Bright Cluster Manager 5.1

User Manual

Revision: 341

Date: Tue, 06 Jul 2010

Table of Contents

Preface

Welcome to the User Manual for the Bright Cluster Manager 5.1 cluster environment. This manual is intended for users of a cluster running Bright Cluster Manager.

This manual covers the basics of using the Bright Cluster Manager user environment to run compute jobs on the cluster. Although it does cover some aspects of general Linux usage, it is by no means comprehensive in this area. Readers are advised to make themselves familiar with the basics of a Linux environment.

Our manuals constantly evolve to match the development of the Bright Cluster Manager environment, the addition of new hardware and/or applications and the incorporation of customer feedback. Your input as a user and/or administrator is of great value to us and we would be very grateful if you could report any comments, suggestions or corrections to us at `manuals@brightcomputing.com`.

1

Introduction

1.1 What is a Beowulf Cluster

Beowulf is the earliest surviving epic poem written in English. It is a story about a hero of great strength and courage who defeated a monster called Grendel. Nowadays, Beowulf is a multi computer architecture used for parallel computations. It is a system that usually consists of one head node and one or more slave nodes connected together via Ethernet or some other type of network. It is a system built using commodity hardware components, like any PC capable of running Linux, standard Ethernet adapters, and switches.

Beowulf also uses commodity software like the Linux operating system, the GNU C compiler and Message Passing Interface (MPI). The head node controls the whole cluster and serves files and information to the slave nodes. It is also the cluster's console and gateway to the outside world. Large Beowulf machines might have more than one head node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases slave nodes in a Beowulf system are dumb; the dumber the better.

Nodes are configured and controlled by the head node, and do only what they are told to do. One of the main differences between Beowulf and a Cluster of Workstations (COW) is the fact that Beowulf behaves more like a single machine rather than many workstations. In most cases slave nodes do not have keyboards or monitors, and are accessed only via remote login or possibly serial terminal. Beowulf nodes can be thought of as a CPU + memory package which can be plugged into the cluster, just like a CPU or memory module can be plugged into a motherboard.

This manual is intended for cluster users who need a quick introduction to the Bright Beowulf Cluster Environment. It explains how to use the MPI and batch environments, how to submit jobs to the queuing system, and how to check job progress. The specific combination of hardware and software installed may differ depending on the specification of the cluster. This manual may refer to hardware, libraries or compilers not relevant to the environment at hand.

1.2 Physical hardware layout of a Cluster

A Beowulf Cluster consists of a login, compile and job submission node, called the head (or master) node, and one or more compute nodes, nor-

mally referred to as slave (or worker) nodes. A second (fail-over) head node may be present in order to take control of the cluster in case the main head node fails. Furthermore, a second fast network may also have been installed for high performance communication between the (head and the) slave nodes (see figure 1.1).

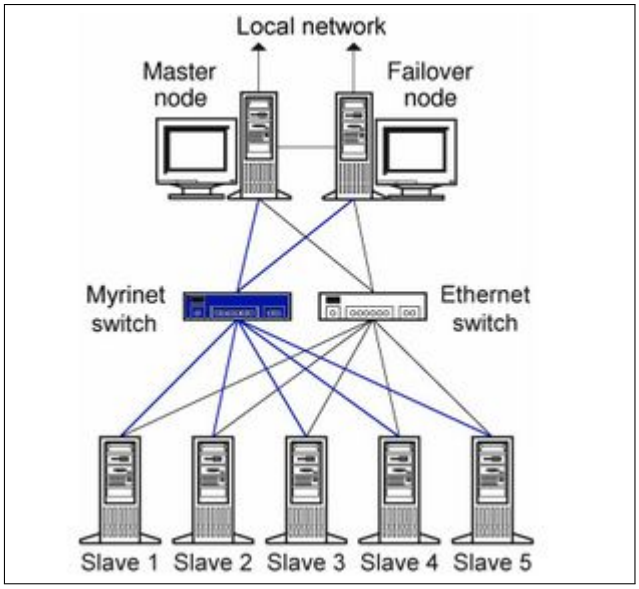

Figure 1.1: Cluster layout

The login node is used to compile software, to submit a parallel or batch program to a job queuing system and to gather/analyse results. Therefore, it should rarely be necessary for a user to log on to one of the slave nodes and in some cases slave node logins are disabled altogether. The head, login and slave nodes communicate with each other through an Ethernet network. Usually a Gigabit Ethernet is in use, capable of transmitting information at a maximum rate of 1000 Megabits/s.

Sometimes an additional network is added to the cluster for even faster communication between the slave nodes. This particular network is mainly used for programs dedicated to solving large scale computational problems, which may require multiple machines and could involve the exchange of vast amounts of information. One such network topology is InfiniBand, capable of transmitting information at a maximum rate of 40 Gigabits/s and 1.2us latency on small packets.

Applications relying on message passing benefit greatly from lower latency. The fast network is always complementary to a slower Ethernet based network.

2

Cluster Usage

2.1 Login To Your Environment

The login node is the node you where can log in to and work from. Simple clusters have a single login node, but large clusters sometimes have multiple login nodes to improve reliability of the cluster. In most clusters, the login node is also the master node from where the cluster is monitored and installed. On the login node you are able to:

- compile your code
- develop applications
- submit applications to the cluster for execution
- monitor running applications

To login using a Unix-like operating system, you can use a terminal. Then type:

```
$ ssh myname@cluster.hostname
```

On a Windows operating system, you can download a SSH client, for instance, PuTTY, and enter the cluster's address and click connect. Enter your username and password when prompted.

If your administrator has changed the default SSH port from 22 to something else, you can specify the port with the `-p <port>` option:

```
$ ssh -p <port> <user>@<cluster>
```

Optionally, you may change your password after logging in using the `passwd` command:

```
$ passwd
```

2.2 Setting Up Your Environment

By default, each user uses the bash shell interpreter. Each time you login, a file named `.bashrc` is executed to set up the shell environment.

In this file, you can add commands, load modules and add custom environment settings you want to use each time you login.

Also, your `.bashrc` is executed by each slave node you run your jobs on, so the application executed will run under the same environment you started with.

For example if you want to use the Open64 compiler each time you login, you may edit the `.bashrc` file with `nano`, `emacs` or `vi` and add the following line:

```
module add open64
```

Then log out and log back in again. You should now have your new environment.

For more further information on environment modules, see section 2.3.

2.3 Environment Modules

On a complex computer system with often a wide choice of software packages and software versions it can be quite hard to set up the correct environment to manage this. For instance, managing several MPI software packages on the same system or even different versions of the same MPI software package is almost impossible on a standard SuSE or Red Hat system as many software packages use the same names for executables and libraries.

As a user you could end up with the problem that you could never be quite sure which libraries have been used for the compilation of a program as multiple libraries with the same name may be installed. Very often a user would like to test new versions of a software package before permanently installing the package. Within Red Hat or SuSE this would be quite a complex task to achieve. Environment modules make this process much easier.

2.3.1 Available commands

```
$ module
   (no arguments)              print usage instructions
   avail                       list available software modules
   li                          currently loaded modules
   load <module name>          add a module to your environment
   add <module name>           add a module to your environment
   unload <module name>        remove a module
   rm <module name>            remove a module
   purge                       remove all modules
   initadd                     add module to shell init script
   initrm                      remove module from shell init script
```

2.3.2 Using the commands

To see the modules loaded into your environment, type

```
$ module li
```

To load a module use the add or load command. You can specify a list of modules by spacing them.

```
$ module add shared open64 openmpi/open64
```

Please note, although version numbers are shown in the `module av` output, it is not necessary to specify version numbers, unless multiple

versions are available for a module. When no version is specified, the latest will be chosen.

To remove one or multiple, use the `module unload` or `module rm` command.

To remove all modules from your environment, use the `module purge` command.

If you are unsure what the module is you can check with `module whatis`

```
$ module whatis sge
sge                  : Adds sge to your environment
```

2.4 Compiling Applications

Compiling application is usually done on the head node or login node. Typically, there are several compilers available on the master node. A few examples: GNU compiler collection, Open64 compiler, Pathscale compilers, Intel compilers, Portland Group compilers. The following table summarizes the available compiler commands on the cluster:

Language	GNU	Open64	Portland	Intel	Pathscale
C	`gcc`	`opencc`	`pgcc`	`icc`	`pathcc`
C++	`g++`	`openCC`	`pgCC`	`icc`	`pathCC`
Fortran77	`gfortran`	`-`	`pgf77`	`ifort`	`-`
Fortran90	`gfortran`	`openf90`	`pgf90`	`ifort`	`pathf90`
Fortran95	`gfortran`	`openf95`	`pgf95`	`ifort`	`pathf95`

Although commercial compilers are available through as packages in the Bright Cluster Manager YUM repository, a license is needed in order to make use of them. Please note that GNU compilers are the de facto standard on Linux and are installed by default and do not require a license. Also Open64 is installed by default on RedHat based installations of Bright Cluster Manager.

To make a compiler available to be used in your shell commands, the appropriate module must be loaded first. See section 2.3 for more information on environment modules. On most clusters two versions of GCC are available:

- The version of GCC that comes natively with the Linux distribution
- The latest version suitable for general use

To use the latest version of GCC, the `gcc` module must be loaded. To revert to the version of GCC that comes natively with the Linux distribution, the `gcc` module must be unloaded.

The commands referred to in the table above are specific for batch type (single processor) applications. For parallel applications it is preferable to use MPI based compilers. The necessary compilers are automatically available after choosing the parallel environment (MPICH, MVAPICH, OpenMPI, etc.). The following compiler commands are available:

Language	C	C++	Fortran77	Fortran90	Fortran95
Command	mpicc	mpiCC	mpif77	mpif90	mpiCC

Please check the documentation of the compiler and the makefile to view which optimization flags are available. Usually there is a `README`, `BUILDING` or `INSTALL` file available with software packages.

2.4.1 Mixing Compilers

Bright Cluster Manager comes with multiple OpenMPI packages for different compilers. However, sometimes it is desirable to mix compilers, for example to combine `gcc` with `pathf90`. In such cases it is possible to override the compiler by setting an environment variable, for example:

```
export OMPI_MPICC=gcc
export OMPI_MPIF90=pathf90
```

Variables that you may set are OMPI_MPICC, OMPI_MPIFC, OMPI_MPIF77, OMPI_MPIF90 and OMPI_MPICXX.

3

Using MPI

The available MPI implementations for MPI-1 are MPICH and MVAPICH. For MPI-2 this is MPICH2 and MVAPICH2. OpenMPI supports both implementations. These MPI compilers can be compiled with GCC, Open64, Intel, PGI and/or Pathscale. Depending on your cluster, you can have several interconnects at your disposal: Ethernet (GE), Infiniband (IB) or Myrinet (MX).

Also depending on your cluster configuration, you can load the different MPI implementations which were be compiled with different compilers. By default all GCC and Open64 compiled MPI implementations are installed.

You can derive the interconnect and compiler from the module or compiler name, e.g: `openmpi-geib-intel-64.x86_64` (OpenMPI compiled for both Gigabit Ethernet (GE) and Infiniband (IB). Compiled with the Intel compiler for 64 bits architecture) Please see `module av` for a complete list of available compilers on your cluster.

3.1 Interconnects

Jobs can use a certain network for intra-node communication.

3.1.1 Gigabit Ethernet

Gigabit Ethernet is the interconnect that is most commonly available. For Gigabit Ethernet, you do not need any additional modules or libraries. The OpenMPI, MPICH and MPICH2 implementations will work over Gigabit Ethernet.

3.1.2 InfiniBand

Infiniband is a high performance switched fabric which is characterized by its high throughput and low latency. OpenMPI, MVAPICH and MVAPICH2 are suitable MPI implementations for InfiniBand.

3.2 Selecting an MPI implementation

To select an MPI implementation, you must load the appropriate module:

- `mpich/ge/gcc/64/1.2.7`
- `mpich/ge/open64/64/1.2.7`

- mpich2/smpd/ge/gcc/64/1.1.1p1
- mpich2/smpd/ge/open64/64/1.1.1p1
- mvapich/gcc/64/1.1
- mvapich/open64/64/1.1
- mvapich2/gcc/64/1.2
- mvapich2/open64/64/1.2
- openmpi/gcc/64/1.3.3
- openmpi/open64/64/1.3.3

Once you have added the appropriate MPI module to your environment, you can start compiling your applications. The mpich, mpich2 and openmpi implementations may be used on Ethernet. On Infiniband, mvapich, mvapich2 and openmpi may be used. The openmpi MPI implementation will attempt to use Infiniband, but will revert to Ethernet if Infiniband is not available.

3.3 Example MPI run

This example covers a MPI run, which you can run inside and outside of a queuing system.

To use mpirun, you must load the relevant environment modules. This example will use mpich over Gigabit Ethernet (ge) compiled with GCC.

```
$ module add mpich/ge/gcc
```

To use InfiniBand, use for example:

```
$ module add mvapich/gcc/64/1.1
```

Depending on the libraries and compilers installed on your system, the availability of these packages might differ. To see a full list on your system, type module av .

3.3.1 Compiling and Preparing Your Application

The code must be compiled with an MPI compiler. Please find the correct compiler command in the table.

Language	C	C++	Fortran77	Fortran90	Fortran95
Command	mpicc	mpiCC	mpif77	mpif90	mpiCC

This example will use the MPI C compiler.

```
$ mpicc myapp.c
```

This will produce a binary a.out which can then be executed using the mpirun command.

3.3.2 Creating a Machine File

A machine file contains a list of nodes which can be used by an MPI program. Usually the workload management system will create a machine file based on the nodes that were allocated for a job. However, if you are running an MPI application "by hand", you are responsible for creating a machine file manually. Depending on the MPI implementation the layout of this file may differ.

If you choose to have the workload management system allocate nodes for your job and you may skip creating a machine file.

Machine files can generally be created in two ways:

- Listing the same node several times to indicate that more than one process should be started on each node:

```
node001
node001
node002
node002
```

- Listing nodes once, but with a suffix for the number of CPU cores to use on each node:

```
node001:2
node002:2
```

In both examples two CPUs on `node001` and `node002` will be used.

3.3.3 Running the Application

Using `mpirun` outside of the workload management system to run the application:

```
$ mpirun -np 4 -machinefile hosts.txt ./a.out
0: We have 4 processors
0: Hello 1! Processor 1 reporting for duty

0: Hello 2! Processor 2 reporting for duty

0: Hello 3! Processor 3 reporting for duty
```

To run the application through the workload management system, a job script is needed. The workload management system is responsible for generating a machine file.
The following is an example for PBS (Torque)

```
#!/bin/sh
mpirun -np 4 -machinefile $PBS_NODEFILE ./a.out
```

An example for SGE:

```
#!/bin/sh
mpirun -np 4 -machinefile $TMP/machines ./a.out
```

Running jobs through a workload management system will be discussed in detail in chapter 4. Appendix A contains a number of simple MPI programs.

4

Workload Management

A workload management system (also know and a queueing system, job scheduler or batch system) manages the available resources such as CPUs and memory. It also manages jobs which have been submitted by users.

4.1 Workload Management Basics

A workload management system primarily manages the cluster resources and jobs. Every workload management system fulfills some basic tasks, such as:

- Monitor the status of the nodes (up, down, load average)
- Monitor all available resources (available cores, memory on the nodes)
- Monitor the jobs state (queued, on hold, deleted, done)
- Control the jobs (freeze/hold the job, resume the job, delete the job)

Some advanced options in workload management systems can prioritize jobs and add checkpoints to freeze a job.

Whenever a job is submitted, the workload management system will check on the resources requested by the jobscript. It will assign cores and memory to the job and send the job to the nodes for computation. If the required number of cores or memory are not yet available, it will queue the job until these resources become available.

The workload management system will keep track of the status of the job and return the resources to the available pool when a job has finished (either deleted, crashed or successfully completed).

The primary reason why workload management systems exist, is so that users do not manually have to keep track of who is running jobs on which nodes in a cluster. Users may still run jobs on the compute nodes outside of the workload management system. However, this is not recommended on a production cluster.

Jobs can only be submitted through a script; a *jobscript*. This script looks very much like an ordinary shell script and you can put certain commands and variables in there that are needed for your job; e.g. load relevant modules or set environment variables. You can also put in some directives for the workload management system, for example, request certain resources, control the output, set an email address.

Bright Cluster Manager comes either with SGE or PBS (i.e. Torque/Maui) pre-configured. These workload management systems will be described in chapters 5 and 6 respectively.

5

SGE

Sun Grid Engine (SGE) is a workload management and job scheduling system first developed to manage computing resources by Sun Microsystems. SGE has both a graphical interface and command line tools for submitting, monitoring, modifying and deleting jobs.

SGE uses 'jobscripts' to submit and execute jobs. Various settings can be put in the scriptfile such as number of processors, resource usage and application specific variables.

The steps for running a job through SGE:

- Create a jobscript
- Select the directives to use
- Add your scripts and applications and runtime parameters
- Submit it to the workload management system

5.1 Writing a Job Script

You can not submit a binary directly to SGE, you will need a jobscript for that. A jobscript can contain various settings and variables to go with your application. Basically it looks like:

```
#!/bin/bash
#$ Script options  # Optional script directives
shellscripts       # Optional shell commands
application        # Application itself
```

5.1.1 Directives

It is possible to specify options ('directives') with SGE with "`#$`" in your script. Please note the difference in the jobscript file:

Directive	Treated as
`#`	Comment in shell and SGE
`#$`	Comment in shell, directive in SGE
`# $`	Comment in shell and SGE

5.1.2 SGE jobscript options

Available environment variables

```
$HOME - Home directory on execution machine
$USER - User ID of job owner
$JOB_ID - Current job ID
$JOB_NAME - Current job name; see the -N option
$HOSTNAME - Name of the execution host
$TASK_ID - Array job task index number
```

5.1.3 Job script examples

Given are some job scripts. Each job script can use a number of variables and directives. Job script options

The following script options can be defined in the jobscript:

```
#$ {option} {parameter}
```

Available options:

Option / parameter	Description
[-a date_time]	request a job start time
[-ac context_list]	add context variable(s)
[-A account_string]	account string in accounting record
[-b y[es]|n[o]]	handle command as binary
[-c ckpt_selector]	define type of checkpointing for job
[-ckpt ckpt-name]	request checkpoint method
[-clear]	skip previous definitions for job
[-cwd]	use current working directory
[-C directive_prefix]	define command prefix for job script
[-dc simple_context_list]	remove context variable(s)
[-dl date_time]	request a deadline initiation time
[-e path_list]	specify standard error stream path(s)
[-h]	place user hold on job
[-hard]	consider following requests ‘‘hard’’
[-help]	print this help
[-hold_jid job_identifier_list]	define jobnet interdependencies
[-i file_list]	specify standard input stream file(s)
[-j y[es]|n[o]]	merge stdout and stderr stream of job
[-js job_share]	share tree or functional job share
[-l resource_list]	request the given resources
[-m mail_options]	define mail notification events
[-masterq wc_queue_list]	bind master task to queue(s)
[-notify]	notify job before killing/suspending it
[-now y[es]|n[o]]	start job immediately or not at all
[-M mail_list]	notify these e-mail addresses
[-N name]	specify job name
[-o path_list]	specify standard output stream path(s)
[-P project_name]	set job’s project
[-p priority]	define job’s relative priority
[-pe pe-name slot_range]	request slot range for parallel jobs
[-q wc_queue_list]	bind job to queue(s)
[-R y[es]|n[o]]	reservation desired
[-r y[es]|n[o]]	define job as (not) restartable
[-sc context_list]	set job context (replaces old context)
[-soft]	consider following requests as soft
[-sync y[es]|n[o]]	wait for job to end and return exit code

```
[-S path_list]                  command interpreter to be used
[-t task_id_range]              create a job-array with these tasks
[-terse]                        tersed output, print only the job-id
[-v variable_list]              export these environment variables
[-verify]                       do not submit just verify
[-V]                            export all environment variables
[-w e|w|n|v]                    verify mode (error|warning|none|just verify) for jobs
[-@ file]
```

Single node example script

An example script for SGE.

```
#!/bin/sh
#$ -N sleep
#$ -S /bin/sh
# Make sure that the .e and .o file arrive in the
# working directory
#$ -cwd
#Merge the standard out and standard error to one file
#$ -j y
sleep 60
echo Now it is: `date`
```

Parallel example script

For parallel jobs you will need the pe environment assigned to the script. Depending on the interconnect, you might have the choice between a number of parallel environments such as MPICH (ethernet) or MVAPICH (InfiniBand).

```
#!/bin/sh
#
# Your job name
#$ -N My_Job
#
# Use current working directory
#$ -cwd
#
# Join stdout and stderr
#$ -j y
#
# pe (Parallel environment) request. Set your number of processors here.
#$ -pe mpich NUMBER_OF_CPUS
#
# Run job through bash shell
#$ -S /bin/bash

# If modules are needed, source modules environment:
. /etc/profile.d/modules.sh

# Add any modules you might require:

module add shared

# The following output will show in the output file. Used for debugging.

echo ``Got $NSLOTS processors.''
```

```
echo ‘‘Machines:’’
cat $TMPDIR/machines

# Use MPIRUN to run the application
mpirun -np $NSLOTS -machinefile $TMPDIR/machines ./application
```

5.2 Submitting a Job

Please load the SGE module first so you can access the SGE commands:

```
$ module add shared sge
```

With SGE you can submit a job with qsub. The qsub command has the following syntax:

```
qsub [ options ] [ scriptfile | -- [ script args ]]
```

After completion (either successful or not), output will be put in your current directory, appended with the job number which is assigned by SGE. By deault, there is an error and an output file.

```
myapp.e#{JOBID}
myapp.o#{JOBID}
```

5.2.1 Submitting to a specific queue

Some clusters have specific queues for jobs which run are configured to house a certain type of job: long and short duration jobs, high resource jobs, or a queue for a specific type of node.

To see which queues are available on your cluster use qstat:

```
qstat -g c
CLUSTER QUEUE   CQLOAD   USED    RES  AVAIL  TOTAL aoACDS  cdsuE
--------------------------------------------------------------------
long.q          0.01      0      0    144    288      0    144
default.q       0.01      0      0    144    288      0    144
```

Then submit the job, e.g. to the long.q queue:

```
qsub -q long.q sleeper.sh
```

5.3 Monitoring a Job

You can view the status of your job with qstat. In this example the Sleeper script has been submitted. Using qstat without options will only display a list of jobs with no queue status options. When using ’-f’, more information will be displayed.

```
$ qstat
job-ID  prior   name       user         state submit/start at     queue   slots
-----------------------------------------------------------------------------
    249 0.00000 Sleeper1   root         qw    12/03/2008 07:29:00         1
    250 0.00000 Sleeper1   root         qw    12/03/2008 07:29:01         1
    251 0.00000 Sleeper1   root         qw    12/03/2008 07:29:02         1
    252 0.00000 Sleeper1   root         qw    12/03/2008 07:29:02         1
    253 0.00000 Sleeper1   root         qw    12/03/2008 07:29:03         1
```

You can see more details with the -f option (full output):

- The Queuetype `qtype` can be Batch (B) or Interactive (I).
- The `used/tot` or `used/free` column is the count of used/free slots in the queue.
- The `states` column is the state of the queue.

```
$ qstat -f
queuename                      qtype used/tot. load_avg arch          states
---------------------------------------------------------------------------------
all.q@node001.cm.cluster     BI    0/16      -NA-     lx26-amd64    au
---------------------------------------------------------------------------------
all.q@node002.cm.cluster     BI    0/16      -NA-     lx26-amd64    au

############################################################################
 - PENDING JOBS - PENDING JOBS - PENDING JOBS - PENDING JOBS - PENDING JOBS
############################################################################
    249 0.55500 Sleeper1   root         qw    12/03/2008 07:29:00     1
    250 0.55500 Sleeper1   root         qw    12/03/2008 07:29:01     1
```

Job state can be:

- d(eletion)
- E(rror)
- h(old)
- r(unning)
- R(estarted)
- s(uspended)
- S(uspended)
- t(ransfering)
- T(hreshold)
- w(aiting)

The queue state can be:

- u(nknown) if the corresponding `sge_execd` cannot be contacted
- a(larm) - the load threshold is currently exceeded
- A(larm) - the suspend threshold is currently exceeded
- C(alendar suspended) - see `calendar_conf`
- s(uspended) - see qmod
- S(ubordinate)
- d(isabled) - see qmod
- D(isabled) - see `calendar_conf`
- E(rror) - `sge_execd` was unable to locate the `sge_shepherd` - use qmod to fix it.

- o(rphaned) - for queue instances

By default the `qstat` command shows only jobs belonging to the current user, i.e. the command is executed with the option `-u $user`. To see also jobs from other users, use the following format:

```
$ qstat -u ‘‘*’’
```

5.4 Deleting a Job

You can delete a job in SGE with the following command

```
$ qdel <jobid>
```

The job-id is the number assigned by SGE when you submit the job using `qsub`. You can only delete your own jobs. Please note that `qdel` deletes the jobs regardless of whether the job is running or spooled.

6

PBS

The Portable Batch System (PBS or Torque) is a workload management and job scheduling system first developed to manage computing resources at NASA. PBS has both a graphical interface and command line tools for submitting, monitoring, modifying and deleting jobs.

Torque uses PBS 'jobscripts' to submit and execute jobs. Various settings can be put in the scriptfile such as number of processors, resource usage and application specific variables.

The steps for running a job through PBS:

- Create a jobscript
- Select the directives to use
- Add your scripts and applications and runtime parameters
- Submit it to the workload management system

6.1 Writing a Job Script

To use Torque, you create a batch job which is a shell script containing the set of commands you want to run. It also contains the resource requirements for the job. The batch job script is then submitted to PBS. A job script can be resubmitted with different parameters (e.g. different sets of data or variables).

Torque/PBS has several options (directives) which you can put in the jobscript. The following is commented out for PBS (note the space):

```
# PBS
```

The following is commented is a PBS directive:

```
#PBS
```

Please note that the file is essentially a shellscript. You can change the shell interpreter to another shell interpreter by changing the first line to your preferred shell. You can specify to PBS to run the job in a different shell then what the shellscript is using by adding the -S directive.

By default the error and output log are jobname.e# and jobname.o#. This can be changed by using the -e and -o directives.

The queueing system has a walltime per queue, please ask your administrator about the value. If you exceed the walltime you will get the errormessage:

```
=>> PBS: job killed: walltime x(running time)x exceeded limit x(settime)x
```

6.1.1 Sample script

```
#!/bin/bash
#
#PBS -l walltime=1:00:00
#PBS -l mem=500mb
#PBS -j oe

cd ${HOME}/myprogs
myprog a b c
```

The directives with `-l` are resource directives, which specify arguments to the `-l` option of qsub. In the above example script, a job time of one hour and at least 500Mb are requested. The directive `-j oe` requests standard out and standard error to be combined in the same file. PBS stops reading directives at the first blank line. The last two lines simply say to change to the directory myprogs and then run the executable `myprog` with arguments `a b c`. Additional directives

You can specify the number of nodes and the processors per nodes (`ppn`). If you do not specify the any, the default will be 1 node, 1 core.

Here are some examples how the resource directive works.

If you want 8 cores, and it does not matter how the cores are allocated (e.g. 8 per node or 1 on 8 nodes) you can just specify `#PBS -l nodes=8`

6.1.2 Resource directives

2 nodes, with 1 processor per node	`#PBS -l nodes=2:ppn=1`
10 nodes with 8 processors per node	`#PBS -l nodes=10:ppn:8`
Request memory	`#PBS -l mem=500mb`
Set a maximum runtime	`(#PBS -l walltime=hh:mm:ss)`

6.1.3 Job directives

Merge output and error	`#PBS -j o`
Change the error / output	`#PBS -e jobname.err`
	`#PBS -o jobname.log`
Set a job name.	`#PBS -N jobname`
Mail events to user	`#PBS -m #PBS -M myusername@myaddress`
Events :	`(a)bort`
	`(b)egin`
	`(e)nd`
	`(n) do not send email`
Specify the queue	`#PBS -q`
Change login shell	`#PBS -S`

6.1.4 Sample batch submission script

This is an example script to test your queueing system. The walltime is 1 minute. This means the script will run at most 1 minute.

The $PBS_NODEFILE array can be used in your script. This array is created and appended with hosts by the queueing system.

```
#!/bin/bash
#PBS -lwalltime=1:00
#PBS -l nodes=4:ppn=2
echo finding each node I have access to
for node in ‘cat ${PBS_NODEFILE}‘ ; do
        /usr/bin/ssh $node hostname
done
```

This is a small example script, which can be used to submit non-parallel jobs to PBS. Sample PBS script for Infinipath:

```
#!/bin/bash
#!
#! Sample PBS file
#!
#! Name of job

#PBS -N MPI

#! Number of nodes (in this case 8 nodes with 4 CPUs each)
#! The total number of nodes passed to mpirun will be nodes*ppn
#! Second entry: Total amount of wall-clock time (true time).
#! 02:00:00 indicates 02 hours

#PBS -l nodes=8:ppn=4,walltime=02:00:00

#! Mail to user when job terminates or aborts
#PBS -m ae

# If modules are needed, source modules environment:
. /etc/profile.d/modules.sh

# Add any modules you might require:
module add shared

#! Full path to application + application name
application=’’<application>’’

#! Run options for the application
options=’’<options>’’

#! Work directory
workdir=’’<work dir>’’



###############################################################
### You should not have to change anything below this line ####
###############################################################

#! change the working directory (default is home directory)

cd $workdir
```

```
echo Running on host `hostname`
echo Time is `date`
echo Directory is `pwd`
echo PBS job ID is $PBS_JOBID
echo This jobs runs on the following machines:
echo `cat $PBS_NODEFILE | uniq`

#! Create a machine file for MPI
cat $PBS_NODEFILE | uniq > machine.file.$PBS_JOBID

numnodes=`wc $PBS_NODEFILE | awk '{ print $1 }'`

#! Run the parallel MPI executable (nodes*ppn)

echo ``Running mpirun -np $numnodes -machinefile \
       machine.file.$PBS_JOBID $application $options''
mpirun  -np $numnodes -machinefile machine.file.$PBS_JOBID \
        $application $options
```

As can be seen in the script, a machine file is built using the `$PBS_NODEFILE` variable. This variable is supplied by the queuing system and contains the node names that are reserved by the queuing system for running the job. The configuration file is given a unique name (`/tmp/$PBS_JOBID.conf`) in order to make sure that users can run multiple programs concurrently.

6.2 Submitting a Job

To submit a job to the PBS workload management system, please load the following modules:

```
$ module add shared torque maui
```

The command `qsub` is used to submit jobs. The command will return a unique job identifier, which is used to query and control the job and to identify output. See the respective man-page for more options.

```
qsub <options> <jobscript>
-a   datetime  run the job at a certain time
-l   list      request  certain resource(s)
-q   queue     jobs is run in this queue
-N   name      name of job
-S   shell     shell to run job under
-j   oe        join output and error files
```

To submit the job:

```
$ qsub mpirun.job
```

or to submit to in a specific queue:

```
$ qsub -q testq mpirun.job
```

Your job will be managed by Maui, which you can view with `showq`. If you want to delete your job, you can use `qdel`.

6.3 Output

The output will be in your current working directory. By default, error output is written to <scriptname>.e<jobid> and the application output is written to <scriptname>.o<jobid>. You can also concatenate the error and output file in one by using the -j oe directive. If you have specified a specific output/error file, the output can be found there.

A number of useful links:

- Torque examples http://bmi.cchmc.org/resources/software/torque/examples
- PBS script files: http://www.ccs.tulane.edu/computing/pbs/pbs.phtml
- Running PBS jobs and directives: http://www.nersc.gov/nusers/systems/franklin/running_jobs/
- Submitting PBS jobs: http://amdahl.physics.purdue.edu/using-cluster/node23.html

6.4 Monitoring a Job

To use the commands in this example, you will need to load the torque module.

```
$ module add torque
```

The main component is qstat, which has several options. In this example, the most frequently used options will be discussed.

In PBS/Torque, the command qstat -an shows what jobs are currently submitted in the queuing system and the command qstat -q shows what queues are available. An example output is:

```
mascm4.cm.cluster:

Job ID               Username Queue    Jobname  SessID NDS   TSK Memory Time  S
-------------------- -------- -------- ---------------- ------ ----- --- ------
24.mascm4.cm     cvsuppor testq    TestJobPBS   10476     1  --    --  02:00 R
   node004/1+node004/0
25.mascm4.cm     cvsuppor testq    TestJobPBS     --      1  --    --  02:00 Q
    --
```

You can see the queue it has been assigned to, the user, jobname, the number of nodes, requested time (-lwalltime), jobstate (S) and Elapsed time. In this example, you can see one running (R), and one is queued (Q).

Job states	Description
C	Job is completed (regardless of success or failure)
E	Job is exiting after having run
H	Job is held
Q	job is queued, eligible to run or routed
R	job is running
S	job is suspend
T	job is being moved to new location
W	job is waiting for its execution time

To view the queues, use the `-q` parameter. In this example, there is one job running in the testq queue and 4 are queued.

```
$ qstat -q

server: master.cm.cluster

Queue            Memory CPU Time Walltime Node  Run Que Lm  State
---------------- ------ -------- -------- ----  --- --- --  -----
testq              --      --    23:59:59   --    1   4 --   E R
default            --      --    23:59:59   --    0   0 --   E R
                                                ----- -----
                                                    1     4
```

Similar output can be viewed using `showq` which comes from the Maui scheduler. To use this command you will need to load the `maui` module. In this example, one dual-core node is available (1 node, 2 processors), one job is running and 3 are queued (in Idle state).

```
$ showq
ACTIVE JOBS--------------------
JOBNAME            USERNAME      STATE  PROC   REMAINING            STARTTIME

45                 cvsupport    Running     2     1:59:57  Tue Jul 14 12:46:20

     1 Active Job        2 of    2 Processors Active (100.00%)
                         1 of    1 Nodes Active      (100.00%)

IDLE JOBS----------------------
JOBNAME            USERNAME      STATE  PROC     WCLIMIT            QUEUETIME

46                 cvsupport       Idle     2     2:00:00  Tue Jul 14 12:46:20
47                 cvsupport       Idle     2     2:00:00  Tue Jul 14 12:46:21
48                 cvsupport       Idle     2     2:00:00  Tue Jul 14 12:46:22

3 Idle Jobs

BLOCKED JOBS----------------
JOBNAME            USERNAME      STATE  PROC     WCLIMIT            QUEUETIME


Total Jobs: 4   Active Jobs: 1   Idle Jobs: 3   Blocked Jobs: 0
```

6.5 Viewing job details

With `qstat -f` you will get full output of your job. You can see in the output what the jobname is, where the error and output files are stored, and various other settings and variables.

```
$ qstat -f
Job Id: 19.mascm4.cm.cluster
    Job_Name = TestJobPBS
    Job_Owner = cvsupport@mascm4.cm.cluster
    job_state = Q
    queue = testq
    server = mascm4.cm.cluster
    Checkpoint = u
    ctime = Tue Jul 14 12:35:31 2009
    Error_Path = mascm4.cm.cluster:/home/cvsupport/test-package/TestJobPBS
    .e19
    Hold_Types = n
    Join_Path = n
    Keep_Files = n
    Mail_Points = a
    mtime = Tue Jul 14 12:35:31 2009
    Output_Path = mascm4.cm.cluster:/home/cvsupport/test-package/TestJobPB
    S.o19
    Priority = 0
    qtime = Tue Jul 14 12:35:31 2009
    Rerunable = True
    Resource_List.nodect = 1
    Resource_List.nodes = 1:ppn=2
    Resource_List.walltime = 02:00:00
    Variable_List = PBS_O_HOME=/home/cvsupport,PBS_O_LANG=en_US.UTF-8,
    PBS_O_LOGNAME=cvsupport,
    PBS_O_PATH=/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/sbin:/usr/
    sbin:/home/cvsupport/bin:/cm/shared/apps/torque/2.3.5/bin:/cm/shar
    ed/apps/torque/2.3.5/sbin,PBS_O_MAIL=/var/spool/mail/cvsupport,
    PBS_O_SHELL=/bin/bash,PBS_SERVER=mascm4.cm.cluster,
    PBS_O_HOST=mascm4.cm.cluster,
    PBS_O_WORKDIR=/home/cvsupport/test-package,PBS_O_QUEUE=default
    etime = Tue Jul 14 12:35:31 2009
    submit_args = pbs.job -q default
```

6.6 Monitoring PBS nodes

You can view the nodes available for PBS execution with `pbsnodes`. The following output is from 4 dual processor nodes. See also the `np` value (2). When a node is used exclusively by one script, the state will be job-exclusive. If the node is available to run scripts the state is free.

```
$ pbsnodes -a
node001.cm.cluster
     state = free
     np = 2
     ntype = cluster
     status = ...
```

```
node002.cm.cluster
     state = free
     np = 2
     ntype = cluster
     status = ...

node003.cm.cluster
     state = free
     np = 2
     ntype = cluster
     status = ...

node004.cm.cluster
     state = free
     np = 2
     ntype = cluster
     status = ...
```

6.7 Deleting a Job

In case you would like to delete an already submitted job, this can be done using the `qdel` command. The syntax is:

```
$ qdel <jobid>
```

The job id is printed to your terminal when you submit the job. You can use

```
$ qstat
```

or

```
$ showq
```

to get the job id of your job in case you no longer know it.

7

Using GPUs

Bright Cluster Manager contains several tools which can be used for using GPUs for general purpose computations.

7.1 Packages

A number of different GPU-related packages are included in Bright Cluster Manager:

- `cuda3-driver`: Provides the GPU driver
- `cuda3-libs`: Provides the libraries that come with the driver (libcuda etc)
- `cuda3-toolkit`: Provides the compilers, cuda-gdb and math libraries
- `cuda3-profiler`: Provides the CUDA visual profiler
- `opencl-profiler`: Provides the OpenCL visual profiler
- `cuda3-sdk`: Provides additional tools, development files and source examples

7.2 Using CUDA

There are several modules available relating to CUDA.

- `cuda3/blas and cuda3/blas-emu`: Providing paths and settings for the CUBLAS library.
- `cuda3/fft and cuda3/fft-emu` : Providing paths and settings for the CUFFT library.
- `cuda3/profiler`: The CUDA visual profiler
- `cuda3/toolkit`: Provides paths and settings to compile and link CUDA applications.
- `opencl/profiler`: The OpenCL visual profiler

For the `blas` and `fft` libraries, there is a choice between 2 modes. The `-emu` modules append an 'emulated' flag during the linking process. This means that applications linked with the emu flag will not have the code

executed on the GPU, but the entire code will execute on the host itself. This is useful for running on hardware without a CUDA capable GPU for testing purposes.

For general usage and compilation it will be sufficient to load just the CUDA3/toolkit module.

```
module add cuda3/toolkit
```

or shorter:

```
module add cuda3
```

7.2.1 Using different CUDA versions

Depending on the cluster, you may have older CUDA packages installed. You can then choose between `cuda` or `cuda3` modules.

You can find out your current CUDA Runtime versions by running `deviceQuery`

```
$ deviceQuery
Device 0: "Tesla T10 Processor"
  CUDA Driver Version:                           3.0
  CUDA Runtime Version:                          2.30

$ deviceQuery
Device 0: "Tesla T10 Processor"
  CUDA Driver Version:                           3.0
  CUDA Runtime Version:                          3.0
```

The toolkit comes with the necessary tools and compilers to compile CUDA C code.

Documentation on how to get started, the various tools and usage of the CUDA suite can be found in the `$CUDA_INSTALL_PATH/doc` directory.

7.3 Using OpenCL

OpenCL functionality is provided with the `cuda3/toolkit`. Please load this environment module if you would like to use OpenCL. Note that CUDA packages older then 3.0 do not have OpenCL support.

Examples of OpenCL can be found in the `$CUDA_SDK/OpenCL` directory.

7.4 Compiling code

Both CUDA and OpenCL involve running code on different **platforms**:

- host: with one or more CPU's
- devices: with one or more CUDA enabled GPU's.

Accordingly, both the host and device manage their own memory space, and it is possible to copy data between them. Please see both the CUDA and OpenCL Best Practices Guides in the `doc` directory in the CUDA toolkit for more information on how to handle both platforms and their limitations.

To compile code and link the objects for both the host system and the GPU, one should use `nvcc`. Nvcc can distinguish between the two and

can hide the details from the developer. To compile the host code, Nvcc will use gcc automatically.

```
nvcc [options] <inputfile>
```

A simple example to compile CUDA host code:

```
nvcc test.cu
```

Most used options are:

- -deviceemu: Device emulation mode.
- -arch=sm_13: When your CUDA device supports double-precision floating-point, you can enable this switch.

When the device does not support double-precision floating-point or the flag is not set, one can notice messages like:

```
warning : Double is not supported. Demoting to float
```

See nvcc man page for help.

For programming examples, please see the CUDA SDK (`$CUDA_SDK/C/`)

For OpenCL, compiling your code can be done by linking against the OpenCL library:

```
gcc test.c -lOpenCL
g++ test.cpp -lOpenCL
nvcc test.c -lOpenCL
```

7.5 Available tools

7.5.1 CUDA gdb

The CUDA gdb can be started using: `cuda-gdb`

7.5.2 nvidia-smi

`nvidia-smi` can be used to allow exclusive access to the GPU. This means only one application can run on a GPU. By default, a GPU will allow multiple running applications.

Syntax:

```
nvidia-smi [OPTION1 [ARG1]] [OPTION2 [ARG2]] ...
```

The steps to lock a GPU:

- List GPU's
- Select a GPU
- Lock GPU
- After use, release the GPU

After setting the compute rule on the GPU, the first application which executes on the GPU will block out all others attempting to run. This application does not necessarily have to be the one started by the user which locked the GPU!

To list the GPU's, you can use the `-L` argument:

```
$ nvidia-smi -L
GPU 0: (05E710DE:068F10DE)  Tesla T10 Processor  (S/N: 706539258209)
GPU 1: (05E710DE:068F10DE)  Tesla T10 Processor  (S/N: 2486719292433)
```

To set the ruleset the GPU:

```
$ nvidia-smi -g 0 -c 1
```

The ruleset may be one of the following:

- 0 - Normal mode
- 1 - COMPUTE exclusive mode (only one COMPUTE context is allowed to run on the GPU)
- 2 - COMPUTE prohibited mode (no COMPUTE contexts are allowed to run on the GPU)

To check the state of the GPU:

```
$ nvidia-smi -g 0 -s
COMPUTE mode rules for GPU 0: 1
```

In this example, GPU0 is locked, and there is a running application using GPU0. A second application attemping to run on this GPU will not be able to run on this GPU.

```
$ histogram  --device=0
main.cpp(101) : cudaSafeCall() Runtime API error :
no CUDA-capable device is available.
```

After use, you can unlock the GPU to allow multiple users:

```
nvidia-smi -g 0 -c 0
```

7.5.3 CUDA Utility Library

`CUTIL` is a simple utility library designed for use in the CUDA SDK samples. There are 2 parts for CUDA and OpenCL. The locations are:

- $CUDA_SDK/C/lib
- $CUDA_SDK/OpenCL/common/lib

Other applications may also refer to them, the toolkit modules have already been pre-configured accordingly. However, they need to be compiled prior to use. Depending on your cluster, this might have already been done.

```
cd $CUDA_SDK/C
make
cd $CUDA_SDK/OpenCL
make
```

CUTIL provides functions for:

- parsing command line arguments
- read and writing binary files and PPM format images
- comparing arrays of data (typically used for comparing GPU results with CPU)
- timers
- macros for checking error codes
- checking for shared memory bank conflicts

7.5.4 CUDA Hello world example

To compile:

```
[root@gpu01 cuda]# nvcc hello.cu -o hello.exe
[root@gpu01 cuda]# ./hello.exe
Hello World!

/*
** Hello World using CUDA
**
** The string "Hello World!" is mangled then restored using a common CUDA idiom
**
** Byron Galbraith
** 2009-02-18
*/
#include <cuda.h>
#include <stdio.h>

// Prototypes
__global__ void helloWorld(char*);

// Host function
int
main(int argc, char** argv)
{
int i;

// desired output
char str[] = "Hello World!";

// mangle contents of output
// the null character is left intact for simplicity
for(i = 0; i < 12; i++)
str[i] -= i;

// allocate memory on the device
char *d_str;
size_t size = sizeof(str);
cudaMalloc((void**)&d_str, size);

// copy the string to the device
cudaMemcpy(d_str, str, size, cudaMemcpyHostToDevice);

// set the grid and block sizes
dim3 dimGrid(2); // one block per word
dim3 dimBlock(6); // one thread per character

// invoke the kernel
helloWorld<<< dimGrid, dimBlock >>>(d_str);

// retrieve the results from the device
cudaMemcpy(str, d_str, size, cudaMemcpyDeviceToHost);

// free up the allocated memory on the device
cudaFree(d_str);
```

```
// everyone's favorite part
printf("%s\n", str);

return 0;
}

// Device kernel
__global__ void
helloWorld(char* str)
{
// determine where in the thread grid we are
int idx = blockIdx.x * blockDim.x + threadIdx.x;

// unmangle output
str[idx] += idx;
}
```

A

MPI Examples

A.1 Hello world

A quick application to test the MPI compilers and the network.

```
/*
 ‘‘Hello World’’ Type MPI Test Program
*/
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;

  /* all MPI programs start with MPI_Init; all 'N' processes exist thereafter */
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /* find out how big the SPMD world is */
  MPI_Comm_rank(MPI_COMM_WORLD,&myid); /* and this processes' rank is */

  /* At this point, all the programs are running equivalently, the rank is used to
     distinguish the roles of the programs in the SPMD model, with rank 0 often used
     specially... */
  if(myid == 0)
  {
    printf(‘‘%d: We have %d processors\n’’, myid, numprocs);
    for(i=1;i<numprocs;i++)
    {
      sprintf(buff, ‘‘Hello %d! ‘‘, i);
      MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
    }
    for(i=1;i<numprocs;i++)
```

```
    {
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
      printf(``%d: %s\n'', myid, buff);
    }
  }
  else
  {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, ``Processor %d `', myid);
    strcat(buff, idstr);
    strcat(buff, ``reporting for duty\n'');
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }

  /* MPI Programs end with MPI Finalize; this is a weak
     synchronization point */
  MPI_Finalize();
  return 0;
}
```

A.2 MPI skeleton

The sample code below contains the complete communications skeleton for a dynamically load balanced master/slave application. Following the code is a description of some of the functions necessary for writing typical parallel applications.

```
include <mpi.h>
#define WORKTAG     1
#define DIETAG     2
main(argc, argv)
int argc;
char *argv[];
{
        int         myrank;
        MPI_Init(&argc, &argv);   /* initialize MPI */
        MPI_Comm_rank(
        MPI_COMM_WORLD,   /* always use this */
        &myrank);      /* process rank, 0 thru N-1 */
        if (myrank == 0) {
                master();
        } else {
                slave();
        }
        MPI_Finalize();       /* cleanup MPI */
}

master()
{
        int     ntasks, rank, work;
        double       result;
        MPI_Status     status;
        MPI_Comm_size(
```

```
        MPI_COMM_WORLD,   /* always use this */
        &ntasks);          /* #processes in application */
/*
* Seed the slaves.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                work = /* get_next_work_request */;
                MPI_Send(&work,         /* message buffer */
                1,              /* one data item */
                MPI_INT,        /* data item is an integer */
                rank,           /* destination process rank */
                WORKTAG,        /* user chosen message tag */
                MPI_COMM_WORLD);/* always use this */
        }



/*
* Receive a result from any slave and dispatch a new work
* request work requests have been exhausted.
*/
        work = /* get_next_work_request */;
        while (/* valid new work request */) {
                MPI_Recv(&result,       /* message buffer */
                1,              /* one data item */
                MPI_DOUBLE,     /* of type double real */
                MPI_ANY_SOURCE, /* receive from any sender */
                MPI_ANY_TAG,    /* any type of message */
                MPI_COMM_WORLD, /* always use this */
                &status);       /* received message info */
                MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,
                WORKTAG, MPI_COMM_WORLD);
                work = /* get_next_work_request */;
        }
/*
* Receive results for outstanding work requests.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        }
/*
* Tell all the slaves to exit.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
        }
}

slave()
{
        double              result;
        int                 work;
        MPI_Status          status;
        for (;;) {
                MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
```

```
                MPI_COMM_WORLD, &status);
/*
* Check the tag of the received message.
*/
                if (status.MPI_TAG == DIETAG) {
                        return;
                }
                result = /* do the work */;
                MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }
}
```

Processes are represented by a unique rank (integer) and ranks are numbered 0, 1, 2, ..., N-1. MPI_COMM_WORLD means all the processes in the MPI application. It is called a communicator and it provides all information necessary to do message passing. Portable libraries do more with communicators to provide synchronisation protection that most other systems cannot handle.

A.3 MPI Initialization and Finalization

As with other systems, two functions are provided to initialise and clean up an MPI process:

```
MPI_Init(&argc, &argv);
MPI_Finalize( );
```

A.4 Who Am I ? Who Are They ?

Typically, a process in a parallel application needs to know who it is (its rank) and how many other processes exist. A process finds out its own rank by calling:

```
MPI_Comm_rank( ):
Int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

The total number of processes is returned by `MPI_Comm_size( )`:

```
int nprocs;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

A.5 Sending messages

A message is an array of elements of a given data type. MPI supports all the basic data types and allows a more elaborate application to construct new data types at runtime. A message is sent to a specific process and is marked by a tag (integer value) specified by the user. Tags are used to distinguish between different message types a process might send/receive. In the sample code above, the tag is used to distinguish between work and termination messages.

```
MPI_Send(buffer, count, datatype, destination, tag, MPI_COMM_WORLD);
```

A.6 Receiving messages

A receiving process specifies the tag and the rank of the sending process. `MPI_ANY_TAG` and `MPI_ANY_SOURCE` may be used optionally to receive a message of any tag and from any sending process.

```
MPI_Recv(buffer, maxcount, datatype, source, tag, MPI_COMM_WORLD, &status);
```

Information about the received message is returned in a status variable. The received message tag is `status.MPI_TAG` and the rank of the sending process is `status.MPI_SOURCE`. Another function, not used in the sample code, returns the number of data type elements received. It is used when the number of elements received might be smaller than `maxcount`.

```
MPI_Get_count(&status, datatype, &nelements);
```

With these few functions, you are ready to program almost any application. There are many other, more exotic functions in MPI, but all can be built upon those presented here so far.