Introduction to Linux and Cluster Computing Environments for Bioinformatics

Doug Crabill
Senior Academic IT Specialist
Department of Statistics
Purdue University
dgc@purdue.edu
What you will learn

• Linux Supercomputer overview
• Basics of a Linux shell, including moving/editing/creating/deleting files, how to launch/terminate programs, check progress
• Basic shell scripting, parallel execution
• Fundamentals of cluster supercomputer use
• Example of scaling things up
The rice.rcac.purdue.edu cluster
The rice.rcac.purdue.edu cluster
An individual node
The brown.rcac.purdue.edu cluster
The brown.rcac.purdue.edu cluster
The brown.rcac.purdue.edu cluster
Brown supercomputer stats

- 550 Nodes, 13,200 total CPU cores
- Each node has 24 CPU cores, 96GB RAM
- 3.4 Petabytes of scratch space for this cluster alone
- 4.5 Petabytes of long term storage shared among all clusters
- Currently #302 on top500.org, Conte is #190.
Anecdote time!

• A colleague was working on a game theory problem…
Linux and Cluster Computing Environments

Your laptop

Front End

Front End

Front End

Front End
Linux and Cluster Computing Environments

Your laptop

Front End

Front End

Front End

Front End

Back-end nodes/compute nodes
Linux and Cluster Computing Environments

Your laptop

Front End

Front End

Front End

Front End

Scheduler

Back-end nodes/compute nodes
Why Linux?
Why Linux?
Why Linux?

- Can be desktops, but tend to be larger servers in some remote, environmentally controlled data center (or pod!)
- Multiple CPU cores per server (~8-44)
- Large amounts of RAM (64GB – 1TB is common)
- Multiple users can use the same computer simultaneously
Why Linux? (cont.)

- Can interact with a graphical interface
- More common to interact with a text based interface
- Servers tend to stay up for a long time between reboots (months)
- Commonly launch programs and walk away for days, weeks, or months as they run
- Computations can scale up as servers added
But where are the keyboards, mice, and monitors?
But where are the keyboards, mice, and monitors?
ThinLinc Linux graphical interface

- We will use ThinLinc to provide a graphical user interface on a Brown front-end
- From the front-end we’ll connect to a Brown node, aka back-end node, aka compute node, where we will do the real computing
- The ThinLinc client is free (and better), but you can actually use a web browser instead
Logging in via ThinLinc Client
Connected!!!
Toggle full screen on ThinLinc client by pressing the F8 key.
ThinLinc sessions can persist!

- Programs/windows that are open and running can persist after closing the ThinLinc Client
- Smile patiently while I demonstrate persistence
- If you explicitly click Applications->Log Out you will be logged completely out and application state will not persist
What is a “shell”? 

- A text-based user interface used to launch programs. The shell we use is called “bash”
- Used to launch programs, pass arguments to programs, specify input/output files
- Terminal is one way of accessing a shell
- Launch via Applications->Terminal Emulator or Applications->System->Xfce Terminal (my preferred method)
A Terminal
Multiple Terminal windows

• You can have many Terminal windows open at once
• To open an additional Terminal window on the same server as an existing Terminal, type:
  `xfce4-terminal &`
• If you omit the `&`, the first Terminal cannot be used again until the second is closed
• Type `exit` to log out of a shell
Using copy/paste

• Using the Windows shortcuts Control-C and Control-V will generally not work, because those keys mean other things under Linux.
• Either select the text and select Edit/Copy and then Edit/Paste.
• Or select the text which implicitly copies it, and press down on the mouse wheel to paste (don’t roll it, press down like it’s a button).
Filesystems

- Filesystems on Linux similar to network drives on Windows, but without drive letters
- Example directories on different filesystems: `/home/dgc`, `/depot/nihomics`, `/scratch/brown/dgc`
### Filesystems on Brown

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>Size</th>
<th>Used</th>
<th>Avail</th>
<th>Use%</th>
<th>Mounted on</th>
</tr>
</thead>
<tbody>
<tr>
<td>rootfs</td>
<td>47G</td>
<td>10G</td>
<td>37G</td>
<td>22%</td>
<td>/</td>
</tr>
<tr>
<td>devtmpfs</td>
<td>47G</td>
<td>0</td>
<td>47G</td>
<td>0%</td>
<td>/dev</td>
</tr>
<tr>
<td>/dev/sda2</td>
<td>377G</td>
<td>152M</td>
<td>356G</td>
<td>0%</td>
<td>/tmp</td>
</tr>
<tr>
<td>depotint-nfs.rcac.purdue.edu:/depot</td>
<td>4.5P</td>
<td>3.0P</td>
<td>1.5P</td>
<td>67%</td>
<td>/depot</td>
</tr>
<tr>
<td>persistent-nfs.rcac.purdue.edu:/persistent/apps</td>
<td>8.0T</td>
<td>4.5T</td>
<td>3.6T</td>
<td>56%</td>
<td>/apps</td>
</tr>
<tr>
<td>persistent-nfs.rcac.purdue.edu:/persistent/home</td>
<td>80T</td>
<td>71T</td>
<td>9.8T</td>
<td>88%</td>
<td>/home</td>
</tr>
<tr>
<td>172.18.87.9@tcp:172.18.87.10@tcp:/ LustreF</td>
<td>3.4P</td>
<td>374T</td>
<td>3.0P</td>
<td>11%</td>
<td>/scratch/</td>
</tr>
<tr>
<td>brown</td>
<td>172.18.84.184:/persistent/fsadmin</td>
<td>200G</td>
<td>176G</td>
<td>25G</td>
<td>/usr/rmt_</td>
</tr>
<tr>
<td>share/fsadmin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brown-fe01 ~ $</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shell features

- Shell environment variables used to control settings for how certain things work
- Thousands of potential commands can be executed
- Commands available varies from one Linux computer to the next, depending on what has been installed, and the value of your PATH environment variable
Shell features (cont.)

- Filename completion (using “Tab” key)
- Command completion (using “Tab” key)
- Command line editing using arrow keys (up-arrow key to go to the previous command)
Let’s get dirty!
Listing files in Terminal

- Type `ls` to list files in the current directory
- Type `ls -l` to list files with more detail
- Type `ll` to list files with even more detail
Navigating directories in Terminal

- Type `pwd` to see full path to current directory
- Type `cd dirname` to change directories
- Type `cd ..` to go to the parent directory, or `cd ../..` to go to the grandparent, etc.
- Type `cd ~` to go to your home directory
- `cd /depot/nihomcics/data`
- Absolute paths start with `/`, relative paths are relative to the current directory
Special directories

- `/home/USERNAME` – Your home directory, where source code, programs, and final results go
- `/scratch/brown/USERNAME` – Enormous scratch directory. Can place original data sets and intermediate results there
- Type `myquota` to see used disk space and limits
Editing, copying, moving files
Editing, copying, moving files

- `gedit filename` – Edits filename
- `mv oldname newname` – Moves a file or directory, possibly to a new directory, possibly renaming the file or directory in the process
- `cp oldname newname` – Copies files
- `cp -r olddir newdir` – Copies `olddir` and all files and subdirectories within to `newdir`
Create/Remove directories, files

- `rm filename` — removes `filename`
- `mkdir dirname` — creates `dirname`
- `rmdir dirname` — removes `dirname`, but only if `dirname` is empty

Let’s practice, and use filename completion and command line editing while we are at it!
Terminating a program

• If you are running a program in a terminal window that you would like to terminate, press Control-C
• This won’t work if you started that program it with an &
See what programs are running

- `ps xuww` - Show what programs we are running now
- PID column shows the Process ID of each program
- Can use `top` to see most CPU intensive programs currently running by everyone on this server. Press `q` or just control-c to exit `top`
Terminate or *kill* or program

- Must first know the process id number (PID) using either `ps xuww` or `top`
- `kill NNNNN` Will kill most programs
- `kill -HUP NNNNN` Use if the previous doesn’t work
- `kill -9 NNNNN` Use if the previous doesn’t work
Let’s practice starting/killing progs

• On a Brown node, type `busy 1000 &`
• Type it again a few times (use the up-arrow!)
• Type `top` to see the PIDs of all the jobs running, press q to quit
• Kill all of the busy jobs by typing the PIDs `like:` `kill` 24933 24937 24939 24944
• Type `top` again to confirm they are gone
Redirecting input/output

- Some programs write output to the Terminal/shell screen
- We can save it using output redirection
- `qstat -a > out1` Saves results of the command `qstat -a` to the file `out1`
- `head < out1` See the first 10 lines of `out1`
- `head < out1 > out2` Save to `out2`
Redirecting input/output

• Can only save the text output that would have normally appeared on the screen. If a program wouldn’t normally generate any text output, nothing will be saved.

• `Terminal > out3` *(Nothing is saved!)*
Interactive shell on back-end node

- So far we’ve been working only on a Brown front-end node. We really want a back-end.
- `qsub -I -X -l walltime=4:0:0,nodes=1:ppn=24 -q standby` (one long typed line)
- Now we have a whole single node to ourselves for interactive use – for 4 hours
Interactive shell on back-end node
Using qstat

```
brown-fe02 $ qstat -u dgc

Output of qstat command.
```
Using `qstat -a`

```plaintext
Job ID       Username     Queue   Jobname            SessID  NDS  TSK   Req'd Memory       Req'd Time   S   Elap Time
-----------  -----------  -------  ------------------  ------  ----  ----  ------------------  -----------  ---  --------
924357.brownadm.rcac.jsstjohn testbrow mpi-hello.sub   --   1    24    --  00:01:00 Q  --
1350132.brownadm.rcac.ashishod standby 318tr_c_f    --   30   720   --  04:00:00 H  --
1350133.brownadm.rcac.ashishod standby 318tr_c_f    --   30   720   --  04:00:00 H  --
1350134.brownadm.rcac.ashishod standby 318tr_c_f    --   30   720   --  04:00:00 H  --
1474944.brownadm.rcac.kvarala kvarala ML_Bootstrap.pbs   --   1    16   96gb  48:00:00 Q  --
1749316.brownadm.rcac.yoop standby pmma               --   2    48    --  04:00:00 H  --
1751758.brownadm.rcac.msakano pcpxtrem Cook30K 1.00   367845  1    8    --  336:00:00 R  332:22:57
1751783.brownadm.rcac.yoop lpl LMO_PS4_02_nebl       --   3    72    --  336:00:00 H  --
1751784.brownadm.rcac.yoop lpl LMO_Yo_02_nebl         --   3    72    --  336:00:00 H  --
1758059.brownadm.rcac.yoop standby Au_bulk            --   4    96    --  04:00:00 H  --
1790112.brownadm.rcac.charlesj ncm STIN              314956  1    24    --  336:00:00 R  311:26:37
1801274.brownadm.rcac.msakano pcpxtrem Cook27K 1.5    67841   1    8    --  336:00:00 R  255:51:40
1801275.brownadm.rcac.msakano pcpxtrem Cook30K 1.5    264185   1    8    --  336:00:00 R  255:51:03
1801574.brownadm.rcac.pkwise bendor b12f12           252481   1    24    --  300:00:00 R  251:57:50
1813977.brownadm.rcac.islan3 pcpxtrem s10s35         191000   1    8    --  330:00:00 R  254:48:53
1814479.brownadm.rcac.islan3 pcpxtrem s10s1k          437925   1    8    --  330:00:00 R  254:42:15
1814480.brownadm.rcac.islan3 pcpxtrem s10s1k          256784   1    8    --  330:00:00 R  254:42:09
1814481.brownadm.rcac.islan3 pcpxtrem s10s1k          110744   1    8    --  330:00:00 R  254:39:59
```
Using qlist

<table>
<thead>
<tr>
<th>Queue</th>
<th>Total</th>
<th>Queue</th>
<th>Run</th>
<th>Free</th>
<th>Max Walltime</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>96</td>
<td>0:30:00</td>
</tr>
<tr>
<td>standby</td>
<td>13,032</td>
<td>90,048</td>
<td>10,176</td>
<td>722</td>
<td>4:00:00</td>
</tr>
<tr>
<td>statdept</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>336:00:00</td>
</tr>
<tr>
<td>wwtung</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>336:00:00</td>
</tr>
</tbody>
</table>
Pipes
Pipes

- Pipes are a feature of the shell that permit the output of one program to be read as the input of another program, without having to save an intermediate file.

- All programs involved run simultaneously and process data as they receive it, similar to different sections of an assembly line.
Example without pipes

```bash
brown-fe01 ~ $ qstat -a > out1
brown-fe01 ~ $ awk '{print $2}' < out1 > out2
brown-fe01 ~ $ sort < out2 > out3
brown-fe01 ~ $ uniq < out3 > out4
brown-fe01 ~ $ wc -l < out4
48
brown-fe01 ~ $ []
```
Now using pipes

```
brown-fe01 ~ $ qstat -a | awk '{print $2}' | sort | uniq | wc -l
46
brown-fe01 ~ $ 
```
module command loads software

- module list
- module avail
- module spider anaconda
- module add r/3.4.3
- module load rstudio  # add and load synonymous
- module del rstudio
- Module changes affect only the current shell!
Use nohup for long running jobs

- If the shell in which you run a job dies, so do all of the programs launched from that shell.
- Use `nohup` to make your scripts immune from being terminated if the parent shell dies.
- Just prepend `nohup` to any command.
- `nohup busy 100`
Imagine Bob buys a bus…
How do we fill the seats (cpus)?

- We could launch several `xfce4-terminal &` and run programs in each window.
- If we run them with an `&` at the end, we could launch many programs from a single terminal (`rmarkdown &` then another `rmarkdown &` then another, and so forth).
That’s nice, but it doesn’t scale

• I still need to use my keyboard and mouse in each of the rstudio windows to launch programs
• It’s fine for small numbers of tasks, but we can do better!
What is a shell script?

- The things we would normally type at a shell/bash/Terminal prompt could be collected together into a single file, like `t1.sh`
- Then we execute that file by typing `sh t1.sh`
- To see each line as it’s executed, add the `-x` option to `sh`: `sh -x t1.sh`
But that’s still just one seat…
Make them run simultaneously

busy 10 &
busy 10 &
busy 10 &
busy 10 &
busy 10 &
Let’s time it!

• time busy 10

• Create a `t2.sh` with 24 lines of `busy 10 &` with the last line containing just the word `wait`

• time sh t2.sh
What if I run > 24 jobs?

- If the number of CPU intensive jobs currently running exceeds the number of CPU cores, the jobs will share the available cores, but with a loss of efficiency.
- Often best to keep the number of CPU intensive jobs less than or equal to the number of CPU cores (type `lscpu` to see number of CPU cores on a server).
Yeah, but I have these 1000 jobs...
Yeah, but I have these 1000 jobs...
Yeah, but I have these 1000 jobs...

- Could create a t1.sh with `24 busy 100 &` followed by a `wait`.
- Then create `t2.sh, t3.sh` similar to `t1.sh`.
- Then create `runall.sh` that executes `t1.sh, then t2.sh, then t3.sh`.
- Works, but if jobs take a variable amount of time it’s inefficient since we wait for the EVERY job in `t1.sh` to complete before starting `t2.sh` and so forth.
ParaFly to the rescue!

- ParaFly reads jobs to run from a file and keeps N of them running at once
- `module load utilities parafly`
- `ParaFly -vv -c joblist.txt -CPU 24`
- Creates `joblist.txt.completed`
- CRITICAL: the jobs given in joblist.txt must not end with “&” or they will ALL start at once!
ParaFly example

• Let’s create t3.sh with 100 random busy jobs (2 ways):
• R -e 'write(sprintf("busy %d", as.integer(runif(100, 1, 11))), "t3.sh")'
• perl -e 'foreach(1..100) {printf "busy %d\n",1+int(rand(10))}' > t3.sh
• time ParaFly -vv -c t3.sh -CPU 24
Go Bob go!
May use parallel instead of ParaFly

• ITaP/RCAC supports ParaFly, but it is less common on other Linux installations
• The GNU parallel command is a functional superset of ParaFly. ITaP does not have it installed on their clusters…
• Not related to R library called parallel
• parallel -t -j 24 < t3.sh
That’s nice, but I use rstudio

- **rstudio** is an Integrated Development Environment (IDE) that’s great for creating your R code
- When your code is mature and you are ready to run it over and over, run it OUTSIDE of rstudio using Rscript
Rscript

• Yes, it’s actually capitalized
• Scripts we have created in Rstudio can be run using Rscript at a shell prompt:

    Rscript t1.R

• Capture the output:

    Rscript t1.R > out

• Everywhere we had used busy we could have used Rscript something.R
Too bad it won’t work on Windows!

• Ummm, it sort of does, with some changes
• Rscript works, but have to use correct path, run from “cmd” prompt.
• Place multiple Rscript commands in a .bat file
• Can launch multiple rstudio instances at once and run different commands in them. Each will use a different CPU core
Brown supercomputer stats

- 550 Nodes, 13,200 total CPU cores
- Each node has 24 CPU cores, 96GB RAM
- 3.4 Petabytes of scratch space for this cluster alone
- 4.5 Petabytes of long term storage shared among all clusters
- [https://www.rcac.purdue.edu/knowledge/brown/all](https://www.rcac.purdue.edu/knowledge/brown/all) for full user’s guide on using this cluster
THAT makes Bob happy!
PBS scheduler commands

- `qstat -a` # See list of jobs in queue
- `qstat -u dgc` # See list of jobs in queue submitted by dgc
- `qsub jobname.sh` # Submit jobname.sh to run on the cluster
- `qdel JOBIDNUMBER` # delete a job from the queue
- `qstat -f JOBIDNUMBER` # See detailed info about this job
Simple qsub submission file

- Qsub accepts command line arguments, OR reads embedded comments

```bash
#!/bin/sh -l
#PBS -q standby
#PBS -l nodes=1:ppn=1
#PBS -l walltime=00:10:00

cd $PBS_O_WORKDIR
busy 10
hostname
```
Submitting the job

• Just type: `qsub scriptname.sh`
Viewing status and the results

- **Use** `qstat -u dgc` **to check status**
- **Output of job 1888656 for myjob.sh goes to** `myjob.sh.o1888656`, **errors from job 1888656 goes to** `myjob.sh.e1888656`
- **Inconvenient to collect the results from a dynamically named file like** `myjob.sh.o1888656`. **Best to write output to a filename of your choosing in your R program**
Our results

```
brown-fe01 ~/biotalk/R1 $ ll
 total 136
  0 drwxr-xr-x 2 dgc stat  4096 Sep 9 17:45 ./
  72 drwxr-xr-x 4 dgc stat 36864 Sep 9 17:28 ../
  64 -rwxr-xr-x 1 dgc stat  9056 Sep 9 17:41 busy*
  0 -rwx------- 1 dgc stat  115 Sep 9 17:43 myjob.sh
  0 -rwx------- 1 dgc stat  115 Sep 9 17:43 myjob.sh.e1888656
  0 -rw------- 1 dgc stat  115 Sep 9 17:45 myjob.sh.o1888656
brown-fe01 ~/biotalk/R1 $ cat myjob.sh.o1888656
brown-a672.rcac.purdue.edu
brown-fe01 ~/biotalk/R1 $ 
```
Our first R job submission

• The R program t2.R:

```r
summary(1 + rgeom(10^7, 1/1000))
```

• R2.sh:

```bash
#!/bin/sh -l
#PBS -q standby
#PBS -l nodes=1:ppn=1
#PBS -l walltime=00:30:00

cd $PBS_O_WORKDIR
module add r
Rscript t2.R > out2
```

• Submit using `qsub R2.sh`
But that’s still just one seat…
Scaling it up…

- Figure out how to break large jobs into many smaller jobs that finish in under four hours to use standby queue
- Submit all of the smaller jobs at once
- Keep the seats full! (Keep all CPU’s busy)
- Check status, collect results when complete
- Profit!
We don’t like to share nodes!

- We don’t want to share a node with other jobs, unless they’re OUR other jobs:

```bash
#!/bin/sh -l
#PBS -q standby
#PBS -l nodes=1:ppn=1
#PBS -l walltime=00:10:00
#PBS -l naccesspolicy=singleuser

cd $PBS_O_WORKDIR
module add r
Rscript t2.R > out2
```
Coupon collector problem

• This would take one week on a single CPU core:
  \[
  \text{sum(sapply(1:10000, function(y) \{mean(1 + rgeom(6.3 * 10^8, y/10000))\}))}
  \]

• One obvious approach is to break it into 10,000 smaller R jobs and submit all 10,000 to the cluster

• But to minimize overhead, better to break it into 200 jobs, each operating on 50 numbers (~60 mins for each of the 200 jobs)

• Create an R script that accepts command line arguments to process many numbers at a time. Estimate walltime carefully!
Coupon collector R code

- t3.R, read arguments into “args”:
  ```R
  args <- commandArgs(TRUE)
  cat(sapply(as.integer(args), function(y) {mean(1 + rgeom(6.3 * 10^8, y/10000))}))
  ```
- R3.sh contains “template” of what we will qsub:
  ```bash
 #!/bin/sh -l
  #PBS -q standby
  #PBS -l nodes=1:ppn=1
  #PBS -l walltime=01:30:00
  #PBS -l naccesspolicy=singleuser

  cd $PBS_O_WORKDIR
  module add r
  ```
Coupon collector R code

• Generate 200 scripts with 50 arguments each using first part of R3.sh, then appending the Rscript command with arguments:

```r
s<-scan("R3.sh", what='c', sep="\n")
sapply(1:200, function(y) {
  s[8]=sprintf("Rscript t3.R %s > out.%03d", paste((y*50-49):(y*50), collapse=" "), y);
  write(s, sprintf("prog%03d.sh", y));
})
write(sprintf("qsub prog%03d.sh", 1:200), "runall.sh")
```
Run coupon collector code

• `sh -x runall.sh`
• **Results are in** `out.001, out.002, out.003, ...`
• Collect them into a single file by typing:
  • `cat out* > final.txt`
Could have used whole nodes instead

- Instead of 200 individual `qsub` jobs with 50 numbers each, we could have submitted 9 `qsub` jobs, each with 24 `Rscript` ... & commands, each processing 50 numbers each, with a `wait` at the bottom

- If we wanted to use a smaller number of nodes (like 4), we could have used ParaFly and processed 50 `Rscript` jobs per node (omit the trailing & and the `wait`!)
My advice

• Realize naive mistakes are common and are easy to make. Expect them!

• Preprocessing your data first before bringing it into R can save time. Use “awk” or “perl”

• Thinking through how to break down a large problem can be critical to your success

• Don’t minimize the time for your first job to complete, but for your last job to complete