Using Scripting Languages on the FAS Clusters

Stephen Weston

FAS HPC Core
Yale University

April 5, 2013
Scripting languages for data analysis

Scripting languages are quite popular for data analysis. Examples of scripting languages used for data analysis include:

- Matlab, Mathematica, R, Python
- Octave, Sage, Perl

Scripting languages are relatively easy to use and powerful.

- Rich variety of data structures
- Simplified memory management
- Wide variety of add-on packages

Can be used interactively or non-interactively.

- Develop scripts interactively
- Deploy scripts non-interactively

Generally low performance compared with Fortran, C, C++.
Clustering is very powerful and useful, but it may take some time to get used to them. Here are some reasons to go to that effort:

- Don’t want to tie up your own machine for many hours or days
- Have many long running jobs to run
- Want to run in parallel to get results quicker
- Need more disk space
- Want easy access to long term storage
- Want to use software installed on the cluster
- Need more memory
Limitations of the FAS clusters

Clusters are not the answer to all large scale computing problems. Some of the limitations of the FAS clusters are:

- Cannot run Windows programs
- Not really intended for interactive jobs (especially graphical)
- Jobs that run for weeks can be a problem (unless checkpointed)
- Nodes have a only moderate amount of memory
What is a cluster?

A cluster usually consists of a hundred to a thousand rack mounted computers, which I call **nodes**. It has one or two head nodes, or login nodes that are externally accessible, but most of the nodes are compute nodes and are only accessed from a login node via a batch queueing system, also called a **job scheduler**.

The CPU used on clusters may be similar to the CPU in your desktop computer, but in other respects they are rather different.

- No monitors, no CD/DVD drives, no audio or video cards
- Don’t always have a hard drive
- Distributed file system
- Moderate amount of RAM: 16 to 48 gigabytes of RAM
- Connected together by an internal network
## Summary of FAS Clusters

<table>
<thead>
<tr>
<th></th>
<th>BDJ</th>
<th>BDK</th>
<th>BDL</th>
<th>Omega</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total nodes</strong></td>
<td>128</td>
<td>192</td>
<td>128</td>
<td>704</td>
</tr>
<tr>
<td><strong>Cores/node</strong></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><strong>Mem/node</strong></td>
<td>16 GB/32 GB</td>
<td>16 GB</td>
<td>48 GB</td>
<td>36 GB/48 GB</td>
</tr>
<tr>
<td><strong>Home dir</strong></td>
<td>NFS</td>
<td>NFS</td>
<td>panasas</td>
<td>lustre</td>
</tr>
<tr>
<td><strong>Scratch dir</strong></td>
<td>lustre</td>
<td>lustre</td>
<td>panasas</td>
<td>lustre</td>
</tr>
<tr>
<td><strong>Temp dir</strong></td>
<td>tmpfs (RAM)</td>
<td>tmpfs (RAM)</td>
<td>tmpfs (RAM)</td>
<td>80 GB local disk</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>DDR IB</td>
<td>gigabit ETH</td>
<td>QDR IB</td>
<td>QDR IB</td>
</tr>
</tbody>
</table>
What is a batch queueing system?

A batch queueing system, or job scheduler, is to a cluster what an operating system is to a single computer, but it’s more like the batch-oriented operating systems from the 60’s and 70’s. You use the batch queueing system in order to run jobs on the cluster.

Examples of batch queueing systems include: Torque, Moab, LSF, PBS Pro, Condor, and SLURM. Torque and PBS Pro are both newer versions of PBS, or Open PBS, both of which are commercially supported.

The FAS clusters use a combination of Moab and Torque.
Why is a batch queueing system needed?

Batch queueing systems may seem old fashioned and awkward, but they are a necessary evil, since clusters would be complete chaos without them.

- Prevent hundreds of jobs from all running on the same node
- Attempt to provide fair node access to all users
- Provide priority access for some jobs
- Allow you to submit a job and then forget about it
Running jobs on a cluster

The most important rule of running jobs on a cluster is:

Don’t run your job on the login node

Running your job on the login node bypasses the job scheduler and can lead to chaos on the login node, possibly causing it to crash so that no one can access the cluster.

In addition, there are a number of things missing from the login nodes that can cause strange and obscure errors.
Running jobs on a cluster

There are two ways of running jobs: interactively and in batch mode.

Interactive mode

- Setting up data files
- Testing scripts on smaller problem
- Installing packages/modules
- Compiling/building programs

Batch mode

- Running your real work
Basic steps for running an interactive job

- Copy scripts and data to the cluster
- Ssh to a login node
- Allocate a compute node
- Move to appropriate directory
- Load the module file(s)
- Run the script
Example of an interactive job

Executed on your local machine:

$ rsync -azv ~/workdir netid@omega.hpc.yale.edu:
$ ssh netid@omega.hpc.yale.edu

Executed on the login node:

$ qsub -I -q fas_devel -l mem=4gb,walltime=1:00:00

Executed on the allocated compute node:

$ cd ~/workdir
$ module load Applications/R/2.15.3
$ R --slave -f compute.R
Basic steps for running a non-interactive job

- Copy scripts and data to the cluster
- Ssh to a login node
- Submit a job script
  - Move to appropriate directory
  - Load the module file(s)
  - Run the script
Example of a non-interactive/batch job

Executed on your local machine:

$ rsync -azv ~/workdir netid@omega.hpc.yale.edu:
$ ssh netid@omega.hpc.yale.edu

Executed on the login node:

$ qsub batch.sh
Here is a very simple batch script that executes an R script. It contains PBS directives that embed qsub options in the script itself, making it easier to submit. These options can be overridden by the command line arguments, however.

```bash
#!/bin/bash
#PBS -q fas_devel
#PBS -l mem=4gb
#PBS -l walltime=1:00:00
cd $PBS_O_WORKDIR
module load Applications/R/2.15.3
R --slave -f compute.R
```
Copy scripts and data to the cluster

On Linux and Mac OS/X, I use the standard scp and rsync commands to copy scripts and data files from my local machine to the cluster, or more specifically, to the login node. The rsync command is particularly useful if you have large files, some of which change occasionally. Rsync tries to minimize the amount of data that is transferred, only copying files that have changed.

$ scp -r ~/workdir netid@omega.hpc.yale.edu:
$ rsync -azv ~/workdir netid@omega.hpc.yale.edu:

On Windows, the WinSCP application is available from the Yale Software Library:

http://software.yale.edu/Library/Windows

Another option is “Bitvise ssh” which is free for individual use.
Ssh to a login node

You must first login to an Omega login node using `ssh`. From a Mac or Linux machine, you simply use the `ssh` command:

```
$ ssh netid@omega.hpc.yale.edu
$ ssh -Y netid@omega.hpc.yale.edu
```

From a Windows machine, you can choose from programs such as PuTTY or WinSCP, both of which are available from the Yale Software Library:

http://software.yale.edu/Library/Windows

For more information on using PuTTY, go to the following URL and search for "create ssh key":

https://hpc.research.yale.edu
Allocate a compute node / Submit a job script

qsub – Submit job for executing

$ qsub -I -q fas_normal -l mem=34gb,walltime=1:00:00
$ qsub -q fas_normal -l mem=34gb,walltime=1:00:00 batch.sh

qstat/showq – View status of jobs

$ qstat -u sw464
$ showq -w user=sw464
$ showq -w class=fas_normal

showstart – Get estimate of when a job will start

$ showstart 1273896

cHECKJOB – Get information about a job

$ checkjob -v 1273896
Specifying resources with `qsub`

When requesting a compute node, it is very important to specify the computing resources that are needed via the `qsub -l` option. For a sequential (ie. non-parallel) job, the most important resources are **walltime** and **mem**. It is best to always specify these resources, since they are also **limits**, and your job will be **killed** if it uses more than the requested amount.

- **mem**  Total amount of memory needed by job
- **walltime**  Maximum time to allow job to run
Requesting walltime with qsub

The value of \texttt{walltime} is particularly important because if you ask for too little time, your job will be killed, but if you ask for too much, it may not be scheduled to run for a long time.

- Specified as DD:HH:MM:SS (days, hours, minutes, seconds)
- Default is one hour
- Try to determine expected runtime during testing

The following requests four hours of time:

\>$\ qsub\ -q\ fas\_normal\ -l\ mem=34gb,walltime=4:00:00\ batch.sh$

If your job checkpoints itself periodically, then this decision is less critical.
Requesting memory with qsub

The value of mem is important because the default value is only 256 megabytes, so if you don’t specify a value, the job will very likely be killed for using too much memory. You also need to be careful not to ask for more than any node has, otherwise your job may never run.

Rather than figure out how much I need, I use the formula:

\[
mem = numnodes \times (mempernode - 2\text{GB})
\]  

(1)

I multiply by numnodes since mem is the total memory needed for the job.

For example, if I’m requesting ten nodes on Omega, I would request 340 gigabytes of memory:

```
$ qsub -q fas_normal -l nodes=10:ppn=8,mem=340gb batch.sh
```

Note that since fas_normal uses “whole node allocation”, there isn’t any reason to ask for the actual amount of memory needed, however that isn’t true on fas_devel and some of the special queues that allow multiple jobs per node.
Specify the queue

You must always specify a queue via the qsub -q argument on the FAS clusters. For the general purpose queues, the primary factor in choosing a queue is the value of **walltime** because the different queues have different restrictions on how long jobs can run.

Bulldog K is primarily intended for long running serial jobs, so most queues allow jobs to run for a week.

```
fas_very_long  4 weeks
  fas_long     1 week
  fas_high     1 week
fas_normal     1 week
  fas_devel    4 hours
```
Maximum walltime for Omega, BDJ, BDL

The maximum walltime allowed by the queues on the other FAS clusters are:

- fas\_very\_long 4 weeks
- fas\_long 3 days
- fas\_high 1 day
- fas\_normal 1 day
- fas\_devel 4 hours

For a job that I think will run for four days, I would use the fas\_very\_long queue on Omega. Note that I actually request five days to be on the safe side:

```bash
$ qsub -q fas\_very\_long -l mem=34gb,walltime=5:00:00:00 batch.sh
```

On Bulldog K I would use fas\_normal:

```bash
$ qsub -q fas\_normal -l mem=14gb,walltime=5:00:00:00 batch.sh
```
Module files

Much of the HPC software on the FAS clusters is installed in non-standard locations, mostly under the '/usr/local/cluster/hpc' directory. This makes it easier to maintain different versions of the same software allowing users to specify the version of an application that they wish to use. This is done with the module command.

For example, before you can execute Matlab, you need to initialize your environment by loading a Matlab “module file”:

```
$ module load Applications/Matlab/R2012b
```

This will modify variables in your environment such as PATH and LD_LIBRARY_PATH so that you can execute the `matlab` command.
Finding module files

There are many module files for the various applications, tools, and libraries installed on the FAS clusters. To find the module file that will allow you to run Matlab, use the ‘modulefind‘ command:

$ modulefind matlab
$ /usr/local/cluster/hpc/Modules/modulefind matlab

This will produce output like:

/home/apps/fas/ Modules:
Applications/Matlab/R2010b
Applications/Matlab/R2012b
Apps/Matlab/R2010b
Apps/Matlab/R2012b
/home/apps/geo/ Modules:
Applications/Applications/Matlab/R2010b
Applications/Applications/Matlab/R2012b

You can get a listing of available module files with:

$ module avail
Example "module load" commands

Here are some “module load” commands for scripting languages installed on Omega:

module load Langs/Python/2.7.3
module load Langs/Perl/5.14.2
module load Applications/R/2.15.3
module load Applications/Matlab/R2012b
module load Applications/Mathematica/9.0.1
Run your script

When you’re finally ready to run your script, you may have some trouble determining the correct command line, especially if you want to pass arguments to the script. Here are some examples:

$ python compute.py input.dat
$ R --slave -f compute.R --args input.dat
$ matlab -nodisplay -nosplash -nojvm < compute.m
$ math -script compute.m
$ MathematicaScript -script compute.m input.dat

You often can get help from the command itself using:

$ matlab -help
$ python -h
$ R --help
Notes on interactive mode

Enable X11 forwarding from ssh and qsub:

$ ssh -Y netid@omega.hpc.yale.edu
$ qsub -X -I -q fas_devel -l mem=34gb

Faster alternative:

$ ssh netid@omega.hpc.yale.edu
$ qsub -I -q fas_devel -l mem=34gb

Once job is running, execute from a different terminal window:

$ ssh -Y netid@omega.hpc.yale.edu
$ ssh -Y compute-XX-YY

where “compute-XX-YY” is the node allocated to your job.

Using VNC is possible on Omega and BDL, but not very easy.
Cluster Pitfalls: Incorrectly estimating required resources

- Exceeding requested limits can get your job killed
- Always specify mem and walltime
- Determining memory needs of scripts can be difficult
Cluster Pitfalls: Waiting a long time for the job to start

- Asking for a lot of walltime or nodes can make you wait
- Use showstart command to see if you're in trouble
- Resubmit if possible asking for less time, fewer nodes
- Asking for less memory probably isn’t going to help
Cluster Pitfalls: Using too much memory

- Best case is your job is killed
- Worst case is a system crash
Cluster Pitfalls: Submitting hundreds of small jobs

- Puts a big load on scheduler, slowing the system down for everyone
- Use SimpleQueue instead if possible
Cluster Pitfalls: Performing many small file operations

- Puts a big load on file servers slowing the system down for everyone
- Use Unix pipes if possible
- Use /tmp when appropriate
  - 80 gigabytes on Omega
  - Clean up when job is finished
Programming/scripting tips for clusters

- Develop and test on your desktop as much as possible
- Start testing on cluster with a smaller problem with fas_devel
- Use checkpointing if possible
- Use logging
- Try to reduce file I/O as much as possible
- Monitor the node during execution - top, ps, pstree
- If it’s not as fast/efficient as you’d hoped, come see us
SimpleQueue Introduction

SimpleQueue is a tool that executes commands in parallel on a cluster. It can be an easy way to parallelize any kind of script without requiring you to learn about that scripting language's parallel programming tools.

- Simple parallel command execution
- Written by Nick Carriero, Yale University
- Can execute any program that can be executed from the command line
- Similar to "GNU Parallel"
SimpleQueue advantages

- Provides easy way to parallelize your script
- Makes checkpointing easy
- Decrease walltime requirements and avoid fas\_very\_long
- Easier to manage than submitting multiple batch jobs
- Integrated with Torque/Moab for ease of use
- Customized for FAS clusters
- Much lower overhead per tasks versus multiple qsub’s
- Don’t need to know how to execute mpirun
SimpleQueue notes

Steps to parallelize a script using SimpleQueue:

- If necessary, modify the script to compute a subset of the job
- Create a task list containing commands to execute
- Create submit script using sqCreateScript command
- Submit the submit script

The lines in the task file often include commands that:

- Move to appropriate directory
- Load the module file(s)
- Run the script with appropriate arguments

Commands within a task should be separated by semi-colons. Tasks **must** be on a single line, although it can be a very long line.
SimpleQueue Example

Here’s an example task file called “tasks.txt”:

```bash
cd ~/job; module load Apps/R; R --slave -f x.R --args i1.dat
cd ~/job; module load Apps/R; R --slave -f x.R --args i2.dat
cd ~/job; module load Langs/python; python x.py i3.dat > o3.dat
cd ~/job; module load Langs/python; python x.py i4.dat > o4.dat
[ rest of the task file not shown ]
```

To run this in parallel on four nodes for eight hours:

```
$ module load Tools/SimpleQueue/3.0
$ sqCreateScript -n 4 -w 8:00:00 tasks.txt > batch.sh
$ qsub batch.sh
```

Note that you can execute sqCreateScript on the login node.
Parallel programming in R

There are lots of parallel programming packages available for R. Many people get confused trying to decide what to use, although now the “parallel” package comes in the standard R distribution.

- Rmpi
- snow
- multicore
- parallel
- foreach
  - doSNOW
  - doMC
  - doParallel
  - doMPI
Foreach Overview

The foreach package provides a looping construct that is a hybrid between a for-loop and an “lapply” function:

```r
install.packages('foreach')
library(foreach)
r <- foreach(i=1:10) %dopar% sqrt(i)
```

- Designed to be easy to use
- Auto-exports data to the workers
- Works with different parallel backends
- Support of iterators for input data
- ”Combine” functions support on-the-fly result processing
- Helps with chunking
- Supports nested loops
Auto-export of data

Many beginners get discouraged when a parallel program fails because a variable isn’t defined on the worker machines. The foreach package tries to solve that beginner’s problem without doing anything too stupid.

In this example the variable ‘x’ is auto-exported to the workers:

```r
x <- 10
r <- foreach(i=1:10) %dopar% {
  x + i
}
```

The `%dopar%` function performs a simple analysis of the symbols used in the body of foreach loop. It tries not to go crazy exporting everything. The ‘.export’ and ‘.noexport’ options can be used to override what is and isn’t exported.
Work with different parallel backends

Installing parallel programming packages can be difficult, so I liked the idea of separating foreach from the parallel backends. You can start using foreach without any parallel backend installed.

- Develop on Raspberry Pi – doSEQ
- Test on desktop – doParallel
- Deploy to cluster – doMPI

The parallel backend needs to be registered by calling a function such as “registerDoMC()” or “registerDoMPI()”. This allows a package to use foreach but allow the end user to register whatever parallel backend they choose.
Support of iterators

I think that support for iterators is my favorite feature of the foreach package. The standard for-loop and “lapply” function can only iterate over vectors and lists, “apply” can only iterate over arrays, but you can write an iterator for anything.

```r
library(iterators)
m <- matrix(rnorm(100), 10)
r <- foreach(a=m) %dopar% mean(a)
r <- foreach(a=iter(m, by='row')) %dopar% mean(a)
```

- Inputs don’t have to all be in memory
- Memory friendly
- Can iterate on things other than vectors and lists
- Variety of iterators are available
- iterators and itertools packages
- Supports iterating over contents to a file
Combine functions

The foreach “.combine” option allows you to specify a function that will be called to combine the task results.

```r
m <- matrix(rnorm(100), 10)
r <- foreach(x=m, .combine='c') %dopar% mean(x)
r <- foreach(x=m, .combine='+' ) %dopar% x

# Return a matrix by combining the results with 'cbind'
y <- rnorm(10)
z <- rnorm(10)
r <- foreach(x=m, s=y, .combine='cbind') %dopar% {
  s * x + z
}
```

- Handles results on the fly if supported by the backend
- Outputs don’t have to all be in memory
- Can be used to write results to a file
Parallel random forest

library(randomForest)
library(parallel)
library(doMC)
registerDoMC(detectCores())

x <- matrix(runif(500), 100)
y <- gl(2, 50)
ntree <- 1000

rf <- foreach(n=idiv(ntree, chunks=getDoParWorkers()),
               .combine='combine', .multicombine=TRUE,
               .packages='randomForest') %dopar% {
  randomForest(x, y, ntree=n)
}
library(parallel)
library(doMC)
registerDoMC(detectCores())

nstart <- 100

best <- function(...) {
  results <- list(...)
  i <- sapply(results, function(result) result$tot.withinss)
  results[[which.min(i)]]
}

result <- foreach(n=idiv(nstart, chunks=getDoParWorkers()),
  .combine='best', .multicombine=TRUE,
  .packages='MASS') %dopar% {
  kmeans(Boston, 4, nstart=n)
}