

HP AlphaServer SC User Guide

Quadrics Ltd. **Document Version 12 - Nov 14th 2003 - AA-RLB1E-TE**

The information supplied in this document is believed to be correct at the time of publication. No liability is assumed for its use or for the infringements of the rights of others resulting from its use. No license or other rights are granted in respect of any rights owned by any of the organizations mentioned herein.

This document may not be copied, in whole or in part, without the prior written consent of Quadrics Ltd.

Copyright 1998,1999,2000,2001,2002,2003 Quadrics Ltd.

The specifications listed in this document are subject to change without notice.

Intel is the registered trademark of Intel Corporation.

AMD is the trademark of Advanced Micro Devices, Inc.

UNIX® is a registered trademark of The Open Group.

TotalView and Etnus are registered trademarks of Etnus LLC.

All other product names mentioned herein may be trademarks of their respective companies.

Some information in Chapter 4 of this document is based on HP documentation, which includes the following copyright notice: Copyright 2000-2003 Hewlett-Packard Development Company, L.P.

The Quadrics Ltd. (Quadrics) web site can be found at:

```
http://www.quadrics.com/
```

Quadrics' address is:

One Bridewell Street
Bristol
BS1 2AA
UK

Tel: +44-(0)117-9075375
Fax: +44-(0)117-9075395

Circulation Control: None

Document Revision History

Revision	Date	Author	Remarks
3	Oct 1999	DR	RMS 2.33 Release
4	Oct 1999	DR	RMS 2.36 Release
9	June 2001	DR	AlphaServer SC V2 Product Release
10	June 2001	RMC	Linux Product Release
11	Sept 2002	DR, RMC	HP AlphaServer SC V2.5 Product Release
12	Nov 2003	DR	HP AlphaServer SC V2.6 Product Release

Contents

1 Introduction **1-1**

1.1 Scope of Manual 1-1
1.2 Audience 1-1
1.3 Using this Manual 1-1
1.4 Related Information 1-2
1.5 Location of Online Documentation 1-2
1.6 Reader's Comments 1-3
1.7 Conventions 1-3

2 Getting Started **2-1**

2.1 Introduction 2-1
2.2 About RMS 2-1
2.2.1 The User Interface 2-2
2.2.2 The RMS Daemons 2-3
2.3 Getting Started 2-4
2.3.1 Logging In 2-4
2.3.2 Getting Help 2-5
2.4 Running a Parallel Program 2-6
2.4.1 Partitions 2-6
2.4.2 Priorities and Projects 2-7

3 RMS User Commands **3-1**

3.1 Introduction . . . 3-1
3.2 More on Parallel Programs . . . 3-1
3.3 Getting Resource Information with rinfo . . . 3-3
3.3.1 Node Name Format . . . 3-5
3.3.2 Command Line Options . . . 3-5
3.3.3 Querying the Machine's Users . . . 3-5
3.3.4 Checking Quotas . . . 3-6
3.3.5 Viewing Configuration Details . . . 3-7
3.4 Running Programs with prun . . . 3-7
3.4.1 `prun` Command Line Options . . . 3-8
3.4.2 Selecting a Partition . . . 3-8
3.4.3 Specifying Processes and Nodes . . . 3-8
3.4.4 Input and Output . . . 3-11
3.4.5 RMS Environment Variables . . . 3-14
3.4.6 Memory Limits . . . 3-14
3.4.7 Examining Jobs and Resources . . . 3-16
3.4.8 Program Termination . . . 3-16
3.4.9 Core Files . . . 3-18
3.4.10 Stats Collection . . . 3-19
3.4.11 Propagating DCE Credentials . . . 3-20
3.4.12 Common Problems . . . 3-21
3.5 Allocating Resources with allocate . . . 3-23
3.5.1 Command Line Options . . . 3-24
3.5.2 Allocating Resources to an Interactive Shell . . . 3-24
3.5.3 Verifying Resource Allocation to a Shell . . . 3-25
3.5.4 Allocating Resources to a Shell Script . . . 3-26
3.6 Running a Sequential Program with rmsexec . . . 3-27
3.6.1 Command Line Options . . . 3-27
3.6.2 Selecting a Node . . . 3-27
3.6.3 Defining Load . . . 3-27

4 MPI and Shmem Programming **4-1**

4.1 Introduction . . . 4-1
4.2 MPI Overview . . . 4-2
4.2.1 Introduction to MPI . . . 4-2
4.2.2 Compiling, Linking and Running MPI Programs . . . 4-3
4.2.3 MPI Examples . . . 4-4
4.2.4 Further Information on MPI . . . 4-5
4.3 Shmem Overview . . . 4-5
4.3.1 Introduction to Shmem . . . 4-6
4.3.2 Compiling, Linking and Running Shmem Programs . . . 4-7
4.3.3 Shmem Examples . . . 4-8
4.3.4 Further Information on Shmem . . . 4-8
4.4 Using TotalView . . . 4-8
4.4.1 Running a Parallel Job under TotalView Control . . . 4-9
4.4.2 Attaching to an Executing Parallel Job . . . 4-10
4.4.3 Restarting a Parallel Job . . . 4-10
4.4.4 TotalView Bulk Launch Support . . . 4-11
4.4.5 Problems and Limitations . . . 4-11
4.5 Using Vampir . . . 4-12
4.5.1 Preparing to Use Vampir . . . 4-12
4.5.2 Linking and Tracing a Program . . . 4-12
4.6 MPI Example . . . 4-13
4.6.1 MPI Functions . . . 4-13
4.6.2 Command Line Interface . . . 4-13
4.6.3 Program Output . . . 4-14
4.6.4 Header Files and Variables . . . 4-15
4.6.5 Argument Checking . . . 4-16
4.6.6 Initialization . . . 4-18
4.6.7 Establishing the Peer Group . . . 4-19
4.6.8 Sending Messages . . . 4-20
4.6.9 Subsidiary Functions . . . 4-23

4.6.10 Compiling and Running the Program 4-23
4.7 Shmem Example 4-24
4.7.1 Shmem Functions 4-25
4.7.2 Command Line Interface 4-25
4.7.3 Program Output 4-26
4.7.4 Header Files and Variables 4-26
4.7.5 Argument Checking 4-28
4.7.6 Initialization 4-30
4.7.7 Establishing the Peer Group 4-31
4.7.8 Data Transfer 4-32
4.7.9 Subsidiary Functions 4-34
4.7.10 Compiling and Running the Program 4-35

A RMS Commands **A-1**
A.1 Overview A-1
allocate(1) A-2
prun(1) A-7
rinfo(1) A-22
rmsexec(1) A-29
rmsquery(1) A-31

B Submitting LSF Jobs **B-1**
B.1 Overview of LSF/RMS Interaction B-1
B.2 Setting Up Your Environment B-2
B.3 The bsub command B-2
B.4 Running Pre-Execution Programs B-4
B.5 LSF Topology Scheduler B-5
B.5.1 Allocation Type B-6
B.5.2 Topology B-7
B.5.3 Flags B-8
B.6 RMS Allocation Options Supported by LSF B-8
B.7 How LSF Calculates Load Indices B-9

B.8 Monitoring and Controlling LSF Jobs B-10
B.9 Viewing Nodes Allocated To Your Job B-11
B.9.1 Running Jobs (`bjobs -l`) B-11
B.9.2 Finished Jobs (`bhist -l`) B-11
B.9.3 Job Accounting Information (`bacct -l`) B-12
B.10 LSF Job Termination Reason B-13

C MPI Library Routines **C-1**
C.1 Overview C-1
C.2 MPI-2 One-Sided Support C-1
C.3 Optimized Collectives C-2
C.4 MPI-IO Support C-2
C.5 MPI Command Line Arguments C-3
C.6 Support for `-i8` and `-r8` Fortran compiler flags C-5

D Shmem Library Routines **D-1**
D.1 Overview D-1
D.1.1 Initialization Routines D-2
D.1.2 Cache Routines D-2
D.1.3 Access Routines D-2
D.1.4 Synchronization Routines D-2
D.1.5 Put and Get Routines D-3
D.1.6 Non-blocking Put and Get Routines D-3
D.1.7 Strided or Indexed Put and Get Routines D-4
D.1.8 Collective Communications Routines D-4
D.1.9 Atomic Routines D-5
D.1.10 Synchronization Routines D-6
D.1.11 Remote Locking D-7

E Elan Library Environment Variables **E-1**
E.1 Important MPI and Shmem Settings E-1
E.2 Troubleshooting E-4

E.2.1 Message Buffering E-5
E.2.2 Thread out of memory error E-5
E.2.3 Use of `mmap(2)` E-5
E.2.4 Fortran Interface to Shmem E-6

Glossary **Glossary-1**

Index **Index-1**

List of Figures

2.1 A Network of Nodes 2-2

3.1 Loading and Running a Parallel Program 3-2

List of Tables

B.1 RMS Allocation Options (prun/allocate) Supported by LSF B-9

B.2 LSF Load Indices B-9

E.1 MPI and libelan environment variables E-2

E.2 Shmem Environment Variables E-3

1

Introduction

1.1 Scope of Manual

This manual provides an introduction to running parallel jobs with the Quadrics Resource Management System (RMS) and QsNet. RMS provides a programming environment for running parallel programs. QsNet provides for high performance communication between processes in a parallel application. QsNet is also known as the HP AlphaServer SC Interconnect in an HP AlphaServerSC.

1.2 Audience

This manual is for users who run applications on an HP AlphaServerSC system operating under RMS and for programmers who develop and run parallel programs on such a system.

The manual includes programming examples which assume that the reader is familiar with either the C or Fortran programming languages.

1.3 Using this Manual

This manual contains four chapters and three appendices. The contents of these are as follows:

Chapter 1 (*Introduction*)

Describes the layout of the manual and the conventions used to

present information.

Chapter 2 (*Getting Started*)

Explains how to use RMS to run a simple parallel program.

Chapter 3 (*RMS User Commands*)

Introduces the RMS user commands.

Chapter 4 (*MPI and Shmem Programming*)

Describes how to compile and run a parallel program.

Appendix A (*RMS Commands*)

Contains manual pages for each of the RMS user commands.

Appendix B (*Submitting LSF Jobs*)

Provides details of how to submit jobs via LSF.

Appendix C (*MPI Library Routines*)

Provides details of the implementation of the MPI library.

Appendix D (*Shmem Library Routines*)

Provides details of the implementation of the Shmem library.

Appendix E (*Elan Library Environment Variables*)

Describes some environment variables specific to HP AlphaServer SC systems and programs using the MPI library.

1.4 Related Information

The following manuals provide additional information about RMS:

- *HP AlphaServer SC RMS Reference Manual*
- *Quadrics Shmem Programming Manual*
- *HP AlphaServer SC System Administration Guide*
- *HP AlphaServer SC Installation Guide*

1.5 Location of Online Documentation

New versions of this and other Quadrics documentation can be found on the Quadrics web site `http://www.quadrics.com`.

1.6 Reader's Comments

HP welcomes any comments and suggestions that you have on this document. End users should send all comments and suggestions to their HP support engineer.

1.7 Conventions

The following typographical conventions have been used in this document:

`monospace type`

Monospace type denotes literal text. This is used for command descriptions, file names and examples of output.

`bold monospace type`

Bold monospace type indicates text that the user enters in contrast to on-screen computer output.

`italic monospace type`

Italic (slanted) monospace type denotes some meta text. This is used most often in command or parameter descriptions to show where a textual value is to be substituted.

italic type

Italic (slanted) proportional type is used in the text to introduce new terms. It is also used when referring to labels on graphical elements such as buttons.

`Ctrl/x`

To engage this symbol depress the `Ctrl` key while depressing another key or mouse button (shown here by `x`).

TLA

Small capital letters indicate an abbreviation (see Glossary).

`ls(1)`

A cross-reference to a reference page includes the appropriate section number in parentheses.

#

A number sign represents the superuser prompt.

%, $

A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn and POSIX shells.

2

Getting Started

2.1 Introduction

This chapter provides an overview of the Resource Management System, the software that manages and controls access to the resources of a HP AlphaServer SC system. It describes how to log into the system, get help on RMS and run a simple parallel application using the RMS services.

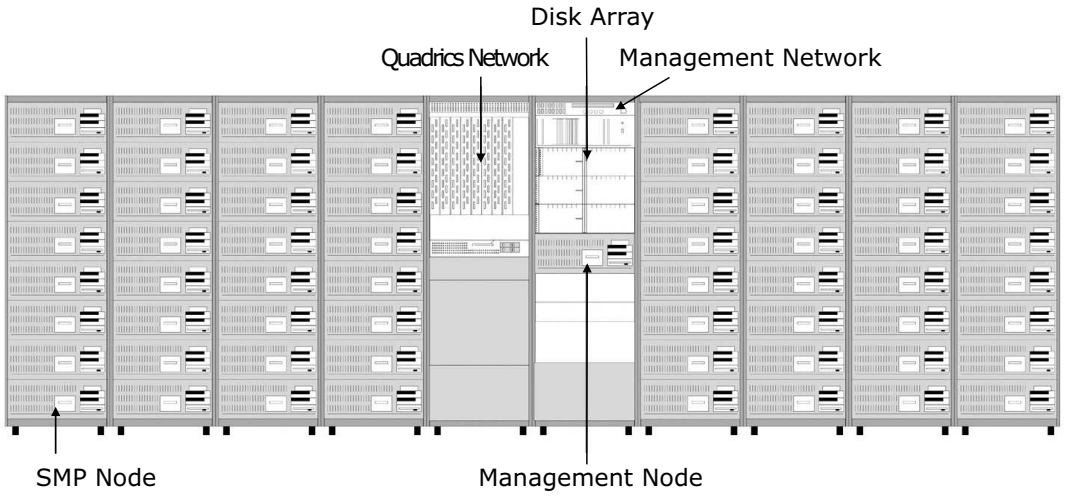
2.2 About RMS

A HP AlphaServer SC system comprises a cluster of AlphaServer nodes as shown in Figure 2.1.

The nodes, which are generally multiprocessor computers, are connected by a management Ethernet. They may also be connected to a HP AlphaServer SC Interconnect for high performance user space communications between application processes. Each node runs a copy of the standard UNIX® operating system.

One or more of the nodes in the cluster are used interactively. These login nodes are generally connected to an external local area network (LAN). The application nodes, used for running parallel programs, are accessed solely through RMS.

Figure 2.1: A Network of Nodes



The RMS daemons, which manage the system, reside either on one of the interactive nodes or on a separate management node. This node, which runs RMS, is given the hostname alias of `rmshost`.

The nodes in an RMS cluster can be grouped into mutually exclusive sets known as *partitions*. Your system administrator may have created partitions to dedicate resources to a particular activity or group of users. The set of partitions running at any given time is called the *active configuration*. An RMS cluster may operate in different configurations (at different times of the day), reflecting a changing pattern of resource allocation.

2.2.1 The User Interface

RMS provides a number of command line utilities that interact with the system on the users' behalf. These utilities perform tasks of general use such as querying the system's resources, loading parallel programs and running them.

Some of the utilities are specifically for system administrators. These utilities are described in the *RMS Reference Manual* together with details of the RMS daemons, which manage the system. This manual concentrates on the utilities designed for users. These are as follows:

`prun` This loads and runs parallel programs. A parallel program is a set of UNIX processes distributed over the nodes in a partition. The processes communicate over the data network using the MPI (Message Passing Interface) or Shmem (shared memory) libraries.

`rinfo`	This displays information about the resources available and shows which applications are running.
`allocate`	This preallocates a set of resources for running a series of jobs on the same nodes.
`rmsexec`	This runs a sequential program on a lightly loaded node.
`rmsquery`	This submits SQL queries to extract information from the RMS database. As a user, you have read-only access to the database.

2.2.2 The RMS Daemons

The RMS daemons manage the system. They interact through sockets and the RMS database. Each daemon is responsible for a different aspect of RMS. The daemons are largely transparent to the users of RMS. They are described here to provide some background information on the operation of the system. For more details, see the *RMS Reference Manual*.

Database Server The Database Server (`msql3d`) manages access to the RMS database services.

Machine Manager The Machine Manager (`mmanager`) oversees the physical operation of the *machine* – the cluster of nodes connected together by the network and running RMS. It updates the RMS database to reflect changes in the status of these nodes.

Partition Manager The Partition Manager (`pmanager`) controls the use of nodes, the allocation of resources to users and the scheduling of parallel programs on each partition. There is a Partition Manager for each partition.

Switch Network Manager

The Switch Network Manager (`swmgr`) supervises the operation of the HP AlphaServer SC Interconnect switch network, logging network errors to the RMS database.

Event Manager The Event Manager (`eventmgr`)is responsible for handling events generated by the other daemons. For example, it runs a handler script in response to events such as a node crash or fan failure. It also provides an interface to clients that wish to wait for certain events to occur.

Transaction Log Manager

The Transaction Log Manager (`tlogmgr`) instigates changes in system state that have been requested in the *Transaction Log*. All such changes are made through this mechanism to ensure that updates of the database are serialized and an audit trail is kept.

Process Manager The RMS Process Manager (`rmsmhd`) runs on each node in the system. It starts all of the other daemons.

RMS Daemon The RMS per-node daemon (`rmsd`) runs on each node in the machine. It loads and runs user processes and monitors resource usage and performance.

2.3 Getting Started

The nodes in an RMS cluster run a standard UNIX operating system. This means they have the usual UNIX command shells, editors, compilers, linkers and libraries; they run the same applications. RMS extends standard UNIX by providing utilities for running parallel applications as well as sequential ones. If you are not familiar with UNIX, please refer to the user documentation supplied with your system.

For Alpha™ systems, see the following Web site:
`http://www.tru64unix.compaq.com/`

Information on the windowing system, the Common Desktop Environment, is also available on these sites. The standard textbook on UNIX is *The UNIX Programming Environment* by Kernighan and Pike. This provides a general introduction to the standard UNIX utilities and command shells.

2.3.1 Logging In

The HP AlphaServer SC system is generally accessed across a LAN from a workstation, using the `telnet`, `rsh` or `ssh`. command, as follows:

```
$ telnet name
```

Contact your system administrator to identify the value of *name* for your system.

As the connection is being made, some messages may be displayed on the screen, giving the Internet address of the system and its hostname. When the connection has been established, a login prompt appears. Enter your login name and, when prompted, your password. The password is not displayed on the screen as you enter it.

```
login: user
Password:
```

When you enter the correct password, the system logs you in to your *home* directory. Each user has their own home directory. A 'message of the day' may be displayed and the command prompt appears.

```
Last login: Mon Nov  2 12:22:22 from diplodocus
You have mail.
$
```

A word of warning about passwords: UNIX security is based on keeping passwords secret. If other people know your password then they can tamper with your work. The operating system provides a controlled mechanism for sharing work and data using access permissions; keep your password secret.

2.3.2 Getting Help

Online RMS documentation is supplied for use in these formats:

1. RMS release notes and manuals are supplied in HTML format for use with a Web browser such as Netscape Navigator or Internet Explorer.
2. RMS release notes and manuals are supplied in PDF for use with a PDF reader. They are also provided in PostScript format for printing.
3. Manual pages for the RMS commands are supplied for use with the UNIX `man(1)` command.
4. RMS commands have a `-h` option that provides help.

Using the man Program

Manual pages provide concise summaries of commands and the files that the commands use. They are useful if you already know something about the command or file and wish to find out more. You can use the `man(1)` command to provide information about itself by entering the following command:

```
$ man man
```

At the end of each page of information, `man` pauses. Press the space bar to read the next page or enter `q` to quit (`man` uses the `more(1)` command to display its pages).

The following command displays information about the C shell:

```
$ man csh
```

For ease of access, the manual pages for the RMS commands are included in Appendix A (*RMS Commands*) of this manual.

2.4 Running a Parallel Program

To run a parallel program, use the RMS utility called `prun`. Without writing any code, you can experiment with `prun` right away. In this example, we use the UNIX program `hostname(1)`.

```
$ prun -N 4 hostname
atlas0
atlas1
atlas2
atlas3
```

The example is a very simple parallel application in which four copies of the sequential program `hostname` are executed at the same time, one per node. There is no interprocess communication.

Note that this example assumes that the system administrator has set up a default partition as no partition has been specified on the command line.

You run a parallel application that does have communicating processes in the same way, using `prun` to load and execute the processes. Try running one of the example programs in the directory `/usr/opt/rms/bin`, as follows:

```
$ prun -N 4 dping
  2:        0 bytes      2.42 uSec     0.00 MB/s
  0:        0 bytes      2.44 uSec     0.00 MB/s
```

The `dping` program prints the time taken (in microsecs) for two processes to communicate over QsNet. `prun` is the *controlling process* of your parallel program: `stdio` from the processes in the parallel program is routed through `prun`. If `prun` is killed, the processes started by `prun` are also killed.

2.4.1 Partitions

In both of the examples shown previously, `prun` executes the parallel application on the default partition, set up by your system administrator. To run a parallel program on a specific partition, use the `-p` option to `prun` and specify the name of a partition on your machine. In the following example, the partition called `small` is selected:

```
$ prun -p small -N 2 dping
  0:        0 bytes      2.47 uSec     0.00 MB/s
```

Each partition is controlled by a scheduler (the Partition Manager). The scheduler shares the processing resources of the partition between competing jobs. For example, a partition with ten nodes could run two five-node jobs concurrently. However, if one job

required eight nodes and another job required six, the scheduler would interleave the two jobs, giving each a certain amount of run time before suspending it so that the other one could run. This feature, called *time slicing*, may not be enabled on your system. If it is not, the second job would block until the first has completed. A smaller job, requiring only two nodes, would run. This is called *space sharing*.

The scheduling policy for a partition is controlled by its `type`. The types supported are `parallel` (the partition is optimized to only run parallel jobs), `login` (the partition runs UNIX login shells and load-balanced sequential tasks), `general` (all of the above) and `batch` (the partition is under the exclusive control of a batch system).

See the *RMS Reference Manual* and the RMS chapter of the *System Administration Guide* for more information on RMS job scheduling.

2.4.2 Priorities and Projects

If one of the jobs had a higher priority than the other, the higher priority job would run to completion before the lower priority job started. In fact, if the lower priority job was already running, the scheduler would suspend it to make way for the higher priority job. Priorities are set by the system administrator. They can be assigned to individuals or to groups of users (such a group is called a *project*) to give them preferential access.

See Section 3.3.4 for more information on the resources available to you. Priorities and projects are discussed further in Section 3.4, in the RMS chapter of the *System Administration Guide* and in the *RMS Reference Manual*.

3

RMS User Commands

3.1 Introduction

RMS provides a set of commands for running parallel programs and monitoring their execution. The set includes utilities that determine what resources are available and commands that request allocation of resources. This chapter describes how to use these tools.

The following command line utilities are described in this chapter:

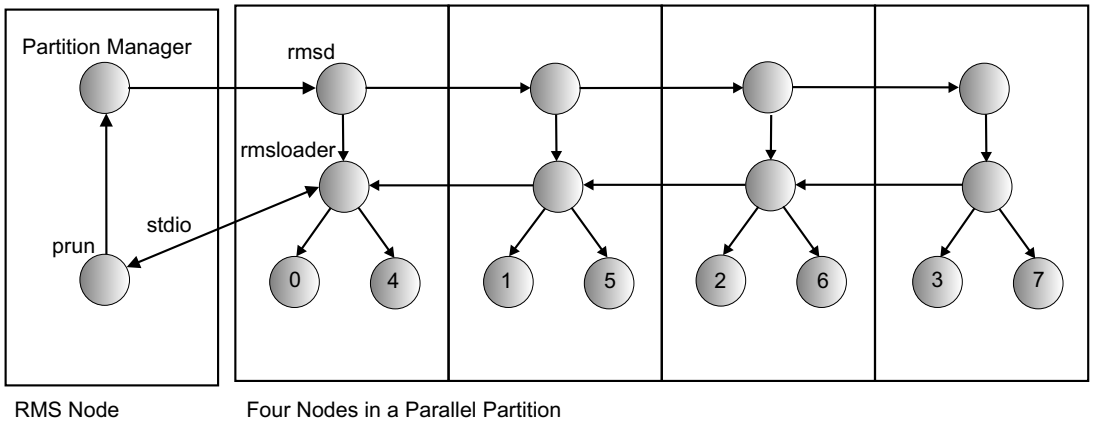
- `rinfo` tells you about the resources available on your system (see Section 3.3).
- `prun` runs parallel programs (see Section 3.4).
- `allocate` enables you to pre-allocate CPUs for a sequence of jobs (see Section 3.5).
- `rmsexec` starts processes on lightly loaded nodes, for example, on a node with free memory or idle CPUs (see Section 3.6).

Section 3.2 explains more about how parallel programs are executed under RMS. Appendix B (*Submitting LSF Jobs*) provides an introduction to using LSF® on an HP AlphaServer SC system.

3.2 More on Parallel Programs

A parallel program consists of a controlling process (`prun`) and a set of application processes distributed over the nodes in a partition. Each application process can have

Figure 3.1: Loading and Running a Parallel Program



multiple threads running over one or more CPUs.

RMS assigns a unique number, known as the *rank*, to each process in a parallel program. The numbers range from `0` to `n-1`, where `n` is the number of processes in the program.

The processes in a parallel program can communicate with each other across the data network. They do this by calling routines from one of the interprocess communication libraries, passing in the rank of the remote process. RMS includes MPI and Shmem libraries that have been optimized for the HP AlphaServer SC Interconnect. These libraries are described in more detail in Chapter 4 (*MPI and Shmem Programming*).

The `prun` command sends a request to the Partition Manager to allocate the resources (CPUs and memory) required by the parallel program. Once the resources are available, the Partition Manager instructs the RMS per-node daemons to load an RMS process called `rmsloader` on each node. The `rmsloaders` execute a `fork(2)` and then `exec(2)` to start the application processes executing as shown in Figure 3.1. The RMS daemon runs as root but the `rmsloader` and the application processes are run under the `uid` of the user. They inherit the user's environment, access permissions, group rights and any DCE keys. The `rmsloader` processes forward output printed on `stdout` and `stderr` to `prun` and hence to the terminal or output file. Similarly, `stdin` is routed to the application processes.

The parallel program terminates when all its processes have exited. The exit status returned is formed by a global `OR` of the exit status of each process. The UNIX convention is that an exit status of zero indicates that a program has completed successfully. A non-zero exit status indicates that a problem has occurred.

If one or more of the application processes is killed by a signal (`SIGSEGV`, for example), the remaining processes are killed and `rmsloader` runs the core file analysis script (`core_analysis`) which prints information on why the job failed.

3.3 Getting Resource Information with rinfo

Before invoking `prun` to run a parallel program, you can check what resources are available for your use. RMS enables the system administrator to configure the cluster of nodes into a number of different partitions, each with different numbers of nodes and different sets of resource quotas and priorities. This means that there may be restrictions on which resources you can use.

Moreover, the administrator can set up a number of these configurations and switch between them to suit different working patterns. For example, during the daytime, many users may be competing to use the resources for development purposes, whereas, at night, there may be a few large production jobs that run for a long time with no user interaction.

The command `rinfo` shows how the machine is configured, who is using the resources and what jobs these users are running on their resources. The following is an example of the output from `rinfo`:

```
$ rinfo
MACHINE        CONFIGURATION
atlas          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   atlas[0-2]
parallel        2/4   running    01:02:29             atlas[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  atlas0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  atlas0
```

This example shows one resource request `996` that has been allocated 2 CPUs in the `parallel` partition. The CPUs have been allocated on node `atlas0` and are being used to run job `1115`.

The output is split into four sections:

1. Machine

   This section shows the following:

   - The name of the machine
   - The name of the active configuration

2. Partitions

   This section has an entry for each partition in the active configuration. Each entry shows the following:

- The partition name
- The number of processors in use and the total
- The status of the partition
- The partition's up time, in hours, minutes and seconds
- The upper time limit imposed on jobs, in hours, minutes and seconds
- The names of the nodes in the partition

The `rinfo` output includes an entry for the `root` partition. It includes all nodes in the system. Only the system administrator can run jobs on this partition.

3. Resources

   This section has an entry for each resource request. A resource in this context means a set of processors with their associated memory and devices. Each entry shows the following:

   - The name of the resource (generated automatically) prefixed by the name of the partition and a full stop character
   - The number of processors assigned to the resource
   - The status of the resource
   - The amount of time the resource has been allocated, shown in hours, minutes and seconds
   - The name of the user who has been allocated the resource
   - The names of the nodes that have been allocated to the resource.

4. Jobs

   This section has an entry for each job running on the machine. Each entry shows the following:

   - The name of the job (generated automatically) prefixed by the name of the partition and a full stop character
   - The number of processors that the job is using
   - The status of the job
   - The amount of time the job has been running, shown in hours, minutes and seconds
   - The name of the user who is running the job
   - The names of the nodes across which the job is distributed

   If no resources are in use, only the machine and partition sections are displayed.

3.3.1 Node Name Format

Note that node names are specified by a pattern-matching syntax used by the UNIX shell (see `glob(3)`). For example, `atlas[8-10,12,15]` refers to the nodes named: `atlas8`, `atlas9`, `atlas10`, `atlas12` and `atlas15`. The numbers in square brackets all share the common stem that precedes the square brackets. Within the square brackets, two numbers separated by a hyphen denote an inclusive range while numbers separated by commas or white space represent a list. The UNIX pattern-matching syntax is extended to include numbers of more than one digit.

3.3.2 Command Line Options

`rinfo` has a number of command line options that let you restrict or expand the amount of information displayed.

```
rinfo [-chlmnpqr] [-o fields]
       [-j [-l] [job1 job2 ...]]
       [-u [-jlr] [user1 user2 ...]]
       [-L [partition] [statistic]]
       [-s [server [hostname]] | all ]
       [-t host | node]
```

Use the `-h` option to get a list of all of the available options and valid arguments. This option works in the same way for all of the RMS commands. The following sections examine some of the more commonly used options. See Appendix A (*RMS Commands*) for more details.

3.3.3 Querying the Machine's Users

By default, `rinfo` lists the current resource requests and jobs of all users, as shown in the following example:

```
$ rinfo
MACHINE        CONFIGURATION
atlas          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   atlas[0-2]
parallel        4/4   running    01:02:29             atlas[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  atlas0
parallel.997      2 allocated       00:02        dave  atlas0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  atlas0
parallel.1116     2   running       00:02        dave  atlas0
```

To restrict the output to the resource section only, specify the `-r` option. For just the jobs section, specify the `-j` option. Used with no argument, these options show just the resource or job names.

To restrict the output to information about your own usage, specify the `-u` option to `rinfo`.

3.3.4 Checking Quotas

The system administrator can set limits on the way a partition is used. Usually, these limits are set at the project level so that each user working on a project is automatically subject to the limits set for the project as a whole. For example, there might be a limit of 20 on the number of processors available for project `alpha` at any one time. Therefore, if two users, working on project `alpha`, each had 8 processors allocated to them, only 4 processors would be available for a third project member, even though `rinfo` might show that the partition had 64 processors and there were no other users.

The system administrator can limit the following:

- The maximum number of CPUs
- The maximum amount of memory in MB per CPU
- The scheduling priority

The system administrator can also set limits on individual users. These are always more restrictive than those imposed at the project level.

To check what restrictions have been set, run `rinfo` with the `-q` option.

```
$ rinfo -q
PARTITION       CLASS         NAME     CPUS   MEMLIMIT  PRIORITY
parallel         user       duncan      0/8        256        25
parallel      project        alpha    16/20        256        25
parallel      project      default     4/16        256        10
```

This example shows that the system administrator has created three sets of access control restrictions that apply to the user `duncan`. In the first, the user `duncan` is restricted to a maximum of 8 CPUs with 256MB of memory per CPU. His jobs are assigned priority 25. None of these CPUs is in use. The second control restricts `duncan` and the other members of the `alpha` project to a total of 20 CPUs, 16 of which are in use. The third control restricts the members of the `default` project to a total of 16 CPUs, 4 of which are in use.

The user `duncan` can run jobs in either project, but if he selects the (higher priority) `alpha` project he will only be able to use 4 CPUs at this point in time. This is because

allocating more than 4 CPUs would cause the `alpha` project to exceed its usage limit. Such a request would be blocked pending the completion of other requests made by members of this project – `rinfo` would show the resource request in the `blocked` state.

You can explicitly lower your CPU and memory quotas; this may help in getting your jobs run sooner. `prun` has command line options for specifying the number of CPUs required, as described in Section 3.4. To reduce the memory limit, set the environment variable `RMS_MEMLIMIT` to the memory required per process (in MB) before using `prun`.

To specify the name of the project you are working on, use the `-P` option to `prun` or set the environment variable `RMS_PROJECT` before using `prun`.
Appendix A (*RMS Commands*) lists all of the environment variables that can be used with `prun`.

3.3.5 Viewing Configuration Details

The `-c` option displays the names of all of the machine configurations:

```
$ rinfo -c
day
night
```

The long format of the `-c` option provides more information about the partitions

```
$ rinfo -cl
CONFIGURATION   PARTITION   STATUS    ALLC  SUSP  QUED  BLKD  NODES
day             login       running      1     0     0     0  atlas[0-3]
day             parallel    running      2     0     4     0  atlas[4-63]
night           parallel          -                          atlas[0-63]
```

You can find out the names of all of the active partitions with the `-p` option. This also gives the number of CPUs in each partition:

```
$ rinfo -p
parallel 16
root      2
login     4
```

The `-m` option displays the name of the system:

```
$ rinfo -m
atlas
```

3.4 Running Programs with prun

`prun` is the RMS utility for running parallel programs. It loads multiple copies of a single application program onto a range of nodes and runs them. `prun` acts as the program's interface to RMS, handling `stdio` and forwarding certain signals.

You specify to prun how many processes to load and on which partition. In addition, the prun options enable you to select more precisely the distribution of the processes in the partition. Unless you have already allocated a resource for the program (see Section 3.5), prun does so on your behalf, blocking until appropriate CPUs become available.

3.4.1 prun Command Line Options

The syntax of the prun command is as follows:

```
prun [-hIOrstv] [-B base] [-c cpus] [-e mode] [-i mode] [-l tag] [-o mode]
     [-N nodes | all] [-n procs] [-m block | cyclic] [-P project] [-p partition]
     [-R request] program [args ...]
```

Use the -h option to get a list of all of the options and valid arguments. This section introduces some of the most widely used options. See Appendix A (*RMS Commands*) for full details.

3.4.2 Selecting a Partition

With the -p option, you can select a partition for the program. If you omit this option, a default partition, nominated by the system administrator, is used.

In the following example, two copies of myprog are loaded onto the partition called parallel:

```
$ prun -p parallel -n 2 myprog
Hello from myprog
Hello from myprog
```

You can specify your own default partition by setting the environment variable RMS_PARTITION.

If no partition is specified and the system administrator has not set up a default, you will get an error message.

3.4.3 Specifying Processes and Nodes

You can specify how many instances of a program to run by using the -n option as shown in the last example. You can also specify how the processes are distributed across the nodes in the partition. In the following example, two instances of hostname(1) are requested. The arguments after the program name are passed to both of the processes.

```
$ prun -n 2 hostname
atlas4
atlas4
```

Note that both instances of `hostname` ran on the same node, `atlas4`. By default, RMS allocates one process per processor, using all of the processors on one node before moving on to the next. You can override this behaviour by using the `-N` option, which specifies how many nodes are required for the program. In the following example, four nodes are requested and an instance of `hostname` is executed on each:

```
$ prun -N 4 hostname
atlas4
atlas5
atlas6
atlas7
```

The `-n` and `-N` options can be used in combination to place more than one process on each node. In the following example, four processes are executed, two per node:

```
$ prun -n 4 -N 2 hostname
atlas4
atlas4
atlas5
atlas5
```

The RMS scheduler allocates one CPU per process, distributing the processes evenly (provided `n` is divisible by `N`) over the requested number of nodes. If you are not concerned with how the processes in your application are distributed over nodes then use the `-n` option alone and your application will be run as soon as sufficient CPUs are available. If you require the same number of CPUs on each node, use the `-N` option.

The `-c` option allows you to specify how many CPUs you want for each process. This is for use in multithreaded applications.

The following example runs four `myprog` processes on two nodes, allocating two CPUs per process – 8 CPUs in total:

```
$ prun -c 2 -n 4 -N 2 myprog
```

RMS does not take any stance on how the additional CPUs are to be used. This is up to the program.

RMS does not normally run more processes per node than there are CPUs. However, there are circumstances in which this can be useful. The `-O` option allows you to do this, as shown in the following example:

```
$ prun -n 5 -N 1 myprog
prun: Error: can't allocate 5 cpus on 1 node: max cpus per node is 4

$ prun -O -n 5 -N 1 myprog
myprog running on atlas0
```

```
myprog running on atlas0
myprog running on atlas0
myprog running on atlas0
myprog running on atlas0
...
```

The `-m` option to `prun` gives you control over how the processes are distributed over the nodes. The alternatives are `block` (the default) and `cyclic`. Both are illustrated below together with the `-t` option which prefixes the process number (the rank) onto each line of output.

```
$ prun -n 4 -N 2 -t -m block hostname
0 atlas0
1 atlas0
2 atlas1
3 atlas1
$ prun -n 4 -N 2 -t -m cyclic hostname
0 atlas0
2 atlas0
1 atlas1
3 atlas1
```

To use a specific range of nodes use the `-B` option to specify the base node and the `-N` to specify the number of nodes, for example:

```
$ prun -B 2 -N 2 -t hostname
0 atlas2
1 atlas3
```

By default `prun` does not add to the output of the parallel program. Use the `-v` option to generate a little more information on what is going on, for example:

```
$ prun -v -N 2 hostname
prun: resource 30102 starting 2 processes on 2 cpus default memlimit no timelimit
atlas0
atlas1
```

Using `-vv` or `-v` twice will generate additional output as the program starts and as each process exits, for example:

```
$ prun -vv -N 2 hostname
prun: running /usr/bin/hostname
prun: requesting 2 CPUs
prun: resource 30103 starting 2 processes on 2 cpus default memlimit no timelimit
prun: stdio server running
atlas0
prun: loader 0 exited with status 0
atlas1
prun: loader 1 exited with status 0
```

3.4.4 Input and Output

Each process in a user's application has three standard input/output (I/O) streams:

1. `stdin` or unit 5 in Fortran
2. `stdout` or unit 6 in Fortran
3. `stderr` or unit 0 in Fortran

The use of these streams by parallel programs is different from that of sequential programs (that is, standard UNIX applications that execute independently of all other processes).

When the parallel processes start executing, `stdout` and `stderr` are routed to `prun`. Normal `write` operations to these file descriptors have the expected effect, as do calls to the `isatty(3)` function. Other `ioctl` functions are not reliable. `stdin` is also routed through `prun`. The default behavior is as follows:

`stdin`	This is directed to the process with rank `0`. As mentioned in Section 3.2, RMS assigns each process a rank in the range `0` to `n-1`, where `n` is the number of processes in the program.
`stdout`	This is used for line-buffered output from all processes.
`stderr`	This is used for unbuffered output from all processes.

Input

By default, `prun` directs `stdin` to the process with rank 0. Processes can determine their rank by checking the value of the environment variable `RMS_RANK`, which is set by `prun`.

To redirect `stdin` in different ways, use the `-i` option to `prun`: you can specify which processes may read from `stdin`; alternatively, you can specify that processes read `stdin` from a file.

In the following three examples, the first allows all processes to receive the data on `stdin`, the second restricts `stdin` to a single process identified by its rank and the final example prevents `stdin` from reaching any of the processes.

```
$ prun -i all -n 6 myprog
$ prun -i 3 -n 6 myprog
$ prun -i none -n 6 myprog
```

The first of the following examples specifies a file as the source of `stdin`. If the file does not exist, a read by any of the processes returns the value `EOF`. The second example

specifies a separate file for each process. prun generates file names by substituting each process's rank for the % character, so that process 0 reads stdin from myinfile.0, process 1 reads it from myinfile.1 and so on. As with the previous method of redirecting stdin, if the file does not exist, a read returns the value EOF.

```
$ prun -i myinfile -n 6 myprog
$ prun -i myinfile.% -n 6 myprog
```

To specify your own default method for handling stdin set the environment variable RMS_STDINMODE to one of the input modes described above. For example:

```
$ setenv RMS_STDINMODE myinfile.%
```

Output

By default, stdout and stderr from all processes in a parallel program are forwarded to prun. When the processes print simultaneously to stdout, the output appears in an arbitrary order. In fact, the ordering may be different each time the program runs.

The -t option of prun tags the output of each process with the rank of the process. This makes it easy to identify the source of messages output by the program, as shown in the following example:

```
$ prun -n 4 -t pwd
2 /home/user
0 /home/user
3 /home/user
1 /home/user
```

stdout is line-buffered and forwarded to prun when a newline character is received. This means that the output from each process is displayed on a separate line. Standard output that does not end in a newline character is buffered by the rmsloader.

stderr is forwarded from all processes to prun immediately with no buffering.

To redirect stdout and stderr in different ways, use the -o and -e options respectively to prun. You can specify which processes may write to stdout and stderr; alternatively, you can specify that processes write stdout and stderr to a file or a file per process.

In the following three examples, the first allows all processes to write to stdout, the second restricts stdout to a single process identified by its rank and the final example prevents all processes from writing to stdout.

```
$ prun -o all -n 4 -t pwd
3 /home/user
0 /home/user
```

```
2 /home/user
1 /home/user
$ prun -o 3 -n 4 -t pwd
3 /home/user
$ prun -o none -n 4 -t pwd
```

The first of the following examples redirects `stderr` from all processes to a file. The second example specifies a separate file for each process. `prun` generates file names by substituting each process's rank for the % character, so that process 0 writes `stderr` to `myerrfile.0`, process 1 writes to `myerrfile.1` and so on.

```
$ prun -e myerrfile -n 4 -t pwd
$ prun -e myerrfile.% -n 4 myprog
```

The `-i`, `-o` and `-e` options can be combined. The following example directs standard input to process 2 and directs standard output to a file but cancels the forwarding of error output to the terminal.

```
$ prun -i 2 -o myoutfile -e none -n 4 myprog
```

To specify your own default method for handling `stdout`, set the environment variable `RMS_STDOUTMODE` to one of the output modes described above. Set `RMS_STDERRMODE` to select a default method for `stderr`. For example:

```
$ setenv RMS_STDOUTMODE out.%
```

`prun` provides support for prefixing each line of output from a parallel program with a process dependent tag. The `-t` option generates a tag identifying the process by its rank. The `-l` option allows a more general character string prefix to be specified. The following special characters in the string are expanded:

%g	the global rank of the process
%G	the number of processes
%h	the rank of the node
%H	the number of nodes
%l	the rank of the process on this node
%L	the number of processes on the node
%m	text base of the hostname
%n	numeric part of the hostname
%p	the pid of the process
%t	the current time (hh:mm:ss)
%@	the hostname of the node (the same as "%m%-n")
%%	a single %

For example, to prefix each line of output with the hostname use

```
$ prun -l"%@: " -N4 pwd
atlas0: /home/duncan
atlas3: /home/duncan
atlas2: /home/duncan
atlas1: /home/duncan
```

3.4.5 RMS Environment Variables

The RMS environment variables are described in full in Appendix A (*RMS Commands*). We have already mentioned some of them, for example, `prun` sets the variable `RMS_RANK` and reads the variable `RMS_PROJECT`. Another environment variable that you can set is `RMS_IMMEDIATE`. This tells `prun` to exit rather than block if the resources required for the program are not available immediately.

The following example sets two environment variables (from the C shell) and uses some environment variables created by `prun`:

```
$ setenv RMS_IMMEDIATE
$ setenv RMS_PROJECT database
$ prun -n 4 csh -c 'echo process $RMS_RANK of $RMS_NPROCS'
process 3 of 4
process 2 of 4
process 0 of 4
process 1 of 4
```

First of all, `RMS_IMMEDIATE` is set so that `prun` will not block if insufficient resources are available. Instead, it will return immediately. This has the same effect as running `prun` with the `-I` option.

Then we specify with `RMS_PROJECT` that the subsequent jobs belong to the project called `database`. This will affect the CPU usage and memory limits and priority applied to the jobs and also the accounting records that RMS keeps. If you do not set `RMS_PROJECT`, default values set by the system administrator will apply to the jobs. You can also specify a project by using the `-P` option of `prun`.

When `prun` runs the four shell processes as requested, the rank of each instance of `echo` is displayed together with the total number of processes, `RMS_NPROCS`. RMS passes all of the environment variables to the processes it executes.

3.4.6 Memory Limits

RMS can impose memory limits on the processes in a parallel application. The default limits ensure that each process has a fair share of the memory available. The system administrator can raise or lower the limits on a per user or per project basis. Limits on users are always more restrictive than those on projects.

Use `rinfo` with the `-q` option or `prun` with the `-v` option to see the limits that apply to you (or the projects to which you belong).

```
$ rinfo -q
PARTITION       CLASS         NAME     CPUS   MEMLIMIT  PRIORITY
parallel      project      default     0/22         96        50

$ prun -v -n 2 dping
prun: starting 2 processes on 2 cpus memlimit 96 MB no timelimit
  0:        0 bytes      2.43 uSec     0.00 MB/s
```

Exceeding the memory limit causes `brk()`, `malloc()`, `mmap()` and `sbrk()` system calls to fail with `ENOMEM`. Programs with static arrays larger than the memory limit fail immediately. For example, if you run a process like the following with too small a memory limit, it will exit before entering `main()` and then `prun` will exit immediately.

```
#include <stdio.h>

int array[50 * 1024 * 1024];

main (int argc, char **argv)
{
    printf("hello world\n");
    exit(0);
}
```

If you use `prun` with the `-v` option, it prints a warning if all processes exit because of insufficient memory.

```
$ prun -v -n 2 myprog
prun: starting 2 processes on 2 cpus memlimit 10 MB no timelimit
897757:myprog: /sbin/loader: Fatal Error: Program datasize exceeds process datasize limit.
899346:myprog: /sbin/loader: Fatal Error: Program datasize exceeds process datasize limit.
prun: Warning: exit 1 on all nodes. Possible memory limit or link/loader error
```

To check on the size of your application processes, use the command `size(1)`.

```
$ size myprog
text    data        bss          dec        hex filename
8192    8192  209707392    209723776    c802180 /home/duncan/myprog
```

The example shows that `myprog` requires 200 MB for each process. To set this limit, use the environment variable `RMS_MEMLIMIT` before starting your program. The units are MB per process. Note that you cannot set this environment variable to a value that is higher than the value set by the system administrator.

```
$ setenv RMS_MEMLIMIT 200
```

RMS memory limits are per CPU. A process started with the option `-c2` has 2 CPUs per process and hence twice the normal memory limit – unless a per-process limit has been set with the `RMS_MEMLIMIT` environment variable.

3.4.7 Examining Jobs and Resources

When you start running jobs with `prun`, use `rinfo` with the `-j` and `-r` options with a job name or resource name argument (respectively) to focus on a specific job or resource.

```
$ rinfo -j 1978
JOB          STATUS       COMMAND                   HOSTNAMES
1978         running      dping -n1000000 1m        atlas[0-1]

RESOURCE     PARTITION    CPUS   PRI   USERNAME     PID          PROJECT
2395         parallel     2      50    duncan       947803       default

ETIME        ATIME        UTIME        STIME        PAGEFLTS     MEM
00:09        00:18        00:14        00:00        47           9560

PROCESSES PER NODE
2 atlas[0-1]
```

Add the `-l` option to inspect the process IDs as well:

```
$ rinfo -jl 1978
JOB          STATUS       COMMAND                   HOSTNAMES
1978         running      dping -n1000000 1m        atlas[0-1]

RESOURCE     PARTITION    CPUS   PRI   USERNAME     PID          PROJECT
2395         parallel     2      50    duncan       947803       default

ETIME        ATIME        UTIME        STIME        PAGEFLTS     MEM
00:09        00:18        00:14        00:00        47           9560

PROCESSES
atlas0        947811   947801
atlas1       1160558  1160557
```

3.4.8 Program Termination

A parallel program terminates when all its processes have exited or when one or more processes is killed by a signal. If a program exits cleanly, the exit status returned is formed by a global `OR` of the exit status of each process. This allows an application to return a small number of carefully chosen non-zero exit status values when something goes wrong, to indicate what the problem is.

Two application exit codes have a special meaning. If a process exits with code 123 (`GLOBAL_ERROR`), the whole program will exit immediately and `prun` will exit with

status 123. If a process exits with code 127 (GLOBAL_EXIT), the whole program will exit immediately and prun will exit with status 0.

If one or more of the application processes is killed (for example, by the signal SIGSEGV), prun exits immediately with a status value of 128 plus the signal number that caused the process to be killed.

```
$ prun -v program
...
prun: program (pid 767544) killed by signal 11
...
$ echo $?
139
```

If the job is killed by the system administrator, its exit status will be 137; that is, 128 plus 9 (the signal number that caused the process to be killed).

```
$ prun -N 2 -v myprog
prun: resource 30110 starting 2 processes on 2 cpus memlimit 20 MB no timelimit
prun: loaders exited without returning status
$ echo $?
137
```

If the processes in a parallel program are started using a shell script then, by convention, the shell script exits with a status of 128 plus the signal number. RMS interprets this as an error and the program is terminated.

When RMS detects that a process it started was killed by a signal, it runs a core file analysis script. This script looks for core files created by the program and prints out a backtrace showing where the process failed, for example:

```
$ prun -N1 segv
prun: ./segv (host atlas1 process 1 pid 899515) killed by signal 11
prun: generating backtrace for ./segv
Thread terminated at PC 0x120001108 by signal SEGV
Stack trace for thread 5
>0  0x120001108 in main() "segv.c":4
#1  0x1200010a8 in __start(0x1, 0x11fffc018, 0x11fffc028, ...) in ./segv
prun: dumping elan exception state for ./segv
```

If the code has been compiled with the -g option, RMS will print a symbolic backtrace. In this simple example, the problem is on line 4 of segv.c. See Section 3.4.9 for details on what happens to the core file.

If a parallel program uses an invalid process ID or an invalid address in one of its interprocess communication calls, the hardware will generate an exception and the Elan driver will send a SIGSEGV to the program. The core file analysis script runs a utility called edb to extract infomation on Elan exceptions. For example:

```
$ prun -N 2 myprog
...
prun: dumping elan exception state for ./shmem
Dma - Invalid Process : Process=4 Res=22
    WakeupFnt=4 Cntx=9a SuspAddr=83 TrapType=MI_DmaLoop
    FaultAddr=4 EventAddr=100eead0 FSR=2450
0   FaultAddr=0 EventAddr=0 FSR=0
1   FaultAddr=0 EventAddr=0 FSR=0
2   FaultAddr=0 EventAddr=0 FSR=0
3   FaultAddr=0 EventAddr=0 FSR=0
    Type 009a2003 size 00000008 source 0cb0d040 dest 1fffbff8
    Dest event 0cc1e700 cookie/proc 00000004
    Source event 0cb0f440 cookie/proc 00040000
```

In this case, the process was trying to send 8 bytes of data to process 4, an invalid process.

```
$ prun -N 2 myprog
...
prun: dumping elan exception state for ./shmem
Inputter - Invalid Address : Res=14
    Fault Area FaultAddr=5fffbfe8 EventAddr=100ee5d0 FSR=66fb State=2 Status=0
    FaultAddr=5fffbfe8 EventAddr=100ee5d0 FSR=66fb
    NumTransactions=4 Overflow=0 AckSent=0 BadTransaction=0 QueuePointer=0
    WriteBlock Type=Double Size= 8 Addr=5fffbfe8 Cntx=0000008d Type ...
    DMA Identify    Addr=00200001 Cntx=0000008d Type=MI_InputDoTrap ...
    Setevent        Addr=0cc1e700 Cntx=0000008d Type=MI_InputDoTrap ...
    EOP BADACK      Cntx=0000008d Type=MI_InputDoTrap ...
```

In this case, the process was trying to send 8 bytes of data to an unmapped stack address.

3.4.9 Core Files

The RMS policy on handling core files when a process within a parallel job fails is as follows:

- Kill the remaining processes in the job – they are useless anyway.
- Generate core in the directory `/local/core/rms/`*`resource-id`*, where *`resource-id`* is the identifier of the resource. You can check the resource identifier by using `rinfo` with the `-r` option or by examining the value of the `RMS_RESOURCE_ID` environment variable. The core file has a name of the form `core.`*`program.hostname.instance`*. This file name is displayed by `prun`.
- Run a debugger on the core file and output the backtrace to `stderr`.

- Optionally, delete the core file and directory. This takes place when RMS is freeing the allocated resource.

  By default, core files are kept unless the system administrator has decided otherwise. You can override this decision by setting the environment variable RMS_KEEP_CORE.

  One reason for deleting core files and their directories is that, typically, production jobs are compiled with optimizations so there is little diagnostic information available in the core file. Another reason is that having hundreds of core files scattered over the local disks will quickly become a maintenance problem.

To diagnose a failing program, compile the program with the -g flag so that debugging and symbolic information is included in the executable file. Set RMS_KEEP_CORE to TRUE so that the core file is saved for debugging.

If your site policy is to delete core files, you can make a local copy using the following procedure:

1. Allocate a resource for the job, using the allocate command; then run the job using the prun command.
2. When the program fails, it produces a core file. The prun command prints the pathname of the core file.
3. Copy the core file to your home directory and exit the allocate subshell.

```
$ allocate -N 4
$ prun  myprog
myprog: process 0 killed by signal 11
...
...
$ echo $RMS_RESOURCEID
parallel.397
$ prun -B 0 -N 1 cp '/local/core/rms/397/core*' .
$ exit
```

If you want to catch core files from production runs, that is to say without using allocate, run the job in a script that copies the core file to a permanent location, or persuade the system administrator to do this in a site-specific core-file-analysis script. Alternatively, arrange for core file saving to be enabled temporarily.

3.4.10 Stats Collection

The Elan libraries can generate statistics on the inter-process communication calls in a parallel program. Statistics are written to a per-process file in the current working directory on delivery of a signal and as the process exits. The environment variable

RMS_STATINTERVAL controls the rate at which RMS sends the the signal LIBELAN_STATSIG to the application. Set it to 10 to get an update every 10 seconds. Alternatively just set the LIBELAN_STATSIG variable and use rcontrol to send the signal as and when required. For example

```
$ setenv RMS_STATINTERVAL 10
$ setenv LIBELAN_STATSIG 31
$ prun -N8 program
...

$ atlas: rinfo -jl
JOB             CPUS    STATUS        TIME    USERNAME  NODES
parallel.758       2   running       00:20      duncan  atlas[0-1]

$ rcontrol kill job 758 signal 31
```

The choice of which signal to use depends upon the application, SIGUSR1 (30) or SIGUSR2 (31) is recommended.

3.4.11 Propagating DCE Credentials

If DCE is installed on your system, you can log into DCE and have the RMS prun command propagate your DCE credentials to each node on which the job is run. You can do this in two ways:

- Use the dce_login command provided in the HP AlphaServer SC kit.
- Use the scgenkeytab command.

Using the dce_login Command

If DCE is installed on your system and you use the dce_login command to obtain DCE network credentials, you can log into DCE using the dce_login command, and have the RMS prun command propagate your DCE credentials to each node on which the job is scheduled to run.

To do this, you must use the version of the dce_login command that is provided in the HP AlphaServer SC kit. Use the ls command to check that this new version of the dce_login command is installed correctly.

```
$ ls -l /usr/bin/dce_login
lrwxrwxrwx 1 root  system    27 Feb 12 13:53 /usr/bin/dce_login -> /usr/sra/bin/scsec_login
```

The output should indicate that /usr/bin/dce_login is a symbolic link to the /usr/sra/bin/scsec_login file, as shown in the above example. If not, your system administrator must replace the original version of dce_login with that supplied in the

HP AlphaServer SC kit, as described in Chapter 8 of the *HP AlphaServer SC Installation Guide*.

To disable DCE credential propagation, set the SCSECDISABLED environment variable to 1.

Using the scgenkeytab Command

If you obtain DCE network credentials by a method other than the `dce_login` command (for example, if you use Secure Shell), you must run the `scgenkeytab` command after you log into the system and before you run the `prun` command. The `scgenkeytab` command prompts you for your DCE Principal name and password. In this situation, if you do not use the `scgenkeytab` command, the `prun` command will fail as follows:

```
$ prun hostname
prun: Error: scsec: keytab not found, please run scgenkeytab
```

As the name implies, the `scgenkeytab` command generates a keytab file, which contains your DCE password in an encrypted format. This keytab file remains valid while your DCE password is valid; that is, until the password expires or you change your password. While the keytab file is valid, you do not need to rerun the `scgenkeytab command` – if you wish, you can log out and then log back in and run the `prun` command.

However, if you change your DCE password or if your password expires, you must rerun the `scgenkeytab` command before using the `prun` command. In this case, if you do not rerun the `scgenkeytab` command, the `prun` command will fail as follows:

```
$ prun -n1 a.out
loader[atlas0]: sc_sec_sec_creds failed: Invalid password (dce/sec)
```

Note that the `scgenkeytab` command and the `prun` command must both be run within the same CFS cluster domain, but you do not have to run both commands on the same node.

3.4.12 Common Problems

There are some common problems and error messages that you may encounter when running applications. This section suggests some solutions.

Program Hangs

A program may hang if `prun` cannot allocate the resources required for the program and blocks, waiting until resources become available. Resource requests made by `allocate` behave in the same way.

If you enable verbose reporting with the `-v` option or by setting `RMS_VERBOSE`, `prun` outputs a message as your jobs starts:

```
$ prun -v -N 2 hostname
prun: starting 2 processes on 2 cpus memlimit 96
atlas0
atlas1
```

If the job is blocked waiting for resources, you get a warning message. Some time later, you get a start message followed by any output from your program.

```
$ prun -v -n 2 hostname
prun: Warning: waiting for free cpus
prun: starting 2 processes on 2 cpus memlimit 96
atlas0
atlas0
```

If `prun` hangs between the two messages, you can suspend it by pressing `Ctrl/Z` and use `rinfo` to find out what is going on. `rinfo` should report that your resource request is either `queued` or `blocked`. If the request is `queued`, it must wait for the completion of jobs submitted by other users or projects, as insufficient resources are free. If it is `blocked`, it must wait for the completion of other jobs that you (or your project) have submitted; that is to say, starting your job would cause the CPU usage quota or memory limit for you or your current project to be exceeded.

To prevent your request from blocking, set the `-I` option to `prun` or set the environment variable `RMS_IMMEDIATE`. This makes the job fail if resources are not available.

Error Messages

The error messages you may encounter include the following:

`prun: can't find command ./junk`

The problem here is that the program you specified cannot be located using your current search path. The solution is to add the program's directory to your `PATH` environment variable.

`prun: Warning: RMS server pmanager-parallel (rmshost) not responding`

The problem in this case is that the partition you specified with the `-p` option (or the default partition, if you did not specify one) is unavailable. The solution is to specify an alternative partition or ask the system administrator to restart the partition. Use `rinfo` (see Section 3.3 or Appendix A (*RMS Commands*)) to check the status of the partitions.

`prun: Error: no such partition as` *`name`*

The problem here is clear: the specified partition does not exist. The

most likely cause of this error is an incorrectly entered partition name. Once again, use `rinfo` to check the names of the partitions.

`prun: failed to start loader`

The RMS daemon, `rmsd`, which runs on each node, starts an `rmsloader` process on each node. These loaders connect back to `prun`, as shown in Figure 3.1. They find out from `prun` what they should run and then `exec(2)` the user's program and forward its I/O. This should all happen quickly. Things can go wrong, for example, if the working directory is not mounted uniformly on all nodes. A timeout is built into `prun` to catch such problems. This is the message that appears when the timeout expires.

`... Fatal Error: Program datasize exceeds process datasize limit`

The memory requirements of the process being started exceed the limits on the job.

Job is Killed

If your job is killed, `prun` exits early with a message and a status of `137`. This is the normal exit status of `128` plus the signal number of `SIGKILL`.

```
$ prun -v dping -n1000000 0 8k
prun: starting 2 processes on 2 cpus default memlimit no timelimit
  0:        0 bytes      2.69 uSec     0.00 MB/s
  0:        1 bytes      3.54 uSec     0.28 MB/s
  0:        2 bytes      3.70 uSec     0.54 MB/s
prun: loaders exited without returning status
$ echo $?
137
```

Similar messages are printed if a node that was running the job crashes.

3.5 Allocating Resources with allocate

RMS makes a distinction between allocating a resource (CPUs and memory) and running jobs on them (see Section 3.3 for more details on resources). `allocate` simply allocates resources. `prun` combines both tasks: allocating resources and running jobs.

Use `allocate` before `prun` if you want to run a sequence of jobs that have the same resource requirements. By using `allocate`, you only have to wait once for the resource to be allocated. It is also useful if you want to run several jobs concurrently.

There are two ways to use `allocate`. They are described in this section. Basically, `allocate` has an optional, final argument which is the name of a shell script. If you do

not specify a script, `allocate` spawns an interactive shell that has the resource allocated to it until you exit the shell. If you do specify a shell script, the resource is allocated to this script until it exits.

3.5.1 Command Line Options

`allocate` has a number of options with the same option letter and meaning as their `prun` counterparts. Use the `-h` option to get a list of the available options and valid arguments. See Appendix A (*RMS Commands*) for full details.

```
allocate [-hIv] [-B base] [-C CPUs] [-N nodes | all] [-n CPUs] [-P project]
[-p partition] [-R request] [script [args ...]]
```

The most frequently used options to `allocate` are `-N`, which allows you to specify the number of nodes, and `-C`, the number of CPUs per node. The `-N` option takes either a numeric argument specifying the number of nodes to be allocated, or the argument `all` specifying all of the nodes in the partition.

Note that `allocate` requests CPUs whereas `prun` creates processes. The `-n` option to `allocate` requests the total number of CPUs to be allocated whereas the `-n` option to `prun` specifies the number of processes to run. The `-C` option to `allocate` specifies the number of CPUs per node whereas the `-c` option to `prun` specifies the number of CPUs required per process.

3.5.2 Allocating Resources to an Interactive Shell

When `allocate` is run without specifying a shell script as the final argument, it spawns an interactive shell that has the resource allocated to it. The resource is freed when you exit the shell or when a time limit, imposed by the system administrator on parallel jobs, expires; whichever comes first. In the following example, both of the `prun` commands execute concurrently on the partition called `parallel`:

```
$ allocate -N 4 -p parallel
$ prun -n 2 myprog &
$ prun -n 2 test &
$ wait
$ exit
```

The & character at the end of the line tells the shell to run the program in the background and return to the command prompt.

In the next example, the two `prun` commands are executed sequentially, both on the same two nodes in the `parallel` partition:

```
$ allocate -N 2 -p parallel
$ prun hostname
```

```
atlas32
atlas33
$ prun hostname
atlas32
atlas33
$ exit
```

If this example was run without allocating a resource to the shell, you could not guarantee that the second use of `prun` would start immediately after the first completed; nor could you guarantee that both runs would use the same two nodes in the partition.

3.5.3 Verifying Resource Allocation to a Shell

When you allocate a resource to an interactive command shell, you can check that the resource has been successfully allocated by using `rinfo` (see Section 3.3 and Appendix A (*RMS Commands*)) or by echoing the `RMS_RESOURCEID` environment variable as shown in the following example:

```
$ rinfo
MACHINE        CONFIGURATION
atlas          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   atlas[0-2]
parallel        2/4   running    01:02:29             atlas[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  atlas0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  atlas0

$ echo $RMS_RESOURCEID
parallel.996
```

This shows that a resource with the identifier `996` on the `parallel` partition has been allocated to `user` and has been in use for 5 seconds.

Another way of verifying that the resource has been allocated to the shell is to change the shell prompt so that it shows the name of the resource. You can do this by editing the setup file that the shell reads when you log in. The changes for a C shell and Bourne shell setup file are as follows.

Changing the Prompt with the C Shell

Add the following lines to the `.cshrc` file in your home directory to make the shell prompt change whenever you have been allocated resources:

```
# additions to .cshrc
if ($RMS_RESOURCEID) then
  set prompt="${RMS_RESOURCEID}:"
else
  set prompt="$user@‘hostname‘:"
endif
```

When you next log in and run `allocate` to get an interactive shell with a resource allocated to it, the prompt will be the name of the resource (as shown by `rinfo`, prefixed by the partition name):

```
$ allocate -p parallel -n 6
parallel.4: exit
$
```

Changing the Prompt with bash

Add these lines to the `.bashrc` file in your home directory to make the shell prompt change whenever you have been allocated resources:

```
# additions to .bashrc
if [ $RMS_RESOURCEID ] ; then
  PS1="$RMS_RESOURCEID:"
else
  PS1="$user@‘hostname‘:" "
fi
```

When you next log in and run `allocate` to get an interactive shell with a resource allocated to it, the prompt will be the name of the resource (as shown by `rinfo`, prefixed by the partition name):

```
$ allocate -p parallel -n 6
parallel.5: exit
$
```

3.5.4 Allocating Resources to a Shell Script

Rather than allocate a resource to an interactive shell, as just described, you can allocate it to a shell script. Calls to `prun` within this shell script result in the execution of parallel programs on the allocated resource.

In the following example, the shell script called `script` uses `prun` to run four user jobs, one after the other:

```
$ cat script
#!/bin/sh
# script to run
```

```
prun preprocess
prun iterate -f12000
prun iterate -f24000
prun postprocess
```

`allocate` is used to get a resource (8 nodes from the `parallel` partition), to assign the resource to the shell script and to run the script:

```
$ allocate -N 8 -p parallel script
```

3.6 Running a Sequential Program with rmsexec

`rmsexec` provides a mechanism for running sequential processes on lightly loaded nodes – nodes, for example, with free memory or low CPU usage. Note that this load balancing service may not be available on all partitions. It is a configuration option, selected by the system administrator.

3.6.1 Command Line Options

`rmsexec` has a number of options that enable you to influence the choice of node. See Appendix A (*RMS Commands*) for full details.

```
rmsexec [-hv] [-p partition] [-s stat] [hostname] program [args ...]
```

Use the `-h` option to get a list of the available options and valid arguments.

3.6.2 Selecting a Node

`rmsexec` restricts its search for a lightly loaded node to the partitions you are entitled to use (as defined by the system administrator). You can restrict the search still further by naming a particular partition with the `-p` option, as shown in the following example:

```
$ rmsexec -p parallel myseqprog
```

You can also request a processor on a specific node. The following example requests the node `atlas2`:

```
$ rmsexec atlas2 myseqprog
```

3.6.3 Defining Load

To specify the criterion for judging load, use the `-s` option. There are five statistics that can be applied:

`usercpu`	The percentage of CPU time spent in the user state.
`syscpu`	The percentage of CPU time spent in the system state – a measure of the I/O load on a node.
`idlecpu`	The percentage of CPU time spent in the idle state.
`freemem`	The amount of free memory in MB.
`users`	The number of users.

By default, the `usercpu` statistic is used. Statistics can be used on their own, in which case a node is chosen that is lightly loaded according to this statistic. Alternatively, you can specify a threshold as shown in the following examples:

```
$ rmsexec -s usercpu myseqprog
$ rmsexec -s "usercpu < 50" myseqprog
$ rmsexec -s "freemem > 256" myseqprog
```

4

MPI and Shmem Programming

4.1 Introduction

Programs may be run in parallel using either the MPI (Message Passing Interface) library or the Shmem (put/get) library for process synchronization and communications. This chapter introduces you to MPI and Shmem programming on the HP AlphaServer SC, demonstrating how to compile and link programs and run them under RMS. It also describes how to run programs with the TotalView debugger and the Vampir visualization and analysis tool. Two example programs are provided. Both show a simple `ping` application (for measuring interprocess communication latency and bandwidth): one in each programming style.

The information in this chapter is organized as follows:

- MPI overview (Section 4.2)
- Shmem overview (Section 4.3)
- Using TotalView to debug MPI programs (Section 4.4)
- Using Vampir to analyze MPI programs (Section 4.5)
- Using the MPI library to implement the example program (Section 4.6)
- Using the Shmem library to implement the example program (Section 4.7)

4.2 MPI Overview

This section provides an overview of the MPI library. The information is organized as follows:

- Introduction to MPI (Section 4.2.1)
- Compiling, linking and running MPI programs (Section 4.2.2)
- Further sources of information on MPI (Section 4.2.4)

4.2.1 Introduction to MPI

The MPI library is a standard message passing library for parallel applications. Using MPI, parallel processes cooperate to perform their task by passing messages to each other. MPI includes point-to-point message passing and collective operations between a user-defined group of processes.

Processes identify each other according to their *rank* in the group. The rank is an integer in the range `0` to `n-1`, where `n` is the total number of processes in the program. A process can query its rank and the size of its group.

The initial group of processes includes all of the processes in the program and is known as the *world* group. The world group may be subdivided into subgroups. Processes can be identified according to their rank in the subgroup. In this way, virtual topologies, such as graphs, can be created that map directly onto the application domain.

A *communicator* is a higher-level grouping construct that contains a group and a communications context (scoping information).

Each message passing routine has four variables that can be used to synchronize the sender and receiver: the sender's rank, the receiver's rank, a user-defined tag and the communications context. The following example shows a send routine. The sender's rank is implicit.

```
MPI_Send(txbuf, nob, MPI_BYTE, receiver, tag, MPI_COMM_WORLD);
```

`MPI_Comm_World` is a communicator that contains all of the processes in the world group. This communicator is set up for the process when it is initialized for MPI. `MPI_BYTE` specifies the datatype of the message data. MPI performs data conversion transparently and supports both built-in and user-defined datatypes.

The receive routine includes a status argument, used to determine the success of the operation. Wildcards can be used for the sender's rank and the tag.

```
MPI_Recv(rxbuf, nob, MPI_BYTE, sender, tag, MPI_COMM_WORLD, &status);
```

The message passing routines support the following:

- Blocking (synchronous), point-to-point send and receive
- Non-blocking (asynchronous), point-to-point send and receive
- Collective message passing operations derived from the four primitives: broadcast, scatter, gather and reduce

In addition to the communications routines, MPI provides the following categories of service:

- Environmental queries (process rank, number of processes, etc.)
- Timing information for measuring performance
- Profiling information for monitoring performance

The MPI library is layered on top of the tagged message passing routines provided by the Elan library. These routines make use of tagged message ports, known as *TPORT*s, for point-to-point communications.

On a SMP node, the tagged message passing routines (and, hence, MPI) use shared memory segments to communicate between processes on the same node, and the HP AlphaServer SC Interconnect to communicate between nodes. There are a number of parameters that you can tune by setting environment variables to help to optimize the performance of your MPI programs. For further details, see Appendix E (*Elan Library Environment Variables*).

The MPI library is based on MPICH release 1.2.4 with the addition of MPI-2 one-sided support and optimized collectives. See Appendix C (*MPI Library Routines*) for details.

4.2.2 Compiling, Linking and Running MPI Programs

This section describes how to compile, link, and run MPI programs written in C and Fortran.

Compiling and Linking C MPI Programs

To build C MPI programs, use the MPI header file in your source files and specify the MPI compiler in your `make` files, as follows:

1. Include the MPI header file, `mpi.h`, in your program, with the following include directive:

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

2. Compile the program, linking it with the MPI and Elan libraries.

```
$ cc -o myprog myprog.c -lmpi -lelan
```

Compiling and Linking Fortran MPI Programs

To build Fortran MPI programs, use the MPI header file in your source files and specify the MPI compiler in your `make` files, as follows:

1. Include the MPI header file, `mpif.h`, in your program with the following include directive:

```
INCLUDE 'mpif.h'
```

2. Compile the program with `f77` or `f90` and link with the MPI and Elan libraries.

```
$ f90 -o myprog myprog.f -lfmpi -lmpi -lelan
```

Running MPI Programs

To execute an MPI program enter the RMS command `prun`, followed by the name of the program:

```
$ prun -n 4 myprog
```

The `-n` flag instructs RMS to start four copies of `myprog`. For more information on `prun`, see Section 3.4 and Appendix A (*RMS Commands*).

4.2.3 MPI Examples

Working through a minimal example in C, for example

```
#include "mpi.h"
main(int argc, char **argv)
{
    int proc, nproc;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &proc);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    printf("process %d of %d\n", proc, nproc);

    MPI_Finalize();
}
```

```
$ cc -o mpi1 mpi1.c -lmpi
$ prun -N4 ./mpi1
```

```
process 0 of 4
process 1 of 4
process 2 of 4
process 3 of 4
```

and in Fortran

```
        program mpi2
        include 'mpif.h'
        integer ierr, proc, procs, rc

        call MPI_INIT(ierr)
        call MPI_COMM_RANK( MPI_COMM_WORLD, proc, ierr )
        call MPI_COMM_SIZE( MPI_COMM_WORLD, procs, ierr )

        print *,"process ", proc, " of ", procs

        call MPI_FINALIZE(rc)
        end
```

```
$ f90 -o mpi2 mpi2.f -lmpi
$ prun -N4 ./mpi2
 process            2  of            4
 process            0  of            4
 process            1  of            4
 process            3  of            4
```

4.2.4 Further Information on MPI

You can find more details about the MPI library from the following Web site:

http://www.mcs.anl.gov/mpi/index.html

Section 4.4 describes how to use TotalView to debug MPI programs on HP AlphaServer SC systems. Section 4.5 describes how to use Vampir to trace and analyze MPI programs on HP AlphaServer SC systems.

4.3 Shmem Overview

This section provides an overview of the Shmem library. The information is organized as follows:

- Introduction to Shmem (Section 4.3.1)
- Compiling, linking and running Shmem programs (Section 4.3.2)
- Further sources of information on Shmem (Section 4.3.4)

4.3.1 Introduction to Shmem

The Shmem library provides direct access (through `put` and `get` calls) to the memory of remote processes. A message passing library, such as MPI, requires that the remote process issues a receive to complete the transmission of each message. The Shmem library, by contrast, provides the initiating process with direct access to the target memory. The one-sided communication used by Shmem maps well onto the DMA hardware in the Elan network adapter. A consequence of this is that Shmem communications latencies are very low.

Shmem provides the following categories of routine:

- Put routines write data to another process.
- Get routines read data from another process.
- Collective routines distribute work across a set of processes.
- Atomic routines perform an atomic fetch-and-operate, such as fetch-and-increment or swap.
- Synchronization routines order the actions of processes. For instance, the `barrier` routine might be used to prevent one process from accessing a data location before another process has updated that location. The Shmem programming model requires that you think about the synchronization points in your application and the communication that must go on between them.
- Reduction routines reduce an array to a scalar value by performing a cumulative operation on some or all of the array elements. For example, a summation is a reduction that adds all of the elements of an array together to yield one number.

The Shmem library also includes a number of initialization and management routines. See Appendix D (*Shmem Library Routines*) for further information on the Shmem routines supported.

Shmem routines provide high performance by minimizing the overhead associated with data passing requests, maximizing bandwidth and minimizing data latency (the time from when a process requests data to when it can use the data). By performing a direct memory-to-memory copy, Shmem typically takes less steps to perform an operation than a message passing system. For example, in a generic message passing system, for a put operation the sender performs a send, then the receiver performs a receive; for a get operation, the requesting process sends a description of the data required, the sender acts on the request by sending the data, then the requesting process receives the data. By contrast, Shmem requires only one step: either send the data or get the data. However, additional synchronization steps are almost always required when using

Shmem. For example, the programmer must ensure that the receiving process does not try to use the data before the updated copy is available.

4.3.2 Compiling, Linking and Running Shmem Programs

This section describes how to compile, link, and run Shmem programs written in C and Fortran.

Compiling and Linking C Shmem Programs

To compile C Shmem programs, use the Shmem header file in your source files and specify the Shmem library in your `make` files, as follows:

1. Include the Shmem header file, `<shmem.h>`, in your program, with the following include directive:

   ```
   #include <shmem.h>
   ```

2. Specify the `shmem` library to the linker with the `-l` option on the command line:

   ```
   $ cc -o myprog myprog.c -lshmem -lelan
   ```

Compiling and Linking Fortran Shmem Programs

To compile Fortran Shmem programs, use the Shmem header file in your source files and specify the Shmem library in your `make` files, as follows:

1. Include the Shmem header file, `shmem.fh`, in your program with the following include directive:

   ```
   INCLUDE 'shmem.fh'
   ```

2. Specify the `shmem` library to the linker with the `-l` option on the command line:

   ```
   $ f77 -o myprog myprog.f -lshmem -lelan
   ```

Running Shmem Programs

To execute a Shmem program, enter the RMS command `prun`, followed by the name of the program:

```
$ prun -n 4 myprog
```

The `-n` flag instructs RMS to start four copies of `myprog`. For more information on `prun`, see Section 3.4 and Appendix A (*RMS Commands*).

4.3.3 Shmem Examples

Working through a minimal example in C, for example

```
#include "shmem.h"
main(int argc, char **argv)
{
    shmem_init();
    printf("process %d of %d\n", my_pe(), num_pes());
}
```

```
$ cc -o sh1 sh1.c -lshmem
$ prun -N4 ./sh1
process 0 of 4
process 1 of 4
process 2 of 4
process 3 of 4
```

and in Fortran, for example

```
        program sh1
        include 'shmem.fh'
        call shmem_init()
        print *,'process ', my_pe(), ' of ', num_pes()
        end
```

```
$ f90 -o sh2 sh2.f -lshmem
$ prun -N4 ./sh2
 process            3  of           4
 process            1  of           4
 process            0  of           4
 process            2  of           4
```

4.3.4 Further Information on Shmem

For more information about Shmem, see the *Quadrics Shmem Programming Manual*. This document can be downloaded from the Quadrics web site http://www.quadrics.com.

4.4 Using TotalView

TotalView is the recommended source-level debugger for HP AlphaServer SC systems. Versions 5 and 6 of TotalView, licensed from Etnus LLC, have been integrated with RMS and the Quadrics MPI library. It has an easy-to-use interface and support for debugging parallel programs. You can find more details about Etnus from the following Web site:

http://www.etnus.com

TotalView runs on the same node as `prun`. It starts a remote server process called the TotalView Debugger Server, `tvdsvr`, on each of the nodes used by your parallel program.

TotalView allows you to select which of your processes to inspect. Each is displayed in its own window together with the source code, status, program counter, threads, breakpoints, stack trace and stack frame.

TotalView cooperates with RMS to perform the following functions:

- Acquire the processes spawned by `prun` at startup, before they have entered the main program.
- Attach to a parallel job already started by `prun` and acquire all of the processes in the job, wherever they reside in the machine.

TotalView is integrated with the Quadrics MPI implementation allowing the inspection of MPI message queues from the debugger.

Before using TotalView, update your `PATH` environment variable, as described in Chapter 3 of the *TotalView User's Guide*.

4.4.1 Running a Parallel Job under TotalView Control

This section describes how to run a parallel job under the control of TotalView.

1. Ensure that you can create X Windows on your local workstation. This normally requires that you set the `DISPLAY` environment variable to `workstation:0.0` where `workstation` is the name of the workstation on your desk. You may also need to grant permission to create such windows by running the `xhost` command on your workstation.
2. To start a parallel job under the control of TotalView, use the following command:

   ```
   $ totalview prun -a arguments
   ```

   where *`arguments`* are the command line arguments for `prun`, as in the following example:

   ```
   $ totalview prun -a -n 2 myprog 0 1
   ```

   The `-a` option is a TotalView option which specifies that the arguments which follow are for the program TotalView is running. The program is specified by the first argument to TotalView. For information about `prun`, see Section 3.4 and Appendix A (*RMS Commands*).
3. Select the following TotalView option:

   ```
   Go/Halt/Step/Next/Hold -> Go Process (g)
   ```

When `prun` has acquired the resources to execute the job, TotalView starts its remote servers on the appropriate nodes by using its remote server startup mechanism (see *Starting the Debugger Server for Remote Debugging* in Chapter 4 of the *TotalView User's Guide*).

After the remote servers have started, TotalView acquires the processes that make up the parallel job.

4. TotalView prompts you to indicate whether you want to stop the processes before they enter the main program. Choose one of the following options:

   Stop the processes

   Choose this option if you have not saved a breakpoint file for the current program and you want to set breakpoints before the program runs.

   Let the processes run

   If you have run a program and it has crashed, then running under TotalView and choosing this option will cause the program to crash again, except that, this time, TotalView will show you where the program failed.

4.4.2 Attaching to an Executing Parallel Job

Use the following procedure to attach to a parallel job that is already executing:

1. Start TotalView without using any arguments, as follows:

   ```
   $ totalview
   ```

2. In the root window, select the following option:

   ```
   Show All Unattached Processes (N)
   ```

   The unattached processes window is displayed, showing a list of processes to which you can attach.

3. Select the `prun` process in this window, to attach to the parallel job as a whole.

4.4.3 Restarting a Parallel Job

Restarting a program is faster than the initial program startup. This is because the TotalView servers remain in place and do not have to be restarted. You can kill a job and restart it from the beginning as follows:

1. Select the following option:

```
Arguments/Create/Signal -> Restart Program
```

The initial `prun` process and all of the parallel processes are terminated. The `prun` process is restarted to spawn the parallel program again.

2. If you want to preserve breakpoints in your code, select the following option to save them to a file before you restart the program:

```
STOP/BARR/EVAL/GIST -> Save All Action Points
```

TotalView automatically reloads the breakpoints when it restarts the program.

4.4.4 TotalView Bulk Launch Support

By default TotalView will use `rsh(1)` to start its remote server process on each node used by a parallel program. This option is not available on some systems and is inefficient on others. An alternative is to use the TotalView bulk launch mechanism as follows:

1. Start TotalView in the normal way, for example:

```
$ totalview prun -a -N16 mping
```

2. Select the `Preferences` option from the TotalView `File` menu. Within this select the `Bulk Launch` tab.
3. Enable the `Bulk Launch` option
4. Enter the following string in `Command` field (if it is not already present):

```
prun -T -1 tvdsvr -working_directory %D -callback_host %H -callback_ports %L \
                  -set_pws %P -verbosity %V
```

5. Save your new preferences. This will return you to the TotalView process window from where you can start the parallel program in the normal way.

4.4.5 Problems and Limitations

TotalView starts up remote servers on each node on which your parallel program runs. By default, it uses `rsh(1)` when it starts these servers. You must ensure that you can `rsh` to the nodes on which your parallel program runs, for this to work.

You can change the command used to start the remote servers, using the `Server Launch Window` command in the TotalView root window. See the section *Changing the Option* in Chapter 4 of the *TotalView User's Guide*.

4.5 Using Vampir

You can use Vampir and Vampirtrace to prepare, build, trace and analyze MPI programs running on HP AlphaServer SC systems. Vampir is a visualization and analysis tool for MPI programs. Vampirtrace is the Vampir MPI profiling library.

The Vampir software is available from Pallas GmbH. You can find more details about Pallas from the following Web site:

http://www.pallas.com

After installing Vampir, link your MPI program with Vampirtrace. When you run the program, Vampirtrace uses the MPI profiling interface to gather information about the program's execution behavior. The information is kept locally in each processor's memory and saved in a trace file when the program exits. The trace file is then fed into Vampir for analysis.

4.5.1 Preparing to Use Vampir

To use Vampir, set up the following environment variables:

1. Set the environment variable PAL_ROOT to the directory where you have installed the Vampir kits and licenses, for example:

   ```
   $ setenv PAL_ROOT /usr/local/vampir
   ```

2. Set the environment variable DISPLAY to the node on which you want Vampir to display its graphics. Make sure that you have permission to display on that node.

4.5.2 Linking and Tracing a Program

Use the following steps to link and trace your program:

1. Link your program with the Vampirtrace library and the MPI profiling interface. For C programs, the command line is as follows:

   ```
   $ cc -o test test.o -L/usr/local/vampir/lib/lib -1VT -lpmpi -lmpi -lelan
   ```

   For Fortran programs, the command line is as follows:

   ```
   $ f77 -o test -test.o -L/usr/local/vampir/lib/lib -lfmpi -lVT -lpmpi -lmpi -lelan
   ```

2. Run your program to generate a trace file for Vampir to use:

   ```
   $ prun -N 2 test
   ...
   Writing logfile test.bpv
   Finished writing file
   ```

The trace file has the same name as the program with a `.bpv` suffix, for example, `test.bpv`.

3. If you are running the Vampir viewer on a local workstation then copy back the trace file with `ftp` or `scp`.
4. Use Vampir to view and analyze the trace, as follows:

```
$ vampir test.bpv
```

4.6 MPI Example

The `mping` program uses the MPI library to synchronize its processes and to perform interprocess communications.

The C source code for this example is in the file `mping.c` in the directory `/usr/opt/rms/examples`.

4.6.1 MPI Functions

The following functions from the MPI library are used and the header file `mpi.h` is included to declare them.

1. `MPI_Init()` initializes the process to use the library.
2. `MPI_Comm_rank()` establishes the rank or number of the process within the set of parallel processes.
3. `MPI_Comm_size()` determines the number of processes in the parallel program.
4. `MPI_Barrier()` synchronizes all of the processes.
5. `MPI_WTime()` reads the value of a timer, which counts in seconds.
6. `MPI_Recv()` receives a message.
7. `MPI_Send()` sends a message.

We will look at these functions more closely when we see them in context.

4.6.2 Command Line Interface

This is the command line interface for the program, `mping`.

```
mping -n number[k|K|m|M] [-eh] nob [maxNob [incNob]]
```

The options for the programs are as follows:

`-n` *`number`*`[k|K|m|M]`

Specifies the number of times to ping. The *`number`* may have a `k` or an `m` appended to it (or their uppercase equivalents) to denote multiples of 1024 and 1,048,576 respectively. By default, the program pings 10,000 times.

`-e` Instructs every process to print its timing statistics.

`-h` Displays the list of options.

`nob` `[`*`maxNob`* `[`*`incNob`*`]]`

`nob` specifies to `mping` how many bytes there are in each packet. If *`maxNob`* is given, it specifies a maximum number of bytes to send in each packet and invokes the following behavior. After each *n* repetitions (as specified with the `-n` option), the packet size is increased by `incNob` (the default is a doubling in size) and another set of repetitions is performed until the packet size exceeds `maxNob`. This means that if neither of the optional parameters is specified, only one set of repetitions is performed.

4.6.3 Program Output

At the start of the program, if printing has been enabled for all processes by specifying the `-e` option, a message like this is displayed by each process:

```
$ prun -N 2 ./mping -e
0(2): MPI PING reps 10000 minNob 0 maxNob 0 incNob 0
1(2): MPI PING reps 10000 minNob 0 maxNob 0 incNob 0
```

where `1` is the rank which identifies the process and `2` gives the number of processes running in parallel.

After each set of repetitions, timing statistics are displayed, similar to the following:

```
  1 pinged   0:       64 bytes      6.29 uSec    10.18 MB/s
```

This indicates that process `1` pinged process `0` with 64 byte packets. The pinging operation took 6.29 microseconds (on average) and the transfer rate was 10.18MB per second.

If printing has been enabled for all processes with the `-e` option, this message is displayed by each process. By default, only one process in each pair displays the message.

4.6.4 Header Files and Variables

The header files and variables used by the program are shown here. The variables are declared in `main()`.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>

#include <mpi.h> [1]

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

    double      t, tv[2]; [2]

    MPI_Status  status; [3]
    int         tag = 0x69;
    char        *rbuf, *tbuf;

    int         reps = 10000; [4]
    int         minNob = 1;
    int         maxNob = 1;
    int         incNob, nob;

    int         proc, peer, nproc; [5]

    int         doprint = 0; [6]
    char        *progName;
    int         c, r, i;
    ...
}
```

The header files and variables are used as follows.

[1] Besides the standard C header files, the `mpi.h` header file is required for the MPI library.

[2] The two time variables are used to time each set of repetitions of sending and receiving a message. The `tv` array is used to record two time readings, returned by the MPI function `MPI_Wtime()`:

1. The time before the set of repetitions begins.
2. The time after the set of repetitions has ended.

The variable `t` is used to hold the difference between these two readings, converted to microseconds.

3 The `status` variable is used to determine the status of a message received by a call to `MPI_Recv()`.

The `tag` is used to identify the messages transmitted through the MPI message passing functions. `tbuf` and `rbuf` are pointers to buffers for messages, for which the process allocates space before calling `MPI_Send()` and `MPI_Recv()` to send and receive messages.

4 This group of variables is used to control how many times the process pings its opposite number and the size of packets sent. The variable `reps` is set to the number of repetitions requested with the `-n` option. It has a default setting of 10,000.

The next three variables hold the minimum, maximum and increment values for the packet size. They are used when more than one set of repetitions is requested. The variable `nob` is used to iterate from `minNob` to `maxNob` during a set of repetitions.

5 These variables are used to identify, by means of their rank, the process and its peer (or opposite number), to which it sends packets, and to hold the total number of processes.

6 The variable `doprint` is used to enable (`1`) or disable (`0`) the printing of results by all of the processes.

The `progName` variable is used to extract the name of the program for use with the standard UNIX style `-h` option and `Usage` message, which is displayed when the program is called with the wrong arguments.

The remaining three variables are general purpose iteration variables.

4.6.5 Argument Checking

The first section of `main()` is concerned with checking the arguments passed to the program on the command line.

```
int main(int argc, char *argv[]) {
    ...
    for (progName = argv[0] + strlen (argv[0]);
         progName > argv[0] && *(progName - 1) != '/';
         progName--); 1

    while ((c = getopt (argc, argv, "n:eh")) != -1) 2
        switch (c) {
        case 'n':
```

```
            if ((reps = getSize (optarg)) <= 0)
                usage (progName);
            break;

        case 'e':
            doprint++;
            break;

        case 'h':
            help (progName);

        default:
            usage (progName);
        }

    if (optind == argc) 3
        minNob = 0;
    else if ((minNob = getSize (argv[optind++])) < 0)
        usage (progName);

    if (optind == argc)
        maxNob = minNob;
    else if ((maxNob = getSize (argv[optind++])) < minNob)
        usage (progName);

    if (optind == argc)
        incNob = 0;
    else if ((incNob = getSize (argv[optind++])) < 0)
        usage (progName);
    ...
}
```

1 The program name is passed in as `argv[0]`, the first string on the command line. This string may take the form of a pathname, such as `/home/duncan/mping`. The `progName` variable is set to point to the end of the program name. The loop then steps the variable backwards, one character at a time, until either a file name separator (`/`) or the beginning of the name is reached. This leaves `progName` pointing at the start of the program name.

2 The `while` loop steps through the options given on the command line.

- If the `-n` option has been used, the variable `reps` is set to the requested number of repetitions after a check that the number is greater than `0`. If the number is invalid, the `usage()` function is called. This merely displays the command line syntax for the program and then exits.
- If the `-e` option has been used, the variable `doprint` is

incremented. This variable is used later to enable or disable the printing of statistics.

- The `-h` option calls the `help()` function, which displays the command line syntax for the program and explains the meaning of the various options (or flags), as follows, before exiting:

```
Usage: mping [flags] nob [maxNob] [incNob]
   Flags may be any of:
      -n number         repetitions to time
      -e                everyone print timing info
      -h                print this info

   Numbers may be postfixed with 'k' or 'm'
```

- If any other options besides the three mentioned here are given, the function `usage()` is called to display the correct command line syntax and then exit.

3 The three `if` statements determine whether the optional arguments for specifying a varying packet size have been set. The variable `optind` is defined externally and included by the header files at the start of the program. After stepping through all of the options with the `while` loop, `optind` indexes the first argument in `argv`.

The first argument should be `nob`, the number of bytes in each packet. If the user has not specified this argument, the program continues rather than exiting but assumes a value of `0`. Note that the value is assigned to `minNob` rather than to the variable `nob`. Later on, the value is transferred to `nob` when it acts as an iteration variable.

4.6.6 Initialization

The next section of `main()` is concerned with initializing the process to use the MPI library.

```
int main(int argc, char *argv[]) {
    ...
    MPI_Init(&argc, &argv); 1
    MPI_Comm_rank(MPI_COMM_WORLD, &proc); 2
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    if (nproc == 1)
       exit(1);

    if ((rbuf = (char *)malloc(maxNob ? maxNob : 8)) == NULL) { 3
        perror ("Failed memory allocation");
        exit (1);
    }
```

```
    if ((tbuf = (char *)malloc(maxNob ? maxNob : 8)) == NULL) {
        perror ("Failed memory allocation");
        exit (1);
    }

    for (i = 0; i < maxNob; i++)
        tbuf[i] = i & 255;
    ...
}
```

The initialization process is as follows.

1 The process calls `MPI_Init()` to initialize itself to use the MPI library. The function allocates and initializes a structure referenced by the opaque handle `MPI_COMM_WORLD`. This handle is required as a parameter to the other functions used.

2 The process calls `MPI_Comm_rank()` to determine its rank. The processes are numbered from `0` to `nproc-1` where `nproc` is the number of processes running in parallel, as established by a call to `MPI_Comm_size()`. If there is only one process, the program exits as there is no other process that it can ping.

3 The process allocates memory for the two message buffers using `malloc()`. The buffers are used as the source and destination of the messages that are transferred across the network. Pointers to them are passed to the MPI message passing library functions.

If a maximum number of bytes for the packet size is specified on the command line to `mping`, the process allocates a buffer of this size. By default, the buffers hold 8 bytes. The transmit buffer is initialized by writing a sequence of numbers to it.

4.6.7 Establishing the Peer Group

Before starting the first (and possibly only) set of repetitions, the processes must synchronize and group themselves into pairs.

```
int main(int argc, char *argv[]) {
    ...
    if (doprint) 1
        printf ("%d(%d): MPI PING reps %d"
                "minNob %d maxNob %d incNob %d\n",
                proc, nproc, reps, minNob, maxNob, incNob);

    MPI_Barrier(MPI_COMM_WORLD); 2
```

```
    peer = proc ^ 1; [3]
    if (peer >= nproc)
        doprint = 0;
    ...
}
```

[1] If all of the processes have been enabled for printing with the `-e` option, each prints a message to confirm its identity, the number of processes in the program and the program parameters.

[2] Before starting to ping each other, the processes synchronize, that is, each waits in the call to `MPI_Barrier()` until all have made the call. This guarantees that all of the processes are initialized and ready to send and receive messages before any one of them starts to ping another.

[3] To ping each other, the processes split up into pairs. Each process determines its opposite number or `peer` simply by an exclusive-OR of its own rank with the constant `1`. With an uneven number of processes, one will have no peer. This can be determined by checking that the peer's rank is in the valid range. This singleton is disabled from printing.

4.6.8 Sending Messages

In the final section of `main`, the process pings its peer a given number of times using the MPI message passing functions.

```
int main(int argc, char *argv[]) {
   ...
   for (nob = minNob;
        nob <= maxNob;
        nob = incNob ? nob + incNob : nob ? 2 * nob : 1) { [1]

        r = reps;

        MPI_Barrier(MPI_COMM_WORLD); [2]

        tv[0] = MPI_Wtime();

        if (peer < nproc) {   [3]

            if (proc & 1) {

                r--;
                MPI_Recv(rbuf, nob, MPI_BYTE, peer, tag,
```

```
                MPI_COMM_WORLD, &status);
        }

        while (r-- > 0) { 4

            MPI_Send(tbuf, nob, MPI_BYTE, peer, tag,
                    MPI_COMM_WORLD);

            MPI_Recv(rbuf, nob, MPI_BYTE, peer, tag,
                    MPI_COMM_WORLD, &status);
         }

         if (proc & 1) {

             MPI_Send(tbuf, nob, MPI_BYTE, peer, tag,
                     MPI_COMM_WORLD);
         }
    }

    tv[1] = MPI_Wtime(); 5
    t = dt (&tv[1], &tv[0]) * 1000000.0/ (2 * reps);

    MPI_Barrier(MPI_COMM_WORLD);
    printStats (proc, peer, doprint, nob, t);
}

MPI_Barrier(MPI_COMM_WORLD); 6
return (0);
}
```

The MPI library message passing and interval timing functions are described here.

1 The `for` loop controls how many sets of repetitions are performed. In each set of repetitions, a message containing `nob` bytes is sent from one process to its peer for the number of times specified by `reps`.

The first time through the loop, `nob` is set to `minNob`. This was initialized earlier (see Section 4.6.5) to the value the user entered for `nob` on the command line (by default, `0`).

On subsequent iterations, the value of `nob` is incremented by the value of `incNob`. If no value was specified for `incNob` on the command line, the original value of `nob` is doubled or, if `nob` was unspecified, it is set to `1`.

If the user specified `maxNob`, the `for` loop is iterated until `nob` exceeds the value of `maxNob`. If not, the loop is only executed once.

2 Before the processes begin to time how long the ping operation takes, they synchronize using `MPI_Barrier()`. This ensures that they are

all ready to start sending and receiving messages at the same time.

The timing is done by taking two readings using the function `MPI_Wtime()`, which returns the value of a timer in seconds: one reading before the messages start and one when they have finished.

[3] After testing that the process has a peer (this test has to be repeated in here since all of the processes must participate in the synchronization), the message sending can begin. The odd-numbered processes (`proc & 1`) start first by issuing a receive command.

The call to `MPI_Recv()` causes the process to block waiting for the arrival of a message from the sender with a rank of `peer` and with a message tag of `tag`. `MPI_COMM_WORLD` specifies the communications group to which the process and its `peer` belong.

The `rbuf` parameter to `MPI_Recv()` points to a destination buffer for the message while `nob` specifies its size in units with a datatype of `MPI_BYTE`.

The `status` parameter is filled in on return to indicate details of the transfer. In this application, the value of `status` is not checked.

[4] In the `while` loop, both the odd-numbered and even-numbered processes send a message and then wait for a reply, decrementing the number of repetitions, `r`, each time. The call to `MPI_Send()` specifies:

- A source buffer and its size in units of datatype `MPI_BYTE`.
- The rank of the destination process (`peer`). Note that the Elan adapter takes care of all routing details; the sender does not have to know where in the network the receiver resides.
- A message tag to identify the message.
- The `MPI_COMM_WORLD` parameter specifies the group to which the process and its peer belong.

The process blocks in the call to `MPI_Send()` until the message has been received.

Finally, the odd-numbered processes send the last message in the sequence of repetitions. By making the odd-numbered processes request a receive to begin with while the even-numbered processes send a message, deadlock is avoided.

[5] After the set of repetitions, the process reads the timer again. It calculates the time taken for one ping in each direction (the difference between the two timer readings divided by the number of repetitions).

This value is converted to microseconds (multiplied by 1,000,000) and halved to get the value for a ping in one direction.

Before the processes print the results, they synchronize again. This means that all of the results are displayed at roughly the same time and the printing does not interfere with the network performance.

6 When the process has come out of the `for` loop, it synchronizes with its peers again before exiting.

4.6.9 Subsidiary Functions

The subsidiary functions make no use of the MPI library.

`getSize()` This function checks whether the user has suffixed the number of repetitions, specified on the command line with the `-n` option, with either a `k` or `K` (for kilobytes) or `m` or `M` (for megabytes). If it finds a suffix, it multiplies the number as appropriate (a left shift by one place multiplies by 2).

`dt` This function returns the difference between its two arguments.

`usage` This function prints out the command line syntax for the program and then exits.

`help` This function prints out the command line syntax for the program and enumerates the various options before exiting.

`printStats` This function displays the timing statistics generated during each set of repetitions. Unless printing is enabled for all processes with the `-e` option, only the odd-numbered processes have their statistics displayed.

4.6.10 Compiling and Running the Program

To compile the program `mping.c`, which uses the MPI library:

```
$ cc -o mping mping.c -lmpi
```

Before we run the program with `prun`, we can find out how many processors are available with `rinfo`, as described in Section 3.3.

```
$ rinfo
MACHINE        CONFIGURATION
atlas          day
```

```
PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              8                                   atlas[0-3]
parallel        2/8   running    05:00:12             atlas[0-3]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.48       2 allocated       00:15      duncan  atlas0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.259      2   running       00:04      duncan  atlas0
```

Here we see that the partition called `parallel` is active. There are eight processors in this partition but two of them are allocated to the user called `duncan` who is running a job identified as `259` on the `parallel` partition. This leaves six processors free.

Using the command `prun`, we can get two of these processors allocated to us and run `mping` on each of them. It prints the time taken per communication step in microseconds (uSec):

```
$ prun -p parallel -N 2 mping
  1 pinged   0:        0 bytes      4.54 uSec     0.00 MB/s
```

By giving the `-e` option to `mping`, we can see what the differences in timing are between the two pairs of processes. If four nodes were available, we could request that the program be run one process per node with the `-N` option to `prun`.

```
$ prun -p parallel -N 4 mping -e
3(4): MPI PING reps 10000 minNob 0 maxNob 0 incNob 0
0(4): MPI PING reps 10000 minNob 0 maxNob 0 incNob 0
1(4): MPI PING reps 10000 minNob 0 maxNob 0 incNob 0
2(4): MPI PING reps 10000 minNob 0 maxNob 0 incNob 0
  0 pinged   1:        0 bytes      4.48 uSec     0.00 MB/s
  2 pinged   3:        0 bytes      4.49 uSec     0.00 MB/s
  1 pinged   0:        0 bytes      4.48 uSec     0.00 MB/s
  3 pinged   2:        0 bytes      4.49 uSec     0.00 MB/s
```

At this point, you might like to experiment with running the program on different combinations of nodes. You will see differences in latency and bandwidth depending upon whether the communicating processes are on the same node or different nodes.

4.7 Shmem Example

The `sping` program uses the Shmem library routines to synchronize its processes and to perform interprocess communications.

The C source code for this example is in the file `sping.c` in the directory `/usr/opt/rms/examples`.

4.7.1 Shmem Functions

The following functions from the Shmem library are used and the header file `shmem.h` is included to declare them:

1. `shmem_init()` initializes the process to use the library.
2. `shmem_barrier_all()` synchronizes all of the processes.
3. `shmem_put()` writes data to another process.
4. `shmem_wait()` waits until the value of a flag, supplied as an argument, changes.
5. `my_pe()` returns the rank or number of the process within the set of parallel processes.
6. `num_pes()` returns the number of processes in the program.

4.7.2 Command Line Interface

This is the command line interface for the program, `sping`.

```
sping -n number[k|K|m|M] [-eh] nwords [maxWords [incWords]]
```

The options for the programs are as follows:

`-n` *`number`*`[k|K|m|M]`

Specifies the number of times to ping. The *`number`* may have a `k` or an `m` appended to it (or their uppercase equivalents) to denote multiples of 1024 and 1,048,576 respectively. By default, the program pings 10,000 times.

`-e` Instructs every process to print its timing statistics.

`-h` Displays the list of options.

`nwords` `[`*`maxWords`* `[`*`incWords`*`]]`

`nwords` specifies to `sping` how many words there are in each packet. If *`maxWords`* is given, it specifies a maximum number of words to send in each packet and invokes the following behavior. After each *n* repetitions (as specified with the `-n` option), the packet size is increased by `incWords` (the default is a doubling in size) and another set of repetitions is performed until the packet size exceeds `maxWords`. This means that if neither of the optional parameters is specified, only one set of repetitions is performed.

4.7.3 Program Output

At the start of the program, if printing has been enabled for all processes by specifying the `-e` option, a message like this is displayed by each process:

```
1(8): Shmem PING reps 10000 minWords 1 maxWords 256 incWords 0
```

where `1` is the rank which identifies the process and `8` gives the number of processes running in parallel.

After each set of repetitions, timing statistics are displayed, similar to the following:

```
  0:        4 bytes      2.73 uSec     1.46 MB/s
  0:        8 bytes      2.69 uSec     2.97 MB/s
  0:       16 bytes      2.79 uSec     5.74 MB/s
  0:       32 bytes      2.73 uSec    11.74 MB/s
  0:       64 bytes      2.81 uSec    22.74 MB/s
  0:      128 bytes      3.48 uSec    36.73 MB/s
  0:      256 bytes      3.83 uSec    66.83 MB/s
```

The first line indicates that process `1` pinged process `0` with 1-word packets. The operation took 2.73 microseconds on average.

If printing has been enabled for all processes with the `-e` option, this message is displayed by each process. By default, only one process in each pair displays the message.

4.7.4 Header Files and Variables

The header files and variables used by the program are shown here. The variables are declared in `main()`.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/time.h>

#include <shmem.h> 1

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

    double      t, tv[2]; 2

    long       *rbuf, *tbuf; 3

    int         reps = 10000; 4
```

```
    int         minWords = 1;
    int         maxWords = 1;
    int         incWords, nwords;

    int         proc, peer, nproc; 5

    int         doprint = 0; 6
    char        *progName;
    int         c, r, i;
    ...
}
```

The header files and variables are used as follows:

1 Besides the standard C header files, the `shmem.h` header file is required for the Shmem library.

2 The two time variables are used to time each set of repetitions of sending and receiving data. The `tv` array is used to record two time readings:

1. The time before the set of repetitions begins.
2. The time after the set of repetitions has ended.

The variable `t` is used to hold the difference between these two readings.

3 `tbuf` and `rbuf` are pointers to the data buffers, for which the process allocates space before calling `shmem_put()` and `shmem_wait()`.

4 This group of variables is used to control how many times the process pings its opposite number and the size of packets sent. The variable `reps` is set to the number of repetitions requested with the `-n` option. It has a default setting of 10,000.

The next three variables hold the minimum, maximum and increment values for the packet size. They are used when more than one set of repetitions is requested. The variable `nwords` is used to iterate from `minWords` to `maxWords` during a set of repetitions.

5 These variables are used to identify, by means of their rank, the process and its peer (or opposite number), to which it sends packets, and to hold the total number of processes.

6 The variable `doprint` is used to enable (`1`) or disable (`0`) the printing of results by all of the processes.

The `progName` variable is used to extract the name of the program for use with the standard UNIX style `-h` option and `Usage` message, which is displayed when the program is called with the wrong arguments.

The remaining three variables are general purpose iteration variables.

4.7.5 Argument Checking

The first section of `main()` is concerned with checking the arguments passed to the program on the command line.

```
int main(int argc, char *argv[]) {
    ...
    for (progName = argv[0] + strlen (argv[0]);
         progName > argv[0] && *(progName - 1) != '/';
         progName--); 1

    while ((c = getopt (argc, argv, "n:eh")) != -1) 2
        switch (c) {
        case 'n':
            if ((reps = getSize (optarg)) <= 0)
                usage (progName);
            break;

        case 'e':
            doprint++;
            break;

        case 'h':
            help (progName);

        default:
            usage (progName);
        }

    if (optind == argc) 3
        minWords = 1;
    else if ((minWords = getSize (argv[optind++])) < 0)
        usage (progName);

    if (optind == argc)
        maxWords = minWords;
    else if ((maxWords = getSize (argv[optind++])) < minWords)
        usage (progName);

    if (optind == argc)
        incWords = 0;
    else if ((incWords = getSize (argv[optind++])) < 0)
```

```
        usage (progName);
    ...
}
```

[1] The program name is passed in as `argv[0]`, the first string on the command line. This string may take the form of a pathname, such as `/home/duncan/sping`. The `progName` variable is set to point to the end of the program name. The loop then steps the variable backwards, one character at a time, until either a file name separator (`/`) or the beginning of the name is reached. This leaves `progName` pointing at the start of the program name.

[2] The `while` loop steps through the options given on the command line.

- If the `-n` option has been used, the variable `reps` is set to the requested number of repetitions after a check that the number is greater than `0`. If the number is invalid, the `usage()` function is called. This merely displays the command line syntax for the program and then exits.
- If the `-e` option has been used, the variable `doprint` is incremented. This variable is used later to enable or disable the printing of statistics.
- The `-h` option calls the `help()` function, which displays the command line syntax for the program and explains the meaning of the various options (or flags), as follows, before exiting:

```
Usage: sping [flags] nwords [maxWords] [incWords]
   Flags may be any of:
      -n number         repetitions to time
      -e                everyone print timing info
      -h                print this info

   Numbers may be postfixed with 'k' or 'm'
```

- If any other options besides the three mentioned here are given, the function `usage()` is called to display the correct command line syntax and then exit.

[3] The three `if` statements determine whether the optional arguments for specifying a varying packet size have been set. The variable `optind` is defined externally and included by the header files at the start of the program. After stepping through all of the options with the `while` loop, `optind` indexes the first argument in `argv`.

The first argument should be `nwords`, the number of words in each packet. If the user has not specified this argument, the program

continues rather than exiting but assumes a value of 1. Note that the value is assigned to minWords rather than to the variable nwords. Later on, the value is transferred to nwords when it acts as an iteration variable.

4.7.6 Initialization

The next section of main() is concerned with initializing the process to use the Shmem library.

```
int main(int argc, char *argv[]) {
    ...
    shmem_init(); 1

    proc = my_pe(); 2
    nproc = num_pes();

    if (nproc == 1)
       exit(1);

    if (!(rbuf = (long *)malloc((maxWords ? maxWords : 1) * sizeof(long)))) { 3
        perror ("Failed memory allocation");
        exit (1);
    }

    if (!(tbuf = (long *)malloc((maxWords ? maxWords : 1) * sizeof(long)))) {
        perror ("Failed memory allocation");
        exit (1);
    }

    for (i = 0; i < maxWords; i++) 4
        tbuf[i] = 1000 + (i & 255);
    *rbuf = 0;
    ...
}
```

The initialization process is as follows.

1 The process calls shmem_init() to initialize itself to use the Shmem library.

2 The process uses the function my_pe() to determine its rank and num_pes() to find out how many processes are running in parallel. Both functions are from the Shmem library. The processes are numbered from 0 to nproc-1, where nproc is established by the call to num_pes. If there is only one process, it exits as there is no other process that it can ping.

3 The process allocates memory for the two data buffers using `malloc()`. The buffers are used as the source and destination of the data being transferred across the network. Pointers to them are passed to the Shmem library communications functions.

If a maximum number of words for the packet size is specified on the command line to `sping`, the process allocates a buffer of this size. By default, the buffers are 1 word in size.

4 The transmit buffer is initialized by writing a sequence of numbers to it, starting at `1000`. The first word of the receive buffer is initialized to `0`. The reason for this is explained in Section 4.7.8.

4.7.7 Establishing the Peer Group

Before starting the first (and possibly only) set of repetitions, the processes must synchronize and group themselves into pairs.

```
int main(int argc, char *argv[]) {
    ...
    if (doprint) 1
        printf ("%d(%d): Shmem PING reps %d"
                "minWords %d maxWords %d incWords %d\n",
                proc, nproc, reps, minWords, maxWords, incWords);

    shmem_barrier_all(); 2

    peer = proc ^ 1; 3
    if (peer >= nproc)
        doprint = 0;
    ...
}
```

1 If all of the processes have been enabled for printing with the `-e` option, each prints a message to confirm its identity, the number of processes in the program and the program parameters.

2 Before starting to ping each other, the processes synchronize, that is, each waits in the call to `shmem_barrier_all()` until all have made the call. This guarantees that all of the processes are initialized and ready to communicate before any of the processes start to communicate.

3 To ping each other, the processes split up into pairs. Each process determines its opposite number or `peer` simply by an exclusive-OR of its own rank with the constant `1`. With an uneven number of

processes, one will have no peer. This can be determined by checking that the peer's rank is in the valid range. This singleton is disabled from printing.

4.7.8 Data Transfer

In the final section of `main`, the process pings its peer a given number of times using the Shmem communications functions.

```
int main(int argc, char *argv[]) {
   ...
   for (nwords = minWords;
        nwords <= maxWords;
        nwords = incWords ? nwords + incWords : nwords ? 2 * nwords : 1) { 1

        r = reps;

        shmem_barrier_all(); 2

        tv[0] = gettime();

        if (peer < nproc) {   3

            if (proc & 1) {
                r--;
                shmem_wait(&rbuf[nwords-1], 0);
                rbuf[nwords-1] = 0;
            }

            while (r-- > 0) { 4

                shmem_put(rbuf, tbuf, nwords, peer);
                shmem_wait(&rbuf[nwords-1], 0);
                rbuf[nwords-1] = 0;
             }

             if (proc & 1) {
                shmem_put(rbuf, tbuf, nwords, peer);
             }
         }

         tv[1] = gettime(); 5
         t = dt (&tv[1], &tv[0]) / (2 * reps);

         shmem_barrier_all();
         printStats (proc, peer, doprint, nwords, t);
    }
    shmem_barrier_all(); 6
    return (0);
}
```

The Shmem library communications functions and the interval timing functions are described here.

[1] The `for` loop controls how many sets of repetitions are performed. In each set of repetitions, `nwords` words of data is sent from one process to its peer for the number of times specified by `reps`.

The first time through the loop, `nwords` is set to `minWords`. This was initialized earlier (see Section 4.7.5) to the value the user entered for `nwords` on the command line (by default, `1`).

On subsequent iterations, the value of `nwords` is incremented by the value of `incWords`. If no value was specified for `incWords` on the command line, the original value of `nwords` is doubled.

If the user specified `maxWords`, the `for` loop is iterated until `nwords` exceeds the value of `maxWords`. If not, the loop is only executed once.

[2] Before the processes begin to time how long the ping operation takes, they synchronize using `shmem_barrier_all()`. This ensures that they are all ready to start communicating at the same time.

The timing is done by taking two readings using the function `gettime()`, which returns the current time in microseconds: one before the data transfer starts and another after it has finished.

[3] After testing that the process has a peer (this test has to be repeated in here since all of the processes must participate in the synchronization), the data transfer can begin. The odd-numbered processes (`proc & 1`) start first by issuing a receive command.

The call to `shmem_wait()` waits until the value in the last word of the buffer pointed to by `rbuf` differs from the second argument, which is set to `0`. During initialization (see Section 4.7.6), the first word of the buffer was set to `0` while the first word of the transmit buffer `tbuf` was set to `1000`.

As soon as `shmem_wait()` returns, the value of the last word in the receive buffer is reset to `0`, ready for the next time.

[4] In the `while` loop, both the odd-numbered and even-numbered processes send data and then wait for a reply, decrementing the number of repetitions, `r`, each time. The call to `shmem_put()` specifies:

- A destination buffer.
- A source buffer.

- The length of the data transfer in words.
- The rank of the destination process. Note that the Elan adapter takes care of all routing details; the sender does not have to know where in the network the destination resides.

The `shmem_put()` routine returns early, it does not wait for the data to arrive (see the *Quadrics Shmem Programming Manual* for details). The call to `shmem_wait()` blocks until the value of the test location changes.

Finally, the odd-numbered processes perform the last in the sequence of repetitions.

5 After the set of repetitions, the process reads the timer again. It calculates the time taken for one ping in each direction (the difference between the two timer readings divided by the number of repetitions). This value (expressed in microseconds) is halved to get the value for a ping in one direction.

Before the processes print the results, they synchronize again. This means that all of the results are displayed at roughly the same time and the printing does not interfere with the network performance.

6 When the process has come out of the `for` loop, it synchronizes with its peers again before exiting.

4.7.9 Subsidiary Functions

The subsidiary functions make no use of the Shmem library.

`getSize()` This function checks whether the user has suffixed the number of repetitions, specified on the command line with the `-n` option, with either a `k` or `K` (for kilobytes) or `m` or `M` (for megabytes). If it finds a suffix, it multiplies the number as appropriate (a left shift by one place multiplies by 2).

`gettime` This function returns the current time in microseconds.

`dt` This function returns the difference between its two arguments.

`usage` This function prints out the command line syntax for the program and then exits.

`help` This function prints out the command line syntax for the program and enumerates the various options before exiting.

`printStats` This functions displays the timing statistics generated during each set of repetitions. Unless printing is enabled for all processes with the `-e` option, only the odd-numbered processes have their statistics displayed.

4.7.10 Compiling and Running the Program

To compile the program `sping.c`, which uses the Shmem library, we use the C compiler as shown below

```
$ cc -o sping sping.c -lshmem
```

Before we run the program with `prun`, we can find out how many processors are available with `rinfo`, as described in Section 3.3.

```
$ rinfo
MACHINE        CONFIGURATION
atlas          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              8                                   atlas[0-3]
parallel        2/8   running    05:00:12             atlas[0-3]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.48       2 allocated       00:15      duncan  atlas0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.259      2   running       00:04      duncan  atlas0
```

Here we see that the partition called `parallel` is active. There are eight processors in this partition but two of them are allocated to the user called `duncan` who is running a job identified as `259` on the `parallel` partition. This leaves six processors free.

Using the command `prun`, we can get two of these processors allocated to us and run `sping` on each of them.

```
$ prun -p parallel -N 2 sping
  0:        4 bytes      2.75 uSec     1.45 MB/s
```

By giving the `-e` option to `sping`, we can see what the differences in timing are between the two pairs of processes. If four nodes were available, we could run the program one process per node by using the `-N` option to `prun`.

```
$ prun -p parallel -N 4 sping -e
1(4): Shmem ping reps 1000 minWords 1 maxWords 1 incWords 0
  1:        4 bytes      2.75 uSec     1.46 MB/s
3(4): Shmem ping reps 1000 minWords 1 maxWords 1 incWords 0
```

Shmem Example

```
  3:        4 bytes      2.74 uSec     1.46 MB/s
2(4): Shmem ping reps 1000 minWords 1 maxWords 1 incWords 0
  2:        4 bytes      2.74 uSec     1.46 MB/s
0(4): Shmem ping reps 1000 minWords 1 maxWords 1 incWords 0
  0:        4 bytes      2.75 uSec     1.46 MB/s
```

A

RMS Commands

A.1 Overview

The RMS user commands are described in alphabetical order in this appendix. They are as follows:

`allocate`	The `allocate` program reserves access to a set of resources either for running multiple tasks in parallel or for running a sequence of commands on the same CPUs.
`prun`	The `prun` program loads and runs parallel programs. It can also run multiple copies of a sequential program.
`rinfo`	The `rinfo` program displays information about the resources available and about the jobs that are running.
`rmsexec`	The `rmsexec` program runs a sequential program on a lightly loaded node.
`rmsquery`	The `rmsquery` program submits SQL queries to the database. The queries can extract information from the database but non-root users cannot update it.

See the *RMS Reference Manual* for details of other RMS commands.

NAME

allocate – Reserves access to CPUs

SYNOPSIS

```
allocate [-hIv] [-B base] [-C CPUs] [-N nodes | all] [-n CPUs]
         [-P project] [-p partition] [-R request]
         [script [args ...]]
```

OPTIONS

`-B` *base*
: Specifies the number (node id) of the base node (the first node to use). By default, the base node is unassigned, leaving the scheduler free to select nodes that are not in use.

`-C` *CPUs*
: Specifies the number of CPUs required per node (default 1).

`-h`
: Displays the list of options.

`-I`
: Allocate CPUs immediately or fail. By default, `allocate` blocks until resources become available.

`-N` *nodes*`|all`
: Specifies the number of nodes to allocate (default 1). To allocate one CPU on each node in the partition, use the argument `all` as follows: `allocate -N all`. Either the `-C` option or the `-n` option can be combined with `-N` but not both.

`-n` *CPUs*
: Specifies the total number of CPUs required.

`-P` *project*
: Specifies the name of the project with which the job should be associated for scheduling and accounting purposes.

`-p` *partition*
: Specifies the target partition from which the resources are to be allocated.

`-R` *request*
: Specify particular resource requests flags. The types of *request* currently supported are as follows:

  `hwbcast=0|1`
  : With a value of 1, this specifies a contiguous range of nodes and constrains the scheduler to queue the request until a contiguous range becomes available.

`immediate=0|1` With a value of 1, this specifies that the request should fail if it cannot be met immediately (this is the same as the `-I` option).

`rails=`*n* In a multirail system, this specifies the number of rails required. For example:

```
$ allocate -N4 -Rrails=2
...
```

`railmask=`*mask* In a multirail system, this specifies which rails are required using a bitmask to select them. Set bit `1«r` in the mask to select rail `r`. Valid values of the rail identifier `r` run from `0` to `R-1` where `R` is the number of rails in the system. For example to select rail one:

```
$ allocate -N2 -Rrailmask=2
...
```

`samecpus=0|1` With a value of 1, this specifies the same set of CPUs on each node, for example, the first two CPUs, 0 and 1, on each node.

Multiple requests can be entered as a comma-separated list, for example, `-R hwbcast=1,immediate=1`.

`-v` Specifies verbose operation.

DESCRIPTION

The `allocate` program allocates resources for subsequent use by the `prun(1)` command. `allocate` is intended for use where a user wants to run a sequence of commands or several programs concurrently on the same set of CPUs.

The `-p`, `-N`, `-C`, `-B` and `-n` options control which CPUs are allocated. The `-N` option specifies how many nodes are to be allocated. When this option is specified the user is allocated a constant number of CPUs per node (default 1). The `-C` option specifies the number of CPUs required per node. The alternative `-n` option specifies the total number of CPUs to allocate. It does not force the allocation of a constant number of CPUs per node.

The `-B` option specifies the base of a range of nodes. The `-N` option specifies its extent. So for example, `-B0 -N4` specifies four nodes from base node 0. Note that nodes that have been configured out are excluded. The `-B` option should be used to gain access to a specific file system or device that is not available on all nodes. If the `-B` option is used, the scheduler allocates the same number of CPUs on each node. Using this option causes

a request to block until the base node and any additional nodes required to run the program are free (unless the `-I` has been given).

The `-p` option specifies the partition from which CPUs can be allocated. CPUs cannot be allocated from the `root` partition.

The Partition Manager, `pmanager`, allocates processing resources to users as and when the resources are requested and become available. By default, a contiguous range of nodes is allocated to the request where possible. This enables programs to take advantage of the system's hardware broadcast facilities. The `-R` option can be used with `hwbcast` set to 1 to ensure that the range of nodes allocated is contiguous.

Before allocating resources, the Partition Manager checks the resource limits imposed on the current project. The project can be specified explicitly with the `-P` option. This overrides the value of the environment variable `RMS_PROJECT` or any default setting in the `users` table.

The *`script`* argument (with optional arguments) can be used in two different ways, as follows:

1. *`script`* is not specified, in which case an interactive command shell is spawned with the resources allocated to it. The user can confirm that resources have been allocated to an interactive shell by using the `rinfo` command (see Page A-22).

   The resources are reserved until the shell exits or until a time limit expires, whichever happens first. The system administrator can set a limit on the time for which CPUs may be allocated (see) or on the time for which allocated CPUs are idle (see `pmanager-idletimeout` attribute).

   Parallel programs, executed from this interactive shell, run on the CPUs allocated with `allocate`. They can run sequentially or concurrently; once CPUs have been allocated the user can choose how to use them.

2. *`script`* specifies a shell script, in which case the resources are allocated to the named subshell and freed when execution of the script completes.

ENVIRONMENT VARIABLES

The following environment variables may be used to identify resource requirements and modes of operation to `allocate`. They are used where no equivalent command line options are given.

`RMS_DEBUG` Specifies whether to execute in verbose mode and display diagnostic messages. The value is a mask taking values between `0` (no debug) and `2047`. Setting this variable will generate additional information that may be useful in diagnosing problems. If this environment

variable is not set, the `-v` option enables reporting of resource request debugging information.

`RMS_IMMEDIATE` Controls whether to exit (value `1`) rather than block (value `0`) if resources are not immediately available. The `-I` option overrides the value of this environment variable. By default, `allocate` blocks until resources become available. `root` resource requests are always met.

`RMS_MEMLIMIT` Specifies the maximum amount of memory required in megabytes. This must be less than or equal to the limit set by the system administrator.

`RMS_PARTITION` Specifies the name of a partition. The `-p` option overrides the value of this environment variable.

`RMS_PROJECT` Specifies the name of the project with which the request should be associated for accounting purposes. The `-P` option overrides the value of this environment variable.

`RMS_NRAILS` Specifies the number of rails (layers of switch network) required.

`RMS_TIMELIMIT` Specifies the execution time limit in seconds. The program will be signaled either after this time has elapsed or after any time limit imposed by the system has elapsed. The shorter of the two time limits is used.

`SHELL` Specifies the user shell to execute when no command script is given.

`allocate` passes all existing environment variables through to the shell that it executes. In addition, it sets the following environment variable:

`RMS_RESOURCEID` The identifier of the allocated resource.

`RMS_MACHINE` The RMS machine name, for example `atlas`.

EXAMPLES

To run a sequence of jobs on the same CPUs:

```
$ allocate -n 16 jobscript
```

where `jobscript` is a shell script such as the following:

```
#!/bin/sh
# simple job script
prun -n 16 program1
prun -n 16 program2
```

If the script was run directly then each resource request would block until resources became available and there would be no guarantee of both requests using the same CPUs. By running the script under `allocate`, there is only one resource request and both jobs are run on the same CPUs.

To run two programs on the same CPUs at the same time:

```
$ allocate -N 16 -C 2 << EOF
prun program1 &
prun program2 &
rinfo
wait
EOF
```

SEE ALSO

`prun`, `rinfo`

NAME

prun – Runs a parallel program

SYNOPSIS

```
prun [-IOstv] [-B base] [-c cpus] [-e mode] [-i mode] [-l tag]
     [-m block | cyclic] [-N nodes | all] [-n procs] [-o mode]
     [-P project] [-p partition] [-R request] program [args ...]
prun [-dtv] -p root | -r [-B base] [-H hostnamelist] [-l tag]
     [-N nodes] [-R request] program [args ...]
prun -f procfile [-EItv] [-e mode] [-i mode] [-l tag] [-o mode]
     [-P project] [-p partition]
prun [-h]
```

OPTIONS

`-B` *base*	Specifies the number (node id) of the base node (the first node to use). By default, the base node is unassigned, leaving the scheduler free to select nodes that are not in use.
`-c` *cpus*	Specifies the number of CPUs required per process (default 1).
`-d`	Distribute `rmsloader` to `/tmp` on each node prior to execution of the program. This option allows RMS to be used to install itself across a machine. This option can only used with the `-r` or `-p root` options and as such is restricted to admin users.
`-E`	*Explain* a procfile. This option can only be used with the `-f` option.
`-e` *mode*	Specifies how standard error output is redirected. Valid values for *mode* and their meanings are described below.
`-f` *procfile*	Generate a resource request from instructions in the *procfile*, allocate CPUs and start sufficient instances of `pfloader` to load the processes specified in the file. The format of a procfile is described below.
`-H` *hostlist*	Run processes on listed hosts. This option can only used with the `-r` or `-p root` options and as such is restricted to admin users.

`-h` Display the list of options.

`-I` Allocate CPUs immediately or fail. By default, `prun` blocks until resources become available.

`-i` *mode* Specifies how standard input is redirected. Valid values for *mode* and their meanings are described below.

`-l` *tag* Specify a tag to prefix each line of output.

`-m block|cyclic`
Specifies whether to use block (the default) or cyclic distribution of processes over nodes.

`-N` *nodes*`|all` Specifies the number of nodes required. To use all nodes in a partition, select the `all` argument as follows: `prun -N all`. If the number of nodes is not specified then the RMS scheduler will allocate one CPU per process.

`-n` *procs* Specifies the number of processes required. The `-n` and `-N` options can be combined to control how processes are distributed over nodes. If neither is specified, `prun` starts one process.

`-O` Allows resources to be over-committed. Set this flag to run more than one process per CPU.

`-o` *mode* Specifies how standard output is redirected. Valid values for *mode* and their meanings are described below.

`-P` *project* Specifies the name of the project with which the job should be associated for scheduling and accounting purposes.

`-p` *partition* Specifies the partition on which to run the program. By default, the partition specified in the `default-partition` attribute in the `attributes` table is used. The default is `parallel` .

`-R` *request* Requests a particular configuration of resources. The types of *request* currently supported are as follows:

`cluster=`*cluster* `|` `cluster-member=`*cluster-member*
This requests that all processes be run on the named *cluster* or on all nodes with the specified *cluster-member* ID.

`force=0|1` This is used only with the `-r` or `-p root` options. With a value of 1, this specifies that processes should be run on configured out nodes as well as configured in ones.

`hwbcast=0|1` With a value of 1, this specifies a contiguous range of nodes and constrains the scheduler to queue the request until a contiguous range of nodes becomes available.

`immediate=0|1` With a value of 1, this specifies that the request should fail if it cannot be met immediately (the same as the `-I` option).

`maxcpus=`*n* Specifies the minimum number of CPUs to be requested when `prun` is used with the `-O` option.

`mincpus=`*n* Specifies the minimum number of CPUs to be requested when `prun` is used with the `-O` option.

`pty=0|1` With a value of 1, this specifies that RMS should create a `pty` for process `0` of the parallel program. Input to `prun` is passed raw to this process. Output from this process is returned raw. All signals caught by `prun` are forwarded. For example to allocate a single CPU and run a remote shell:

```
$ prun -n1 -B4 -Rpty=1 csh
$ hostname
atlas4
```

`rails=`*n* In a multirail system, this specifies the number of rails required. For example:

```
$ prun -N2 -Rrails=1 tping -fdping 64k
  0:    65536 bytes    501.35 uSec   261.44 MB/s
$ prun -N2 -Rrails=2 tping -fdping 64k
  0:    65536 bytes    317.64 uSec   412.64 MB/s
```

`railmask=`*mask* In a multirail system, this specifies which rails are required using a bitmask to select them. Set bit `1«r` in the mask to select rail `r`. Valid values of the rail identifier `r` run from `0` to `R-1` where `R` is the number of rails in the system. For example to select rail one:

```
$ prun -N2 -Rrailmask=2 myprog
...
```

`samecpus=0|1` With a value of 1, this specifies the same set of CPUs on each node, for example, the first two CPUs, 0 and 1, on each node.

	Multiple requests can be entered as a comma-separated list, for example, `-R hwbcast=1,immediate=1`.
`-r`	Run processes using `rsh` (or `ssh` if the Used for administrative operations such as starting and stopping RMS. Always uses the `root` partition and runs one process per node. If the `-d` option is used with `-r` then `prun` with use `rcp` (or `scp`) to distribute copies of `rmsloader` prior to use. This allows `prun` to be used to install RMS.
`-s`	Print statistics as the job exits.
`-t`	Prefix output with the process number (a special case of the `-l` option).
`-T` *rid*	TotalView support, run one process per node on a named resource.
`-v`	Specifies verbose operation. Repeated use of the `-v` option increases the level of output: `-vv` shows each stage in running a program; `-vvv` enables logging of debugging output from the `rmsloader` processes on each node.

DESCRIPTION

The `prun` program executes multiple copies of the specified *program* on a partition. `prun` automatically requests resources for the program unless it is executed from a shell that already has resources allocated to it (see Page A-2).

The way in which processes are allocated to CPUs is controlled by the `-c`, `-n`, `-p`, `-B` and `-N` options or by the `-f` option. The `-n` option specifies the total number of processes to run. The `-c` option specifies the number of CPUs required per process, this defaults to 1. The `-N` option specifies how many nodes are to be used.

If the `-N` option is not used then the scheduler selects CPUs for the program from any of the available nodes. By default, RMS will allocate a contiguous range of nodes where possible. This enables programs to take advantage of the system's hardware broadcast facilities. To constrain RMS to make the range contiguous, use the `-R hwbcast=1` option. If the `-N` option is used, the scheduler allocates the specified number of nodes (in a contiguous range, if possible) and the same number of CPUs on each node.

The `-B` option specifies the base of a range of nodes. The `-N` option specifies its extent. So for example, `-B0 -N4` specifies four nodes from base node 0. Note that nodes that have been configured out are excluded. The `-B` option should be used to gain access to a specific file system or device that is not available on all nodes. If the `-B` option is used, the scheduler allocates the same number of CPUs on each node. Using this option causes

a request to block until the base node and any additional nodes required to run the program are free.

The `-I` option specifies that resource requests should fail if they cannot be met immediately. The default is to block until CPUs are available.

The `-m` option specifies how processes are to be distributed over nodes. The choice is between `block` (the default) and `cyclic`. If a program has `n` processes with identifiers `0,1,...n-1` distributed over `N` nodes then, in a block distribution, the first `n/N` processes are allocated to the first node and so on. If the distribution is cyclic, process 0 runs on the first node, process 1 on the second and so on until process `N-1` is placed on the last node, at which stage the distribution wraps around with process `N` running on the first node and so on. If `n` is not divisible by `N` then one extra process runs on each of the first `n%N` nodes.

The `-p` option specifies the partition to use. If no partition is specified then the default partition is used. The default partition is stored in the `default-partition` attribute in the `attributes` table. Note that use of the `root` partition (all nodes in the machine) is restricted to administrative users and is equivalent to using the `-r` option.

The `-r` or `-p root` options are used for running administrative operations on all the nodes in a machine. Processes will be run using `rsh`. By default, processes will be run on all nodes which are configured in. To run on configured out nodes as well (to start RMS on a node before configuring it in, for example) use the `-Rforce=1` option. Use the `-H` *`hostnamelist`* option or the `-B` and `-N` options to run on particular nodes. On large machines, `rsh` or `ssh` processes will be spawned so that there will be a maximum of 25 running concurrently.

The `-d` option is used only with the `-r` or the `-p root` options. If the `-d` option is used, a copy of `rmsloader` will be copied to `/tmp` on each node and executed in preference to the default. This allows `prun -r` to operate before RMS has been installed on a node. `rcp` is used to distribute the files.

The processes in a parallel program inherit their `ulimits` from `prun`. If the system administrator has set a memory limit then this may result in reduced values of the data size, stack size and virtual memory size limits.

Before allocating resources, `prun` checks the resource limits imposed on the current project. The project can be specified explicitly with the `-P` option. This overrides the value of the environment variable `RMS_PROJECT` or any default setting in the `users` table.

The `-f` option instructs `prun` to generate a resource request from the information in a *`procfile`*. This file specifies the processes required and their distribution. When resources have been allocated, `prun` starts the requisite number of instances of the `pfloader` application loader to load and run the processes.

Each line in the file may take one of the two forms:

number hostname[:path] executable [args]
number index[:path] executable [args]

The meaning of these lines is as follows:

number This indicates the number of processes required. It can be specified in two ways:

1. A triplet of numbers in the form: *lowest;highest;step* is used to specify explicitly the ranks required for processes. The first process is assigned the rank *lowest*, the next has the rank *lowest + step*, the next *lowest + (step*2)* and so on until the rank exceeds *highest*.
2. A single number that specifies how many processes are required with no indication of their ranks.

RMS assigns all of the ranks specified explicitly first. The remaining ranks are then allocated to processes specified in the second way. The assignments in each category are carried out in the order in which the lines appear in the file. `prun` reports an error if any of the ranks specified overlap or if a rank is missing.

hostname This specifies which node will host the processes.

index The node id of the host on which the processes will be executed.

path The working directory for the processes is set to *path*.

executable This specifies the executable to run.

arguments The *arguments* are passed to the processes.

The `-f` option can be combined with the options shown in the synopsis. It can also be used with the `-R` option and the following requests: `immediate` and `rails`. If other options are specified, `prun` silently ignores them. An additional `-E` option can be used in conjunction with the `-f` option. This instructs `prun` to explain how the processes will be distributed without actually launching any processes.

The `-T` option to `prun` is provided to support the TotalView bulk startup mechanism. When used with the id of a resource for which CPUs have been allocated `prun` will start one copy of the application process on each of the allocated nodes. Use of the `-T` option is not restricted to TotalView, you might use it, for example, to run the `ps` command on the same nodes as an application is using.

By default, when running a parallel program, prun forwards standard input to process 0. The -i option requests a different mode of operation. Valid values for *mode* and their meanings are as follows:

rank Forward standard input to the process *rank* where 0 ≤ *rank* < n and n is the number of processes in the program.

all Broadcast standard input to all of the processes.

none Do not forward standard input.

file prun opens the named *file* and associates it with the standard input stream so that each process reads standard input from the file. If the file does not exist, a read returns EOF.

file.% prun expands the % character to generate and open a separate file name for each process: process 0 reads standard input from *file*.0, process 1 reads standard input from *file*.1 and so on. If the file does not exist, a read returns EOF.

If the *mode* is *rank* or all, prun polls its standard input and forwards the data to the rmsloader of the application process (or processes if the *mode* is all). rmsloader writes the data to the standard input pipe for the process. This write may fail if the pipe is full (the application has not read the data). If this happens, rmsloader will periodically attempt to resend the data to the pipe. prun will not poll for further standard input until it has received an acknowledgement from the process (or all processes in the case of broadcast input) to say that this operation has completed.

The -o and -e options control the redirection and filtering of standard output and standard error respectively. Valid values for *mode* and their meanings for these options are as follows:

rank Redirect to prun standard output (or standard error) from the process identified by *rank* where 0 ≤ *rank* < n and n is the number of processes in the program.

all Redirect standard output (or standard error) from all processes to prun. This is the default.

none Do not redirect standard output (or standard error) from any process.

file prun opens the named *file* for output and associates it with the standard output (standard error) stream so that each process writes standard output (standard error) to the file.

file.% prun expands the % character to generate and open for output a separate file name for each process: process 0 writes standard output (standard error) to *file*.0, process 1 writes to *file*.1 and so on.

Standard output from a parallel program is line-buffered and redirected to prun when a newline character is received. Output that does not end in a newline is buffered by rmsloader.

Standard error is unbuffered and forwarded to prun as soon as it is received by rmsloader.

There is no global synchronization of output from a parallel program. If multiple processes output data, the order in which the data is output will not necessarily be the same each time the program is run. The *elanrte* functions described in the *Elan Programming Manual* provide for ordered output.

prun provides support for prefixing each line of output from a parallel program with a process dependent tag. The -t option generates a tag identifying the process by its rank. The -l option allows a more general character string prefix to be specified. The following special characters in the string are expanded:

%g	the global rank of the process
%G	the number of processes
%h	the rank of the node
%H	the number of nodes
%l	the rank of the process on this node
%L	the number of processes on the node
%m	text base of the hostname
%n	numeric part of the hostname
%p	the pid of the process
%t	the current time (hh:mm:ss)
%@	the hostname of the node (the same as "%m%-n")
%%	a single %

The -t option to prun is equivalent to a label of "%_g ". By default prun pads numeric fields with leading zeros to aid tools that parse this output. This behaviour can be

adjusted by adding a modifier after the '%' as follows: hyphen ('-') do not pad, underscore ('_') pad with spaces.

prun forwards certain signals to the processes in a parallel application (SIGUSR1, SIGUSR2, SIGQUIT and SIGWINCH by default). If the pty request flag is set all catchable signals are forwarded. Signals can also be sent using rcontrol or the RMS calls rms_killJob() and rms_killResource(). Note however that signals sent in this way will be delivered to all of the processes in the parallel program including rmsloader, which may cause rmsloader to be killed and the parallel program to be terminated. The following signals can be sent by rcontrol without (necessarily) terminating the parallel program: SIGTERM, SIGINT, SIGHUP, SIGUSR1, SIGUSR2, SIGXCPU, SIGWINCH and SIGQUIT.

prun exits when all of the processes in the parallel program have exited or when one process has been killed. If all processes exit cleanly then the exit status of prun is the global OR of their individual exit status values. If one of the processes is killed, prun will exit with a status value of 128 plus the signal number. prun can also exit with the following codes:

0 An application process exited with the code 127 (GLOBAL EXIT), which indicates success, causing all of the processes to exit.

123 An application process exited with the code 123 (GLOBAL ERROR) causing all of the processes to exit.

124 The job was killed when one or more of the nodes it was using crashed or was configured out.

125 One or more processes were still running when the exit timeout expired.

126 prun was run with the -I option and resources were not available.

127 prun was run with invalid arguments.

Note

Users of the Tru64 UNIX shell csh should be aware that its treatment of exit codes of 128 or greater is different from that of other shells. The shell variable $status is set to signal number minus 128. For example if a parallel program is killed with signal 11 the prun exit code will be -117 rather than 139.

If an application process started by prun is killed, RMS will run a post mortem core analysis script that generates a backtrace if it can find a core file for the process.

Core files are saved in the directory *local-corepath/resource-id*. The value of *local-corepath* is defined in the attributes table. The *resource-id* can be listed

by `rinfo` (see Page A-22). `prun` also sets the environment variable `RMS_RESOURCE_ID` to the value of the resource identifier.

The attribute `rms-keep-core` in the `attributes` table determines whether core files are saved. The environment variable `RMS_KEEP_CORE` can be set to override the value in the `attributes` table.

If the system has been set up to run `prologue` or `epilogue` scripts then they will run before or after your job respectively. If the `prologue` script fails, your program will not run and the following message will be displayed:

```
prun: Error: allocate failed: atlas0 /usr/local/rms/prologue status 1
```

The node will be configured out if this happens (failure to run the prologue script correctly indicates a serious problem with a node).

ENVIRONMENT VARIABLES

The following environment variables may be used to identify resource requirements and modes of operation to `prun`. These environment variables are used where no equivalent command line options are given:

`RMS_DEBUG`	Whether to execute in verbose mode and display diagnostic messages. Setting a value of `1` or more generates additional information that may be useful in diagnosing problems.
`RMS_EXITTIMEOUT`	Specifies the time allowed in seconds between the first process exit and the last. This option can be useful in parallel programs where one process can exit leaving the others blocked in interprocess communication. It should be used in conjunction with an exit barrier at the end of correct execution of the program. A permanent exit timeout can be set by setting the `exit-timeout` system attribute.
`RMS_IMMEDIATE`	Controls whether to exit rather than block if resources are not immediately available. The `-I` option overrides the value of this environment variable. By default, `prun` blocks until resources become available. Requests to use the `root` partition do not block, but they are restricted to system admin users.
`RMS_IOLABEL`	Specifies a process dependent prefix for each line of output.
`RMS_KEEP_CORE`	Controls whether core files are saved. This overrides the default behaviour set by the system administrator (in the `rms-keep-core` attribute in the `attributes` table.

RMS_MEMLIMIT	The maximum amount of memory required per process in megabytes. This must be less than or equal to any limit set by the system administrator.
RMS_NRAILS	Specifies the number of HP AlphaServer SC Interconnect rails required.
RMS_PARTITION	Specifies the name of a partition. The `-p` option overrides the value of this environment variable.
RMS_PROJECT	The name of the project with which the job should be associated for scheduling and accounting purposes. The `-P` option overrides the value of this environment variable.
RMS_RSH_LIMIT	Specifies the number of concurrent `rsh` processes started when using the `-r` or `-p root` options only. The `rsh` processes exit as soon as `rmsloader` is running, but having large numbers of them running at the same time may degrade system performance. The default value (25) allows `prun -r` to execute reasonably quickly on a large machine.
RMS_SSH	Specifies that the `-r` or `-p root` options will start processes with `ssh` instead of `rsh`.
RMS_STATINTERVAL	Specifies the rate at which the `statsig` signal is delivered to the application processes. Delivery of this signal will prompt `libelan` to dump performance statistics. Set the environment variable to the time in seconds between samples. The signal to be delivered is specified using the `libelan` environment variable `LIBELAN_STATSIG`.
RMS_STDINMODE	Specifies the *mode* for forwarding standard input to a parallel program. The `-i` option overrides the value of this environment variable. Values for *mode* are the same as those used with the `-i` option.
RMS_STDOUTMODE	Specifies the *mode* for redirecting standard output from a parallel program. The `-o` option overrides the value of this environment variable. Values for *mode* are the same as those used with the `-o` option.
RMS_STDERRMODE	Specifies the *mode* for redirecting standard error from a parallel program. The `-e` option overrides the value of this environment variable. Values for *mode* are the same as those used with the `-e` option.

RMS_TIMELIMIT	Specifies the CPU allocation time limit in seconds. The program will be signaled either after this time has elapsed or after any time limit imposed by the system has elapsed. The shorter of the two time limits is used. Note that this timelimit does not apply where prun is used on preallocated CPUs.

prun passes all existing environment variables through to the processes that it executes. In addition, it sets the following environment variables:

RMS_JOBID	The identifier for the job.
RMS_NNODES	The number of nodes used by the application.
RMS_NODEID	The logical identifier of the node within the set allocated to the application.
RMS_NPROCS	The total number of processes in the application.
RMS_PROCFILERANK	The procfile rank of the process in the application when the -f *procfile* option has been used.
RMS_RANK	The rank of the process in the application. The rank ranges from 0 to n-1, where n is the number of processes in the program.
RMS_RESOURCEID	The identifier of the allocated resource.

EXAMPLES

In the following example, prun is used to run a four-process program with no specification of where the processes should run.

```
$ prun -n 4 hostname
atlas0
atlas0
atlas0
atlas0
```

The machine atlas has four CPUs per node and so, by default, the scheduler allocates all four CPUs on one node to run the program. Add the -N option, as follows, to control how the processes are distributed over nodes.

```
$ prun -n 4 -N 2 hostname
atlas0
atlas0
atlas1
atlas1
```

```
$ prun -n 4 -N 4 hostname
atlas1
atlas3
atlas0
atlas2
```

The `-m` option controls how processes are distributed over nodes. It is used in the following example in conjunction with the `-t` option which tags each line of output with the identifier of the process that wrote it.

```
$ prun -t -n 4 -N 2 -m block hostname
0 atlas0
1 atlas0
2 atlas1
3 atlas1
$ prun -t -n 4 -N 2 -m cyclic hostname
0 atlas0
2 atlas0
1 atlas1
3 atlas1
```

The following examples illustrate the use of the `-l` option to prefix each line of output with a process dependent tag:

```
$  prun -N4 -l "%g: " hostname
1: atlas1
3: atlas3
0: atlas0
2: atlas2

$ prun -N4 -n16 -l "%g (%n.%l) " hostname | sort
00 (0.0) atlas0
01 (0.1) atlas0
02 (0.2) atlas0
03 (0.3) atlas0
04 (1.0) atlas1
05 (1.1) atlas1
06 (1.2) atlas1
07 (1.3) atlas1
08 (2.0) atlas2
09 (2.1) atlas2
10 (2.2) atlas2
11 (2.3) atlas2
12 (3.0) atlas3
13 (3.1) atlas3
14 (3.2) atlas3
15 (3.3) atlas3

$ prun -N4 -l "%t " hostname
11:50:42 atlas1
```

```
11:50:42 atlas0
11:50:42 atlas2
11:50:42 atlas3
```

The examples so far have used simple UNIX utilities to illustrate where processes are run. Parallel programs are run in just the same way. The following example measures DMA performance between a pair of processes on different nodes.

```
$ prun -N 2 dping 0 1k
  0:        0 bytes      1.88 uSec     0.00 MB/s
  0:        1 bytes      2.74 uSec     0.36 MB/s
  0:        2 bytes      2.64 uSec     0.76 MB/s
  0:        4 bytes      1.98 uSec     2.02 MB/s
  0:        8 bytes      1.98 uSec     4.04 MB/s
  0:       16 bytes      2.04 uSec     7.85 MB/s
  0:       32 bytes      2.09 uSec    15.31 MB/s
  0:       64 bytes      2.97 uSec    21.57 MB/s
  0:      128 bytes      3.08 uSec    41.55 MB/s
  0:      256 bytes      3.37 uSec    75.96 MB/s
  0:      512 bytes      4.32 uSec   118.59 MB/s
  0:     1024 bytes      5.89 uSec   173.96 MB/s
```

The `-s` option instructs `prun` to print a summary of the resources used by the job when it finishes.

```
$ prun -s -N 2 dping 0 32
  0:        0 bytes      1.87 uSec     0.00 MB/s
  0:        1 bytes      2.76 uSec     0.36 MB/s
  0:        2 bytes      2.66 uSec     0.75 MB/s
  0:        4 bytes      1.98 uSec     2.02 MB/s
  0:        8 bytes      1.97 uSec     4.05 MB/s
  0:       16 bytes      2.03 uSec     7.88 MB/s
  0:       32 bytes      2.07 uSec    15.43 MB/s
Elapsed time         0.97 secs    Allocated time       1.94 secs
User time            0.68 secs    System time          0.21 secs
Cpus used               2
```

Note that the allocated time (in CPU seconds) is twice the elapsed time (in seconds) because two CPUs were allocated.

The `-f` option is used in the following examples to show how a file can be used to specify different distributions of processes.

A file with the following lines specifies an uneven block distribution.

```
3 atlas0 a.out -x 1
4 atlas1 a.out -x 1
2 atlas2 a.out -x 1
3 atlas3 a.out -x 1
```

Three processes (with ranks 0-2) will run on `atlas0`, four processes (ranks 3-6) will run on `atlas1`, two processes (ranks 7-8) will run on `atlas2` and three processes (ranks 9-11) will run on `atlas3`. All of the processes will execute `a.out` with the arguments `-x 1`.

The following lines specify a master-slave configuration of processes.

```
1 atlas0 master
3 atlas0 slaves
```

Process 0 will be an instance of `master`; processes 1-3 will be instances of `slaves`.

A cyclic distribution is achieved by the following lines:

```
0;15;4 atlas5 a.out -y 0
1;15;4 atlas6 a.out -y 0
2;15;4 atlas7 a.out -y 0
3;15;4 atlas8 a.out -y 0
```

Processes 0, 4, 8 and 12 will be assigned to `atlas5`; processes 1, 5, 9 and 13 will be assigned to `atlas6`; processes 2, 6, 10 and 14 will be assigned to `atlas7` and processes 3, 7, 11 and 15 will be assigned to `atlas8`.

The final example sets some of the ranks explicitly:

```
0;0;1 atlas6 source
5;5;1 atlas7 sink
4     atlas8 worker
```

Process 0 will be an instance of `source`, process 5 will be an instance of `sink` and processes 1-4 will be instances of `worker`.

SEE ALSO

`allocate`, `rinfo`

NAME

rinfo – Displays resource usage and availability information for parallel jobs

SYNOPSIS

```
rinfo [-chlmNnpqr] [-E class | type] [-e all |
      class:name:type [-H hours]] [-j [choice="opt">jid]]
      [-L [partition] [statistic]] [-LM] [-o orderByList] [-s daemon
      [hostname | all]] [-t node | name] [-u [userList]]
```

OPTIONS

`-c` List the configuration names. With the `-l` option, shows detailed information about configurations and partitions.

`-E class | type` Show event classes or types.

`-e all | class:name:type [-H hours]`

Show all events or events matching *`class:name:type`*. Use `rinfo` with the `-E class` option or the `-E type` option to list possible event classes and types. If `*` is used for *`class`*, *`name`* or *`type`* then all events will be matched. If the `-H` *`hours`* option is used, only events in the preceding *`hours`* hours will be shown. If the `-l` option is used, a banner is printed.

`-h` Display the list of options.

`-l` Give more detailed information.

`-m` Show the machine name.

`-N` Show the status of the HP AlphaServer SC Interconnect network error service on each node.

`-n` Show the status of each node. This can be combined with `-l`.

`-p` Identify each active partition by name and indicate the number of CPUs in each partition.

`-q` Print information on the user's quotas and projects.

`-j [jid]` With no argument, list current jobs. With a job identifier argument, display full details about the specified job. When used with the `-l` option and a job identifier argument, display the full hostnames and process IDs for the job.

`-L [partition] [statistic]`

Print the hostname of a lightly loaded node in the machine or the specified *partition*. RMS provides a load-balancing service, accessible through `rmsexec`, that enables users to run their processes on lightly loaded nodes, where loading is evaluated according to a given *statistic* (see Page A-29) .

`-LM` Show license management information.

`-o orderByList` Display resource information selected by and in the order given by *orderByList*, which is a comma-separated list of field names from the resources table. If the `-j` option is used, job information is displayed and the list must contain fields from the jobs table. An error message is generated if the field names are invalid.

`-r` Show the allocated resources.

`-s daemon [hostname] | all`

Show the status of the *daemon*. Used with the argument `all`, `rinfo` shows the status of all daemons running on the `rmshost` node. For daemons that run on multiple nodes, such as `rmsd`, the optional *hostname* argument specifies the hostname of the node on which the daemon is running.

`-t node | name` Where *node* is the QsNet network identifier of a node, `rinfo` translates it into the hostname; where *name* is a hostname, `rinfo` translates it into the network identifier.

`-u userList` With no argument, only display information about the user's resources. With a list of user names, only display information about the listed users' resources.

DESCRIPTION

The `rinfo` program displays information about resource usage and availability. Its default output is in four parts that identify: the machine, the active configuration, resource requests and the current jobs. Note that the latter sections are only displayed if jobs are active.

```
$ rinfo
MACHINE        CONFIGURATION
atlas          day

PARTITION      CPUS    STATUS        TIME   TIMELIMIT  NODES
root              6                                   atlas[0-2]
parallel         2/4   running    01:02:29             atlas[0-1]

RESOURCE       CPUS    STATUS        TIME    USERNAME  NODES
parallel.996      2 allocated       00:05        user  atlas0

JOB            CPUS    STATUS        TIME    USERNAME  NODES
parallel.1115     2   running       00:04        user  atlas0
```

The machine section gives the name of the machine and the active configuration.

For each partition in the active configuration, `rinfo` shows the number of CPUs in use, the total number of CPUs, the partition status, the time since the partition was started, any CPU time limits imposed on jobs, and the node names. This information is extracted from the `partitions` table. The description of the `root` partition shows the resources of the whole machine.

The resource section identifies the resource allocated to the user, the number of CPUs that the resource includes, the user name, the node names and the status of the resource. The `time` field specifies how long the resource has been held in hours, minutes and seconds.

The job section gives the job identifier, the user name, the number of CPUs the job is using, on which nodes and the status of the job. The `time` field specifies how long the job has been running in hours, minutes and seconds.

The `-u` option restricts the output to that relating to a specific user. The `-r` and `-j` options restrict the information output to that relating to specified resources and jobs but generate additional detail including the process ID of the `prun` process that started the job, resource usage, the number of processes running and their distribution, process IDs and hostnames.

The fields displayed for jobs and resources and the ordering of these fields can be specified with the `-o` option.

EXAMPLES

When used with the `-cl` flags, `rinfo` prints information on the available configurations and shows current status.

```
$ rinfo -cl
CONFIGURATION   PARTITION   STATUS    ALLC  SUSP  QUED  BLKD  NODES
night           parallel    -                                atlas[0-15]
```

```
day             test        running      1     0     0     0  atlas[8-15]
day             parallel    running      0     0     0     0  atlas[0-7]
```

When used with the `-q` flag, `rinfo` prints information on the user's projects, CPU usage limits, memory limits and priorities.

```
$ rinfo -q
PARTITION       CLASS         NAME     CPUS   MEMLIMIT   PRIORITY
parallel      project      default      0/8        100          0
parallel      project      divisionA  16/64       none          1
```

In this example, the access controls allow any user to run jobs on up to 8 CPUs with a memory limit of 100MB. Jobs submitted for the `divisionA` project run at priority 1, have no memory limit and can use up to 64 CPUs. 16 of these 64 CPUs are in use.

When used with the `-s` option, `rinfo` prints information on the status of the RMS servers. Addition of the `-l` option generates banners in the output and a summary of the status of the per-node daemons.

```
$ rinfo -s all
tlogmgr              rmshost      running        239241
eventmgr             rmshost      running        239246
mmanager             rmshost      running        239260
swmgr                rmshost      running        239252
pmanager-parallel    rmshost      running        239175

$ rinfo -l -s rmsd
SERVER               HOSTNAME     STATUS         PID
rmsd                 atlas0       running        740600
rmsd                 atlas1       running        1054968
rmsd                 atlas2       running        1580438
rmsd                 atlas3       running        2143669
rmsd                 atlasms      running        239212

STATUS               NODES
running              atlas[0-3] atlasms
```

In the above example, the system is functioning correctly. In the following example, one of the nodes has crashed.

```
$ rinfo -l -s rmsd
SERVER               HOSTNAME     STATUS         PID
rmsd                 atlas0       running        740600
rmsd                 atlas1       running        1054968
rmsd                 atlas2       not responding
rmsd                 atlas3       running        2143669
rmsd                 atlasms      running        239212
```

The next example restricts the output to resources allocated to two users and requests more detailed output.

```
$ rinfo -rul dick harry
RESOURCE       CPUS  STATUS         TIME   USERNAME  NODES
parallel.41965 2     allocated  01:31:30   dick     atlas3
parallel.41964 2     allocated  01:32:08   harry    atlas2
parallel.41963 1     allocated  01:32:14   harry    atlas1
```

The following example uses the `-o` option to select which fields from the `resources` and `jobs` tables are displayed and their order.

```
$ rinfo -o username,name,status -u dick
USERNAME       NAME        STATUS
dick           41965       allocated

$ rinfo -j -o name,resource,status -u dick
NAME       RESOURCE        STATUS
40091      41965           running
```

The following examples show how `rinfo` can be used to list events. First, to show all events, listed in time order, with a banner:

```
$ rinfo -l -e all
Date     Time             Class       Name    Type Description
 ...
12/20/02 16:48:26          node      atlas0   status running
12/20/02 16:48:26          node      atlas2   status running
12/20/02 16:48:26          node      atlas3   status running
12/20/02 16:49:10     partition    parallel   status blocked
12/20/02 16:54:26          node      atlas1   status active
12/20/02 16:54:42          node      atlas1   status configured out
12/20/02 16:54:51     partition    parallel   status running
12/20/02 17:13:26     partition    parallel   status closing
12/20/02 17:13:26     partition    parallel   status down
12/20/02 17:15:25     partition    parallel   status starting
12/20/02 17:15:26     partition    parallel   status running
12/20/02 17:17:58 switch_module     QR1T00   control power/control problem
```

To restrict the output to node status events in the last hour use:

```
$ rinfo -e node:*:status -H 1
12/20/02 16:48:26          node      atlas0   status running
12/20/02 16:48:26          node      atlas2   status running
12/20/02 16:48:26          node      atlas3   status running
12/20/02 16:54:26          node      atlas1   status active
12/20/02 16:54:42          node      atlas1   status configured out
```

The following example uses the `-LM` option to show license management information.

```
$ rinfo -LM
RMS License Management Information
```

```
Feature        Tokens   Expires
RMS/partition  32       permanent

Feature        Tokens   In use since
Total          32
RMS/partition  16       Fri Dec 20 16:17:09 2002
```

The `-N` causes `rinfo` to poll the HP AlphaServer SC Interconnect network error service on each node and display a summary of the results. For example:

```
$ rinfo -N
rinfo: neterr service running on atlas[0-3]
```

When used with the `-jl` options `rinfo` will display information on the running jobs. The `-j` option with of one or more job ids will show more detailed information on specific jobs. If the `-l` option is used as well then `rinfo` will print the process ids on each node. For example:

```
$ rinfo -jl
JOB             CPUS    STATUS        TIME    USERNAME  NODES
parallel.31023     4   running       00:11      duncan  atlas[0-3]

$ rinfo -j 31023
JOB          STATUS       COMMAND                  HOSTNAMES
31023        unknown      sleep 1000               atlas[0-3]

RESOURCE     PARTITION    CPUS   PRI   USERNAME     PID          PROJECT
38558        parallel     4      50    duncan       864376       default

ETIME        ATIME        UTIME        STIME        PAGEFLTS     MEM
03:06        12:24        00:00        00:00        0            0

PROCESSES PER NODE
2 atlas[0-3]

$ rinfo -jl 31023
JOB          STATUS       COMMAND                  HOSTNAMES
31023        unknown      sleep 1000               atlas[0-3]

RESOURCE     PARTITION    CPUS   PRI   USERNAME     PID          PROJECT
38558        parallel     4      50    duncan       864376       default

ETIME        ATIME        UTIME        STIME        PAGEFLTS     MEM
03:36        14:24        00:00        00:00        0            0

PROCESSES
atlas0       865507   864461
atlas1      1429295  1428940
atlas2      1590783  1590735
atlas3      2550854  2550825
```

When used with the `-rl` options `rinfo` will display information on the resource requests. The `-r` option with of one or more resource ids will show more detailed information on specific requests. If the `-l` option is used as well then `rinfo` will print the process ids on each node. For example:

```
$ rinfo -rl
RESOURCE        CPUS    STATUS        TIME    USERNAME  NODES
parallel.38558     4 allocated       11:23      duncan  atlas[0-3]

$ rinfo -r 38558
RESOURCE     PARTITION    CPUS   PRI   USERNAME     PID          PROJECT
38558        parallel     4      50    duncan       864376       default

ETIME        ATIME        UTIME        STIME        PAGEFLTS     MEM
11:36        46:24        00:00        00:00        0            0

STATUS       HOSTNAMES
allocated    atlas[0-3]

JOBS
31023

PROCESSES PER NODE
2 atlas[0-3]

$ rinfo -rl 38558
RESOURCE     PARTITION    CPUS   PRI   USERNAME     PID          PROJECT
38558        parallel     4      50    duncan       864376       default

ETIME        ATIME        UTIME        STIME        PAGEFLTS     MEM
11:36        46:24        00:00        00:00        0            0

STATUS       HOSTNAMES
allocated    atlas[0-3]

JOBS
31023

HOSTNAME     JOB       PROCESSES
atlas0       31023       865507   864461
atlas1       31023      1429295  1428940
atlas2       31023      1590783  1590735
atlas3       31023      2550854  2550825
```

SEE ALSO

`allocate`, `prun`

NAME

rmsexec – Runs a sequential program on a lightly loaded node

SYNOPSIS

rmsexec [-hv] [-p *partition*] [-s *stat*] [*hostname*] *program* [*args ...*]

OPTIONS

`-h`	Display the list of options.
`-v`	Specifies verbose operation.
`-p` *partition*	Specifies the target partition. The request will fail if load-balancing is not enabled on the partition .
`-s` *stat*	Specifies the statistic on which to base the load-balancing calculation (see below).

DESCRIPTION

The `rmsexec` program provides a mechanism for running sequential programs on lightly loaded nodes – nodes, for example, with free memory or low CPU usage. It locates a suitable node and then runs the *program* on it.

The user can select a node from a specific partition (of type `login` or `general`) with the `-p` option. Without the `-p` option, `rmsexec` uses the default load-balancing partition (specified with the `lbal-partition` attribute in the `attributes` table). In addition, the *hostname* of the node can be specified explicitly. The request will fail if this node is not available to the user. System administrators may select any node.

The `-s` option can be used to specify a statistic on which to base the loading calculation. Available statistics are:

`usercpu`	Percentage of CPU time spent in the user state.
`syscpu`	Percentage of CPU time spent in the system state - a measure of the I/O load on a node.
`idlecpu`	Percentage of CPU time spent in the idle state.

`freemem`	Free memory in megabytes.
`users`	Lowest number of users.

By default, `usercpu` is used as the statistic. Statistics can be used on their own, in which case a node is chosen that is lightly loaded according to this statistic, or you can specify a threshold using *`statistic < value`* or *`statistic > value`*

EXAMPLES

Some examples follow:

```
$ rmsexec -s usercpu myprog
$ rmsexec -s "usercpu < 50" myprog
$ rmsexec -s "freemem > 256" myprog
```

SEE ALSO

`prun`, `rinfo`

NAME

rmsquery – Submits SQL queries to the RMS database

SYNOPSIS

rmsquery [-ehsuv] [-d *database*] [-m *machine*] [*SQLquery*]

OPTIONS

-d *database*	Select database by name.
-e	Exit quickly with an error if the RMS database server is not running.
-h	Display the list of options.
-m *machine*	Select database by machine name.
-s	Shows supported SQL syntax.
-u	Print dates as seconds since January 1st 1970. The default is to print dates as a string created with `localtime(3)`.
-v	Verbosely prints field names above each column of output.

DESCRIPTION

`rmsquery` is used to submit SQL queries to the RMS database. Users are restricted to using the `select` statement to extract information from the database. System administrators may also submit SQL statements that update the database: `create`, `delete`, `drop`, `insert` and `update`. Note that queries modifying the database are logged. When used without arguments, `rmsquery` operates interactively and a sequence of commands can be issued.

The source for `rmsquery` is available from the software download pages of the Quadrics website http://www.quadrics.com. When used interactively, this version of `rmsquery` supports GNU readline and history mechanisms. Type `history` to see recent commands, use `Ctrl/p` and `Ctrl/n` to step back and forward through them. The `tables` command lists the tables in the selected database. The command `fields` followed by the name of a table lists the fields in a table. The command `verbose` toggles printing of field names. To quit interactive mode, type `Ctrl/d` or `exit` or `quit`.

When used in interactive mode the following commands can be submitted in the place of a query:

`exit (or ^D)`	Exits and saves history file.
`fields` *`table`*	Prints the names of the fields in a table.
`help`	Prints the list of available commands.
`history`	Prints query history.
`syntax`	Prints a brief description of supported SQL syntax (the equivalent of the `-s` option).
`tables`	Prints the names of all of the tables in the database.
`verbose`	Prints field names above the columns (the equivalent of the `-v` option).

The sources for `rmsquery` are distributed under the terms of the GNU General Public License. See `http://www.gnu.org` for details and more information on GNU readline and history. The sources can be downloaded from the Quadrics website. Details of the SQL language can be found on the Quadrics support page `http://www.quadrics.com`.

EXAMPLES

An example follows of a `select` statement that results in a list of the names of all of the nodes in the machine. Note that the *`query`* must be quoted. This is because `rmsquery` expects a single argument.

```
$ rmsquery "select name from nodes"
atlas0
atlas1
atlas2
atlas3
```

In the following example, `rmsquery` is used to print information on all jobs run by a user:

```
$ rmsquery "select name,status,hostnames,cpus,startTime,endTime from \
                    resources where username='user'"
7     finished  atlas[0-3]   4      12/21/99 11:16:44  12/21/99 11:16:46
8     finished  atlas0       2      12/21/99 11:54:23  12/21/99 11:54:29
9     finished  atlas[0-3]   4      12/21/99 11:54:35  12/21/99 11:54:39
```

The `-v` option prints field names. In the following example, `rmsquery` is used to print resource usage statistics:

```
$ rmsquery -v "select * from acctstats"
name  uid   project  started            etime   atime   utime   stime  ...
-------------------------------------------------------------------------
7     1507  1        12/21/99 11:16:44    2.00    8.00    0.10    0.22  ...
8     1507  1        12/21/99 11:54:23    6.65   13.30   10.62    0.10  ...
9     1507  1        12/21/99 11:54:35    4.27   16.63   12.28    0.44  ...
```

When used without arguments, `rmsquery` operates interactively and a sequence of commands can be issued.

```
$ rmsquery -v
sql> select name, status from partitions
name       status
------------------
login      running
parallel   running
sql>
```

B

Submitting LSF Jobs

B.1 Overview of LSF/RMS Interaction

When LSF is used to manage jobs, the LSF system is responsible for deciding which hosts will run the job. LSF does not start the job itself; instead, it uses RMS services to execute the job. The process is as follows:

- The user runs the `bsub` command to submit the job to an LSF queue, using the `-n` flag to specify how many CPUs should be allocated to the job.
- The job stays in the queue until the required number of CPUs are available and the resource requirements are met. Resource requirements can be defined either at the job submission stage or at the queue definition level. For some HP AlphaServer SC Version 2.6 LSF topology scheduling policies, the free CPUs must be in a block of contiguous nodes – this acts as an additional constraint.
- When LSF and RMS determine that suitable CPUs are available and that the resource requirements have been met, LSF sends an allocate request to RMS. This is equivalent to the `allocate(1)` command. At this stage, `rinfo(1)` would show that a resource has been allocated.
- LSF then runs one copy of the user's job within this allocation. This is equivalent to `prun -n1` executed within an `allocate(1)` shell. At this stage, `rinfo(1)` would show one job belonging to the user.
- Frequently, the user's job is a `prun` command or a script containing one or more `prun` commands. These `prun` commands start processes within the previously allocated

resource. Each of these prun commands is shown by rinfo(1). Therefore, it is normal to see two RMS jobs associated with each LSF job – the first is the prun -n1 job, the second is associated with the user's prun commands.

B.2 Setting Up Your Environment

To use LSF, you must set up various path and environment variables. The /usr/share/lsf/conf directory contains two files, cshrc.lsf and profile.lsf. You can source one of these files to perform this setup – the choice of file depends on the shell.

Instead of sourcing these files each time you log in, you can incorporate them into your .cshrc or .profile files as follows:

- If using C shell (csh or tcsh), add the following lines to the ~/.cshrc file:

```
if ( -f /usr/share/lsf/conf/cshrc.lsf ) then
  source /usr/share/lsf/conf/cshrc.lsf
endif
```

- If not using C shell, add the following lines to the $HOME/.profile file:

```
if [ -f /usr/share/lsf/conf/profile.lsf ] ; then
  . /usr/share/lsf/conf/profile.lsf
fi
```

You must place this if statement after the statements that set the default PATH and MANPATH environment variables in the .cshrc or .profile file.

B.3 The bsub command

To submit an LSF job, use the bsub command. If you do not specify a queue, LSF jobs are submitted to the default queue, as shown in the following example:

```
$ bsub -n num_cpus my_job
Job <1234> is submitted to default queue <normal>
```

where *num_cpus* is the number of CPUs to be allocated, my_job is the job being submitted, 1234 is the job ID assigned by LSF (this is not the same as the RMS job number or the RMS resource number) and normal is the name of the default batch queue on the system. If RMS topology options are set on the default queue they are used; otherwise the RMS_SNODE allocation type is used by default if the job is run on an RMS host.

The normal queue is the default queue, unless the LSF administrator has changed the default.

Jobs remain pending in the queue until all conditions for their execution are met. Each queue has execution conditions that apply to all jobs in the queue, and additional conditions can be specified when jobs are submitted.

my_job can be a compiled executable, a prun command, or a shell script. Shell-specific command syntax is not supported when submitting jobs to run under RMS. For example, bsub "cmd1 | cmd2" is not valid because of the pipe (|).

The following examples show various ways to use the bsub command. The first example shows how to use the bsub command to run a sequential job; the other examples show how the bsub and prun commands interact so that you can run either a parallel job or a combination of parallel and sequential jobs:

- To allocate one CPU and submit a sequential job a.out, run either of the following (equivalent) commands:

```
$ bsub -n 1 a.out
$ bsub a.out
```

- To allocate eight CPUs and submit an eight-CPU parallel job a.out, run either of the following (equivalent) commands:

```
$ bsub -n 8 prun -n 8 a.out
$ bsub -n 8 prun a.out
```

- To allocate eight CPUs and submit a six-CPU parallel job a.out, run the following command:

```
$ bsub -n 8 prun -n 6 a.out
```

- To allocate eight CPUs and start a Bourne-shell script that runs two parallel jobs, a1.out and a2.out, one after the other, write a script (script.sh for example) such as the following:

```
#!/bin/sh
prun -n 4 a1.out
prun -n 8 a2.out
```

  then submit the job as follows:

```
$ bsub -n 8 ./script.sh
```

When the prun command is used with the bsub command, the following prun options cannot be used: -B, -c, -E, -f, -H, -I, -P, -p, -R, and -T. The examples shown in this section use only the basic options for running a job. If you need more control over how your job is scheduled or which nodes are allocated to a job, you can use other bsub options such as the -R and -m options. You can also use the -ext option to define topology scheduling options that specify the numbers of nodes or CPUs to be used, a

railmask, and other parameters that influence how your job is scheduled. For more information on the `-ext` option, see Section B.5.

For more information, including a description of the other `bsub` options, see the *HP AlphaServer SC Platform LSF® Reference* or *Running Jobs with HP AlphaServer SC Platform LSF®*. For more information about job operations, see *Administering HP AlphaServer SC Platform LSF®*.

B.4 Running Pre-Execution Programs

Although LSF daemons only run on the management node of a virtual host, batch jobs can run on any nodes within a virtual host that satisfy the scheduling and topological requirements. By default, pre-execution commands can only run on the node where the `sbatchd` daemon runs.

To enable pre-execution on other nodes in an RMS allocation, the prun job you run in your pre-execution program picks up the following environment variables set by LSF:

- `RMS_RESOURCEID=`*`partition.resource_id`*
- `RMS_MACHINE`

These variables give LSF the information it needs to run the job on the nodes required by the pre-execution program.

Note

Post-execution commands will only run on the daemon node, and will only run after the RMS allocation has been destroyed.

You can control where to run a pre-execution program so that it runs on one of the following:

- The daemon node of a virtual host
- The first execution node in an allocation
- All nodes in an allocation, one instance per node

To run a pre-execution program on the daemon node of a virtual host, run the pre-execution program normally. The pre-execution program does not need to make use of the `RMS_MACHINE` and `RMS_RESOURCEID` environment variables.

To run a pre-execution program on the first execution node in an allocation, use the `prun` command as follows:

```
$ prun -n 1 my_pre_exec
```

where *my_pre_exec* is the name of the pre-execution program.

To run a pre-execution program on all nodes in an allocation, one instance per node, use the `prun` command with the `-m cyclic` option to start the pre-execution program on all processors of each node. The pre-execution program that `prun` invokes must compare the `RMS_RANK` and `RMS_NODEID` prun environment variables so that when `RMS_RANK=RMS_NODEID`, the pre-execution process on the first processor of each node should keep running, while allowing the remaining pre-execution processes on a node to exit normally.

See the the `prun` manual page for more information about the `-m` and `-n` options.

B.5 LSF Topology Scheduler

The topology scheduler for RMS jobs determines which hosts will execute the RMS jobs.

The system administrator specifies a default queue-level topology scheduler. If the default scheduler is mandatory, you cannot overwrite the default specification when submitting jobs. If the default scheduler is non-mandatory, you can overwrite the default specification with job-level options, by using the `-ext` option of the `bsub` command.

The syntax of the `-ext` option is as follows:

```
-ext "RMS[[allocation_type][;topology][;flags]]"
```

You can use either `-ext` or `-extsched`. For example, the following commands are both valid:

```
-extsched "RMS[RMS_SNODE]"
-ext "RMS[RMS_SNODE]"
```

Note

The `-extsched` syntax in the form:

```
-extsched extsched_options
```

which was used in earlier versions of LSF is also supported in Version 6.0. However, the new syntax must be used for new features in LSF Version 6.0, as follows:

```
-ext "RMS[[allocation_type][;topology][;flags]]"
```

or

```
-extsched "RMS[[allocation_type][;topology][;flags]]"
```

B.5.1 Allocation Type

allocation_type specifies the type of node allocation, and is one of the following values:

- RMS_SNODE

  RMS_SNODE specifies *sorted node allocation*. Nodes do not need to be contiguous: gaps are allowed between the leftmost and rightmost nodes of the allocation map.

  LSF sorts nodes according to RMS topology (numbering of nodes and domains), which takes precedence over LSF sorting order.

  The allocation is more compact than in RMS_SLOAD; allocation starts from the leftmost node allowed by the LSF host list, and continues rightward until the allocation specification is satisfied.

  RMS_SNODE is the default allocation type.

- RMS_SLOAD

  RMS_SLOAD specifies *sorted load allocation*. Nodes do not need to be contiguous: gaps are allowed between the leftmost and rightmost nodes of the allocation map.

  LSF sorts nodes based on host preference and load information, which takes precedence over RMS topology (numbering of nodes and domains).

  The allocation starts from the first host specified in the list of LSF hosts, and continues until the allocation specification is satisfied.

  Use RMS_SLOAD on smaller clusters, where the job placement decision should be influenced by host load, or where you want to keep a specific host preference.

  Using an exclamation point (`!`) in host selection to indicate mandatory first execution host (for example, `-m "hostA! hostB"`) is supported for `RMS_SNODE` and `RMS_MCONT` allocation types. If you specify `RMS_SLOAD` with mandatory first execution host, LSF changes the allocation type to `RMS_SNODE`.

- RMS_MCONT

  RMS_MCONT specifies *mandatory contiguous node allocation*. The allocation must be contiguous: between the leftmost and rightmost nodes of the allocation map, each node must either have at least one CPU that belongs to this allocation, or this node must be configured out completely.

  The sorting order for RMS_MCONT is RMS topological order; LSF preferences are not taken into account.

  The allocation is more compact than in RMS_SLOAD, but requires contiguous nodes. Allocation starts from the leftmost node that allows contiguous allocation. Nodes that are out of service are not considered as gaps.

If you specify a CPU range (`bsub -n` *`min_processors,max_processors`*), LSF builds a CPU allocation on a first-fit basis. For example, if you specify the following:

```
bsub -n 1,20
```

a partition that has an area at the beginning of the partition with 10 contiguous CPUs, and another area at the end of the partition with 18 contiguous CPUs, has two areas that can fit the job. LSF places the job in the first area that satisfies the minimum specification; in this case, LSF places the job at the beginning of the partition.

B.5.2 Topology

`topology` specifies the topology of the allocation, and can have the following values:

- `nodes=`*`nodes`* `| ptile=`*`cpus_per_node`*

  `nodes` specifies the number of nodes that the allocation requires, or the number of CPUs per node.

  The `ptile` topology option is different from the LSF `ptile` keyword used in the `span` section of the resource requirement string (`bsub -R "span[ptile=n]"`). If the `ptile` topology option is specified in `-ext`, the value of `bsub -n` must be an exact multiple of the `ptile` value.

  The following example is valid, because the value of `-n` (12) is exactly divisible by the value of `ptile` (4).

  ```
  $ bsub -n 12 -ext "RMS[ptile=4] prun mt_app"
  ```

  To force a multi-threaded application to run within a node, use `-ext "RMS[nodes=1]"`. For example:

  ```
  $ bsub -n 3 -ext "RMS[RMS_SLOAD;nodes=1]" prun mt_app
  ```

  To enforce the condition that a job only takes nodes where there is no other job running, use `ptile=`*`cpus_per_node`*, as in the following example (run on an HP AlphaServer ES40 node):

  ```
  $ bsub -n 20 -ext "RMS[RMS_SLOAD;ptile=4]" prun parallel_app
  ```

  Note

  If you specify a CPU range (`bsub` *`min_processors,max_processors`*) with the `RMS[nodes]` option, only the *`min_processors`* argument is considered; the *`max_processors`* argument is ignored.

- `base=`*`base_node_name`*

  If `base` is specified with the RMS_SNODE or RMS_MCONT allocation, the starting node for the allocation is the `base` node name, instead of the leftmost node allowed by the LSF host list.

  If `base` is specified with the RMS_SLOAD allocation, RMS_SNODE allocation is used.

B.5.3 Flags

`flags` specifies other RMS allocation options. The only supported flags are `rails=`*`number`* or `railmask=`*`bitmask`*. LSF ignores any other RMS allocation options that you specify with the `-ext` option.

In HP AlphaServer SC systems, rails are the layers of an HP AlphaServer SC Interconnect. In multirail systems, you can use the `rails` and `railmask` options as follows:

- `rails` – specifies how many rails to use
- `railmask` – specifies a bitmask to control which rail is used. For example, a value of 1 indicates that rail 0 should be used and a value of 2 specifies that rail 1 should be used.

Specify either the `rails` option or the `railmask` option, but not both.

The following example shows how to submit a job that uses two rails:

```
$ bsub -n 4 -q rms -ext "RMS[RMS_MCONT;rails=2]"
```

When you submit a job, LSF checks the validity of the rail options against the value of the LSB_RMS_MAXNUMRAILS parameter (if set) in the `lsf.conf` file. If any rail-option values are not valid at resource allocation time, the allocation will fail and the job will pend forever.

For more information about the `rails` and `railmask` options, see the *HP AlphaServer SC RMS Reference Manual*. For more information about rails in HP AlphaServer SC systems, see the *HP AlphaServer SC System Administration Guide*.

B.6 RMS Allocation Options Supported by LSF

Table B.1 lists the RMS allocation options that are supported by LSF – other `prun` options are not supported with LSF.

Table B.1: RMS Allocation Options (prun/allocate) Supported by LSF

RMS Option	Description	LSF Equivalent
`-B`	Base node index	`-extsched "base=`*`base_node_name`*`"`
`-C`	Number of CPUs per Node (used with allocate)	`-ext "RMS[ptile=`*`cpus_per_node`*`]"`
`-e`	Error redirection	`bsub -e`
`-I`	Allocate immediately or fail	LSF overrides; uses immediately to allocate
`-i`	Input redirection	`bsub -i`
`-m`	Block or cyclic job distribution	Must be passed directly (not *via* LSF) by the user on the `prun` command line
`-N`	Number of nodes to use	`-extsched "nodes=`*`nodes`*`"`
`-n`	Number of CPUs to use	`-n`
`-o`	Output redirection	`bsub -o`
`-R`	Immediate, same as -I	LSF overrides
	`rails`	Passed with `-extsched`
	`railmask`	Passed with `-extsched`

B.7 How LSF Calculates Load Indices

All load indices on RMS hosts (that is, hosts that have the RMS resource defined) are queried from the SC database. All indices for non-RMS hosts (such as the management server) are obtained *via* Tru64 UNIX system calls.

Table B.2 lists the load indices that are reported for RMS hosts – note that these indices are reported only for configured-in nodes.

Table B.2: LSF Load Indices

Load Index	Description
`r15s`	Sum of exponentially averaged values of `node_stats.rql5` over 15 seconds
`r1m`	Sum of exponentially averaged values of `node_stats.rql5` over 1 minute
`r5m`	Sum of exponentially averaged values of `node_stats.rql5` over 5 minutes

(continued on next page)

Table B.2: LSF Load Indices (cont.)

Load Index	Description
ut	Sum of (100 - node_stats.idlecpu)/100
pg	Sum of node_stats.pages
io	Not available, reported as -
ls	Sum of node_stats.users
it	Not collected, reported as -
tmp	Minimum of all node_stats.freetmp values
swp	Minimum of all node_stats.freeswap values
mem	Minimum of all node_stats.freemem values
maxtmp	Minimum of all nodes.maxtmp values
maxswap	Minimum of all nodes.maxswap values
maxmem	Minimum of all nodes.maxmem values

For more information about LSF load indices, see *Administering HP AlphaServer SC Platform LSF®*.

B.8 Monitoring and Controlling LSF Jobs

Once you have submitted LSF jobs using the bsub command, you can monitor or modify the jobs by using the following LSF commands:

bjobs	Displays information about LSF jobs
bhist	Displays historical information about jobs
bstop	Suspends unfinished jobs
bresume	Resumes one or more suspended jobs
bkill	Sends signals to kill, suspend, or resume unfinished jobs
bhosts	Displays information about hosts, for example, host name, status, job slot limits, job state statistics.
bacct	Displays accounting statistics about finished jobs

Note

When a partition is `blocked` or `down`, the status of the `prun` jobs becomes `unknown`, and the `bjobs` command shows the jobs as still running.

For more information about these and other LSF commands, see *HP AlphaServer SC Platform LSF® Reference* or *Running Jobs with HP AlphaServer SC Platform LSF®*.

B.9 Viewing Nodes Allocated To Your Job

B.9.1 Running Jobs (`bjobs -l`)

Use the `bjobs -l` command to view RMS allocation information for a running job. For example, the following job allocates nodes on hosts `atlasD0` and `atlasD1`, with resource ID `3759` in the partition named `parallel`:

```
$ bjobs -l 723
Job <723>, User <user1>, Project <default>, Status <DONE>, Queue <rms>,
                     Extsched <RMS[nodes=3;base=atlas1]>, Command <hostname>
Wed Aug 6 17:09:44: Submitted from host <atlasD0>, CWD <$HOME>, 7 Processors R
                    equested;
 STACKLIMIT
   5256 K
Wed Aug 6 17:09:58: Started on 7 Hosts/Processors <3*atlasD0> <4*atlasD1>,
                    Execution Home </home/user1>, Execution CWD </home/user1>;
Wed Aug 6 17:09:58: rms_rid=parallel.3759;rms_alloc=3*atlas1 2*atlas[2-3];
Wed Aug 6 17:10:01: Done successfully. The CPU time used is 0.1 seconds.
 SCHEDULING PARAMETERS:
           r15s   r1m   r15m   ut   pg   io   ls   it   tmp   swp   mem
 loadSched   -     -      -     -    -    -    -    -    -     -     -
 loadStop    -     -      -     -    -    -    -    -    -     -     -

  EXTERNAL MESSAGES:
 MSG_ID FROM POST_TIME MESSAGE
ATTACHMENT
 0          -        -               -           -
 1       user1 Aug 6 17:09 RMS[nodes=3;base=atlas1] N
```

B.9.2 Finished Jobs (`bhist -l`)

Use the `bhist -l` command to view RMS allocation information for finished jobs. For example:

```
$ bhist -l 723
Job <723>, User <user1>, Project <default>, Extsched <RMS[nodes=3;base=atlas1
                    ]>, Command <hostname>
Wed Aug 6 17:09:44: Submitted from host <atlasD0>, to Queue <rms>, CWD <$HOME>,
```

```
                    7 Processors Requested;
Wed Aug 6 17:09:58: Dispatched to 7 Hosts/Processors <3*atlasD0> <4*atlasD1>;
Wed Aug 6 17:09:58: rms_rid=parallel.3759;rms_alloc=3*atlas1 2*atlas[2-3];
Wed Aug 6 17:09:58: Starting (Pid 971318);
Wed Aug 6 17:10:00: Running with execution home </home/user1>, Execution CWD
                    </home/user1>, Execution Pid <971318>;
Wed Aug 6 17:10:01: Done successfully. The CPU time used is 0.1 seconds;
Wed Aug 6 17:10:01: Post job process done successfully;

Summary of time in seconds spent in various states by Wed Aug 6 17:10:01
   PEND    PSUSP    RUN      USUSP    SSUSP    UNKWN   TOTAL
   14      0        3        0        0        0       17
```

B.9.3 Job Accounting Information (`bacct -l`)

Use the `bacct -l` command to view RMS allocation information logged to `lsb.acct`. For example:

```
$ bacct -l 3088
Accounting information about jobs that are:
- submitted by all users.
- accounted on all projects.
- completed normally or exited
- executed on all hosts.
- submitted to all queues.
------------------------------------------------------------------------------
Job <3088>, User <user1>, Project <default>, Status <DONE>, Queue <rms>,
Command
                     <prun hostname>
Wed Aug 6 17:09:44: Submitted from host <atlasD0>, CWD <$HOME>;
Wed Aug 6 17:09:58: Dispatched to 7 Hosts/Processors <3*atlasD0> <4*atlasD1>;
Wed Aug 6 17:09:58: rms_rid=parallel.3759;rms_alloc=3*atlas1 2*atlas[2-3];
Wed Aug 6 17:10:01: Completed <done>.
Accounting information about this job:
     CPU_T     WAIT     TURNAROUND    STATUS    HOG_FACTOR    MEM    SWAP
      0.43        8             43      exit        0.0100  1024K      0K
------------------------------------------------------------------------------
SUMMARY: ( time unit: second )
 Total number of done jobs:      0        Total number of exited jobs:    1
 Total CPU time consumed:      0.4        Average CPU time consumed:    0.4
 Maximum CPU time of a job:    0.4        Minimum CPU time of a job:    0.4
 Total wait time in queues:    8.0
 Average wait time in queue:   8.0
 Maximum wait time in queue:   8.0        Minimum wait time in queue:   8.0
 Average turnaround time:       43 (seconds/job)
 Maximum turnaround time:       43        Minimum turnaround time:       43
 Average hog factor of a job: 0.01 ( cpu time / turnaround time )
 Maximum hog factor of a job: 0.01        Minimum hog factor of a job: 0.01
```

B.10 LSF Job Termination Reason

When a job finishes, LSF reports the last job termination action it took against the job and logs it to the `lsb.acct` file. Use the `bacct -l` command to view the job exit information logged to the `lsb.acct` file, as follows:

```
$ bacct -l

% bacct -l 3088
Accounting information about jobs that are:
- submitted by all users.
- accounted on all projects.
- completed normally or exited
- executed on all hosts.
- submitted to all queues.
------------------------------------------------------------------------------
Job <3088>, User <user1>, Project <default>, Status <EXIT>, Queue <rms>,
Command
                    <prun hostname>
Wed Aug 6 17:09:44: Submitted from host <atlasD0>, CWD <$HOME>;
Wed Aug 6 17:09:58: Dispatched to 7 Hosts/Processors <3*atlasD0> <4*atlasD1>;
Wed Aug 6 17:09:58: rms_rid=parallel.3759;rms_alloc=3*atlas1 2*atlas[2-3];
Wed Aug 6 17:09:58: Completed ,exit>; TERM_RUNLIMIT: job was killed after LSF
run limit expired
Accounting information about this job:
CPU_T     WAIT     TURNAROUND   STATUS     HOG_FACTOR    MEM    SWAP
0.00      0        3224         exit       0.0000        0K     0K

SUMMARY: ( time unit: second )
Total number of done jobs:        0      Total number of exited jobs:    1
Total CPU time consumed:        0.0      Average CPU time consumed:    0.0
Maximum CPU time of a job:      0.0      Minimum CPU time of a job:    0.0
Total wait time in queues:      0.0
Average wait time in queue:     0.0
Maximum wait time in queue:     0.0      Minimum wait time in queue:   0.0
Average turnaround time:         43 (seconds/job)
Maximum turnaround time:       3224      Minimum turnaround time:     3224
Average hog factor of a job:   0.00 ( cpu time / turnaround time )
Maximum hog factor of a job:   0.00      Minimum hog factor of a job: 0.00
```

For more information on LSF job termination reasons, see the *HP AlphaServer SC System Administration Guide*.

C

MPI Library Routines

C.1 Overview

The MPI library is based on MPICH release 1.2.4 with the addition of MPI-2 one-sided support, optimized collectives and MPI-IO.

C.2 MPI-2 One-Sided Support

The following MPI-2 window creation functions are supported:

```
MPI_Window_create()     MPI_Alloc_mem()
MPI_Free_mem()
```

As the Elan can both read and write virtual memory, the functions `MPI_Alloc_mem()` and `MPI_Free_mem()` simply call `malloc()`.

The following window attribute functions are supported:

```
MPI_Win_get_attr()      MPI_WIN_BASE()
MPI_WIN_SIZE()          MPI_WIN_DISP_UNIT()
```

The following MPI-2 communications calls are supported:

```
MPI_Put()               MPI_Get()
MPI_Accumulate()
```

The following MPI-2 synchronization functions are supported:

```
MPI_Win_start()         MPI_Win_complete()
MPI_Win_post()          MPI_Win_wait()
MPI_Win_lock()          MPI_Win_unlock()
MPI_Win_fence()
```

All MPI-2 advisory declarations are defined, allowing full MPI-2 compatibility, though currently these have no effect on operation.

C.3 Optimized Collectives

The following collective operations have been optimized for the HP AlphaServer SC Interconnect:

```
MPI_Barrier()           MPI_Bcast()
MPI_Reduce()            MPI_Allreduce()
MPI_Gather()            MPI_Allgather()
MPI_Alltoall()
```

C.4 MPI-IO Support

The following MPI-IO hints are specifically supported for the PFS file system:

striping_factor Number of I/O devices across which the file should be striped. Default value is `pfs_map.pfsmap_slice.ps_count` (from the `PFSIO_GETFSMAP` ioctl). For example:

```
MPI_Info_set(info, "striping_factor", "4");
```

striping_unit The striping unit in bytes. Default value is `pfs_map.pfsmap_stride` (from the `PFSIO_GETFSMAP` ioctl). For example:

```
MPI_Info_set(info, "striping_unit", "65536");
```

cb_buffer_size The buffer size for collective I/O. Default value is `8388608` bytes. For example:

```
MPI_Info_set(info, "cb_buffer_size", "8388608");
```

cb_nodes Number of processes that should perform disk accesses during collective I/O. Default value is `MPI_Comm_size()` called with the communicator used by `MPI_File_open`. For example:

```
MPI_Info_set(info, "cb_nodes", "4");
```

start_iodevice The I/O device from which to start striping the file. Default value is `pfs_map.pfsmap_slice.ps_base` (from the `PFSIO_GETFSMAP` ioctl). For example:

```
MPI_Info_set(info, "start_iodevice", "2");
```

ind_rd_buffer_size Buffer size for data sieving in independent reads. Default value is `2097152` bytes. For example:

```
MPI_Info_set(info, "ind_rd_buffer_size", "2097152");
```

ind_wr_buffer_size Buffer size for data sieving in independent writes. Default value is `1048576` bytes. For example:

```
MPI_Info_set(info, "ind_wr_buffer_size", "1048576");
```

The ROMIO hints `cb_buffer_size`, `cb_nodes`, `ind_rd_buffer_size`, and `ind_wr_buffer_size` are applicable but not specific to our PFS file system.

C.5 MPI Command Line Arguments

The Quadrics MPI library supports a number of command line arguments that control its use of `libelan`. The following arguments are supported

--elan-help Show this help

--elan-version Display version information. Displays the *compile stamp* for the library, the version of libelan MPICH was compiled against and the version of `libelan` the application is linked against. Prints a warning if the two versions of libelan are incompatible.

Output generated by `--elan-version` should be included in MPI problem reports.

--elan-no-check-version

Do not abort when linked with incompatible libelan. If the `libelan` API changes then all previous version of the library are marked as

incompatible. It is possible that the program will still run as normal however as the change might be in a part of the interface that the application does not use. Using this argument disables this check in MPI_Init which will allow programs to run in this case.

--elan-abort-with-core

Whether to dump a core file. Has the same effect as setting the `MPI_ABORT_WITH_CORE` environment variable.

--elan-base-flags *flags*

Pass *flags* to `elan_baseInit()`.

--elan-tport-flags *flags*

Pass *flags* to `elan_tportInit()`.

--elan-group-flags *flags*

Pass *flags* when creating Elan groups and subgroups.

--elan-lock-flags *flags*

Pass *flags* to `elan_lockCtrlInit()`.

--elan-mqueue-flags *flags*

Pass *flags* to `elan_mQueueInit()`.

--elan-global-alloc 0 | 1

Globally allocate queue. Controls the way that MPI allocates the input queue used for message passing. Defaults to true.

--elan-rma 0 | 1 Disable or enable the MPI-2 remote memory access features of the library. Defaults to enabled (1).

--elan-coll *bitmask*

Enable/Disable optimized collectives. `elan-coll` controls which functions use the optimized collectives when using the `COMM_WORLD` communicator.

--elan-coll-sub *bitmask*

Enable/Disable optimized collectives. `elan-coll-sub` controls which functions use the optimized collectives when using communicators other than `COMM_WORLD`.

The command line options for passing flags to Elan initialization functions have only recently been added and are mainly for expansion so there are not currently many valid flags. See the *Elan Programming Manual* for more information. The *flags* value is a bitmask.

The *bitmask* passed to --elan-coll or --elan-coll-sub can contain the following values

0x002	MPI_Barrier
0x004	MPI_Bcast
0x008	MPI_Gather
0x020	MPI_AllGather
0x080	MPI_AlltoAll
0x200	MPI_Reduce

Setting bit 0x1 causes all optimizations to be turned on. If you disable optimized collectives for COMM_WORLD they automatically become disabled for other communicators.

C.6 Support for -i8 and -r8 Fortran compiler flags

As of version MPI.1.24-20 support has been added for Fortran compilers which support the -i8 and -r8 options. Passing -i8 and/or -r8 to the mpif77 or mpif90 scripts will cause these flags to be passed on to the underlying compile commands and the appropriate include and library paths to be set.

For users not using mpif77 and mpif90 the include paths and library paths will have to be set manually. The directory /usr/opt/mpi contains include_i8, include_r8, include_i8r8, lib_i8, lib_r8 and lib_i8r8 directories. The -I, -L and flags should be set appropriately to pick up header files and libraries from these directories rather than /usr/opt/mpi/include and /usr/opt/mpi/lib. The include directories contain only the mpif.h header file and the f90 module directories so you need to add the default include directory to your include path as well. For example, if compiling without -i8 or -r8 you would use:

```
$ f90 -I$MPI/include -o ftest ftest.f -L$MPI/lib -lmpi
```

Where MPI is set to /usr/opt/mpi. If you added the -i8 flag you would prepend the include_i8 directory to the include path and replace lib with lib_i8:

```
$ f90 -i8 -I$MPI/include_i8 -I$MPI/include -o ftest ftest.f -L$MPI/lib_i8 -lmpi
```

And similarly for the -r8 and -i8 -r8 options:

Support for `-i8` and `-r8` Fortran compiler flags

```
$ f90 -r8 -I$MPI/include_r8 -I$MPI/include -o ftest ftest.f -L$MPI/lib_r8 -lmpi
$ f90 -i8 -r8 -I$MPI/include_i8r8 -I$MPI/include -o ftest ftest.f -L$MPI/lib_i8r8 -lmpi
```

Where `MPI` is set to `/usr/opt/mpi`.

D

Shmem Library Routines

D.1 Overview

This appendix itemizes the Shmem routines, noting which are supported and which are not. The routines are grouped in the following categories:

- Initialization (Section D.1.1)
- Cache (Section D.1.2)
- Accessibility (Section D.1.3)
- Synchronization with put and get (Section D.1.4)
- Put and get (Section D.1.5)
- Non blocking put and get (Section D.1.6)
- Strided or indexed put and get (Section D.1.7)
- Collective communications (Section D.1.8)
- Atomic operations (Section D.1.9)
- Remote synchronization (Section D.1.10)
- Remote locking (Section D.1.11)

See the *Quadrics Shmem Programming Manual* for further information on the Shmem library. This document can be downloaded from the Quadrics web site `http://www.quadrics.com`.

D.1.1 Initialization Routines

```
start_pes()                    num_pes()
my_pe()
```

The three initialization routines are fully supported. `start_pes()` expects all of the processes (also known as processing elements or PEs) to have been started by RMS. The function initializes the caller and then synchronizes the caller with the other processes. The rank of the process is returned by `my_pe()`. The total number of processes is returned by `num_pes()`.

D.1.2 Cache Routines

```
shmem_clear_cache_inv()        shmem_udcflush()
shmem_set_cache_inv()          shmem_udcflush_line()
shmem_set_cache_line_inv()
```

The cache routines maintain cache coherency on Cray systems. They are implemented as `NO-OP`s on the HP AlphaServer SC . That is to say, the routines perform no operation; they just return to the caller successfully.

D.1.3 Access Routines

```
shmem_stack()                  shmem_ptr()
```

In general, Shmem supports access to a contiguous region of the virtual address space, starting at the base of the DATA segment and extending up past the BSS and the heap of the process. The stack is mapped, but communicating from the stack requires care as a stack variable it is not necessarily symmetric (at the same address in each process). Calling `shmem_stack()` or `shmem_ptr()` causes the program to exit with a fatal error.

D.1.4 Synchronization Routines

```
shmem_fence()                  shmem_quiet()
```

Both sychronization routines are supported, but `shmem_fence()` is implemented with a call to `shmem_quiet()`. The function `shmem_quiet()` waits for outstanding Elan operations to complete.

D.1.5 Put and Get Routines

```
shmem_short_p()                 shmem_short_g()
shmem_int_p()                   shmem_int_g()
shmem_long_p()                  shmem_long_g()
shmem_float_p()                 shmem_float_g()
shmem_double_p()                shmem_double_g()

shmem_put()                     shmem_get()
shmem_put32()                   shmem_get32()
shmem_put64()                   shmem_get64()
shmem_put128()                  shmem_get128()
shmem_putmem()                  shmem_getmem()

shmem_double_put()              shmem_double_get()
shmem_float_put()               shmem_float_get()
shmem_int_put()                 shmem_int_get()
shmem_long_put()                shmem_long_get()
shmem_longdouble_put()          shmem_longdouble_get()
shmem_longlong_put()            shmem_longlong_get()
shmem_short_put()               shmem_short_get()
```

The put and get routines are all fully supported.

D.1.6 Non-blocking Put and Get Routines

The non-blocking put and get routines are specific to the Quadrics implementation of Shmem;

```
shmem_put_nb()                  shmem_get_nb()
shmem_put32_nb()                shmem_get32_nb()
shmem_put64_nb()                shmem_get64_nb()
shmem_put128_nb()               shmem_get128_nb()
shmem_putmem_nb()               shmem_getmem_nb()

shmem_double_put_nb()           shmem_double_get_nb()
shmem_float_put_nb()            shmem_float_get_nb()
shmem_int_put_nb()              shmem_int_get_nb()
```

(continued on next page)

(continued from previous page)

```
shmem_long_put_nb()              shmem_long_get_nb()
shmem_longdouble_put_nb()        shmem_longdouble_get_nb()
shmem_longlong_put_nb()          shmem_longlong_get_nb()
shmem_short_put_nb()             shmem_short_get_nb()
```

D.1.7 Strided or Indexed Put and Get Routines

```
shmem_iput()                     shmem_iget()
shmem_iput32()                   shmem_iget32()
shmem_iput64()                   shmem_iget64()
shmem_iput128()                  shmem_iget128()
shmem_short_iput()               shmem_short_iget()
shmem_int_iput()                 shmem_int_iget()
shmem_long_iput()                shmem_long_iget()
shmem_longlong_iput()            shmem_longlong_iget()

shmem_float_iput()               shmem_float_iget()
shmem_double_iput()              shmem_double_iget()
shmem_longdouble_iput()          shmem_longdouble_iget()

shmem_ixput()                    shmem_ixget()
shmem_ixput32()                  shmem_ixget32()
```

The strided or indexed put and get routines are fully supported. They are implemented as repeated calls to the basic put and get routines listed in Section D.1.5.

D.1.8 Collective Communications Routines

```
barrier()                        shmem_broadcast()
shmem_barrier_all()              shmem_broadcast32()
shmem_barrier()                  shmem_broadcast64()

shmem_collect()                  shmem_fcollect()
shmem_collect32()                shmem_fcollect32()
shmem_collect64()                shmem_fcollect64()
```

(continued on next page)

(continued from previous page)

```
shmem_short_sum_to_all()          shmem_short_prod_to_all()
shmem_int_sum_to_all()            shmem_int_prod_to_all()
shmem_float_sum_to_all()          shmem_float_prod_to_all()
shmem_double_sum_to_all()         shmem_double_prod_to_all()
shmem_longdouble_sum_to_all()     shmem_longdouble_prod_to_all()
shmem_complexf_sum_to_all()       shmem_complexf_prod__to_all()
shmem_complexd_sum_to_all()       shmem_complexd_prod_to_all()

shmem_short_max_to_all()          shmem_short_min_to_all()
shmem_int_max_to_all()            shmem_int_min_to_all()
shmem_float_max_to_all()          shmem_float_min_to_all()
shmem_double_max_to_all()         shmem_double_min_to_all()
shmem_longdouble_max_to_all()     shmem_longdouble__min_to_all()

shmem_short_or_to_all()           shmem_short_and_to_all()
shmem_int_or_to_all()             shmem_int_and_to_all()

shmem_short_xor_to_all()
shmem_int_xor_to_all()
```

The collective communcations routines are fully supported, including parameters that specify subsets from the set of processes.

D.1.9 Atomic Routines

```
shmem_short_add()                 shmem_short_inc()
shmem_short_fadd()                shmem_short_finc()
shmem_swap()
shmem_short_swap()                shmem_short_mswap()
shmem_int_swap()                  shmem_int_mswap()
shmem_long_swap()                 shmem_long_mswap()
shmem_float_swap()
shmem_double_swap()
shmem_short_cswap()
shmem_int_cswap()
```

(continued on next page)

(continued from previous page)

```
shmem_long_cswap()
```

The atomic routines are supported. Atomicity is only guaranteed if the addresses passed are updated solely by the Shmem routines. Atomic operations are performed by an Elan thread on the target node. Applications using the 16-bit atomic routines on HP AlphaServer SC systems must be compiled with the `-ev6` option..

D.1.10 Synchronization Routines

```
shmem_wait()                    shmem_wait_until()
shmem_wait_nb()                 shmem_poll_nb()
```

The remote synchronization routines are supported. They wait until a store location is modified by a put from another node. This means that synchronization is only guaranteed if the addresses passed are updated solely by Shmem routines.

Warning

An Elan DMA may be fragmented or repeated by the network hardware. When using `shmem_wait()` you should test the last word of data being transfered, earlier words may be modified before the transfer is complete. See the *Quadrics Shmem Programming Manual* for additional considerations on systems with multiple network rails.

The function `shmem_wait_nb()` waits for the completion of a non-blocking transfer. The function `shmem_poll_nb()` polls for the completion of a non-blocking transfer.

D.1.11 Remote Locking

The following remote locking routines are supported:

```
shmem_clear_lock()              shmem_test_lock()
shmem_set_lock()
```

E

Elan Library Environment Variables

E.1 Important MPI and Shmem Settings

The Elan library provides a set of tagged message-passing routines that make use of tagged message ports, known as *tport*s, for point-to-point communications. The following environment variables can be used to tune the behavior of these routines (see Table E.1). Since the MPI library is layered on top of the Elan tagged message-passing routines, these environment variables also affect the performance of MPI programs on HP AlphaServer SC systems.

HP AlphaServer SC Version 2.6 incorporates an optimization to the Shmem library, to allow Shmem operations between processes on the same node to simply copy from the buffer of one process to that of the other without using the Elan device to access the data directly in memory. This intranode optimization is available for anonymous memory only. The SHMEM_SMP_* environment variables listed in Table E.2 affect the operation of this intranode optimization.

Table E.1: MPI and libelan environment variables

Environment Variable	Description	Syntax	Default Value
LIBELAN_ALLOC_SIZE	The size of the `libelan` main memory heap in bytes. This area is used for storing messages for which a matching receive has not yet been posted.	<positive decimal string>	209715200 (200MB)
LIBELAN_GROUP_HWBCAST	This Boolean variable determines whether the hardware broadcast function should be used in preference to the tree-based broadcast mechanism, implemented in software.	ENABLED | DISABLED	ENABLED
LIBELAN_SHM_ENABLE	This Boolean variable determines whether shared-memory should be used for MPI communications within the same node. If disabled, all MPI communications occur *via* the HP AlphaServer SC Interconnect	ENABLED | DISABLED | 1 | 0	ENABLED
LIBELAN_SHM_FIFODEPTH	Determines whether the MPI library uses optimizations that are implemented in the libelan library. If enabled, these optimizations will utilize the hardware broadcast facility of the switch when appropriate.	<positive decimal string>	3
LIBELAN_SHM_FRAGSIZE	The size of each shared-memory FIFO fragment in bytes. The amount of MPI message data that can be in flight between two processes within a node is a function of the size of the FIFO fragment and the number of fragments making up the FIFO.	<positive decimal string>	65536 (64KB)
LIBELAN_TPORT_BIGMSG	The lower threshold size in bytes of a big message. Messages of this size and above are sent synchronously and reduce the buffering of messages on the receiver side.	<positive decimal string>	65536 (64KB)
LIBELAN_WAITTYPE	Specifies whether the process polls (spinwaits) for the arrival of a message or sleeps. By default, the process will poll. The polling period can be modified by setting LIBELAN_WAITTYPE to the number of micro-seconds to poll before sleeping.	WAIT | POLL | <positive decimal string>	10000

(continued on next page)

Table E.1: MPI and libelan environment variables (cont.)

Environment Variable	Description	Syntax	Default Value
MPI_ABORT_WITH_CORE	Controls whether or not an application that calls MPI_Abort will generate core files. If this environment variable is set to FALSE or 0, core files are not generated. If this environment variable is undefined, or set to TRUE or 1, core files are generated. User resource limits for coredumpsize will have precedence over this environment variable in all cases (for more information, see the `setrlimit(2)` reference page). As a result, if coredumpsize is set to 0, core files are not generated.	TRUE | FALSE | 1 | 0	TRUE
MPI_USE_LIBELAN	Determines whether the MPI library uses optimizations that are implemented in the libelan library. If enabled, these optimizations will utilize the hardware broadcast facility of the switch when appropriate.	TRUE | FALSE | 1 | 0	TRUE

Table E.2: Shmem Environment Variables

Environment Variable	Description	Syntax	Default Value
SHMEM_SMP_ENABLE	Controls the Shmem SMP optimization. If this environment variable is set to 0, the SMP optimization is completely disabled and the HP AlphaServer SC Interconnect is used for all Shmem operations. Also, all other SHMEM_SMP_* tuning values are inoperative. If this environment variable is undefined or set to 1, the SMP optimization is enabled.	1 | 0	1

(continued on next page)

Table E.2: Shmem Environment Variables (cont.)

Environment Variable	Description	Syntax	Default Value
SHMEM_SMP_ENABLE_ELANMEM	Controls intranode optimizations for memory obtained *via* elan_allocMain. If this environment variable is set to 0, it will disable SMP optimization of elan_allocMain regions, but not heap. Shmem operations on memory obtained *via* elan_allocMain will be performed over the HP AlphaServer SC Interconnect, while Shmem operations on heap memory will be performed *via* SMP-optimized paths, where possible. If this environment variable is set to 1, Shmem operations on memory obtained *via* elan_allocMain will also be performed *via* SMP-optimized paths, where possible.	1 | 0	1
SHMEM_SMP_DYNAMICMEM	Specifies the limit to which the window into a peer process's heap memory can extend beyond the limit that existed when `shmem_init()` was called. For example, setting this variable to 512M will allow an extra 512MB of heap memory to use the intranode optimization. The default value is zero map only what is malloc'd at `shmem_init()` time. This value should be increased to the maximum memory usage of the program if the program calls malloc after calling `shmem_init()`. If this variable is not set, the HP AlphaServer SC Interconnect will be used for operations on memory outside the reserved range.	<n> | <n>K | <n>M where <n> is a positive integer	0

E.2 Troubleshooting

Information and advice on a number of common problems follows.

E.2.1 Message Buffering

MPI programs that send large numbers of messages without performing matching receives will eventually run out of system buffer memory. If this happens, one or more of the following error message will be displayed:

```
tportBuf: Main memory exhausted ...
elan3_alloc: Exhausted Elan allocator space
```

You can put off (or in some cases avoid) this problem by increasing the size of the buffer pool (or by setting the environment variable `LIBELAN_TPORT_BIGMSG` but, in general, you should not rely on system buffering: it uses up memory and reduces performance. If you set a smaller value of `LIBELAN_TPORT_BIGMSG` then messages may not be buffered.

E.2.2 Thread out of memory error

Code fails with the following error indicating some Elan/MPI problem:

```
0 THRD: elan3_alloc: Exhausted ALLOC 452f10 type 71434240 allocator space (0) :
wanted 209707008 bytes
THRD: ASSERT FAILURE(0): elan3_free() line 1290 file alloc.c
0 THRD: elan3_free: INCORRECT BLK RETURNED 4419000
```

The application code is relying on massive amounts of system buffering. The Elan allocator (Main memory) has been asked for 209707008 bytes. The default allocator size is only 200MB so it looks like you have run out.

Check that the application is not over-relying on system buffering in MPI by making sure they have posted sufficient `MPI_(I)Recv()` calls to match the incoming messages.

Other MPI implementations do not have the Elan's ability to overlap comms and compute and hence problems like this do not show up.

An alternate approach is to increase the size of the Elan allocator by setting the environment variable `LIBELAN_ALLOC_SIZE` to a larger value, but this is not addressing the real issue.

A second alternative is to send more (or all) messages synchronously by setting the environment variable `LIBELAN_TPORT_BIGMSG` to, for example, 4k or 8k (the default is 64k). This prevents system buffering of messages whose size exceeds the BIGMSG size.

E.2.3 Use of `mmap(2)`

You will not get optimal performance when sending MPI messages from buffers allocated with `mmap(2)`, this memory is not mapped into the Elan by default and the data must be copied to a system buffer before it can be transmitted.

E.2.4 Fortran Interface to Shmem

The length arguments to the FORTRAN API versions of `shmem_get4` (and other Shmem commands) all require that a 64-bit integer (`size_t`) be used as the length. If you pass a 32-bit integer as the length, your program may fail with Elan exceptions regarding the "addressability" of a certain address and length, as shown in the following example:

```
ELAN_EXCEPTION @ 0: 14 (Invalid argument)
elan_get: source 140009420 len 0x200000008 is not elan addressable
```

Glossary

Abbreviations

API Application Program Interface — specification of interface to software package (library).

CFS Cluster File System — the file system for Tru64 UNIX clusters.

CGI Common Gateway Interface — a standard method for generating HTML pages dynamically from an application so that a Web server and a Web browser can exchange information. A CGI script can be written in any language and can access various types of data, for example, a SQL database.

CPU Central Processing Unit — the part of the computer that executes the machine instructions that make up the various user and system programs.

CRC Cyclic Redundancy Check — a method of error detection.

CVS Concurrent Versions System — a revision control utility for managing software releases and controlling the concurrent editing of files by multiple software developers.

DIMM Dual In-Line Memory Module.

DMA Direct Memory Access — high performance I/O technique where peripherals read/write memory directly and not through a CPU.

GNU GNU's Not UNIX — A UNIX-like development effort of the Free Software Foundation, headed by Richard Stallman.

HTML HyperText Markup Language — a generic markup language, comprising a set of tags, that enables structured documents to be delivered over the World Wide Web and viewed by a browser.

HTTP HyperText Transfer Protocol — a communications protocol commonly used between a Web server and a Web browser together with a URL (Uniform Resource Locator).

LED Light-Emitting Diode.

MIMD Multiple Instruction, Multiple Data — parallel processing computer architecture characterized as having multiple processors each (potentially) executing a different instruction sequence on different data.

MMU Memory Management Unit — part of CPU that provides protection between user processes and support for virtual memory.

MPI Message Passing Interface — parallel processing API.

MPP Massively Parallel Processing — processing that involves the use of a large number of processors in a coordinated fashion.

PCI Peripheral Component Interconnect — the Elan is connected to a node through this interface.

PDF Portable Document Format — the page description language used by Adobe Acrobat, derived from PostScript, for displaying pages on the screen.

PIO Programmed input/output — I/O directed through a CPU (see also DMA).

PTE Page Table Entry — an entry in the page table which maps the base address of a page to physical memory.

RISC Reduced Instruction Set Computer — a computer whose machine instructions represent relatively simple operations that can be executed very quickly.

RMS Resource Management System — Quadrics software for managing clusters of UNIX nodes.

SDRAM Synchronous Dynamic Random Access Memory — high performance computer memory architecture.

Shmem A one-sided (put/get) inter-process communication interface used on high-performance parallel systems.

SMP Symmetric MultiProcessor — a computer whose main memory is shared by more than one processor.

SNMP Simple Network Management Protocol — a protocol used to monitor and control devices on the Internet.

SQL Structured Query Language — a database language.

TLB Translation Lookaside Buffer — part of the MMU that caches the result of virtual to physical address translations to minimize translation times in subsequent accesses to the same page.

URL Uniform Resource Locator — a standard protocol for addressing information on the World Wide Web.

UTC Coordinated Universal Time[1] — on UNIX systems it is represented as the time elapsed in seconds since January 1st, 1970 at 00:00:00.

Terms

barrier A synchronization point in a parallel computation that all of the processes must reach before they are allowed to continue.

bisectional bandwidth
The worst case bandwidth across the diameter of the network.

critical section A section of program statements that can yield incorrect results if more than one thread tries to execute the section at the same time.

Elan capability An Elan capability describes the user's rights to run a parallel program. It includes information on the distribution of processes over nodes and Elan contexts and a private (random) key .

Elan event A parallel-processing synchronization primitive implemented by the Elan card.

[1]Used to be called GMT.

Elan memory The SDRAM on the Elan card.

Flit A communications cycle unit of information.

HTTP cookies Cookies provide a general mechanism that HTTP server-side connections use to store and to retrieve information on the client side of the connection.

machine The term machine is used as a shorthand for HP AlphaServer SC System. The machine name (for example `atlas`) is the name of the HP AlphaServer SC system and its nodes are named `atlas0`, `atlas1`, ...

main memory The memory normally associated with the main processor, that is to say, memory on the CPU's high speed memory bus.

main processor The main CPU (or CPUs for a multi-processor) of a node, an Alpha™ 21264 for example.

management network

A private network used by the RMS daemons for control and diagnostics.

multirail system

A system that has more than one Elan card connected to each node, each Elan card being connected to a different switch network.

multi-threaded program

A multi-threaded program is one that is constructed such that, during its execution, multiple sequences of instructions are executed concurrently (possibly by different CPUs). Each thread of execution has a separate stack but otherwise they all share the same address space.

node A system with memory, one or more CPUs and one or more Elan cards running an instance of the operating system.

poll Loop and check on each loop whether a specified event has occurred.

rail A network connecting together one adapter in each node. A HP AlphaServer SC Interconnect system may have multiple rails.

rank An integer value that identifies a single process from a set of parallel processes.

reduce Combine the results of a parallel computation into a single value.

remote memory The memory (Elan card or main) of a node when accessed by another node over the network.

resource A set of CPUs allocated to a user to run one or more parallel jobs.

slice A local copy of a global object.

switch network The network constructed from the Elan cards and Elite cards.

thread An independent sequence of execution. Every host process has at least one thread.

virtual memory A feature provided by the operating system, in conjunction with the MMU, that provides each process with a private address space that may be larger than the amount of physical memory accessible to the CPU.

virtual process A (possibly multi-threaded) component of a parallel program executing on a node.

word A 64-bit value on Alpha, IA64 and Opteron platforms, 32-bit on IA32.

Index

A

allocate, 3-23, A-2

C

commands, 2-2
  allocate, 3-23
  prun, 3-7
  rinfo, 3-3
  rmsexec, 3-27
  rmsquery, A-1
compiling, 4-35
configuration, 3-3
  active, 2-2
configurations, 3-1, 3-7
core files, 3-2, 3-18

D

daemons, 2-3
database, 2-3
Database Server, 2-3
DCE credentials, 3-2, 3-20
dce_login, 3-20
debugging, 4-8
documentation
  feedback, 1-3

E

environment variables
  Elan, E-1
  MPI, E-1
  RMS_DEBUG, A-16
  RMS_EXITTIMEOUT, A-16
  RMS_IMMEDIATE, 3-14, A-16
  RMS_IOLABEL, A-16
  RMS_JOBID, A-18
  RMS_KEEP_CORE, 3-19, A-16
  RMS_MEMLIMIT, 3-7, A-17
  RMS_NNODES, A-18
  RMS_NODEID, A-18
  RMS_NPROCS, 3-14, A-18
  RMS_NRAILS, A-17
  RMS_PARTITION, 3-8, A-17
  RMS_PROCFILERANK, A-18
  RMS_PROJECT, 3-7, 3-14, A-17
  RMS_RANK, 3-14, A-18
  RMS_RESOURCE_ID, 3-18
  RMS_RESOURCEID, A-18
  RMS_RSH_LIMIT, A-17
  RMS_SSH, A-17
  RMS_STATINTERVAL, A-17
  RMS_STDERRMODE, 3-13, A-17
  RMS_STDINMODE, 3-12, A-17
  RMS_STDOUTMODE, 3-13, A-17
  RMS_TIMELIMIT, A-18
  Shmem, E-3
error messages, 3-22
Event Manager, 2-3

H

help, 2-5
home directory, 2-5

J

jobs, 3-4

L

load balancing, 3-27, A-29
logging in, 2-4
LSF, B-1
    bsub, B-2
    flags, B-8
    load indices, B-9
    scheduling, B-5

M

machine, 3-3
Machine Manager, 2-3
manual pages, 2-5
memory limits, 3-14
    environment variables, A-17
MPI, 4-2
    collectives, C-2
    command line args, C-3
    compiling C programs, 4-3
    compiling Fortran programs, 4-4
    example, 4-4, 4-13
    library functions, 4-13, C-1
    MPI-IO, C-2
    one-sided, C-1
    running programs, 4-4

N

node names, 3-5

P

parallel programs, 2-6, 3-1
Partition Manager, 2-3
partitions, 2-2, 2-6, 3-3, 3-8
    types, 2-7
passwords, 2-4
pfloader, A-7
priorities, 2-7
process
    controlling, 2-6, 3-1
    distribution, 3-8
    rank, 3-2
    tags, 3-12
process file, A-7, A-11
Process Manager, 2-3
projects, 2-7
prompt
    changing, 3-25
prun, 3-7, A-7
    exit status, 3-2, 3-16
    input redirection, 3-11
    nodes, 3-8
    output redirection, 3-12
    processes, 3-8
    selecting a partition, 3-8
    stats collection, 3-19
    stdio handling, 3-11

R

rank, 3-2, 4-13, 4-25
resources, 3-4
    allocation, 3-23, A-2
    quotas, 3-6
rinfo, 3-3, A-22
    jobs and resources, 3-16
    memory limits, 3-14
RMS Daemon, 2-3
rmsexec, 3-27, A-29
rmshost, 2-2

rmsloader, 3-2
rmsquery, A-31

S

scgenkeytab, 3-21
scheduling
    space sharing, 2-6
    time slicing, 2-6
sequential programs, 3-27
shell
    &, 3-24
    background jobs, 3-24
    glob, 3-5
    prompt, 3-25
    pwd, 3-12
Shmem, 4-5
    compiling C programs, 4-7
    compiling Fortran programs, 4-7
    example, 4-24
    library functions, 4-25, D-1
    running programs, 4-7
SQL, 2-3
Switch Network Manager, 2-3

T

TotalView, 4-8
tracing, 4-12
Transaction Log Manager, 2-3
troubleshooting, 3-21
    error messages, 3-22

U

UNIX, 2-4
    man, 2-5
    more, 2-5
    size, 3-15
    telnet, 2-4

V

Vampir, 4-12