

# Introduction

## Purpose & Policies

The RCS general purpose cluster is a high-performance computing environment built to assist researchers in their work. This cluster can be used by any researcher, lab, or other project in furtherance of their research.

The cluster is made up of a variety of resources, including large Sparc based mini-frames and smaller Intel and AMD based "blades" running Linux. These nodes have a common set of software, tools, and applications that can handle a variety of research tasks.

The cluster and its associated tools are primarily used within a Unix shell environment. Thus, familiarity with Unix concepts, navigating the shell and commandline, as well as shell scripting is also necessary.

If you have any questions, contact Research Computing Support.

## Conceptual Overview

### What's a Cluster, Anyway

While popular usage typically defines a cluster as a bunch of tightly integrated, inexpensive computer nodes, the term can also refer to any aggregation of computing resources under an administrative umbrella. The RCS general purpose cluster is such an aggregation- we have both loosely integrated systems (e.g. bedrock and gazoo) and the more traditional Linux cluster running on smaller computers.

While many typically think that a cluster is only useful for parallel computation, the fact is that this cluster is valuable for many different compute tasks. The cluster can be used for highly-parallel tasks- jobs that use many compute nodes, communicating and sharing data amongst all the nodes. It is also useful for embarrassingly-parallel tasks- where there are tens, hundreds, even thousands of independent tasks, tasks that do not communicate with each other. This cluster can also be used to offload single tasks to compute nodes- possibly running on nodes with more power than those available to you, or freeing up your resources for other tasks. The cluster is also available for interactive work, giving you a shell on a system for compiling or testing applications, methods, and tools.

While the cluster is capable of running jobs for very long periods of time, it is not appropriate for tasks such as web servers, databases, etc.

Another commonly held misconception is that by virtue of running on a cluster, applications will automatically run across multiple nodes. This is simply not true. Some clusters will engage in process migration, moving processes between different nodes- however, this is different from a process which spawns many tasks that are capable of communicating information between them. Writing an application to run across multiple CPUs \*or\* nodes requires special techniques and libraries such as pthreads, MPI, or PVM.

## Holding it Together

As indicated, a cluster is a loose aggregation of compute systems. The "glue" holding this cluster together is something termed a "resource manager". This resource manager is made up of two pieces of software-Torque and Moab. Each provides complementary services:

Torque

handles queuing, dispatch, execution, and post-processing of jobs onto cluster nodes

Moab

the intelligence behind Torque- handles scheduling and prioritization and enforces limits and other policies.

## How it Works

All jobs that are to run on the cluster must pass through the Resource Manager. When your job is ready:

1. You give the job to Torque via the `qsub` command
2. Torque puts the job in a queue where it waits
3. Periodically Moab checks the queue:
  1. All jobs in the queue are evaluated according to the policies within Moab and the attributes of the job (such as who has submitted it, how much time, CPU, or memory is required, etc.) and given a *priority* score
  2. Moab attempts to fit the highest-scored jobs onto available resources. When a match between a job and available resources is found, Moab tells Torque which job to run and where to run it.
4. When Torque is given this command, Torque starts the job on the first node in the list of nodes assigned to the job- the "manager of managers," or MOM, which:
  1. copies in any files requested (stage-in)
  2. sets up the execution environment (running the "prologue")
  3. executes the job
5. When the job exits, the MOM "cleans up":
  1. copying out any files requested (stage-out)
  2. clearing out the node (running the "epilogue")
  3. copying out any error/output messages
  4. indicating the exit status of the job to the Torque server

## Accessing the Cluster

The cluster is accessed via a shell session with the host hoppy, on which the resource manager runs. To log into hoppy, you will need:

- an RCS account (or "fred account" in common parlance)- mail [rcs@fhcrc.org](mailto:rcs@fhcrc.org) to request an account if you don't already have one.
- a Secure Shell client- most Unix distributions (including Mac OS X) include one. Windows users can download PuTTY.

Once you've got those items, you can open a shell to the host "hoppy". Log in using your RCS account username and password.

## Setting Up Your Account

The "stock" environment that you get when logging into hoppy should work just fine. If you wish to use the Moab commands (`showq`, `checkjob`, et. al.), add `/opt/moab/bin` to your path.

Other changes to your login files may be required if you are using a parallel environment, especially PVM.

## Submitting Jobs

### Overview

A "job" is not too much more than a shell script containing all the shell commands you would use to accomplish your task. When given to the resource manager, it is run by a shell interpreter (e.g. bash, ksh, tcsh).

`qsub` is the primary way in which jobs are submitted to the cluster. This command takes a number of arguments, too many to go into detail here. However, the most common arguments you will use in submission of your jobs is:

- I  
Starts an interactive session- when run, you will have a shell on a cluster node.
- l <resource request>  
This argument allows you to specify the "qualities" of the job you are running. Time, number of CPUs, number of nodes, memory, and disk are all examples of qualities you might request.
- S <shell>  
Specifies the shell that the node should use to interpret your job script. Defaults to your login shell

For example:

```
ps1> qsub -l nodes=2 -S /bin/bash job.script
```

### Basic Job Primer

You will want to first download the "examples" tar file and expand that in your home directory.

This can be retrieved here: <http://hoppy/downloads/examples/gptutorial.tar>

The format of a job script is pretty straight-forward. Again, the `qsub(1B)` man page has many more details, but briefly, the job script is a shell script (C or Bourne syntax according to your preference) that will be run by Torque on a node. A job script may also contain commands for the `qsub` command. If you want to include these commands, add lines (at the beginning of the script) that begin with "#PBS" and a `qsub` argument, vis:

```
#PBS -S /bin/tcsh
#PBS -l nodes=1
...
```

Almost any `qsub` argument is valid- though, for obvious reasons, the "-I" argument cannot be used.

Armed with this knowledge, let's write our first job script...

## A First Job

```
ps1> cd ~/examples/hello.1
ps1> cat hello.qsub
# Request one node and 10 minutes of run-time
#PBS -l nodes=1,walltime=00:10:00
#
# Use the bourne-again shell
#PBS -S /bin/bash
#
# print "hello, world" on stdout
#
GREETING="hello, world"
echo $GREETING
exit 0
```

Now we can submit this script to the resource manager:

```
ps1> qsub hello.qsub
614.hoppy
```

Torque returns the *job id* assigned to our script, which we use to query the queue with the `qstat` command:

```
ps1> qstat 614
Job id          Name          User          Time Use S Queue
-----
614.hoppy      hello.qsub    mrg           0 Q public
```

Which shows the job queued and ready to run. After a bit of time, the job will run- disappearing from the queue. Looking in this directory now, we will see:

```
ps1> ls -l
total 16
-rw-r--r--  1 mrg mrg 213 Mar  9 12:13 hello.qsub
-rw-----  1 mrg mrg   0 Mar  9 12:17 hello.qsub.e614
-rw-----  1 mrg mrg 487 Mar  9 12:17 hello.qsub.o614
```

Two new files have been created: `hello.qsub.e614` and `hello.qsub.o614`. These files contain anything printed to `STDOUT` or `STDERR`. Unless changed in the `qsub` command, the output and error files will always have the form "`<scriptname>.[e|o]<jobid>`"

```
ps1> cat hello.qsub.e614
ps1> cat hello.qsub.o614
PROLOGUE(614.hoppy): Prologue starting 09 Mar 2006 12:17:17(PST)
PROLOGUE(614.hoppy): jobid:614.hoppy, user:mrg, group:mrg
PROLOGUE(614.hoppy): MOM: kpz6379, sisters: KPZ6379
PROLOGUE(614.hoppy): Configuring KPZ6379 .... OK
PROLOGUE(614.hoppy): Prologue Complete
hello, world
EPILOGUE(614.hoppy): Epilogue starting 09 Mar 2006 12:17:18(PST)
EPILOGUE(614.hoppy): jobid:614.hoppy, user:mrg, group:mrg
EPILOGUE(614.hoppy): Cleaning up KPZ6379 .... OK
EPILOGUE(614.hoppy): Epilogue Complete
```

There weren't any errors, so the "e" file is empty. The output file contains quite a bit more than just our "hello, world"- it includes output from the prologue and epilogue scripts that set up the MOM (primary) node. This output is useful when problems occur- it contains the list of hosts assigned to the job, the account running the job, and a time-frame of when things were happening.

## A Second Job - The Job Environment and Working Directory

Glossed over in that first example are concepts such as the environment and working directory for the job.

Your initial working directory is your home directory *on the primary (MOM) node assigned to your job*- this will be your "fred" home directory on all nodes, which is mounted when the job starts.

However, you are not restricted to this one directory. There is disk space on the local node for your computation- the circumstances under which you'll want to use this disk space vary, but we'll examine that in greater detail later. The directory is accessed via the job environment variable "\$TMPDIR" which is set by Torque when your job starts.

DO NOT set or change this environment variable without fully understanding the repercussions.

On most nodes, this directory will be `/var/scratch/<jobid>`. The directory is created when the job starts and removed (contents and all) at the job's end. You have full access to this directory to create, move, and delete files.

The `$TMPDIR` environment is but one useful variable set by Torque. The login environment on the node is your standard login environment extended with a few Torque-specific variables:

`TMPDIR`

The directory designated for temporary (scratch) space- this directory exists only for the duration of the job on the node.

`PBS_O_HOST`

the name of the host upon which the qsub command is running.

`PBS_O_QUEUE`

the name of the original queue to which the job was submitted.

`PBS_O_WORKDIR`

the absolute path of the current working directory of the qsub command.

`PBS_ENVIRONMENT`

set to `PBS_BATCH` to indicate the job is a batch job, or to `PBS_INTERACTIVE` to indicate the job is a PBS interactive job, see `-I` option.

`PBS_JOBID`

the job identifier assigned to the job by the batch system.

`PBS_JOBNAME`

the job name supplied by the user.

`PBS_NODEFILE`

the name of the file contain the list of nodes assigned to the job (for parallel and cluster systems).

`PBS_QUEUE`

the name of the queue from which the job is executed.

`PBS_O_HOME`

`PBS_O_LANG`

`PBS_O_LOGNAME`

`PBS_O_PATH`

`PBS_O_MAIL`

`PBS_O_SHELL`

`PBS_O_TZ`

Set from the submitting environment- i.e. `PBS_O_LANG` is set from `LANG` in the submitting environment

Now let's try out some of these things:

```

ps1> cat hello.2.qsub
#
# This example shows the environment on the
# node assigned to the job as well as use of the
# temporary directory created by TMPDIR
#
# Request one node and 10 minutes of run-time
#PBS -l nodes=1,walltime=00:10:00
#
# Use the bourne-again shell
#PBS -S /bin/bash
#
echo "Hello, world... here we go...."
/bin/echo -n "Home directory is mounted from: " ; mount | grep $HOME
/bin/echo -n "Current working directory is: " ; pwd
echo "Let's find that temporary directory: \"${TMPDIR}\""
cd $TMPDIR
/bin/echo -n "Current working directory is: " ; pwd
df -k `pwd`
echo
echo
echo "Lets see our environment"
env

echo "Done!"
exit 0

```

Submit as before and wait for the output:

```

ps1> cat hello.2.qsub.o615
PROLOGUE(615.hoppy): Prologue starting 09 Mar 2006 13:09:13(PST)
PROLOGUE(615.hoppy): jobid:615.hoppy, user:mrg, group:mrg
PROLOGUE(615.hoppy): MOM: kpz6401, sisters: KPZ6401
PROLOGUE(615.hoppy): Configuring KPZ6401 .... OK
PROLOGUE(615.hoppy): Prologue Complete
Hello, world... here we go....
Home directory is mounted from: fred:/export/home00/mrg on /home/mrg
type nfs
(rw,nobrowse,intr,rsz=32768,wsz=32768,addr=140.107.36.21)
Current working directory is: /home/mrg
Let's find that temporary directory: "/var/scratch/615.hoppy"
Current working directory is: /var/scratch/615.hoppy
Filesystem          1K-blocks      Used Available Use% Mounted on
/dev/md0             60577868      2846424  54654224   5% /var/scratch

Lets see our environment
MANPATH=
HOSTNAME=kpz6401
PVM_RSH=/usr/bin/ssh
SHELL=/bin/bash
... trimmed to save space ...

```

```

PBS_O_HOST=hobby
PBS_VNODENUM=0
LOGNAME=mrg
PBS_QUEUE=public
PBS_O_MAIL=/var/spool/mail/mrg
LESSOPEN=|/usr/bin/lesspipe.sh %s
PBS_NODEFILE=/var/spool/torque/aux//615.hobby
G_BROKEN_FILENAMES=1
PBS_O_PATH=/home/mrg/bin/Linux:/opt/moab/bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
_=/bin/env
OLDPWD=/home/mrg
Done!
EPILOGUE(615.hobby): Epilogue starting 09 Mar 2006 13:09:14(PST)
EPILOGUE(615.hobby): jobid:615.hobby, user:mrg, group:mrg
EPILOGUE(615.hobby): Cleaning up KPZ6401 .... OK
EPILOGUE(615.hobby): Epilogue Complete
ps1>

```

This should give you some basic idea of what kind of environment your script will be running in. There's many different ways these variables can be used depending on how much your script needs to interact with the resource manager.

Another valuable tool is the `-v` flag. This allows you to make other environment variables available within the execution environment. The flag takes a comma-separated list of variables (or **variable=value** pairs) and creates those variables within the execution environment:

```

ps1> cat ./hello.3.qsub
# Request one node and 10 minutes of run-time
#PBS -l nodes=1,walltime=00:10:00
#
# Use the bourne-again shell
#PBS -S /bin/bash
#
# print the contents of the "GREETING"
# environment variable
#
echo $GREETING
exit 0

ps1>

```

Now we can, at submit time, control the output from our job script:

```

ps1> qsub -v GREETING="Hi World" hello.3.qsub
621.hobby
ps1> cat hello.3.qsub.o621
PROLOGUE(621.hobby): Prologue starting 09 Mar 2006 13:53:38(PST)
PROLOGUE(621.hobby): jobid:621.hobby, user:mrg, group:mrg
PROLOGUE(621.hobby): MOM: kpz6394, sisters: KPZ6394
PROLOGUE(621.hobby): Configuring KPZ6394 .... OK
PROLOGUE(621.hobby): Prologue Complete

```

### Hi World

```
EPILOGUE(621.hoppy): Epilogue starting 09 Mar 2006 13:53:39(PST)
EPILOGUE(621.hoppy): jobid:621.hoppy, user:mrg, group:mrg
EPILOGUE(621.hoppy): Cleaning up KPZ6394 .... OK
EPILOGUE(621.hoppy): Epilogue Complete
ps1> export GREETING="Hiya World"
ps1> qsub -v GREETING hello.3.qsub
622.hoppy
ps1> cat hello.3.qsub.o622
PROLOGUE(622.hoppy): Prologue starting 09 Mar 2006 13:55:41(PST)
PROLOGUE(622.hoppy): jobid:622.hoppy, user:mrg, group:mrg
PROLOGUE(622.hoppy): MOM: kpz6394, sisters: KPZ6394
PROLOGUE(622.hoppy): Configuring KPZ6394 .... OK
PROLOGUE(622.hoppy): Prologue Complete
Hiya World
EPILOGUE(622.hoppy): Epilogue starting 09 Mar 2006 13:55:42(PST)
EPILOGUE(622.hoppy): jobid:622.hoppy, user:mrg, group:mrg
EPILOGUE(622.hoppy): Cleaning up KPZ6394 .... OK
EPILOGUE(622.hoppy): Epilogue Complete
```

This example shows two ways of exporting this variable to the execution environment- first, explicitly setting the value on the qsub command line, the other by setting the value in the shell, specifying just the variable name to qsub.

You can also put -v arguments within the script using the "#PBS" form:

```
ps1> cat hello.4.qsub
# Request one node and 10 minutes of run-time
#PBS -l nodes=1,walltime=00:10:00
#
# Use the bourne-again shell
#PBS -S /bin/bash
#
# print the contents of the "GREETINGS"
# environment variable- expect that in the
# current environment
#PBS -v GREETING
#
echo $GREETING
exit 0
```

```
ps1> export GREETING="Hello, World"
ps1> qsub hello.4.qsub
623.hoppy
ps1> cat hello.4.qsub.o623
PROLOGUE(623.hoppy): Prologue starting 09 Mar 2006 14:01:09(PST)
PROLOGUE(623.hoppy): jobid:623.hoppy, user:mrg, group:mrg
PROLOGUE(623.hoppy): MOM: kpz6394, sisters: KPZ6394
PROLOGUE(623.hoppy): Configuring KPZ6394 .... OK
PROLOGUE(623.hoppy): Prologue Complete
Hello, World
EPILOGUE(623.hoppy): Epilogue starting 09 Mar 2006 14:01:10(PST)
EPILOGUE(623.hoppy): jobid:623.hoppy, user:mrg, group:mrg
EPILOGUE(623.hoppy): Cleaning up KPZ6394 .... OK
EPILOGUE(623.hoppy): Epilogue Complete
ps1>
```

This is an extremely useful method for changing the behavior of a script at run-time. For example, it enables you to write one script to handle many different file-names instead of creating a separate job script for each.

## A Third Job - File Staging

All our compute nodes have access to your fred home directory. Thus, your jobs can work exclusively within that directory, obviating any need to move files around. However, there are times when this approach fails. If your task imposes a high file-IO load (i.e. reads and writes files frequently) overall performance will suffer, sometimes greatly. In these circumstances, performance can be improved by using the local scratch directory on the node. This raises the problem of getting files into the node.

A powerful feature of Torque is the ability to "stage" files into and out of the execution node. It is possible to copy files within the job script- using staging makes your job more robust:

- if stage-in of a file fails, the job is re-queued, not failed
- stage-out allows partial files to be copied back prior to being deleted, easing the task of debugging
- staging isn't counted against your walltime

Let's try the "hello, world" example, but this time, let's use staging to manage files:

```
ps1> cat hello.5.qsub
# Request one node and 10 minutes of run-time
#PBS -l nodes=1,walltime=00:10:00
#
# Use the bourne-again shell
#PBS -S /bin/bash
#
# print the contents of the "GREETINGS"
# environment variable- expect that in the
# current environment
#PBS -v GREETING
#
# This allows us to remove the "interesting"
# bits from the STDOUT and STDERR files typically
# generated- we'll write to a local file then
# stage it out using torque
#
#PBS -W stageout=$TMPDIR/hello.out@gazoo:$HOME/examples/hello-
example/hello.out
#
echo $GREETING > $TMPDIR/hello.out
exit 0
ps1> echo $GREETING
Hello, World
ps1> qsub hello.5.qsub
626.hoppy
```

In this example, we are copying out the results of our application- from a local directory on the execution node to our home directory on hoppy.

The only variables understood by the staging mechanism are TMPDIR, HOME, and PBS\_JOBID.

The syntax of the stage-out request is:

```
stageout=<local path>@<remote host>:<path>
```

This copies "local path" on the execution node to "path" on "remote host"

So let's look at what the job produced:

```
ps1> ls -l
total 56
-rw-r--r-- 1 mrg mrg 213 Mar  9 12:13 hello.1.qsub
-rw-r--r-- 1 mrg mrg 632 Mar  9 13:08 hello.2.qsub
-rw-r--r-- 1 mrg mrg 219 Mar  9 13:49 hello.3.qsub
-rw-r--r-- 1 mrg mrg 279 Mar  9 15:30 hello.4.qsub
-rw-r--r-- 1 mrg mrg 550 Mar  9 15:31 hello.5.qsub
-rw----- 1 mrg mrg  0 Mar  9 15:34 hello.5.qsub.e626
-rw----- 1 mrg mrg 474 Mar  9 15:34 hello.5.qsub.o626
-rw----- 1 mrg mrg  13 Mar  9 15:34 hello.out
ps1> cat hello.5.qsub.o626
PROLOGUE(626.hoppy): Prologue starting 09 Mar 2006 15:34:05(PST)
PROLOGUE(626.hoppy): jobid:626.hoppy, user:mrg, group:mrg
PROLOGUE(626.hoppy): MOM: kpz6369, sisters: KPZ6369
PROLOGUE(626.hoppy): Configuring KPZ6369 .... OK
PROLOGUE(626.hoppy): Prologue Complete
EPILOGUE(626.hoppy): Epilogue starting 09 Mar 2006 15:34:05(PST)
EPILOGUE(626.hoppy): jobid:626.hoppy, user:mrg, group:mrg
EPILOGUE(626.hoppy): Cleaning up KPZ6369 .... OK
EPILOGUE(626.hoppy): Epilogue Complete
ps1>
```

So now, our "hello, world" does not appear in the standard-out file. We captured that to another file during execution of the script, then copied it back out to our home directory:

```
ps1> cat hello.out
Hello, World
ps1>
```

Neat, huh? Staging files in works similarly, though the copy is reversed:

```
ps1> cat hello.6.qsub
# Request one node and 10 minutes of run-time
#PBS -l nodes=1,walltime=00:10:00
#
# Use the bourne-again shell
#PBS -S /bin/bash
#
# Now we're staging-in a file to the execution
# node and generating our hello world from that.
#PBS -W stagein=$TMPDIR/hello.tmp@gazoo:$HOME/examples/hello-
example/hello.dat
#
# This allows us to remove the "interesting"
# bits from the STDOUT and STDERR files typically
# generated- we'll write to a local file then
# stage it out using torque
#
```

```
#PBS -W stageout=$TMPDIR/hello.out@gazoo:$HOME/examples/hello-
example/hello.out
#
cd $TMPDIR
echo "DEBUGGING: now in `pwd`"
ls -l
cat hello.tmp > $TMPDIR/hello.out
echo "DEBUGGING: wrote output"
ls -l
exit 0

ps1> cat hello.dat
HELLO, WORLD
ps1> qsub hello.6.qsub
628.hoppy
```

If all goes well, we should have a "hello.out" in addition to our standard output:

```
ps1> cat hello.out
HELLO, WORLD
```

Which we do! Success. Now we can look at the standard output and error files for other interesting tid-bits:

```
ps1> cat hello.6.qsub.o628
PROLOGUE(628.hoppy): Prologue starting 09 Mar 2006 15:47:45(PST)
PROLOGUE(628.hoppy): jobid:628.hoppy, user:mrg, group:mrg
PROLOGUE(628.hoppy): MOM: kpz6369, sisters:
PROLOGUE(628.hoppy): Prologue Complete
DEBUGGING: now in /var/scratch/628.hoppy
total 4
-rw----- 1 mrg mrg 13 Mar  9 15:44 hello.tmp
DEBUGGING: wrote output
total 8
-rw----- 1 mrg mrg 13 Mar  9 15:47 hello.out
-rw----- 1 mrg mrg 13 Mar  9 15:44 hello.tmp
EPILOGUE(628.hoppy): Epilogue starting 09 Mar 2006 15:47:45(PST)
EPILOGUE(628.hoppy): jobid:628.hoppy, user:mrg, group:mrg
EPILOGUE(628.hoppy): Epilogue Complete
```

The output (.o) file we can now use for debugging, since output is being saved to a different file. In this output file, we can see the staged-in file (which is present prior to execution) as well as our generated file.

## Parallel Jobs

The obvious next step is running a job across multiple nodes. However, this is not exactly straightforward. This is typically done with a parallel execution environment such as MPI or PVM, which use specialized processes and daemons to run applications. These parallel execution environments handle the passing of data between the different processes and maintaining the state of various worker processes.

This tutorial does not go into detail about running parallel jobs, rather just providing some high-level concepts behind requesting multiple nodes. For an in-depth discussion for running parallel jobs, see the Parallel Job Tutorial, available at [http://hoppy/hpcwiki/index.php/Parallel\\_Job\\_Tutorial](http://hoppy/hpcwiki/index.php/Parallel_Job_Tutorial).

We'll start with a very simple example using two nodes:

```
ps1> cat parallelhi.qsub
# Request one node and 10 minutes of run-time
#PBS -l nodes=2,walltime=00:10:00
#
# Use the bourne-again shell
#PBS -S /bin/bash
#
# print "hello, world" on stdout
#
GREETING="hello, world"
for NODE in `cat ${PBS_NODEFILE}` ;
do
    echo "Starting job on ${NODE}"
    ssh $NODE "/bin/echo -n \"${GREETING} \" ;uname -n ; "
done
exit 0

ps1> qsub parallelhi.qsub
636.hoppy
ps1> qstat 636
Job id              Name                User                Time Use S Queue
-----
636.hoppy           parallelhi.qsub     mrg                  0 Q public
ps1> cat parallelhi.qsub.o636
PROLOGUE(636.hoppy): Prologue starting 09 Mar 2006 16:22:03(PST)
PROLOGUE(636.hoppy): jobid:636.hoppy, user:mrg, group:mrg
PROLOGUE(636.hoppy): MOM: kpz6369, sisters: KPZ6369 KPZ6458
PROLOGUE(636.hoppy): Configuring KPZ6369 .... OK
PROLOGUE(636.hoppy): Configuring KPZ6458 .... OK
PROLOGUE(636.hoppy): Prologue Complete
Starting job on KPZ6369
hello, world kpz6369
Starting job on KPZ6458
hello, world kpz6458
EPILOGUE(636.hoppy): Epilogue starting 09 Mar 2006 16:22:04(PST)
EPILOGUE(636.hoppy): jobid:636.hoppy, user:mrg, group:mrg
EPILOGUE(636.hoppy): Cleaning up KPZ6369 .... OK
EPILOGUE(636.hoppy): Cleaning up KPZ6458 .... OK
EPILOGUE(636.hoppy): Epilogue Complete
ps1>
```

Now this is **not** a parallel task- but it does show some of the basics.

Additional nodes are selected in the *resource list* (the arguments to the `-l` flag). In this example, we request two "nodes", which torque interprets as "hosts". This *resource list* allocates a single CPU on each node. If you want to use multiple processors on a node, you must request that with an additional resource request, vis:

```
#PBS -l nodes=2:ppn=2
```

The hosts are configured such that you can use SSH between the nodes your job is allocated without entering a password (this is necessary for PVM and MPI applications).

One special environment variable you will typically use is `PBS_NODEFILE`, which is a text file containing a list of all the nodes assigned to the job. The first node listed is the MOM for the execution environment- it is to this node (and only this node) that files are staged.

## **Job Management**

### Monitoring

`qstat/showq` Job states, queued, held, etc.

### Managing

`qalter, qsig, qdel` Holding jobs, re-running, jobs, ordering jobs, dependancies, etc.

### Other Job Preferences

Mail notifications, job names