Introduction to Phoenix Scheduler

Steve Pederson
Stephen Bent
Dan Kortschak
Ramona Rogers
Robert Qiao
Bowen Chen
Exequiel Sepulveda

University of Adelaide 26th September, 2017

Licensing

Contents

Licensing **3**

Contents **4**

Workshop Information **5**
- The Trainers . . . 6
- Computer Setup . . . 7

Writing Scripts **9**
- Shell Scripts . . . 10
- Moving towards High Performance Computing . . . 15

Using Modules **17**
- Basic commands of modules . . . 18
- Loading modules . . . 20
- Using modules . . . 20
- A complete exercise of modules . . . 21

Writing Basic Slurm Scripts **23**
- Basic template of a Slurm script . . . 24
- Submitting Scripts . . . 25
- Managing Scripts . . . 26

Space for Personal Notes or Feedback **29**

Workshop Information

The Trainers

Ms Ramona Rogers
Computing Officer
IT Support
The University of Adelaide
South Australia
ramona.rogers@adelaide.edu.au

Mr Robert Qiao
Phoenix Support Team
The University of Adelaide
South Australia
robert.qiao@adelaide.edu.au

Mr Bowen Chen
Phoenix Support Team
The University of Adelaide
South Australia
bowen.chen@adelaide.edu.au

Mr Exequiel Sepúlveda
Phoenix Support Team
The University of Adelaide
South Australia
exequiel.sepulvedaescobedo@adelaide.edu.au

Computer Setup

We will all be working on `Phoenix` directly, which is the University of Adelaide's High Performance Computing (HPC) system. The software client `Bitvise` or `putty` which you will have already installed, enables us to access these machines in a familiar Desktop style, even though the majority of our time will be spent within the terminal.

In case you need to install Bitvise, download the installer from here: Bitvise SSH Client

You will need to open a SSH session to `phoenix`. First, we need to create a session with the basic parameters

1. Hostname phoenix.adelaide.edu.au
2. Username your student or staff ID

Now we have created the session, you will be asked for your password.

Now that you are connected, you will notice we are now in the head node of `Phoenix`. Welcome to `Phoenix`.

Writing Scripts

Primary Author(s):
Stephen Bent, Robinson Research Institute, University of Adelaide
Steve Pederson, Bioinformatics Hub, University of Adelaide
stephen.pederson@adelaide.edu.au
Robert Qiao, Research Services Phoenix Team, University of Adelaide
robert.qiao@adelaide.edu.au

Contributor(s):
Dan Kortschak, Adelson Research Group, University of Adelaide
dan.kortschak@adelaide.edu.au
Jimmy Breen, Robinson Research Institute & Bioinformatics Hub, University of Adelaide
jimmy.breen@adelaide.edu.au

We often need to perform repetitive tasks, or need to perform complex series of procedures. Rather than typing instruction receptively and stearing the screen waiting for each process to finish before providing next instruction, writing the set of instructions into a script and interpreter/compiler does the waiting and stearing for us is a very powerful way of liberating our time. They are also an excellent way of ensuring the commands you have used in your research are retained for future reference. Keeping copies of all electronic processes to ensure reproducibility is a very important component of any research. Writing scripts requires an understanding of several key concepts which form the foundation of much computer programming, so let's walk our way through a few of them.

Shell Scripts

Now that we've been through just some of the concepts & tools we can use when writing scripts, it's time to tackle one of our own where we can bring it all together.
Every bash script begins with what is known as a *shebang*, which we would commonly recognise as a hash sign followed by an exclamation mark, i.e `#!`. This is immediately followed by `/bin/bash`, which tells the interpreter to run the command `bash` in the directory `/bin`. (This is actually where the program `bash` lives on a Linux system.) This opening sequence is vital & tells the computer how to respond to all of the following commands. As a string this looks like:

```
#!/bin/bash
```

The hash symbol generally functions as a comment character in scripts. Sometimes we can include lines in a script to remind ourselves what we're trying to do, and we can preface these with the hash to ensure the interpreter doesn't try to run them. It's presence as a comment here, followed by the exclamation mark, is specifically looked for by the interpreter but beyond this specific occurrence, comment lines are generally ignored by scripts & programs.

Some Example Scripts

Let's now look at some simple scripts. These are really just examples of some useful things you can do & may not really be the best scripts from a technical perspective. Hopefully they give you some pointers so you can get going

To setup this part of the course, please login to your Phoenix account and download the example file to your /fastdir directory now

```
cp -r /apps/examples/training_linux/ ~/fastdir/
```

please check to make sure you command blow contains valid 4 files. If you see error messages, please indicate to tutors, you should get this step fixed before carry on further

```
ls -al ~/fastdir/training_linux
```

A Simple Example to Start

Don't try to enter these commands directly in the terminal!!! They are designed to be placed in a script which we will do after we've inspected the contents of the script. First, just have a look through the script & make sure you understand what the script is doing.

Also remember that any long lines of code may be automatically broken into new lines on your page by the '\' character. We don't want to enter this character when we create our script. Note that the line numbers on the left of the code don't change when this happens, e.g. line 9.

Before we go any further, have a look at the following script.

```
#!/bin/bash
#
# First we'll declare some variables with some text strings
ME='Put your name here'
MESSAGE='This is your first script'

# Now well place these variables into a command to get some output
echo -e "Hello ${ME}\n${MESSAGE}\nWell Done!"
```

Firstly, you may notice some lines that begin with the # character. These are *comments* which have no impact on the execution of the script, but are written so you can understand what you were thinking when you wrote it. If you look at your code 6 months from now, there is a very strong chance that you won't recall exactly what you were thinking, so these comments can be a good place just to explain something to the future version of yourself. There is a school of thought which says that you write code primarily for humans to read, not for the computer to understand.

In the above script, there are two variables. Although we have initially set them to be one value, they are still variables. What are their names?

First we'll create an empty file which will become our script. We'll give it the suffix `.sh` as that is the common convention for bash scripts.

```
cd ~/firstname
touch wellDone.sh
```

Now using the text editor *nano*, enter the above code into this file *setting your actual name as the ME variable*, and save it by using `Ctrl+o`, which is indicated as `^O` in the nano screen.

```
nano wellDone.sh
```

Once you're finished, you can exit the `nano` editor by hitting `Ctrl+x`.

Another coding style which can be helpful is the enclosing of each variable name in curly braces every time the value is called. Whilst not being strictly required, this can make it easy for you to follow in the future when you're looking back. Variables have also been names using strictly upper-case letters. This is another optional coding style, but can also make things clear for you as you look back through your work. Most command line tools use strictly lower-case names, so this is another reason the upper-case variable names can be helpful.

Unfortunately, this script cannot be executed yet but we can easily enable execution of the code inside the script. If you recall the flags from earlier which denoted the read/write/execute permissions of a file, all we need to do is set the execute permission for this file. First we'll look at the files in the folder using `ls -l` and note these triplets should be `rw-` for the user & the group you belong to. To make this script executable, enter the following in your terminal.

```
cd ~/firstname
chmod +x wellDone.sh
ls -l
```

Notice that the third flag in the triplet has now become an `x`. This indicates that we can now execute the file in the terminal. As a security measure, Linux doesn't allow you to execute a script from within the same directory so to execute it enter the following:

```
./wellDone.sh
```

Making a Small Change

Now let's change the variable `ME` in the script to read as

```
ME=$1
```

and save this as `wellDone2.sh`. (You may like to create this first using `cp`) You'll now need to set the execute permission again.

```
chmod +x wellDone2.sh
```

This time we have set the script to *receive input from stdin* (i.e. the terminal), and we will need to supply a value, which will then be placed in the variable `ME`. Choose whichever random name you want and enter the following

```
./wellDone2.sh Boris
```

As you can imagine, this style of scripting can be useful for iterating over multiple objects. A trivial example, which builds on a now familiar concept would be to try the following.

```
for n in Boris Fred; do (./wellDone2.sh $n); done
```

A more complicated script

Here's a more complicated script with some more formal procedures. This is a script which will extract only the ProbeFeature features from the .gff file we have been working with, and export them to a separate file. Look through each line carefully & write down your understanding of what each line is asking the program to do.

```
#!/bin/bash
# Declare some helpful variables
FILEDIR=~/fastdir/training_linux/scripting
FILENAME=NC_015214.gff
OUTFILE=NC_015214_CDS.txt
# Make sure the directory exists
if [ -d ${FILEDIR} ]; then
  echo Changing to ${FILEDIR}
  cd ${FILEDIR}
else
  echo Cannot find directory ${FILEDIR}
  exit 1
fi

# If the file exists, extract the important ProbeFeature data
if [ -a ${FILENAME} ]; then
  echo Extracting ProbeFeature data from ${FILEDIR}/${FILENAME}
  echo "SeqID Source Start Stop Strand Tags" > ${OUTFILE}
  awk '{if (($3=="ProbeFeature")) print $1, $2, $4, $5, $7, $9}' \
      ${FILENAME} >> \
  ${OUTFILE}
else
  echo Cannot find ${FILENAME}
  exit 1
fi
```

Notice that this time we didn't require a file to be given to the script. We defined it within the script, as we did for the output file.

The directory & file checking stages were of the form if [...]. This is a curious command that checks for the presence of something. The options -d & -a specify a directory or file respectively.

Will the above script generate a tab, comma or space delimited text file?

Open the `Nano` text editor & save the blank file in your directory as *extract_CDS.sh*. Now write this above script into the editor, but *taking care to use the directory where you have the .gff file stored in the appropriate place.* Once you have written the script, save it & close it. Now make it executable and run it.

Moving towards High Performance Computing

High Performance Computing

In current genomics era, where we regularly work with large datasets, the amount of resources available on desktop computers are often insufficient to enable your script finish in a reasonable time. For these datasets, it maybe useful to work on a high performance computing system, which will enable your data or command to be run simultaneously on > 8 threads, i.e. in parallel. It is possible to gain access to large computing resources through the University's Phoenix HPC https://www.phoenix.adelaide.edu.au, enabling analysis of large datasets efficiently.

Having many users on one machine at one time also means that there needs to be a system which determines who runs what and when. Phoenix uses a schedular "SLURM", whereby users submit jobs to a queue, and then executed when the appropriate resources on the machine become available.

A typical slurm job script contains extra parameters such as:

- `-n`: The number of threads to allow
- `--time`: The maximum time it is allowed to takes to completion
- `--mem`: The amount of memory to allocate to the job

```
#!/bin/bash
#SBATCH -p batch
```

```
#SBATCH -N 1
#SBATCH -n 8
#SBATCH --time=20:00:00
#SBATCH --mem=20GB

# Execution code going below
<execution code>
```

We don't need to write this script. It is included here as a simple example of a real world script as used by Phoenix users. This script may look a little intimidating at first, but slowly work your way through each line & try to understand what each line is specifying.

More detailed info will be included in the last chapter of this course.

Using Modules

Primary Author(s):
Exequiel Sepulveda, Research Services, University of Adelaide
exequiel.sepulvedaescobedo@adelaide.edu.au

Contributor(s):
Robert Qiao, Research Services, University of Adelaide robert.qiao@adelaide.edu.au

There are many applications that need to change or define configurations to work properly. For example, they need to add the folder where binaries are located to the PATH environmental variable. If an application has many binaries located in /apps/MYAPP/bin, setting the PATH variable can be done by:

```
export PATH=$PATH:/apps/MYAPP/bin
```

Doing manually these modifications may be very tedious. The situation worsen if there are many applications that need to do the same with many potential conflicts and side effects. To keep environmental variables under control, Phoenix has an program that manages applications in a safe way. This program is called ”modules”

Basic commands of modules

Phoenix has installed hundreds of applications for users to use. By default, all applications are located in /apps/software and each one has a module configuration located in /apps/modules/all. Modules program has an unique executable named module.
Try to get the help from module program:

```
  module --help
```

The relevant options of module command are:

Option	**Description of function**	**Useful options**
`avail [name]`	Display the list of modules	-r for using regular expressions, -d for listing only the default version of each module
`spider [name]`	Explore and list modules	-r for using regular expressions, -d for listing only the default version of each module
`load name`	Load the specific module name	
`unload name`	Unload the specific module name	
`list`	List all loaded modules	
`show name`	Show the information of the module name	
`purge`	Unload all loaded modules to have a fresh starts	

The `avail/spider` option should be always used with a text to serach for, because listing all modules installed may be very slow and useless. The correct way to use avail/spider is using a text after, for example, to search Perl modules:

```
module avail Perl
module spider Perl
```

Spider and Avail are similiar but the output format is different. As an exercise, try to find all matlab modules using spider and avail options.

```
module spider matlab
module avail matlab
```

Repeat the same exercise to search for Python modules installed on Phoenix.

In general, most of modules follow this pattern: MODULE/VERSION[-TOOLCHAIN]. For example, for the module Python/3.6.1-foss-2016b: MODULE=Python, VERSION=3.6.1 and TOOLCHAIN=foss-2016b. Which toolchain should be selected is critical to understand.

Toolchain is a set of compiler, utilities and libraries (all they are modules!). The general rule is to use the newest official toolchain, which is foss-2016b. On Phoenix there are two main toolchains, foss and intel. The foss toolchain includes GNU compilers, whereas intel includes Intel compilers. You should use modules with the same toolchain. If you need to use modules with a different toolchain, a best practice is to use the purge option before changing toolchain.

Loading modules

Once you have found the module you want to load, the next step is loading it.
For example, search for R modules and load the newest one:

```
module spider R

    Versions
R/3.2.1-foss-2015b
R/3.3.0-foss-2016uofa
R/3.4.0-foss-2016b
```

```
module load R/3.4.0-foss-2016b
```

To see what happens after the module is loaded, use the option list:

```
module list
```

Why are there many loaded modules? It is because R/3.4.0-foss-2016b module has many prerequisites that are automatically loaded as well.
In most the cases, module configurations include automatically all modules needed. In few cases, because a precise control is necessary, this should be manually done.

Using modules

Once you have loaded modules, the current session will be correctly configured to use those modules. For example, you could load any Python module and check if the version is the correct one:

```
 module load Python

 python --version
```

Be careful with modules containing programs that are part of the operating system. For example, C/C++ compiler, Python, Perl, among others. If you don't load the right module, you may end using the wrong version.

What is the Python version included in the operating system?

A complete exercise of modules

In this section you are asked to complete a full exercise to reach a master level of modules. Run the traditional ”Hello World!” program in four different languages and versions.

Python 2.X:

```
print "Hello World, from Python 2.x"
```

Python 3.X:

```
print("Hello World, from Python 3.x")
```

R:

```
print("Hello World, from R")
```

Matlab:

```
display("Hello World, from matlab");
```

And these are the examples of how to run programs on those languages:
R:

```
echo 'COMMAND' | R --no-save
```

Matlab:

```
matlab -noFigureWindows -nodisplay -r 'COMMAND; exit;'
```

Python:

```
python -c COMMAND
```

The exercise is to find the modules to run each program and execute them. Replace COMMAND accordingly to each language.

Writing Basic Slurm Scripts

Primary Author(s):
Exequiel Sepulveda, Research Services, University of Adelaide
exequiel.sepulvedaescobedo@adelaide.edu.au

Contributor(s):
Robert Quiao, Research Services, University of Adelaide
robert.quiao@adelaide.edu.au

Phoenix is a shared HPC facility, therefore, all its resources are not for exclusive use of a particular user. The Phoenix architecture includes a head node and many computing nodes. The head node is where you are placed, after connect to Phoenix and is not designed for real computing.
The computing nodes are only available to our scheduler Slurm, which receives many scripts from all users and decides when and where to run those scripts according to the resources availables and account priorities. When Slurm receives a now job (or script) to run, he evaluates when is the sooner available time to run the job and assigns one or many computing nodes. The job will be ran by computing nodes, but not by the head node.

Slurm accounting, resources and priorities are complex enough. We have a workshop dedicated to this.

Therefore, to use the real power of Phoenix you need to write a slurm script. The script template is very simple: it is a normal script as you have written many, but with several Slurm meta-parameters at the beginning of the script.

Basic template of a Slurm script

A useful template for any slurm job is like this:

```
#!/bin/bash
#SBATCH -p batch
#SBATCH -N 1
#SBATCH -n 1
#SBATCH --time=00:05:00
#SBATCH --mem=1GB

#If you want to get feedback by email
#SBATCH --mail-type=ALL
#SBATCH --mail-user=firstname.lastname@adelaide.edu.au

#LOAD HERE ALL MODULES YOU NEED
#module load MODULE1
#module load MODULE2
#module load MODULE3

#EXECUTE HERE WHATEVER YOU WANT
#hostname
```

Let's have a look to each Slurm meta-parameter:

`#SBATCH -p batch` indicates the queue where the job will be submited to.

Phoenix has several queues, for example batch, test and bigmem. The batch queue is the default one for almost all kind of jobs. The queue test is useful to test jobs since the waiting times are reduced but also resources are limited just for testing.

`#SBATCH -N 1` indicates the number of computing nodes needed to run the job. Usually should be set to 1. `#SBATCH -n 1` indicates the number of cores needed to run the job. Usually should be set to 1.

Most of programs are serial and do not have parallel capabilities. If you do not know the capability of a program, assume the program is serial and, therefore, nodes and cores should be set to 1. If the program can use multicores, nodes should be set to 1 and cores to the desired number of cores, but less or equals to 32, which is the current maximum. If the program can use Message Passing Interface (MPI) and multicores, nodes should be set to equals or greater than 1, and cores to the desired number of cores, but less or equals to nodes*32.

`#SBATCH --time=00:05:00` indicates the maximum walltime of the job. In this example the maximum time will be 5 minutes.
`#SBATCH --mem=1GB` indicates the maximum amount of memory. In this example the maximum memory will be 1 Gigabytes.

Both time and mem meta-parameters are hard limits parameters. This means, if the job spends more time or consumes more memory, the jos will be cancelled. There is a trade-off. You would use conservative values, but if you overestimate them, the scheduler will take longer to executed it.

Getting notifications by e-mail is optional, but it is very recommended to know the progress of a job.
`#SBATCH --mail-type=ALL` indicates that Slurm will notify any state change, such as Cancelation, Success and Failure. `#SBATCH --mail-user=firstname.lastname@adelaide.edu.au` indicates the email address to send those notifications.

Submitting Scripts

Once you have a script ready to submit, you need to ”submit” the script to the scheduler by the use of the command `sbatch`. Let’s assume there is a script *job_script.sh*. To submit it to the schedule, just use the following command:

```
$ sbatch jobs_script.sh
Submitted batch job 3853482
```

In return you get a job number or *JOBID*, 3853482 is the example above, which is very important to record. Also, a new file with name slurm-*JOBID*.out is generated with the output of the script.

Create and submit a script that shows the hostname. Hint: the command for asking the hostanme is `hostname`.

Managing Scripts

Finally, after a job submission, you would need to manage its execution and get the information of completition. There are four basic commands to manage jobs:

Command	**Description of function**	**Options**
`squeue`	Display the list of queued jobs	-u USER
`scancel`	Cancel the execution of JOBID	JOBID
`scontrol show job`	Show useful information of a queued or running job	JOBID
`rcstat`	Show useful informartion of a job, specially for a completed job	JOBID

Now, from the last exercise in the Modules chapter (the execution of Hello World example in four languages), the final exercise:

Adapt that script as a Slurm script and submit it to the scheduler.

You can monitor its execution.

The command `rcstat` is useful to get the job statistics back, such as the real time spent, cpu and memory used, among others. You should use this information to adjust those values for the next submissions in order to request the right resources as closer as possible to the real ones.

Space for Personal Notes or Feedback