Bright Cluster Manager 6.0

User Manual

Revision:

Date:

Table of Contents

Preface

Welcome to the User Manual for the Bright Cluster Manager 6.0 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

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 queueing system, and how to check job progress. The specific combination of hardware and software installed may differ depending on the specification of the cluster, which means that parts of this manual may not be relevant to the user's particular cluster.

1.1 What Is A Beowulf Cluster?

1.1.1 Background And History

In the history of the English language, Beowulf is the earliest surviving epic poem written in English. It is a story about a hero with the strength of many men who defeated a fearsome monster called Grendel.

In computing, a Beowulf class cluster computer is a multicomputer architecture used for parallel computations, i.e., it uses many computers together so that it has the brute force to defeat fearsome number-crunching problems.

The architecture was first popularized in the Linux community when the source code used for the original Beowulf cluster built at NASA was made widely available. The Beowulf class cluster computer design usually consists of one head node and one or more regular nodes connected together via Ethernet or some other type of network. While the original Beowulf software and hardware has long been superseded, the name given to this basic design remains "Beowulf class cluster computer", or less formally "Beowulf cluster".

1.1.2 Brief Hardware And Software Description

On the hardware side, commodity hardware is generally used in Beowulf clusters to keep costs down. These components are mainly Linux-compatible PCs, standard Ethernet adapters, InfiniBand interconnects, and switches.

On the software side, commodity software is generally used in Beowulf clusters to keep costs down. For example: the Linux operating system, the GNU C compiler and the Message Passing Interface (MPI) standard.

The head node controls the whole cluster and serves files and information to the 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 compute nodes in a Beowulf system are dumb—in general, the dumber the better—with the focus on the processing capability of the node within the cluster, rather than other abilities a computer might generally have. A node may therefore have

- one or several "number-crunching" processors. The processors may also be GPUs
- enough memory to deal with the processes passed on to the node
- a connection to the rest of the cluster

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 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.

1.2 Brief Network Description

A Beowulf Cluster consists of a login, compile and job submission node, called the head, and one or more compute nodes, often referred to as 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) nodes (see figure 1.1).

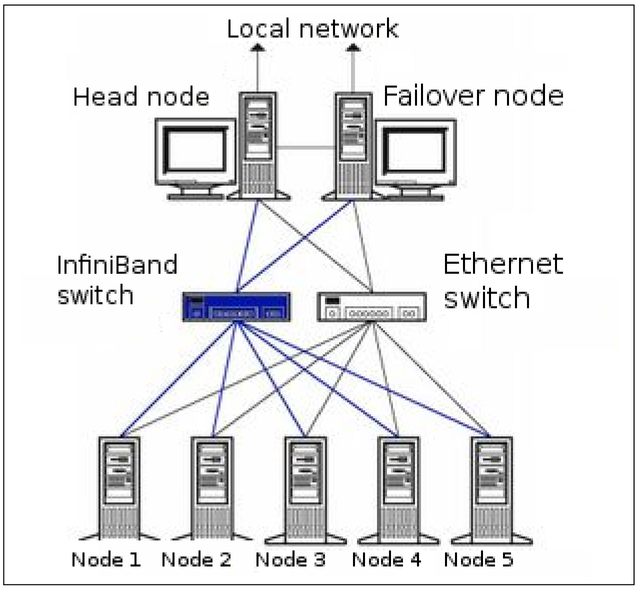


Figure 1.1: Cluster layout

The login node is used to compile software, to submit a parallel or batch program to a job queueing system and to gather/analyze results.

Therefore, it should rarely be necessary for a user to log on to one of the nodes and in some cases node logins are disabled altogether. The head, login and compute nodes usually communicate with each other through a gigabit Ethernet network, capable of transmitting information at a maximum rate of 1000 Mbps. In some clusters 10 gigabit Ethernet (10GE, 10GBE, or 10GigE) is used, capable of up to 10 Gbps rates.

Sometimes an additional network is used by the cluster for even faster communication between the compute 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, commonly capable of transmitting information at a maximum rate of 56Gbps and about 1.2μs latency on small packets, for clusters in 2011.The commonly available maximum transmission rates will increase over the years as the technology advances.

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

2

Cluster Usage

2.1 Login To The Cluster Environment

The login node is the node where the user logs in and works from. Simple clusters have a single login node, but large clusters sometimes have multiple login nodes to improve the reliability of the cluster. In most clusters, the login node is also the head node from where the cluster is monitored and installed. On the login node:

- code can be compiled
- applications can be developed
- applications can be submitted to the cluster for execution
- running applications can be monitored

To carry out an ssh login to the cluster, a terminal can be used from unix-like operating systems:

Example

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

On a Windows operating system, an SSH client such as for PuTTY can be downloaded. The cluster's address must be added, and the connect button clicked. The username and password must be entered when prompted.

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

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

Optionally, after logging in, the password used can be changed using the `passwd` command:

```
$ passwd
```

2.2 Setting Up The User Environment

By default, each user uses the bash shell interpreter. Each time a user login takes place, a file named `.bashrc` is executed to set up the shell environment for the user. The shell environment can be customized to suit user preferences. For example,

- the prompt can be changed to indicate the current host and directory
- the size of the command history file can be increased
- aliases can be added for frequently used command sequences
- environment variables can be created or modified
- the location of software packages and versions that are to be used by a user (the path to a package) can be set

Because there is a huge choice of software packages and versions, it can be hard to set up the right environment variables and paths for software that is to be used. To make setting up the environment easier, Bright Cluster Manager provides preconfigured environment modules (section 2.3).

2.3 Environment Modules

It can be quite hard to set up the correct environment to use a particular software package and version.

For instance, managing several MPI software packages on the same system or even different versions of the same MPI software package is quite difficult for most users on a standard SUSE or Red Hat system because many software packages use the same names for executables and libraries.

A user could end up with the problem of never being 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, using the `module` command, make this process much easier

2.3.1 Available commands

```
$ module help

  Modules Release 3.2.6 2007-02-14 (Copyright GNU GPL v2 1991):

  Usage: module [ switches ] [ subcommand ] [subcommand-args ]

Switches:
        -H|--help               this usage info
        -V|--version            modules version & configuration options
        -f|--force              force active dependency resolution
        -t|--terse              terse    format avail and list format
        -l|--long               long     format avail and list format
        -h|--human              readable format avail and list format
        -v|--verbose            enable  verbose messages
        -s|--silent             disable verbose messages
        -c|--create             create caches for avail and apropos
        -i|--icase              case insensitive
        -u|--userlvl <lvl>      set user level to (nov[ice],exp[ert],adv[anced])
  Available SubCommands and Args:
        + add|load              modulefile [modulefile ...]
        + rm|unload             modulefile [modulefile ...]
        + switch|swap           [modulefile1] modulefile2
        + display|show          modulefile [modulefile ...]
        + avail                 [modulefile [modulefile ...]]
        + use [-a|--append]     dir [dir ...]
        + unuse                 dir [dir ...]
        + update
        + refresh
        + purge
        + list
        + clear
        + help                  [modulefile [modulefile ...]]
        + whatis                [modulefile [modulefile ...]]
        + apropos|keyword       string
        + initadd               modulefile [modulefile ...]
        + initprepend           modulefile [modulefile ...]
        + initrm                modulefile [modulefile ...]
        + initswitch            modulefile1 modulefile2
        + initlist
        + initclear
```

2.3.2 Changing The Current Environment

The modules loaded into the user's environment can be seen with:

```
$ module list
```

Modules can be loaded using the add or load options. A list of modules can be added by spacing them:

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

The "module avail" command lists all modules that are available for loading (some output elided):

Example

```
[fred@bright52 ~]$ module avail
---------------------------- /cm/local/modulefiles -----------------------------
cluster-tools/5.2 dot             module-info       shared
cmd               freeipmi/1.0.2  null              use.own
cmsh              ipmitool/1.8.11 openldap          version

---------------------------- /cm/shared/modulefiles ----------------------------
acml/gcc/64/4.4.0                  intel/compiler/64/12.0/2011.5.220
acml/gcc/mp/64/4.4.0               intel-cluster-checker/1.7
acml/gcc-int64/64/4.4.0            intel-cluster-runtime/3.2
acml/gcc-int64/mp/64/4.4.0         intel-tbb-oss/ia32/30_221oss
acml/open64/64/4.4.0               intel-tbb-oss/intel64/30_221oss
...
```

In the list there are two kinds of modules:

- **local modules**, which are specific to the node, or head node only
- **shared modules**, which are made available from a shared storage, and which only become available for loading after the shared module is loaded.

The shared module is obviously a useful local module, and is therefore loaded for the user by default on a default cluster.

Although version numbers are shown in the "module avail" output, it is not necessary to specify version numbers, unless multiple versions are available for a module[1].

To remove one or more modules, the "module unload" or "module rm" command is used.

To remove all modules from the user's environment, the "module purge" command is used.

The user should be aware that some loaded modules can conflict with others loaded at the same time. For example, loading openmpi/gcc/64/ without removing an already loaded openmpi/gcc/64/ can result in confusion about what compiler opencc is meant to use.

2.3.3 Changing The Default Environment

The initial state of modules in the user environment can be set as a default using the "module init*" subcommands. The more useful ones of these are:

- module initadd: add a module to the initial state
- module initrm: remove a module from the initial state
- module initlist: list all modules loaded initially
- module initclear: clear all modules from the list of modules loaded initially

Example

```
$ module initclear
$ module initlist
bash initialization file $HOME/.bashrc loads modules:
        null
$ module initadd shared gcc/4.4.6 openmpi/gcc/64 torque
$ module initlist
bash initialization file $HOME/.bashrc loads modules:
        null shared gcc/4.4.6 openmpi/gcc/64 torque
```

The new initial state module environment for the user is loaded from the next login onwards.

If the user is unsure about what the module does, it can be checked using "module whatis":

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

The man pages for module gives further details on usage.

[1] For multiple versions, when no version is specified, the alphabetically-last version is chosen. This usually is the latest, but can be an issue when versions move from, say, 9, to 10. For example, the following is sorted in alphabetical order: v1 v10 v11 v12 v13 v2 v3 v4 v5 v6 v7 v8 v9.

2.4 Compiling Applications

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

Language	GNU	Open64	Portland	Intel
C	gcc	opencc	pgcc	icc
C++	g++	openCC	pgCC	icc
Fortran77	gfortran	openf90 -ff77	pgf77	ifort
Fortran90	gfortran	openf90	pgf90	ifort
Fortran95	gfortran	openf95	pgf95	ifort

GNU compilers are the de facto standard on Linux and are installed by default. They do not require a license. AMD's Open64 is also installed by default on Bright Cluster Manager. Commercial compilers by Portland and Intel are available as packages via the Bright Cluster Manager YUM repository, and require the purchase of a license to use them. To make a compiler available to be used in a user's shell commands, the appropriate environment module (section 2.3) must be loaded first. On most clusters two versions of GCC are available:

1. The version of GCC that comes along with the Linux distribution. For example, for CentOS 6.0:

   Example

   ```
   [fred@bright52 ~]$ which gcc; gcc --version | head -1
   /usr/bin/gcc
   gcc (GCC) 4.4.4 20100726 (Red Hat 4.4.4-13)
   ```

2. The latest version suitable for general use that is packaged as a module by Bright Computing:

   Example

   ```
   [fred@bright52 ~]$ module load gcc
   [fred@bright52 ~]$ which gcc; gcc --version | head -1
   /cm/shared/apps/gcc/4.4.6/bin/gcc
   gcc (GCC) 4.4.6
   ```

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 compilers in the preceding table are ordinarily used for applications that run on a single node. However, the applications used may fork, thread, and run across as many nodes and processors as they can access if the application is designed that way.

The standard, structured way of running applications in parallel is to use the MPI-based libraries, which link to the underlying compilers in the preceding table. The underlying compilers are automatically made available after choosing the parallel environment (MPICH, MVAPICH, Open MPI, etc.) via the following compiler commands:

Language	**C**	**C++**	**Fortran77**	**Fortran90**	**Fortran95**
Command	mpicc	mpiCC	mpif77	mpif90	mpif95

2.4.1 Mixing Compilers

Bright Cluster Manager comes with multiple Open MPI packages corresponding to the different available compilers. However, sometimes mixing compilers is desirable. For example, C-compilation may be preferred using `icc` from Intel, while Fortran90-compilation may be preferred using `openf90` from Open64. In such cases it is possible to override the default compiler path environment variable, for example:

```
[fred@bright52 ~]$ module list
Currently Loaded Modulefiles:
  1) null                    3) gcc/4.4.6               5) torque/2.5.5
  2) shared                  4) openmpi/gcc/64/1.4.2
[fred@bright52 ~]$ mpicc --version --showme; mpif90 --version --showme
gcc --version
gfortran --version
[fred@bright52 ~]$ export OMPI_CC=icc; export OMPI_FC=openf90
[fred@bright52 ~]$ mpicc --version --showme; mpif90 --version --showme
icc --version
openf90 --version
```

Variables that may be set are OMPI_CC, OMPI_FC, OMPI_F77, and OMPI_CXX. More on overriding the Open MPI wrapper settings is documented in the man pages of `mpicc` in the environment section.

3

Using MPI

MPI libraries allow the compilation of code so that it can be used over many processors at the same time.

The available MPI implementations for the variant MPI-1 are MPICH and MVAPICH. For the variant MPI-2 they are MPICH2 and MVAPICH2. Open MPI supports both variants. These MPI libaries can be compiled with GCC, Open64, Intel, or PGI.

Also, depending on the cluster, the interconnect available may be: Ethernet (GE), InfiniBand (IB) or Myrinet (MX).

Also depending on the cluster configuration, MPI implementations for different compilers can be loaded. By default MPI implementations that are installed are compiled and made available using both GCC and Open64.

The interconnect and compiler implementation can be worked out from looking at the module and package name. The modules available can be searched through for the compiler variant, and then the package providing it can be found:

Example

```
[fred@bright52 ~]$ # search for modules starting with the name openmpi
[fred@bright52 ~]$ module -l avail 2>&1 | grep ^openmpi
openmpi/gcc/64/1.4.2                              2011/05/03  0:37:51
openmpi/intel/64/1.4.2                            2011/05/02  8:24:28
openmpi/open64/64/1.4.2                           2011/05/02  8:43:13
[fred@bright52 ~]$ rpm -qa | grep ^openmpi
openmpi-ge-gcc-64-1.4.2-108_cm5.2.x86_64
openmpi-geib-open64-64-1.4.2-108_cm5.2.x86_64
openmpi-geib-intel-64-1.4.2-108_cm5.2.x86_64
```

Here, for example,

```
openmpi-geib-intel-64.x86_64
```

implies: Open MPI compiled for both Gigabit Ethernet (“ge”) and InfiniBand (“ib”), compiled with the Intel compiler for a 64-bit architecture.

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, no additional modules or libraries are needed. The Open MPI, 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. Open MPI, MVAPICH and MVAPICH2 are suitable MPI implementations for InfiniBand.

3.2 Selecting An MPI implementation

Once the appropriate compiler module has been loaded, the MPI implementation is selected along with the appropriate library modules. In the following list, <*compiler*> indicates a choice of gcc, intel, open64, or pgi:

- mpich/ge/<*compiler*>/64/1.2.7
- mpich2/smpd/ge/<*compiler*>/64/1.3.2p1
- mvapich/<*compiler*>/64/1.2rc1
- mvapich2/<*compiler*>/64/1.6
- openmpi/<*compiler*>/64/1.4.2
- blas/<*compiler*>/64/1 [1]
- blacs/openmpi/*compiler*/64/1.1patch03
- globalarrays/<*compiler*>/openmpi/64/5.0.2
- gotoblas/<*name of CPU*>[2]/64/1.26

After the appropriate MPI module has been added to the user environment, the user can start compiling 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 an MPI run, which can be run inside and outside of a queuing system.

To use mpirun, the relevant environment modules must be loaded. For example, to use mpich over Gigabit Ethernet (ge) compiled with GCC.

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

Similarly, to use InfiniBand:

[1] Not recommended, except for testing purposes, due to lack of optimization

[2] <*name of CPU*> indicates a choice of the following: barcelona, core, opteron, penryn, or prescott

```
$ module add mvapich/gcc/64/1.2rc1
```

Depending on the libraries and compilers installed on the system, the availability of these packages might differ. To see a full list on the system the command "`module avail`" can be typed.

3.3.1 Compiling And Preparing The Application

The code must be compiled with MPI libraries and an underlying compiler. The correct library command can be found in the following table:

Language	**C**	**C++**	**Fortran77**	**Fortran90**	**Fortran95**
Command	`mpicc`	`mpiCC`	`mpif77`	`mpif90`	`mpif95`

The following example uses the MPI with a C compiler:

```
$ mpicc myapp.c
```

This creates 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.

The workload management system creates a machine file based on the nodes allocated for a job when the job is submitted with the workload manager job submission tool. So if the user chooses to have the workload management system allocate nodes for the job then creating a machine file is not needed.

However, if an MPI application is being run "by hand" outside the workload manager, then the user is responsible for creating a machine file manually. Depending on the MPI implementation the layout of this file may differ.

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
  ```

3.3.3 Running The Application

A Simple Parallel Processing Executable

A simple "hello world" program designed for parallel processing can be built with MPI. After compiling it can be used to send a message about how and where it is running:

```
[fred@bright52 ~]$ cat hello.c
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[])
{
    int id, np, i;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int processor_name_len;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Get_processor_name(processor_name, &processor_name_len);

    for(i=1;i<2;i++)
    {printf("Hello world from process %03d out of %03d, processor name %s\n",
        id, np, processor_name);
    }

    MPI_Finalize();
    return 0;
}
[fred@bright52 ~]$ mpicc hello.c -o hello
[fred@bright52 ~]$ ./hello
Hello world from process 000 out of 001, processor name bright52.cm.cluster
```

However, it still runs on a single processor unless it is submitted to the system in a special way.

Running An MPI Executable In Parallel Without A Workload Manager

After the relevant module files are chosen (section 3.3) for MPI, an executable compiled with MPI libraries runs on nodes in parallel when submitted with `mpirun`. When using `mpirun` manually, outside a workload manager environment, the number of processes `-np`) as well as the number of hosts (`-machinefile`) should be specified. For example, on a 2-compute-node, 4-processor cluster:

Example

```
fred@bright52 ~]$ module add mvapich/gcc/64/1.2rc1   #or as appropriate
fred@bright52 ~]$ mpirun -np 4 -machinefile mpirun.hosts -nolocal hello
Hello world from process 003 out of 004, processor name node002.cm.cluster
Hello world from process 002 out of 004, processor name node001.cm.cluster
Hello world from process 001 out of 004, processor name node002.cm.cluster
Hello world from process 000 out of 004, processor name node001.cm.cluster
```

Here, the `-nolocal` option prevents the executable running on the local node itself, and the file `mpirun.hosts` is a list of node names:

Example

```
[fred@bright52 ~]$ cat mpirun.hosts
node001
node002
```

Running the executable with mpirun as shown does not take the resources of the cluster into account. To handle running jobs with cluster resources is of course what workload managers such as SLURM are designed to do.

Running an application through a workload manager is introduced in Chapter 4.

Appendix A contains a number of simple MPI programs.

4
Workload Management

4.1 What Is A Workload Manager?

A workload management system (also known as a queueing system, job scheduler or batch submission system) manages the available resources such as CPUs, GPUs, and memory for jobs submitted to the system by users.

Jobs are submitted by the users using *job scripts*. Job scripts are constructed by users and include requests for resources. How resources are allocated depends upon policies that the system administrator sets up for the workload manager.

4.2 Why Use A Workload Manager?

Workload managers are used so that users do not manually have to keep track of node usage in a cluster in order to plan efficient and fair use of cluster resources.

Users may still perhaps run jobs on the compute nodes outside of the workload manager, if that is administratively permitted. However, running jobs outside a workload manager tends to eventually lead to an abuse of the cluster resources as more people use the cluster, and thus inefficient use of available resources. It is therefore usually forbidden as a policy by the system administrator on production clusters.

4.3 What Does A Workload Manager Do?

A workload manager uses policies to ensure that the resources of a cluster are used efficiently, and must therefore track cluster resources and jobs. To do this, a workload manager must:

- Monitor the node status (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 managers can prioritize jobs and add checkpoints to freeze a job.

4.4 Job Submission Process

Whenever a job is submitted, the workload management system checks on the resources requested by the job script. It assigns cores and memory to the job, and sends the job to the nodes for computation. If the required number of cores or memory are not yet available, it queues the job until these resources become available. If the job requests resources that are always going to exceed those that can become available, then the job accordingly remains queued indefinitely.

The workload management system keeps track of the status of the job and returns the resources to the available pool when a job has finished (that is, been deleted, has crashed or successfully completed).

4.5 What Do Job Scripts Look Like?

A job script looks very much like an ordinary shell script, and certain commands and variables can be put in there that are needed for the job. The exact composition of a job script depends on the workload manager used, but normally includes:

- commands to load relevant modules or set environment variables
- directives for the workload manager to request resources, control the output, set email addresses for messages to go to
- an execution (job submission) line

When running a job script, the workload manager is normally responsible for generating a machine file based on the requested number of processor cores (np), as well as being responsible for the allocation any other requested resources.

The executable submission line in a job script is the line where the job is submitted to the workload manager. This can take various forms.

Example

For the SLURM workload manager, the line might look like:

```
srun --mpi=mpich1_p4 ./a.out
```

Example

For Torque or PBS Pro it may simply be:

```
mpirun ./a.out
```

Example

For SGE it may look like:

```
mpirun -np 4 -machinefile $TMP/machines ./a.out
```

4.6 Running Jobs On A Workload Manager

The details of running jobs through the following workload managers is discussed next:

- SLURM (Chapter 5)
- SGE (Chapter 6)
- Torque (with Maui or Moab) and PBS Pro (Chapter 7)

4.7 Running Jobs In Cluster Extension Cloud Nodes Using cmsub

Extra computational power from cloud service providers such as Amazon can be used by an appropriately configured cluster managed by Bright Cluster Manager.

If the head node is running outside a cloud services provider, and at least some of the compute nodes are in the cloud, then this "hybrid" cluster configuration is called a Cluster Extension cluster, with the compute nodes in the cloud being the cloud extension of the cluster.

For a Cluster Extension cluster, job scripts to a workload manager should be submitted using the cmsub cluster manager utility. This allows the job to be considered for running on the extension (the cloud nodes). Jobs that are to run on the local regular nodes (not in a cloud) are not dealt with by cmsub.

The environment module (section 2.3) cluster-tools must be loaded in order for the cmsub utility to be available for use.

The basic usage for cmsub is:

```
cmsub [OPTIONS] script
```

Users that are used to running jobs as root should note that the root user cannot usefully run a job with cmsub.

The user can submit some cloud-related values as options to cmsub on the command line, followed by the job script.

Example

```
$ cat myscript1
#!/bin/sh
hostname

$ cmsub --regions=eu-west-1 myscript1
  Upload job id: 1
    User job id: 2
Download job id: 3
```

The cloud-related values can also be specified in a job-directive style format in the job script itself, using the "#CMSUB" tag to indicate a cloud-related option.

Example

```
$ cat myscript2
#!/bin/sh
#CMSUB --regions=us-west-2,eu-west-1
#CMSUB --input-list=/home/user/myjob.in
#CMSUB --output-list=/home/user/myjob.out
#CMSUB --remote-output-list=/home/user/file-which-will-be-created
#CMSUB --input=/home/user/onemoreinput.dat
#CMSUB --input=/home/user/myexec
myexec

$ cmsub myscript2
  Upload job id: 4
    User job id: 5
Download job id: 6
```

The man page for cmsub gives details on the cloud-related option values.

5 SLURM

SLURM (Simple Linux Utility for Resource Management) is a workload management system developed originally at the Lawrence Livermore National Laboratory. It has both a graphical interface and command line tools for submitting, monitoring, modifying and deleting jobs.

SLURM is normally used with *job scripts* to submit and execute jobs. Various settings can be put in the job script, such as number of processors, resource usage and application specific variables.

The steps for running a job through SLURM are to:

- Create the script or executable that will be handled as a job
- Create a job script that sets the resources for the script/executable
- Submit the job script to the workload management system

The details of SLURM usage depends upon the MPI implementation used. The description in this chapter will cover using SLURM's Open MPI implementation, which is quite standard. SLURM documentation can be consulted (`https://computing.llnl.gov/linux/slurm/mpi_guide.html`) if the implementation the user is using is very different.

5.1 Loading SLURM Modules And Compiling The Executable

In section 3.3.3 an MPI "Hello, world!" executable that can run in parallel is created and run in parallel outside a workload manager.

The executable can be run in parallel using the SLURM workload manager. For this, the SLURM module should first be loaded by the user on top of the chosen MPI implementation, in this case Open MPI:

Example

```
[fred@bright52 ~]$ module list
Currently Loaded Modulefiles:
  1) gcc/4.4.6              3) shared
  2) openmpi/gcc/64/1.4.2   4) cuda40/toolkit/4.0.17
[fred@bright52 ~]$ module add slurm; module list
Currently Loaded Modulefiles:
  1) gcc/4.4.6              3) shared                    5) slurm/2.2.4
  2) openmpi/gcc/64/1.4.2   4) cuda40/toolkit/4.0.17
```

The "hello world" executable from section 3.3.3 can then be compiled and run for one task outside the workload manager as:

```
mpicc hello.c -o hello
mpirun -np 1 hello
```

5.2 Running The Executable With salloc

Running it as a job managed by SLURM can be done interactively with the SLURM allocation command, salloc, as follows

```
[fred@bright52 ~]$ salloc mpirun hello
```

SLURM is more typically run as a batch job (section 5.3). However execution via salloc uses the same options, and it is more convenient as an introduction because of its interactive behavior.

In a default Bright Cluster Manager configuration, SLURM autodetects the cores available and by default spreads the tasks across the cores that are part of the allocation request.

To change how SLURM spreads the executable across nodes is typically determined by the options in the following table:

Short Option	**Long Option**	**Description**
-N	--nodes=	Request this many nodes on the cluster. Use all cores on each node by default
-n	--ntasks=	Request this many tasks on the cluster. Defaults to 1 task per node.
-c	--cpus-per-task=	request this many CPUs per task, (not implemented by Open MPI yet)
(none)	--ntasks-per-node=	request this number of tasks per node .

The full options list and syntax for salloc can be viewed with "man salloc".

The requirement of specified options to salloc must be met before the executable is allowed to run. So, for example, if --nodes=4 and the cluster only has 3 nodes, then the executable does not run.

5.2.1 Node Allocation Examples

The following session illustrates and explains some node allocation options and issues for SLURM using a cluster with just 1 compute node and 4 CPU cores:

Default settings: The hello MPI executable with default settings of SLURM runs successfully over the first (and in this case, the only) node that it finds:

```
[fred@bright52 ~]$ salloc mpirun hello
salloc: Granted job allocation 572
Hello world from process 0 out of 4, host name node001
Hello world from process 1 out of 4, host name node001
Hello world from process 2 out of 4, host name node001
Hello world from process 3 out of 4, host name node001
salloc: Relinquishing job allocation 572
```

The preceding output also displays if -N1 (indicating 1 node) is specified, or if -n4 (indicating 4 tasks) is specified.

The node and task allocation is almost certainly not going to be done by relying on defaults. Instead, node specifications are supplied to SLURM along with the executable.

To understand SLURM node specifications, the following cases consider and explain where the node specification is valid and invalid.

Number of nodes requested: The value assigned to the -N|--nodes= option is the number of nodes from the cluster that is requested for allocation for the executable. In the current cluster example it can only be 1. For a cluster with, for example, 1000 nodes, it could be a number up to 1000.

A resource allocation request for 2 nodes with the --nodes option causes an error on the current 1-node cluster example:

```
[fred@bright52 ~]$ salloc -N2 mpirun hello
salloc: error: Failed to allocate resources: Node count specification in\
valid
salloc: Relinquishing job allocation 573
```

Number of tasks requested per cluster: The value assigned to the -n|--ntasks option is the number of tasks that are requested for allocation from the cluster for the executable. In the current cluster example, it can be 1 to 4 tasks. The default resources available on a cluster are the number of available processor cores.

A resource allocation request for 5 tasks with the --ntasks option causes an error because it exceeds the default resources available on the 4-core cluster:

```
[fred@bright52 ~]$ salloc -n5 mpirun hello
salloc: error: Failed to allocate resources: More processors requested t\
han permitted
```

Adding and configuring just one more node to the current cluster would allows the resource allocation to succeed, since an added node would provide at least one more processor to the cluster.

Number of tasks requested per node: The value assigned to the --ntasks-per-node option is the number of tasks that are requested for allocation from each node on the cluster. In the current cluster example, it can be 1 to 4 tasks. A resource allocation request for 5 tasks per node with --ntasks-per-node fails on this 4-core cluster, giving an output like:

```
[fred@bright52 ~]$ salloc --ntasks-per-node=5 mpirun hello
salloc: error: Failed to allocate resources: More processors requested t\
han permitted
```

Adding and configuring another 4-core node to the current cluster would still not allow resource allocation to succeed, because the request is for at least 5 cores per node, rather than per cluster.

Restricting the number of tasks that can run per node: A resource allocation request for 2 tasks per node with the `--ntasks-per-node` option, and simultaneously an allocation request for 1 task to run on the cluster using the `--ntasks` option, runs successfully, although it uselessly ties up resources for 1 task per node:

```
[fre@bright52 ~]$ salloc --ntasks-per-node=2 --ntasks=1 mpirun hello
salloc: Granted job allocation 574
Hello world from process 0 out of 1, host name node005
salloc: Relinquishing job allocation 574
```

The other way round, that is, a resource allocation request for 1 task per node with the `--ntasks-per-node` option, and simultaneously an allocation request for 2 tasks to run on the cluster using the `--ntasks` option, fails because on the 1-cluster node, only 1 task can be allocated resources on the single node, while resources for 2 tasks are being asked for on the cluster:

```
[fred@bright52 ~]$ salloc --ntasks-per-node=1 --ntasks=3 mpirun hello
salloc: error: Failed to allocate resources: Requested node configuratio\
n is not available
salloc: Job allocation 575 has been revoked.
```

5.3 Running The Executable As A SLURM Job Script

Instead of using options appended to the `salloc` command line as in section 5.2, it is usually more convenient to send jobs to SLURM with the `sbatch` command acting on a job script.

A job script is also sometimes called a batch file. In a job script, the user can add and adjust the SLURM options, which are the same as the `salloc` options of section 5.2. The various settings and variables that go with the application can also be adjusted.

5.3.1 SLURM Job Script Structure

A job script submission for the SLURM batch job script format is illustrated by the following:

```
[fred@bright52 ~]$ cat slurmhello.sh
#!/bin/sh
#SBATCH -o my.stdout
#SBATCH --time=30     #time limit to batch job
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=4
module add shared openmpi/gcc/64/1.4.2 slurm
mpirun hello
```

The structure is:

shebang line: shell definition line.

SBATCH lines: optional job script *directives* (section 5.3.2).

shell commands: optional shell commands, such as loading necessary modules.

application execution line: execution of the MPI application using `sbatch`, the SLURM submission wrapper.

In SBATCH lines, ‘‘`#SBATCH`’’ is used to submit options. The various meanings of lines starting with ‘‘`#`’’ are:

Line Starts With	Treated As
`#`	Comment in shell and SLURM
`#SBATCH`	Comment in shell, option in SLURM
`# SBATCH`	Comment in shell and SLURM

After the SLURM job script is run with the `sbatch` command (Section 5.3.4), the output goes into file `my.stdout`, as specified by the ”`-o`” command.

If the output file is not specified, then the file takes a name of the form ”`slurm-<`*`jobnumber`*`>.out`”, where <*jobnumber*> is a number starting from 1.

The command ”`sbatch --usage`” lists possible options that can be used on the command line or in the job script. Command line values override script-provided values.

5.3.2 SLURM Job Script Options

Options, sometimes called ”directives”, can be set in the job script file using this line format for each option:

```
#SBATCH {option} {parameter}
```

Directives are used to specify the resource allocation for a job so that SLURM can manage the job optimally. Available options and their descriptions can be seen with the output of `sbatch --help`. The more overviewable usage output from `sbatch --usage` may also be helpful.

Some of the more useful ones are listed in the following table:

Directive Description	Specified As
Name the job <*jobname*>	`#SBATCH -J <jobname>`
Request at least <*minnodes*> nodes	`#SBATCH -N <minnodes>`
Request <*minnodes*> to <*maxnodes*> nodes	`#SBATCH -N <minnodes>-<maxnodes>`
Request at least <*MB*> amount of temporary disk space	`#SBATCH --tmp <MB>`
Run the job for a time of <*walltime*>	`#SBATCH -t <walltime>`
Run the job at <*time*>	`#SBATCH --begin <time>`
Set the working directory to <*directorypath*>	`#SBATCH -D <directorypath>`
Set error log name to <*jobname.err*>*	`#SBATCH -e <jobname.err>`
Set output log name to <*jobname.log*>*	`#SBATCH -o <jobname.log>`
Mail <*user@address*>	`#SBATCH --mail-user <user@address>`
Mail on any event	`#SBATCH --mail-type=ALL`
Mail on job end	`#SBATCH --mail-type=END`
Run job in partition	`#SBATCH -p <destination>`
Run job using GPU with ID <*number*>, as described in section 8.5.2	`#SBATCH --gres=gpu:<number>`

By default, both standard output and standard error go to a file "`slurm-<%j>.out`", where <%j*> is the job number.

5.3.3 SLURM Environment Variables

Available environment variables include:

```
SLURM_CPUS_ON_NODE - processors available to the job on this node
SLURM_JOB_ID - job ID of executing job
SLURM_LAUNCH_NODE_IPADDR - IP address of node where job launched
SLURM_NNODES - total number of nodes
SLURM_NODEID - relative node ID of current node
SLURM_NODELIST - list of nodes allocated to job
SLURM_NTASKS - total number of processes in current job
SLURM_PROCID - MPI rank (or relative process ID) of the current process
SLURM_SUBMIT_DIR - directory from with job was launched
SLURM_TASK_PID - process ID of task started
SLURM_TASKS_PER_NODE - number of task to be run on each node.
CUDA_VISIBLE_DEVICES - which GPUs are available for use
```

Typically, end users use `SLURM_PROCID` in a program so that an input of a parallel calculation depends on it. The calculation is thus spread across processors according to the assigned `SLURM_PROCID`, so that each processor handles the parallel part of the calculation with different values.

More information on environment variables is also to be found in the man page for `sbatch`.

5.3.4 Submitting The SLURM Job Script

Submitting a SLURM job script created like in the previous section is done by executing the job script with `sbatch`:

```
[fred@bright52 ~]$ sbatch slurmhello.sh
Submitted batch job 703
[fred@bright52 ~]$ cat slurm-703.out
Hello world from process 001 out of 004, processor name node001
...
```

Queues in SLURM terminology are called "partitions". SLURM has a default queue called `defq`. The administrator may have removed this or created others.

If a particular queue is to be used, this is typically set in the job script using the `-p` or `--partition` option:

```
#SBATCH --partition=bitcoinsq
```

It can also be specified as an option to the `sbatch` command during submission to SLURM.

5.3.5 Checking And Changing Queued Job Status

After a job has gone into a queue, the queue status can be checked using the `squeue` command. The job number can be specified with the `-j` option to avoid seeing other jobs. The man page for `squeue` covers other options.

Jobs can be canceled with "`scancel` *<job number>*".

The `scontrol` command allows users to see and change the job directives while the job is still queued. For example, a user may have specified a job, using the `--begin` directive, to start at 10am the next day by mistake. To change the job to start at 10pm tonight, something like the following session may take place:

```
[fred@bright52 ~]$ scontrol show jobid=254 | grep Time
   RunTime=00:00:04 TimeLimit=UNLIMITED TimeMin=N/A
   SubmitTime=2011-10-18T17:41:34 EligibleTime=2011-10-19T10:00:00
   StartTime=2011-10-18T17:44:15 EndTime=Unknown
   SuspendTime=None SecsPreSuspend=0
```

The parameter that should be changed is "`EligibleTime`", which can be done as follows:

```
[fred@bright52 ~]$ scontrol update jobid=254 EligibleTime=2011-10-18T22:00:00
```

An approximate GUI SLURM equivalent to `scontrol` is the `sview` tool. This allows the job to be viewed under its jobs tab, and the job to be edited with a right click menu item. It can also carry out many other functions, including canceling a job.

Webbrowser-accessible job viewing is possible from the workload tab of the `User Portal` (section 9.0.2).

6 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 *job scripts* to submit and execute jobs. Various settings can be put in the job script, such as number of processors, resource usage and application specific variables.

The steps for running a job through SGE are to:

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

6.1 Writing A Job Script

A binary cannot be submitted directly to SGE—a job script is needed for that. A job script can contain various settings and variables to go with the application. A job script format looks like:

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

6.1.1 Directives

It is possible to specify options ('directives') to SGE by using ‘‘#$’’ in the script. The difference in the meaning of lines that start with the ‘‘#’’ character in the job script file should be noted:

Line Starts With	Treated As
#	Comment in shell and SGE
#$	Comment in shell, directive in SGE
# $	Comment in shell and SGE

6.1.2 SGE Environment Variables

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; (like the -N option in qsub, qsh, qrsh, q\
login and qalter)
$HOSTNAME - Name of the execution host
$TASK_ID - Array job task index number
```

6.1.3 Job Script Options

Options can be set in the job script file using this line format for each option:

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

Available options and their descriptions can be seen with the output of `qsub -help`:

Table 6.1.3: SGE Job Script Options

Option and parameter	Description
`-a date_time`	request a start time
`-ac context_list`	add context variables
`-ar ar_id`	bind job to advance reservation
`-A account_string`	account string in accounting record
`-b y[es]|n[o]`	handle command as binary
`-binding [env|pe|set] exp|lin|str`	binds job to processor cores
`-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`	delete 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
`-hold_jid_ad job_identifier_list`	define jobnet array 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

...continued

Table 6.1.3: SGE Job Script Options...continued

Option and parameter	Description
`-jsv jsv_url`	job submission verification script to be used
`-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)
`-shell y[es]|n[o]`	start command with or without wrapping <loginshell> -c
`-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
`-tc max_running_tasks`	throttle the number of concurrent tasks (experimental)
`-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|p`	verify mode (error|warning|none|just verify|poke) for jobs
`-wd working_directory`	use working_directory
`-@ file`	read commandline input from file

More detail on these options and their use is found in the man page for qsub.

6.1.4 The Executable Line

In a job script, the executable line is launched with the job launcher command after the directives lines have been dealt with, and after any other shell commands have been carried out to set up the execution environment.

Using mpirun In The Executable Line

The mpirun job-launcher command is used for executables compiled with MPI libraries. Executables that have not been compiled with MPI libraries, or which are launched without any specified number of nodes, run on a single free node chosen by the workload manager.

The executable line to run a program myprog that has been compiled with MPI libraries is run by placing the job-launcher command mpirun before it as follows:

```
mpirun myprog
```

Using cm-launcher With mpirun In The Executable Line

For SGE, for some MPI implementations, jobs sometimes leave processes behind after they have ended. A default Bright Cluster Manager installation provides a cleanup utility that removes such processes. To use it, the user simply runs the executable line using the cm-launcher wrapper before the mpirun job-launcher command:

```
cm-launcher mpirun myprog
```

The wrapper tracks processes that the workload manager launches. When it sees processes that the workload manager is unable to clean up after a job is over, it carries out the cleanup instead. Using cm-launcher is recommended if jobs that do not get cleaned up correctly are an issue for the user or administrator.

6.1.5 Job Script Examples

Some job script examples are given in this section. Each job script can use a number of variables and directives.

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 the pe environment must be assigned to the script. Depending on the interconnect, there may be a 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
```

6.2 Submitting A Job

The SGE module must be loaded first so that SGE commands can be accessed:

```
$ module add shared sge
```

With SGE a job can be submitted with qsub. The qsub command has the following syntax:

```
qsub [ options ] [ jobscript | -- [ jobscript args ]]
```

After completion (either successful or not), output is put in the user's current directory, appended with the job number which is assigned by SGE. By default, there is an error and an output file.

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

6.2.1 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 the cluster the qstat command can be used:

```
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
```

The job is then submitted, for example to the `long.q` queue:

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

6.3 Monitoring A Job

The job status can be viewed with `qstat`. In this example the `sleeper.sh` script has been submitted. Using `qstat` without options will only display a list of jobs, with no queue status options:

```
$ 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
```

More details are visible when using the `-f` (for full) option:

- 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 jobs from other users too, the following format is used:

```
$ qstat -u ''*''
```

6.4 Deleting A Job

A job can be deleted in SGE with the following command

```
$ qdel <jobid>
```

The job-id is the number assigned by SGE when the job is submitted using `qsub`. Only jobs belonging to the logged-in user can be deleted. Using `qdel` will delete a user's job regardless of whether the job is running or in the queue.

7

PBS Variants: Torque And PBS Pro

Bright Cluster Manager works with Torque and PBS Pro, which are two forks of Portable Batch System (PBS). PBS was a workload management and job scheduling system first developed to manage computing resources at NASA in the 1990s.

Torque and PBS Pro can differ significantly in the output they present when using their GUI visual tools. However because of their historical legacy, their basic design structure and job submission methods from the command line remain very similar for the user. Both Torque and PBS Pro are therefore covered in this chapter. The possible Torque schedulers (Torque's built-in scheduler, Maui, or Moab) are also covered when discussing Torque.

Torque and PBS Pro both offer a graphical interface and command line tools for submitting, monitoring, modifying and deleting jobs.

For submission and execution of jobs, both workload managers use PBS "job scripts". The user puts values into a job script for the resources being requested, such as the number of processors, memory. Other values are also set for the runtime parameters and application-specific variables.

The steps for running a job through a PBS job script are:

- Creating an application to be run via the job script
- Creating the job script, adding directives, applications, runtime parameters, and application-specific variables to the script
- Submitting the script to the workload management system

This chapter covers the using the workload managers and job scripts with the PBS variants so that users can get a basic understanding of how they are used, and can get started with typical cluster usage.

In this chapter:

- section 7.1 covers the components of a job script and job script examples
- section 7.2.1 covers submitting, monitoring, and deleting a job with a job script

More depth on using these workload managers is to be found in the PBS Professional User Guide and in the online Torque documentation at `http://www.adaptivecomputing.com/resources/docs/`.

7.1 Components Of A Job Script

To use Torque or PBS Pro, a batch job script is created by the user. The job script is a shell script containing the set of commands that the user wants to run. It also contains the resource requirement directives and other specifications for the job. After preparation, the job script is submitted to the workload manager using the qsub command. The workload manager then tries to make the job run according to the job script specifications.

A job script can be resubmitted with different parameters (e.g. different sets of data or variables).

7.1.1 Sample Script Structure

A job script in PBS Pro or Torque has a structure illustrated by the following basic example:

Example

```
#!/bin/bash
#
#PBS -l walltime=1:00:00
#PBS -l nodes=4
#PBS -l mem=500mb
#PBS -j oe

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

The first line is the standard "shebang" line used for scripts.

The lines that start with #PBS are PBS directive lines, described shortly in section 7.1.2.

The last two lines are an example of setting remaining options or configuration settings up for the script to run. In this case, a change to the directory myprogs is made, and then run the executable myprog with arguments a b c. The line that runs the program is called the executable line (section 7.1.3).

To run the executable file in the executable line in parallel, the job launcher mpirun is placed immediately before the executable file. The number of nodes the parallel job is to run on is assumed to have been specified in the PBS directives.

7.1.2 Directives

Job Script Directives And qsub Options

A job script typically has several configurable values called job script directives, set with job script directive lines. These are lines that start with a "#PBS". Any directive lines beyond the first executable line are ignored.

The lines are comments as far as the shell is concerned because they start with a "#". However, at the same time the lines are special commands when the job script is processed by the qsub command. The difference is illustrated by the following:

- The following shell comment is only a comment for a job script processed by qsub:

  ```
  # PBS
  ```

- The following shell comment is also a job script directive when processed by `qsub`:

  ```
  #PBS
  ```

Job script directive lines with the "`#PBS` " part removed are the same as options applied to the `qsub` command, so a look at the `man` pages of `qsub` describes the possible directives and how they are used. If there is both a job script directive and a `qsub` command option set for the same item, the `qsub` option takes precedence.

Since the job script file is a shell script, the shell interpreter used can be changed to another shell interpreter by modifying the first line (the "`#!`" line) to the preferred shell. Any shell specified by the first line can also be overridden by using the "`#PBS -S`" directive to set the shell path.

Walltime Directive

The workload manager typically has default walltime limits per queue with a value limit set by the administrator. The user sets walltime limit by setting the "`#PBS -l walltime`" directive to a specific time. The time specified is the maximum time that the user expects the job should run for, and it allows the workload manager to work out an optimum time to run the job. The job can then run sooner than it would by default.

If the walltime limit is exceeded by a job, then the job is stopped, and an error message like the following is displayed:

```
=» PBS: job killed:  walltime <runningtime> exceeded limit <settime>
```

Here, *<runningtime>* indicates the time for which the job actually went on to run, while *<settime>* indicates the time that the user set as the walltime resource limit.

Resource List Directives

Resource list directives specify arguments to the `-l` directive of the job script, and allow users to specify values to use instead of the system defaults.

For example, in the sample script structure earlier, a job walltime of one hour and a memory space of at least 500MB are requested (the script requires the size of the space be spelled in lower case, so "`500mb`" is used).

If a requested resource list value exceeds what is available, the job is queued until resources become available.

For example, if nodes only have 2000MB to spare and 4000MB is requested, then the job is queued indefinitely, and it is up to the user to fix the problem.

Resource list directives also allow, for example, the number of nodes (`-l nodes`) and the virtual processor cores per nodes (`-l ppn`) to be specified. If no value is specified, the default is 1 core per node.

If 8 cores are wanted, and it does not matter how the cores are allocated (e.g. 8 per node or 1 on 8 nodes) the directive used in Torque is:

```
#PBS -l nodes=8
```

For PBS Pro v11 this also works, but is deprecated, and the form "`#PBS -l select=8`" is recommended instead.

Further examples of node resource specification are given in a table on page 41.

Job Directives: Job Name, Logs, And IDs

If the name of the job script file is `jobname`, then by default the output and error streams are logged to `jobname.o<number>` and `jobname.e<number>` respectively, where <*number*> indicates the associated job number. The default paths for the logs can be changed by using the `-o` and `-e` directives respectively, while the base name (`jobname` here) can be changed using the `-N` directive.

Often, a user may simply merge both logs together into one of the two streams using the -j directive. Thus, in the preceding example, "`-j oe`" merges the logs to the output log path, while "`-j eo`" would merge it to error log path.

The job ID is an identifier based on the job number and the FQDN of the login node. For a login node called `bright52.cm.cluster`, the job ID for a job number with the associated value <*number*> from earlier, would by default be <*number*>.bright52.cm.cluster, but it can also simply be abbreviated to <*number*>.

Job Queues

Sending a job to a particular job queue is sometimes appropriate. An administrator may have set queues up so that some queues are for very long term jobs, or some queues are for users that require GPUs. Submitting a job to a particular queue <*destination*> is done by using the directive "`#PBS -q <destination>`".

Directives Summary

A summary of the job directives covered, with a few extras, are shown in the following table:

Directive Description	Specified As
Name the job <*jobname*>	`#PBS -N <jobname>`
Run the job for a time of <*walltime*>	`#PBS -l <walltime>`
Run the job at <*time*>	`#PBS -a <time>`
Set error log name to <*jobname.err*>	`#PBS -e <jobname.err>`
Set output log name to <*jobname.log*>	`#PBS -o <jobname.log>`
Join error messages to output log	`#PBS -j eo`
Join output messages to error log	`#PBS -j oe`
Mail to <*user@address*>	`#PBS -M <user@address>`
Mail on <*event*>	`#PBS -m <event>`
where <*event*> takes the	`(a)bort`
value of the letter in	`(b)egin`
the parentheses	`(e)nd`
	`(n) do not send email`
Queue is <*destination*>	`#PBS -q <destination>`
Login shell path is <*shellpath*>	`#PBS -S <shellpath>`

Resource List Directives Examples

Examples of how requests for resource list directives work are shown in the following table:

Resource Example Description	"#PBS -l" Specification
Request 500MB memory	`mem=500mb`
Set a maximum runtime of 3 hours 10 minutes and 30 seconds	`walltime=03:10:30`
8 nodes, anywhere on the cluster	`nodes=8*`
8 nodes, anywhere on the cluster	`select=8**`
2 nodes, 1 processor per node	`nodes=2:ppn=1`
3 nodes, 8 processors per node	`nodes=3:ppn=8`
5 nodes, 2 processors per node and 1 GPU per node	`nodes=5:ppn=2:gpus=1*`
5 nodes, 2 processors per node, and 1 GPU per node	`select=5:ncpus=2:ngpus=1**`
5 nodes, 2 processors per node, 3 virtual processors for MPI code	`select=5:ncpus=2:mpiprocs=3**`
5 nodes, 2 processors per node, using any GPU on the nodes	`select=5:ncpus=2:ngpus=1**`
5 nodes, 2 processors per node, using a GPU with ID 0 from nodes	`select=5:ncpus=2:gpu_id=0**`

*For Torque 2.5.5
**For PBS Pro 11

Some of the examples illustrate requests for GPU resource usage. GPUs and the CUDA utilities for Nvidia are introduced in Chapter 8. In the Torque and PBS Pro workload managers, GPU usage is treated like the attributes of a resource which the cluster administrator will have pre-configured according to local requirements.

For further details on resource list directives, the Torque and PBS Pro user documentation should be consulted.

7.1.3 The Executable Line

In the job script structure (section 7.1.1), the executable line is launched with the job launcher command after the directives lines have been dealt with, and after any other shell commands have been carried out to set up the execution environment.

Using `mpirun` In The Executable Line

The `mpirun` command is used for executables compiled with MPI libraries. Executables that have not been compiled with MPI libraries, or which are launched without any specified number of nodes, run on a single free node chosen by the workload manager.

The executable line to run a program `myprog` that has been compiled with MPI libraries is run by placing the job-launcher command `mpirun` before it as follows:

```
mpirun myprog
```

Using cm-launcher With mpirun In The Executable Line

For Torque, for some MPI implementations, jobs sometimes leave processes behind after they have ended. A default Bright Cluster Manager installation provides a cleanup utility that removes such processes. To use it, the user simply runs the executable line using the cm-launcher wrapper before the mpirun job-launcher command:

```
cm-launcher mpirun myprog
```

The wrapper tracks processes that the workload manager launches. When it sees processes that the workload manager is unable to clean up after the job is over, it carries out the cleanup instead. Using cm-launcher is recommended if jobs that do not get cleaned up correctly are an issue for the user or administrator.

7.1.4 Example Batch Submission Scripts

Node Availability

The following job script tests which out of 4 nodes requested with "-l nodes" are made available to the job in the workload manager:

Example

```
#!/bin/bash
#PBS -l walltime=1:00
#PBS -l nodes=4
echo -n "I am on: "
hostname;

echo finding ssh-accessible nodes:
for node in $(cat ${PBS_NODEFILE}) ; do
    echo -n "running on: "
    /usr/bin/ssh $node hostname
done
```

The directive specifying walltime means the script runs at most for 1 minute. The ${PBS_NODEFILE} array used by the script is created and appended with hosts by the queueing system. The script illustrates how the workload manager generates a ${PBS_NODEFILE} array based on the requested number of nodes, and which can be used in a job script to spawn child processes. When the script is submitted, the output from the log will look like:

```
I am on: node001
finding ssh-accessible nodes:
running on: node001
running on: node002
running on: node003
running on: node004
```

This illustrates that the job starts up on a node, and that no more than the number of nodes that were asked for in the resource specification are provided.

The list of all nodes for a cluster can be found using the pbsnodes command (section 7.2.6).

Using InfiniBand

A sample PBS script for InfiniBand is:

```
#!/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 by the script, then source modules environment:
. /etc/profile.d/modules.sh

# Add any modules you might require:
module add shared mvapich/gcc torque maui pbspro

#! 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 job runs on the following machines:
echo $(cat $PBS_NODEFILE | uniq)

$mpirun_command="mpirun $application $options"

#! Run the parallel MPI executable (nodes*ppn)
echo Running $mpirun_command
eval $mpirun_command
```

In the preceding script, no machine file is needed, since it is automatically built by the workload manager and passed on to the mpirun parallel job launcher utility. The job is given a unique ID and run in parallel on the nodes based on the resource specification.

7.1.5 Links To Other Resources About Job Scripts In Torque And PBS Pro

A number of useful links are:

- Torque examples:
  http://bmi.cchmc.org/resources/software/torque/examples
- PBS Pro script files:
  http://www.ccs.tulane.edu/computing/pbs/pbs.phtml
- Running PBS Pro jobs and directives:
  http://wiki.hpc.ufl.edu/index.php/Job_Submission_Queues

7.2 Submitting A Job

7.2.1 Preliminaries: Loading The Modules Environment

To submit a job to the workload management system, the user must ensure that the following environment modules are loaded:

- If using Torque with no external scheduler:

  ```
  $ module add shared torque
  ```

- If using Torque with Maui:

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

- If using Torque with Moab:

  ```
  $ module add shared torque moab
  ```

- If using PBS Pro:

  ```
  $ module add shared pbspro
  ```

Users can pre-load particular environment modules as their default using the "module init*" commands (section 2.3.3).

7.2.2 Using qsub

The command qsub is used to submit jobs to the workload manager system. The command returns a unique job identifier, which is used to query and control the job and to identify output. The usage format of qsub and some useful options are listed here:

```
USAGE: qsub [<options>] <job script>

Option  Hint   Description
------  ----   -----------
```

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

For example, a job script called `mpirun.job` with all the relevant directives set inside the script, may be submitted as follows:

Example

```
$ qsub mpirun.job
```

A job may be submitted to a specific queue `testq` as follows:

Example

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

The `man` page for `qsub` describes these and other options. The options correspond to PBS directives in job scripts (section 7.1.1). If a particular item is specified by a `qsub` option as well as by a PBS directive, then the `qsub` option takes precedence.

7.2.3 Job Output

By default, the output from the job script <*scriptname*> goes into the home directory of the user for Torque, or into the current working directory for PBS Pro.

By default, error output is written to <*scriptname*>`.e`<*jobid*> and the application output is written to <*scriptname*>`.o`<*jobid*>, where <*jobid*> is a unique number that the workload manager allocates. Specific output and error files can be set using the `-o` and `-e` options respectively. The error and output files can usefully be concatenated into one file with the `-j oe` or `-j eo` options. More details on this can be found in the `qsub` man page.

7.2.4 Monitoring A Job

To use the commands in this section, the appropriate workload manager module must be loaded. For example, for Torque, `torque` module needs to be loaded:

```
$ module add torque
```

`qstat` Basics

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

In PBS/Torque, the command "`qstat -an`" shows what jobs are currently submitted or running on the queuing system. An example output is:

```
[fred@bright52 ~]$ qstat -an
bright52.cm.cluster:
                                                   Req'd  Req'd   Elap
```

```
Job ID      Username Queue  Jobname SessID NDS TSK Memory Time  S Time
----------- -------- ------ ------- ------ --- --- ------ ----- - -----
78.bright52 fred     shortq tjob     10476   1   1  555mb 00:01 R 00:00
79.bright52 fred     shortq tjob       --    1   1  555mb 00:01 Q   --
```

The output shows the Job ID, the user who owns the job, the queue, the job name, the session ID for a running job, the number of nodes requested, the number of CPUs or tasks requested, the time requested (`-l walltime`), the job state (S) and the elapsed time. In this example, one job is seen to be running (R), and one is still queued (Q). The `-n` parameter causes nodes that are in use by a running job to display at the end of that line.

Possible job states are:

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

The command "`qstat -q`" shows what queues are available. In the following 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
```

`showq` **From Maui**

If the Maui scheduler is running, and the Maui module loaded (`module add maui`), then Maui's `showq` command displays a similar output. In this example, one dual-core node is available (1 node, 2 processors), one job is running and 3 are queued (in the `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
```

Viewing Job Details With `qstat` And `checkjob`

Job Details With `qstat` With `qstat -f` the full output of the job is displayed. The output shows 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
```

Job Details With `checkjob` The `checkjob` command (only for Maui) is particularly useful for checking why a job has not yet executed. For a job that has an excessive memory requirement, the output looks something like:

```
[fred@bright52 ~]$ checkjob 65

checking job 65

State: Idle
Creds:  user:fred  group:fred  class:shortq  qos:DEFAULT
WallTime: 00:00:00 of 00:01:00
SubmitTime: Tue Sep 13 15:22:44
  (Time Queued  Total: 2:53:41  Eligible: 2:53:41)

Total Tasks: 1

Req[0]  TaskCount: 1  Partition: ALL
Network: [NONE]  Memory >= 0  Disk >= 0  Swap >= 0
Opsys: [NONE]  Arch: [NONE]  Features: [NONE]
Dedicated Resources Per Task: PROCS: 1  MEM: 495M

IWD: [NONE]  Executable:  [NONE]
Bypass: 0  StartCount: 0
PartitionMask: [ALL]
Flags:       RESTARTABLE

PE:  1.01  StartPriority:  173
job cannot run in partition DEFAULT (idle procs do not meet requirement\
s : 0 of 1 procs found)
idle procs:   3  feasible procs:   0

Rejection Reasons: [CPU          :    3]
```

The `-v` option gives slightly more detail.

7.2.5 Deleting A Job

An already submitted job can be deleted using the `qdel` command:

```
$ qdel <jobid>
```

The job ID is printed to the terminal when the job is submitted. To get the job ID of a job if it has been forgotten, the following can be used:

```
$ qstat
```

or

```
$ showq
```

7.2.6 Monitoring Nodes In Torque And PBS Pro

The nodes that the workload manager knows about can be viewed using the `pbsnodes` command.

The following output is from a cluster made up of 2-core nodes, as indicated by the value of 2 for `ncpu` for Torque and PBS Pro. If the node is available to run scripts, then its state is `free` or `time-shared`. When a node is used exclusively (section 8.5.2) by one script, the state is `job-exclusive`.

For Torque the display resembles (some output elided):

```
[fred@bright52 ~]$ pbsnodes -a
node001.cm.cluster
     state = free
     np = 3
     ntype = cluster
     status = rectime=1317911358,varattr=,jobs=96...ncpus=2...
     gpus = 1

node002.cm.cluster
     state = free
     np = 3
     ...
     gpus = 1

     ...
```

For PBS Pro the display resembles (some output elided):

```
[fred@bright52 ~]$ pbsnodes -a
node001.cm.cluster
     Mom = node001.cm.cluster
     ntype = PBS
     state = free
     pcpus = 3
     resources_available.arch = linux
     resources_available.host = node001
     ...
     sharing = default_shared

node002.cm.cluster
     Mom = node002.cm.cluster
     ntype = PBS
     state = free
     ...

     ...
```

8

Using GPUs

GPUs (Graphics Processing Units) are chips that provide specialized parallel processing power. Originally, GPUs were designed to handle graphics processing as part of the video processor, but their ability to handle non-graphics tasks in a similar manner has become important for general computing. GPUs designed for general purpose computing task are commonly called General Purpose GPUs, or GPGPUs.

A GPU is suited for processing an algorithm that naturally breaks down into a process requiring many similar calculations running in parallel.

Physically, one GPU is typically a built-in part of the motherboard of a node or a board in a node, and consists of hundreds of processing cores. There are also dedicated standalone units, commonly called GPU Units, consisting of several GPUs in one chassis, and which are typically assigned to particular nodes via PCI-Express connections.

Bright Cluster Manager contains several tools which can be used to set up and program GPUs for general purpose computations.

8.1 Packages

A number of different GPU-related packages are included in Bright Cluster Manager. For CUDA 4.0 these are:

- `cuda40-driver`: Provides the GPU driver
- `cuda40-libs`: Provides the libraries that come with the driver (libcuda etc)
- `cuda40-toolkit`: Provides the compilers, cuda-gdb, and math libraries
- `cuda40-tools`: Provides the CUDA tools SDK
- `cuda40-profiler`: Provides the CUDA visual profiler
- `cuda40-sdk`: Provides additional tools, development files and source examples

CUDA versions 4.1, 4.2, and 5.0 are also provided by Bright Cluster Manager. The exact implementation depends on how the system administrator has configured CUDA.

8.2 Using CUDA

After installation of the packages, for general usage and compilation it is sufficient to load just the CUDA4/toolkit module.

```
module add cuda40/toolkit
```

© Bright Computing, Inc.

Also available are several other modules related to CUDA:

- `cuda40/blas/4.0.17`: Provides paths and settings for the CUBLAS library.
- `cuda40/fft`: Provides paths and settings for the CUFFT library.

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

Extensive documentation on how to get started, the various tools, and how to use the CUDA suite is in the `$CUDA_INSTALL_PATH/doc` directory.

8.3 Using OpenCL

OpenCL functionality is provided with the environment module `cuda40/toolkit`.

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

8.4 Compiling Code

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

- `host`: with one or more CPUs
- `device`: with one or more CUDA enabled GPUs

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

The `nvcc` command by default compiles code and links the objects for both the host system and the GPU. The `nvcc` command distinguishes between the two and it 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 code to an executable is:

```
nvcc testcode.cu -o testcode
```

The most used options are:

- `-g` or `-debug` *<level>*: This generates debug-able code for the host
- `-G` or `-device-debug` *<level>*: This generates debug-able code for the GPU
- `-o` or `-output-file` *<file>*: This creates an executable with the name *<file>*
- `-arch=sm_13`: This can be enabled if the CUDA device supports compute capability 1.3, which includes double-precision

If double-precision floating-point is not supported or the flag is not set, warnings such as the following will come up:

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

The `nvcc` documentation manual, *"The CUDA Compiler Driver NVCC"* has more information on compiler options.

The CUDA SDK has more programming examples and information accessible from the file `$CUDA_SDK/C/Samples.html`.

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

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

8.5 Available Tools

8.5.1 CUDA gdb

The CUDA debugger can be started using: `cuda-gdb`. Details of how to use it are available in the *"CUDA-GDB (NVIDIA CUDA Debugger)"* manual, in the `doc` directory. It is based on GDB, the GNU Project debugger, and requires the use of the "`-g`" or "`-G`" options compiling.

Example

```
nvcc -g -G testcode.cu -o testcode
```

8.5.2 nvidia-smi

The NVIDIA System Management Interface command, `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 for making a GPU exclusive:

- List GPUs
- Select a GPU
- Lock GPU to a compute mode
- 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 that set the exclusivity lock on the the GPU!

To list the GPUs, the `-L` argument can be used:

```
$ 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 on the GPU:

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

The ruleset may be one of the following:

- 0 - Default mode (multiple applications allowed on the GPU)
- 1 - Exclusive thread mode (only one compute context is allowed to run on the GPU, usable from one thread at a time)
- 2 - Prohibited mode (no compute contexts are allowed to run on the GPU)
- 3 - Exclusive process mode (only one compute context is allowed to run on the GPU, usable from multiple threads at a time)

To check the state of the GPU:

```
$ nvidia-smi -i 0 -q
COMPUTE mode rules for GPU 0: 1
```

In this example, GPU0 is locked, and there is a running application using GPU0. A second application attempting 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, the GPU can be unlocked to allow multiple users:

```
nvidia-smi -i 0 -c 0
```

8.5.3 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, and the toolkit libraries have already been pre-configured accordingly. However, they need to be compiled prior to use. Depending on the cluster, this might have already have been done.

```
[fred@demo ~] cd
[fred@demo ~] cp -r $CUDA_SDK
[fred@demo ~] cd $(basename $CUDA_SDK); cd C
[fred@demo C] make
[fred@demo C] cd $(basename $CUDA_SDK); cd OpenCL
[fred@demo OpenCL] make
```

CUTIL provides functions for:

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

8.5.4 CUDA "Hello world" Example

A hello world example code using CUDA is:

Example

```
/*
  CUDA example
  "Hello World" using shift13, a rot13-like function.
  Encoded on CPU, decoded on GPU.

  rot13 cycles between 26 normal alphabet characters.

  shift13 shifts 13 steps along the normal alphabet characters
  So it translates half the alphabet into non-alphabet characters

  shift13 is used because it is simpler than rot13 in c
  so we can focus on the point
  (c) Bright Computing
  Taras Shapovalov <taras.shapovalov@brightcomputing.com>
 */
#include <cuda.h>
#include <cutil_inline.h>
#include <stdio.h>

// CUDA kernel definition: undo shift13
__global__ void helloWorld(char* str) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  str[idx] -= 13;
}

int main(int argc, char** argv )
{
  char s[] = "Hello World!";
  printf("String for encode/decode: %s\n", s);

  // CPU shift13
  int len = sizeof(s);
  for (int i = 0; i < len; i++) {
    s[i] += 13;
  }
  printf("String encoded on CPU as: %s\n", s);

  // Allocate memory on the CUDA device
  char *cuda_s;
  cudaMalloc((void**)&cuda_s, len);

  // Copy the string to the CUDA device
  cudaMemcpy(cuda_s, s, len, cudaMemcpyHostToDevice);

  // Set the grid and block sizes  (dim3 is a type)
  // and "Hello World!" is 12 characters, say 3x4
  dim3 dimGrid(3);
  dim3 dimBlock(4);

  // Invoke the kernel to undo shift13 in GPU
  helloWorld<<< dimGrid, dimBlock >>>(cuda_s);

  // Retrieve the results from the CUDA device
  cudaMemcpy(s, cuda_s, len, cudaMemcpyDeviceToHost);

  // Free up the allocated memory on the CUDA device
  cudaFree(cuda_s);

  printf("String decoded on GPU as: %s\n", s);
  return 0;
}
```

The preceding code example may be compiled and run with:

```
[fred@bright52 ~]$ nvcc hello.cu -o hello
[fred@bright52 ~]$ module add shared openmpi/gcc/64/1.4.4 slurm
```

```
[fred@bright52 ~]$ salloc -n1 --gres=gpu:1 mpirun hello
salloc: Granted job allocation 2263
String for encode/decode: Hello World!
String encoded on CPU as: Uryy|-d|yq.
String decoded on GPU as: Hello World!
alloc: Relinquishing job allocation 2263
salloc: Job allocation 2263 has been revoked.
```

The number of characters displayed in the encoded string appear less than expected because there are unprintable characters in the encoding due to the cipher used being not exactly rot13.

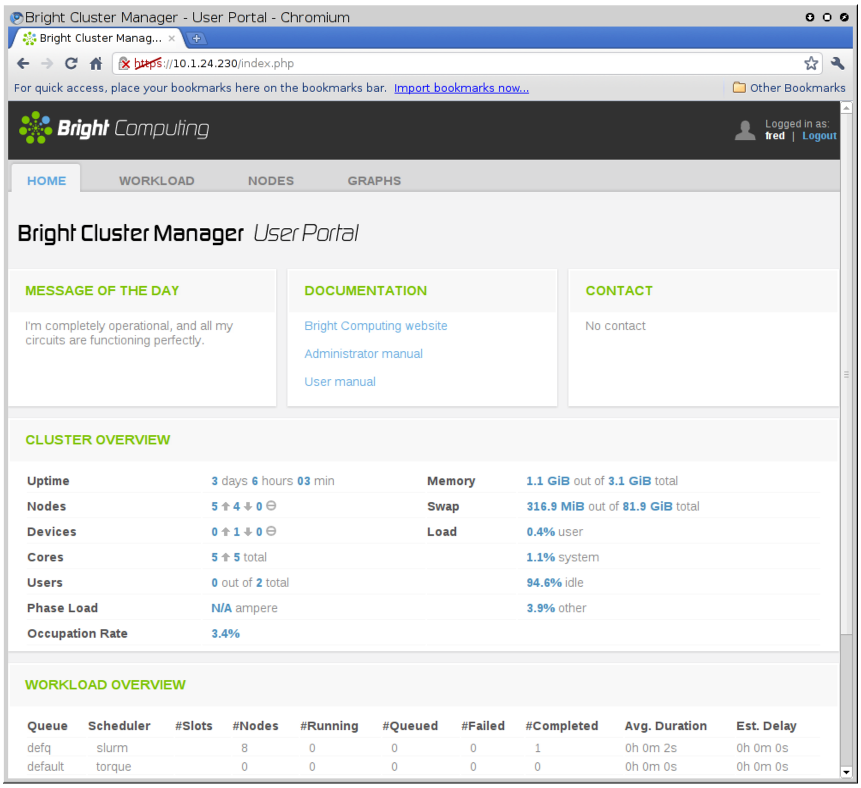
9 User Portal

The user portal allows users to login via a browser and view the state of the cluster themselves. It is a read-only interface.

The first time a browser is used to login to the cluster portal, a warning about the site certificate being untrusted appears in a default Bright Cluster configuration. This can safely be accepted.

9.0.1 Home Page

The default home page allows a quick glance to convey the most important cluster-related information for users (figure 9.1):


Bright Cluster Manager - User Portal - Chromium
Bright Cluster Manag...
https://10.1.24.230/index.php
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...
Other Bookmarks
Bright Computing
Logged in as:
fred | Logout
HOME
WORKLOAD
NODES
GRAPHS

Bright Cluster Manager *User Portal*

MESSAGE OF THE DAY

I'm completely operational, and all my circuits are functioning perfectly.

DOCUMENTATION

Bright Computing website

Administrator manual

User manual

CONTACT

No contact

CLUSTER OVERVIEW

Uptime	3 days 6 hours 03 min	Memory	1.1 GiB out of 3.1 GiB total
Nodes	5 ↑ 4 ↓ 0 ⊖	Swap	316.9 MiB out of 81.9 GiB total
Devices	0 ↑ 1 ↓ 0 ⊖	Load	0.4% user
Cores	5 ↑ 5 total		1.1% system
Users	0 out of 2 total		94.6% idle
Phase Load	N/A ampere		3.9% other
Occupation Rate	3.4%		

WORKLOAD OVERVIEW

Queue	Scheduler	#Slots	#Nodes	#Running	#Queued	#Failed	#Completed	Avg. Duration	Est. Delay
defq	slurm		8	0	0	0	1	0h 0m 2s	0h 0m 0s
default	torque		0	0	0	0	0	0h 0m 0s	0h 0m 0s

Figure 9.1: User Portal: Default Home Page

The following items are displayed on a default home page:

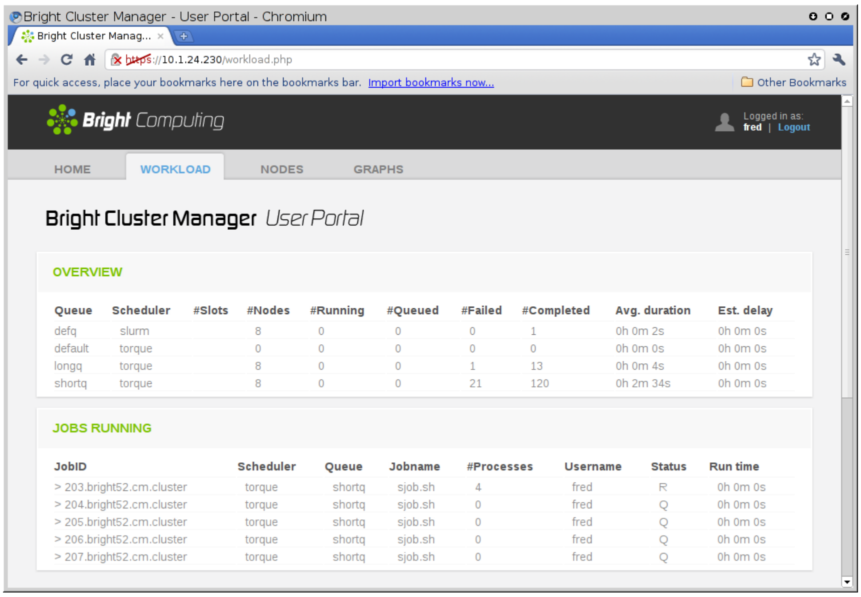
- a Message Of The Day. The administrator may put up important messages for users here

© Bright Computing, Inc.

- links to the documentation for the cluster
- contact information. This typically shows how to contact technical support
- an overview of the cluster state, displaying some cluster parameters
- a workload overview. This is a table displaying a summary of queues and their associated jobs

9.0.2 The WORKLOAD Tab

The page opened by clicking on the WORKLOAD tab allows a user to see workload-related information for the cluster (figure 9.2).


Bright Cluster Manager - User Portal - Chromium
Bright Cluster Manag...
https://10.1.24.230/workload.php
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...
Other Bookmarks
Bright Computing
Logged in as:
fred | Logout
HOME
WORKLOAD
NODES
GRAPHS

Bright Cluster Manager *User Portal*

OVERVIEW

Queue	Scheduler	#Slots	#Nodes	#Running	#Queued	#Failed	#Completed	Avg. duration	Est. delay
defq	slurm		8	0	0	0	1	0h 0m 2s	0h 0m 0s
default	torque		0	0	0	0	0	0h 0m 0s	0h 0m 0s
longq	torque		8	0	0	1	13	0h 0m 4s	0h 0m 0s
shortq	torque		8	0	0	21	120	0h 2m 34s	0h 0m 0s

JOBS RUNNING

JobID	Scheduler	Queue	Jobname	#Processes	Username	Status	Run time
> 203.bright52.cm.cluster	torque	shortq	sjob.sh	4	fred	R	0h 0m 0s
> 204.bright52.cm.cluster	torque	shortq	sjob.sh	0	fred	Q	0h 0m 0s
> 205.bright52.cm.cluster	torque	shortq	sjob.sh	0	fred	Q	0h 0m 0s
> 206.bright52.cm.cluster	torque	shortq	sjob.sh	0	fred	Q	0h 0m 0s
> 207.bright52.cm.cluster	torque	shortq	sjob.sh	0	fred	Q	0h 0m 0s

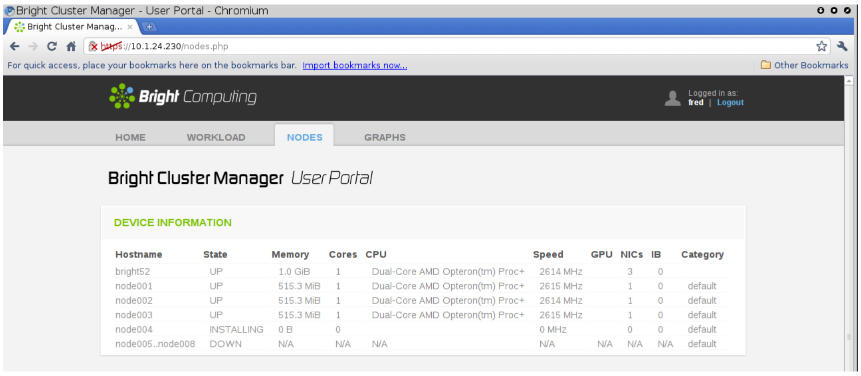
Figure 9.2: User Portal: Workload Page

The following two tables are displayed:

- A workload overview table (the same as the table in the home page).
- A table displaying the current jobs running on the cluster

9.0.3 The NODES Tab

The page opened by clicking on the NODES tab shows a list of nodes on the cluster (figure 9.3).


Bright Cluster Manager - User Portal - Chromium
Bright Cluster Manag...
https://10.1.24.230/nodes.php
For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...
Other Bookmarks
Bright Computing
Logged in as:
fred | Logout
HOME
WORKLOAD
NODES
GRAPHS

Bright Cluster Manager *User Portal*

DEVICE INFORMATION

Hostname	State	Memory	Cores	CPU	Speed	GPU	NICs	IB	Category
bright52	UP	1.0 GiB	1	Dual-Core AMD Opteron(tm) Proc+	2614 MHz		3	0	
node001	UP	515.3 MiB	1	Dual-Core AMD Opteron(tm) Proc+	2615 MHz		1	0	default
node002	UP	515.3 MiB	1	Dual-Core AMD Opteron(tm) Proc+	2614 MHz		1	0	default
node003	UP	515.3 MiB	1	Dual-Core AMD Opteron(tm) Proc+	2615 MHz		1	0	default
node004	INSTALLING	0 B	0		0 MHz		0	0	default
node005..node008	DOWN	N/A	N/A	N/A	N/A	N/A	N/A	N/A	default

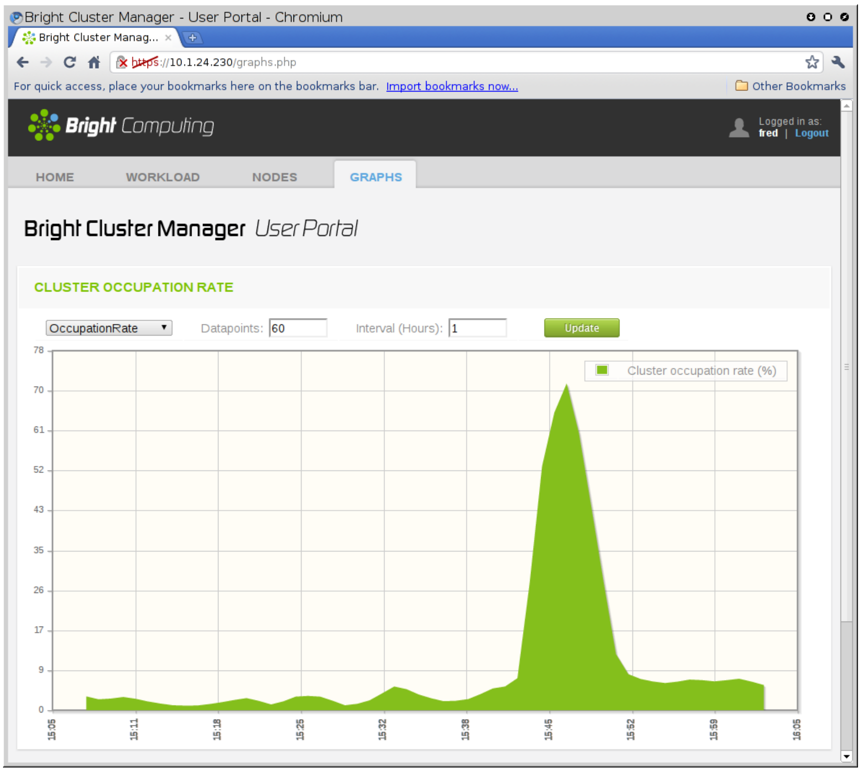
Figure 9.3: User Portal: Nodes Page

The following information about the head or regular nodes is presented:

- `Hostname`: the node name
- `State`: For example, `UP`, `DOWN`, `INSTALLING`
- `Memory`: RAM on the node
- `Cores`: Number of cores on the node
- `CPU`: Type of CPU, for example, Dual-Core AMD Opteron™
- `Speed`: Processor speed
- `GPU`: Number of GPUs on the node, if any
- `NICs`: Number of network interface cards on the node, if any
- `IB`: Number of InfiniBand interconnects on the node, if any
- Category: The node category that the node has been allocated by the administrator (by default it is `default`)

9.0.4 The GRAPHS Tab

By default the GRAPHS tab displays the cluster occupation rate for the last hour (figure 9.4).


Bright Cluster Manager - User Portal - Chromium

https://10.1.24.230/graphs.php

For quick access, place your bookmarks here on the bookmarks bar. Import bookmarks now...

Logged in as: fred | Logout

Bright Cluster Manager User Portal

CLUSTER OCCUPATION RATE

OccupationRate Datapoints: 60 Interval (Hours): 1 Update

Figure 9.4: User Portal: Graphs Page

Selecting other values is possible for

- Workload Management Metrics. The following workload manager metrics can be viewed:
  - RunningJobs

  - QueuedJobs
  - FailedJobs
  - CompletedJobs
  - EstimatedDelay
  - AvgJobDuration
  - AvgExpFactor

- Cluster Management Metrics. The following metrics can be viewed
  - OccupationRate
  - NetworkBytesRecv
  - NetworkBytesSent
  - DevicesUp
  - NodesUp
  - TotalMemoryUsed
  - TotalSwapUsed
  - PhaseLoad
  - CPUCoresAvailable
  - GPUAvailable
  - TotalCPUUser
  - TotalCPUSystem
  - TotalCPUIdle
- Datapoints: The number of points used for the graph can be specified. The points are interpolated if necessary
- Interval (Hours): The period over which the data points are displayed

The `Update` button must be clicked to display any changes made.

A

MPI Examples

A.1 “Hello world”

A quick application to test the MPI libraries 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 head/compute node 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) {
                head();
        } else {
                computenode();
        }
        MPI_Finalize();       /* cleanup MPI */
}

head()
{
        int     ntasks, rank, work;
        double       result;
        MPI_Status     status;
        MPI_Comm_size(
        MPI_COMM_WORLD,   /* always use this */
        &ntasks);                  /* #processes in application */
/*
* Seed the compute nodes.
*/
        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 compute node 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 compute nodes to exit.
*/
        for (rank = 1; rank < ntasks; ++rank) {
                MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
        }
}

computenode()
{
        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 initialize and clean up an MPI process:

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

A.4 What Is The Current Process? How Many Processes Are There?

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, almost any application can be programmed. There are many other, more exotic functions in MPI, but all can be built upon those presented here so far.