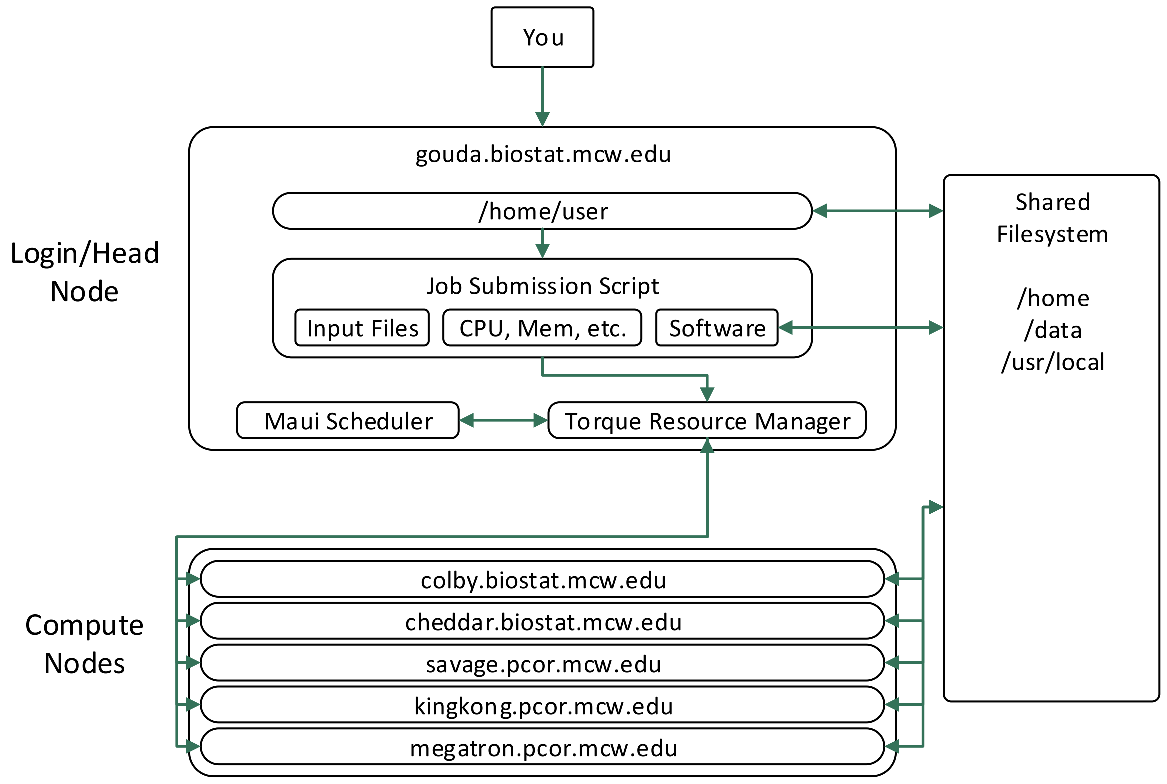
Cheese Cluster User Guide

Cheese Cluster Design:



1 login/head node – gouda.biostat.mcw.edu

5 compute nodes – colby.biostat.mcw.edu, cheddar.biostat.mcw.edu, savage.pcor.mcw.edu, kingkong.pcor.mcw.edu, megatron.pcor.mcw.edu

Cluster Etiquette:

A cluster is a wonderful shared resource that allows users to store files and run jobs simultaneously. In order to maintain proper function and use, please follow these guidelines:

1. All jobs must be run through the queueing system.

   Reason: For the cluster to work properly as a shared resource, all jobs must go through the queueing system. If you are running jobs outside of the queueing system on compute nodes, you may cause those nodes to fail, and other users may lose work as a result.

2. Do not start computationally intensive work on the cluster head node.

   Reason: The head node runs user logins, manages jobs, and mounts data storage. All three of those services must work for the cluster to function. If you start intensive computing on the head node, you may cause some or all of those services to fail or the node itself to fail, resulting in lost work for you and others.

3. User login is restricted to the cluster head node unless access to a compute node is needed to debug a failed job.

   Reason: Any processing on a compute node that is done outside the queueing system can cause the node to fail. Simply put, if you're not supposed to be computing there, you don't need to be logged in.

Job Submission & Monitoring

Gouda uses Torque and Maui for management and scheduling. This is a well-documented solution for high performance computing systems. Below you will find information on common TORQUE/Maui commands for submitting, monitoring, and diagnosing your jobs, as well as helpful examples for writing your own job submission scripts.

Common TORQUE Shell Commands

qsub	submit a job	$ qsub *myjob.sh*
qstat	show status of jobs	$ qstat
qdel	delete a job	$ qdel *job_id*
qhold	place a hold on a job	$ qhold *job_id*
qrls	release a hold on a job	$ qrls *job_id*

Common Maui Shell Commands

checkjob	display job information	$ checkjob *job_id*
showbf	show resource availability	$ showbf
showq	display queue of active and idle jobs	$ showq
showstart	show estimated start time of job	$ showstart *job_id*
showstats	show usage stats for user	$ showstats -u *user_id*

Monitoring Jobs

qstat -a	list all jobs
qstat -u *user_id*	list jobs of *user_id*
qstat -f	list full information of all jobs
qstat -f *job_id*	list full information about *job_id*
qstat -r	list all running jobs
pbsnodes	list full information about compute nodes

Batch Processing Jobs

Batch processing is the most common and effective method of submitting work to RCC clusters. In batch processing the queuing system takes control of the user's submitted job, runs it on the appropriate compute resource, and returns the output to the user. The queuing system accepts jobs in the form of submission scripts, which define the parameters of the job, i.e. # of nodes, # of processors, input files, output files, etc.

Torque Job Scripts

- Example job Script - *myTORQUEscript.sh*:

```
#!/bin/bash
#PBS -N myjob                      # Set job name to myjob
#PBS -m ae                         # Email status when job completes
#PBS -M your@email.address         # Email to this address
#PBS -l nodes=1:ppn=4              # Request 1 node with 4 processors
#PBS -l mem=8gb                    # Request 8gb memory
#PBS -l walltime=1:00:00           # Request 1hr job time
#PBS -j oe                         # Join output with error output
#PBS -q queue                      # Request specific queue

cd $PBS_O_WORKDIR                  # Change to submission directory

command input_file output_file     # Run your command
```

Submit Job

```
$ qsub myTORQUEscript.sh            # Submit job to TORQUE queuing system
```

Monitor Job

```
$ qstat                             # List all jobs
$ qstat -f job_id                   # List all information about job_id
```

Interactive Jobs

While batch processing is best, the TORQUE queuing system also supports interactive workload. Interactive jobs are best used for iterative workflow requiring user input. Example use includes interactive tools such as RStudio, MATLAB, or SAS.

Submit Job

```
[test@gouda ~]$ qsub -I
qsub: waiting for job 92 to start
qsub: job 92 ready
```

This will result in a new prompt:

```
[test@megatron ~]$
```

Notice you are now on the compute node kepler04 and ready to submit your workflow interactively.

End Job

```
[test@megatron ~]$ exit
logout

qsub: job 92 completed
```

Windowed Applications

Some applications that you may run in an interactive job will require X Windows for their GUI interface. Examples include Matlab, RStudio, etc.

- Submit an interactive job with X Windows enabled:

```
[test@gouda~]$ qsub -I -X
```

Interactive Job TORQUE Scripts

You may wish to specify job parameters in a job script for your interactive job.

- Example interactive job script:

```
#!/bin/bash
#PBS -N myjob                       # Set job name to myjob
#PBS -l nodes=1:ppn=4               # Request 1 node with 4 processors
#PBS -l mem=8gb                     # Request 8gb memory
#PBS -l walltime=1:00:00            # Request 1hr job time
#PBS -I                             # Request Interactive job
cd $PBS_O_WORKDIR                   # Change to submission directory
```

Available Queues

Queue	Max Walltime	Max Cores	Comments
small	72 hrs	8	Single-node jobs using up to 8 cores
medium	96 hrs	32	Multi-node jobs up to 32 cores
large	168 hrs	128	Multi-node jobs up to 128 cores

Diagnosing Job Failure

Job failure can occur due to user error, system failure, or other compounding factors. It is often the most frustrating part of computational work. Please follow these steps if you think your job has failed:

- What is the job status? Use command **qstat -j *job_id***.
  - Job status is **C**. Job has cleared, check logs for job completion or errors.
  - Job status is **H**. Job is held and will not run.
- Does the job violate queue policies? Use command **showq**.
  - Job is eligible. Requested resource may not be available. Job will run when available.
  - Job is blocked. Job will not run due to violation of queue policy. Use command **checkjob -v *job_id***.
- Use **tracejob** to diagnose job failure.

```
$ tracejob [-a|s|l|m|q|v|z] [-c count] [-w size] [-p path] [ -n <DAYS>] [-
f filter_type] <JOBID>
-p : path to PBS_SERVER_HOME
-w : number of columns of your terminal
-n : number of days in the past to look for job(s) [default 1]
-f : filter out types of log entries, multiple -f's can be specified
error, system, admin, job, job_usage, security, sched, debug, debug2, or
absolute numeric hex equivalent
-z : toggle filtering excessive messages
```

```
-c : what message count is considered excessive
-a : don't use accounting log files
-s : don't use server log files
-l : don't use scheduler log files
-m : don't use MOM log files
-q : quiet mode - hide all error messages
-v : verbose mode - show more error messages

$ tracejob -v job_id
```

Finding More Information

TORQUE/Maui is a popular, widely used solution with lots of available documentation. While the basics are provided here, specific details of each TORQUE/Maui commands are available online.

http://docs.adaptivecomputing.com/maui/pdf/mauiadmin.pdf

Software

Using R:

R is traditionally used as an interactive data analysis tool. It works well on a desktop computer where one user can make use of all resources. In a cluster environment, users are not alone and must share the available resources. This differnece can someitmes lead to R users running unwanted jobs in the wrong place, such as the cluster login/head node, which can negatively affect other cluster users. Therefore, RCC has defined several use cases of the R software package.

Small Interactive Jobs:

Small interactive jobs include light plotting, simple analysis of small data sets, etc. These jobs never take more than one core, a few GB of memory, and never last more than a few min. These small, fast jobs are allowed on the cluster login node, gouda.biostat.mcw.edu. However, use caution when running these jobs and double-check that they will not use larger resources. Users that violate this policy will be warned and advised on proper usage.

Multi-core Jobs Using Rscript:

Multi-core jobs should be run on the cluster compute nodes using the Torque queuing system. There are several options for running these jobs in Torque, including the Rscript command and the BatchJobs library. Both methods interface R with Torque, however, their use cases are different. The Rscript command should be used when you have written an R program .r file and would like to run this script on the cluster. The BatchJobs library should be used when you would like to test individual functions in a semi-interactive way and submit this work to the cluster.

Example Torque submission script:

```
#!/bin/bash
#PBS -N myRexample                  # Set job name to myjob
#PBS -l nodes=1:ppn=4               # Request 1 node with 4 processors
#PBS -l mem=8gb                     # Request 8gb memory
#PBS -l walltime=1:00:00            # Request 1hr job time
#PBS -V                             # Use submission environment

cd $PBS_O_WORKDIR                   # Change to submission directory

Rscript Rtest.r                     # Run R script “Rtest.r”
```

Submit the job:

```
$ qsub myRtest.sh             # Submit job to Torque
```

Multi-core Jobs Using BatchJobs:

BatchJobs is an R library that interfaces the R command-line with the cluster's Torque queuing system.

Load and start R:

```
$ R
```

Load R BatchJobs library:

```
> library(BatchJobs)
Loading required package: BBmisc
Sourcing configuration file:
'/usr/lib64/R/library/BatchJobs/etc/BatchJobs_global_config.R'
BatchJobs configuration:
  cluster functions: Torque
  mail.from:
  mail.to:
  mail.start: none
  mail.done: none
  mail.error: none
  default.resources: nodes=1, cores=1, memory=2gb, walltime=8:00:00
  debug: FALSE
  raise.warnings: FALSE
  staged.queries: TRUE
  max.concurrent.jobs: Inf
  fs.timeout: NA
```

Define data and function:

```
> my_data <- (1:10)
> my_func <- function(x) x^2
```

Define an object to store jobs (creates a directory "batchtest-files"):

```
> reg <- makeRegistry(id = "batchtest")
```

Map data and function to jobs in "batchtest" object:

```
> jobs <- batchMap(reg, my_func, my_data)
```

Submit jobs to cluster (change nodes, cores, mem, and walltime to fit needs):

```
> jobsubmit <- submitJobs(reg, resources = list(nodes = 1, cores = 1, mem
= 2gb, walltime = 8:00:00))
```

Check job results:

```
> reduceResultsVector(reg, fun = function(job, res) res, progressbar =
FALSE)
Syncing registry ...
Reducing 10 results...
  1   2   3   4   5   6   7   8   9  10
  1   4   9  16  25  36  49  64  81 100
```

Results files:

```
batchtest-files/
|-- BatchJobs.db
|-- conf.RData
|-- exports
|-- functions
|   `-- c0000e09480f70b05365872c2e90ce8a.RData
|-- jobs
|   |-- 01
|   |   |-- 1-result.RData
|   |   |-- 1.R
|   |   `-- 1.out
|   |-- 02
|   |   |-- 2-result.RData
|   |   |-- 2.R
|   |   `-- 2.out
|   |-- ...
|
|-- pending
|-- registry.RData
`-- resources
    `-- resources_1493676907.RData
```

Results can be displayed in vector format as shown above. Users may also want to view results in the output files. Each job registry object that is created starts a new file tree as shown here with the name *registryname*-files. Job results are located in numbered directories within the jobs directory. Output files are named *jobnumber*.out and are numbered in the order of submission.