```

This combining behavior does not happen with the `--j` option. When `--j` is specified by `MPRUN_FLAGS` as well as on the `mprun` command line, the command line use overrides the environment variable setting.

Running on the Same Node(s) as a Another Specified Job

Use the `--j` option to specify that the program you want to execute should run on the same node or nodes as a particular job ID (`jid`). For example, to run `a.out` on the same node(s) as a job whose job ID is `85`, issue the command

```
% mprun --j 85 a.out
```

If `--j` follows the `--np` or `--R` option on the command line, it overrides those options. If `--np`, together with `--W` or `--S`, follows `--j` on the command line, `--j` determines which nodes to run on, while the other options determine the number of processes to map onto these nodes.

You can use the `mpps` command to find out the job ID of any job.

Default Process Spawning

By default, `mprun` spawns multiple processes on SMPs. For example, if you have a two-node partition in which one node has two CPUs and the other has four CPUs, then the command

```
% mprun --np 6 a.out
```

runs six copies of `a.out`, two on the two-CPU node and four on the four-CPU node.

The `--j` and `--R` options override this behavior.

Alternatively, you can use the `--Ns` option to disable spawning of processes on individual CPUs of a node. Instead, `--Ns` will cause only one process to be started on each node.

Use the `--Ys` option to force spawning on nodes when used with `--R`. `--Ys` does not override `--j`.

Mapping MPI Ranks to Nodes

Using the `--Z` Option

The `--Z` option causes the CRE to organize a job's processes into subsets of a specified size and to group all processes in a subset on the same node. You specify the subset size with a numerical argument to `--Z`. For example,

```
% mprun --Z 3 --np 8 a.out
```

groups the job's processes by threes. These groups *may* be distributed onto different nodes, but there is no guarantee that they will be; two or more groups may be started on the same CPU.

Note - The `--Z` option is incompatible with the `--S` and `--W` options.

Using RRS to Map Ranks to Nodes

You can construct an RRS expression (see "Expressing More Complex Resource Requirements" on page 23) that causes `mprun` to distribute a specified number of processes (MPI ranks) to a set of nodes in a specified order. The RRS expression assigns to each node in the set a single-character alias preceded by a number, which together make up a sequence of count/alias pairs. For example:

```
"[2a2b2c2d]:a.name=hpc-node0 & b.name=hpc-node1 & c.name=hpc-node2 & d.name=hpc-node3"
```

The number that precedes a node's alias tells the CRE how many processes to start on that node. In this example, it assigns two processes to each of the nodes defined by the aliases `a`, `b`, `c`, and `d`. This number can be different for each node, but it *must not* exceed the number of CPUs on that node.

The CRE distributes processes to the nodes in the order in which they are listed in the RRS expression, starting the rank 0 process on the first node in the list. Once the prescribed number of processes have been started on the first node, the CRE moves to the second node and then to subsequent nodes, starting the specified number of

processes on each node in turn. An alias cannot be repeated in the sequence, but one node can be defined with more than one alias.

The RRS rank-mapping expression must satisfy the following conditions:

- Up to 26 node aliases can be defined; aliases are *not* case-sensitive. Every node alias must be preceded by a number, which may have more than one digit.
- The number of processes assigned to a given node *cannot* be greater than the number of CPUs on that node.
- The `--np` value *cannot* be greater than the total number of processes allocated by the RRS expression. You cannot use use the `--W` option to get around this restriction by wrapping the processes.

The following example shows this technique being applied on a 4x4 partition. Two processes are started on each of four, four-CPU nodes.

```
% mprun --o --np 8 --R "[2a2b2c2d]:a.name=hpc-node0 & b.name=hpc-node1 & c.name=hpc-node2 & d.nam
r0:hpc-node0
r1:hpc-node0
r2:hpc-node1
r3:hpc-node1
r4:hpc-node2
r5:hpc-node2
r6:hpc-node3
r7:hpc-node3
```

The `--o` option prepends each output line with the MPI rank of the process that writes it. Two CPUs on each node are not participants in this job.

The next example shows different numbers of processes being allocated to each node. One process is started on the first node, two on the second, and so forth.

```
% mprun --o --np 10 --R "[1a2b3c4d]:a.name=hpc-node0 & b.name=hpc-node1 & c.name=hpc-node2 & d.na
r0:hpc-node0
r1:hpc-node1
r2:hpc-node1
r3:hpc-node2
r4:hpc-node2
r5:hpc-node2
r6:hpc-node3
r7:hpc-node3
r8:hpc-node3
r9:hpc-node3
```

The following example shows the error message that is returned when the number of processes assigned to a node exceeds the number of CPUs on that node.

```
% mprun --o --np 6 --R "[2a1b3c]:a.name=hpc-node0 & b.name=hpc-node1 & c.name=hpc-node0" uname --
mprun: no_mp_jobs: No nodes in partition satisfy RRS
```

In this case, the node `hpc-node0` is aliased twice—as `2a` and `3c`—so that it can be repeated in the sequence. This use of multiple aliases is legal, but `hpc-node0` has four CPUs and the total number of processes assigned by `2a` and `3c` is five, which violates the second condition listed above.

The next example shows what happens when an alias does not start with a number. In this case, the alias for `hpc-node0` violates the first condition listed above.

```
% mprun --o --np 6 --R "[a2b3c]:a.name=hpc-node0 & b.name=hpc-node1 & c.name=hpc-node2" uname --n
mprun: no_mp_jobs: No nodes in partition satisfy RRS
```

Specifying the Behavior of I/O Streams

Introducing `mprun` I/O

By default, all standard output (`stdout`) and standard error (`stderr`) from an `mprun`-launched job will be merged and sent to `mprun`'s standard output. This is ordinarily the user's terminal. Likewise, `mprun`'s standard input (`stdin`) is sent to the standard input of all the processes.

You can redirect `mprun`'s standard input, output, and error using the standard shell syntax. For example,

```
% mprun --np 4 echo hello > hellos
```

You can also change what happens to the standard input, output, and error of each process in the job. For example,

```
% mprun echo hello > message
```

sends `hello` across the network from the echo process to the `mprun` process, which writes it to a file called `message`.

The `mprun` command's own options allow you to control I/O in other ways. For example, rather than making remote processes communicate with `mprun` (when it may not be necessary), you can make each process write to or read from a file on the node on which it is running. For example, you can make each process send its standard output or standard error to a file on its own node. In the following example, each node will write `hello` to a local file called `message`:

```
% mprun --I "1w=message" echo hello
```

mprun also provides options that you can use to control standard output and standard error streams. For example, you can

- Use the --D option to make the standard error from each process go to the standard error of mprun, instead of its standard output. For example,

```
% mprun --D a.out
```

sends standard output from a.out to the standard output of mprun and sends the standard error of a.out to the standard error of mprun.

- Use the --B option to merge the standard output and standard error streams from each process and direct them to files named out.*jid*.*rank*, where *jid* is the job ID of the job and *rank* is the rank of this process within the job. The files are located in the job's working directory. There is no standard input stream.
- Use the --N option to shut off all standard I/O to all the processes. That is, with this option, you specify that there are to be no stdin, stdout, and stderr connections. Use the –N option for situations in which standard I/O is not necessary; you can reduce the overhead incurred by establishing standard I/O connections for each remote process and then closing those connections as each process ends.
- Use the --n option to cause stdin to be read from /dev/null. This can be useful when running mprun in the background, either directly or through a script. Without --n, mprun will block in this situation, even if no reads are posted by the remote job. When --n is specified, the user process encounters an EOF if it attempts to read from stdin. This is comparable to the behavior of the --n option to rsh.

Note - The set of mprun options that control stdio handling cannot be combined. These options override one another. If more than one is given on a command line, the last one overrides all of the rest. The relevant options are: -D, -N, -B, -n, -i, -o, and -I.

Creating a Custom Configuration

Use the --I option to specify a custom configuration for the I/O streams associated with a job, including standard input, output, and error. The --I option takes as an argument a comma-separated series of *file descriptor strings*. These strings specify what is to happen with each of the job's I/O streams.

In Solaris, each process has a numbered set of *file descriptors* associated with it. The standard I/O streams are assigned the first three file descriptors:

- 0 – standard input (stdio)

- `1` – standard output (`stdout`)
- `2` – standard error (`stderr`)

The argument list to `-I` can include a string for each file descriptor associated with a job; if any file descriptor is omitted, its stream won't be connected to any device.

Restriction: If you include strings to redirect both standard output and standard error, you must also redirect standard input. If the job has no standard input, you can redirect file descriptor `0` to `/dev/null`.

The file descriptor strings in the `-I` argument list can be in any order. Quotation marks around the strings are optional.

File Descriptor Attributes

The file descriptor string assigns one or more of the following attributes to a file descriptor:

- `r` – File descriptor is to be read from.
- `w` – File descriptor is to be written to.
- `p` – File descriptor is to be attached to a pseudo-terminal (pty).

You must specify either `r` or `w` for each file descriptor — that is, whether the file descriptor is to be written to or read from.

Thus, the string

```
5w
```

means that the stream associated with file descriptor `5` is to be written. And

```
0rp
```

means that the standard input is to be read from the pseudo-terminal.

If you use the `p` (pty) attribute, you must have one `rp` and one `wp` in the complete series of file descriptor strings. In other words, you must specify both reading from and writing to the pty. No other attributes can be associated with `rp` and `wp`.

The following attributes are output-related and thus can only be used in conjunction with `w`:

- `l` – Line-buffered output.
- `t` – Tag the line-buffered output with process rank information.
- `a` – Stream is to be appended to the specified file.

Note - NFS does not support append operations.

Use the `l` attribute in combination with the `w` attribute to line-buffer the output of multiple processes. This takes care of the situation in which output from one process arrives in the middle of output from another process. For example,

```
% mprun --np 2 echo "Hello"
HelHello
lo
```

With the `l` attribute, you ensure that processes don't intrude on each other's output. The following example shows how using the `l` attribute could prevent the problem illustrated in the previous example:

```
% mprun --np 2 --I "0r, 1wl" echo "Hello"
Hello
Hello
```

Use the `t` attribute in place of `l` to force line-buffering and, additionally, to prefix each line with the rank of the process producing the output. For example,

```
% mprun --np 2 --I "0r, 1wt" echo "Hello"
r0:Hello
r1:Hello
```

The `b` attribute is input-related and thus can be used only in combination with `r`. In multiprocess jobs, the `b` attribute specifies that input is to go only to the first process, rather than to all processes, which is the default behavior.

The `m` attribute pertains to reading from a pseudo-terminal and thus can be used only with `rp`. The `m` attribute in combination with `rp` causes keystrokes to be echoed multiple times when multiple processes are running. The default is to display multiple keystrokes only once.

File Descriptor String Syntax

You can direct one file descriptor's output to the same location as that specified by another file descriptor by using the syntax

fd attr=@other_fd

For example,

`2w=@1`

means that the standard error is to be sent wherever the standard output is going. You cannot do this for a file descriptor string that uses the `p` attribute.

If the behavior of the second file descriptor in this syntax is changed later in the `-I` argument list, the change does not affect the earlier reference to the file descriptor. That is, the `-I` argument list is parsed from left to right.

You can tie a file descriptor's output to a file by using the syntax

fd attr=filename

For example,

`10w=output`

says that the stream associated with file descriptor `10` is to be written to the file `output`. Once again, however, you cannot use this feature for a file descriptor defined with the `p` attribute.

In the following example, the standard input is read from the pty, the standard output is written to the pty, and the standard error is sent to the file named `errors`:

```
% mprun --I "0rp,1wp,2w=errors" a.out
```

If you use the `w` attribute without specifying a file, the file descriptor's output is written to the corresponding output stream of the parent process; the parent process is typically a shell, so the output is typically written to the user's terminal.

For multiprocess jobs, each process creates its own file; the file is opened on the node on which the process runs.

Note - If output is redirected such that multiple processes open the same file over NFS, the processes will overwrite each other's output.

In specifying the individual file names for processes, you can use the following symbols:

- `&J` – The job ID of the job
- `&R` – The rank of the process within the job

The symbols will be replaced by the actual values. For example, assuming the job ID is 15, this file descriptor string

`1w=myfile.&J.&R`

redirects standout output from a multiprocess job to a series of files named `myfile.15.0`, `myfile.15.1`, `myfile.15.2`, and so on, one file for each rank of the job.

In the following example, there is no standard input (it comes from `/dev/null`), and the standard output and standard error are written to the files `out.`*`job`*`.`*`rank`*:

```
% mprun --I "0r=/dev/null,1w=out.&J.&R,2w=@1" a.out
```

This is the behavior of the `--B` option. See "Introducing `mprun` I/O " on page 33. Note the inclusion in this example of a file descriptor string for standard input even though the job has none. This is required because both standard output and standard error are redirected.

`mprun` *Options versus Shell Syntax*

The default I/O behavior of `mprun` (merged standard error and standard output) is equivalent to

```
% mprun --I "0rp,1wp,2w=@1" a.out
```

The `--D` option provides separate standard output and standard error streams; it is equivalent to:

```
% mprun --I "0rp,1wp,2w" a.out
```

You can use the `--o` option to force each line of output to be prepended with the rank of the process writing it. This is equivalent to

```
% mprun -I "0rp,1wt,2w=@1" a.out
```

If you redirect output to a shared file, you must use standard shell redirection rather than the equivalent `--I` formulation (`--I "lwt=outfile"`). The same restriction also applies to the linebuffer formulation (`--I "lwt=outfile"`).

For example, the following command line concatenates the outputs of the individual processes of a job and writes them to `outfile.dat`:

```
% mprun -np 4 myprogram > outfile.dat
```

The following command line concatenates the outputs of the individual processes and appends them to the previous content of the output file:

```
% mprun -np 4 myprogram >> outfile.dat
```

The following table describes three `mprun` command-line options that provide the same control over standard I/O as some `--I` constructs, but are much simpler to express. Their `--I` equivalents are also shown.

TABLE 3–5 `mprun` Shortcut Summary

Command	Description
mprun –i	Standard input to `mprun` is sent only to rank 0, and not to all other ranks. Equivalent to *mprun –I "0rpb,1wp,2w=@1"* a.out
mprun –B	Standard output and standard error are written to the file `out.`*job.rank*. Equivalent to *mprun –I "0r=/dev/null,1w=out.&J.&R,2w=@1" a.out*
mprun –o	Use line buffering on standard output, prefixing each line with the rank of the process that wrote it. Equivalent to *mprun –I "0rp,1wt,2w=@1"* a.out

Note - Specifying `-o` (forcing processes to prepend rank on output lines), or the equivalent `-I` syntax (such as `-I1wt`) will not work if redirection is also specified with `-I` (such as with `-I1w=outfile`). Use the standard shell redirection operator instead.

These shortcuts are not exact substitutions. The CRE uses ptys correctly, whether the –I option is present or absent. Also, the CRE merges standard error with standard output when it is appropriate. If either `stderr` or `stdout` is redirected (but not both), ptys are not used and `stderr` and `stdout` are separated. If both `stderr` and `stdout` are redirected, ptys are still not used, but `stderr` and `stdout` are combined.

Caution Regarding the Use of `--i` Option

Use the `-i` option to `mprun` with caution, since the `-i` option provides only one `stdin` connection (to rank 0). If that connection is closed, keyboard signals are no longer forwarded to those remote processes. To signal the job, you must go to another window and issue the `mpkill` command. For example, if you issue the command `mprun --np 2 --i cat` and then type the `Ctrl-d` character (which causes `cat` to close its `stdin` and exit), rank 0 will exit. However, rank 1 is still running, and can no longer be signaled from the keyboard.

Changing the Working Directory

Use the `--C` option to specify the path of an alternative working directory to be used by the program. If you don't specify `--C`, the default is the current working directory. For example,

```
% mprun --C /home/collins/bin a.out
```

changes the working directory for `a.out` to `/home/collins/bin`.

Executing with a Different User or Group Name

Use the `--U` option to execute with the specified user ID or user name. For example,

```
% mprun --U traveler a.out
```

executes `a.out` as the user `traveler`.

Use the `--G` option to execute with the specified group ID or group name.

```
% mprun --G qa-team a.out
```

executes `a.out` as the group `qa-team`.

You must have the appropriate level of permissions to use these options. For example, you must belong to the group you specify, or be the superuser.

Getting Information

Use the `--h` option to display a list of `mprun` options and their meanings.

Use the `--V` option to display the command's version number.

If you specify either `--h` or `--V`, it must be the only option on the command line.

Use the `--J` option to display the program's jid, along with the name of the cluster and the number of processes, after executing `mprun`.

Specifying a Different Argument Vector

By default, `mprun` passes the vector of a program's command-line arguments to the program in the standard way. For example, if you issue the command

```
% mprun a.out arg1 arg2
```

`mprun` passes an array in which the name of the program, `a.out`, is the first element (`argv[0]`), and `arg1` and `arg2` are the second and third elements.

In cluster-level programming, it is sometimes useful to specify an `argv[0]` that is not the name of the program. You can use the `--A` option to do this. The argument to `--A` is the name of the program to be executed. You can then follow this with an argument of your choice in the `arg0` position. For example, if you want to pass `newarg` as the `argv[0]` to the program `a.out`, along with `arg1` and `arg2`, you could issue the command

```
% mprun --A a.out newarg arg1 arg2
```

Exit Status

The exit status of `mprun` specifies the number of processes that exited with nonzero exit status.

Omitting `mprun`

You can execute a serial program without using `mprun`. For example, you could simply type

```
% a.out
```

In that case, the program executes locally, on the node where you are logged in. By doing this, however, you give up the benefits of load-balancing provided by the CRE.

Note - You cannot run Sun MPI programs in this way; you must use `mprun`.

Sending a Signal to a Process

The `mpkill` command is comparable to the Solaris `kill` command. You use it to terminate all processes of the jobs with the specified job IDs running on the Sun HPC cluster, or to send a signal to it.

You can send any standard Solaris signal. Use the `--l` option to obtain a list of the supported signals, or the `--d` option to list them along with brief descriptions.

Specify the signal's name or number, followed by the job ID, to send that signal to the job. For example,

```
% mpkill --CONT 59
```

sends a `SIGCONT` to the processes that constitute job `59`.

Issuing `mpkill` without specifying a signal sends a `SIGTERM` to the job.

To find out a job's job ID, use the command `mpps` or the `--J` option to `mprun`.

`mpkill` returns the following status values:

- `0` – The command executed successfully.
- `1` – An error was encountered during execution. For example, the job was not known.
- `2` – The command was partially successful. This typically occurs when you send a signal to a job in which one or more of the processes has already exited and therefore could not receive the signal.

Note, this is usually not an error, since the reason you are using `mpkill` is most likely to eliminate a job that has hung in this intermediate state.

CHAPTER 4

Getting Information

The CRE user interface includes two commands for obtaining information about a Sun HPC cluster's configuration (mpinfo) and information about jobs running on the cluster (mpps).

mpps: Finding Out Job Status

The mpps command is comparable to the Solaris ps command. It returns information about jobs and processes currently running on the Sun HPC cluster.

By default mpps shows basic information about the user's jobs currently running in the default partition. For example,

```
% mpps
   JID   NPROC  UID   STATE  AOUT
   41    3      slu   RUN    AAA
   46    4      slu   EXNG   tmp
   49    1      slu   EXIT   tmp
   99    9      slu   EXNG   uname
   100   9      slu   EXNG   uname
```

In the response,

- JID is the executing program's job ID.
- NPROC is the number of processes in the job.
- UID is the user ID of the person who executed the program.
- STATE is the execution status of the job's processes. (See below for a list of possible process states.)
- AOUT is the name of the executable program.

Table 4–1 lists the states reported by `mpps`. Some states refer only to jobs, some only to processes, and some to both. (See “Displaying Process Information” on page 45.)

TABLE 4–1 Job and Process States

State	mpps Display	Meaning
CORE	CORE	The job or process exited due to a signal and core was dumped.
COREING	CRNG	The job is exiting due to a signal. The first process to die dumped core.
EXIT	EXIT	The job or process exited normally.
EXITING	EXNG	The job is exiting. At least one process exited normally.
FAIL	FAIL	The job or process failed on startup or was aborted.
FAILING	FLNG	Initialization of the job failed, or a job abort has been signaled.
ORPHAN	ORPHAN	The process has been “orphaned,” that is, the node on which it exists has gone offline.
RUNNING	RUN	The job or process is running.
SEXIT	SEXIT	The job or process exited due to a signal.
SEXITING	SEXNG	The job is exiting due to a signal. The first process to die was killed by a signal. At least one of its processes is still in the *RUN* state.
SPAWNING	SPAWN	The job or process is being spawned.
STOP	STOP	The job or process is stopped.

Use the `--f` option to display, in addition, the start time for each job and the job’s arguments.

Use the `--e` option to display information on all jobs, not just your jobs.

Specifying the Partition

To show information about jobs running in all partitions, use the `--A` option.

To show information about jobs running in a specific partition, use the `--a` option, followed by the name of the partition.

Displaying Process Information

Use the `--p` option to also view information about the processes that make up the jobs. The process information is listed below each job. For example,

```
% mpps --p
   JID  NPROC  UID   STATE  AOUT   RANK  PID    STATE  NODE   2320    4   shaw  RUN    sleep  0
```

In this example,

- `RANK` is the process's rank within the job.
- `PID` is the process's process ID.
- `STATE` is the process's execution status.
- `NODE` is the node on which the process is running.

Displaying Specific Process and Job Information

You can also use the `--P` option to display one or more specific process values and the `–J` option to display one or more job values. Separate multiple values either with spaces or with commas and no spaces.

Arguments to `--P` are

- `rank` – the rank of the process within the job.
- `pid` – the process's process ID.
- `state` – the current execution state of the process.
- `iod` – the process ID of the I/O daemon for this process.
- `load` – the load on the node on which the process is executing.
- `node` – the name of the node on which the process is executing.

You can list these via the `--lp` option.

Arguments to `--J` are

- `part` – the name of the partition in which the job will run.
- `jid` – the job's unique ID, which can be used as an argument to `mpkill.`
- `nproc`– the number of processes requested (the actual number of processes started may differ if the `–W` or `–S` flags were used with `mprun`).
- `uid` – the user on whose behalf the job will be run (normally the user who submitted the job; see the `–U` flag to `mprun` for details).

- gid – the group on whose behalf the job will be run (normally the group of the user who submitted the job; see the –G flag to mprun for details).
- state – there are six states:
  - BUILD – The job is being submitted.
  - WAIT – The job is waiting to run.
  - SPAWN – The job is preparing to run.
  - RUN – The job is running.
  - RSTRT – The job has been killed because one of the nodes on which it was running went down; the job will be restarted.
- running – the number of processes actually running for this job. This is not always equal to the number of processes started for this job, since processes that have exited are not counted.
- wkdir – the directory in which the job's processes will be (or were) started.
- aout – the name of the program to be run.
- paout – the full path of the program to be run.
- ctime – the job creation time (when mprun was invoked for the job).
- args – the command-line arguments for the program to be run.
- stime – the time the job was started.
- prio – the job priority (higher numbers run first).

mpinfo: Configuration and Status

Use the mpinfo command to display information about the configuration of partitions and nodes, and status information about nodes.

Overview

You can display information on all partitions or nodes, or on any subset of them. You can either list the partitions or nodes, or you can use the --R option, along with a resource requirement specifier (RRS), to have the CRE determine which objects should be displayed. See "Expressing More Complex Resource Requirements" on page 23 for information on RRSs. If you specify a partition, you must include only partition attributes in the RRS; if you specify a node, you must use only node attributes.

Use the `--A` option to specify an attribute whose value you want to display. If you want to display more than one attribute, separate them by commas with no spaces. Alternatively, you can issue multiple `--A` options on the same command line. If you omit `--A`, `mpinfo` displays values for a default set of attributes.

Use the `--v` option to display information about all attributes for one or more partitions or nodes. These include attributes defined by the system administrator.

When a Boolean attribute is displayed, `yes` indicates that the attribute is set, and `no` indicates that the attribute is not set.

Partitions

Use the `--P` option to display information for all partitions.

Use the `--p` option, followed by the name of the partition, to display information about an individual partition. To display information about multiple partitions, list the names, either separating them with commas and no spaces or enclosing the list in quotation marks.

Partition attributes whose settings you can view via `mpinfo` are shown in Table 4–2; the heading displayed for each attribute is shown in parentheses after its description.

The following summarizes various points discussed earlier.

- You can specify one or more of these attributes via the `--A` option, or as part of an RRS as an argument to the `--R` option. You can use either the attribute's real name or, in some cases, a shorter version.
- For attributes that are defined as negatives (for example, `no_logins`), you can specify a positive version (for example, `logins`) for `--A`.
- You can list the settings of all attributes (including any system administrator–defined attributes) on a per-partition basis via the `--v` option.
- You can list the names and brief descriptions of these attributes via the `--lp` option.

TABLE 4–2 Partition attributes available via `mpinfo`

Attribute (`mpadmin` form)	**Description (`mpinfo` output heading)**
`enabled`	Set if the partition is enabled, that is, if it is ready to accept jobs (*ENA*).
`maxt`	Maximum number of simultaneously running processes allowed on each node of the partition (*MAXT*).

TABLE 4–2 Partition attributes available via `mpinfo` *(continued)*

Attribute (`mpadmin` form)	**Description (`mpinfo` output heading)**
`name`	Name of the partition (*NAME*).
`login`	Allow logins. When `login` is set, LOG is set. Note that this is the inverse of the `mpadmin` meaning. (*LOG*).
`mp`	Allow multinode jobs. When no_mp_jobs is unset, MP is set. Note that this is the inverse of the `mpadmin` meaning. (*MP*).
`nodes`	Number of nodes in the partition (*NODES*).

The following example illustrates the default `mpinfo` output for partitions:

```
% mpinfo --P
  NAME         NODES: Tot(cpu) Enb(cpu) Onl(cpu) ENA LOG MP
  part10              1(  4)   1(  4)   1(  4) no  yes yes
  part11              1(  4)   1(  4)   1(  4) yes yes yes
```

The following example displays the names, numbers of nodes, and enabled status for all partitions:

```
% mpinfo --A name,enabled,nodes --P
 NAME         ENA NODES: Tot(cpu) Enb(cpu) Onl(cpu)
 part10       no           1(  4)   1(  4)   1(  4)
 part11       yes          1(  4)   1(  4)   1(  4)
```

Nodes

Use the `--N` option to display information about all nodes.

Use the `--n` option, followed by the name(s) of one or more nodes. When listing multiple node names, separate the names with commas without spaces.

The following table shows the node attributes that you can display via `mpinfo`. The heading that is displayed for each attribute is shown in parentheses at the end of each description.

Note these points:

- You can specify one or more of these attributes via the `--A` option, or as part of an RRS as an argument to the `--R` option. You can use either the attribute's real name or, in some cases, a shorter version.
- You can list the settings of all attributes (including any system administrator–defined attributes) on a per-node basis via the `--v` option.

- You can list the names and brief descriptions of these attributes via the `--ln` option.

TABLE 4–3 Node attributes available via `mpinfo`

Attribute	Short Form	Description (`mpinfo` output heading)
`cpu_idle`	`idle`	Percent of time CPU is idle (*IDLE*).
`cpu_iowait`	`iowait`	Percent of time CPU spends waiting for I/O (*IWAIT*).
`cpu_kernel`	`kernel`	Percent of time CPU spends in kernel (*KERNL*).
`cpu_swap`	`swap`	Percent of time CPU spends waiting for swap (*SWAP*).
`cpu_type`	`cpu`	CPU architecture (*CPU*).
`cpu_user`	`user`	Percent of time CPU spends running user's program (*USER*).
`domain`		DNS domain.
`enabled`		If set, node is available for spawning jobs on it.
`load1`		Load average for the past minute (*LOAD1*).
`load5`		Load average for the past five minutes (*LOAD5*).
`load15`		Load average for the past 15 minutes (*LOAD15*).
`manufacturer`	`manuf`	Hardware manufacturer (*MANUFACTURER*).
`mem_free`	`memf`	Node's available RAM (in Mbytes) (*FMEM*).
`mem_total`	`memr`	Node's total physical memory (in Mbytes) (*MEM*).
`name`		Name of the node (*NAME*).

TABLE 4–3 Node attributes available via `mpinfo` *(continued)*

Attribute	Short Form	Description (`mpinfo` output heading)
`ncpus`	`ncpu`	Number of CPU modules in the node (*NCPU*).
`os_arch_kernel`	`mach`	Node's kernel architecture (*MACH*).
`os_max_proc`	`maxproc`	Maximum number of processes allowed on the node (note that this is *all* processes, including cluster daemons) (*MPROC*).
`os_name`	`os`	Name of the operating system running on the node (*OS*).
`os_release`	`osrel`	Operating system's release number (*OSREL*).
`os_release_maj`	`osmaj`	The major number of the operating system release number (*MAJ*).
`os_release_min`	`osmin`	The minor number of the operating system release number (*MIN*).
`os_version`	`osver`	Operating system's version (*OSVER*).
`partition`		The partition of which the node is a member (*PARTITION*).
`serial_number`	`serno`	Hardware serial number (*SERIAL*).
`swap_free`	`swapf`	Node's available swap space (in Mbytes) (*FSWP*).
`swap_total`	`swapr`	Node's total swap space (in Mbytes) (*SWAP*).

The following is an example of the `mpinfo` output for nodes:

```
% mpinfo --N
node0 87 =>mpinfo --N
NAME  UP PARTITION OS    OSREL NCPU FMEM   FSWP    LOAD1 LOAD5 LOAD15
node0 y  p0        SunOS 5.6   1    0.89   158.34  0.09  0.11  0.13
node1 y  p0        SunOS 5.6   1    31.41  276.12  0.00  0.01  0.01
node2 y  p1        SunOS 5.6   1    25.59  279.77  0.00  0.00  0.01
node3 y  p1        SunOS 5.6   1    25.40  279.88  0.00  0.00  0.01
```

The following example shows only the names of nodes and the partition they're in:

```
% mpinfo --N --A name,partition
NAME         PARTITION
node0        part0
node1        part0
node2        part1
node3        part1
```

Cluster

Use the `--C` option to display information about the entire cluster. For example,

```
% mpinfo --C
NAME   ADMINISTRATOR    DEF_INTER_PART
node0  wmitty           part0
```

where:

- `NAME` – The name of the cluster
- `ADMINISTRATOR` – The name of its administrator
- `DEF_INTER_PART` – The default interactive partition

CHAPTER 5

Debugging Programs

Prism is a component in the Sun HPC ClusterTools suite of software. You can use it to debug and visualize data in serial or message-passing programs on a Sun HPC cluster. For complete information on Prism, see the *Prism 6.0 User's Guide* and *Prism 6.0 Reference Manual*. This chapter gives a brief overview of how to start up Prism.

To use Prism, you must first log in to the Sun HPC cluster, as described in Chapter 2. If you are using the graphical version of Prism, you must be running Solaris 2.6 or Solaris 7 with either OpenWindows or CDE.

You can start Prism by entering

```
% prism
```

and then loading your executable program from within Prism.

Alternatively, you can specify the program's name on Prism's command line. In this case, Prism will start up with the program already loaded. For example,

```
% prism a.out
```

Once the program is loaded in Prism, you can execute it, debug it, and visualize data in it. The program executes on the same node as Prism.

Note - Prism does not debug programs at the thread level.

Debugging Sun MPI Programs

If you are going to use Prism to debug a Sun MPI program, use the `--np` option with `mprun` to specify how many processes are to be started. For example,

```
% prism --np 4 a.out
```

When you use the `--np` option, you can also use other Prism options, such as `--p`, to determine where the job's processes are to run and how they are mapped onto nodes. For example,

```
% prism --p part0 --np 4 a.out
```

starts Prism as well as the message-passing program `a.out` on the partition `part0`. Client Prism processes are also started with each of the `a.out` processes. They receive instructions from and return information to the master Prism daemon that is started by `mprun`.

You can attach to a running Sun MPI program by specifying its job ID after the name of the executable program. For example,

```
% prism --np 1 a.out 462
```

You can find out the job ID of a program by issuing the `mpps` command or by using the `--J` option to `mprun`.

The setting of the `MPRUN_FLAGS` environment variable applies to both `mprun` starting Prism and to Prism starting the parallel processes. This means that the default options are likely to be incorrect for one or the other, since you would typically want to start Prism on one node in a shared partition, and the Sun MPI processes on multiple nodes, possibly in a dedicated partition.

CHAPTER 6

Performance Tuning

Sun MPI uses a variety of techniques to deliver high-performance, robust, and memory-efficient message passing under a wide set of circumstances. In certain situations, however, applications will benefit from nondefault behaviors. The Sun MPI environment variables discussed in this section allow you to tune these default behaviors. A list of all Sun MPI environment variables, with brief descriptions, can be found in Appendix A and in the `MPI` man page.

Current Settings

User tuning of MPI environment variables can be restricted by the system administrator through a configuration file, `hpc.conf`. To determine whether such resctrictions are in place on your local cluster use the `MPI_PRINTENV` (described below) to verify settings. .

In most cases, performance will be good without tuning any environment variables. Nevertheless, here are some performance guidelines for using MPI environment variables. In some cases, diagnosis of whether environment variables would be helpful is aided by Prism profiling with TNF probes, as described in the *Prism User's Guide* and the *Sun MPI Programming and Reference Guide.*

Runtime Diagnostic Information

Certain Sun MPI environment variables cause extra diagnostic information to be printed out at run time:

```
% setenv MPI_PRINTENV 1
% setenv MPI_SHOW_INTERFACES 3
% setenv MPI_SHOW_ERRORS 1
```

Running on a Dedicated System

If your system has sufficient capacity for running your MPI job, you can commit processors aggressively to your job. At a minimum, the CPU load should not exceed the number of physical processors. The CPU load for your job is the number of MPI processes in the job, but the load is greater if your job is multithreaded. The load on the system must also be shared with any other jobs are running on the same system. You can check the current load can be checked with the `mpinfo` command.

To run your job more aggressively on a dedicated system, set the `MPI_SPIN` and `MPI_PROCBIND` environment variables:

```
% setenv MPI_SPIN 1
```

Use this only if you will leave at least one processor per node free to service system daemons. Profiling with Prism introduces background daemons that cause a slight but noticeable load, so you must be careful to avoid overloading when attempting to profile a code with this setting.

```
% setenv MPI_PROCBIND 1
```

Set the `MPI_PROCBIND` variable only if there are no other MPI jobs running and your job is single-threaded.

Safe Use of System Buffers

In some MPI programs, processes send large volumes of data with blocking sends before starting to receive messages. The MPI standard specifies that users must explicitly provide buffering in such cases, perhaps using `MPI_Bsend` calls. In practice, however, some users rely on the standard send routine (`MPI_Send`) to supply unlimited buffering. By default, Sun MPI prevents deadlock in such

situations through general polling, which drains system buffers even when no receives have been posted by the user code.

For best performance on typical, safe programs, you can suppress general polling should by setting `MPI_POLLALL`:

```
% setenv MPI_POLLALL 0
```

Trading Memory for Performance

Depending on message traffic, performance can stall if system buffers become congested, but it can be superior if buffers are large. Here, we examine performance for on-node messages via shared-memory buffers.

It is helpful to think of data traffic per connection, the “path” from a particular sender to a particular receiver, since many Sun MPI buffering resources are allocated on a per-connection basis. A sender may emit bursts of messages on a connection, during which time the corresponding receiver may not be depleting the buffers. For example, a sender may execute a sequence of send operations to one receiver during a period in which that receiver is not making any MPI calls whatsoever.

You may need to use profiling to diagnose such conditions. For more information on profiling, see the *Prism User’s Guide* and the *Sun MPI Programming and Reference Guide*.

Rendezvous or Eager Protocol?

Is your program sending many long, unexpected messages? Sun MPI offers message *rendezvous*, which requires a receiver to echo a ready signal to the sender before data transmission can begin. This can improve performance for the case of a pair of processes that communicate with a different order for their sends as for their receives, since receive-side buffering would be reduced. To allow rendezvous behavior for long messages, set the `MPI_EAGERONLY` environment variable:

```
% setenv MPI_EAGERONLY 0
```

The threshold message size for rendezvous behavior can be tuned independently for each protocol with `MPI_SHM_RENDVSIZE`, `MPI_TCP_RENDVSIZE`, and `MPI_RSM_RENDVSIZE`.

Note - Rendezvous will often degrade performance by coupling senders to receivers. Also, for some “unsafe” codes, it can produce deadlock.

Many Broadcasts or Reductions

Does your program include many broadcasts or reductions on large messages? Large broadcasts may benefit from increased values of `MPI_SHM_BCASTSIZE`, and large reductions from increased `MPI_SHM_REDUCESIZE`. Also, if many different communicators are involved, you may want to increase `MPI_SHM_GBPOOLSIZE`. In most cases, the default values will provide best performance.

Shared-Memory Point-to-Point Message Passing

The size of each shared-memory buffer is fixed at 1 Kbyte. Most other quantities in shared-memory message passing are settable with MPI environment variables.

A *short* message, at most `MPI_SHM_SHORTMSGSIZE` bytes long, is fit into one postbox and no buffers are used. Above that size, message data is written into buffers and controlled by postboxes.

Only starting at `MPI_SHM_PIPESTART` bytes, however, are multiple postboxes used, which is known as *pipelining*. The amount of buffer data controlled by any one postbox is at most `MPI_SHM_PIPESIZE` bytes. By default, `MPI_SHM_PIPESTART` is well below `MPI_SHM_PIPESIZE`. For the smallest pipelined messages, then, a message is broken roughly into two, and each of two postboxes controls roughly half the message.

Above `MPI_SHM_CYCLESTART` bytes, messages are fed cyclically through two sets of buffers, each set of size `MPI_SHM_CYCLESIZE` bytes. During a cyclic transfer, the footprint of the message in shared memory buffers is 2*`MPI_SHM_CYCLESIZE` bytes.

The postbox area consists of `MPI_SHM_NUMPOSTBOX` postboxes per connection. By default, each connection has its own pool of buffers, each pool of size `MPI_SHM_CPOOLSIZE` bytes.

By setting `MPI_SHM_SBPOOLSIZE`, users may specify that each sender has a pool of buffers, of `MPI_SHM_SBPOOLSIZE` bytes each, to be shared among its various connections. If `MPI_SHM_CPOOLSIZE` is also set, then any one connection may consume only that many bytes from its send-buffer pool at any one time.

Memory Considerations

In all, the size of the shared-memory area devoted to point-to-point messages is

```
n * ( n -- 1
)
  * (
      MPI_SHM_NUMPOSTBOX *
( 64 + MPI_SHM_SHORTMSGSIZE )
    + MPI_SHM_CPOOLSIZE
    )
```

bytes when per-connection pools are used (that is, when `MPI_SHM_SBPOOLSIZE` is not set) and

```
n * ( n -- 1
) * MPI_SHM_NUMPOSTBOX *
( 64 + MPI_SHM_SHORTMSGSIZE )
+
n * MPI_SHM_SBPOOLSIZE
```

bytes when per-sender pools are used (that is, when `MPI_SHM_SBPOOLSIZE` is set).

Cyclic message passing limits the size of shared memory that is needed to transfer even arbitrarily large messages.

Shared-Memory Collectives

Collective operations in Sun MPI are highly optimized and make use of a “general buffer pool” within shared memory.

`MPI_SHM_GBPOOLSIZE` sets the amount of space available on a node for the “optimized” collectives in bytes. By default, it is set to 20971520 bytes. This space is used by `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Reduce_scatter`, and `MPI_Barrier`, provided that two or more of the MPI processes are on the node.

When a communicator is created, space is reserved in the general buffer pool for performing barriers, short broadcasts, and a few other purposes.

For larger broadcasts, shared memory is allocated out of the general buffer pool. The maximum buffer-memory footprint in bytes of a broadcast operation is set by an environment variable as

(*n* / 4) * 2 * `MPI_SHM_BCASTSIZE`

where *n* is the number of MPI processes on the node. If less memory is needed than this, then less memory is used. After the broadcast operation, the memory is returned to the general buffer pool.

For reduce operations,

n * *n* * MPI_SHM_REDUCESIZE

bytes are borrowed from the general buffer pool.

The broadcast and reduce operations are pipelined for very large messages. By increasing MPI_SHM_BCASTSIZE and MPI_SHM_REDUCESIZE, one can improve the efficiency of these collective operations for very large messages, but the amount of time it takes to fill the pipeline can also increase.

If MPI_SHM_GBPOOLSIZE proves to be too small and a collective operation happens to be unable to borrow memory from this pool, the operation will revert to slower algorithms. Hence, under certain circumstances, performance could dictate increasing MPI_SHM_GBPOOLSIZE.

Running over TCP

TCP ensures reliable dataflow, even over lossy networks, by retransmitting data as necessary. When the underlying network loses a lot of data, the rate of retransmission can be very high and delivered MPI performance will suffer accordingly. Increasing synchronization between senders and receivers by lowering the TCP rendezvous threshold with MPI_TCP_RENDVSIZE may help in certain cases. Generally, increased synchronization will hurt performance, but over a lossy network it may help mitigate catastrophic degradation.

If the network is not lossy, then lowering the rendezvous threshold would be counterproductive and, indeed, a Sun MPI safeguard may be lifted. For reliable networks, use

```
% setenv MPI_TCPSAFEGATHER 0
```

Remote Shared Memory (RSM) Point-to-Point Message Passing

The RSM protocol has some similarities with the shared memory protocol, but it also has substantial deviations, and environment variables are used differently.

The maximum size of a short message is MPI_RSM_SHORTMSGSIZE bytes, with default value of 401 bytes. Short RSM messages can span multiple postboxes, but they still do not use any buffers.

The most data that will be sent under any one postbox for pipelined messages is MPI_RSM_PIPESIZE bytes. There are MPI_RSM_NUMPOSTBOX postboxes for each RSM connection.

If MPI_RSM_SBPOOLSIZE is unset, then each RSM connection has a buffer pool of MPI_RSM_CPOOLSIZE bytes. If MPI_RSM_SBPOOLSIZE is set, then each process has a pool of buffers that is MPI_RSM_SBPOOLSIZE bytes per remote node for sending messages to processes on the remote node.

Unlike the case of the shared-memory protocol, values of the MPI_RSM_PIPESIZE, MPI_RSM_CPOOLSIZE, and MPI_RSM_SBPOOLSIZE environment variables are merely requests. Values set with the setenv or printed when MPI_PRINTENV is used may not reflect effective values. In particular, only when connections are actually established are the RSM parameters truly set. Indeed, the effective values could change over the course of program execution if lazy connections are employed.

Striping refers to passing messages over multiple links to get the speedup of their aggregate bandwidth. The number of stripes used is MPI_RSM_MAXSTRIPE or all physically available stripes, whichever is less.

Use of rendezvous for RSM messages is controlled with MPI_RSM_RENDVSIZE.

Memory Considerations

Memory is allocated on a node for each remote MPI process that sends messages to it over RSM. If np_local is the number of processes on a particular node, then the memory requirement on the node for RSM message passing from any one remote process is

np_local * (MPI_RSM_NUMPOSTBOX * 128 + MPI_RSM_CPOOLSIZE)

bytes when MPI_RSM_SBPOOLSIZE is unset, and

np_local * MPI_RSM_NUMPOSTBOX * 128 + MPI_RSM_SBPOOLSIZE

bytes when MPI_RSM_SBPOOLSIZE is set.

The amount of memory actually allocated may be higher or lower than this requirement:

- The memory requirement is rounded up to some multiple of 8192 bytes with a minimum of 32768 bytes.
- This memory is allocated from a 256-Kbyte (262,144-byte) segment.
  - If the memory requirement is greater than 256 Kbytes, then insufficient memory will be allocated.
  - If the memory requirement is less than 256 Kbytes, some allocated memory will go unused. (There is some, but only limited, sharing of segments.)

If less memory is allocated than is required, then requested values of `MPI_RSM_CPOOLSIZE` or `MPI_RSM_SBPOOLSIZE` may be reduced at run time. This can cause the requested value of `MPI_RSM_PIPESIZE` to be overridden as well.

Each remote MPI process requires its own allocation on the node as described above.

If multiple stripes are employed, the memory requirement increases correspondingly.

Performance Considerations

The pipe size should be at most half as big as the connection pool

2 * `MPI_RSM_PIPESIZE` <= `MPI_RSM_CPOOLSIZE`

Otherwise, pipelined transfers will proceed slowly. The library adjusts `MPI_RSM_PIPESIZE` appropriately.

Reducing striping has no performance advantage, but varying `MPI_RSM_MAXSTRIPE` can give you insight into the relationship between application performance depends and internode bandwidth.

For pipelined messages, a sender must synchronize with its receiver to ensure that remote writes to buffers have completed before postboxes are written. Long pipelined messages can absorb this synchronization cost, but performance for short pipelined messages will suffer. In some cases, raising `MPI_RSM_SHORTMSGSIZE` can mitigate this effect.

APPENDIX **A**

Environment Variables

Many environment variables are available for fine-tuning your Sun MPI environment. All 39 Sun MPI environment variables are listed here with brief descriptions. The same descriptions are also available on the `MPI` man page. If you want to return to the default setting after having set a variable, simply unset it (using `unsetenv`). The effects of some of the variables are explained in more detail in Chapter 6.

The environment variables are listed here in six groups:

Informational

MPI_PRINTENV

When set to 1, causes the environment variables and `hpc.conf` parameters associated with the MPI job to be printed out. The default is 0.

MPI_QUIET

If set to 1, suppresses Sun MPI warning messages. The default value is 0.

MPI_SHOW_ERRORS

If set to 1, the MPI_ERRORS_RETURN error handler prints the error message and returns the error. The default value is 0.

MPI_SHOW_INTERFACES

When set to 1, 2 or 3, information regarding which interfaces are being used by an MPI application prints to stdout. Set MPI_SHOW_INTERFACES to 1 to print the selected internode interface. Set it to 2 to print all the interfaces and their rankings. Set it to 3 for verbose output. The default value, 0, does not print information to stdout.

General Performance Tuning

MPI_POLLALL

When set to 1, the default value, all connections are polled for receives, also known as full polling. When set to 0, only those connections are polled where receives are posted. Full polling helps drain system buffers and so lessen the chance of deadlock for “unsafe” codes. Well-written codes should set MPI_POLLALL to 0 for best performance.

MPI_PROCBIND

Binds each MPI process to its own processor. By default, MPI_PROCBIND is set to 0, which means processor binding is off. To turn processor binding on, set it to 1. The system administrator may allow or disable processor binding by setting the pbind parameter in the hpc.conf file on or off. If this parameter is set, the MPI_PROCBIND environment variable is disabled. Performance can be enhanced with processor binding, but very poor performance will result if processor binding is used for multithreaded jobs or for more than one job at a time.

MPI_SPIN

Sets the spin policy. The default value is 0, which causes MPI processes to spin nonaggressively, allowing best performance when the load is at least as great as the number of CPUs. A value of 1 causes MPI processes to spin aggressively, leading to

best performance if extra CPUs are available on each node to handle system daemons and other background activities.

Tuning Memory for Point-to-Point Performance

MPI_RSM_CPOOLSIZE

The requested size, in bytes, to be allocated per stripe for buffers for each remote-shared-memory connection. This value may be overridden when connections are established. The default value is 16384 bytes.

MPI_RSM_NUMPOSTBOX

The number of postboxes per stripe per remote-shared-memory connection. The default is 15 postboxes.

MPI_RSM_PIPESIZE

The limit on the size (in bytes) of a message that can be sent over remote shared memory via the buffer list of one postbox per stripe. The default is 8192 bytes.

MPI_RSM_SBPOOLSIZE

If set, `MPI_RSM_SBPOOLSIZE` is the requested size in bytes of each RSM send buffer pool. An RSM send buffer pool is the pool of buffers on a node that a remote process would use to send to processes on the node. A multiple of 1024 must be used. If unset, then pools of buffers are dedicated to connections rather than to senders.

MPI_RSM_SHORTMSGSIZE

The maximum size, in bytes, of a message that will be sent via remote shared memory without using buffers. The default value is 401 bytes.

MPI_SHM_CPOOLSIZE

The amount of memory, in bytes, that can be allocated to each connection pool. When `MPI_SHM_SBPOOLSIZE` is not set, the default value is 24576 bytes. Otherwise, the default value is `MPI_SHM_SBPOOLSIZE`.

MPI_SHM_CYCLESIZE

The limit, in bytes, on the portion of a shared-memory message that will be sent via the buffer list of a single postbox during a cyclic transfer. The default value is 8192 bytes. A multiple of 1024 that is at most `MPI_SHM_CPOOLSIZE`/2 must be used.

MPI_SHM_CYCLESTART

Shared-memory transfers that are larger than `MPI_SHM_CYCLESTART` bytes will be cyclic. The default value is 24576 bytes.

MPI_SHM_NUMPOSTBOX

The number of postboxes dedicated to each shared-memory connection. The default value is 16.

MPI_SHM_PIPESIZE

The limit, in bytes, on the portion of a shared-memory message that will be sent via the buffer list of a single postbox during a pipeline transfer. The default value is 8192 bytes. The value must be a multiple of 1024.

MPI_SHM_PIPESTART

The size, in bytes, at which shared-memory transfers will start to be pipelined. The default value is 2048. Multiples of 1024 must be used.

MPI_SHM_SBPOOLSIZE

If set, `MPI_SHM_SBPOOLSIZE` is the size, in bytes, of the pool of shared-memory buffers dedicated to each sender. A multiple of 1024 must be used. If unset, then pools of shared-memory buffers are dedicated to connections rather than to senders.

MPI_SHM_SHORTMSGSIZE

The size (in bytes) of the section of a postbox that contains either data or a buffer list. The default value is 256 bytes.

Note - If MPI_SHM_PIPESTART, MPI_SHM_PIPESIZE, or MPI_SHM_CYCLESIZE is increased to a size larger than 31744 bytes, then MPI_SHM_SHORTMSGSIZE may also have to be increased. See Chapter 6 for more information.

Numerics

MPI_CANONREDUCE

Prevents reduction operations from using any optimizations that take advantage of the physical location of processors. This may provide more consistent results in the case of floating-point addition, for example. However, the operation may take longer to complete. The default value is 0, meaning optimizations are allowed. To prevent optimizations, set the value to 1.

Tuning Rendezvous

MPI_EAGERONLY

When set to 1, the default, only the eager protocol is used. When set to 0, both eager and rendez-vous protocols are used.

MPI_RSM_RENDVSIZE

Messages communicated by remote shared memory that are greater than this size will use the rendezvous protocol unless the environment variable MPI_EAGERONLY is set to 1. Default value is 16384 bytes.

MPI_SHM_RENDVSIZE

Messages communicated by shared memory that are greater than this size will use the rendezvous protocol unless the environment variable `MPI_EAGERONLY` is set. The default value is 24576 bytes.

MPI_TCP_RENDVSIZE

Messages communicated by TCP that contain data of this size and greater will use the rendezvous protocol unless the environment variable `MPI_EAGERONLY` is set. Default value is 49152 bytes.

Miscellaneous

MPI_COSCHED

Specifies the user's preference regarding use of the `spind` daemon for coscheduling. Values can be 0 (prefer no use) or 1 (prefer use). This preference may be overridden by the system administrator's policy. This policy is set in the `hpc.conf` file and can be 0 (forbid use), 1 (require use), or 2 (no policy). If no policy is set and no user preference is specified, coscheduling is not used.

Note - If no user preference is specified, the value 2 will be shown when environment variables are printed with `MPI_PRINTENV`.

MPI_FLOWCONTROL

Limits the number of unexpected messages that can be queued from a particular connection. Once this quantity of unexpected messages has been received, polling the connection for incoming messages stops. The default value, 0, indicates that no limit is set. To limit flow, set the value to some integer greater than zero.

MPI_FULLCONNINIT

Ensures that all connections are established during initialization. By default, connections are established lazily. However, you can override this default by setting the environment variable `MPI_FULLCONNINIT` to 1, forcing full-connection initialization mode. The default value is 0.

MPI_MAXFHANDLES

The maximum number of Fortran handles for objects other than requests. `MPI_MAXFHANDLES` specifies the upper limit on the number of concurrently allocated Fortran handles for MPI objects other than requests. This variable is ignored in the default 32-bit library. The default value is 1024. Users should take care to free MPI objects that are no longer in use. There is no limit on handle allocation for C codes.

MPI_MAXREQHANDLES

The maximum number of Fortran request handles. `MPI_MAXREQHANDLES` specifies the upper limit on the number of concurrently allocated MPI request handles. Users must take care to free up request handles by properly completing requests. The default value is 1024. This variable is ignored in the default 32-bit library.

MPI_OPTCOLL

The MPI collectives are implemented using a variety of optimizations. Some of these optimizations can inhibit performance of point-to-point messages for “unsafe” programs. By default, this variable is 1, and optimized collectives are used. The optimizations can be turned off by setting the value to 0.

MPI_RSM_MAXSTRIPE

Defines the maximum number of stripes that can be used during communication via remote shared memory. The default value is the number of stripes in the cluster, with a maximum default of 2.

MPI_SHM_BCASTSIZE

On SMPs, the implementation of `MPI_Bcast()` for large messages is done using a double-buffering scheme. The size of each buffer (in bytes) is settable by using this environment variable. The default value is 32768 bytes.

MPI_SHM_GBPOOLSIZE

The amount of memory available, in bytes, to the general buffer pool for use by collective operations. The default value is 20971520 bytes.

MPI_SHM_REDUCESIZE

On SMPs, calling `MPI_Reduce()` causes all processors to participate in the reduce. Each processor will work on a piece of data equal to the `MPI_SHM_REDUCESIZE` setting. The default value is 256 bytes. Care must be taken when setting this variable because the system reserves `MPI_SHM_REDUCESIZE` * *np* * *np* memory to execute the reduce.

MPI_SPINDTIMEOUT

When coscheduling is enabled, limits the length of time (in milliseconds) a message will remain in the poll waiting for the `spind` daemon to return. If the timeout occurs before the daemon finds any messages, the process re-enters the polling loop. The default value is 1000 ms. A default can also be set by a system administrator in the `hpc.conf` file.

MPI_TCP_CONNLOOP

Sets the number of times `MPI_TCP_CONNTIMEOUT` occurs before signaling an error. The default value for this variable is 0, meaning that the program will abort on the first occurrence of `MPI_TCP_CONNTIMEOUT`.

MPI_TCP_CONNTIMEOUT

Sets the timeout value in seconds that is used for an `accept()` call. The default value for this variable is 600 seconds (10 minutes). This timeout can be triggered in both full- and lazy-connection initialization. After the timeout is reached, a warning message will be printed. If `MPI_TCP_CONNLOOP` is set to 0, then the first timeout will cause the program to abort.

MPI_TCP_SAFEGATHER

Allows use of a congestion-avoidance algorithm for `MPI_Gather()` and `MPI_Gatherv()` over TCP. By default, `MPI_TCP_SAFEGATHER` is set to 1, which means use of this algorithm is on. If you know that your underlying network can handle gathering large amounts of data on a single node, you may want to override this algorithm by setting `MPI_TCP_SAFEGATHER` to 0.

APPENDIX **B**

Troubleshooting

This appendix describes some common problem situations, resulting error messages, and suggestions for fixing the problems. Sun MPI error reporting, including I/O, follows the MPI-2 standard. By default, errors are reported in the form of standard error classes. These classes and their meanings are listed in Table B–1 (for non-I/O MPI) and Table B–2 (for MPI I/O) and are also available on the `MPI` man page.

Three predefined error handlers are available in Sun MPI 4.0:

- `MPI_ERRORS_RETURN` – The default, returns an error code if an error occurs.
- `MPI_ERRORS_ARE_FATAL` – I/O errors are fatal, and no error code is returned.
- `MPI_THROW_EXCEPTION` – A special error handler to be used only with C++.

MPI Messages

You can make changes to and get information about the error handler using any of the following routines:

- `MPI_Comm_create_errhandler`
- `MPI_Comm_get_errhandler`
- `MPI_Comm_set_errhandler`

Messages resulting from an MPI program fall into two categories:

- *Error messages* – Error messages stem from within MPI. Usually an error message explains why your program cannot complete, and the program aborts.
- *Warning messages* – Warnings stem from the environment in which you are running your MPI program and are usually sent by `MPI_Init`. They are not associated with an aborted program, that is, programs continue to run despite warning messages.

Error Messages

Sun MPI error messages use a standard format:

`[`*x y z*`] Error in` *function_name*`:` *errclass_string*`:`*intern*`(`*a*`):`*description*`:`*unixerrstring*

where

- `[`*x y z*`]` is the *process communication identifier*, and:
  - *x* is the job id (or jid).
  - *y* is the name of the communicator if a name exists; otherwise it is the address of the opaque object.
  - *z* is the rank of the process.

    The process communication identifier is present in every error message.
- *function_name* is the name of the associated MPI function. It is present in every error message.
- *errclass_string* is the string associated with the MPI error class. It is present in every error message.
- *intern* is an internal function. It is optional.
- *a* is a system call, if one is the cause of the error. It is optional.
- *description* is a description of the error. It is optional.
- *unixerrstring* is the UNIX error string that describes system call *a*. It is optional.

Warning Messages

Sun MPI warning messages also use a standard format:

`[`*x y z*`] Warning` *message*

where

- *message* is a description of the error.

Standard Error Classes

Listed below are the error return classes you may encounter in your MPI programs. Error values may also be found in `mpi.h` (for C), `mpif.h` (for Fortran), and `mpi++.h` (for C^{++}).

TABLE B–1 Sun MPI Standard Error Classes

Error Code	Value	Meaning
MPI_SUCCESS	0	Successful return code.
MPI_ERR_BUFFER	1	Invalid buffer pointer.
MPI_ERR_COUNT	2	Invalid count argument.
MPI_ERR_TYPE	3	Invalid datatype argument.
MPI_ERR_TAG	4	Invalid tag argument.
MPI_ERR_COMM	5	Invalid communicator.
MPI_ERR_RANK	6	Invalid rank.
MPI_ERR_ROOT	7	Invalid root.
MPI_ERR_GROUP	8	Null group passed to function.
MPI_ERR_OP	9	Invalid operation.
MPI_ERR_TOPOLOGY	10	Invalid topology.
MPI_ERR_DIMS	11	Illegal dimension argument.
MPI_ERR_ARG	12	Invalid argument.
MPI_ERR_UNKNOWN	13	Unknown error.
MPI_ERR_TRUNCATE	14	Message truncated on receive.
MPI_ERR_OTHER	15	Other error; use `Error_string`.
MPI_ERR_INTERN	16	Internal error code.
MPI_ERR_IN_STATUS	17	Look in status for error value.
MPI_ERR_PENDING	18	Pending request.

TABLE B–1 Sun MPI Standard Error Classes *(continued)*

Error Code	Value	Meaning
MPI_ERR_REQUEST	19	Illegal MPI_Request handle.
MPI_ERR_KEYVAL	36	Illegal key value.
MPI_ERR_INFO	37	Invalid info object.
MPI_ERR_INFO_KEY	38	Illegal info key.
MPI_ERR_INFO_NOKEY	39	No such key.
MPI_ERR_INFO_VALUE	40	Illegal info value.
MPI_ERR_TIMEDOUT	41	Timed out.
MPI_ERR_RESOURCES	42	Out of resources.
MPI_ERR_TRANSPORT	43	Transport layer error.
MPI_ERR_HANDSHAKE	44	Error accepting/connecting.
MPI_ERR_SPAWN	45	Error spawning.
MPI_ERR_LASTCODE	46	Last error code.

MPI I/O message are listed separately, in Table B–2.

MPI I/O Error Handling

Sun MPI I/O error reporting follows the MPI-2 standard. By default, errors are reported in the form of standard error codes (found in `/opt/SUNWhpc/include/mpi.h`). Error classes and their meanings are listed in Table B–2. They can also be found in `mpif.h` (for Fortran) and `mpi++.h` (for C^{++}).

You can change the default error handler by specifying `MPI_FILE_NULL` as the file handle with the routine `MPI_File_set_errhandler`, even no file is currently open. Or, you can use the same routine to change a specific file's error handler.

TABLE B–2 Sun MPI I/O Error Classes

Error Class	Value	Meaning
`MPI_ERR_FILE`	20	Bad file handle.
`MPI_ERR_NOT_SAME`	21	Collective argument not identical on all processes.
`MPI_ERR_AMODE`	22	Unsupported `amode` passed to open.
MPI_ERR_UNSUPPORTED_DATAREP	23	Unsupported `datarep` passed to `MPI_File_set_view`.
`MPI_ERR_UNSUPPORTED_OPERATION`	24	Unsupported operation, such as seeking on a file that supports only sequential access.
`MPI_ERR_NO_SUCH_FILE`	25	File (or directory) does not exist.
`MPI_ERR_FILE_EXISTS`	26	File exists.
`MPI_ERR_BAD_FILE`	27	Invalid file name (for example, path name too long).
`MPI_ERR_ACCESS`	28	Permission denied.
`MPI_ERR_NO_SPACE`	29	Not enough space.
`MPI_ERR_QUOTA`	30	Quota exceeded.
`MPI_ERR_READ_ONLY`	31	Read-only file system.
`MPI_ERR_FILE_IN_USE`	32	File operation could not be completed, as the file is currently open by some process.
`MPI_ERR_DUP_DATAREP`	33	Conversion functions could not be registered because a data representation identifier that was already defined was passed to `MPI_REGISTER_DATAREP`.

TABLE B–2 Sun MPI I/O Error Classes *(continued)*

Error Class	Value	Meaning
MPI_ERR_CONVERSION	34	An error occurred in a user-supplied data-conversion function.
MPI_ERR_IO	35	I/O error.
MPI_ERR_INFO	37	Invalid info object.
MPI_ERR_INFO_KEY	38	Illegal info key.
MPI_ERR_INFO_NOKEY	39	No such key .
MPI_ERR_INFO_VALUE	40	Illegal info value.
MPI_ERR_LASTCODE	46	Last error code.