Sun HPC ClusterTools™ 6 Software User's Guide

Sun Microsystems, Inc.
www.sun.com

Part No. 819-4131-10
March 2006, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Please
Recycle

Contents

Tables

Figures

Preface

This manual explains how to use distributed resource management packages for effective resource management and utilization accounting. The following packages work in conjunction with Sun Message-Passing Interface (Sun MPI) parallel applications:

- Sun N1™ Grid Engine (N1GE) Version 6
- Load Sharing Facility (LSF) HPC Version 6.2 from Platform Computing Corporation
- OpenPBS Portable Batch System (PBS) 2.3.16 and Altair PBS Professional 7.1

Before You Read This Book

The *Sun HPC ClusterTools™ 6 Software Release Notes* includes release noteinformation for the other components in this suite. For information about writing MPI programs, refer to the *Sun MPI 7.0 Software Programming and Reference Guide.* For information about a specific distributed resource management package, refer to the documentation supplied with that package.

Using UNIX Commands

This document might not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at

  http://www.sun.com/documentation

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. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

* The settings on your browser might differ from these settings.

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

This book focuses on Sun MPI and assumes familiarity with the *MPI Standard*. The following materials provide useful background about using Sun MPI and about the *MPI Standard*.

Application	Title	Part Number
Sun HPC ClusterTools Documentation	*Read Me First: Guide to Sun HPC ClusterTools Software Documentation*	819-4136-10
Sun HPC ClusterTools Software	*Sun HPC ClusterTools 6 Software Release Notes*	819-4129-10
	Sun HPC ClusterTools 6 Software Installation Guide	819-4130-10
	Sun HPC ClusterTools 6 Software Performance Guide	819-4134-10
	Sun HPC ClusterTools 6 Software Administrator's Guide	819-4132-10
Sun MPI Programming	*Sun MPI 7.0 Software Programming and Reference Guide*	819-4133-10

For more information about Sun N1 Grid Engine software, see the Sun N1 Grid Engine web site at:

http://www.sun.com/software/gridware

If you are using the Load Sharing Facility (LSF) Suite from Platform Computing Corporation, consult the documentation available from their website:

http://www.platform.com

Altair PBS Professional documentation is available from:

http://www.altair.com

Documentation, Support, and Training

Sun Function	URL
Documentation	http://www.sun.com/documentation/
Support	http://www.sun.com/support/
Training	http://www.sun.com/training/

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

http://www.sun.com/hwdocs/feedback

Please include the title and part number of your document with your feedback:

Sun HPC ClusterTools 6 Software User's Guide, part number 819-4131-10

CHAPTER 1

Introduction to Sun HPC ClusterTools Software

Sun HPC ClusterTools™ 6 software is a set of parallel development tools that extend the Sun network computing solutions to high-end distributed-memory applications. This chapter summarizes its required configuration and principal components. It contains the following sections:

Supported Configurations

Sun HPC ClusterTools 6 software requires the Solaris™ 10 Operating System (Solaris 10 OS). All programs that execute under the Solaris 10 OS will execute in the Sun HPC ClusterTools environment.

Sun HPC ClusterTools 6 software supports Sun Studio 8, 9, 10, and 11 C, C++, and Fortran compilers.

Sun HPC ClusterTools 6 software can run MPI jobs of up to 2048 processes on as many as 256 nodes. It also provides load balancing and support for spawning MPI processes.

The Sun HPC ClusterTools software runs on clusters connected by any TCP/IP-capable interconnect, such as high-speed Ethernet, Gigabit Ethernet, and Infiniband.

Sun HPC Cluster Runtime Environment (CRE)

Sun HPC ClusterTools 6 software provides a command line interface (also called CRE for Cluster Runtime Environment) that starts jobs and provides status information. It performs four primary operations:

- Executes programs
- Kills programs
- Displays job information
- Displays node information

Each of these operations is summarized below. Subsequent chapters contain the procedures.

Executing Programs With `mprun`

Sun HPC ClusterTools 6 software can start both serial and parallel jobs. It is particularly useful for balancing computing load in serial jobs executed across shared partitions, where multiple processes can be competing for the same node resources. The syntax and use of `mprun` are described in Chapter 4.

Killing Programs

The runtime environment uses the `mpkill` command to kill jobs in progress and send signals to those jobs. Its syntax and use are described in Chapter 6.

Displaying Job Information

The runtime environment uses the `mpps` command to display information about jobs and their processes. Its syntax and use are described in Chapter 7.

Displaying Node Information

The runtime environment uses the `mpinfo` command to display information about nodes and their partitions. Its syntax and use are described in Chapter 10.

Integration With Distributed Resource Management Systems

Sun HPC ClusterTools 6 software provides new integration facilities with three select distributed resource management (DRM) systems. These systems provide proper resource allocation, parallel job control and monitoring, and proper job accounting. They are:

- Sun N1 Grid Engine (N1GE) Version 6
- Load Sharing Facility (LSF) HPC Version 6.2 from Platform Computing
- OpenPBS Portable Batch System (PBS) 2.3.16 and Altair PBS Professional 7.1

The support of other available DRM systems than those stated above are possible through the use of open APIs. Please contact your Sun representative for more information.

You can launch parallel jobs directly from these distributed resource management systems. The DRM interacts closely with Sun CRE for proper resource description and with the multiple processes comprising the requested parallel job.

For a description of the scalable and open architecture of the DRM integration facilities, see "How the CRE Environment Is Integrated With Distributed Resource Management Systems" on page 11. For instructions, see Chapter 5.

Sun MPI and MPI I/O

Sun MPI is a highly optimized version of the Message Passing Interface (MPI) communications library. It implements all of the *MPI 1.2 Standard* and the *MPI 2.0 Standard*. Its highlights are:

- Integration with the Sun HPC ClusterTools Runtime Environment (CRE)
- Support for multithreaded programming
- Seamless use of different network protocols; for example, code compiled on a Sun HPC cluster that has fast Ethernet network can be run without change on a cluster that has an ATM network
- Multiprotocol support so that MPI picks the fastest available medium for each type of connection (such as shared memory, fast Ethernet, or ATM)
- Communication via shared memory for fast performance on clusters of SMPs
- Optimized collectives for symmetric multiprocessors (SMPs) and clusters of SMPs

- Full F77, C, and C++ support, and basic F90 support

Debugging With TotalView

TotalView is a third-party multiprocess debugger from Etnus that runs on many platforms. Support for using the TotalView debugger on Sun MPI applications includes:

- Making Sun HPC ClusterTools software compatible with the Etnus debugger TotalView
- Allowing Sun MPI jobs to be debugged by TotalView using the Sun Grid Engine (SGE), the Portable Batch System (PBS), and Platform Computing's Load Sharing Facility (LSF)
- Displaying Sun MPI message queues
- Allowing multiple instantiations of TotalView on a single cluster
- Supporting TotalView in Sun HPC ClusterTools software

Refer to the TotalView documentation at `http://www.etnus.com` for more information about using TotalView.

MPProf

MPProf is a message-passing profiler intended for use with Sun MPI programs. It extracts information about calls to Sun MPI routines, storing the data in a set of intermediate files, one file per process. It then uses the intermediate data to generate a report profiling the program's message-passing activity.

MPProf's data-gathering operations are enabled by setting an environment variable before running the user program. If this environment variable is not set, program execution proceeds without generating profiling data. The MPProf report generator is invoked with the command-line utility, `mpprof`. The report is an ASCII text file that provides the following types of information:

- The percentage of total execution time spent in MPI calls across all processes
- The percentage of time each process spent in MPI calls
- The number of calls, time spent, and bytes sent or received per MPI routine, averaged over all processes, with percent variation among processes
- Connectivity statistics (message count and volume) between processor pairs
- The settings of MPI environment variables that have performance implications

MPProf also includes a data conversion utility, `mpdump`, which converts the intermediate data to user-readable ASCII files with the data in a raw (unanalyzed) state. You can then use the `mpdump` output files as input to a report generator, which you would supply in place of `mpprof`.

The MPProf tool is best suited for code analysis situations where message-passing behavior is of primary interest and where simplicity and ease-of-use are also important. For a comprehensive analysis of a complex MPI program, you would need to use MPProf in combination with other profiling tools. For example,

- Use a trace-history viewer to collect information about sequential relationships in message-passing events. This would complement the MPProf focus on aggregate statistics.
- Use the Sun Studio Developer Performance Analyzer for analyzing parts of the code other than the MPI routines.

CHAPTER 2

Fundamental Concepts

This chapter summarizes a few basic concepts that you should understand to get the most out of Sun's HPC ClusterTools software. It contains the following sections:

Clusters and Nodes

High performance computing clusters[1] are groups of Sun SMP servers interconnected by any Sun-supported, TCP/IP-capable interconnect. Each server in a cluster is called a *node*.

Note – A *cluster* can consist of a single SMP server. However, to execute MPI jobs on even a single-node cluster, Sun Cluster Runtime Environment (CRE) must be running on that cluster.

When using CRE, you can select the cluster and nodes on which your MPI programs will run and how your processes will be distributed among them. For instructions, see Chapter 4, "Running Programs With `mprun`."

1. SunCluster™ is a completely different technology used for high availability (HA) applications.

Partitions

You can group a cluster's nodes into *partitions*. Partitions let you run different jobs simultaneously on different subsets of the cluster. You can also use partitions to create groups of nodes with similar characteristics such as memory size, CPU count, or I/O support, so you can target jobs that benefit from those characteristics.

Note – Sun servers can be configured into "logical nodes," called *domains*. You can also group these domains into CRE partitions. For more information about domains, refer to the documentation that came with your Sun Fire hardware.

You can define multiple partitions within a cluster.

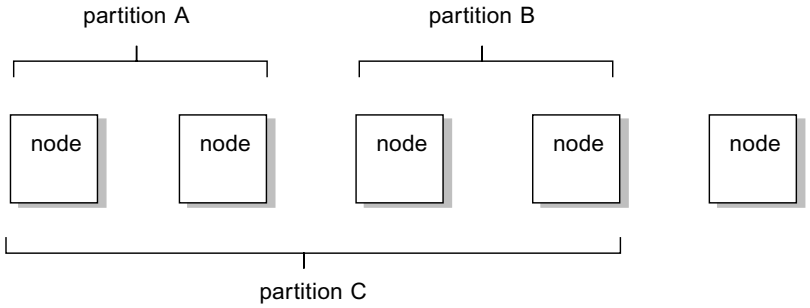


FIGURE 2-1 Defining Multiple Partitions Within a Cluster

Partitions do not have to include every node in the cluster. Nodes that are not included in any partition are called *independent* or *free-floating* nodes.

A single node can be included in more than one partition. However, two partitions with overlapping nodes cannot run jobs simultaneously. Only one of them can be enabled at a time. In the example above, partitions A and B can run jobs simultaneously with each other, but not with partition C.

How Partitions Are Enabled and Selected

A job can run only on a partition that has been *enabled*. Normally, the system administrator who manages the cluster enables and *disables* partitions (for more information, see the *Sun HPC ClusterTools Software Administrator's Guide*).

To find out which partitions are currently enabled, use the `-P` option to the `mpinfo` command, as described in "To Display Information About All Partitions (`-P`)" on page 106.

If only one partition is enabled, all jobs must run on that partition. If multiple partitions are enabled, where your particular job runs depends upon which environment variables the cluster administrator set and which options to the `mprun` command you entered. To determine the partition, CRE steps through the criteria shown in FIGURE 2-2, in order.

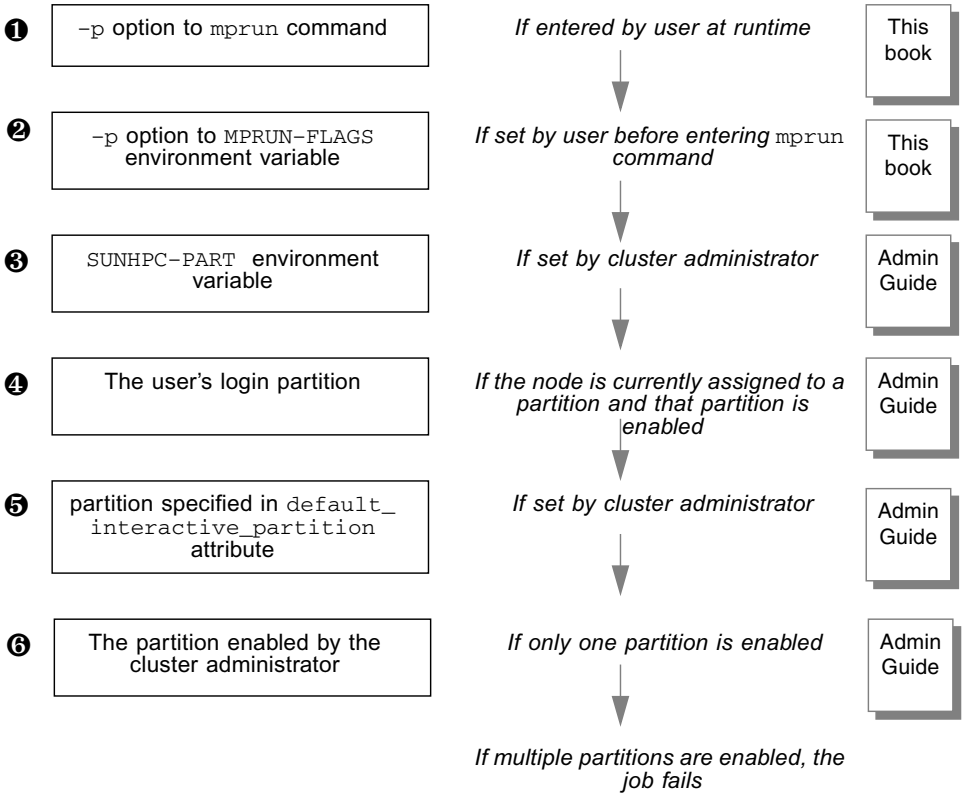


FIGURE 2-2 CRE Partition Selection Criteria

Load Balancing

CRE load-balances programs when more CPUs are available than are required for a job. When you issue the `mprun` command to start a job, 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, 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 was placed on a heavily loaded node. It also ensures that some nodes do not sit idle while other nodes are heavily loaded, which keeps the overall throughput of the partition as high as possible.

Processes

When a serial program executes on a Sun HPC cluster, it becomes a Solaris process with a Solaris *process ID*, or *PID*. When CRE executes a distributed message-passing program it spawns multiple Solaris processes, each with its own PID.

CRE allows you to control several aspects of jobs and process execution, such as:

- Number of processes per job
- Process spawning
- Mapping processes to nodes

For tasks and instructions, see Chapter 4.

Jobs

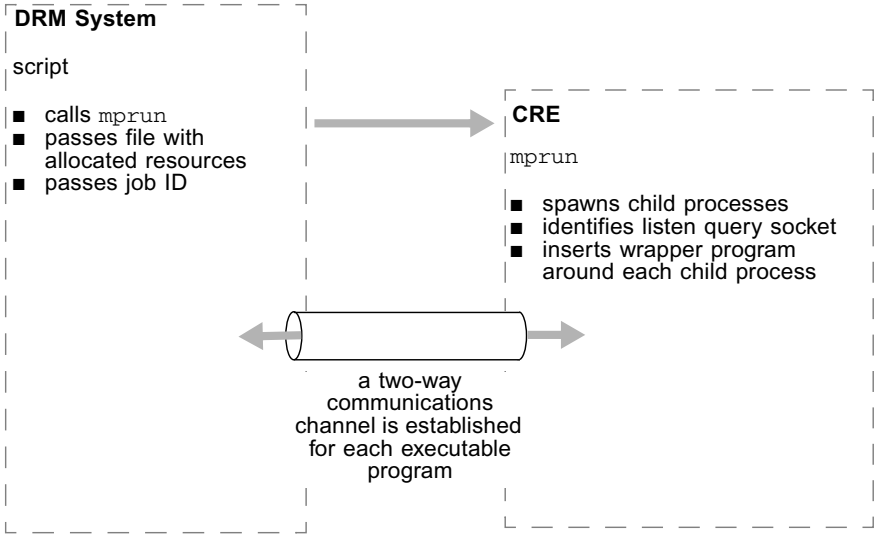
CRE assigns a *job ID*, or *jid*, to a program. In an MPI job, the jid applies to the overall job. Many CRE commands take jids as arguments. CRE provides a variety of information about jobs. To find out how to obtain that information, see Chapter 7.

How the CRE Environment Is Integrated With Distributed Resource Management Systems

As described in Chapter 1, the ClusterTools 6 environment provides close integration between CRE and three different DRM systems:

- SGE/SGEE/N1GE
- LSF
- PBS

The integration process is similar for all three, with some individual differences. The DRM system, whether SGE, LSF, or PBS, launches the job through a script. The script calls `mprun`, and passes it a host file of the resources that have been allocated for the job, plus the job ID assigned by the DRM system.



The CRE environment continues to perform most of its normal parallel-processing actions, but its child processes do not fork any executable programs. Instead, each child process identifies a communications channel (specifically, a listen query socket) through which it can be monitored by the CRE environment while running in the DRM system.

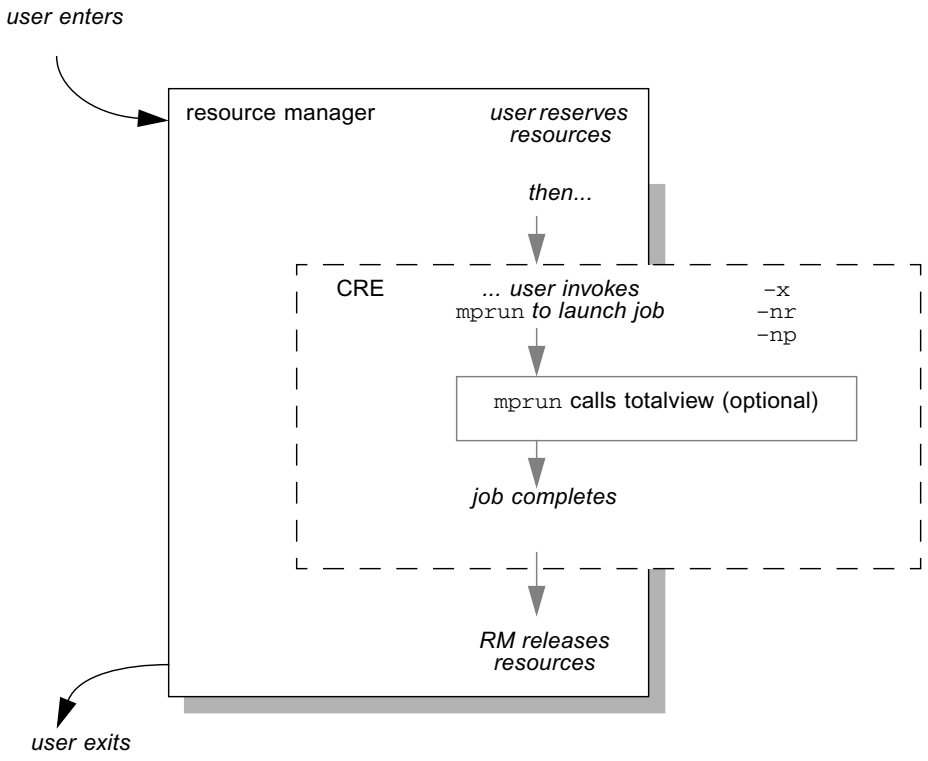
You can also invoke a similar process interactively, without a script. Instructions for script-based and interactive job launching are provided in Chapter 5.

How Programs Are Launched

The exact instructions vary from one resource manager to another, and are affected by CRE's configuration, but they all follow these general guidelines:

1. You can launch the job either interactively or through a script. Instructions for both are provided in Chapter 5 and the following man pages:
   - `lsf_cre(1)`
   - `pbs_cre(1)`
   - `sge_cre(1)`
2. Enter the DRM processing environment before launching jobs with `mprun`.
3. Reserve resources for the parallel job and set other job control parameters from within their resource manager.
4. Invoke the `mprun` command with the applicable resource manager flags. Those flags are described in Chapter 5 and the `mprun(1)` manpage.

Here is a diagram that summarizes the user interaction:



How Distributed Resource Managers Work

If you are using a Distributed Resource Manager (DRM) such as Sun Grid Engine, PBS, or LSF for resource management, all Sun HPC ClusterTools jobs are handled by the DRM's Batch system. Consequently, Sun HPC ClusterTools job submission involves the following:

- When a Sun HPC ClusterTools job is submitted, it is placed in a job queue, running interactively.
- These queues are created by the system administrator. Each queue is defined by a set of job-start criteria, called *job-scheduling policies.* These policies can be specified by the administrator, or default queue policies can be used.

- If a job has particular resource requirements and if a particular queue's job-scheduling policies meet those requirements, you can specify that the job be placed on that queue. If a job does not require special execution conditions, you can leave the choice of queue to the DRM's Batch system.
- The job waits in its queue until it reaches the head of the queue *and* the cluster is able to satisfy the job scheduling policies of that queue. At that point the job is started.

For further information about using DRMs with CRE, see the man pages `sge_cre.1`, `pbs_cre.1`, and `lsf_cre.1`.

How CRE Works With Zones in the Solaris 10 Operating System

The Solaris 10 Operating System (Solaris 10 OS) allows you to create secure, isolated areas within a single instance of the Solaris 10 OS. These areas, called *zones*, provide secure environments for running applications. Applications that execute in one zone cannot monitor or affect activity in another zone. You can create multiple non-global zones to run as virtual instances of the Solaris OS on the same hardware.

The *global zone* is the default zone for the Solaris system. You install Sun HPC ClusterTools software into the global zone. However, any non-global zones running under that Solaris system cannot "inherit" that installation, and installing HPC ClusterTools into an individual non-global zone is not supported. This means that you must install and configure HPC ClusterTools in the global zone only, though you may compile/run/debug in either a global or a non-global zone.

CHAPTER 3

Before You Begin

This chapter provides miscellaneous information about the runtime environment that you should know before you begin to use it. It contains the following sections:

Prerequisites

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

See the *Sun MPI Software Programming and Reference Guide* and the *Sun HPC ClusterTools Software Performance Guide* for information on linking in the Sun MPI libraries.

Command and Man Page Paths

CRE commands typically reside in the directory `/opt/SUNWhpc/bin`. If you are unable to execute them, you may need to add that directory to your path; check with your system administrator.

The man pages for Sun HPC ClusterTools commands reside in the `/opt/SUNWhpc/man` directory.

Authentication Methods

Sun HPC ClusterTools software supports two optional forms of user authentication that require the execution of user-level commands. The two methods are Kerberos Version 5 and DES. If one of these authentication methods is enforced on your Sun HPC cluster, use the commands listed in the following table.

TABLE 3-1 User Commands Required by Authentication Methods

Authentication Method	Required Command
Kerberos 5	While Kerberos Version 5 authentication is in use, you must issue a `kinit` command before running any CRE command.
DES	While DES authentication is in use, you must issue the `keylogin` command before issuing any CRE command.

See your system administrator for guidance.

Core Files

Core files are produced as they normally are in the Solaris environment. However, if more than one process dumps core in a multiprocess program, the resulting core file may be overwritten in the same directory. Use `coreadmin`(1M) to control the naming and placement of core files.

CHAPTER 4

Running Programs With `mprun`

The `mprun` command controls several aspects of program execution. This chapter describes what you can do with the command. It contains the following sections:

Syntax

```
% mprun [ options ] [ – ] program–name [ program–arguments ]
```

Options

The *options* control the behavior of the command. The tasks they perform are summarized in the diagram on the previous page. TABLE 4-5 lists the options in alphabetical order, with a brief description.

The runtime environment applies the options to the `mprun` command according to useful program logic rather than sequential order. Some options override conflicting options that appear earlier in the command line or in the `MPRUN_FLAGS` environment variable. In some cases, the presence of one option causes other options in the command line to be ignored, even if they appear later in the command line. As a result, option precedence varies by task. A table at the beginning of each group of tasks lists precedence order for the options used in those tasks.

Program–Name

If *program–name* conflicts with the name of an `mprun` *option*, use the **–** (dash) symbol to separate the program name from the option list. Be sure to add a space between the – symbol and the dash in the program name. For example:

```
% mprun -np 4 - myprogram
```

Program–Arguments

Enter any required *program–arguments* after the *program–name*.

Pre-Entering Command Options with `MPRUN-FLAGS`

You can pre-enter options to the `mprun` command by setting the `MPRUN-FLAGS` environment variable. Since the `MPRUN-FLAGS` variable only affects default behavior, you can override those options by entering different ones when you enter the `mprun` command itself.

The `MPRUN-FLAGS` environment variable uses the same *options* as the `mprun` command. (For a complete list, see TABLE 4-5.) If you use more than one word, enclose the list in quotation marks.

For example, to make `part2` the default partition, enter:

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`.

```
% printenv MPRUN_FLAGS
```

Environment Variables Available for Scripts

Three environment variables related to `mprun` are available for your scripts:

`MP_RANK`	The rank of a process in the job: 0 – 2047
`MP_NPROCS`	The number of processes in a job: 1 – 2048
`MP_JOBID`	The *jid* of the job

Each variable is automatically set by the `mprun` command at execution time. For example, this instance of `mprun`...

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

... would set the value of the variables to:

`MP_RANK`	0 through 5
`MP_NPROCS`	6
`MP_JOBID`	The same *jid* that can be displayed by `mpps` (see Chapter 7)

Controlling Where the Program Runs

To Perform This Task	Use This Option
How to run with default settings	
How to run on a different cluster	`-c`
How to run on a different partition	`-p`
How to run as multiple processes	`-np`
How to share nodes	`-j`
How to enable process spawning	`-Ys`
How to disable process spawning	`-Ns`
How to wrap multiple processes	`-W`
How to settle for available processes	`-S`
How to include independent nodes	`-u`
To Combine Process Placement Options	

Precedence for Program Execution

This Option...	Nullifies the Previous Instance of This Option ...
`-np`	`-np`
`-Ys`	`-Ns`
`-Ns`	`-Ys`
`-W`	`-S`
`-S`	`-W`
`-u`	`-u`
`-G`	`-G`
`-A`	`-A`
`-C`	`-C`

This Option...	Nullifies the Previous Instance of This Option ...
-c	-c
-p	-p
-r	-r

▼ To Run a Program With Default Settings

To run the program with default settings, enter the command and program name, followed by any required arguments to the program:

```
% mprun program–name
```

▼ To Run on a Different Cluster (-c)

By default, a program runs on your login cluster. To run a program on a different cluster, use the -c option:

```
% mprun -c cluster–name program–name
```

To find the name of a cluster, use the mpinfo command with the -C option, as described in "To Display Information About the Current Cluster (-C)" on page 105. Note case sensitivity.

▼ To Run on a Different Partition (-p)

To run the program on a partition other than your login partition, use the -p option:

```
% mprun -p partition–name program–name
```

The partition must be enabled. If it is not enabled, the job fails. (As described in "Partitions" on page 8, if a node is included in multiple partitions, only one partition can be enabled at a time.)

▼ To Run as Multiple Processes (-np)

By default, an MPI program started with `mprun` runs as one process. To run the program as multiple processes, use the `-np` option:

```
% mprun -np process–count program–name
```

When you request multiple processes, CRE attempts to start one process per CPU. If you request more processes than the number of available CPUs, you must use either the `-W` (Page 24) or `-S` (Page 24) options to prevent `mprun` from failing.

If you enter 0 as the number of processes, the runtime environment starts one process per available CPU. For example:

```
% mprun -np 4 a.out

% mprun -p partition2 -np 0 a.out
```

The first example runs four copies of the program `a.out` on the login partition. The second example runs the job on `partition2`, which has six CPUs. Because the second command specifies "0" processes, the runtime environment runs six copies of `a.out`, one for each available CPU.

```
% mprun -np process–count x threads program–name
```

When launching a multi-threaded program, use the **x** *threads* syntax to specify the number of threads per process. Although the job requires a number of resources equal to *process–count* multiplied by *threads*, only *process–count* processes are started. The ranks are numbered from `0` (zero) to *process–count* minus `1`. The processes are allocated across nodes so that each node provides a number of CPUs equal to or greater than *threads*. If threading requirements cannot be met, the job fails and provides diagnostic messages. As with a processor value of `0`, a thread value of `0` requests all available resources on the node. In this way it is equivalent to the `-Ns` option.

When using CRE as a resource manager, the syntax `-np` *process–count* is equivalent to the syntax `-np` *process–count*`x1`. The default is `-np 1x1`. For the other resource managers, `-np` is equal to 0 by default.

Note – If a batch job calls `MPI_Comm_spawn`(`3SunMPI`) or `MPI_Comm_spawn_multiple`(`3SunMPI`), be sure to use the `-nr` option to reserve the additional resources.

▼ To Share Nodes (-j)

To run a program on the same node(s) as another program, use the -j option:

```
% mprun -j jid [ mprun–options ] program–name
```

The jid argument is the program's *job ID* (described in "Jobs" on page 10).

Place additional *mprun–options*, if any, after the -j option. Here are two examples.

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

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

Both of the examples above run the program a.out on the same node as the program identified by the jid of 85. The second example includes the -Ns option to disable process spawning (Page 23).

▼ To Enable Process Spawning (-Ys)

To enable a program that runs on a node with multiple CPUs to spawn processes, use the -Ys option:

```
% mprun -Ys program–name
```

▼ To Disable Process Spawning (-Ns)

To limit the number of processes a program uses to one per node, use the -Ns option:

```
% mprun -Ns program–name
```

The -Ns option prevents nodes that have multiple CPUs from spawning additional processes.

▼ To Wrap Multiple Processes (-W)

Note – This option is incompatible with the -Z option.

When you have more processes than available CPUs, use the -W option to *wrap* the processes:

```
% mprun -np process–count -W program–name
```

Without the -W option, excess processes would make the job fail. The -W option assigns as many processes as required to each CPU, and executes the processes one at a time. (To include independent nodes in the wrap, use the -u option, described on page 25.)

For example:

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

If the partition part2 had six available CPUs and you specified 10 wrapped processes, CRE would distribute the processes among the CPUs according to load-balancing rules.

(The -S option, described below, provides a different solution to the same problem.)

▼ To Settle for Available Processes (-S)

Note – This option is incompatible with the -Z option.

When you have more processes than available CPUs, use the -S option to *settle* for the number of available CPUs.

```
% mprun -np process–count -S program–name
```

Without the -S option, excess processes would make the job fail. The -S option assigns one process to each CPU, and when it runs out of CPUs, it ignores the remaining processes. (To assign the remaining processes to independent nodes, use the -u option, described below.)

For example:

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

If the partition `part2` had six available CPUs and you specified 10 processes with the `-S` option, CRE would assign one process to each of the six CPUs, and discard the remaining four processes.

(The `-W` option, described on page 24, provides a different solution to the same problem.)

▼ To Include Independent Nodes (`-u`)

When a partition does not have enough CPUs to handle all the processes of a job, and you select either the `-S` option or the `-W` option, you can use the `-u` option to assign the extra processes to independent nodes outside the partition:

```
% mprun -np process–count -W -u program–name

% mprun -np process–count -S -u program–name
```

To be eligible, an independent node must satisfy three requirements:

1. It must be enabled.
2. It cannot belong to another partition that is currently enabled.
3. It must be running the same version of the Solaris Operating System as the nodes in the partition. For the current release of Sun HPC ClusterTools software, this OS must be Solaris 10.

For example, assume `partition2` had six available CPUs and the node had two independent nodes. If you specified 10 wrapped processes and added the `-u` option...

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

... CRE would distribute the ten processes among the 8 CPUs, and use load-balancing rules to assign the remaining two processes.

If you specified 10 processes with the –S option and added the –u option....

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

... CRE would assign one process to each of the six CPUs, one to each independent node, and discard the remaining two processes.

▼ To Combine Process Placement Options

As described in "To Run as Multiple Processes (–np)" on page 22, you can request *x* processes, if as many as *x* processors are available, using the –np option. For example,

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

If you specify 0 as the number of processes, the runtime environment starts one process per available CPU.

However, if you combine the –np option with the –Ns option (assign one process per node) or the –W option (assign processes to the available nodes until the –np argument is satisfied),

TABLE 4-1 Combining mprun Process Placement Options

Option Combination	Interpretation
mprun -np 0 -Ns a.out	Request a process on each node.
mprun -np *x* **-W a.out**	Request *x* processes, without regard to distribution on the nodes.
mprun -np 0 -Ns -W a.out	Request one process per node, wrap until all processors in your cluster are used.
mprun -np x -Ns -W a.out	Request one process per node until your cluster has *x* of them.

Mapping MPI Processes to Nodes

To Perform This Task	Use This Option
How to distribute processes among nodes	`-l`
How to distribute processes by block	`-Z` or `-Zf`
How to distribute processes by rank map	`-m`
How to select nodes by resource requirement	`-R`

If you assign to a node a number of processes that is greater than the number of CPUs on that node, the runtime environment complies with your request unless the value of `total_max_procs` prevents it.

Precedence for Mapping

Four primary `mprun` options affect rank placement: `-l`, `-m`, `-Z`, and `-Zt`. Four ancillary options also influence rank placement: `-W`, `-S`, `-np`, and `-u`. The following table summarizes an interaction matrix for these options:

This Option	Nullifies Previous Instances of	And Ignores
`-l`	`-l -m -Z -Zt -j -R`	
`-m`	`-l -m -Z -Zt -j -R`	
`-R`	`-l -m -Z -Zt -j -R`	
`-j`	`-l -m -Z -Zt -j -R`	`-u`
`-Z`	`-l -m -Z -Zt -j`	`-Ns -Ys`
`-Zt`	`-l -m -Z -Zt -j`	`-Ns -Ys`

▼ To Distribute Processes Among Nodes (`-l`)

To distribute processes among individual nodes, use the `-l` option following the `-np` option:

```
% mprun -np process–count -l rank–spec program–name
```

process–count

The –np option (described in "To Run as Multiple Processes (-np)" on page 22) specifies the number of *processes* the program uses.

rank–spec

The *rank–specs* specify how many processes go to each node. Be sure to enclose the set of *rank–specs* with one set of quotation marks, and use commas to separate them from each other:

"*rank–spec*, *rank–spec*, *rank–spec*"

The number of *rank–specs* you use must be a factor of the number of *processes* you specify with the -np option. For example:

```
% mprun -np 1 -l "node0" a.out
% mprun -np 2 -l "node0, node1" a.out
% mprun -np 3 -l "node0, node1, node2" a.out
```

The examples above use one *rank–spec* for one process, two *rank–spec*s for two processes, and three *rank–spec*s for three processes. You cannot use three *rank–spec*s with four processes, for instance, because four processes cannot be evenly distributed across three nodes.

Each *rank–spec* identifies one node and the number of processes that run on it:

rank–spec --> *node–name* [*process–count*]

The *node–name* can be a name or an IP address. The *process–count* argument is optional. If you omit it, as in the examples above, one process is assigned to each node. If you have more processes than nodes, you must include the *process–count* argument to indicate how many processes are assigned to each node. For example:

```
% mprun -np 2 -l "node0 2" a.out
```

In the example above, the program runs with two processes on one node, node0, so you must indicate that both processes are assigned to node0.

In the following example, the program runs with four processes on two nodes, so you must indicate how those processes are assigned to the nodes. Three combinations are possible:

```
% mprun -np 4 -l "node0 2, node1 2" a.out
% mprun -np 4 -l "node0 1, node1 3" a.out
% mprun -np 4 -l "node0 3, node1 1" a.out
```

▼ To Distribute Processes by Block (–Z and –Zt)

Note – –Z is incompatible with –S or –W.

You can arrange a job's processes into blocks. The blocks of processes are then distributed among the nodes. The –Z option distributes the blocks among the available nodes using load balancing. In other words, two blocks may be assigned to the same node if that is the most efficient way to execute the job. To force each block to be assigned to a separate node instead, use the –Zt option. Use the –Z or –Zt option ahead of the –np option:

```
% mprun -Z block–count -np process–count program–name
% mprun -Zt block–count -np process–count program–name
```

Here are some examples:

```
% mprun -Z 2 -np 4 a.out
% mprun -Zt 2 -np 4 a.out
```

In the example above, the –Z option specifies two blocks. Because the total number of processes is four (–np 4), each block has two processes. They are distributed among available nodes as efficiently as possible. The –Zt option also creates two blocks, each with two processes, but they are distributed to two separate nodes.

Here are more examples:

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

Both examples above create three blocks, two with three processes each, and one with two processes.

▼ To Distribute Processes by Rank Map (–m)

To distribute processes among nodes with a rank map file, use the –m option:

```
% mprun -np process–count -m rankmap–file program–name
```

Use the –m *rankmap–file* option to assign processes to nodes as specified in the file *rankmap–file*. The rankmap in the file is specified as one or more nodenames, each followed optionally by the number of processes to assign to that node (in rank order); the default is one. The rankmap file can also accept IP addresses instead of nodenames.

Multiple nodenames (or IP addresses) may be separated by newlines; if multiple nodenames appear on the same line, they are separated by commas.

You can obtain the names and IP addresses of nodes using the –Nv option to the mpinfo command.

```
% mpinfo -Nv
```

Restrictions

The rank map specified with the –m option will be rejected if any of the following conditions are true:

- One or more of the requested nodes is not enabled or is otherwise invalid
- The max_total_procs value set via the mpadmin command defeats the requested number of ranks for a node
- The requested nodes span multiple enabled partitions
- The requested nodes are running different versions of the operating system
- One or more of the following options is listed either in the command line or in the MPRUN_FLAGS environment variable: –j, –Ns, –R, –Ys, or –Z

process–count

If the *process–count* used with the –np option is greater than the number of ranks specified in the rank map, you must use either –S (to settle for the available number of ranks in the rank map) or –W (to wrap the requested processes on the specified nodes). Otherwise your job will fail.

If the value specified in the –np option is less than the number of ranks specified in the rank map, the rank assignment will be limited to the value of –np.

If you use –np 0, the number of processes will be derived from the number or ranks described in the rank map.

rankmap–file

A rank map file has this syntax:

rank–map file–-> *node–name* [,]
node–name [,]
node–name [,] ...

A *node–name* can be a name or an IP address. Since commas can be used to separate node names in a file, you could simply place the contents of an inline rank map in a file. However, new-line characters (\n) are also recognized as separators in rank map files, so you will probably find it easier to list each node on its own line. For example:

```
mars 2
venus 2
jupiter 2
```

▼ To Reserve Resources For Spawning or Multithreading (–nr)

This syntax reserves a number of resources equal to *numprocs* x *threads*. These resources are held in reserve over and above the number of resources specified by the –np option. Use this option when the batch job contains calls to MPI_Comm_spawn(3SunMPI) or MPI_Comm_spawn_multiple(3SunMPI). Specify a number of resources equal to or greater than the total number of processes that will be spawned. For example,

```
% mprun -nr numprocs [ x threads ]...
```

In a multithreaded environment, use the *x*threads syntax to specify the number of threads per process. The syntax `-nr` *numprocs* is equivalent to the syntax `-nr` *numprocs*`x1`. The default is `-nr 0x1`.

A *threads* setting of `0` allocates the processes among all available reserved resources. It is equivalent to the `-Ns` option.

▼ To Select Nodes by Resource Requirement (`-R`)

To distribute processes among nodes by resource requirement, use the `-R` option:

```
% mprun -np process–count -R resource–requirement–spec program–name
```

process–count

The processes are distributed among the nodes that satisfy the criteria in the *resource requirement spec* (RRS).

resource–requirement–spec

The RRS accommodates computing requirements that are more complex than those accepted by rank maps. It has this syntax:

RRS `-->` `"`*resource–requirement* `[&` | | *resource–requirement* `]...`"

The `&` symbol is a logical `AND` operation. In other words, a node must satisfy all the criteria in the spec. The | symbol is a logical `OR` operation. A node must satisfy either of the criteria in the spec. Use them alone or in combination:

resource–requirement `&` *resource–requirement*
resource–requirement | *resource–requirement*

Each individual *resource–requirement* has this syntax:

resource–requirement `-->` *resource* `[` *operator value* `]`

The *resource* argument identifies the resource whose requirement is specified. For a list of resources, see TABLE 4-2.

The *operator* argument is an arithmetic or logical symbol such as = or > that indicates the relationship between the *resource* and its *value*. For example:

```
"name=node0"
```

In the example above, the processes are distributed to a node whose `name` resource is equal to `node0`. For a list of *operators*, see TABLE 4-3.

The *value* argument is simply the value of the *resource* that must be met. Although the *operator* and *value* are optional, they are used in the great majority of cases.

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

The result is an ordered list of CPUs that meet your requirements. If a job uses only one process, the process is sent to the first CPU on the list. If a job uses *n* processes, they are distributed among 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.

TABLE 4-2 lists the predefined *resources* you can use. Your system administrator may have defined additional resources for your particular cluster. To display them, use the `mpinfo` command described in Chapter 10.

TABLE 4-2 Predefined Resources

Resource	Description
`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_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.
`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 host name.
`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.

TABLE 4-2 Predefined Resources *(Continued)*

Resource	Description
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.

TABLE 4-3 RRS Operators

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.)
!	Boolean FALSE.
<<	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 -
=, !=, >=, <=, >, <, <<, >>
!
&, |
?
```

Examples

Here are some examples of resource requirement specifiers in use.

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

% mpinfo -N -R "partition.name=part1"

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

The last example specifies that you only want nodes whose individual load averages over the previous five minutes were less than four.

When the value of an attribute contains a floating point number or a string decimal number, you must enclose the number in single quotes. For example:

```
% mpinfo -R "os_release='5.8'"
```

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

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

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

If you use the << or >> operator, CRE does not provide load-balancing. In the previous example, 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
```

The example above 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 two. 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 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
```

The example above specifies that the node must have over 256 Mbytes of available RAM.

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

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

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 this command:

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

Controlling Input/Output

To Perform This Task	Use This Option
How to redirect output to `mprun`	`-D`
How to redirect output to individual files	`-B`
How shut off all standard I/O	`-N`
How to redirect with an argument vector	`-A`
How read standard input from `/dev/null`	`-n`
How to redirect with a custom configuration	`-I`

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 behavior 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.

Likewise, the `mprun` standard input (`stdin`) is sent to the standard input of all the processes.

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

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

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

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

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

Precedence for Input/Output

Option	Nullifies Previous
`-B`	`-D -N -B -I`
`-D`	`-D -N -B -I`
`-I`	`-D -N -B -I`
`-N`	`-D -N -B -I`

The set of `mprun` options that controls `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`.

▼ To Redirect Output to `mprun` (`-D`)

To redirect a job's `stdout` and `stderr` to those of the `mprun` command, use the `-D` option:

```
% mprun -D program–name
```

▼ To Redirect Output to Individual Files (-B)

You can merge the standard output and standard error streams from each process and direct them to individual files by using the `-B` option.

```
% mprun -B program–name
```

The `-B` option writes one file for each process. The filename has this nomenclature:

`out.`*jid*`.`*rank*

The *jid* is the program's job ID. The *rank* is the rank of the process. The files are stored in the job's working directory.

▼ To Shut Off All Standard I/O (-N)

To shut off all standard I/O to all processes, use the `-N` option:

```
% mprun -N program–name
```

This option closes all `stdin`, `stdout`, and `stderr` connections for the job. For instance, 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.

▼ To Redirect With an Argument Vector (-A)

By default, `mprun` passes the vector of a program's command-line arguments to the program in the standard way. In cluster-level programming, it is sometimes useful to specify a first argument that is not the name of the program. You can use the `-A` option to do this.

```
% mprun -A program–name  argument . . .
```

The argument to `-A` is the name of the program to be executed. After the program name you can add the argument of your choice. 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 and arg1 and arg2 are the second and third elements. Or, to pass newarg as the first argument to the program a.out, along with arg1 and arg2, you could issue the command:

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

▼ To Read Standard Input From /dev/null (-n)

To read stdin from /dev/null, use the -n option:

```
% mprun -n program–name
```

Reading input from /dev/null can be useful when running mprun in the background, either directly or through a script. Without -n, mprun would block in this situation, even if no reads were posted by the remote job. With -n, the user process encounters an EOF if it attempts to read from stdin. This behavior is similar to the behavior of the -n option to rsh.

▼ To Redirect With a Custom Configuration (-I)

To redirect output with a custom configuration, use the -I option:

```
% mprun -I custom–configuration program–name
```

custom–configuration

A custom configuration tells the runtime environment how to handle each job's I/O streams (standard input, output, and error). It has this syntax:

custom–configuration —> *file–descriptor* [, *file–descriptor*]...

file–descriptor

Each *file–descriptor* provides handling instructions for one process. It has this syntax:

file–descriptor --> *stream-number attribute*

Quotation marks are optional. You can place the *file–descriptors* in any order. A custom configuration can include a *file–descriptor* for each stream associated with a job; if any *file–descriptor* is omitted, its stream is not connected to any device.

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`.

stream–number

The *stream* identifies the input, output, or error stream. The standard I/O streams are assigned these numbers:

Stream	Stream Number
standard input (`stdin`)	`0`
standard output (`stdout`)	`1`
standard error (`stderr`)	`2`

attribute

The handling instructions for each *stream* are specified by the *attribute.*

Attribute	Description	Dependencies
`r`	Read from the stream	
`w`	Write to the stream	
`p`	Attach the stream to a pseudo-terminal (`pty`)	
`b`	Input only goes to the first process	Must use with `r`
`i`	Input only goes to rank 0, not to any other ranks	
`l`	Make the output line-buffered	Must use with `w`
`t`	Tag the line-buffered output with rank number	Must use with `w`
`a`	Append the stream to a file	Must use with `w`
`m`	Echo keystrokes multiple times for multiple processes	Must use with `rp`

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`.

Note – NFS does not support append operations.

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
```

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 do not 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"
[Return]
Hello
Hello
```

Be sure to press the Return or Enter key to begin the output.

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"
[Return]
r0:Hello
r1:Hello
```

As with the -l option, be sure to press the Return or Enter key to begin the output.

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.

Redirecting Output to Other File Descriptors

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

fdattr=@other_fd

For example, 2w=@1means 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.

Redirecting File Descriptor Output to a File

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

fdattr=filename

For example, 10w=output means 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. 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.

Maximum Number of File Descriptors

By default, the maximum number of file descriptors that a process can have open is 1024. This is because CRE enforces only the hard limit for file descriptors and ignores any file descriptor soft limit that may be set.

Note – CRE enforces soft limits for all other kernel parameters.

The default, per-process limit of 1024 file descriptors is likely to be more than enough for all but the most extreme MPI job execution requirements. You can, however, easily accommodate exceptional file descriptor demands by taking the following steps:

- Compiling and linking the MPI application to 64-bit libraries
- Increasing the open file descriptor limit to a value that will satisfy expected demands

For example, to increase the file descriptor hard limit to 2048, add the following line to the `/etc/system` file on each node in the cluster:

```
set rlim_fd_max=2048
```

Using `mprun` Options Instead of 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 4-4 mprun I/O 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.

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.

These shortcuts are not exact substitutions. CRE uses ptys correctly, whether the -I option is present or absent. Also, 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.

Controlling Other Job Attributes

To Perform This Task	Use This Option
How to include shell-specific actions	
How to move a process to the background	
How to change the working directory	`-C`
How to use a different user name	`-U`
How to use a different group name	`-A`
How to run a job on a different project	`-P`
How to display command help	`-h`
How to display the command's version number	`-V`
How to display job status information	`-J`
How to store the job name in a file	`-d`
How to tag output with its rank number	`-o`

▼ To Include Shell-Specific Actions

To perform actions that are shell specific, such as executing compound commands, invoke the appropriate shell as part of the `mprun` command:

```
% mprun shell–command  shell–options
```

Here are two examples:

```
% mprun csh -c 'echo $USER'

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

▼ To Move a Process to the Background

To move either a process started with `mprun` or a script that issues `mprun` commands to the background, redirect `stdin` to a file, like this:

```
% mprun < /dev/null program–name
```

You can also use the `-n` option to `mprun` so that standard input is read from `/dev/null`. See "To Read Standard Input From `/dev/null` (`-n`)" on page 40.

```
% mprun -n program–name
```

When `mprun` stops, whether via Control–Z or in terminal output, the job under control of `mprun` is stopped.

▼ To Change the Working Directory (`-C`)

Use the `-C` option to specify the path of an alternative working directory to be used by the processes spawned when you run your program:

```
% mprun -C working–directory program–name
```

Setting a path with `-C` does not affect where the runtime environment looks for executables. If you do not specify `-C`, the default is the current working directory. For example:

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

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

▼ To Use a Different User Name (`-U`)

To start a program with a different user name or ID, use the `-U` option:

```
% mprun -U username program–name
% mprun -U userid program–name
```

If you are not the user identified by *username*, you must have superuser privileges.

▼ To Use a Different Group Name (-G)

To start a program with a different group name or ID, use the -G option:

```
% mprun -G group–name  program–name
% mprun -G groupid  program–name
```

You must belong to the group you use, or be the superuser.

▼ To Run a Job on a Different Project (-P)

For accounting purposes, any job you run is part of your current project. You can set a default project by changing the value of the variable SUNHPC_PROJECT. That value overrides your current project. However, you can override both values by adding the -P option to the mprun command:

```
% mprun -P project–name
```

▼ To Specify Verbose Output (-v)

Use this syntax to specify verbose output. For example,

```
% mprun -v
```

▼ To Display Command Help (–h)

To display a list of `mprun` options, use the `–h` option (alone):

```
% mprun -h
USAGE:   mprun {options} [-] <exec> [<arg1> [<arg2> ...]]
  where {options} may include:
        -h           Displays this help/usage text
        -V           Displays tool version information
        -c <cluster> Specifies the cluster to use
        -p <partition>  Specifies the partition to use
        -A <aout>    Specify the argv [0] explicitly
        -U <uid>     Specify uid to execute as
        -G <gid>     Specify gid to execute as
        -I <iofds>   Specify the I/O fd set to multiplex
        -Is          Specify CRE I/O (use with -x)
        -C <path>    Specify an alternate working directory
        -P <project> Specify a project name
        -r <path>    Chroot to working dir before execution
        -J           Show job id after exec
        -np <PxT>    Specify the number of processes/threads in job
       -nr <PxT>    Specify the number of processes/threads to reserve
        -R <rrs>     Specify Resource Requirement String
        -W           Allow wrapping of hosts
        -S           Settle for available hosts
        -j <job name>  Run this job on same resources as <job name>
        -i           Only rank 0 gets stdin
        -o           Rank-tag stdout
        -D           Separate stdout/stderr streams
        -N           No stdio connections
        -B           Batch stream handling
        -n           No stdin connection
        -Ns          No spawning on SMP's
        -Ys          Enable spawning on SMP's
        -Z <n>       Group procs <n> to an SMP
        -Zt <n>      Group/tile procs <n> to an SMP
        -l "<host> [<procs>][,...]" Specify rankmap string
        -m <file>    Specify rankmap file
        -u           Use any partition independent nodes
      -t <n>      Multiply daemon and mprun timeouts by factor n; n > 1
        -d <filename>  Dump JID to a file
       -v          Verbose. Gives extra information during job startup.
       -x <RM>      Run processes under control of resource manager RM
```

▼ To Display the Command's Version (-V)

To display the command's version number, use the -V (upper case) option (alone):

```
% mprun -V
```

▼ To Display Job Status Information (-J)

To display information about the job after it finishes executing, add the -J option to the command:

```
% mprun options -J program–name
```

In this example, the job ID (*jid*), cluster name, and number of processes are displayed after the job finishes executing:

```
% mprun -np 4 -J a.out
```

▼ To Store Job Name in a File (-d)

To store the job name in a user-specified file for later access, use the -d option:

```
% mprun options -d output–file hostname
```

▼ To Tag Output With Its Rank Number (-o)

To precede each output line with the number of the rank that wrote it, use the -o option:

```
% mprun options -o program–name
```

Command Reference (mprun)

TABLE 4-5 Options for mprun

Option	Description
-A	Redirect output with an argument vector
-B	Redirect stderr and stdout output streams to individual files
-C	Run the program using a different working directory
-c	Run the job on a different cluster
-d	Store the job name in a file
-D	Redirect output to mprun
-G	Start the program with a different group name
-h	List the options of the mprun command
-I	Redirect output with a custom configuration
-i	Standard input is sent only to rank 0
-J	Display a program's *jid* and number of processes after it finishes executing
-j	Run a program on the same nodes as another program
-l	Distribute processes among nodes
-m	Distribute processes among nodes as specified in a rank map file
-n	Read standard input from /dev/null
-N	How to shut off all I/O connections
-np	Run a program on multiple processes
-Ns	Run a program with process spawning disabled
-o	Tag each output line with the rank of the process that wrote it.
-p	Run the program on a different partition
-P	Run the job as part of a different project
-R	Distribute nodes among processes using a resource requirement spec
-S	Settle for the available number of processes
-U	Start the program with a different user name

TABLE 4-5 Options for `mprun` *(Continued)*

Option	Description
`-u`	Include independent nodes when you distribute processes among the nodes of a partition
`-V`	Display the command's version information
`-W`	Wrap multiple processes around available nodes
`-Ys`	Execute the program with process spawning enabled
`-Z`	Distribute processes among nodes by block
`-Zt`	Distribute processes among nodes by block, but force each block to use a different node

CHAPTER 5

Running Programs With `mprun` in Distributed Resource Management Systems

This chapter describes the options to the `mprun` command that are used for distributed resource management, and provides instructions for each resource manager. It has four sections:

`mprun` Options for DRM Integration

Call `mprun` from within the resource manager, as explained in "Integration With Distributed Resource Management Systems" on page 3. Use the –x flag to specify the resource manager, and the `-np` and `-nr` flags to specify the resources you need. In addition, the `-Is` flag selects the default CRE I/O environment, the `-v` flag produces verbose output, and the `-J` flag displays a fuller identification of each process.

Some mprun flags do not make sense for a batch job, and will cause the `mprun` request to be rejected if used with the –x flag. See "Improper Flag Combinations for Batch Jobs" on page 57.

These are the DRM integration options for `mprun`:

`-Is`

When launching mprun from a resource manager, the `-Is` option selects CRE's default I/O behavior, overriding the I/O behavior of the resource manager. You do not need this option when using any of these mprun flags: `-I`, `-D`, `-N`, `-B`, `-n`, `-i`, or `-o`.

Each of those flags already invokes CRE's default I/O behavior. You also do not need this option if you prefer to keep the resource manager's default I/O behavior.

`-np` *numprocs* [`x` *threads*]

Request the specified number of processes in a job. The default is 1 if you are using CRE. For other resource managers, the default is 0. . Use the argument 0 to specify that you want to start one process for each available CPU, based on your resource requirements.

When launching a multithreaded program, use the `x` *threads* syntax to specify the number of threads per process. Although the job requires a number of resources equal to *numprocs* multiplied by *threads*, only *numprocs* processes are started. The ranks are numbered from 0 to *numprocs* minus 1. The processes are allocated across nodes so that each node provides a number of CPUs equal to or greater than *threads*. If threading requirements cannot be met, the job fails and provides diagnostic messages.

A *threads* setting of `0` allocates the processes among all available resources. It is equivalent to the `-Ns` option.

The syntax `-np` *numprocs* is equivalent to the syntax `-np` *numprocs*`x1`. The default is `-np 1x1`.

If your batch job calls `MPI_Comm_spawn`(`3SunMPI`) or `MPI_Comm_spawn_multiple`(`3SunMPI`), be sure to use the `-nr` option to reserve the additional resources.

Note – The `-np` switch is not supported under LSF. The LSF resource manager selects where the processes will run and how many. This cannot be changed by using the `-np` switch with LSF. If you do specify the `-np` switch under LSF, you will get an error message.

`-x` *resource_manager*

The `-x` option specifies the *resource_manager*. Supported *resource_ managers* are:

- `pbs`
- `lsf`

- sge

  If you set a default resource manager in the hpc.conf(4) file, mprun is automatically launched with that resource manager and you don't have to use the -x option. To override the default, use the -x *resource–manager* flag.

-v

Verbose output. Each interaction of the CRE environment with the resource manager is displayed in the output.

Improper Flag Combinations for Batch Jobs

Do not use the following flags with the –x *resource_manager* flag; if you do, the mprun request will be rejected:

- -C
- -G
- -j
- -l
- -m
- -S
- -U
- -W
- -Z

Running Parallel Jobs in the PBS Environment

First reserve the number of resources by invoking the qsub command with the -l option. The -l option specifies the number of nodes and the number of processes per node. For example, this command sequence reserves four nodes with four processes per node for the job myjob.sh:

```
% qsub -l nodes=4:ppn=4 myjob.sh
```

Once you enter the PBS environment, you can launch an individual job or a series of jobs with `mprun`. Use the `-x pbs` option to the `mprun` command. The `mprun` command launches the job using the rankmap file produced by PBS and stored in the environment variable `PBS_NODEFILE`. The job ranks are children of PBS, not CRE.

You can run a CRE job within the PBS environment in two different ways:

▼ To Run an Interactive Job in PBS

1. **Enter the PBS environment interactively with the `-I` option to `qsub`, and use the `-l` option to reserve resources for the job.**

   Here is an example.

```
hpc-u2-6% qsub -l nodes=1:ppn=2 -I
```

   The command sequence shown above enters the PBS environment and reserves one node with two processes for the job. Here is the output:

```
qsub: waiting for job 20.hpc–u2–6 to start
qsub: job 20.hpc–u2–6 ready
Sun Microsystems Inc. SunOS 5.10 Generic January 2005
pbs%
```

2. **Launch the `mprun` command with the `-x pbs` option.**

   Here is an example that launches the `hostname` command with a verbose output:

```
pbs% mprun -x pbs -v hostname
```

The hostname program uses the rankmap specified by the PBS environment variable PBS_NODEFILE. The output shows the hostname program being run on ranks r0 and r1:

```
[mprun:/opt/SUNWhpc/lib/pbsrun -v --
/opt/SUNWhpc/lib/mpexec -x pbs -v --/usr/bin/hostname]

[pubsrun:r0-r1:/opt/SUNWhpc/lib/mpexec -x pbs -v
-- /usr/bin/hostname]

[mpexec:r0:/usr/bin/hostname]

[mpexec:r1:/usr/bin/hostname]
```

▼ To Run a Script Job in PBS

1. **Write a script that calls** mprun **with the** -x pbs **option.**

   As described on page 56, the -x flag identifies the resource manager that will be used for the job launched by mprun. In the following examples, the script is called myjob.csh. Here is an example of the script.

   ```
   mprun -x pbs -v hostname
   ```

   The line above launches the hostname program in verbose mode, using PBS as the resource manager.

2. **Enter the PBS environment and use the** -l **option to** qsub **to reserve resources for the job.**

   Here is an example of how to use the -l option with the qsub command.

   ```
   hpc-u2% qsub -l nodes=1:ppn=2 myjob.csh
   ```

   The command sequence shown above enters the PBS environment and reserves one node with two processes for the job that will be launched by the script named myjob.csh.

3. **Invoke the script.**

Here is the output to the script `myjob.csh`.

```
[mprun:/opt/SUNWhpc/lib/pbsrun -v
--/opt/SUNWhpc/lib/mpexec-x pbs -v --/usr/bin/hostname]

[pbsrun:r0-r1:/opt/SUNWhpc/lib/mpexec -x pbs -v
-- /usr/bin/hostname]

[mpexec:r0:/usr/bin/hostname]

[mpexec:r1:/usr/bin/hostname]
```

As you can see, because the `mprun` command was invoked with the `-x pbs` option, it calls the `pbsrun` command, which calls `mpexec`, which forks into two calls of the `hostname` program, one for each node.

Running Parallel Jobs in the LSF Environment

▼ To Run an Interactive Job in LSF

1. **Enter the LSF environment with the `bsub` command.**

   a. **Use the `-Is` option to select interactive mode.**

   b. **Use the `-n` option to reserve resources for the job.**

c. Use the `-q` option to select the queue.

Here is an example that shows how to use the `-q` option.

```
burl-ct-v4% bsub -n 4 -q short -Is csh
```

The command sequence shown above enters the LSF environment in interactive mode, reserves 4 nodes, and selects the `short` queue. Here is the output:

```
Job <24559> is submitted to queue <short>
<<Waiting for dispatch...>>
<<Starting on burl-ct-v4>>
burl-ct-v4
```

2. **Use `pam` and `mprun` as shown below to start the parallel job.**

   `pam` requires the `-g` switch, which specifies the generic job launcher framework. `mprun` requires the `-x lsf` switch in order to specify that it is running under the LSF resource manager.

```
 burl-ct-v4-4 137 =>bsub -n 4 -q short -Is
bsub> No command is specified. Job not submitted.
 burl-ct-v4-4 138 =>bsub -n 4 -q short -Is tcsh
Job <1151> is submitted to queue <short>.
<<Waiting for dispatch ...>>
<<Starting on burl-ct-v4-5>>
 burl-ct-v4-5 41 =>pam -g mprun -x lsf -v hostname
[lsf hostlist: burl-ct-v4-5 4]
[aout_exec:  /usr/bin/newtask -p default /opt/SUNWhpc/lib/mpexec -k 4
-x lsf -v -- /hpc/rte/LSF/cluster1/6.2/sparc-sol10-64/bin/TaskStarter
-p burl-ct-v4-5:36519 -c /hpc/rte/LSF/cluster1/conf -a SOL64
/usr/bin/hostname]
[mpexec: r0: /hpc/rte/LSF/cluster1/6.2/sparc-sol10-64/bin/TaskStarter
-p burl-ct-v4-5:36519 -c /hpc/rte/LSF/cluster1/conf -a SOL64
/usr/bin/hostname]
[mpexec: r1: /hpc/rte/LSF/cluster1/6.2/sparc-sol10-64/bin/TaskStarter
-p burl-ct-v4-5:36519 -c /hpc/rte/LSF/cluster1/conf -a SOL64
/usr/bin/hostname]
[mpexec: r2: /hpc/rte/LSF/cluster1/6.2/sparc-sol10-64/bin/TaskStarter
-p burl-ct-v4-5:36519 -c /hpc/rte/LSF/cluster1/conf -a SOL64
/usr/bin/hostname]
[mpexec: r3: /hpc/rte/LSF/cluster1/6.2/sparc-sol10-64/bin/TaskStarter
-p burl-ct-v4-5:36519 -c /hpc/rte/LSF/cluster1/conf -a SOL64
/usr/bin/hostname]
burl-ct-v4-5
burl-ct-v4-5
burl-ct-v4-5
burl-ct-v4-5
[Job lsf.1151 on burl-ct-v4-5: r0: exit status 0]
[Job lsf.1151 on burl-ct-v4-5: r1: exit status 0]
[Job lsf.1151 on burl-ct-v4-5: r2: exit status 0]
[Job lsf.1151 on burl-ct-v4-5: r3: exit status 0]
Job  mprun -x lsf -v hostname

TID   HOST_NAME   COMMAND_LINE          STATUS          TERMINATION_TIME
===== ========== ================  ======================  =========
==========
00000 burl-ct-v4 /usr/bin/hostnam  Done                    02/28/2006
13:16:35
00001 burl-ct-v4 /usr/bin/hostnam  Done                    02/28/2006
13:16:35
00002 burl-ct-v4 /usr/bin/hostnam  Done                    02/28/2006
13:16:35
00003 burl-ct-v4 /usr/bin/hostnam  Done                    02/28/2006
13:16:35
 burl-ct-v4-5 42 =>
```

▼ To Run a Script Job in LSF

1. **Write a script that calls `mprun` with the `-x lsf` option.**

   As described on page 56, the `-x` flag identifies the resource manager that will be used for the job launched by `mprun`. Here is an example of the script.

```
mprun -x lsf -v hostname
```

   The line above launches the `hostname` program in verbose mode, using LSF as the resource manager.

2. **Enter the LSF environment with the `bsub` command.**

   a. **Use the `-n` option to reserve resources for the job.**

   b. **Use the `-q` option to select the queue.**

   c. **Invoke the script. Make sure you precede it with `pam -g`.**

```
 burl-ct-v4-4 139 =>bsub -n 4 -q short pam -g myjob.csh
Job <1152> is submitted to queue <short>.
 burl-ct-v4-4 140 =>
```

   The command sequence shown above enters the LSF environment, reserves 4 nodes, selects the `short` queue, and invokes the script `myjob.csh`, which calls `mprun`.

Running Parallel Jobs in the SGE Environment

You can launch MPI programs from within SGE in two different ways:

- To Run an Interactive Job in SGE
- To Run a Script Job in SGE

Note – Sun N1 Grid Engine 6 (N1GE6) is the supported version of Sun Grid Engine in Sun HPC ClusterTools 6. For the purposes of this manual, N1GE6 is referred to as SGE.

Before you can use SGE with HPC ClusterTools, you need to set up the queue and parallel environment (PE) in SGE/N1GE. For information about how to set up the queue and the PE to work with HPC ClusterTools, refer to the *Sun HPC ClusterTools 6 Software Administrator's Guide*.

▼ To Run an Interactive Job in SGE

1. **Enter the SGE environment with the `qsh` command.**

   a. Use the `-pe` option to reserve resources for the job.

   b. Use the `cre` option to specify CRE as the parallel processing environment.

   Here is an example.

```
hpc-u2-6% qsh -pe cre 2
```

   The command sequence shown above enters the SGE environment in interactive mode, reserves 2 nodes, and specifies CRE as the parallel processing environment. Here is the output from the command sequence:

```
waiting for interactive job to be scheduled ...

Your interactive job 24 has been successfully scheduled.
```

2. **Enter the `mprun` command with the `-x sge` option.**

```
hpc-u2-6% mprun -x sge -v hostname
```

   The output shows the hostname program being run on ranks r0 and r1:

```
 [r0: aout: qrsh, args:
          qrsh -inherit -V hpc-u2-7 /opt/SUNWhpc/lib/mpexec
          -x sge -- hostname]

[r1: aout: qrsh, args:
          qrsh -inherit -V hpc-u2-6 /opt/SUNWhpc/lib/mpexec
          -x sge -- hostname]
```

▼ To Run a Script Job in SGE

1. **Write a script that calls `mprun` with the `-x sge` option.**

   As described on page 56, the `-x` flag identifies the resource manager that will be used for the job launched by `mprun`. Here is an example of a script.

```
set echo
mprun -x sge -v hostname
```

   The line above launches the `hostname` program in verbose mode, using SGE as the resource manager.

2. **Enter the SGE environment with the `qsub` command.**

   a. **Use the `-pe` option to reserve resources for the job.**

   b. **Use the `-cre` option to specify CRE as the parallel processing environment.**

   c. **In the `qsub` syntax, use the script name instead of the program name.**

   Here is an example of how to use `qsub` with the script name.

```
hpc-u2-6% qsub -pe cre 2 myjob.csh
```

   The command sequence shown above enters the CRE environment, reserves 2 nodes, and invokes the script `myjob.csh`, which calls `mprun`. Here is the output:

```
 your job 33 ("myjob.csh") has been submitted
```

   This is all you need to do to run the job.

3. **To display the output from the job, find the output file and display its contents.**

   a. **Use the `ls` command to list the files into which the script has loaded the output.**

   This example uses the job number to identify the output files:

```
hpc-u2-6% ls *33
myjob.csh.e33 myjob.csh.o33 myjob.csh.pe33 myjob.csh.po33
```

   The file that contains the job's errors is named `myjob.csh.e33`. The file that contains the job's output has the name `myjob.csh.o33`.

b. To view the job's output, display the contents of the job's output file.

Continuing with the example above:

```
hpc-u2-6% cat myjob.csh.o33

[r0: aout: qrsh, args:
qursh -inherit -V hpc-u2-6
/opt/SUNWhpc/lib/mpexec -x sge --hostname]

[r1: aout: qrsh, args:
qursh -inherit -V hpc-u2-7
/opt/SUNWhpc/lib/mpexec -x sge --hostname]
```

CHAPTER 6

Killing or Sending Signals to Programs With `mpkill`

What You Can Do

To Perform This Task	Use This Option
How to kill a running program	`mpkill`
How to remove all traces of a job	`-C`
How to display a list of signals	`-l -d`
How to send a signal to a job	*–signal*

Return Values

The `mpkill` command returns these values:

- `0` – The command executed successfully.
- `1` – An error occurred 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 that 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.

▼ To Kill a Running Program

To kill a running program, use the `mpkill` command and the program's job ID:

```
% mpkill jid
```

The `mpkill` command stops all the processes associated with the Job ID.

The job ID now begins with the name of the resource manager (`cre`, `lsf`, `pbs`, or `sge`). For example: `lsf.1289`. To obtain a program's job ID, use the `mpps` command, described in "To Display Information About Individual Jobs (`-J`)" on page 75.

▼ To Remove All Traces of a Job

If you have killed a job but it continues to appear in the output of the `mpps` command (described in Chapter 7), log in as root to the master node and invoke the `mpkill` command with the `-C` option and the *jid.*

```
% mpkill -C jid
```

The `-C` option purges the job from the CRE database, including unpublishing names associated with the job.

Note – Processes spawned in the ClusterTools Runtime Environment are not killed by the `mpkill` or `kill` commands so long as they have (spawned) child processes running. To remove the parent process, you must first remove all of its child processes.

▼ To Display a List of Supported Signals (`-l -d`)

To simply list the supported signals, use the `-l` option.

```
% mpkill -l
```

To display a list with brief descriptions, use the `-d` option.

```
% mpkill -d
```

▼ To Send a Signal to a Job

To send a signal to a job, use this syntax:

```
% mpkill –signal jid
```

For example:

```
% mpkill -CONT sge.59
```

The example above sends a `SIGCONT` signal to the processes of the program whose job ID is sge.`59`.

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

TABLE 6-1 Options for `mpkill`

Command	Description
none	Stop all processes associated with a particular job
`-C`	Remove all traces of a job, including unpublished names, from the CRE database
`-l`	Display a list of supported signals
`-d`	Display a descriptive list of supported signals
–signal	Send a signal to a job

CHAPTER 7

Displaying Program Information With `mpps`

What You Can Do

To Perform This Task	Use this Option
How to display job status	*none*
How to display information about individual jobs	`-J`
How to display name, PID, and host of current job	`-b`
How to display information about all jobs	`-e`
How to display a job's start time	`-f`
How to display job information by partition	`-A -a`
How to display job information by process	`-P -p`

▼ To Display Job Status

To display status information about your jobs running in the default partition, enter the `mpps` command without options:

```
% mpps
```

For example:

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

The status fields are described in TABLE 7-1.

TABLE 7-1 Job Status Displayed by `mpps`

`mpps` Output	Description
CORE	The job or process exited due to a signal and core was dumped.
CORING	The job is exiting due to a signal. The first process to die dumped core.
EXIT	The job or process exited normally.
EXITING	The job is exiting. At least one process exited normally.
FAIL	The job or process failed while starting, or was aborted.
FAILING	Initialization of the job failed, or a job abort has been signaled.
RUN	The job or process is running.
SIGNALING	The job or process exited due to a signal.
SIGNALED	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.
SPAWN	The job or process is being spawned.
STOPPED	The job or process is stopped.

▼ To Display Information About Individual Jobs (-J)

To display information about a job, use the -J option and a *job–attribute*.

```
% mpps -J job–attribute[,job–attribute...]
```

Separate multiple *job–attributes* either with a comma or a space, but not both.

TABLE 7-2 Job attributes for -J option to mpps

Attribute	Description
part	The name of the partition running the job
jobname	The job's unique ID, expressed as <*resource–manager*>.*jobname*
mprun_pid	The process identifier (PID) of the current mprun job
mprun_host	The host of the current mprun job
nproc	The number of processes requested (the actual number of processes started may differ if the -W ("To Wrap Multiple Processes (-W)" on page 24) or -S ("To Settle for Available Processes (-S)" on page 24) flags were used with mprun)
uid	The user on whose behalf the job was run (normally the user who submitted the job)
gid	The group on whose behalf the job was run (normally the group of the user who submitted the job)
state	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 in this job. Not always equal to the number of processes started for the job because processes that have exited are not counted
wkdir	The directory in which the job's processes start
aout	The name of the program
paout	The full path of the program
ctime	The time when mprun was invoked

TABLE 7-2 Job attributes for `-J` option to `mpps`

Attribute	Description
`args`	The command-line arguments of the program
`stime`	The time the job was started
`prio`	The job priority (higher numbers run first)

▼ To Display Job Name, PID, and Host of Current Job (`-b`)

Use the `-b` option to display job name, process identifier, and host of a current MPI job.

```
% mpps -b
```

▼ To Display Information About All Jobs (`-e`)

Use the `-e` option to display information about all jobs.

```
% mpps -e
```

▼ To Display a Job's Start Time (`-f`)

Use the `-f` option to display the start time for each job.

```
% mpps -f
```

▼ To Display Job Information by Partition (`-A -a`)

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

```
% mpps -A
```

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

```
% mpps -a partition–name
```

▼ To Display Job Information by Process (`-p -P`)

Use the `-p` option to include information about the processes that make up the jobs:

```
% mpps -p
```

For example:

```
% mpps -p
JID       NPROC  UID   STATE  AOUT   RANK  PID    STATE  NODE
lsf.2320  4      shaw  RUN    sleep  0     10190  RUN    node6
                                     1      4744  RUN    node7
                                     2     16564  RUN    node4
                                     3      9412  RUN    node5
```

The output fields are described in TABLE 7-3, below.

To display information about a particular process attribute, use the `-P` option:

```
% mpps -P process–attribute[,process–attribute...]
```

Separate multiple *process–attributes* either with a comma or a space, but not both. Use the attributes described in TABLE 7-3, below.

TABLE 7-3 Process attributes for `-P` option to `mpps`

Attribute	Description
`rank`	The rank of the process within the job
`pid`	The process ID
`state`	The current execution state of the process

TABLE 7-3 Process attributes for `-P` option to `mpps`

Attribute	Description
`iod`	The process ID of the I/O daemon for this process
`load`	The load on the node executing the process
`node`	The name of the node executing the process

Command Reference (`mpps`)

TABLE 7-4 Options for `mpps`

Option	Description
none	Display status information about your jobs running in the default partition
`-J`	Display information about a particular job
`-b`	Display name, process identifier, and host of current job
`-e`	Display information about all jobs
`-f`	Display the time a job started
`-A`	Display information about jobs running in all partitions
`-a`	Display information about jobs running in a particular partition
`-P`	Display process information about a job
`-p`	Display information about a particular process attribute

CHAPTER 8

Profiling Programs With MPPROF

This chapter explains how to use the MPProf utilities and environment variables to extract profiling information from executing MPI programs and generate formatted reports from that data. It also explains how to convert intermediate binary files to unanalyzed ASCII files. It has four sections:

Enabling MPI Profiling

To enable MPI profiling with MPProf, set the `MPI_PROFILE` environment variable to 1 before starting the MPI program.

```
% setenv MPI_PROFILE 1
```

When MPProf is enabled, it will extract information about calls to Sun MPI routines and store the information in a set of intermediate files. A separate data file is created for each MPI process rank in which MPI calls were made.

MPProf also creates an index file describing the locations of the intermediate files. The index file name has the form:

```
mpprof.index.rm.jid
```

where *rm* is the name of the resource manager and *jid* is the job ID. The report generator, `mpprof`, uses this index file to gather the profiling data and associate it with particular MPI processes.

Note – If the MPI program uses an instrumented loadable protocol module (PM), MPProf passes PM-related profiling data back to the PM. This allows the reporting of PM profile data independent of `mpprof`.

Controlling Data Collection

MPProf provides the following environment variables to control various aspects of MPProf behavior:

- `MPI_PROFDATADIR` – Specify a location where the intermediate files will be stored.
- `MPI_PROFINDEXDIR` – Specify a location where the index file will be stored.
- `MPI_PROFINTERVAL` – Specify an interval between intermediate file updates.
- `MPI_PROFMAXFILESIZE` – Specify the maximum size, in kilobytes, that can be written to any intermediate file.

These environment variables are explained in the following sections.

MPI_PROFDATADIR

The environment variable `MPI_PROFDATADIR` can be used to specify a nondefault location where the intermediate files will be created for each process. This directory must exist.

If profiling is enabled and `MPI_PROFDATADIR` specifies a directory that does not exist, MPProf will output an error message and abort the program. If `MPI_PROFDATADIR` is not set and profiling is enabled, intermediate files will be created in `/usr/tmp`.

MPI_PROFINDEXFDIR

The environment variable `MPI_PROFINDEXDIR` can be used to specify a nondefault location for storing the index file. This directory must exist.

If profiling is enabled and MPI_PROFDATADIR specifies a directory that does not exist, MPProf will output an error message and abort the program. If profiling is enabled and MPI_PROFINDEXDIR is not set, the index file will be stored locally in the current directory.

MPI_PROFINTERVAL

The environment variable MPI_PROFINTERVAL can be used to specify a data sampling period. When this value is a number greater than 0, the intermediate files will be updated at the prescribed intervals. The data recorded with each update represent the MPI activity that occurred since the previous update.

Setting MPI_PROFINTERVAL to 0 forces updates to be made for every MPI call. Setting MPI_PROFINTERVAL to Inf (meaning infinity) causes the intermediate files to be updated only once, at MPI_Finalize time. If MPI_PROFINTERVAL is unset or is set without a value, the default value of 60 seconds will be used.t

The following example sets an interval of 5 seconds:

```
% setenv MPI_PROFINTERVAL 5
```

In this case, the first update of intermediate files would occur approximately 5 seconds after the MPI_Init call, with additional updates appended to the file at 5-second intervals.

If time intervals are used and an MPI program terminates before the MPI_Finalize call, any updates that were made can be used by mpprof to generate a profile of program operations up to the point of termination.

MPI_PROFMAXFILESIZE

The environment variable MPI_PROFMAXFILESIZE can be used to specify, in kilobytes, the maximum amount of data a process can record in its intermediate file. The default value is 51,200 kilobytes (approximately 50 megabytes). Setting MPI_PROFMAXFILESIZE to unlimited removes any limits on the size to which the intermediate files can grow. Setting MPI_PROFMAXFILESIZE to 0 is an error and will cause MPProf to abort the program

If a write to a given intermediate file exceeds the MPI_PROFMAXFILESIZE limit, the write operation will complete, but that process will be unable to record additional profiling data. Profiling can continue for other processes that have not reached the file size limit.

Using mpprof to Generate Reports

This section shows how to use the mpprof command, which is used to generate reports from profiling information that is collected by the MPI library. To enable MPI profiling, you set the MPI_PROFILE environment variable to 1 before executing the mprun command.

When you use the mpprof command, you always specify an index file that points to files that contain profiling data. The naming convention for the index file is mpprof.index.*rm*.*jid*, where *rm* is the resource manager used and *jid* is the job ID assigned to the MPI program. When profiling is enabled, the MPI library creates the index file in the current directory by default.

Note – The mpdump command, discussed in the next section, is used to translate binary profiling data into ASCII text.

---

mpprof Command Syntax

The mpprof command syntax is

```
mpprof [-h] [-V] [-r] [-S] [-g output-directory] [-p process-set]
[-c columns] -s start-time] [-e end-time] [-o output-file]
index-file
```

where the following command options are available:

TABLE 8-1 mpprof Command Options

Option	Description
-h	Lists the optional arguments and exits.
-V	Displays the program version during startup.
-r	Removes the intermediate files associated with the specified *index-file*. By default, the remove command will use rsh for each node in order to unlink the files. If the -r option is used with the -S option, the secure shell (ssh) will be used instead of rsh. If the -r option is used with the –p option, only a subset of the files will be removed.
-S	Use the secure ssh and scp commands instead of rsh and rcp.

TABLE 8-1 `mpprof` Command Options

Option	Description
`–g`	Gathers (copies) the intermediate files associated with the specified index file to the specified *`output-directory`*. The file names will be preserved and will overwrite any existing files of the same name. A new index file will be created in the specified *`output-directory`* to be used with the files. If the `–p` option is used with the `–g` option, a subset of the files will be copied. .The `–g` option can be used with the `–r` option as a way to archive the intermediate data files. If the `–g` *`output-directory`* contains previously stored intermediate files, a new index file will be created. No data will be lost.
`–p`	Specifies a subset of the processes to be analyzed. The `–p` *`process-set`* option is entered as a comma-separated list of tuples. If the `–p` option is used with `–S` and/or `–g`, the operation will be performed only on the specified subset. If a process set encompasses the complete set of ranks, the `–p` option will have no effect. If the `–p` option is given a stride, the upper bound of the stride is omitted. For example, if a stride is specified as 1:4:2, the upper bound (4) is not included in the set. In this example, the resulting set is {1,3}. See the section "Reporting on Specific Processes" below for additional information.
`–c`	Specifies the output width of the generated report. The smallest number of *`columns`* supported is 35. When `mpprof` prints to a terminal window, the display width will be adjusted automatically unless this option is specified. Wrapping will occur if the terminal window has fewer than 35 columns.
`–s`	Specifies that any profiling data generated before the specified *`start-time`* should be ignored. This time reference is measured from the point at which the program calls BMPI_Init. The `–s` option is useful only when a profiling interval has been specified for the intermediate files. If no *`start-time`* is specified on the `mpprof` command line, the earliest time specified in any intermediate file is used to begin data recording.
`–e`	Specifies that any profiling data occurring after the specified *`end-time`* should be ignored. This time is also measured in seconds, beginning from the time the program calls MPI_Init. This option can be used with, or in place of, the `–s` option. If no *`end-time`* is specified, the latest time in any intermediate file is used for reporting.
`–o`	Specifies that the report should be written to the specified *`output-file`*. If this option is not used or if the *`output-file`* is '–', the report will be written to stdout. If the *`output-file`* already exists, it will be overwritten.

Generating a Message Passing Report

The `mpprof` command generates a report file based on processes listed in a specified index file. The following example generates a report file called `report.txt` that is based on process profile data stored in files that are specified in an index file called `mpprof.index.cre.14`:

```
% mpprof –o report.txt mpprof.index.cre.14
```

When you enter this command, a new file called `report.txt` is created in the current directory. If you did not use the `–o report.txt` command line option, the report would be sent to the standard output.

Reporting on Specific Processes

You can fine-tune profiling output by specifying a subset of processes of the job, using the `–p` option as shown in the following example:

```
% mpprof –p 0,3–5,9:12,18:27:3 –o report.txt mpprof.index.cre.14
```

The output report file is always specified before the index file. The preceding command causes these processes to be analyzed for the report: 0, 3, 4, 5, 9, 10, 11, 12, 18, 21, 24, and 27. To view the report, open the `report.txt` file in the current directory.

Reporting Processes That Occur After a Specified Time Interval

The following example runs the `mpprof` command to generate a report on all processes that occur starting 2 minutes (120 seconds) or later after a call to `MPI_Init`, which initializes the accumulation of data on processes:

```
% mpprof –s 120 –o report.txt mpprof.index.cre.14
```

The –s option is useful when you want to exclude startup routines from reporting because startup routines may not be critical to performance. To view the report, open the `report.txt` file in the current directory.

To Save Report Output for Later Use

If you want to save report output to a specific directory other than the current directory, use the `-g` option:

```
% mpprof -g report.txt mpprof.index.cre.14
```

You can combine the `-g` *reportname* option with the `-r` option, removing the intermediate data files created during the profiling process at the same time as you store the report output:

```
% mpprof -r -g report.txt mpprof.index.cre.14
```

A Sample Report

The sample report shown is the result of running the following command, which places the report in a file called `report.txt`:

```
% mpprof -p 0,3-5,9:12,18:27:3 -o report.txt mpprof.index.cre.39
```

Note – The file `mpprof.index.cre.39` was created when this MPI command was run:
`%` **`mprun -np 28 /opt/SUNWhpc/examples/mpi/connectivity`**
The command generates 28 separate files.

The sample report is shown below:

```
OVERVIEW
========

The program being reported on is "/opt/SUNWhpc/examples/mpi/connectivity,"
which ran as job name "cre.39" on SatOct 29 14:04:29 2005.

Profiled Time Range:

  Start at elapsed time 0.000030 secs
  End at elapsed time   0.008181 secs
  Total duration is     0.008151 secs
  Fraction spent in MPI 85.9%

Elapsed time is measured from the end of MPI_Init. Data is being reported for
```

```
12 processes of a 28-process job.

LOAD BALANCE
============

Data is being reported on 12 MPI processes. The following histogram shows how
these processes were distributed as a function of the fraction of the time the
processes spent in MPI routines:

  Number of MPI Processes

 10-|
    |
  9-|
    |
  8-|
    |
  7-|
    |
  6-|
    |
  5-|
    |
  4-|
    |
  3-|
    |
  2-|
    |
  1-|  # #      #      #          #       # #  #         ##        #      #
    |  # #      #      #          #       # #  #         ##        #      #
  0-+----+----+----+----+----+----+----+----+----+----+----+----+----+------
   84.5 84.7 84.9 85.1 85.3 85.5 85.7 85.9 86.1 86.4 86.6 86.8 87.0 87.2
     Percentage time in MPI

Rank   Hostname   MPI Time
  12  hpc-smp6-0    84.51%
  24  hpc-smp6-0    84.62%
  27  hpc-smp6-0    84.91%
   0  hpc-smp6-0    85.18%
   5  hpc-smp6-0    85.62%
  10  hpc-smp6-0    85.96%
   4  hpc-smp6-0    86.03%
   3  hpc-smp6-0    86.18%
  21  hpc-smp6-0    86.59%
  11  hpc-smp6-0    86.63%
  18  hpc-smp6-0    87.00%
   9  hpc-smp6-0    87.30%
```

```
Low MPI time for an MPI process may indicate the process has too much of a compute
load. A high compute load forces the other processes to wait, increasing their
MPI time.

To focus reporting on one or more particular processes you may use the -p option
on the command line for mpprof. Type "mpprof -h" for more information.

MPI ENVIRONMENT VARIABLES
=========================

MPI_POLLALL:
You ran with full polling of connections. This means that Sun MPI monitored all
connections for incoming messages, whether your program explicitly posted
receive requests for those connections or not. Typically, this leads to a
degradation in performance.

Suggestion: Set the environment variable MPI_POLLALL to "0".

Warning: If your program relies on MPI_Send to provide substantial internal
buffering of messages, this suggestion could result in deadlock. On the other
hand, that would be an indication that the program in not MPI compliant. If such
deadlock results, it may be resolved by disregarding this suggestion. Even
better performance could result, however, by modifying the program to post the
appropriate receives or, in some cases, by setting MPI environment variables to
increase internal buffers.

SHM
===

The loadable protocol module "shm" has nothing to report.


SUGGESTION SUMMARY
==================

Summary of environment variable suggestions:

  Set: MPI_POLLALL=0

In the C shell, these environment variables may be set by the following commands:

setenv MPI_POLLALL 0

In the Bourne or Korn shell, these environment variables may be set by the
following commands:

export MPI_POLLALL=0

BREAKDOWN BY MPI ROUTINE
```

```
========================

Here, averages over all MPI processes profiled are reported. The numbers in
parentheses roughly indicate the variations there are among all of the MPI
processes. These variations are computed as (1-min/max)/2 where "min" and "max"
are the minimum and maximum values, respectively, for each statistic reported.
A total of 5 different MPI APIs were called.

 MPI Routine          Time         Calls Made      Sent        Received
MPI_Barrier     0.001625 (49.8%)    1  (0.0%)     0  (0.0%)     0  (0.0%)
MPI_Comm_rank   0.000003 (13.5%)    1  (0.0%)     0  (0.0%)     0  (0.0%)
MPI_Comm_size   0.000002 (10.1%)    1  (0.0%)     0  (0.0%)     0  (0.0%)
MPI_Recv        0.003758 (39.3%)   27  (0.0%)     0  (0.0%)   108  (0.0%)
MPI_Send        0.001612 (43.2%)   27  (0.0%)   108  (0.0%)     0  (0.0%)

Where "Time" is in seconds and "Sent" and "Received" are in bytes.

TIME DEPENDENCE
===============

Here is a rough depiction of time variations in MPI usage over the reported time
range. The fraction of time spent in each of the top 4 MPI routines overall are
shown for 60.0-second time periods.

Each time period is specified with an integer. This integer roughly corresponds
to the number of time periods passed, with the first time period lasting from
0-60.0 seconds. There are 1 different time periods. Time periods may be missing
if no MPI calls were made during the period. Times for MPI calls that persist
over multiple reporting intervals will only be reported in a single interval;
these reported times may be greater than 100%.

period  MPI_Recv  MPI_Barrier  MPI_Send  MPI_Comm_rank
     1      0.0%         0.0%      0.0%           0.0%

CONNECTIONS
===========

A connection is a sender/receiver pair. For 28 processes, there are 28x28=784
connections, including send-to-self connections.

You asked to see data on a 12-member subset of the MPI processes. Only point-
to-point messages that were sent from this subset are reported. The following
statistics represent 12x12=144 connections.

Here are statistics on the messages sent for each connection, reported on a
scale of 0-99 with 99 corresponding to 1 messages:

sender
     0  3  4  5  9 10 11 12 18 21 24 27
```

```
receiver
  0  _ 99 99 99 99 99 99 99 99 99 99 99
  1 99 99 99 99 99 99 99 99 99 99 99 99
  2 99 99 99 99 99 99 99 99 99 99 99 99
  3 99  _ 99 99 99 99 99 99 99 99 99 99
  4 99 99  _ 99 99 99 99 99 99 99 99 99
  5 99 99 99  _ 99 99 99 99 99 99 99 99
  6 99 99 99 99 99 99 99 99 99 99 99 99
  7 99 99 99 99 99 99 99 99 99 99 99 99
  8 99 99 99 99 99 99 99 99 99 99 99 99
  9 99 99 99 99  _ 99 99 99 99 99 99 99
 10 99 99 99 99 99  _ 99 99 99 99 99 99
 11 99 99 99 99 99 99  _ 99 99 99 99 99
 12 99 99 99 99 99 99 99  _ 99 99 99 99
 13 99 99 99 99 99 99 99 99 99 99 99 99
 14 99 99 99 99 99 99 99 99 99 99 99 99
 15 99 99 99 99 99 99 99 99 99 99 99 99
 16 99 99 99 99 99 99 99 99 99 99 99 99
 17 99 99 99 99 99 99 99 99 99 99 99 99
 18 99 99 99 99 99 99 99 99  _ 99 99 99
 19 99 99 99 99 99 99 99 99 99 99 99 99
 20 99 99 99 99 99 99 99 99 99 99 99 99
 21 99 99 99 99 99 99 99 99 99  _ 99 99
 22 99 99 99 99 99 99 99 99 99 99 99 99
 23 99 99 99 99 99 99 99 99 99 99 99 99
 24 99 99 99 99 99 99 99 99 99 99  _ 99
 25 99 99 99 99 99 99 99 99 99 99 99 99
 26 99 99 99 99 99 99 99 99 99 99 99 99
 27 99 99 99 99 99 99 99 99 99 99 99  _

Here are statistics on the bytes sent for each connection, reported on a scale
of 0-99 with 99 corresponding to 4 bytes:

    sender
     0  3  4  5  9 10 11 12 18 21 24 27
receiver
  0  _ 99 99 99 99 99 99 99 99 99 99 99
  1 99 99 99 99 99 99 99 99 99 99 99 99
  2 99 99 99 99 99 99 99 99 99 99 99 99
  3 99  _ 99 99 99 99 99 99 99 99 99 99
  4 99 99  _ 99 99 99 99 99 99 99 99 99
  5 99 99 99  _ 99 99 99 99 99 99 99 99
  6 99 99 99 99 99 99 99 99 99 99 99 99
  7 99 99 99 99 99 99 99 99 99 99 99 99
  8 99 99 99 99 99 99 99 99 99 99 99 99
  9 99 99 99 99  _ 99 99 99 99 99 99 99
 10 99 99 99 99 99  _ 99 99 99 99 99 99
 11 99 99 99 99 99 99  _ 99 99 99 99 99
 12 99 99 99 99 99 99 99  _ 99 99 99 99
```

```
 13 99 99 99 99 99 99 99 99 99 99 99 99
 14 99 99 99 99 99 99 99 99 99 99 99 99
 15 99 99 99 99 99 99 99 99 99 99 99 99
 16 99 99 99 99 99 99 99 99 99 99 99 99
 17 99 99 99 99 99 99 99 99 99 99 99 99
 18 99 99 99 99 99 99 99 99  _ 99 99 99
 19 99 99 99 99 99 99 99 99 99 99 99 99
 20 99 99 99 99 99 99 99 99 99 99 99 99
 21 99 99 99 99 99 99 99 99 99  _ 99 99
 22 99 99 99 99 99 99 99 99 99 99 99 99
 23 99 99 99 99 99 99 99 99 99 99 99 99
 24 99 99 99 99 99 99 99 99 99 99  _ 99
 25 99 99 99 99 99 99 99 99 99 99 99 99
 26 99 99 99 99 99 99 99 99 99 99 99 99
 27 99 99 99 99 99 99 99 99 99 99 99  _

The average length of point-to-point messages was 4 bytes per message.
```

Using `mpdump` to Convert Intermediate Binary Files to ASCII Files

The `mpdump` command converts each raw (unanalyzed) intermediate file that is generated by the MPI library into a readable ASCII file. The `mpdump` command produces files that have the `.txt` extension. For example, the following command creates a series of ASCII files in the current directory (the default) based on all the processes that are listed in the index file `mpprof.index.cre.14`:

```
% mpdump mpprof.index.cre.14
```

The `mpdump` Command Syntax

The syntax of the `mpdump` command is

```
mpdump [-h] [-V] [-S] [-p process-set] [-o output-directory]
index-file
```

where the following command options are available:

TABLE 8-2 Options to the `mpdump` Command

Option	Description
`-h`	Lists the optional arguments and exits.
`-V`	Displays the program version during start up.
`-S`	Uses the secure ssh and scp commands instead of rsh and rcp.
`-p`	Specifies a subset of the processes to be included in the `mpdump` output. The `-p` *process–set* option is entered as a comma-separated list of tuples. If the `-p` option is used with `-S` and/or `-g`, the operation will be performed only on the specified subset. If a process set encompasses the complete set of ranks, the `-p` option will have no effect. If the `-p` option is given a stride, the upper bound of the stride is omitted. For example, if a stride is specified as 1:4:2, the upper bound (4) is not included in the set. In this example, the resulting set is {1,3}.
`-o`	Specifies that the mpdump output files are to be written to the named directory. The named directory must already exist. If the specified *`output-directory`* does not exist, an error message will be issued. If `-o` is not specified, the output files will be written to the current directory.

A Sample `mpdump` File

The following command creates 28 ASCII text files that are based on the 28 files that were created in the previous section using the `mprun` command.

```
% mpdump mpprof.index.cre.39
```

Each of the 28 files contains profiling data about a process in the job. The contents of *one* of the files is shown below:

```
H: version="MPProf 1.0" np=28 rank=0 timeperiod=60.000000;
H: jobname="cre.39" pid=8399 date="Sat Oct 29 14:04:29 2005"
H: maxfilesize=52428800 hostname="hpc-smp6-0"
H: arg0="/opt/SUNWhpc/examples/mpi/connectivity"
V: variable=MPI_POLLALL actual=1 user=<unset>;
V: variable=MPI_PROCBIND actual=1 user=L;
V: variable=MPI_SPIN actual=1 user=1;
V: variable=MPI_CANONREDUCE actual=0 user=<unset>;
V: variable=MPI_OPTCOLL actual=1 user=<unset>;
V: variable=MPI_EAGERONLY actual=1 user=<unset>;
V: variable=MPI_COSCHED actual=2 user=<unset>;
```

```
V: variable=MPI_FLOWCONTROL actual=0 user=<unset>;
V: variable=MPI_FULLCONNINIT actual=0 user=<unset>;
V: variable=MPI_WARMUP actual=0 user=<unset>;
x:shm.2: H: version="MPProf 1.0" hostname="hpc-smp6-0";
x:shm.2: V: variable=MPI_SHM_CPOOLSIZE actual=24576 user=<unset>;
x:shm.2: V: variable=MPI_SHM_CYCLESIZE actual=8192 user=<unset>;
x:shm.2: V: variable=MPI_SHM_CYCLESTART actual=24576 user=<unset>;
x:shm.2: V: variable=MPI_SHM_NUMPOSTBOX actual=16 user=<unset>;
x:shm.2: V: variable=MPI_SHM_PIPESIZE actual=8192 user=<unset>;
x:shm.2: V: variable=MPI_SHM_PIPESTART actual=2048 user=<unset>;
x:shm.2: V: variable=MPI_SHM_SBPOOLSIZE actual=0 user=<unset>;
x:shm.2: V: variable=MPI_SHM_SHORTMSGSIZE actual=256 user=<unset>;
x:shm.2: V: variable=MPI_SHM_RENDVSIZE actual=24576 user=<unset>;
x:shm.2: V: variable=MPI_SHM_GBPOOLSIZE actual=20971520 user=<unset>;
h: start=0.000030 end=0.008181 snapshot=1;
c: routine=MPI_Barrier time=0.005588 bytessent=0 bytesrecv=0 calls=1;
c: routine=MPI_Comm_rank time=0.000003 bytessent=0 bytesrecv=0 calls=1;
c: routine=MPI_Comm_size time=0.000002 bytessent=0 bytesrecv=0 calls=1;
c: routine=MPI_Recv time=0.001103 bytessent=0 bytesrecv=108 calls=27;
c: routine=MPI_Send time=0.000248 bytessent=108 bytesrecv=0 calls=27;
p: destrank=1 bytessent=4 msgssent=1;
p: destrank=2 bytessent=4 msgssent=1;
p: destrank=3 bytessent=4 msgssent=1;
p: destrank=4 bytessent=4 msgssent=1;
p: destrank=5 bytessent=4 msgssent=1;
p: destrank=6 bytessent=4 msgssent=1;
p: destrank=7 bytessent=4 msgssent=1;
p: destrank=8 bytessent=4 msgssent=1;
p: destrank=9 bytessent=4 msgssent=1;
p: destrank=10 bytessent=4 msgssent=1;
p: destrank=11 bytessent=4 msgssent=1;
p: destrank=12 bytessent=4 msgssent=1;
p: destrank=13 bytessent=4 msgssent=1;
p: destrank=14 bytessent=4 msgssent=1;
p: destrank=15 bytessent=4 msgssent=1;
p: destrank=16 bytessent=4 msgssent=1;
p: destrank=17 bytessent=4 msgssent=1;
p: destrank=18 bytessent=4 msgssent=1;
p: destrank=19 bytessent=4 msgssent=1;
p: destrank=20 bytessent=4 msgssent=1;
p: destrank=21 bytessent=4 msgssent=1;
p: destrank=22 bytessent=4 msgssent=1;
p: destrank=23 bytessent=4 msgssent=1;
p: destrank=24 bytessent=4 msgssent=1;
p: destrank=25 bytessent=4 msgssent=1;
p: destrank=26 bytessent=4 msgssent=1;
p: destrank=27 bytessent=4 msgssent=1;
x:shm.2: alloc_mem=MPI_Alloc_mem no_memory_allocated=0;
e:
```

CHAPTER 9

Using the DTrace Utility With Sun MPI

This chapter discusses how to use the Solaris Dynamic Tracing utility (DTrace) with Sun MPI. DTrace is a comprehensive dynamic tracing utility that you can use to monitor the behavior of applications programs as well as the operating system itself. You can use DTrace on live production systems to understand those systems' behavior and to track down any problems that might be occurring.

The D language is the programming language used to create the source code for DTrace programs.

The material in this chapter assumes knowledge of the D language and how to use DTrace.

For more information about the D language and DTrace, refer to the *Solaris Dynamic Tracing Guide* (Part Number 817-6223). This guide is part of the Solaris 10 OS Software Developer Collection.

Solaris 10 OS documentation can be found on the web at the following location:

`http://www.sun.com/documentation`

Follow these links to the *Solaris Dynamic Tracing Guide:*

Solaris Operating Systems -> Solaris 10 -> Solaris 10 Software Developer Collection

---

Note – The sample program `mpicommleak` and other sample scripts are located at:

`/opt/SUNWhpc/examples/mpi/dtrace`

The following topics are covered in this chapter:

- `mprun` Privileges
- Running DTrace with MPI Programs
- Simple MPI Tracing

mprun Privileges

Before you run a program under DTrace, you need to make sure that you have the correct `mprun` privileges.

In order to run the script under `mprun`, make sure that you have `dtrace_proc` and `dtrace_user` privileges. Otherwise, DTrace will return the following error because it does not have sufficient privileges:

```
dtrace:  failed to initialize dtrace:  DTrace requires additional
privileges
```

To determine whether you have the appropriate privileges on the entire cluster, perform the following steps:

1. **Use your favorite text editor to create the following shell script, called `mpppriv.sh`:**

```
#!/bin/sh
# mpppriv.sh -  run ppriv under a shell so you can get the privileges
#               of the process that mprun creates
ppriv $$
```

2. **Type the following command:**

```
% mprun -np 0 -Ns mpppriv.sh
```

If the output of `ppriv` shows that the E privilege set has the `dtrace` privileges, then you will be able to run `dtrace` under `mprun` (see the two examples below) Otherwise, you will need to adjust your system to get `dtrace` access.

The following example shows the output from `ppriv` if the correct user privileges have not been set:

```
% ppriv $$
4084:  -csh
flags = <none>
       E:  basic
       I:  basic
       P:  basic
       L:  all
```

This example shows `ppriv` output when the privileges have been set:

```
% ppriv $$
2075:  tcsh
flags = <none>
       E:basic,dtrace_proc,dtrace_user
       I:basic,dtrace_proc,dtrace_user
       P:basic,dtrace_proc,dtrace_user
       L:  all
```

Note – To update your privileges, ask your system administrator to add the `dtrace_user` and `dtrace_proc` privileges to your account in the `/etc/user_attr` file.

After the privileges have been changed, you can use the `ppriv` command to execute the `dtrace` commands under `mprun`.

Running DTrace with MPI Programs

There are two ways to use Dynamic Tracing with MPI programs:

- Run the program directly under DTrace, or
- Attach DTrace to a running MPI program

Running an MPI Program Under DTrace

For illustration purposes, assume you have a program named `mpiapp`. To trace the program `mpiapp` using the `mpitrace.d` script, type the following command:

```
% mprun -np 4 dtrace -s mpitrace.d -c mpiapp
```

The advantage of tracing an MPI program in this way is that all the processes in the job will be traced from the beginning. This method is probably most useful in doing performance measurements, when you need to start at the beginning of an application and you need all the processes in a job to participate in collecting data.

This approach also has some disadvantages. One disadvantage of running a program like the one in the above example is that all the tracing output for all four processes is directed to standard output (`stdout`). One way around this problem is to create a script similar to the following:

```
#!/bin/sh
# partrace.sh - a helper script to dtrace Sun MPI jobs from the
#          start of the job.
dtrace -s $1 -c $2 -o $2.$MP_JOBID.$SUNHPC_PROC_RANK.trace
```

To trace a parallel program and get separate trace files, type the following command to run the `partrace.sh` shell script:

```
% mprun -np 4 partrace.sh mpitrace.d mpiapp
```

This will run `mpiapp` under dtrace using the `mpitrace.d` script. The script saves the trace output for each process in a job under a separate file name, based on the job ID and rank of the process.

Attaching to MPI Processes

The second way to use `dtrace` with Sun MPI is to attach `dtrace` to a running process. Perform the following procedure:

1. **Type the following command to get the process ID (PID) of the running process and the nodes on which it is running.**

```
% mpps -p
      JOBNAME NPROC      UID         STATE   AOUT
      cre.1       2      joeuser             RUN         mpiapp
             RANK        PID         STATE   NODE
             0           6390        RUN     mynode
             1           6391        RUN     mynode
```

2. **Decide which rank you want to use to attach** `dtrace`**, and then log in to the node that contains the rank you want to use (in this example, rank 1 for the job** `cre.1`**). In the example, you would log in to the node** `mynode`**.**

3. **Type the following command to attach to the rank 1 process (identified by its process ID, which is 6391 in the example) and run the DTrace script** `mpitrace.d`**:**

```
% dtrace -p 6391 -s mpitrace.d
```

Simple MPI Tracing

DTrace enables you to easily trace programs. When used in conjunction with MPI and the more than 200 functions defined in the MPI standard, DTrace provides an easy way to determine which functions might be in error during the debugging process, or those functions which might be of interest. After you determine the function showing the error, it is easy to locate the desired job, process, and rank on which to run your scripts. As demonstrated above, DTrace allows you to perform these determinations while the program is running

Although the MPI standard provides the MPI profiling interface, using DTrace does provide a number of advantages. The advantages of using DTrace include the following:

- The PMPI interface requires you to restart a job every time you make changes to the interposing library.
- DTrace allows you to define probes that let you capture tracing information on MPI without having to code the specific details for each function you want to capture.
- DTrace's scripting language D has several built-in functions that help in debugging problematic programs.

The following example shows a simple script that traces the entry and exit into all the MPI API calls.

```
mpitrace.d:
pid$target:libmpi:MPI_*:entry
{
printf("Entered %s...", probefunc);
}

pid$target:libmpi:MPI_*:return
{
printf("exiting, return value = %d\n", arg1);
}
```

When you use this example script to attach DTrace to a job that performs `send` and `recv` operations, the output looks similar to the following:

```
% dtrace -q -p 6391 -s mpitrace.d
Entered MPI_Send...exiting, return value = 0
Entered MPI_Recv...exiting, return value = 0
Entered MPI_Send...exiting, return value = 0
Entered MPI_Recv...exiting, return value = 0
Entered MPI_Send...exiting, return value = 0 ...
```

You can easily modify the `mpitrace.d` script to include an argument list. The resulting output resembles `truss` output. For example:

```
mpitruss.d:
pid$target:libmpi:MPI_Send:entry,
pid$target:libmpi:MPI_*send:entry,
pid$target:libmpi:MPI_Recv:entry,
pid$target:libmpi:MPI_*recv:entry
{
printf("%s(0x%x, %d, 0x%x, %d, %d, 0x%x)",probefunc, arg0, arg1,
arg2, arg3, arg4, arg5);
}
pid$target:libmpi:MPI_Send:return,
pid$target:libmpi:MPI_*send:return,
pid$target:libmpi:MPI_Recv:return,
pid$target:libmpi:MPI_*recv:return
{
printf("\t\t = %d\n", arg1);
}
```

The `mpitruss.d` script shows how you can specify wildcard names to match the functions. Both probes will match all send and receive type function calls in the MPI library. The first probe shows the usage of the built-in `arg` variables to print out the `arglist` of the function being traced.

Take care when wildcarding the entrypoint and the formatting argument output, because you could end up printing either too many arguments, or not enough arguments, for certain functions. For example, in the above case, the `MPI_Irecv` and `MPI_Isend` functions will not have their Request handle parameters printed out.

The following example shows a sample output of the `mpitruss.d` script:

```
% dtrace -q -p 6391 -s mpitruss.d
MPI_Send(0x80470b0, 1, 0x8060f48, 0, 1,0x8060d48) = 0
MPI_Recv(0x80470a8, 1, 0x8060f48, 0, 0, 0x8060d48) = 0
MPI_Send(0x80470b0, 1, 0x8060f48, 0, 1, 0x8060d48) = 0
MPI_Recv(0x80470a8, 1,0x8060f48, 0, 0, 0x8060d48) = 0 ...
```

Tracking Down Resource Leaks

One of the biggest issues with programming is the unintentional leaking of resources (such as memory). With MPI, tracking and repairing resource leaks can be somewhat more challenging because the objects being leaked are in the middleware, and thus are not easily detected by the use of memory checkers.

DTrace helps with debugging such problems using variables, the profile provider, and a callstack function. The `mpicommcheck.d` script (shown in the example below) probes for all the the MPI communicator calls that allocate and deallocate communicators, and keeps track of the stack each time the function is called. Every 10 seconds the script dumps out the current count of MPI communicator calls and the total calls for the allocation and deallocation of communicators. When the `dtrace` session ends (usually by typing Ctrl-C, if you attached to a running MPI program), the script will print out the totals and all the different stack traces, as well as the number of times those stack traces were reached.

In order to perform these tasks, the script uses DTrace features such as variables, associative arrays, built-in functions (`count`, `ustack`) and the predefined variable `probefunc`.

The following example shows the `mpicommcheck.d` script.

```
mpicommcheck.d:
BEGIN
{
  allocations = 0;
  deallocations = 0;
  prcnt = 0;
}

pid$target:libmpi:MPI_Comm_create:entry,
pid$target:libmpi:MPI_Comm_dup:entry,
pid$target:libmpi:MPI_Comm_split:entry
{
  ++allocations;
  @counts[probefunc] = count();
  @stacks[ustack()] = count();
}

pid$target:libmpi:MPI_Comm_free:entry
{
  ++deallocations;
  @counts[probefunc] = count();
  @stacks[ustack()] = count();
}

profile:::tick-1sec
/++prcnt > 10/
{
  printf("============================================================
==============");
  printa(@counts);
  printf("Communicator Allocations = %d \n", allocations);
  printf("Communicator Deallocations = %d\n", deallocations);
  prcnt = 0;
}

END
{
  printf("Communicator Allocations = %d, Communicator
Deallocations = %d\n",
       allocations, deallocations);
}
```

This script attaches `dtrace` to a suspect section of code in your program (that is, a section of code that might contain a resource leak). If, during the process of running the script, you see that the printed totals for allocations and deallocations are

starting to steadily diverge, you might have a resource leak. Depending on how your program is designed, it might take some time and observation of the allocation/deallocation totals in order to definitively determine that the code contains a resource leak. Once you do determine that a resource leak is definitely occurring, you can type Ctrl-C to break out of the `dtrace` session. Next, using the stack traces dumped, you can try to determine where the issue might be occurring.

The following example shows code containing a resource leak, and the output that is displayed using the `mpicommcheck.d` script.

The sample MPI program containing the resource leak is called `mpicommleak`. This program performs three MPI_Comm_dup operations and two MPI_Comm_free operations. The program thus "leaks" one communicator operation with each iteration of a loop.

When you attach `dtrace` to `mpicommleak` using the `mpicommcheck.d` script above, you will see a 10-second periodic output. This output shows that the count of the allocated communicators is growing faster than the count of deallocations.

When you finally end the `dtrace` session by typing Ctrl-C, the session will have output a total of five stack traces, showing the distinct three MPI_Comm_dup and two MPI_Comm_free call stacks, as well as the number of times each call stack was encountered.

For example:

```
% dtrace -q -p 6581 -s mpicommcheck.d
================================================================
  MPI_Comm_free                                                 4
  MPI_Comm_dup                                                  6
Communicator Allocations = 6
Communicator Deallocations = 4
================================================================
  MPI_Comm_free                                                 8
  MPI_Comm_dup                                                  12
Communicator Allocations = 12
Communicator Deallocations = 8
================================================================
  MPI_Comm_free                                                 12
  MPI_Comm_dup                                                  18
Communicator Allocations = 18
Communicator Deallocations = 12
^C
Communicator Allocations = 21, Communicator Deallocations = 14
        libmpi.so.1`MPI_Comm_dup
        mpicommleak`allocate_comms+0x1e
        mpicommleak`main+0x5b
        mpicommleak`0x805091a
        7

        libmpi.so.1`MPI_Comm_dup
        mpicommleak`allocate_comms+0x30
        mpicommleak`main+0x5b
        mpicommleak`0x805091a
        7

        libmpi.so.1`MPI_Comm_dup
        mpicommleak`allocate_comms+0x42
        mpicommleak`main+0x5b
        mpicommleak`0x805091a
        7

        libmpi.so.1`MPI_Comm_free
        mpicommleak`deallocate_comms+0x19
        mpicommleak`main+0x6a
        mpicommleak`0x805091a
        7

        libmpi.so.1`MPI_Comm_free
        mpicommleak`deallocate_comms+0x26
        mpicommleak`main+0x6a
        mpicommleak`0x805091a
        7
```

CHAPTER 10

Displaying Information With `mpinfo`

What You Can Do

Task	Option
How to display information about published names	`-T`
How to display information about any cluster	`-c`
How to display information about the current cluster	`-C`
How to display information about individual partitions	`-p`
How to display information about all partitions	`-P`
How to display information about individual nodes	`-n`
How to display information about all nodes	`-N`
How to display an online list of valid attributes	`-lc -lp -ln`
How to restrict output to individual attributes	`-A`
How to display information in verbose mode	`-v`

▼ To Display Information About Published Names (-T)

Use the -T option:

```
% mpinfo -T
```

The name, associated port value, and the *jid* of the publishing job are displayed. For example:

```
% mpinfo -T
JID  NAME         PORT
14   bachelor     xxx.xxx.xxx.xxx:48944
14   biker        xxx.xxx.xxx.xxx:48944
14   centralia    xxx.xxx.xxx.xxx:48944
14   rockstar     xxx.xxx.xxx.xxx:48944
18   freddie      xxx.xxx.xxx.xxx:501
18   lackluster   xxx.xxx.xxx.xxx:503
18   bennie       xxx.xxx.xxx.xxx:505
18   stellar      xxx.xxx.xxx.xxx:507
```

The fields in the output are described in TABLE 10-1.

Include the -A option to restrict the output to a particular attribute or set of attributes. (See "To Restrict Output to Individual Attributes (-A)" on page 109.)

▼ To Display Information About Any Cluster (-c)

Use the -c option:

```
% mpinfo -c [cluster] -C | -P | - N
```

If you do not enter a *cluster*, the command uses the cluster named by the SUNHPC_CLUSTER environment variable. If that environment variable has not been set, be sure to manually enter a *cluster*, or the command fails.

Use one of the three options -C, -P, or -N to indicate the type of information you want to display (cluster-level, partition, or node). Use only one option at a time. For example:

```
% mpinfo -c hpc-cluster-0 -C
NAME          ADMINISTRATOR DEF_INTER_PART
hpc-cluster-0 -             all

% mpinfo -c hpc-cluster-0 -P
NAME NODES: Tot(cpu) Enb(cpu) Onl(cpu) ENA LOG MP
all         1(28)    1(28)    1(28)    yes yes yes

% mpinfo -c hpc-cluster-0 -N
NAME  UP PARTITION OS    OSREL NCPU FMEM   FSWP LOAD1 LOAD5 LOAD15
node0 y  all       SunOS 5.10 10   748.07 1459 10.54 10.62 10.66
node1 y  all       SunOS 5.10 10   811.63 1492 10.51 10.53 10.55
node2 y  all       SunOS 5.10 10   715.10 1432 10.87 10.88 10.91
node3 y  all       SunOS 5.10 10   837.91 1514 10.06 10.24 10.31
```

The fields in the output are described in TABLE 10-1.

Include the -A option to restrict the output to a particular attribute or set of attributes. (See "To Restrict Output to Individual Attributes (-A)" on page 109.)

▼ To Display Information About the Current Cluster (-C)

Use the -C option (upper case):

```
% mpinfo -C
```

This option is a shortcut for a common use of the -c option:

```
% mpinfo -C
NAME          ADMINISTRATOR DEF_INTER_PART
hpc-cluster-0 -             all

% mpinfo -c hpc-cluster-0 -C
NAME          ADMINISTRATOR DEF_INTER_PART
hpc-cluster-0 -             all
```

The fields in the output are described in TABLE 10-1.

Include the -A option to restrict the output to a particular attribute or set of attributes. (See "To Restrict Output to Individual Attributes (-A)" on page 109.)

▼ To Display Information About Individual Partitions (-p)

Use the -p option:

```
% mpinfo -p partition–name[,partition–name ...]
```

Separate multiple partition names with a comma. You can also enclose the set of partition names in quotation marks.

For example:

```
% mpinfo -p part1,part2
NAME          NODES: Tot(cpu) Enb(cpu) Onl(cpu) ENA LOG MP
part1                 1(  4)   1(  4)   1(  4)  no  yes yes
part2                 1(  4)   1(  4)   1(  4)  yes yes yes
```

The fields of the output are described in TABLE 10-1.

Include the -A option to restrict the output to a particular attribute or set of attributes. (See "To Restrict Output to Individual Attributes (-A)" on page 109.)

▼ To Display Information About All Partitions (-P)

Use the -P option (upper case):

```
% mpinfo -P
```

For example:

```
% 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 fields of the output are described in TABLE 10-1.

Include the -A option to restrict the output to a particular attribute or set of attributes. (See "To Restrict Output to Individual Attributes (-A)" on page 109.)

▼ To Display Information About Individual Nodes (-n)

Use the -n option:

```
% mpinfo -n node–name[,node–name...]
```

When listing multiple node names, separate the names with commas but no spaces. For example:

```
% mpinfo -n node1,node2
NAME UP PARTITION OS    OSREL NCPU FMEM FSWP LOAD1 LOAD5 LOAD15
node1 y  all       SunOS 5.10  2    2252 7935 0.04  0.03  0.03
node2 y  all       SunOS 5.10  2    2084 7743 0.03  0.02  0.02
```

The fields in the output are described in TABLE 10-1.

Include the -A option to restrict the output to a particular attribute or set of attributes. (See "To Restrict Output to Individual Attributes (-A)" on page 109.)

▼ To Display Information About All Nodes (-N)

Use the -N option (upper case).

```
% mpinfo -N
```

For example:

```
% mpinfo -N
NAME  UP PARTITION OS    OSREL NCPU FMEM   FSWP    LOAD1 LOAD5 LOAD1
node0 y  p0        SunOS 5.10   1    0.89   158.34  0.09  0.11  0.13
node1 y  p0        SunOS 5.10   1    31.41  276.12  0.00  0.01  0.01
node2 y  p1        SunOS 5.10   1    25.59  279.77  0.00  0.00  0.01
node3 y  p1        SunOS 5.10   1    25.40  279.88  0.00  0.00  0.01
```

The fields in the output are described in TABLE 10-1.

Include the `-A` option to restrict the output to a particular attribute or set of attributes. (See "To Restrict Output to Individual Attributes (`-A`)" on page 109.)

▼ To Display an Online List of Valid Attributes (`-lc`, `-lp`, `-ln`)

Use the `-lc`, `-lp`, or `-ln` options for clusters, partitions, or nodes:

```
% mpinfo -lc
% mpinfo -lp
% mpinfo -ln
```

For example:

```
% mpinfo -lc
  name          cluster name (NAME)
  admin         cluster administrator (ADMINISTRATOR)
  definter      default interactive partition (DEF_INTER_PART)

% mpinfo -lp
  name          partition name (NAME)
  enabled       partition state (ENA)
  nodes         node count (NODES: Tot(cpu) Enb(cpu) Onl(cpu))
  maxt          max total procs (MAXT)
  login         logins allowed (LOG)
  mp            mp jobs allowed (MP)

% mpinfo -ln
  cpu_idle        idle        cpu idle time (%) (IDLE)
  cpu_iowait      iowait      cpu iowait time (%) (IWAIT)
  cpu_kernel      kernel      cpu kernel time (%) (KERNL)
  cpu_type        cpu         cpu architecture (CPU)
  ...
```

▼ To Restrict Output to Individual Attributes (-A)

Add the -A option to other mpinfo options to restrict the output to specific attributes of a node, partition, or cluster:

```
% mpinfo -p partition -A attribute[,attribute...]
% mpinfo -P -A attribute[,attribute...]
% mpinfo -n node -A attribute[,attribute...]
% mpinfo -N -A attribute[,attribute...]
% mpinfo -c cluster [-C | -P | -N] -A attribute[,attribute...]
% mpinfo -C partition -A attribute[,attribute...]
```

Separate multiple partition attributes with commas but no spaces. For a list of valid attributes, see TABLE 10-1.

This example begins by showing the full set of node attributes displayed when you identify the object (in this case with the -N option), but leave out the -A option. Then it shows how adding the -A option restricts the list to a subset of the information:

```
% mpinfo -N
NAME  UP PARTITION OS    OSREL NCPU FMEM   FSWP LOAD1 LOAD5 LOAD15
node0 y  all       SunOS 5.10 10   750.19 1527 10.52 10.66 10.70
node1 y  all       SunOS 5.10 10   816.07 1576 10.55 10.59 10.61
node2 y  all       SunOS 5.10 10   721.84 1524 10.91 10.95 10.96
node3 y  all       SunOS 5.10 10   840.41 1596 10.42 10.42 10.40

% mpinfo -N -A name
NAME
node0
node1
node2
node3
```

TABLE 10-1 Attributes Displayed by -A option to mpinfo

Object	Attribute	Description
Partition	NAME	Name of the partition
	NODES	Information about the nodes in the partition: Tot – total number of nodes Enb – number that are enabled Onl – number currently online ENA – whether the partition is enabled (YES/NO) LOG – whether the node accepts logins (YES/NO) MP – whether the node accepts multinode jobs (YES/NO)
	MAXT	Maximum number of simultaneously running processes allowed on each node of the partition
Cluster	NAME	Name of the cluster (host name of the master node)
	ADMINISTRATOR	Name of the cluster's administrator
	DEF_INTER_PART	Default interactive partition
Node	cpu_idle	Percent of time CPU is idle (IDLE).
	cpu_iowait	Percent of time CPU spends waiting for I/O (IWAIT).
	cpu_kernel	Percent of time CPU spends in kernel (KERNL).
	cpu_type	CPU architecture (CPU).

TABLE 10-1 Attributes Displayed by `-A` option to `mpinfo` *(Continued)*

Object	Attribute	Description
Node (*continued*)	`cpu_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`	Hardware manufacturer (`MANUFACTURER`).
	`mem_free`	Node's available RAM (in Mbytes) (`FMEM`).
	`mem_total`	Node's total physical memory (in Mbytes) (`MEM`).
	`name`	Name of the node (`NAME`).
	`ncpus`	Number of CPU modules in the node (`NCPU`).
	`os_arch_kernel`	Node's kernel architecture (`MACH`).
	`os_max_proc`	Maximum number of processes allowed on the node (note that this is *all* processes, including cluster daemons) (`MPROC`).
	`os_name`	Name of the operating system running on the node (`OS`).
	`os_release`	Operating system's release number (`OSREL`).
	`os_release_maj`	The major number of the operating system release number (`MAJ`).
	`os_release_min`	The minor number of the operating system release number (`MIN`).
	`os_version`	Operating system's version (`OSVER`).
	`partition`	The partition of which the node is a member (`PARTITION`).
	`serial_number`	Hardware serial number (`SERIAL`).
	`swap_free`	Node's available swap space (in Mbytes) (`FSWP`).
	`swap_total`	Node's total swap space (in Mbytes) (`SWAP`).

▼ To Display Information in Verbose Mode (-v)

Add the -v option to any other option to display its information in verbose mode:

```
% mpinfo -p partition -v
% mpinfo -P -v
% mpinfo -n node -v
% mpinfo -N -v
% mpinfo -c cluster [-C | -P | -N] -v
% mpinfo -C partition -v
```

Verbose mode displays a little more information than standard mode, and makes it easier to read. This example shows how the information is displayed first without, and then with the verbose mode:

```
% mpinfo -N
NAME  UP PARTITION OS    OSREL NCPU FMEM   FSWP LOAD1 LOAD5 LOAD15
node0 y  all       SunOS 5.10 10   839.30 1610 10.18 10.57 10.65
node1 y  all       SunOS 5.10 10   900.45 1646 10.12 10.47 10.54
node2 y  all       SunOS 5.10 10   802.66 1592 10.42 10.78 10.84
node3 y  all       SunOS 5.10 10   927.55 1676 10.11 10.48 10.48

% mpinfo -N -v

Node "node0":
  LPM Interfaces: shm,tcp
  State:  enabled & online
  partition: "all"

  os: SunOS 5.10 (Generic_108528-07)
  arch: sun4u,  cpu: sparc,  ncpus: 10
  manufacturer: Sun_Microsystems,  serial no: 809deb49
  memory: 1280.000M (775.727M free),
  swap: 1932.539M (1579.609M free)
  isalist: sparcv9+vis sparcv9 sparcv8plus+vis sparcv8plus
  sparcv8 sparcv8-fsmuld sparcv7 sparc

  load averages: 10.53, 10.57, 10.64
  cpu states:  0.00% idle, 46.31% user, 53.69% kernel, 0.00% iowait

  local attributes:

Node "node1":
  LPM Interfaces: shm,tcp
  State:  enabled & online
  partition: "all"
.
.
.
```

Command Reference (mpinfo)

TABLE 10-2 Options for mpinfo

Command	Description
-T	Display name publishing information
-c	Display cluster-level, partition, or node information about any cluster
-C	Display cluster-level information about the current cluster; equivalent to mpinfo -c *cluster–name* -C
-p	Display information about individual partitions
-P	Displays information about all partitions in the cluster
-n	Displays information about individual nodes
-N	Displays information about all nodes in the cluster
-lc	List the cluster attributes that can be displayed by -A
-lp	List the partition attributes that can be displayed by -A
-ln	List the node attributes that can be displayed by -A
-A	Restrict the display of cluster, partition, or node information to individual attributes—must combine with one of these options: -c -C -p -P -n -N
-v	Display information in verbose mode—must combine with one of these options: -c -C -p -P -n -N

APPENDIX **A**

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 A-1 (for non-I/O MPI) and TABLE A-2 (for MPI I/O), and are also available on the `MPI` man page.

Three predefined error handlers are available in Sun MPI:

- `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 A-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.
`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.

TABLE A-1 Sun MPI Standard Error Classes *(Continued)*

Error Code	Value	Meaning
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_WIN	46	Invalid window.
MPI_ERR_BASE	47	Invalid base.
MPI_ERR_SIZE	48	Invalid size.
MPI_ERR_DISP	49	Invalid displacement.
MPI_ERR_LOCKTYPE	50	Invalid locktype.
MPI_ERR_ASSERT	51	Invalid assert.
MPI_ERR_RMA_CONFLICT	52	Conflicting accesses to window.
MPI_ERR_RMA_SYNC	53	Erroneous RMA synchronization.
MPI_ERR_NO_MEM	54	Memory exhauste.
MPI_ERR_LASTCODE	55	Last error code.

MPI I/O message are listed separately, in TABLE A-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 A-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 if no file is currently open. Or, you can use the same routine to change a specific file's error handler.

TABLE A-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.
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.

TABLE A-2 Sun MPI I/O Error Classes *(Continued)*

Error Class	Value	Meaning
MPI_ERR_INFO_NOKEY	39	No such key.
MPI_ERR_INFO_VALUE	40	Illegal info value.
MPI_ERR_LASTCODE	55	Last error code.

Exceeding the File Descriptor Limit

If your application attempts to open a file descriptor when the maximum limit of open file descriptors has been reached, the job will fail and display the following message:

```
Too many open file descriptors
```

Should this occur, increase the value of the file descriptor hard limit before starting your job again.

If you are logged in to a C shell as superuser, you can determine the current hard limit value via the `limit` function, as follows:

```
# limit -h descriptors
```

If you are logged in to a Bourne shell as superuser, use the `ulimit` function.

```
# ulimit -Hn
```

Each function returns the file descriptor hard limit that was in effect. Once you know what the previous hard limit was, you can estimate what the new hard limit value should be and set it accordingly.

From a C shell, use the `limit` command to set the new value in the `.login` file.

```
# limit -h descriptors limit
```

From a Bourne shell, use the `ulimit` command to set the new value in the `.profile` file.

```
# ulimit -Hn limit
```

In each case, *limit* is the value of the new hard limit.

Alternatively, you can determine whether the file descriptor hard limit is anything other than the default by looking in the `/etc/system` file to see whether the `rlim_fd_max` parameter has been set to a nondefault value. If not, the file descriptor hard limit will be 1024. To change the hard limit in a Solaris environment, simply add the following line to the `/etc/system` file:

```
set rlim_fd_max=limit
```

Again, *limit* is the value of the new file descriptor hard limit.

Exceeding the TCP Port Limit

If you are running a large (highly parallel), communication-intensive MPI job on a Sun HPC cluster that includes both of the following conditions,

- TCP/IP as the only interconnect medium
- A node that has more than 32 CPUs

the number of TCP ports may be too limited. If the MPI job attempts to access a TCP port when no more are available, the job will fail and print the following message:

```
low level communications error: Cannot assign requested address
```

Most likely, this occurs only when the job is running on the configuration described above *and* one of the following conditions exists:

- `MPI_FULLCONNINIT` is set.
- `MPI_Alltoall` is used.
- The application includes its own all-to-all code.

Other activity on the cluster, such as file I/O or other MPI jobs, will increase the chance of this occurring.

You can avoid exceeding the TCP port limit by taking one or more of the following steps:

- Configure the node with more than 32 nodes into two or more domains. From the TCP perspective, each domain will be seen as a separate node with its own supply of TCP ports.
- Reconfigure the cluster to exclude the node with more than 32 CPUs.
- Avoid running multiple MPI jobs or other tasks that would compete for available TCP ports.
- If two large MPI jobs must run on the same cluster, wait a few minutes between the jobs to give the OS time to reclaim the ports created for the previous job.
- If the application does not include any all-to-all operations, use the default lazy connections mode instead of `MPI_FULLCONNINIT`.
- If the application contains any all-to-all operations, either `MPI_Alltoall` or custom code, use a non-TCP network technology.

Index

E

F

G

H

J

K

L

M

N

O

P

R

S

T

U

V

W