Unix systems/shell scripting/cluster computing

Brian Williamson

BIOST 561: Computational Skills For Biostatistics I

16 November 2017

Unix systems/shell scripting

Since there seems to be no hope of C becoming an expression language (in which case it would be potentially more suitable for the shell) there is equally no hope that the shell and C will be the same language.

Stephen R. Bourne (1977, from a talk in 2015)
British computer scientist, author of the Bourne shell (`sh`)

Motivation

Why not keep running R using a GUI and (R) scripts?

- Requires keeping R/RStudio open while the script is running
- Can take up significant resources on a personal machine
- May need to open multiple R/RStudio windows to run multiple jobs (there are some ways around this)

Many courses (e.g., 570s) and research will require a more sophisticated approach.

The solution!

The department has *cluster computers* for just this purpose!

Cluster computers:

- Allow you to submit multiple jobs at once
- Depending on the system, can schedule jobs for you
- Are optimized for high-throughput performance

Unix systems

Three common types of laptop/desktop operating systems: Windows, Mac, Linux.

Mac and Linux are both Unix-like!

What that means for us: Unix-like operating systems are equipped with “shells” that provide a command line user interface.

Navigating Unix/Linux

Some useful commands (StackOverflow has more!):

Command	Purpose
mkdir	create new directory
rmdir	remove a directory
cd	change directory
pwd	print current working directory (cwd)
ls	list all files and folders in cwd
mv	rename or move a file
cp	copy a file
rm	remove a file
ps	list running processes
top	print running update
cat	display a text file
more	display any file

Navigating Unix/Linux

Some useful shortcuts/more commands (StackOverflow has more!):

Shortcut	Purpose
*	select all items in the cwd
.	look in the cwd
..	look in the parent directory
chmod	change permissions for a file
`./program.sh`	run the executable file `program.sh`

Your best friend is Google if you run into problems!

Running GUI-less: the highlights

There are many nuances to running GUI-less (see the other set of slides), but here are a few of the highlights:

- Set a seed! This ensures that your results are reproducible (by you and others)
- Think hard about what output you want to save, and save it manually either in your R script or in a shell script
- Tell the computer where to look for the executable file to run your code (using a `#!`)
- R scripts (and the `Rscript` executable) can take parameters passed from the command line

Example: Robust SEs

Consider a sample of n observations generated according to

$$\begin{aligned} X_i &\overset{i.i.d.}{\sim} N(0,1), \\ u_i &\overset{i.i.d.}{\sim} N(0,1), \text{ independent of the } X_i, \\ Y_i \mid X_i, u_i &= \beta_0 + \beta_1 X_i + \epsilon_i, \text{ where} \\ \epsilon_i &= |X_i|\, u_i. \end{aligned}$$

Suppose we wish to estimate β_1 using linear regression. We clearly have some heteroskedasticity, so we can compare coverage of 95% CIs using model-based and robust standard errors for $\hat{\beta}_1$.

Robust SEs: the doOne() function

```
doOne <- function(n, beta) {
  ## create the data
  x <- rnorm(n, 0, 1)
  u <- rnorm(n, 0, 1)
  eps <- abs(x)*u
  beta0 <- 1
  y <- beta0 + beta*x + eps

  ## fit the linear regression model
  mod <- lm(y ~ x)

  ## extract model coefficients and SEs
  est <- coefficients(mod)[2]
  se <- vector("numeric", 2)
  se[1] <- sqrt(diag(vcov(mod)))[2]
  se[2] <- sqrt(diag(vcovHC(mod, "HC0")))[2]

  ## Create CIs
  ci <- est + se %o% qnorm(c(0.025, 0.975))
  cover <- beta > ci[, 1] & beta < ci[, 2]
  names(cover) <- c("Model", "Sandwich")

  ## return
  return(c(est, cover))
}
```

Script se_ex1.R

```
#!/usr/local/bin/Rscript
args <- commandArgs(TRUE)
## Parse the arguments
if(length(args) == 0) {
  print("No arguments supplied.")
} else {
  for (i in 1:length(args)) {
    eval(parse(text = args[i]))
  }
}
if(!exists("truebeta")) {
  truebeta <- 2
  cat("\n Note: Assmuming truebeta = 2 \n")
}
if(!exists("n")) {
  n <- 500
  cat("\n Note: Setting n = 500 \n")
}
if(!exists("seed")) {
  seed <- 547
  cat("\n Note: Setting seed = 547 \n")
}
if(!exists("B")) {
  B <- 1000
  cat("\n Note: setting B = 1000 \n")
}
```

Script se_ex1.R (continued)

```
doOne <- function(n, beta) {
 <See previous slide>
}
library(sandwich)
set.seed(seed)
system.time(output <- replicate(B, doOne(n = n, beta = truebeta)))
apply(output, 1, mean)
```

Running jobs: Rscript

On Box (or Cox, more on this later) we can run jobs using Rscript. For the SE example:

```
[brianw26@cox ~/biost_561]$ Rscript se_ex1.R > ex1_output.txt B=10000
seed=547 n=500 truebeta=2
```

... which creates an object `ex1_output.txt` containing all of the output we saved (only the average values and the system time).

Nothing else is saved! Takes 13284 seconds (3.7 hours).

Running jobs: multiple jobs at once

If we need to set off a collection of R processes, we can use & at the end of a line to allow entry of another line, e.g.,

```
[brianw26@cox ~/biost_561]$ Rscript se_ex1.R > ex1_output_200.txt B=10000
seed=547 n=200 truebeta=2 &
[1] 755
[brianw26@cox ~/biost_561]$ Rscript se_ex1.R > ex1_output_500.txt B=10000
seed=547 n=500 truebeta=2 &
[2] 786
```

Running jobs: multiple jobs at once

Or, we can write a shell script! This is especially useful if we need to set off multiple jobs at once, or we have different settings that our simulation needs to run with.

1. Make a script file of command-line commands (not R), e.g.,
   ```
   #!/bin/sh
   Rscript myscript.R n=100 seed=147 &
   Rscript myscript.R n=300 seed=347 &
   Rscript myscript.R n=500 seed=547
   ```
   and save this off as `myMasterScript.sh`

2. `chmod u+x myMasterScript.sh` grants Yo**u**, the file owner, permission to e**x**ecute the file

3. `./myMasterScript.sh` runs the script

Example: robust SE shell script

For the robust SE example, this would be

```
[brianw26@cox ~/biost_561]$ more se_ex1.sh
#!/bin/sh

Rscript se_ex1.R > ex1_output_n_100.txt B=10000
seed=101 n=101 truebeta=2 &
Rscript se_ex1.R > ex1_output_n_300.txt B=10000
seed=301 n=101 truebeta=2 &
Rscript se_ex1.R > ex1_output_n_500.txt B=10000
seed=501 n=101 truebeta=2 &

[brianw26@cox ~/biost_561]$ chmod u+x se_ex1.sh
[brianw26@cox ~/biost_561]$ ./se_ex1.sh
```

Example: robust vs model-based SEs

After 3.7 hours, we get the following output (what could we have saved to make things slightly more interesting?)

```
[brianw26@cox ~/biost_561]$ cat ex1_output.txt
     user   system   elapsed
13286.337    0.343 13284.293
       x    Model   Sandwich
1.999183 0.738700   0.948900
```

Example: robust vs model-based SEs (interesting?)

Here's some code* from the end of an R script that saves off the coefficient estimates and whether or not the CI covers the truth:

```
library(sandwich)
set.seed(seed)
system.time(output <- replicate(B, doOne(n = n, beta = truebeta)))
# apply(output, 1, mean)
save(file=paste("ex3_output_n_", n, "_beta_", truebeta, "_b_", B,
                "_seed_", seed, ".Rdata", sep = ""))
```

* the rest is the same as `se_ex1.R`

Example: robust vs model-based SEs (interesting?)

Cluster computing/shell scripting

When I was younger, I thought of myself as a pretty good programmer...

Ian Sommerville (2015)
Scottish computer scientist

Cluster computing

All of us can constantly improve our programming.

In addition to learning new languages (e.g., shell), we can also learn new ways to speed up/optimize code.

The clusters allow us to do the latter.

Running GUI-less: on to the clusters

Sometimes (all the time) we do not want to leave a R session open on our laptop while a simulation runs.

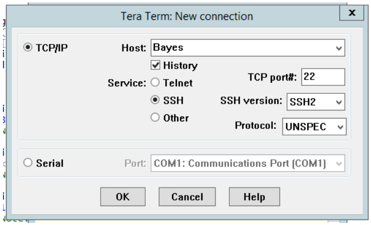
There are two main shared department resources for computing (in my experience): Cox (not a cluster) and Bayes (a cluster).

- Cox: 12 core computer

- Bayes:
  - Compute cluster with 4 compute nodes, each with 12 cores (additional PI-specific nodes)
  - Designed for high-throughput computing
  - Managed by Sun Grid Engine (SGE)

Logging on to a cluster: Windows (including Box)

1. Open Tera Term [or your favorite secure shell (SSH) client]
2. Enter the address of your favorite cluster, e.g., `bayes.biostat.washington.edu`
3. Make sure that the “New connection” window is filled out as in the figure, except for “Host” (on Box things are different)
4. Click `OK`, enter your UW BIOST username and password when prompted (the user and pass you use to log in to Box)



Tera Term: New connection

(● TCP/IP) Host: Bayes

[✓] History

Service: ( ) Telnet (●) SSH ( ) Other

TCP port#: 22

SSH version: SSH2

Protocol: UNSPEC

( ) Serial Port: COM1: Communications Port (COM1)

Logging on to a cluster: Mac/Linux

1. Open a Terminal window
2. Type `ssh mynetid@cluster.washington.edu`
    - replace `mynetid` with Your UW NetID, and
    - replace `cluster` with Your cluster, e.g., `bayes`
3. Enter password (same password as Box login) when prompted — the field will remain blank but your password will be received

Using the clusters

Once logged in, navigation is always the same, since the clusters are all Unix based!

All files from your `P:/` drive are on every cluster

Now we can use all of those commands from slides 6–7. However, how we run “jobs” depends on the cluster:

- Cox: one job at a time (basically)
- Bayes: many jobs at once, scheduled by SGE

Using the clusters: Windows file compatability

Sometimes we can run into errors when running files on Unix machines. A place where I often go wrong is by saving files in Windows and then trying to run the file on Unix.

A common type of error comes from *control characters*, commonly seen as end of line characters in Windows. For the permutation test example:

```
[brianw26@cox ~/biost_561]$ ./perm_test_2.sh
./perm_test_2.sh: Command not found.
[brianw26@cox ~/biost_561]$ cat -vt ./perm_test_2.sh
#!/bin/sh^M
Rscript permutation_test_example_guiless.R myn=100 myseed=101 myB=10000 &^M
Rscript permutation_test_example_guiless.R myn=300 myseed=301 myB=10000 &^M
Rscript permutation_test_example_guiless.R myn=500 myseed=501 myB=10000^M
```

To run our script successfully, we need to remove these characters either by hand using `vim` or `emacs` to edit the file, or by running `dos2unix myfile.sh`.

Using the clusters: being nice

The department clusters are shared resources, here for our benefit. There are a few ways to be nice about how you use the clusters:

1. Bayes is a compute cluster — one head node (schedules jobs) hooked up to compute nodes (run the jobs). Bayes is optimized to run jobs on the compute nodes, not the head node. Do not run memory-intensive or long jobs on the head node!

2. Submitting jobs on Bayes automatically "nices" for you — SGE lowers your priority as your number of currently running jobs increases, and increases priority again as jobs finish running

3. Running long/memory-intensive jobs on Cox slows down the computer for all users

Running a single job on Bayes

Write a shell script that calls your function(s). For example:

```
[brianw26@bayes0 ~/biost_561]$ more call_ex1.sh
#!/bin/sh

Rscript se_ex1.R > ex1_output.txt B=10000 seed=547 n=500 truebeta=2
```

- The first line tells Bayes to run using the Bourne shell (sh). You can change this to another shell if you want
- The script calls Rscript to run the file `se_ex1.R` and pipes any output to `ex1.output.txt`
- The values `B=10000` etc. are command line arguments

To submit the job:

```
[brianw26@bayes0 ~/biost_561]$ qsub -cwd call_ex1.sh
Your job 3875039 ("call_ex1.sh") has been submitted
```

Submitting jobs on Bayes

There are many options for `qsub`. A few examples:

- `-cwd` executes the script in the current working directory
- `-e` and `-o` can be used to send error and output files (from `qsub`, not R) to a folder, e.g., `iotrash/`
- `-N` names a job
- `-t <tasklist>` submits a job array

See the manual for more detailed information.

Submitting jobs on Bayes: options

You can also set defaults in Your home directory (e.g., `P:/`) in a file called `.sge_request`. These also come from the manual. Mine is

```
-j y
-cwd
-S /bin/bash
-q normal.q
-M brianw26@uw.edu
-m e
```

- `-j y` says that I can submit a binary or script file
- `-S` tells SGE the default program to run my script
- `-q` tells SGE which queue to run my script on
- `-M` tells SGE who to email (me)
- `-m` tells SGE when to email (`e` is end of job)

Checking/altering jobs on Bayes

Once you have submitted a job, there are many options for checking/altering it:

- `qstat` checks queue status
  - `-f` shows full status of jobs
  - `-u <user>` shows status of jobs for user `user`
  - `-j <job_id>` gives details about a single job
- `qhold <job_id>[.tasklist]` puts a hold on job `job_id` (and optionally array elements in `tasklist` — more on this later)
- `qrls <job_id>` removes a hold on a job
- `qdel <job_id>` deletes a job

Submitting multiple jobs to Bayes

An alternative to submitting multiple jobs by hand, or parallelizing code directly, is to submit a batch of individual jobs to Bayes. Advantages:

- This is what Bayes is set up for
- It is a simple way to run a set of simulations using varying parameters (I do this all the time)

I recommend including seed number and other parameter values in the filename used to save output. This makes it easy to loop over saved output/objects to combine output later.

Submitting multiple jobs to Bayes: loops

Create a submission script that loops over arguments:

```
[brianw26@bayes0 ~/biost_561]$ more submit_ex1_batch.sh
#!/bin/sh

B=5000
seed=47

for n in 100 300 500
do
for truebeta in {1..5}
do
    qsub -cwd -e iotrash/ -o iotrash/ -q normal.q@b01.local ./call_ex1_once.sh
$B $seed $n $truebeta
done
done
```

NB: You'll need to create a folder in your `cwd` called `iotrash`, and make `call_ex1_once.sh` executable for this to work.

Submitting multiple jobs to Bayes: loops

Write your script to accept command line parameters:

```
[brianw26@bayes0 ~/biost_561]$ more call_ex1_once.sh
#!/bin/sh

Rscript se_ex1.R > ex1_output_b$1_s$2_n$3_beta$4.txt B=$1 seed=$2 n=$3
truebeta=$4
```

The values of $1, $2, . . . , $n are the 1st, 2nd, . . . , nth command line arguments.

In this example, four arguments are passed to the `se_ex1.R` script and they are included in the file name.

Submitting multiple jobs to Bayes: job arrays

An alternative to using a shell script loop is to submit your job as a job array. Each job array task has a unique ID, which you can read in to set up a simulation.

To do this successfully, your R script has to contain a matrix with all combinations of simulation parameters that you want to run over. Here's a chunk from `se_ex2.R`:

```
#!/usr/local/bin/Rscript

doOne <- function(n, beta) {
<Code omitted -- see previous slide>
}

## get the array task id
job.id <- as.numeric(Sys.getenv("SGE_TASK_ID"))

## set up simulation parameters
ns <- c(100, 300, 500)
B <- 5000
truebetas <- seq(1, 5, 1)
```

More from `se_ex2.R`

```
param.grid <- expand.grid(ns, truebetas)
param.grid$B <- B
param.grid$seed <- param.grid[, 1] + param.grid[, 2]
names(param.grid) <- c("n", "truebeta", "B", "seed")

current <- param.grid[job.id, ]
library(sandwich)
set.seed(current$seed)
system.time(output <- replicate(B, doOne(n = current$n,
                                         beta = current$truebeta)))
save(output, file = paste("ex2_output_b_", current$B, "_s_",
                          current$seed, "_n_", current$n,
                          "_beta_", current$truebeta, sep = ""))
```

Submitting multiple jobs to Bayes: job arrays

Once your R script can handle submission as a job array, set up a script to call it once and then a script to submit it as a job array, e.g.,

```
[brianw26@bayes0 ~/biost_561]$ more call_ex2_once.sh
#!/bin/sh

Rscript se_ex2.R
```

```
[brianw26@bayes0 ~/biost_561]$ more submit_ex2_batch.sh
#!/bin/sh

qsub -t 1-15 -cwd -e iotrash/ -o iotrash/ -q normal.q -l "h=b01|b02|b03|b04"
-shell no -b yes ./call_ex2_once.sh
```

Submitting multiple jobs to Bayes: being nice

Remember that Bayes is a shared resource. If you find the cluster empty, and fill up all of the nodes with long jobs, remember to **monitor** these jobs and the queue and be prepared to suspend some of those jobs as other users' jobs enter the queue. A good rule of thumb is to only have 20 jobs running at a single time.

Ways to help share the cluster:

- Submit batch jobs in job arrays — this makes the queue easier for everyone to read
- Use manual holds or link automatic holds on jobs, using the `-tc` argument in `qsub`
- Reduce priority of jobs if necessary
- Estimate how long each job will take, and take this into consideration when submitting large batches