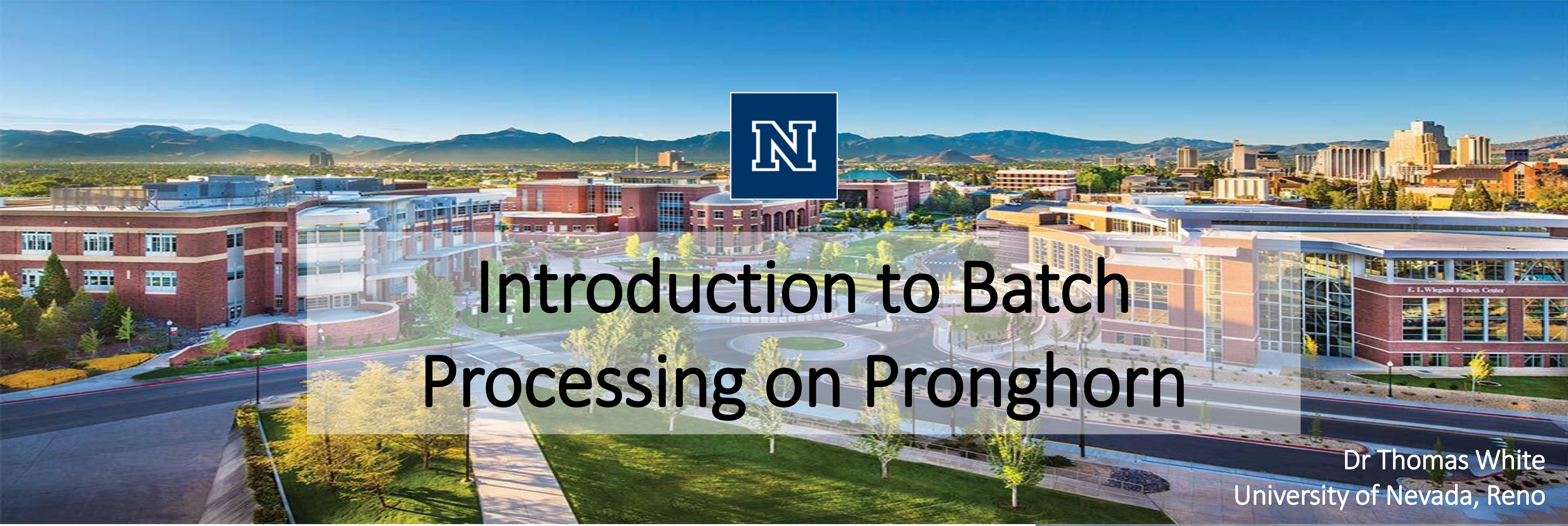


Introduction to Batch Processing on Pronghorn

Dr Thomas White
University of Nevada, Reno

Module 7: “Introduction to Bach Processing on Pronghorn”
October 23 2021
Lead: *Thomas White (Physics)*

This workshop will introduce the fundamentals of high-performance and parallel computing. I will introduce basic parallel computing concepts while students learn how to access and utilize Pronghorn, the University's high-performance supercomputer. During class, you will obtain hands-on experience running parallelized calculations through the SLURM job submission system.

Pre-reqs: Module 3 (Introduction to Linux).

Guide to this module (how to pass!)

Slides with exercises for you are marked in the upper right with an orange box.

Exercise #: Do Something

Commands you may need to enter into the terminal are highlighted in red and in the ‘consolas’ font as follows:

- `$pwd`
- `$cd`
- `$ssh <netid>@pronghorn.rc.unr.edu`

Guide to this module (how to pass!)

At the end of this module, I will ask you to run a parallelized python script on the pronghorn high performance computer which will generate a number. You will need to enter this number into the webcampus quiz in order to pass. This number will be unique to you, so make sure you are following along. **If you get stuck, please ask!**

Please use the post-it notes to get the attention of those in the room if you get stuck. Don't fall behind!

Help Available in the Room

- Gunner Stone (TA)
- Akshay Krishna (TA)
- John Anderson (HPC Systems Administrator)

Logging In

SSH or Secure Shell is a network communication protocol that enables two computers to communicate.

Please ssh to the appropriate login node below, substituting YOUR netid for "**`[netid]`**" (no brackets)!

Feel free to use MobaXterm (win), Putty (win), or the Terminal (mac/linux).

```
$ssh <netid>@pronghorn.rc.unr.edu
```

You will actually want to login TWICE. Consider ONE of the logins to be your "monitoring" session, and the other where you'll do the bulk of the work. Make sure both windows are visible. With MobaXTerm you can drag the "tabs" off and arrange them as needed. On a Mac with Terminal, Command-N will make a new window.

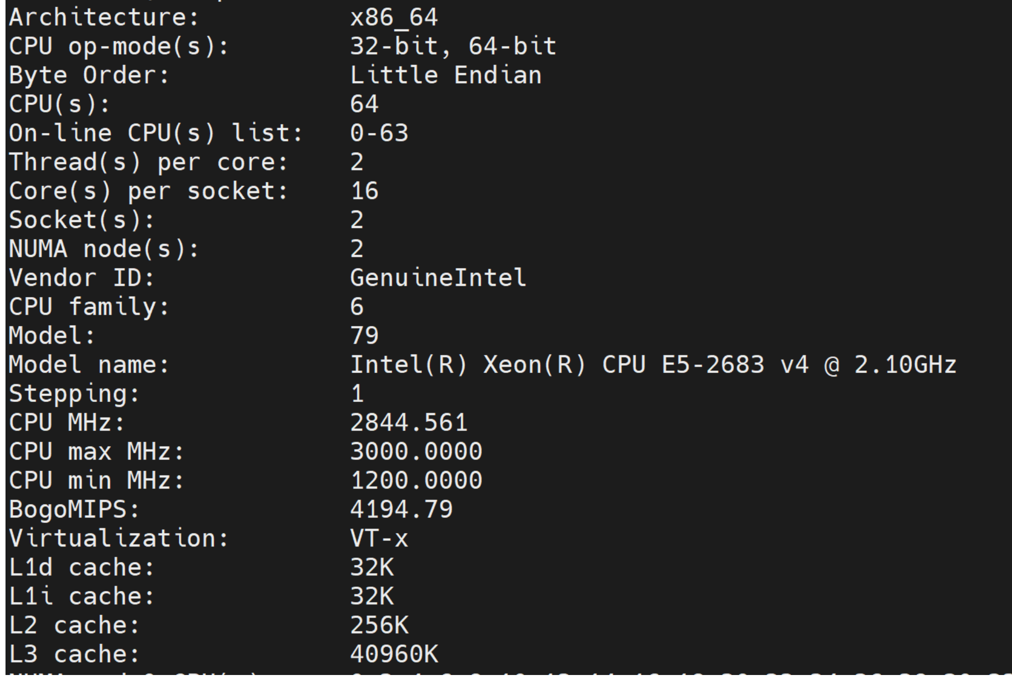
If you are stuck, please use the red post-it note to request help.

You are connected to Pronghorn's login node, which is the landing point for connecting to the system, and where you'll initiate parallel processing.

Getting Our Bearings

You should now be logged into one of the two login nodes. Which login node you are you on? How many cpu's does the login node have?

- You can remind yourself which node you are logged into with: `$hostname`
- Get details of the node with: `$lscpu`



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                64
On-line CPU(s) list:   0-63
Thread(s) per core:    2
Core(s) per socket:    16
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz
Stepping:              1
CPU MHz:               2844.561
CPU max MHz:           3000.0000
CPU min MHz:           1200.0000
BogoMIPS:              4194.79
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              40960K
```

Pronghorn has two main storage locations, your home directory and the "association" directory.

- You should start in your home directory: `$pwd`
- Your home directory can be found at: `$cd /data/gpfs/home/<username>` or `$cd ~`

Your home directory has 50GB of storage, which may be fine for some basic files but many people who use HPC are working on BIG DATA problems so you may need additional storage. That is where the association storage comes in. Your advisor can buy more storage here if you need it. This is stored in:

```
$cd /data/gpfs/assoc
$ls
```

Don't forget to tab complete to save yourself time.

YOU MAY NOT HAVE AN ACCESSIBLE ASSOCIATION FOLDER unless you've paid for it or are part of a group that has. You may also be a member of multiple associations. In theory, you shouldn't be able to see the contents of associations you are not part of.

Being a Good HPC Citizen (Shared Storage)

Try taking a peek into Jonathan Greenberg's directory (HINT: you shouldn't be able to):

```
$cd /data/gpfs/assoc/gears
```

- The GRAD778 association storage can be found at: `$cd /data/gpfs/assoc/grad_778-1`
- Look at the files in our association storage: `$ls -l`
- Look in the file 'hello_students.txt': `$more hello_students.txt`

Now, you may be ok with storing every file you own on your desktop, but if you are working with other folks, it is critically important to be organized. We generally recommend someone in a research group develop a storage organization. Let's decide on one now!

I made a subfolder called "StudentFiles" and am asking everyone to make a subfolder in that with their username. We can do this easily with:

```
$mkdir /data/gpfs/assoc/grad_778-1/StudentFiles/$USER
```

Note the "$USER" will automatically fill in your login name.

Now, let's change into that directory…

```
$cd /data/gpfs/assoc/grad_778-1/StudentFiles/$USER
```

Exercise 3: Testing Permissions

Being a Good HPC Citizen (Shared Storage)

The association space is a shared space with other members of your association. They can, in theory, read and edit these files. Let's see this!

Find someone near you to pair up with. One of you create a file in your new directory (don't forget to replace <filename> with a name of your choice):

```
$echo "This is a Secret Message" > <filename>
```

The second person should now try navigate to that file and look inside:

```
$more <filename>
```

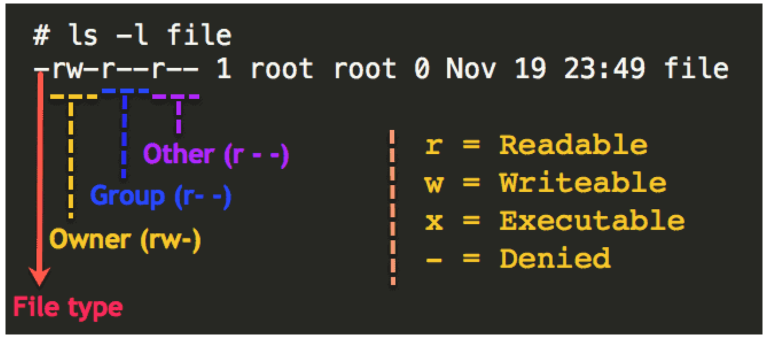
What do you find?

Can you delete the file your partner made?

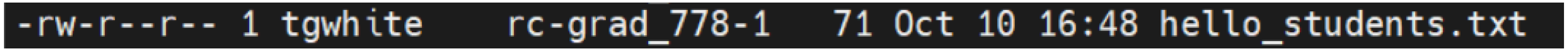
```
$rm <filename>
```

Let's take a look at the permissions of the file we just created:

```
$ls -lh <filename>
```



```
# ls -l file
-rw-r--r-- 1 root root 0 Nov 19 23:49 file
```



```
-rw-r--r-- 1 tgwhite    rc-grad_778-1   71 Oct 10 16:48 hello_students.txt
```

It looks like this file is owned by me and only I have write access, but everyone else can read my file. There is no privacy on shared machines!

Be careful with permissions in shared spaces.

Being a Good HPC Citizen (Shared Storage)

Ok, now let's make some big files! Everyone make a 100 gigabyte file:

```
$fallocate -l 100G /data/gpfs/assoc/grad_778-1/StudentFiles/$USER/foo
```

Who was able to create the file without `fallocate: fallocate failed: Disk quota exceeded?`

Notice it was about 9-10 of you!

It turns out our storage quota for this class is only 1TB, so after 10 of you made a 100 gigabyte file, we ran out of space!

Let's figure out who the culprits are:

```
$du -h /data/gpfs/assoc/grad_778-1/StudentFiles/* --max-depth=1 | sort -hr
```

Notice those are the same people who raised their hands :)

Let's clear out our space:

```
$rm /data/gpfs/assoc/grad_778-1/StudentFiles/$USER/foo
```

Be considerate in shared spaces.

Getting the files for this class

We are going to create a directory in your home for use with this class. Go to you home directory and create a folder called ‘grad778’. Inside this folder, create a folder called ‘class1’. You can do this in multiple steps or with the recursive mkdir command:

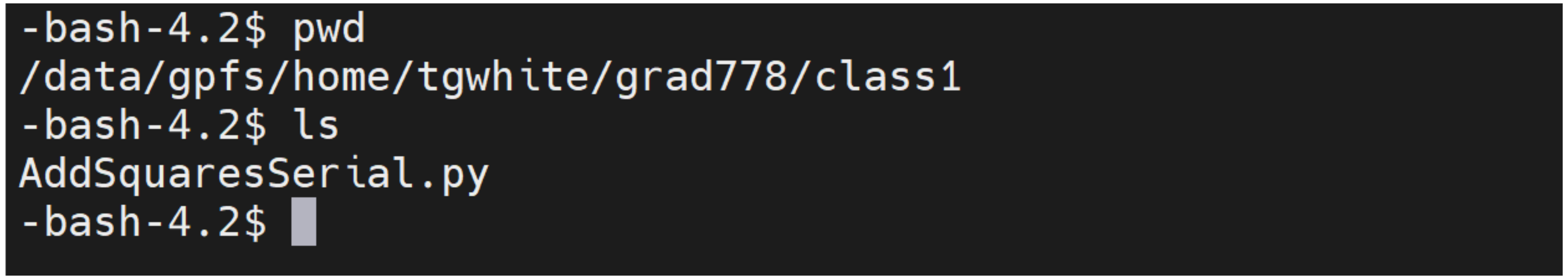
```
$mkdir -p ~/grad778/class1
```

Use the cd command to go into the folder

```
$cd ~/grad778/class1
```

You are going to need some files for this class. The first of these files I have placed into our association folder. Copy the file ‘AddSquaresSerial.py from our association folder into your home directory.

```
$cp /data/gpfs/assoc/grad_778-1/AddSquaresSerial.py .
```



```
-bash-4.2$ pwd
/data/gpfs/home/tgwhite/grad778/class1
-bash-4.2$ ls
AddSquaresSerial.py
-bash-4.2$ 
```

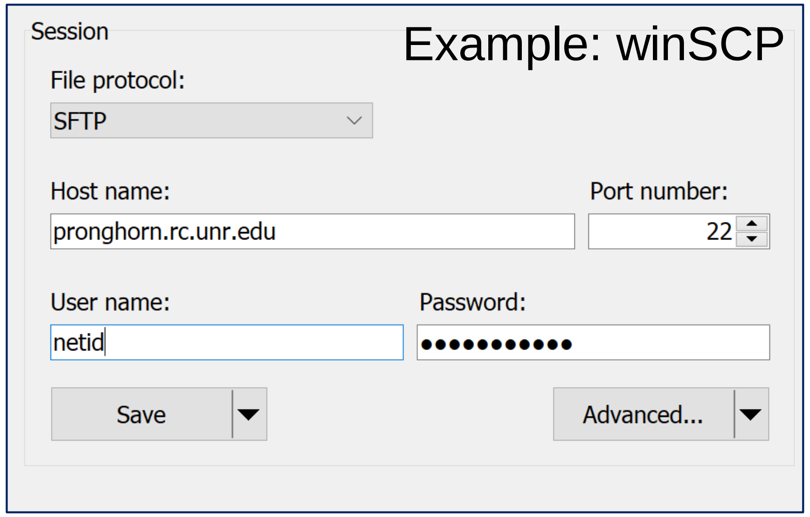
Transferring Files to/from Remote Computers

There are a few more files you are going to need for this class. However, it would be too easy to simply put them all onto Pronghorn for you. The rest of files are on webCampus.

SFTP (Secure File Transfer Protocol) is a file transfer protocol provides secure communication to a remote computer for the purposes of file transfer. Start by downloading the files from webCampus. Using an SFTP program of your choice, copy these files to the 'class1' directory.

List of programs

1. mobaXTerm (Win) ← Live example
2. WinSCP (Win)
3. Filezilla (Linux)
4. CyberDuck (Mac/Linux)
5. sftp (command line)



Example: winSCP

Session

File protocol: SFTP

Host name: pronghorn.rc.unr.edu

Port number: 22

User name: netid

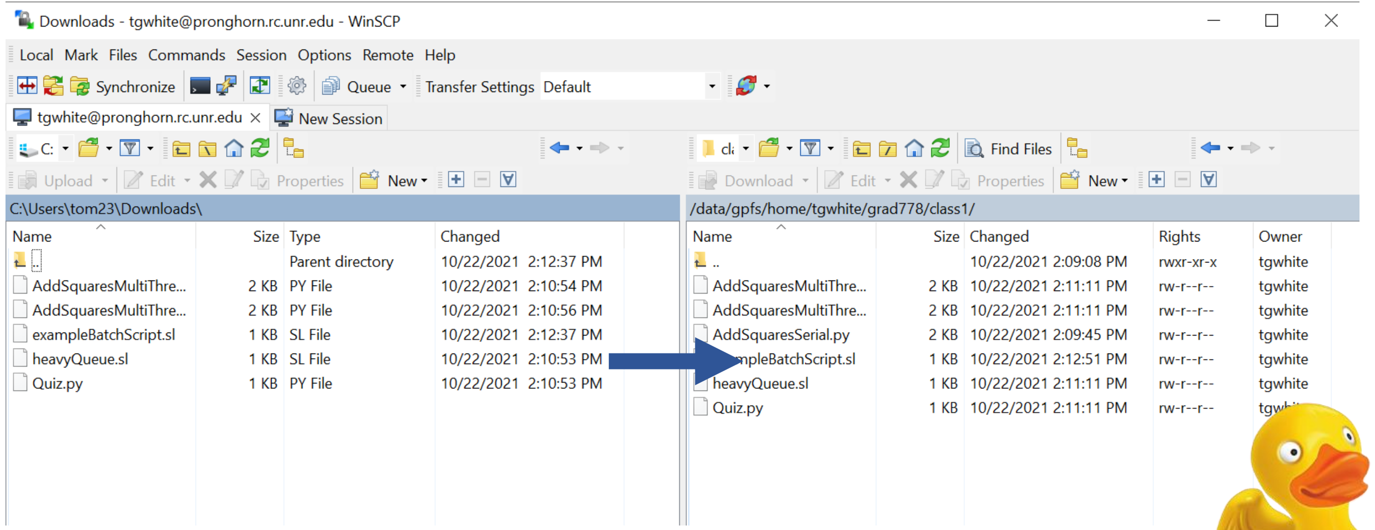
Password: ●●●●●●●●●●●

You should now be connected to Pronghorn, be inside your class1 directory, and have the files:

1. AddSquaresSerial.py
2. AddSquaresMultiThreading.py
3. AddSquaresMultiThreadingMany.py
4. exampleBatchScript.sl
5. heavyQueue.sl
6. Quiz.py

If you are not here, please use the red post-it note to request help.

To give everyone time to catch up, we'll do a quick in-class exercise.



Downloads - tgwhite@pronghorn.rc.unr.edu - WinSCP

Local Mark Files Commands Session Options Remote Help
Synchronize
Queue
Transfer Settings Default
tgwhite@pronghorn.rc.unr.edu
New Session
Find Files
Upload
Download
Edit
Properties
New

C:\Users\tom23\Downloads\

Name	Size	Type	Changed
..		Parent directory	10/22/2021 2:12:37 PM
AddSquaresMultiThre...	2 KB	PY File	10/22/2021 2:10:54 PM
AddSquaresMultiThre...	2 KB	PY File	10/22/2021 2:10:56 PM
exampleBatchScript.sl	1 KB	SL File	10/22/2021 2:12:37 PM
heavyQueue.sl	1 KB	SL File	10/22/2021 2:10:53 PM
Quiz.py	1 KB	PY File	10/22/2021 2:10:53 PM

/data/gpfs/home/tgwhite/grad778/class1/

Name	Size	Changed	Rights	Owner
..		10/22/2021 2:09:08 PM	rwxr-xr-x	tgwhite
AddSquaresMultiThre...	2 KB	10/22/2021 2:11:11 PM	rw-r--r--	tgwhite
AddSquaresMultiThre...	2 KB	10/22/2021 2:11:11 PM	rw-r--r--	tgwhite
AddSquaresSerial.py	2 KB	10/22/2021 2:09:45 PM	rw-r--r--	tgwhite
mpleBatchScript.sl	1 KB	10/22/2021 2:12:51 PM	rw-r--r--	tgwhite
heavyQueue.sl	1 KB	10/22/2021 2:11:11 PM	rw-r--r--	tgwhite
Quiz.py	1 KB	10/22/2021 2:11:11 PM	rw-r--r--	tgw

The Human Computer

The first known use of computer was in a 1613 book called The Yong Mans Gleanings by the English writer Richard Braithwait: "I haue [sic] read the truest computer of Times, and the best Arithmetician that euer [sic] breathed, and he reduceth thy dayes into a short number." *The Oxford English Dictionary*

This usage of the term referred to a human computer, a person who carried out calculations or computations. The word continued with the same meaning until the middle of the 20th century.

The idea of splitting up the work amongst different people is the beginning of parallel processing. Of course, in this class we are interested in splitting up the job between cores on your computer, not people! However, we can investigate a few of the ideas using a human computer.

Macie Roberts' computing group circa 1955 (far right). Barbara Paulson is on the telephone (standing, back left). Helen Ling is at the second desk in the left row. (Credit: JPL)

Introduction to Parallel Processing

What is the point of using a High-Performance Computer (HPC) like Pronghorn to do work? Let's first think about a sequential computing task. Say we want to take six random numbers, square each of them, and add them up.

Here are our six numbers:
2354, 9743, 2856, 1928, 4345, 2384

How long will it take to perform this calculation serially (that means performing each calculation one at a time)?

Calculation	Time
2354 x 2354	
9743 x 9743	
2856 x 2856	
1928 x 1928	
4345 x 4345	
3421 x 3421	
Add Numbers	

Introduction to Parallel Processing

What is the point of using a High-Performance Computer (HPC) like Pronghorn to do work? Let's first think about a sequential computing task. Say we want to take six random numbers, square each of them, and add them up.

Here are our six numbers:
2354, 9743, 2856, 1928, 4345, 2384

How long will it take to perform this calculation serially (that means performing each calculation one at a time)?

Calculation	Time
2354 x 2354	10 seconds
9743 x 9743	10 seconds
2856 x 2856	10 seconds
1928 x 1928	10 seconds
4345 x 4345	10 seconds
3421 x 3421	10 seconds
Add Numbers	20 seconds
	80 seconds

Introduction to Parallel Processing

What if there were two processors?

Person 1 Calculation	Time	Person 2 Calculation	Time
2354 x 2354		9743 x 9743	
2856 x 2856		1928 x 1928	
4345 x 4345		3421 x 3421	
Add Numbers			

Introduction to Parallel Processing

What if there were two processors?

Person 1 Calculation	Time	Person 2 Calculation	Time
2354 x 2354	10 seconds	9743 x 9743	10 seconds
2856 x 2856	10 seconds	1928 x 1928	10 seconds
4345 x 4345	10 seconds	3421 x 3421	10 seconds
Add Numbers	20 seconds		
	50 seconds		**30 seconds**

Notice, person 2 finished their work before person 1; they didn't do anything during the last step.

Even though person two finished in 30 seconds, the total length of the job is given by the time it took person 1 to finish. i.e., 50 seconds.

Introduction to Parallel Processing

What if there were four processors?

Person 1 Calculation	Time	Person 2 Calculation	Time	Person 3 Calculation	Person 4 Calculation
2354 x 2354		9743 x 9743		2856 x 2856	1928 x 1928
4345 x 4345		3421 x 3421		Add Numbers	

Introduction to Parallel Processing

What if there were four processors?

Person 1 Calculation	Time	Person 2 Calculation	Time	Person 3 Calculation		Person 4 Calculation	
2354 x 2354	10 seconds	9743 x 9743	10 seconds	2856 x 2856	10 seconds	1928 x 1928	10 seconds
4345 x 4345	10 seconds	3421 x 3421	10 seconds	Add Numbers	20 seconds		
	20 seconds		**20 seconds**		**30 seconds**		**10 seconds**

The total length of the job is given by the time it took person 3 to finish. i.e., 30 seconds.

Notice, it is possible for person 3 to be waiting for person 1 and 2 to complete their calculations before starting! The jobs are not totally independent.

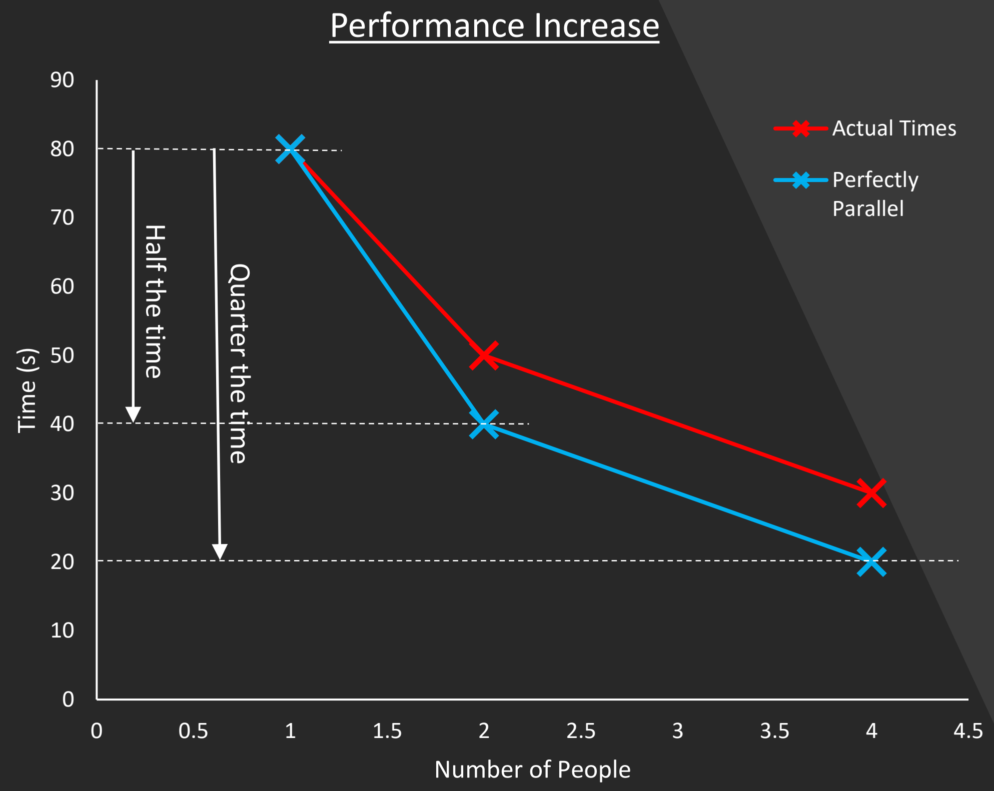
Break

Introduction to Parallel Processing

As we added more people, we computed the answer faster. However, two people did not complete the job half the time, and four did not take a quarter of the time. This is because the job did not parallelize perfectly. In our case, the person computing the addition of the numbers had to wait for everyone else to finish before completing their task.

Not all processes are parallelizable! How about a problem where we want each output to be the square of the previous calculation. In other words, each step requires knowledge of the previous step before it executes. These types of problems are not parallelizable, and we would get no speed up by using multiple people.

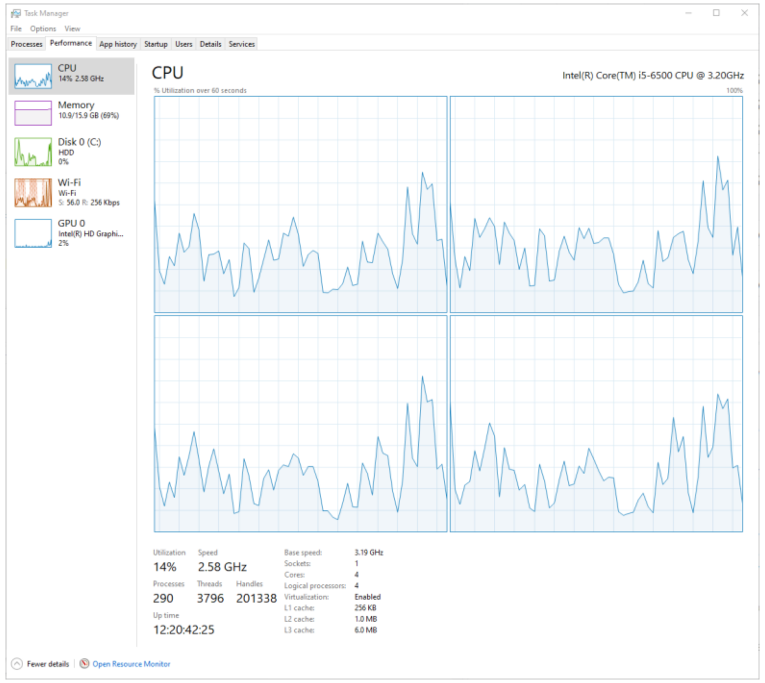
A typical first step in understanding if a problem is conducive to parallel processing is determining if we can break a problem up into smaller problems, like in our previous case. Basic parallel processing is basically a big "for" loop; we iterate through these smaller problems, sending them to different computation resources and (usually) combining the results of the smaller problems.



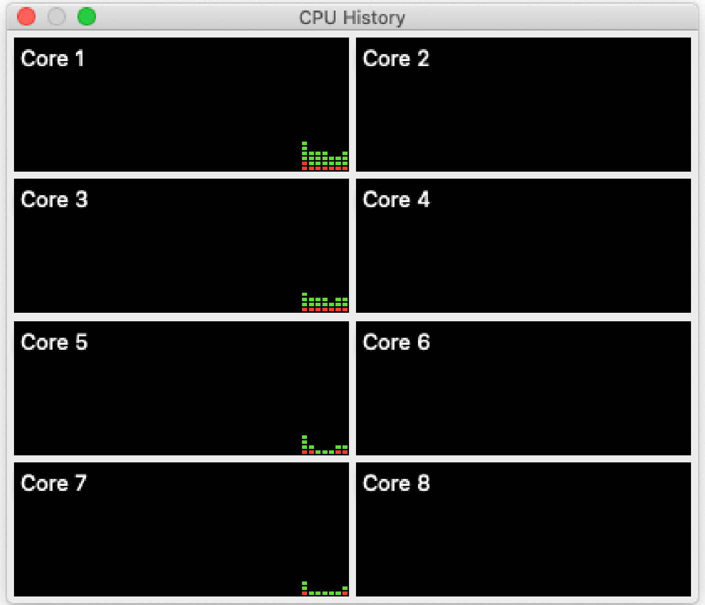
Scales of Parallel Computing

Let's first take a look at your own parallel computing capabilities. Every modern computer has multiple "logical cores" that we can do work with (we will set aside the complexities of logical cores vs. physical cores). If you are on Windows, right click your toolbar, click "Task Manager", select the Performance tab, click CPU. In the moving chart in the middle, if you only see one graph, right click that graph -> Change graph to -> logical processors. Now you can see how many processors you have.

On a Mac, boot up Activity Monitor (in your Applications/Utilities folder), and Window -> CPU History (you may need to make the window bigger to see all logical cores).


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU 14% 2.58 GHz
Memory 10.9/15.9 GB (69%)
Disk 0 (C:) HDD 0%
Wi-Fi Wi-Fi S: 56.0 R: 256 Kbps
GPU 0 Intel(R) HD Graphi... 2%
CPU
Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
% Utilization over 60 seconds
100%
Fewer details
Open Resource Monitor

```
Utilization   Speed                 Base speed:           3.19 GHz
14%           2.58 GHz              Sockets:              1
Processes     Threads    Handles    Cores:                4
290           3796       201338     Logical processors:   4
Up time                             Virtualization:       Enabled
12:20:42:25                         L1 cache:             256 KB
                                    L2 cache:             1.0 MB
                                    L3 cache:             6.0 MB
```



Scales of Parallel Computing

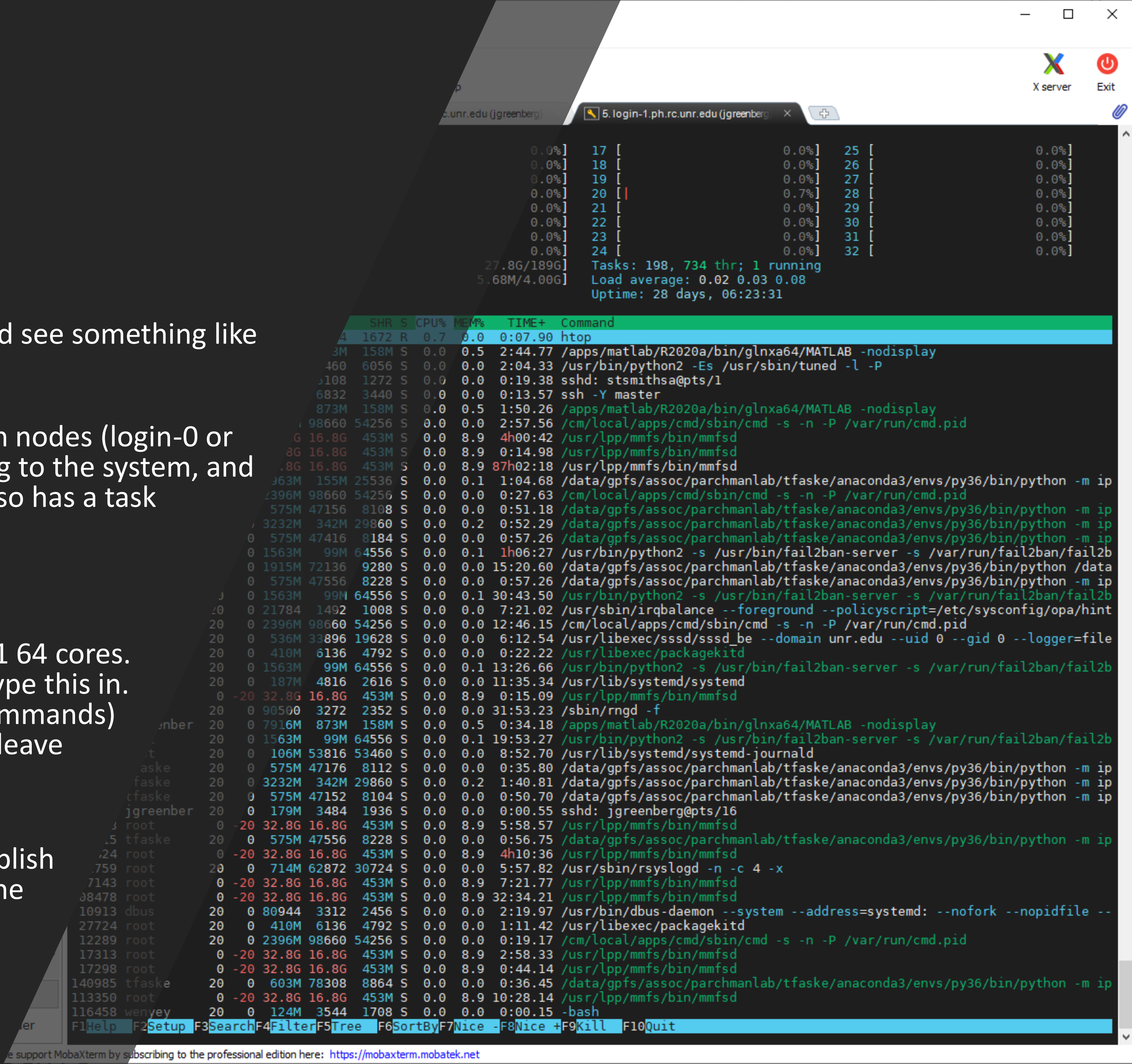
Now, if you are ssh'ed into Pronghorn, you should see something like [username@login-1 ~]$ or -bash-4.2$

You are connected to one of two Pronghorn login nodes (login-0 or login-1), which is the landing point for connecting to the system, and where you'll initiate parallel processing. Linux also has a task manager that is easy to see, try typing:

```
$htop
```

You will see at the top of the screen the login-0/1 64 cores. They may or may not be in use at the time you type this in. At the bottom you can see various processes (commands) running, and the % CPU they are using. You can leave htop by pressing F10.

Now, how can we leverage more cores to accomplish a task in less human-time than if we only used one core. How does Pronghorn help with this problem?



```
           0.0%]   17 [                          0.0%]   25 [                          0.0%]
           0.0%]   18 [                          0.0%]   26 [                          0.0%]
           0.0%]   19 [                          0.0%]   27 [                          0.0%]
           0.0%]   20 [|                         0.7%]   28 [                          0.0%]
           0.0%]   21 [                          0.0%]   29 [                          0.0%]
           0.0%]   22 [                          0.0%]   30 [                          0.0%]
           0.0%]   23 [                          0.0%]   31 [                          0.0%]
           0.0%]   24 [                          0.0%]   32 [                          0.0%]
      27.8G/189G]   Tasks: 198, 734 thr; 1 running
     5.68M/4.00G]   Load average: 0.02 0.03 0.08
                    Uptime: 28 days, 06:23:31

                        SHR S CPU% MEM%   TIME+  Command
                   4   1672 R  0.7  0.0  0:07.90 htop
                  3M   158M S  0.0  0.5  2:44.77 /apps/matlab/R2020a/bin/glnxa64/MATLAB -nodisplay
                 460   6056 S  0.0  0.0  2:04.33 /usr/bin/python2 -Es /usr/sbin/tuned -l -P
                 108   1272 S  0.0  0.0  0:19.38 sshd: stsmithsa@pts/1
                6832   3440 S  0.0  0.0  0:13.57 ssh -Y master
                873M   158M S  0.0  0.5  1:50.26 /apps/matlab/R2020a/bin/glnxa64/MATLAB -nodisplay
         98660 54256 S  0.0  0.0  2:57.56 /cm/local/apps/cmd/sbin/cmd -s -n -P /var/run/cmd.pid
             G 16.8G   453M S  0.0  8.9  4h00:42 /usr/lpp/mmfs/bin/mmfsd
            8G 16.8G   453M S  0.0  8.9  0:14.98 /usr/lpp/mmfs/bin/mmfsd
           .8G 16.8G   453M S  0.0  8.9 87h02:18 /usr/lpp/mmfs/bin/mmfsd
          963M  155M 25536 S  0.0  0.1  1:04.68 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
         2396M 98660 54256 S  0.0  0.0  0:27.63 /cm/local/apps/cmd/sbin/cmd -s -n -P /var/run/cmd.pid
          575M 47156  8108 S  0.0  0.0  0:51.18 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
         3232M  342M 29860 S  0.0  0.2  0:52.29 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
       0  575M 47416  8184 S  0.0  0.0  0:57.26 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
       0 1563M   99M 64556 S  0.0  0.1  1h06:27 /usr/bin/python2 -s /usr/bin/fail2ban-server -s /var/run/fail2ban/fail2b
       0 1915M 72136  9280 S  0.0  0.0 15:20.60 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python /data
       0  575M 47556  8228 S  0.0  0.0  0:57.26 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
       0 1563M   99M 64556 S  0.0  0.1 30:43.50 /usr/bin/python2 -s /usr/bin/fail2ban-server -s /var/run/fail2ban/fail2b
  20   0 21784  1492  1008 S  0.0  0.0  7:21.02 /usr/sbin/irqbalance --foreground --policyscript=/etc/sysconfig/opa/hint
  20   0 2396M 98660 54256 S  0.0  0.0 12:46.15 /cm/local/apps/cmd/sbin/cmd -s -n -P /var/run/cmd.pid
  20   0  536M 33896 19628 S  0.0  0.0  6:12.54 /usr/libexec/sssd/sssd_be --domain unr.edu --uid 0 --gid 0 --logger=file
  20   0  410M  6136  4792 S  0.0  0.0  0:22.22 /usr/libexec/packagekitd
  20   0 1563M   99M 64556 S  0.0  0.1 13:26.66 /usr/bin/python2 -s /usr/bin/fail2ban-server -s /var/run/fail2ban/fail2b
  20   0  187M  4816  2616 S  0.0  0.0 11:35.34 /usr/lib/systemd/systemd
   0 -20 32.8G 16.8G  453M S  0.0  8.9  0:15.09 /usr/lpp/mmfs/bin/mmfsd
  20   0 90500  3272  2352 S  0.0  0.0 31:53.23 /sbin/rngd -f
enber  20   0 7916M  873M  158M S  0.0  0.5  0:34.18 /apps/matlab/R2020a/bin/glnxa64/MATLAB -nodisplay
  20   0 1563M   99M 64556 S  0.0  0.1 19:53.27 /usr/bin/python2 -s /usr/bin/fail2ban-server -s /var/run/fail2ban/fail2b
t        20   0  106M 53816 53460 S  0.0  0.0  8:52.70 /usr/lib/systemd/systemd-journald
aske     20   0  575M 47176  8112 S  0.0  0.0  0:35.80 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
faske    20   0 3232M  342M 29860 S  0.0  0.2  1:40.81 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
tfaske   20   0  575M 47152  8104 S  0.0  0.0  0:50.70 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
jgreenber 20  0  179M  3484  1936 S  0.0  0.0  0:00.55 sshd: jgreenberg@pts/16
3 root      0 -20 32.8G 16.8G  453M S  0.0  8.9  5:58.57 /usr/lpp/mmfs/bin/mmfsd
.5 tfaske  20   0  575M 47556  8228 S  0.0  0.0  0:56.75 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
24 root     0 -20 32.8G 16.8G  453M S  0.0  8.9  4h10:36 /usr/lpp/mmfs/bin/mmfsd
.759 root  20   0  714M 62872 30724 S  0.0  0.0  5:57.82 /usr/sbin/rsyslogd -n -c 4 -x
7143 root   0 -20 32.8G 16.8G  453M S  0.0  8.9  7:21.77 /usr/lpp/mmfs/bin/mmfsd
08478 root  0 -20 32.8G 16.8G  453M S  0.0  8.9 32:34.21 /usr/lpp/mmfs/bin/mmfsd
10913 dbus     20   0 80944  3312  2456 S  0.0  0.0  2:19.97 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --
27724 root     20   0  410M  6136  4792 S  0.0  0.0  1:11.42 /usr/libexec/packagekitd
12289 root     20   0 2396M 98660 54256 S  0.0  0.0  0:19.17 /cm/local/apps/cmd/sbin/cmd -s -n -P /var/run/cmd.pid
17313 root      0 -20 32.8G 16.8G  453M S  0.0  8.9  2:58.33 /usr/lpp/mmfs/bin/mmfsd
17298 root      0 -20 32.8G 16.8G  453M S  0.0  8.9  0:44.14 /usr/lpp/mmfs/bin/mmfsd
140985 tfaske  20   0  603M 78308  8864 S  0.0  0.0  0:36.45 /data/gpfs/assoc/parchmanlab/tfaske/anaconda3/envs/py36/bin/python -m ip
113350 root     0 -20 32.8G 16.8G  453M S  0.0  8.9 10:28.14 /usr/lpp/mmfs/bin/mmfsd
116458 wenyey  20   0  124M  3544  1708 S  0.0  0.0  0:00.15 -bash
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

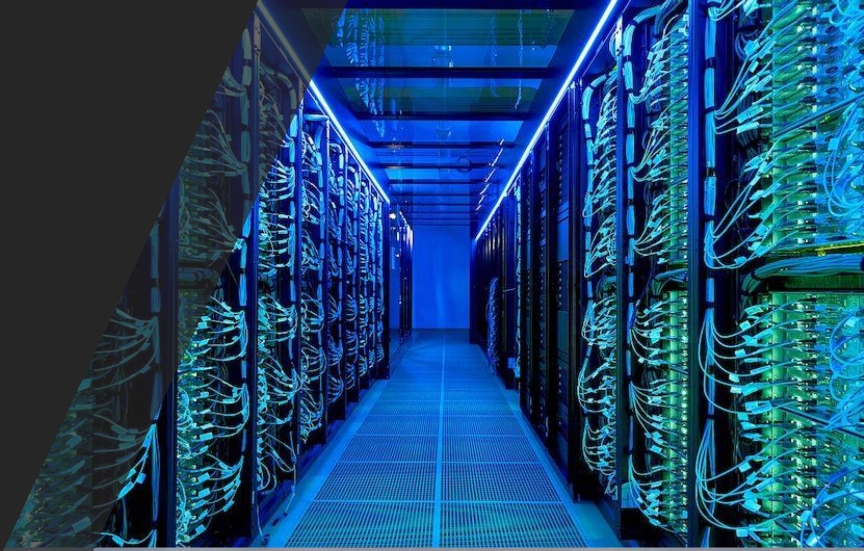
e support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net

UNR's High Performance Computer: Pronghorn!

Pronghorn is the University of Nevada, Reno's High-Performance Computing (HPC) cluster. The GPU-accelerated system is designed, built and maintained by the Office of Information Technology's HPC Team. Pronghorn and the HPC Team supports general research across the Nevada System of Higher Education (NSHE).

Pronghorn is composed of CPU, GPU, Visualization, and Storage subsystems interconnected by a 100Gb/s non-blocking Intel Omni-Path fabric. The CPU partition features **108 nodes**, **3,456 CPU cores**, and **24.8TiB of memory**. The GPU partition features **44 NVIDIA Tesla P100 GPUs**, **352 CPU cores**, and **2.75TiB of memory**. The Visualization partition is composed of **three NVIDIA Tesla V100 GPUs**, **48 CPU cores**, and **1.1TiB of memory**. The storage system uses the IBM SpectrumScale file system to provide **2PiB of high-performance storage**. The computational and storage capabilities of Pronghorn will regularly expand to meet NSHE computing demands.

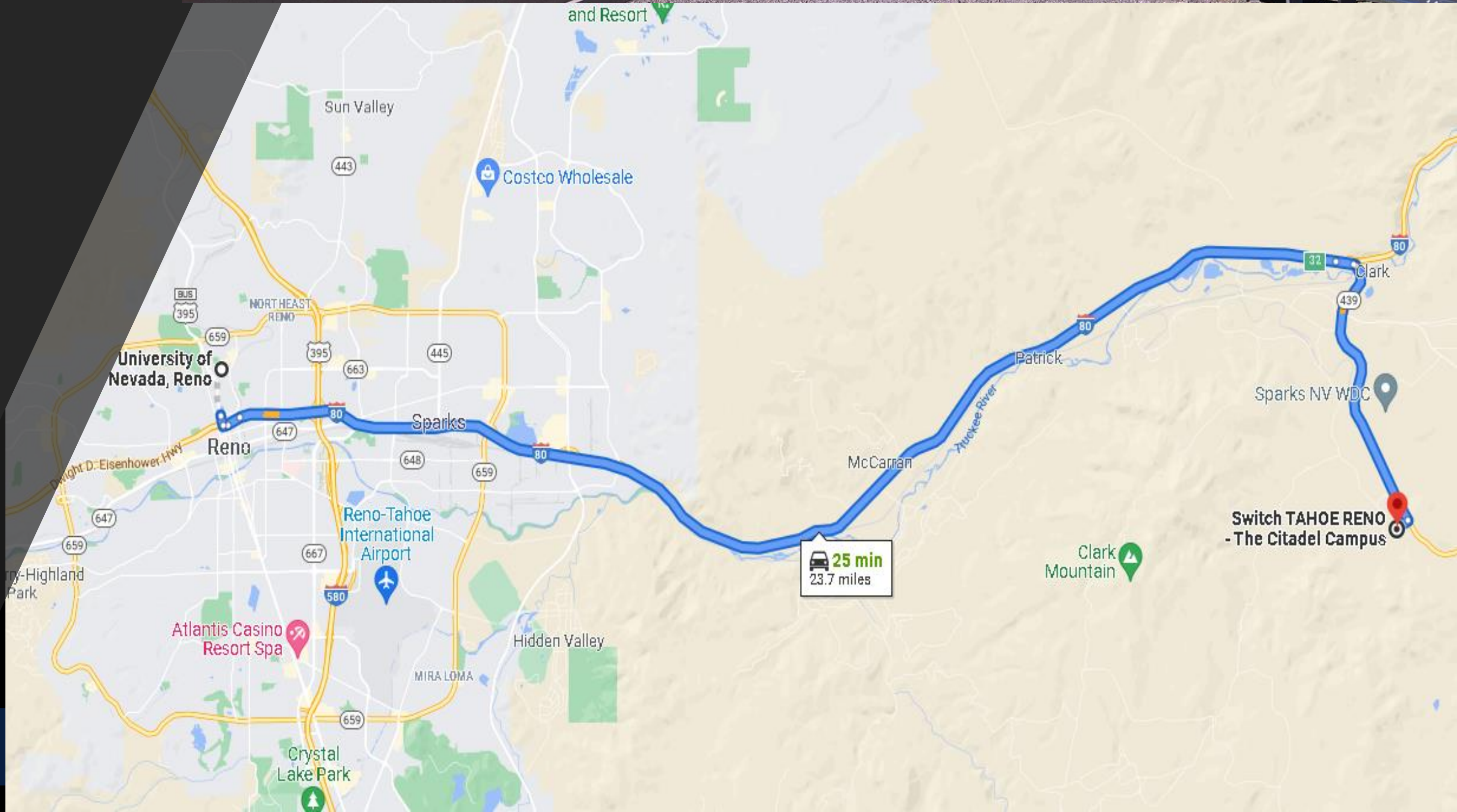
Pronghorn is available to all University of Nevada, Reno faculty, staff, students, and sponsored affiliates. Priority access to the system is available for purchase.



The pronghorn antelope is the fastest mammal on the North American continent

The Geography of Pronghorn

- Pronghorn is collocated at the Switch Citadel Campus located 25 miles East of the University of Nevada, Reno.
- The Switch Citadel is rated Tier 5 Platinum, and at 7.2 million square feet is the largest, most advanced data center campus on the planet.
- Up to 650 MW of power using 100% renewable energy
- 10Gbps circuits at 4-millisecond latency to Silicon Valley/San Francisco and 9-millisecond connection to Los Angeles via the Switch SUPERLOOP
- 24/7/365 on-site network operations center (NOC), fire, safety and security
- Proprietary tri-redundant UPS power system

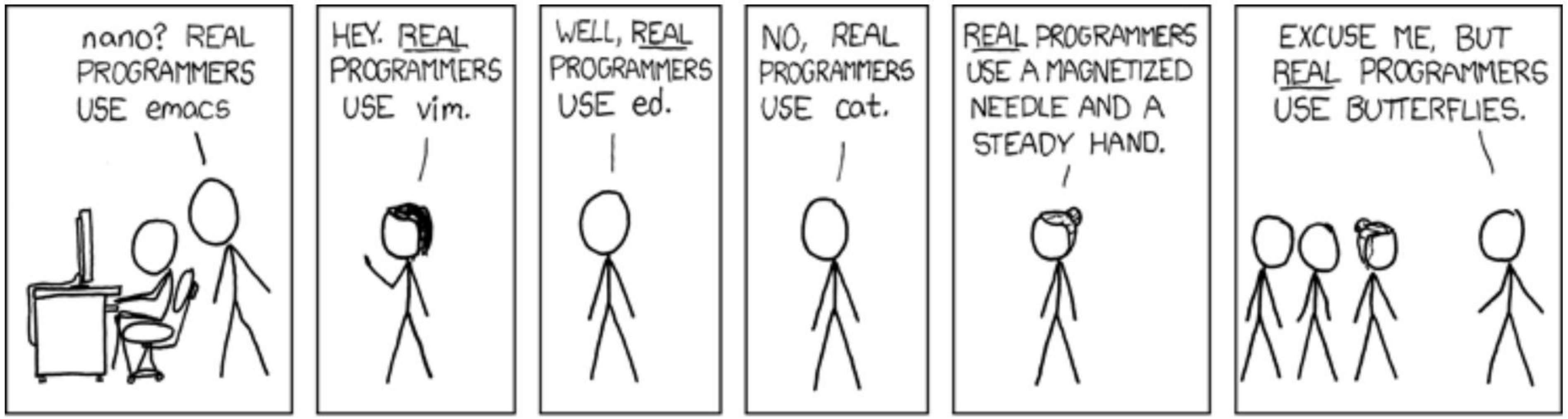


Editing Files on Pronghorn

The login node on most HPC clusters do not typically support text editors with graphical interfaces. Pronghorn DOES have gedit installed. However, using a graphical interface over an SSH connection can be slow. Thus, in this class, we will practice using a command line editor. Note, there is intense debate over the best command line editor. *You can use whichever one you like.* However, I think nano is one of the easiest, and so we will use that. Hopefully, you followed along with the tutorial video on webCampus. Try making a file, adding some text, saving the file, and exiting – make sure you are in the class1 directory in the grad778 directory in your home directory.

```
$touch myFirstFile.txt
$nano myFirstFile.txt
[type some words]
Ctrl+O
Enter
Ctrl+X
```

Don't forget, you cannot use your mouse to go to a location in the document!

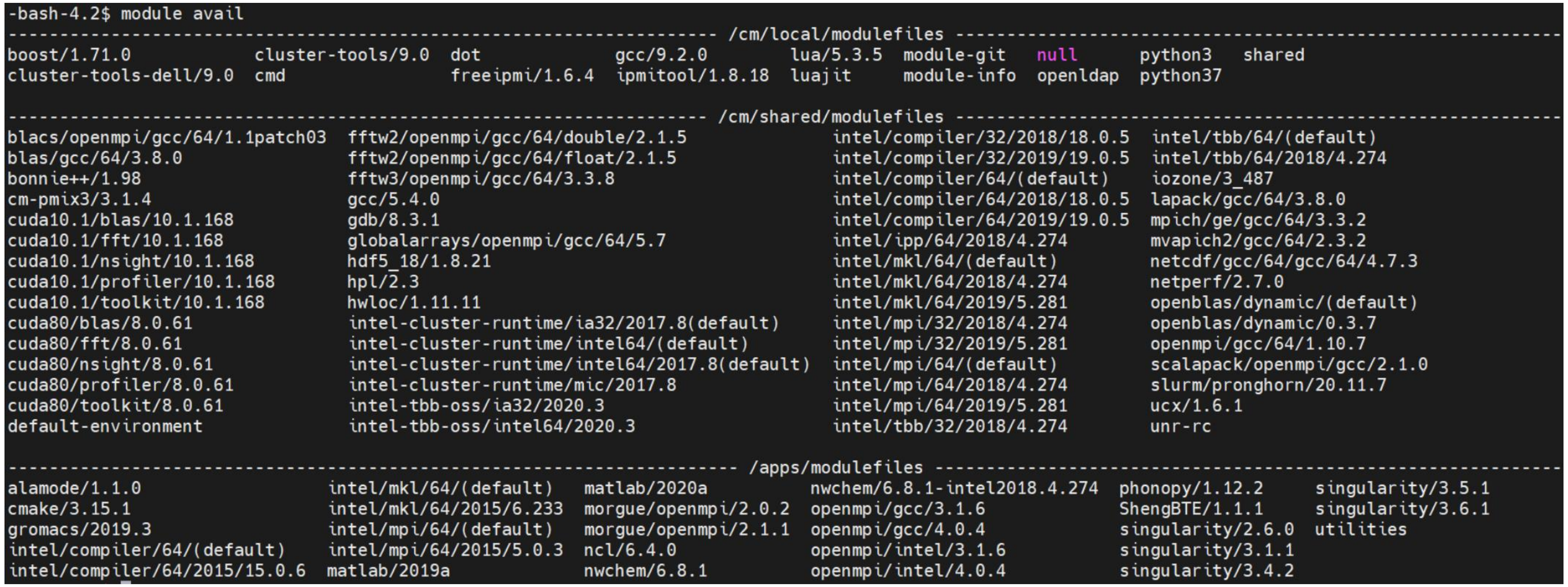

nano? REAL PROGRAMMERS USE emacs
HEY. REAL PROGRAMMERS USE vim.
WELL, REAL PROGRAMMERS USE ed.
NO, REAL PROGRAMMERS USE cat.
REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.
EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.

Accessing Software via modules

The module package is available on Pronghorn allowing users to access non-standard tools or alternate versions of standard packages. This is also an alternative way to configure your environment. Let's see what's available:

```
$module avail
```

Notice that most of the available modules are not applications, rather they are compilers and common libraries that will enable you to compile and build applications yourself.



```
-bash-4.2$ module avail
-------------------------------------------------------------------- /cm/local/modulefiles ---------------------------------------------------------------------
boost/1.71.0            cluster-tools/9.0  dot             gcc/9.2.0        lua/5.3.5  module-git   null      python3   shared
cluster-tools-dell/9.0  cmd                freeipmi/1.6.4  ipmitool/1.8.18  luajit     module-info  openldap  python37

------------------------------------------------------------------- /cm/shared/modulefiles --------------------------------------------------------------------
blacs/openmpi/gcc/64/1.1patch03  fftw2/openmpi/gcc/64/double/2.1.5               intel/compiler/32/2018/18.0.5  intel/tbb/64/(default)
blas/gcc/64/3.8.0                fftw2/openmpi/gcc/64/float/2.1.5                intel/compiler/32/2019/19.0.5  intel/tbb/64/2018/4.274
bonnie++/1.98                    fftw3/openmpi/gcc/64/3.3.8                      intel/compiler/64/(default)    iozone/3_487
cm-pmix3/3.1.4                   gcc/5.4.0                                       intel/compiler/64/2018/18.0.5  lapack/gcc/64/3.8.0
cuda10.1/blas/10.1.168           gdb/8.3.1                                       intel/compiler/64/2019/19.0.5  mpich/ge/gcc/64/3.3.2
cuda10.1/fft/10.1.168            globalarrays/openmpi/gcc/64/5.7                 intel/ipp/64/2018/4.274        mvapich2/gcc/64/2.3.2
cuda10.1/nsight/10.1.168         hdf5_18/1.8.21                                  intel/mkl/64/(default)         netcdf/gcc/64/gcc/64/4.7.3
cuda10.1/profiler/10.1.168       hpl/2.3                                         intel/mkl/64/2018/4.274        netperf/2.7.0
cuda10.1/toolkit/10.1.168        hwloc/1.11.11                                   intel/mkl/64/2019/5.281        openblas/dynamic/(default)
cuda80/blas/8.0.61               intel-cluster-runtime/ia32/2017.8(default)      intel/mpi/32/2018/4.274        openblas/dynamic/0.3.7
cuda80/fft/8.0.61                intel-cluster-runtime/intel64/(default)         intel/mpi/32/2019/5.281        openmpi/gcc/64/1.10.7
cuda80/nsight/8.0.61             intel-cluster-runtime/intel64/2017.8(default)   intel/mpi/64/(default)         scalapack/openmpi/gcc/2.1.0
cuda80/profiler/8.0.61           intel-cluster-runtime/mic/2017.8                intel/mpi/64/2018/4.274        slurm/pronghorn/20.11.7
cuda80/toolkit/8.0.61            intel-tbb-oss/ia32/2020.3                       intel/mpi/64/2019/5.281        ucx/1.6.1
default-environment              intel-tbb-oss/intel64/2020.3                    intel/tbb/32/2018/4.274        unr-rc

---------------------------------------------------------------------- /apps/modulefiles ----------------------------------------------------------------------
alamode/1.1.0                  intel/mkl/64/(default)     matlab/2020a          nwchem/6.8.1-intel2018.4.274  phonopy/1.12.2      singularity/3.5.1
cmake/3.15.1                   intel/mkl/64/2015/6.233    morgue/openmpi/2.0.2  openmpi/gcc/3.1.6             ShengBTE/1.1.1      singularity/3.6.1
gromacs/2019.3                 intel/mpi/64/(default)     morgue/openmpi/2.1.1  openmpi/gcc/4.0.4             singularity/2.6.0   utilities
intel/compiler/64/(default)    intel/mpi/64/2015/5.0.3    ncl/6.4.0             openmpi/intel/3.1.6           singularity/3.1.1
intel/compiler/64/2015/15.0.6  matlab/2019a               nwchem/6.8.1          openmpi/intel/4.0.4           singularity/3.4.2
```

Software Versioning

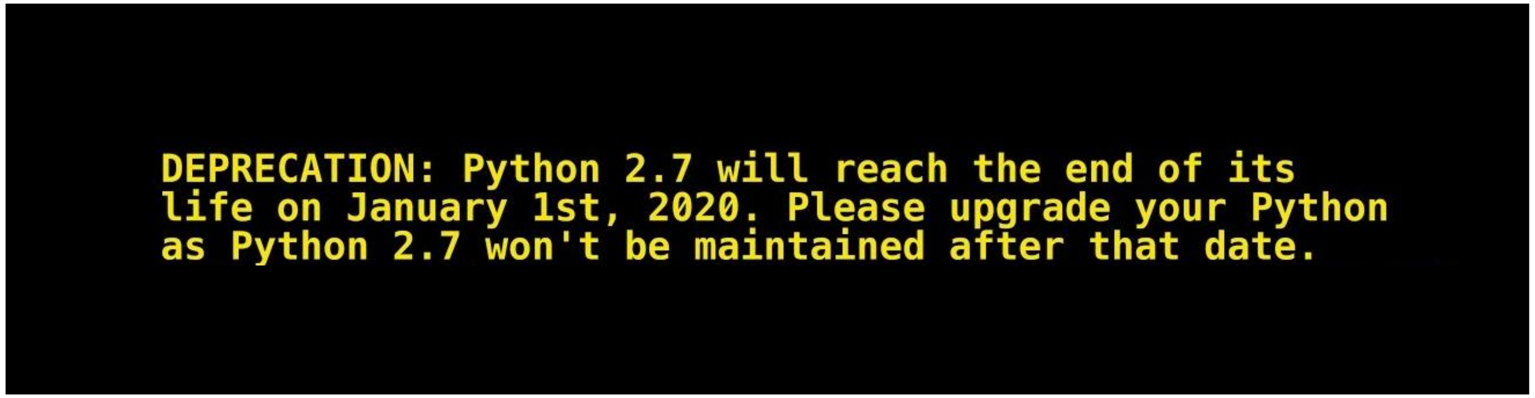
What version of python is installed on Pronghorn?

```
$python --version
```

Software Versioning

What version of python is installed on Pronghorn?

```
$python --version
```



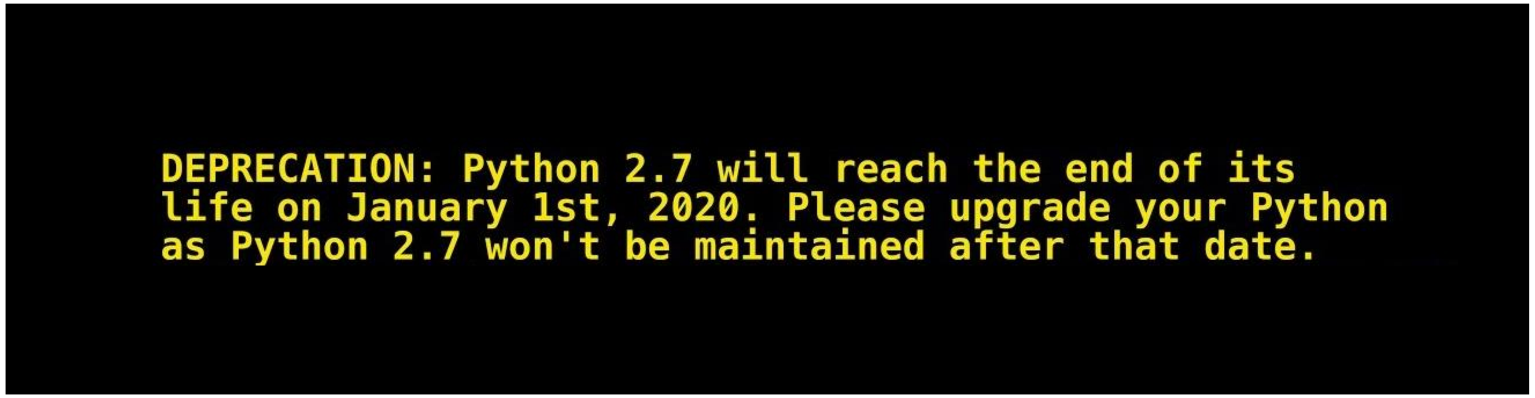
```
DEPRECATION: Python 2.7 will reach the end of its
life on January 1st, 2020. Please upgrade your Python
as Python 2.7 won't be maintained after that date.
```

Exercise 8: Load Python 3

Software Versioning

What version of python is installed on Pronghorn?

```
$python --version
```



```
DEPRECATION: Python 2.7 will reach the end of its
life on January 1st, 2020. Please upgrade your Python
as Python 2.7 won't be maintained after that date.
```

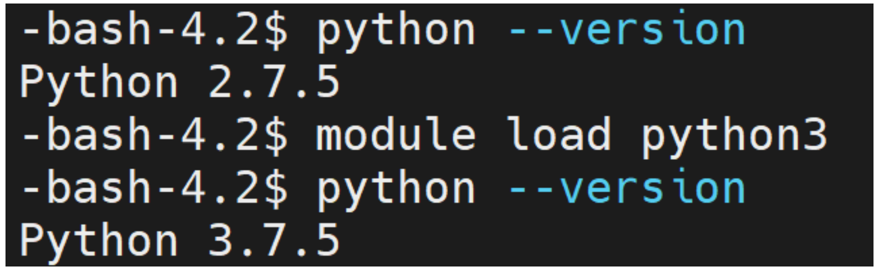
To run the programs in this class, you will need python version 3.
Load it now:

```
$module load python3
```

You can check to see if you have the latest version of python.

```
$python --version
```

You will need to reload the modules you need each time you login to pronghorn



```
-bash-4.2$ python --version
Python 2.7.5
-bash-4.2$ module load python3
-bash-4.2$ python --version
Python 3.7.5
```

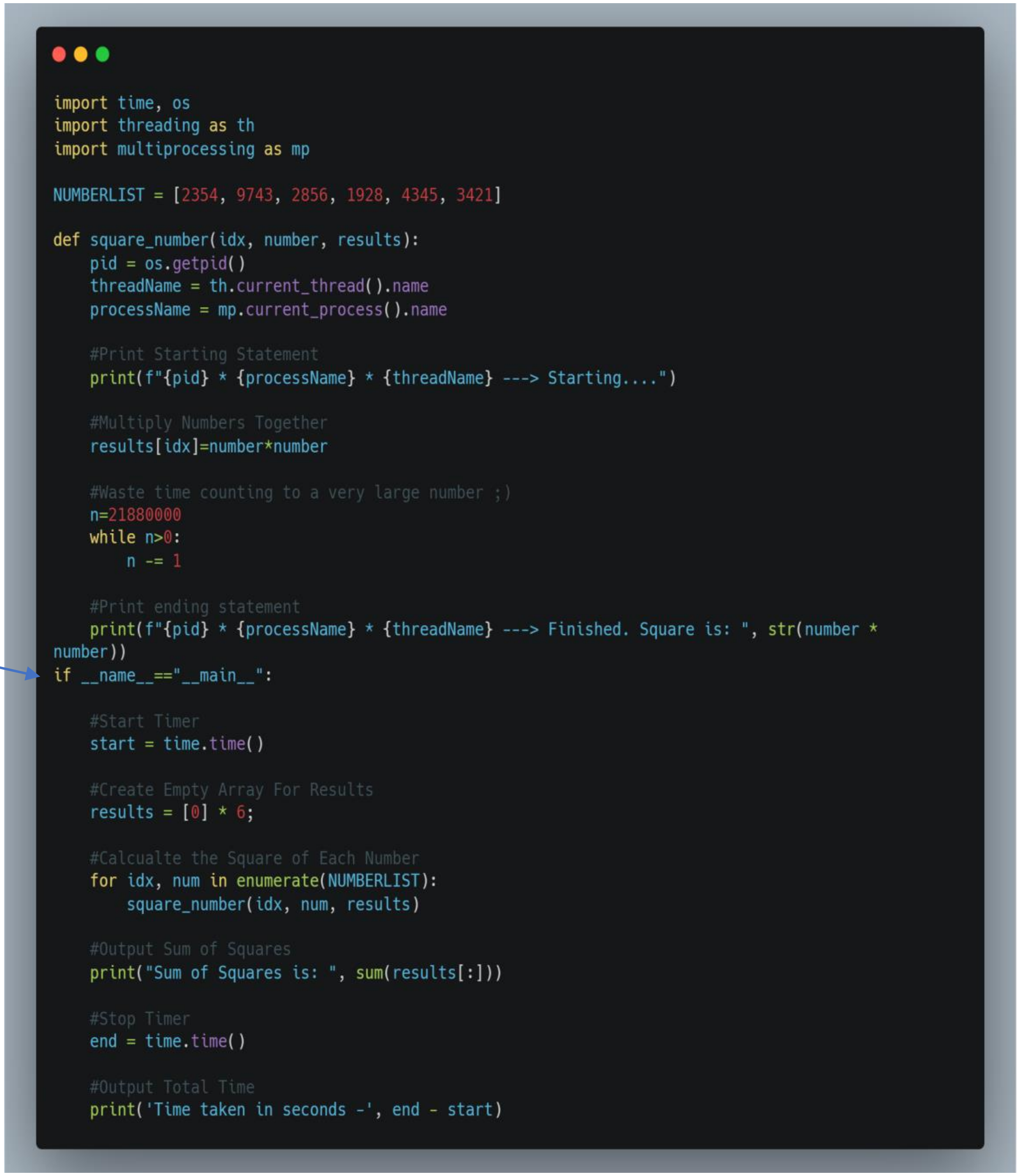
Running Our First Program

The time as come to run our first real program.

First take a look in the file:

```
$more AddSquaresSerial.py
```

Start here at main and see if you can follow along with the file



```python
import time, os
import threading as th
import multiprocessing as mp

NUMBERLIST = [2354, 9743, 2856, 1928, 4345, 3421]

def square_number(idx, number, results):
    pid = os.getpid()
    threadName = th.current_thread().name
    processName = mp.current_process().name

    #Print Starting Statement
    print(f"{pid} * {processName} * {threadName} ---> Starting....")

    #Multiply Numbers Together
    results[idx]=number*number

    #Waste time counting to a very large number ;)
    n=21880000
    while n>0:
        n -= 1

    #Print ending statement
    print(f"{pid} * {processName} * {threadName} ---> Finished. Square is: ", str(number *
number))
if __name__=="__main__":

    #Start Timer
    start = time.time()

    #Create Empty Array For Results
    results = [0] * 6;

    #Calcualte the Square of Each Number
    for idx, num in enumerate(NUMBERLIST):
        square_number(idx, num, results)

    #Output Sum of Squares
    print("Sum of Squares is: ", sum(results[:]))

    #Stop Timer
    end = time.time()

    #Output Total Time
    print('Time taken in seconds -', end - start)
```

Running Our First Program

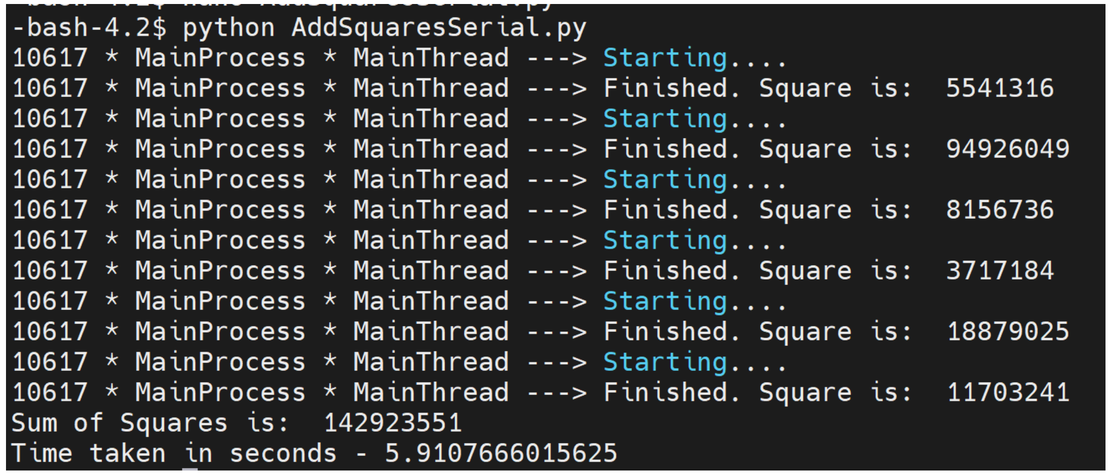
The time as come to run our first real program.

First take a look in the file:

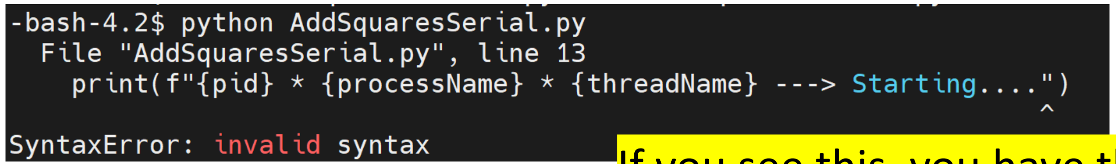
```
$more AddSquaresSerial.py
```

Now run the file

```
$python AddSquaresSerial.py
```



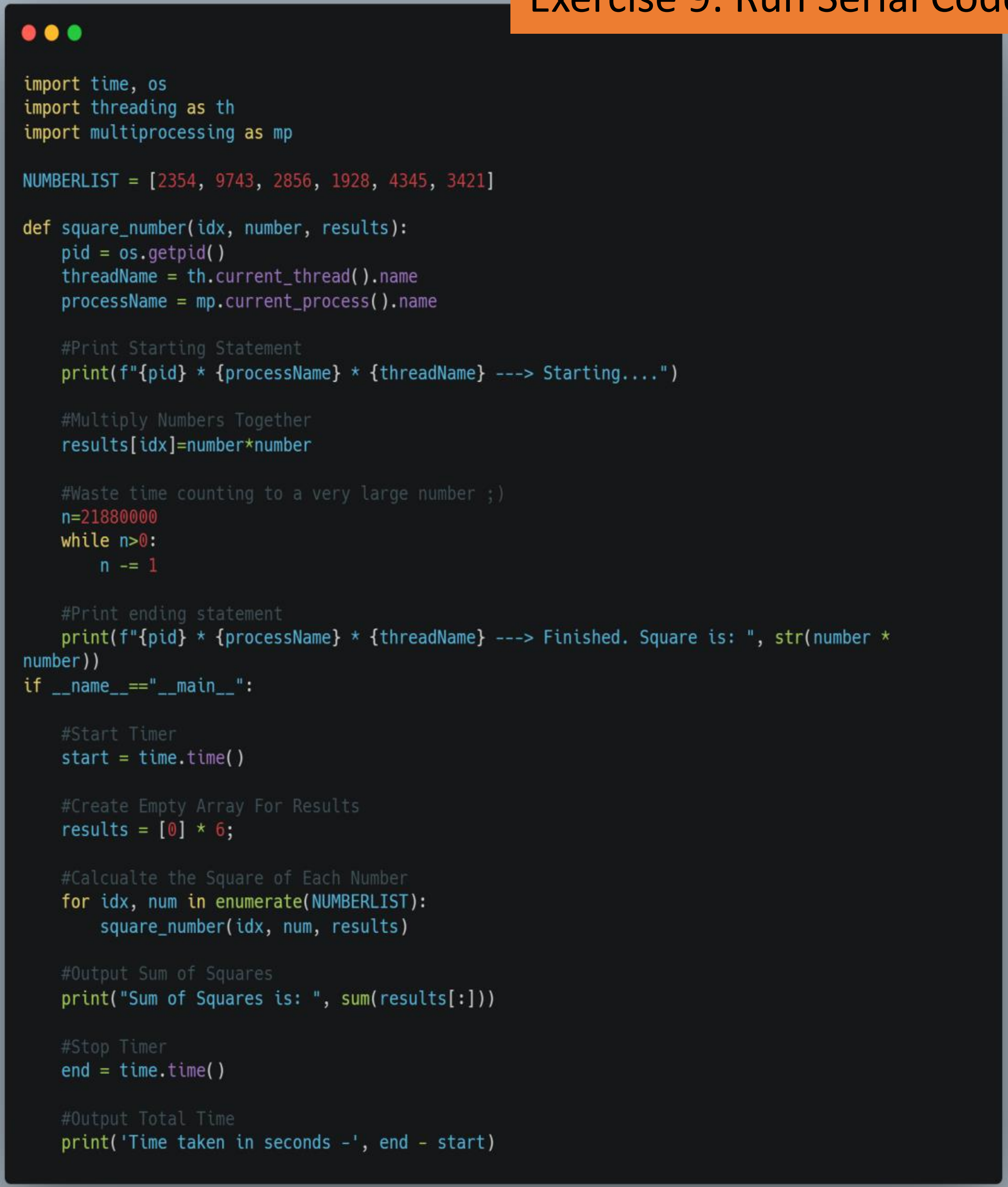
```
-bash-4.2$ python AddSquaresSerial.py
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  5541316
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  94926049
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  8156736
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  3717184
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  18879025
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  11703241
Sum of Squares is:  142923551
Time taken in seconds - 5.9107666015625
```



```
-bash-4.2$ python AddSquaresSerial.py
  File "AddSquaresSerial.py", line 13
    print(f"{pid} * {processName} * {threadName} ---> Starting....")
                                                                   ^
SyntaxError: invalid syntax
```

If you see this, you have the wrong version of python

Exercise 9: Run Serial Code



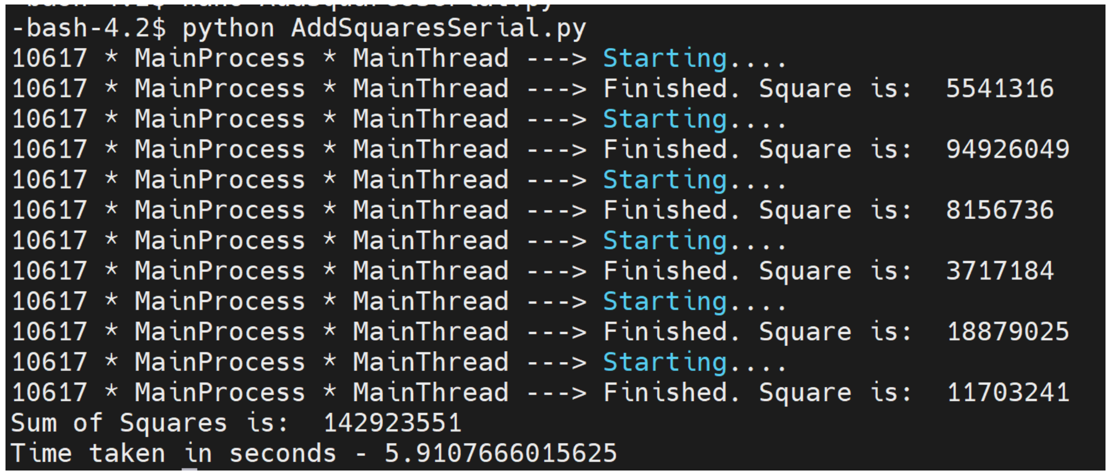
```python
import time, os
import threading as th
import multiprocessing as mp

NUMBERLIST = [2354, 9743, 2856, 1928, 4345, 3421]

def square_number(idx, number, results):
    pid = os.getpid()
    threadName = th.current_thread().name
    processName = mp.current_process().name

    #Print Starting Statement
    print(f"{pid} * {processName} * {threadName} ---> Starting....")

    #Multiply Numbers Together
    results[idx]=number*number

    #Waste time counting to a very large number ;)
    n=21880000
    while n>0:
        n -= 1

    #Print ending statement
    print(f"{pid} * {processName} * {threadName} ---> Finished. Square is: ", str(number *
number))
if __name__=="__main__":

    #Start Timer
    start = time.time()

    #Create Empty Array For Results
    results = [0] * 6;

    #Calcualte the Square of Each Number
    for idx, num in enumerate(NUMBERLIST):
        square_number(idx, num, results)

    #Output Sum of Squares
    print("Sum of Squares is: ", sum(results[:]))

    #Stop Timer
    end = time.time()

    #Output Total Time
    print('Time taken in seconds -', end - start)
```

Running Our First Program

Our program just calculated exactly what we calculated in class. Using just one CPU our program calculated the square of each of our six numbers (one at a time on the MainProcess) and then summed the results.

Each number took around 1 second to square for a total time of about 6 seconds.

Q. Where did this program run?



```
-bash-4.2$ python AddSquaresSerial.py
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  5541316
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  94926049
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  8156736
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  3717184
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  18879025
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  11703241
Sum of Squares is:  142923551
Time taken in seconds - 5.9107666015625
```

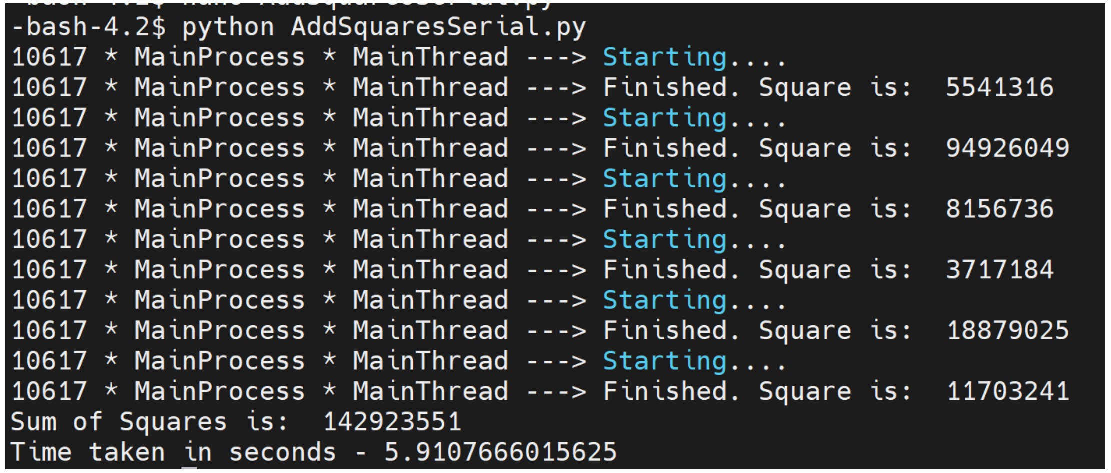
Running Our First Program

Our program just calculated exactly what we calculated in class. Using just one CPU our program calculated the square of each of our six numbers (one at a time on the MainProcess) and then summed the results.

Each number took around 1 second to square for a total time of about 6 seconds.

Q. Where did this program run?

A. On the login nodes (Either login-0 or login-1).



```
-bash-4.2$ python AddSquaresSerial.py
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  5541316
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  94926049
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  8156736
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  3717184
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  18879025
10617 * MainProcess * MainThread ---> Starting....
10617 * MainProcess * MainThread ---> Finished. Square is:  11703241
Sum of Squares is:  142923551
Time taken in seconds - 5.9107666015625
```

There are 80 people in this class and 32 cores available on the login node! We must be careful! If we all tried running this program at once we be using more cores than available.

AS A WARNING, DO NOT GET IN THE HABIT OF RUNNING CODE ON THE LOGIN NODES, THEY ARE NOT DESIGNED FOR COMPUTE WORKLOADS.

Earlier we said that Pronghorn has 108 nodes with 3,456 CPU cores.

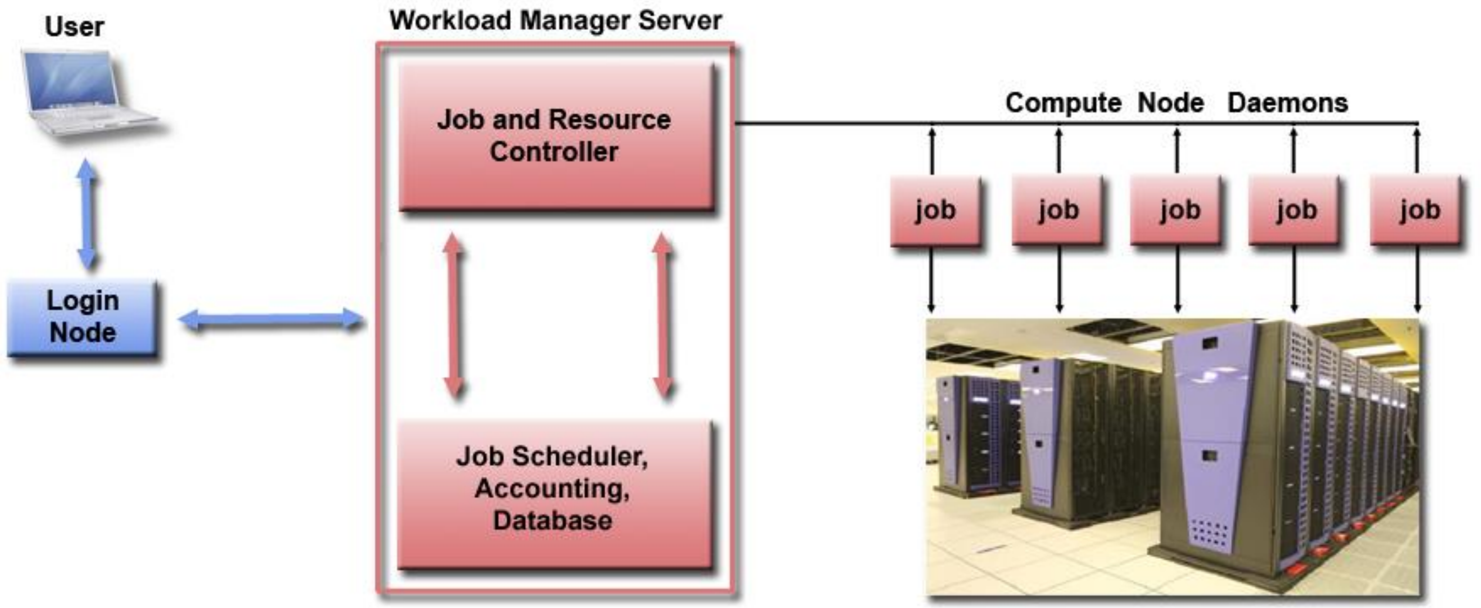
```
$scontrol show nodes
```

How do we access these nodes and how do we ensure we don't all try to use them at once?

Introduction to the SLURM scheduler

SLURM is what is known as a workload manager. The SLURM scheduler will let us access all the nodes on Pronghorn (if we are allowed/have paid for access) and schedule all the jobs sent to it by all the users ensuring the resources are available.

SLURM takes the vast number of different jobs sent to it by all users in the system, reserves "resources" (# of nodes per job, # of cores per node, memory per job), and then executes the jobs based on the user or association's priority.



A "job" is basically the top level of what you are trying to accomplish -- a workflow, set of commands/programs to run, etc. Typically, we define a single job at a time and submit it to the SLURM system.

Running Our First Scheduled Program

For this class you are part of the grad778 association, and we have two nodes available for our use during this class. To submit a job to one of these nodes we can use the 'srun' command. Try typing out this:

```
$srun --partition=cpu-core-0 --account=cpu-s5-grad_778-1 python AddSquaresSerial.py
```

$\underbrace{\texttt{--partition=cpu-core-0}}_{\text{Which nodes to use}}\ \underbrace{\texttt{--account=cpu-s5-grad\_778-1}}_{\text{Who is paying?}}\ \underbrace{\texttt{python AddSquaresSerial.py}}_{\text{What we want it to do?}}$

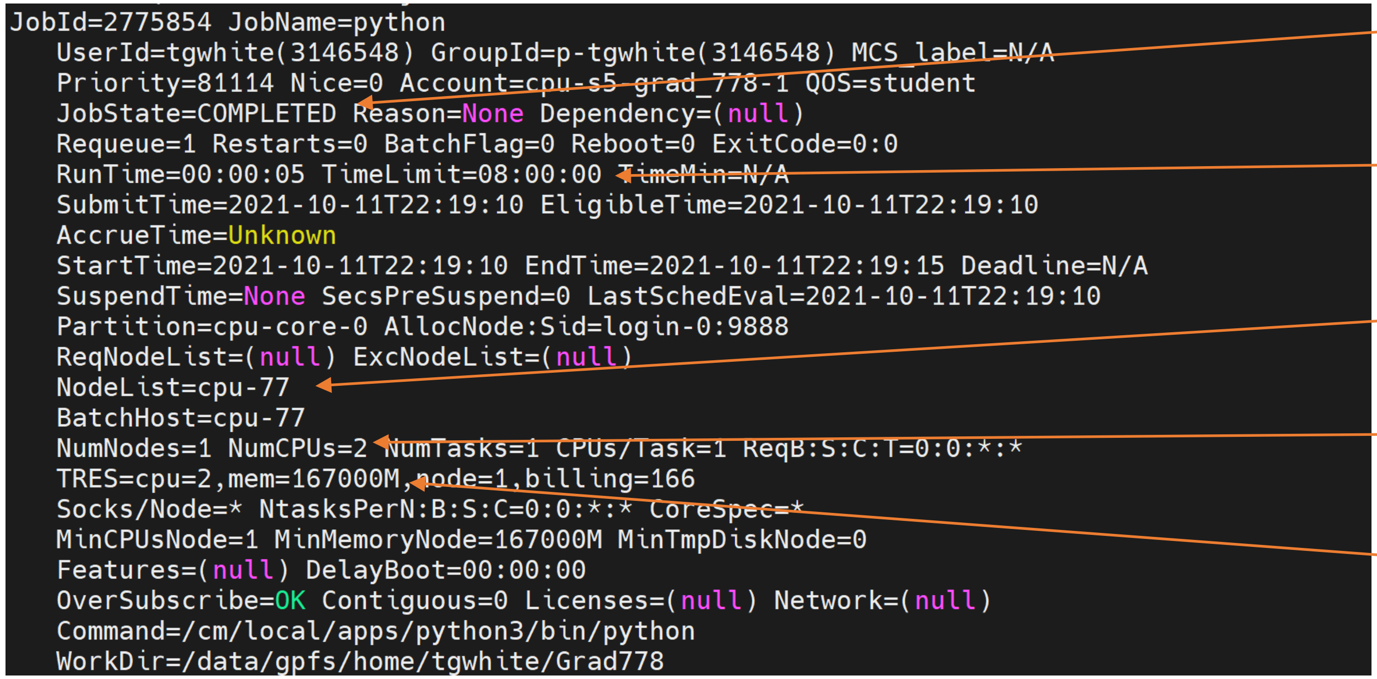
Hands up if your program ran (i.e., it printed the sum of squares to the screen)

Running Our First Scheduled Program

For this class you are part of the grad778 association, and we have two nodes available for our use during this class. To submit a job to one of these nodes we can use the ‘srun’ command. Try typing out this:

$$\texttt{\$srun } \underbrace{\texttt{--partition=cpu-core-0}}_{\text{Which nodes to use}} \; \underbrace{\texttt{--account=cpu-s5-grad\_778-1}}_{\text{Who is paying?}} \; \underbrace{\texttt{python AddSquaresSerial.py}}_{\text{What we want it to do?}}$$

Hands up if your program ran (i.e., it printed the sum of squares to the screen)



```
JobId=2775854 JobName=python
   UserId=tgwhite(3146548) GroupId=p-tgwhite(3146548) MCS_label=N/A
   Priority=81114 Nice=0 Account=cpu-s5-grad_778-1 QOS=student
   JobState=COMPLETED Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=0 Reboot=0 ExitCode=0:0
   RunTime=00:00:05 TimeLimit=08:00:00 TimeMin=N/A
   SubmitTime=2021-10-11T22:19:10 EligibleTime=2021-10-11T22:19:10
   AccrueTime=Unknown
   StartTime=2021-10-11T22:19:10 EndTime=2021-10-11T22:19:15 Deadline=N/A
   SuspendTime=None SecsPreSuspend=0 LastSchedEval=2021-10-11T22:19:10
   Partition=cpu-core-0 AllocNode:Sid=login-0:9888
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=cpu-77
   BatchHost=cpu-77
   NumNodes=1 NumCPUs=2 NumTasks=1 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
   TRES=cpu=2,mem=167000M,node=1,billing=166
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=1 MinMemoryNode=167000M MinTmpDiskNode=0
   Features=(null) DelayBoot=00:00:00
   OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/cm/local/apps/python3/bin/python
   WorkDir=/data/gpfs/home/tgwhite/Grad778
```

Last week I was able to run this command . . .

It took ~6 seconds to complete but we asked for resources for 8 hours!

The job ran on node cpu-77 (not the login node)

Reserved 2 CPUs

Reserved a HUGE 167 GB of memory

If your job did not run, it was because we asked for far more memory than available right now!

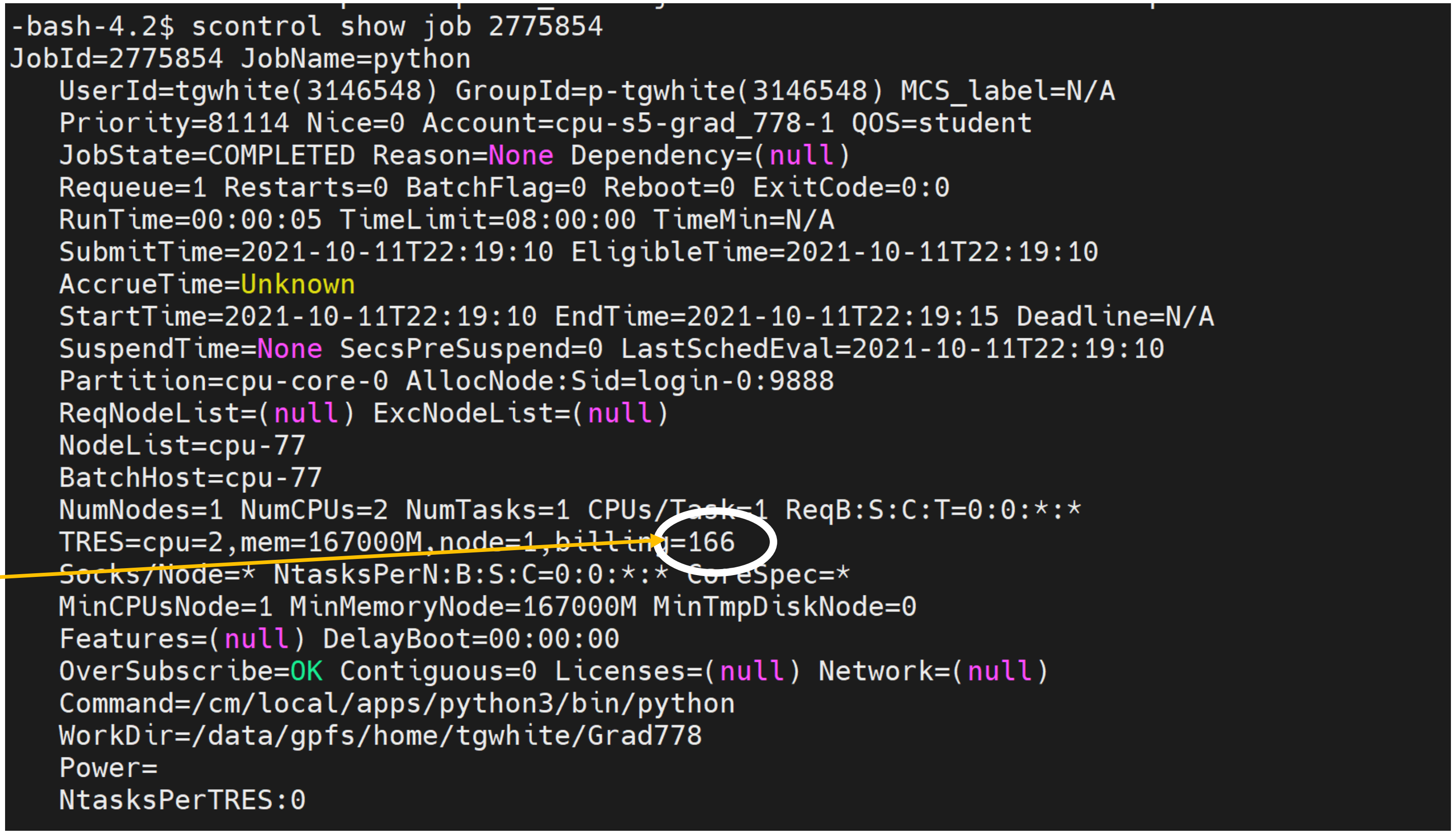
Introduction to the SLURM scheduler

Let’s make note of a two things:

1. We were given two CPUs and a huge amount of memory because those are the defaults set in SLURM on Pronghorn.

2. Even though our program is written serially (i.e., doesn’t take advantage of multiple CPUs or, in this case, all that memory) we still prevented other users from using them. If we had been paying for these resources, we would be charged ~80 times as much as we needed to pay.

Be careful with default settings – they can cost you (or your advisor) money



```
-bash-4.2$ scontrol show job 2775854
JobId=2775854 JobName=python
   UserId=tgwhite(3146548) GroupId=p-tgwhite(3146548) MCS_label=N/A
   Priority=81114 Nice=0 Account=cpu-s5-grad_778-1 QOS=student
   JobState=COMPLETED Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=0 Reboot=0 ExitCode=0:0
   RunTime=00:00:05 TimeLimit=08:00:00 TimeMin=N/A
   SubmitTime=2021-10-11T22:19:10 EligibleTime=2021-10-11T22:19:10
   AccrueTime=Unknown
   StartTime=2021-10-11T22:19:10 EndTime=2021-10-11T22:19:15 Deadline=N/A
   SuspendTime=None SecsPreSuspend=0 LastSchedEval=2021-10-11T22:19:10
   Partition=cpu-core-0 AllocNode:Sid=login-0:9888
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=cpu-77
   BatchHost=cpu-77
   NumNodes=1 NumCPUs=2 NumTasks=1 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
   TRES=cpu=2,mem=167000M,node=1,billing=166
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=1 MinMemoryNode=167000M MinTmpDiskNode=0
   Features=(null) DelayBoot=00:00:00
   OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/cm/local/apps/python3/bin/python
   WorkDir=/data/gpfs/home/tgwhite/Grad778
   Power=
   NtasksPerTRES:0
```

Exercise 11: Run Serial Code Using SLURM (attempt 2)

Introduction to the SLURM scheduler

NOTE: YOU WILL TYPICALLY NOT USE THE `--reservation` FLAG. We are using that for the class

It's important to request the correct/appropriate number of resources so that:

a) You/your advisor doesn't pay unnecessarily
b) We don't prevent other users from accessing resources they might need

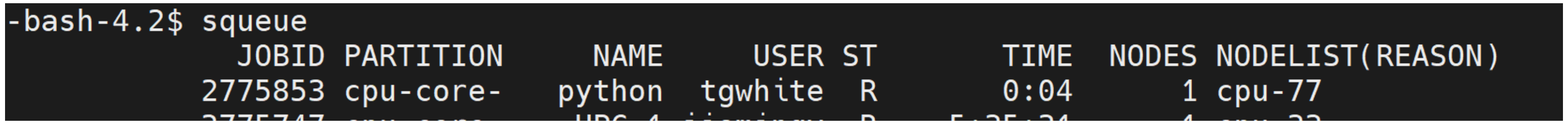
We can add arguments to the srun command requesting a more modest 10 MB of memory and just one CPU.

```
$srun --partition=cpu-core-0 --account=cpu-s5-grad_778-1 --mem=10M --tasks 1 --cpus-per-task=1
--time=00:01:00 --reservation=cpu-s5-grad_778-1_45 python AddSquaresSerial.py
```

Notice that the job doesn't print out the square as it calculates it for each number. Instead, all the output is printed to the screen at the end. It is important to note that these jobs are not meant to be run interactively (i.e., watched). In fact, you may even have had to wait for your job to begin.

While the program is running, try quickly typing squeue into the second terminal/SSH connection. You will have to be quick; it's only running for about 6 seconds.

```
$squeue –u [netid]
```



```
-bash-4.2$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2775853 cpu-core-   python  tgwhite  R       0:04      1 cpu-77
```

In this case, I can see that my job was number 2775853 and ran on node cpu-77. Notice, this isn't the login node anymore.

If you were fast enough to find your job ID you can get lots of extra detail on your job:

```
$scontrol show job [jobID]
```

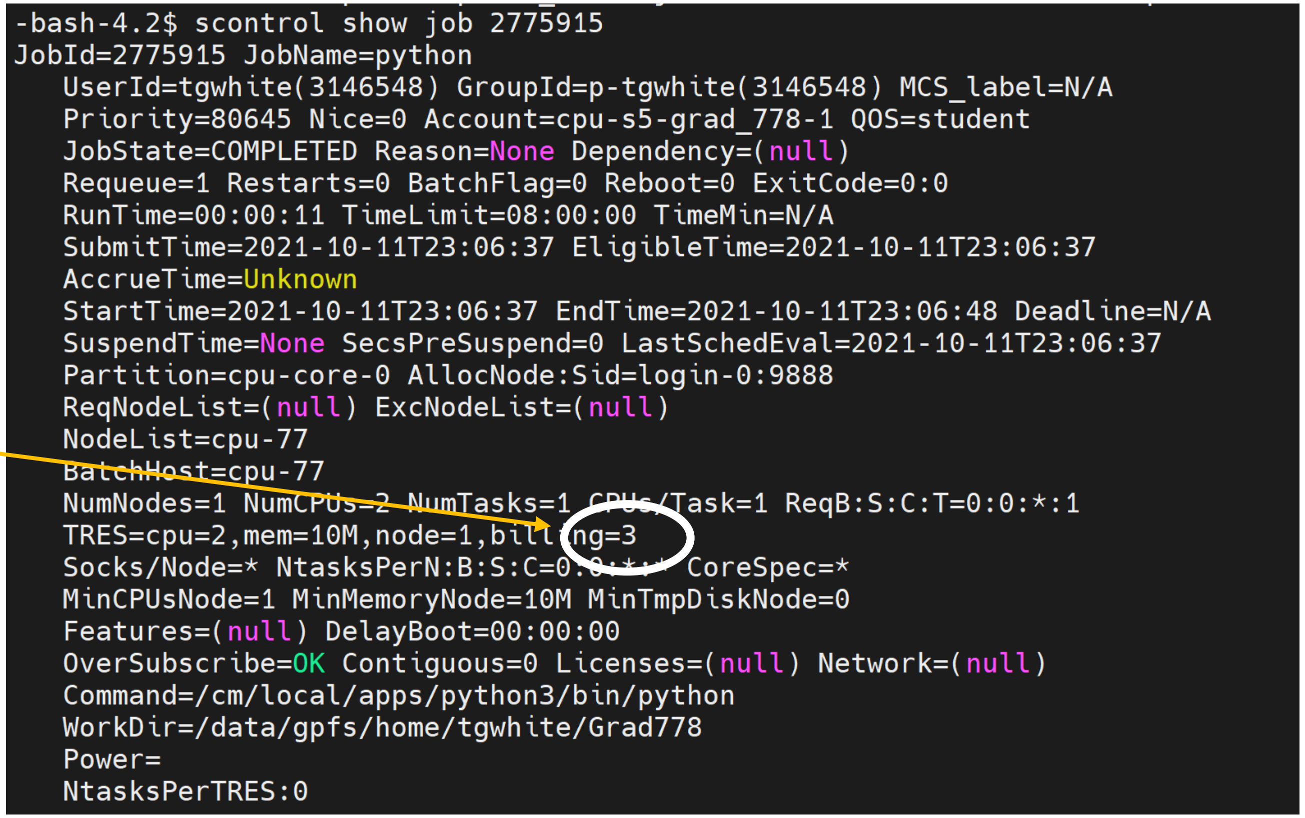
What do you notice about the job?

Introduction to the SLURM scheduler

Notice that we were still given two CPUs. This is because each physical core is made up of two logical cores. One physical CPU will be presented as two logical cores (more on this in the next lecture).

Our billing number reduced from 166 to 3. Our job is now around 50 times cheaper because we have asked for an appropriate amount of resources.

However, our srun command is starting to get VERY long . . . .



```
-bash-4.2$ scontrol show job 2775915
JobId=2775915 JobName=python
   UserId=tgwhite(3146548) GroupId=p-tgwhite(3146548) MCS_label=N/A
   Priority=80645 Nice=0 Account=cpu-s5-grad_778-1 QOS=student
   JobState=COMPLETED Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=0 Reboot=0 ExitCode=0:0
   RunTime=00:00:11 TimeLimit=08:00:00 TimeMin=N/A
   SubmitTime=2021-10-11T23:06:37 EligibleTime=2021-10-11T23:06:37
   AccrueTime=Unknown
   StartTime=2021-10-11T23:06:37 EndTime=2021-10-11T23:06:48 Deadline=N/A
   SuspendTime=None SecsPreSuspend=0 LastSchedEval=2021-10-11T23:06:37
   Partition=cpu-core-0 AllocNode:Sid=login-0:9888
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=cpu-77
   BatchHost=cpu-77
   NumNodes=1 NumCPUs=2 NumTasks=1 CPUs/Task=1 ReqB:S:C:T=0:0:*:1
   TRES=cpu=2,mem=10M,node=1,billing=3
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=1 MinMemoryNode=10M MinTmpDiskNode=0
   Features=(null) DelayBoot=00:00:00
   OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/cm/local/apps/python3/bin/python
   WorkDir=/data/gpfs/home/tgwhite/Grad778
   Power=
   NtasksPerTRES:0
```

```
$srun --partition=cpu-core-0 --account=cpu-s5-grad_778-1 --mem=10M --tasks 1 --cpus-per-task=1
--time=00:01:00 --reservation=cpu-s5-grad_778-1_45 python AddSquaresSerial.py
```

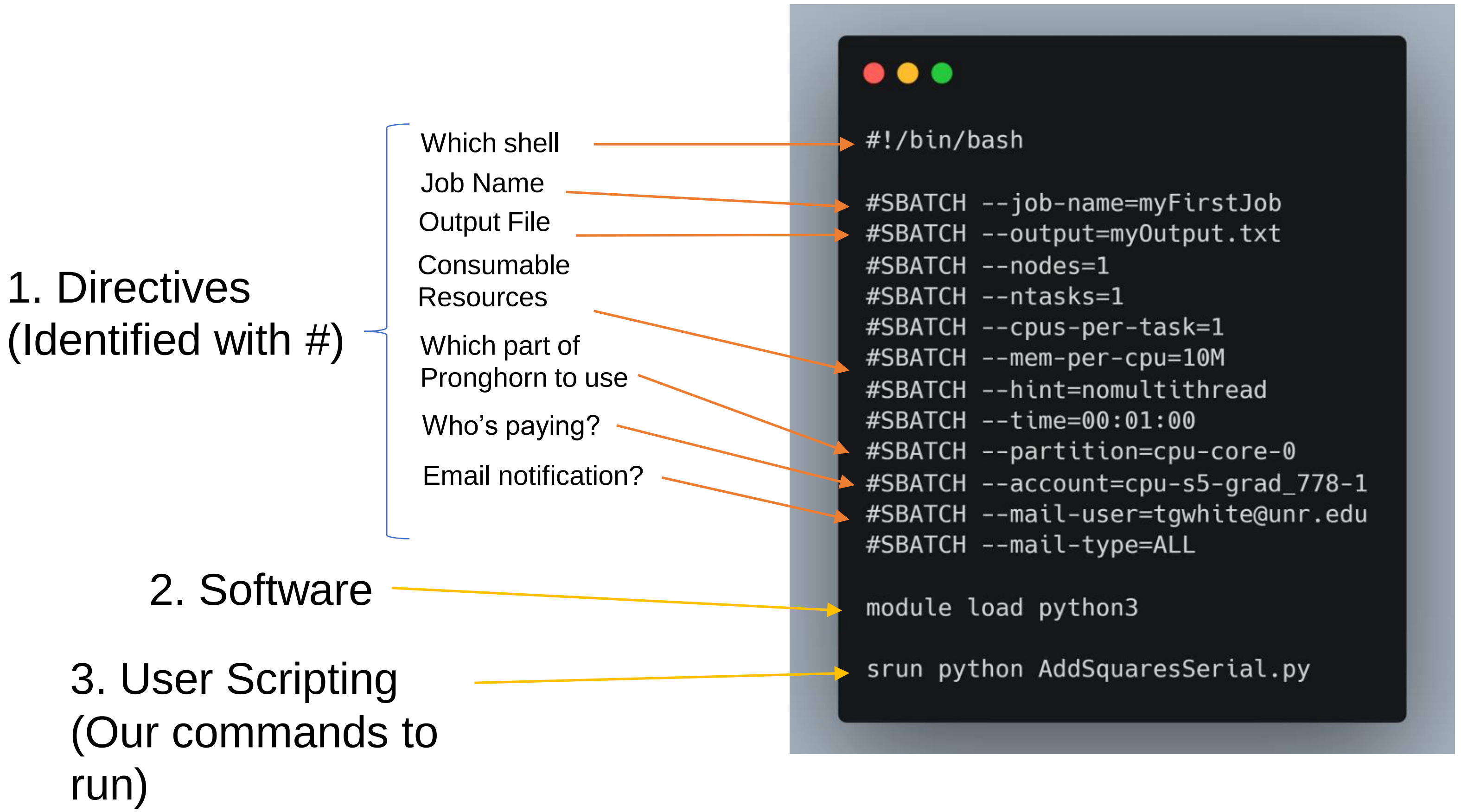
Break

SLURM Batch (SBATCH) Scripts

Typically, we won't use srun to directly run jobs on Pronghorn. This is because srun is used to allocate interactive jobs. i.e., we were still waiting on the command line for the job to start/run just like running a normal program. As a general rule, Pronghorn is a BATCH system, which means you will focus on jobs that **do not require user interaction and** will often be deferred (run at some time in the future). Additionally, the command line arguments are starting to get very long!

Instead, we'll place the srun command inside a file (or job script) with all our instructions to the scheduler. SLURM uses a "job script" written in any interpreted language and uses "#" as the comment character -- typically we'll use the "bash" language to create a job. The job script follows a very specific format that you will get familiar with.

Notice, we specify an output file. In theory, we don't know when this job will be run, so it is useless printing text to the screen (we could be in bed). Instead, the text that would normally be printed to the screen is printed to this file.



1. Directives (Identified with #)
Which shell
Job Name
Output File
Consumable Resources
Which part of Pronghorn to use
Who's paying?
Email notification?
2. Software
3. User Scripting (Our commands to run)

```
#!/bin/bash

#SBATCH --job-name=myFirstJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=10M
#SBATCH --hint=nomultithread
#SBATCH --time=00:01:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-1
#SBATCH --mail-user=tgwhite@unr.edu
#SBATCH --mail-type=ALL

module load python3

srun python AddSquaresSerial.py
```

SLURM Batch (SBATCH) Scripts

--job-name: a name for your job that will appear in the queue

--output: sends standard output from a job into a filename(s) of your choice. Note you can also have your script do this explicitly.

--error: sends standard error from a job into a filename(s) of your choice. Note you can also have your script do this explicitly.

--nodes: how many nodes can be used for the job?

--ntasks: how many unique "tasks" will your script run?

--cpus-per-task: for one task, how many cpus are allocated to it? If your task isn't "internally parallelized", this should be set to 1 cpu per task.

--mem-per-cpu: the maximum needed memory PER CPU. The nodes we are using have ~256GB RAM total (232GB is usable), and 32 physical cores, so evenly distributed we have ~ 8GB per cpu. You can ask for more, however it can cost you.

--hint=nomultithread: if your process does not take advantage of simultaneous multithreading (SMT), it might make understanding code execution easier to disable it.

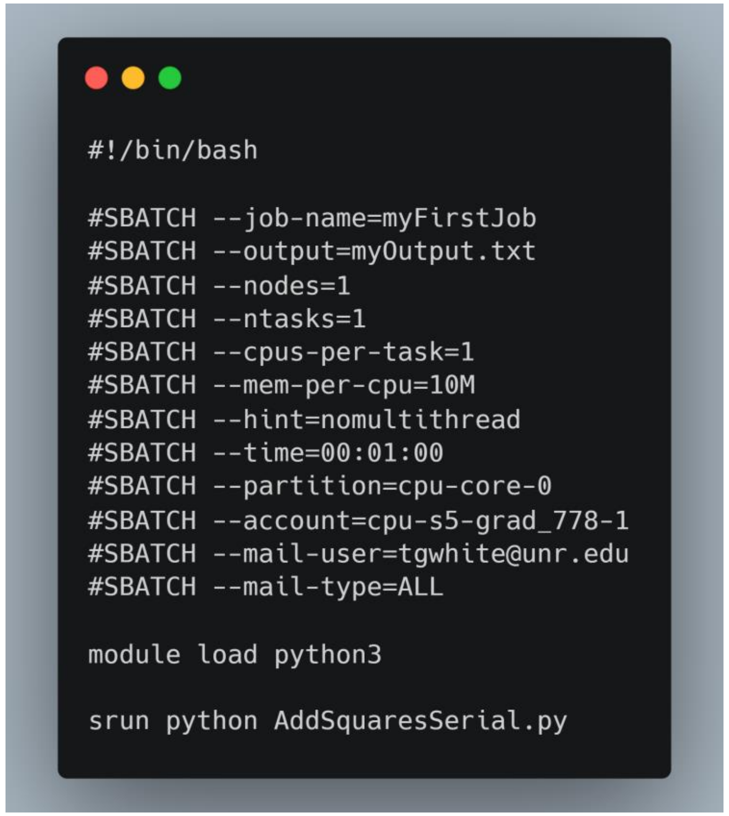
--time: the maximum amount of time your script will take, after which it will be killed. As a general rule, be generous with this time. The more accurate you are, the faster your job will begin running.

--partition: the "queue" name to use.

--account: the account your jobs are being charged to.

--mail-ss to send notifications to (remove this if you don't want emails)

--mail-type=what kind of notifications to email



```
#!/bin/bash

#SBATCH --job-name=myFirstJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=10M
#SBATCH --hint=nomultithread
#SBATCH --time=00:01:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-1
#SBATCH --mail-user=tgwhite@unr.edu
#SBATCH --mail-type=ALL

module load python3

srun python AddSquaresSerial.py
```

Running Our First Scheduled Program

NOTE: YOU WILL TYPICALLY NOT USE THE `--reservation` FLAG. We are using that for the class

1. Edit the simpleJobscript.sl file you downloaded earlier.

```
$nano exampleBatchScript.sl
```

2. Give the job an identifying name.
3. Enter your own email address.
4. Save and exit the file [Ctrl+O], [Ctrl+X]

Now run the job using the Pronghorn scheduler.

```
$sbatch --reservation=cpu-s5-grad_778-1_45 exampleBatchScript.sl
```

You should see it say "Submitted batch job XXXXXX". You should receive an email when it is complete.

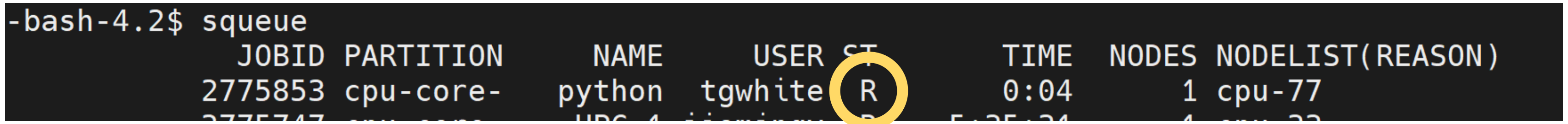
After you have submitted your job, look in the myOutput.txt. Do you see the results of the program?

Try to find your job in the queue. Use the scontrol command to find the details of the job.

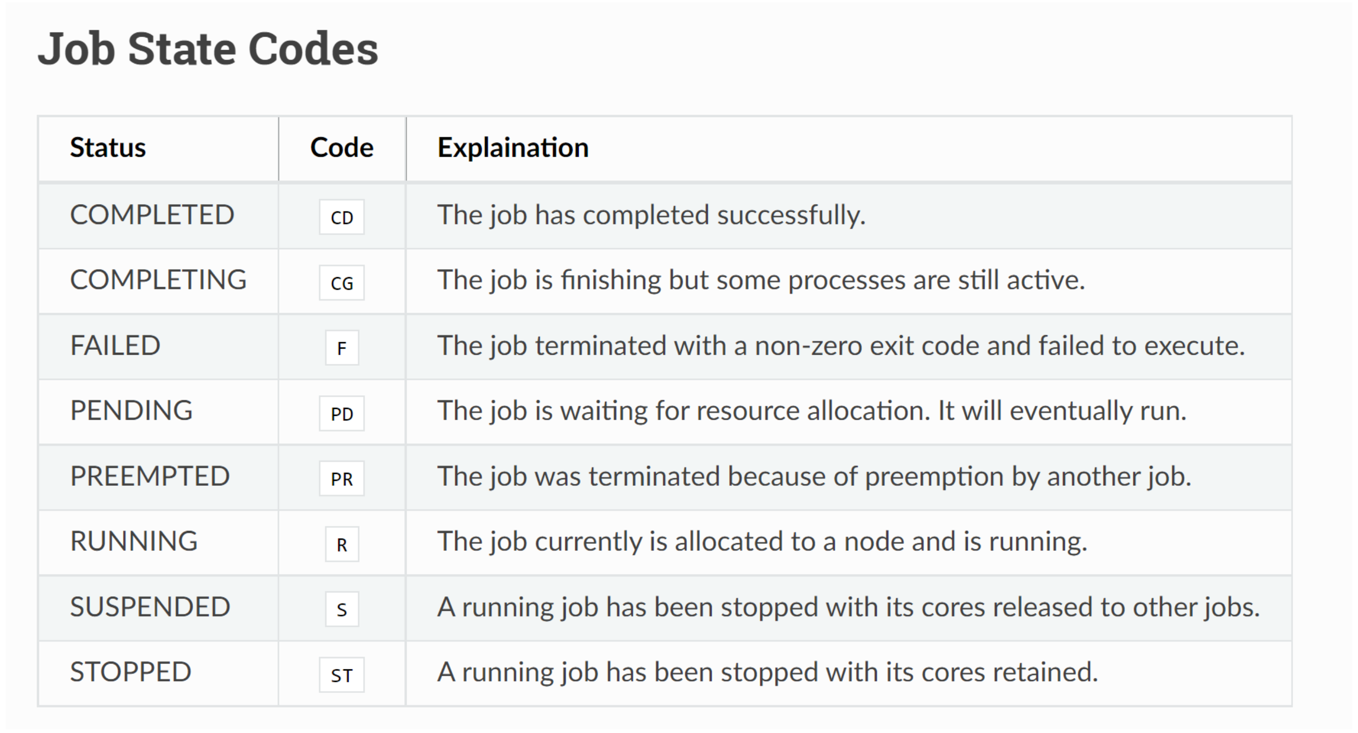
```
$squeue –u <netid>

$scontrol show job <jobID>
```

Important SLURM Commands

We have already seen that we can use squeue to get details of our jobs in the queue. The 'ST' or status column provides information on the status of the job:



```
-bash-4.2$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2775853 cpu-core-   python  tgwhite  R       0:04      1 cpu-77
```



Job State Codes

Status	Code	Explaination
COMPLETED	CD	The job has completed successfully.
COMPLETING	CG	The job is finishing but some processes are still active.
FAILED	F	The job terminated with a non-zero exit code and failed to execute.
PENDING	PD	The job is waiting for resource allocation. It will eventually run.
PREEMPTED	PR	The job was terminated because of preemption by another job.
RUNNING	R	The job currently is allocated to a node and is running.
SUSPENDED	S	A running job has been stopped with its cores released to other jobs.
STOPPED	ST	A running job has been stopped with its cores retained.

If the job is PENDING or STOPPED we can use the `$scontrol show job [jobID]` command to find the reason it hasn't run yet. A full list of reasons is available on the SLURM website: https://slurm.schedmd.com/squeue.html#lbAF

‘Manual’ Process-Based Parallelism

Our previous example only did one thing at a time on one core. We are now going to look at an “embarrassingly parallel” job where we will run multiple processes in parallel. We want to think about this problem as a big "for" loop, where the iterations of the loop MIGHT run in parallel. Instead of submitting our job once, we are going to submit it four (4) times and watch the queue.

Now, copy and paste the same sbatch command into the command line of your session all at the same time (copy all four lines and paste them all into ssh):

```
sbatch --reservation=cpu-s5-grad_778-1_45 exampleBatchScript.sl
sbatch --reservation=cpu-s5-grad_778-1_45 exampleBatchScript.sl
sbatch --reservation=cpu-s5-grad_778-1_45 exampleBatchScript.sl
sbatch --reservation=cpu-s5-grad_778-1_45 exampleBatchScript.sl
```

Quickly watch the squeue. You MIGHT see them all start running simultaneously, but maybe not if the queue is heavily utilized. After 6s of running, each should finish running.

Congrats! You have parallel processed on a HPC! Each of those jobs needed 6s of processing, so you needed ~24s of total CPU time to finish all of them. BUT, if you happened to get all four of those jobs to execute at the same time, AND those jobs started running simultaneously, then it only took 6s of "wall" (human) time to do 24s of processing.

But you may have experienced either 1) a wait (while other jobs were running before yours were allowed to run), and/or 2) the jobs did not run exactly in parallel. In addition, as each process completed it overwrote the previous one ☹

If we had multiple python scripts, each squaring different numbers and writing to multiple files this might have been useful. TIP: If you can split your job into multiple tasks, you should.

Exercise 14: A ‘real’ Parallel Process

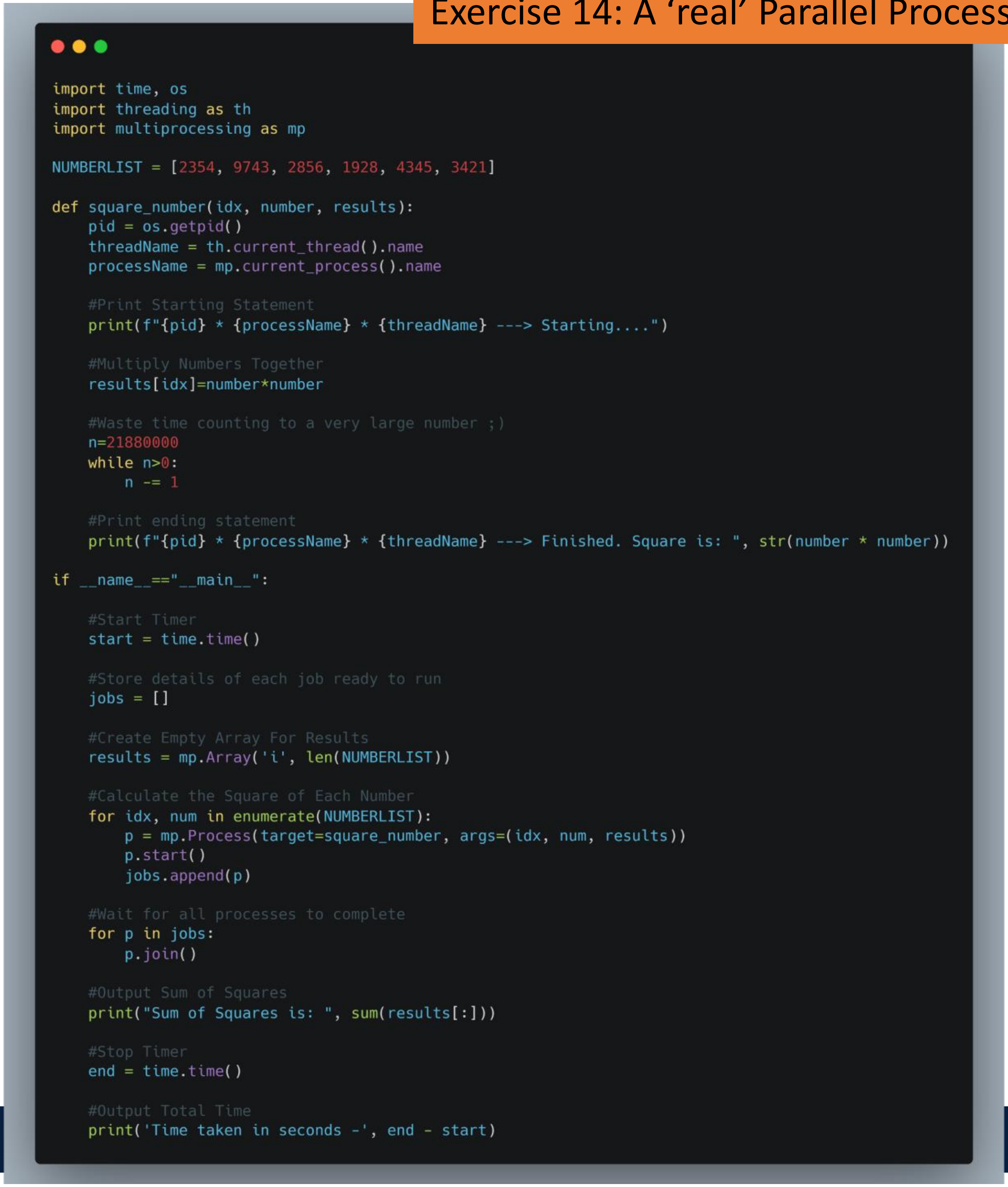
‘Real’ Parallelism

Now we move towards a more complex (but more common) problem, where we are performing a single task that is itself, multithreaded. A thread is part of a larger process but can be run concurrently and often share memory with each other. In this case, we are going to make a python program that creates a multiprocessor "pool" that spawns four "worker" threads. Each thread can square numbers, so this example is just like when we used multiple people to speed up out calculation on the board earlier.

Edit the SBATCH script so that it runs this new python program instead:

AddSquaresMultiThreading.py

Submit the job to the queue. Check the output. How fast did the program run?



```
import time, os
import threading as th
import multiprocessing as mp

NUMBERLIST = [2354, 9743, 2856, 1928, 4345, 3421]

def square_number(idx, number, results):
    pid = os.getpid()
    threadName = th.current_thread().name
    processName = mp.current_process().name

    #Print Starting Statement
    print(f"{pid} * {processName} * {threadName} ---> Starting....")

    #Multiply Numbers Together
    results[idx]=number*number

    #Waste time counting to a very large number ;)
    n=21880000
    while n>0:
        n -= 1

    #Print ending statement
    print(f"{pid} * {processName} * {threadName} ---> Finished. Square is: ", str(number * number))

if __name__=="__main__":

    #Start Timer
    start = time.time()

    #Store details of each job ready to run
    jobs = []

    #Create Empty Array For Results
    results = mp.Array('i', len(NUMBERLIST))

    #Calculate the Square of Each Number
    for idx, num in enumerate(NUMBERLIST):
        p = mp.Process(target=square_number, args=(idx, num, results))
        p.start()
        jobs.append(p)

    #Wait for all processes to complete
    for p in jobs:
        p.join()

    #Output Sum of Squares
    print("Sum of Squares is: ", sum(results[:]))

    #Stop Timer
    end = time.time()

    #Output Total Time
    print('Time taken in seconds -', end - start)
```

‘Real’ Parallelism

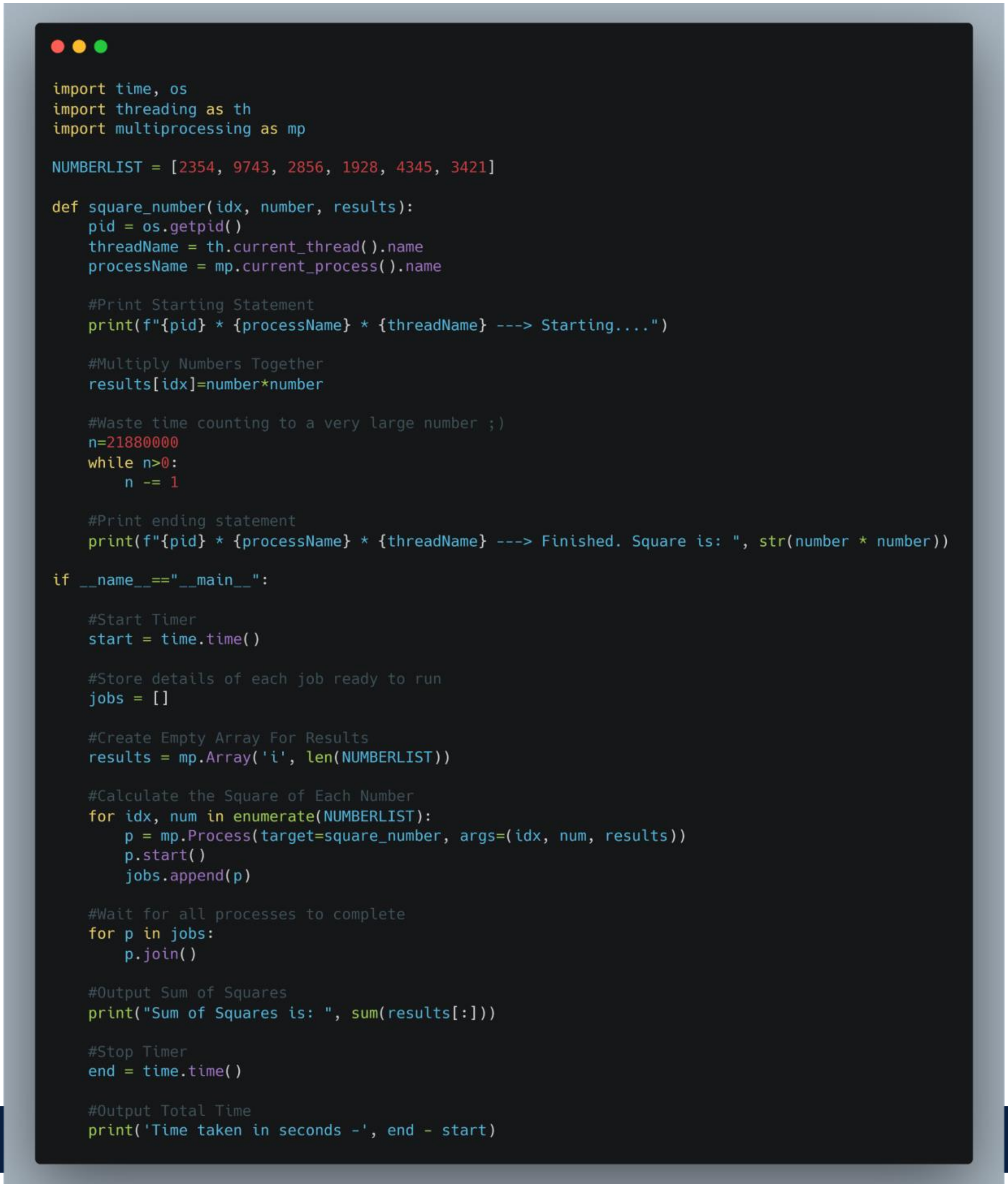
Now we move towards a more complex (but more common) problem, where we are performing a single task that is itself, multithreaded. A thread is part of a larger process but can be run concurrently and often share memory with each other. In this case, we are going to make a python program that creates a multiprocessor "pool" that spawns four "worker" threads. Each thread can square numbers, so this example is just like when we used multiple people to speed up out calculation on the board earlier.

Edit the SBATCH script so that it runs this new python program instead:

AddSquaresMultiThreading.py

Submit the job to the queue. Check the output. How fast did the program run?

Elapsed time was > 6 seconds! But our python script was supposed to use more CPUs to do the job faster. What happened? The SBATCH script did NOT make additional CPUs available to the python code. Let’s fix this!



```
import time, os
import threading as th
import multiprocessing as mp

NUMBERLIST = [2354, 9743, 2856, 1928, 4345, 3421]

def square_number(idx, number, results):
    pid = os.getpid()
    threadName = th.current_thread().name
    processName = mp.current_process().name

    #Print Starting Statement
    print(f"{pid} * {processName} * {threadName} ---> Starting....")

    #Multiply Numbers Together
    results[idx]=number*number

    #Waste time counting to a very large number ;)
    n=21880000
    while n>0:
        n -= 1

    #Print ending statement
    print(f"{pid} * {processName} * {threadName} ---> Finished. Square is: ", str(number * number))

if __name__=="__main__":

    #Start Timer
    start = time.time()

    #Store details of each job ready to run
    jobs = []

    #Create Empty Array For Results
    results = mp.Array('i', len(NUMBERLIST))

    #Calculate the Square of Each Number
    for idx, num in enumerate(NUMBERLIST):
        p = mp.Process(target=square_number, args=(idx, num, results))
        p.start()
        jobs.append(p)

    #Wait for all processes to complete
    for p in jobs:
        p.join()

    #Output Sum of Squares
    print("Sum of Squares is: ", sum(results[:]))

    #Stop Timer
    end = time.time()

    #Output Total Time
    print('Time taken in seconds -', end - start)
```

Exercise 15: How much faster can you make the job?

Playing with the Number of CPUs

Create a new SBATCH script that has the following characteristics:

- The script is named “multiprocessing.sl"
- The job name is “myMultiProcessingJob“
- The output filename is " myMultiProcessingJob_output.txt“
- It runs on 8 CPUs with 10MB of memory per CPU
- Disables multithreading
- Requests one minute of time
- Uses the partition and account available to this class
- Sends you an email when it is complete
- Loads python 3.7.5
- Runs AddSquaresMultiThreading.py

Vary the number of CPUs assigned to the task. Try out 1,2,3,4,5,6,8,16

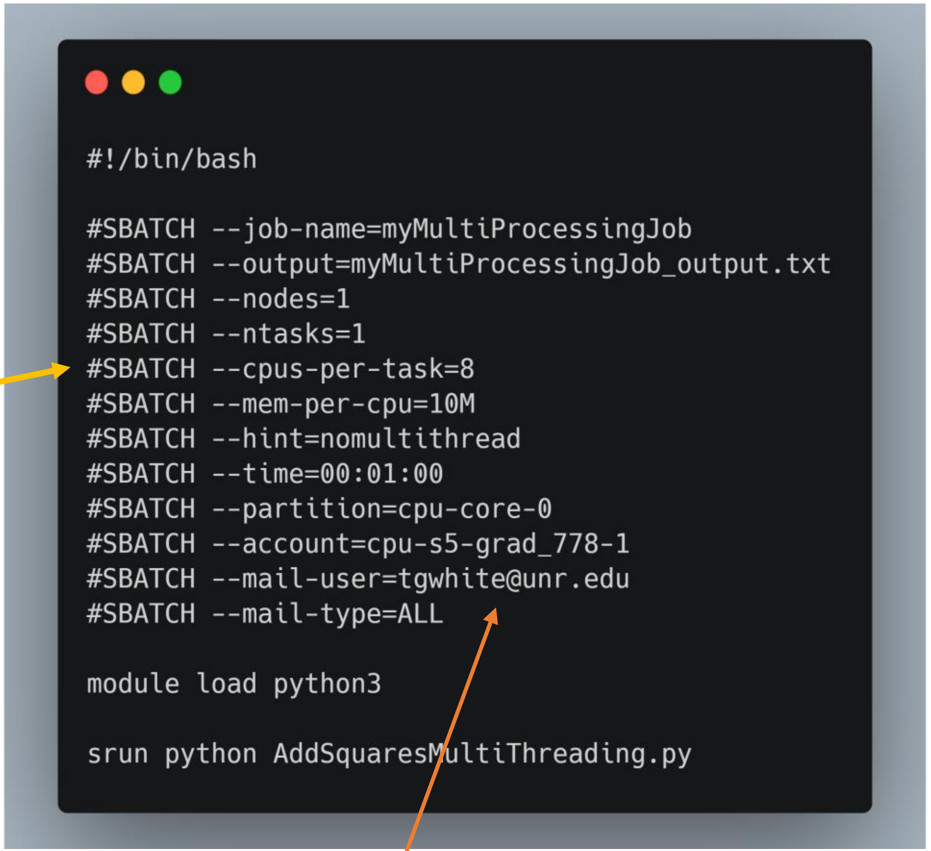
```
sbatch --reservation=cpu-s5-grad_778-1_45 multiprocessing.sl
```

Did each calculation occur on a different process? Check the output!

Create a plot showing how long each job took for each number of CPUs (you can use excel).

How much faster did our program run? Do you notice anything interesting about the curve?

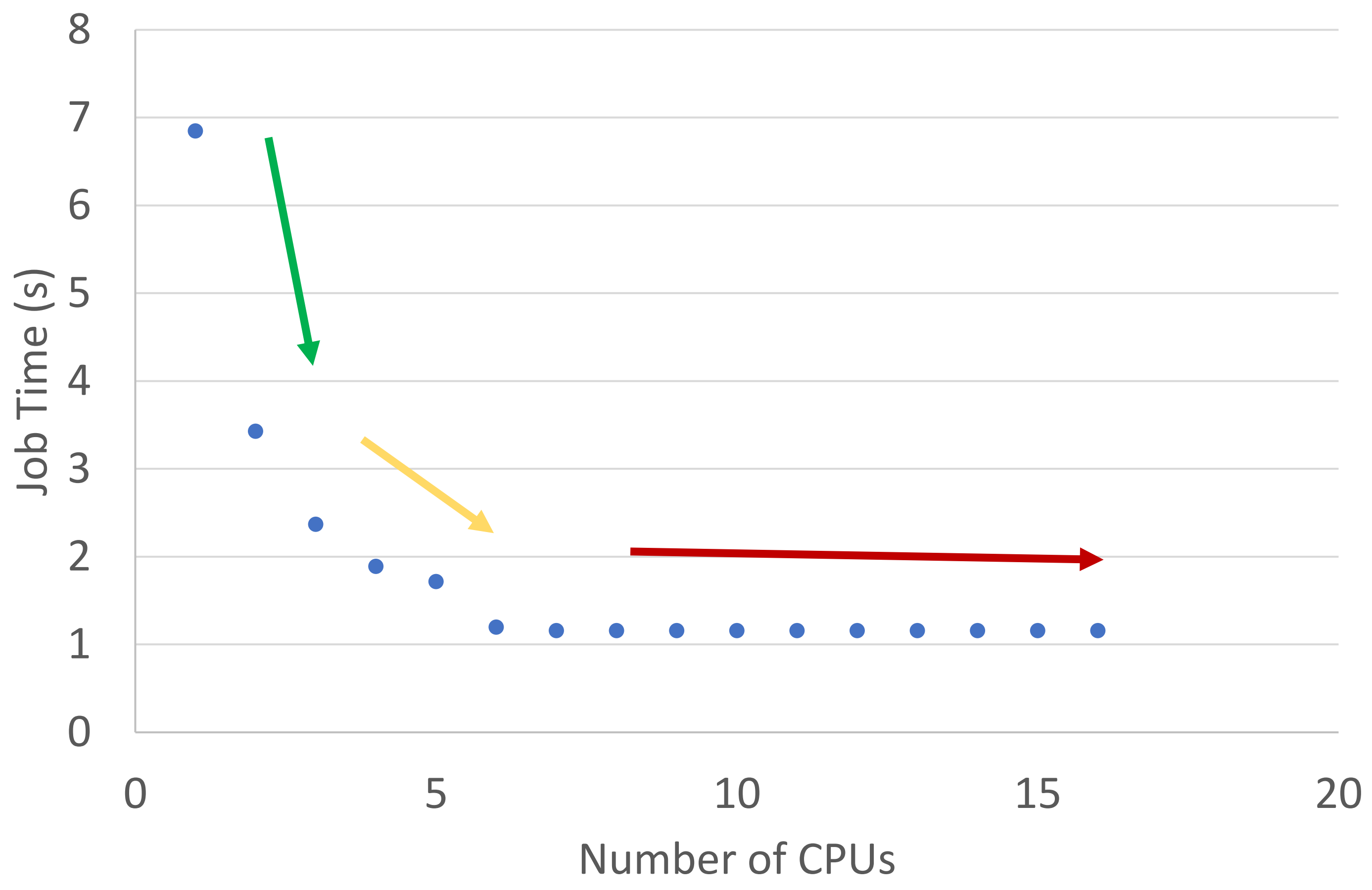
If you have time, compare this to a perfectly parallelizable curve (i.e., two CPUs takes half the time, three 1/3 of the time, etc.).



```
#!/bin/bash

#SBATCH --job-name=myMultiProcessingJob
#SBATCH --output=myMultiProcessingJob_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --mem-per-cpu=10M
#SBATCH --hint=nomultithread
#SBATCH --time=00:01:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-1
#SBATCH --mail-user=tgwhite@unr.edu
#SBATCH --mail-type=ALL

module load python3

srun python AddSquaresMultiThreading.py
```

Please don’t put my email address – I will get many many emails ;)

Job Time vs Number of Processors



Initially, adding CPUs made a huge difference to our time.

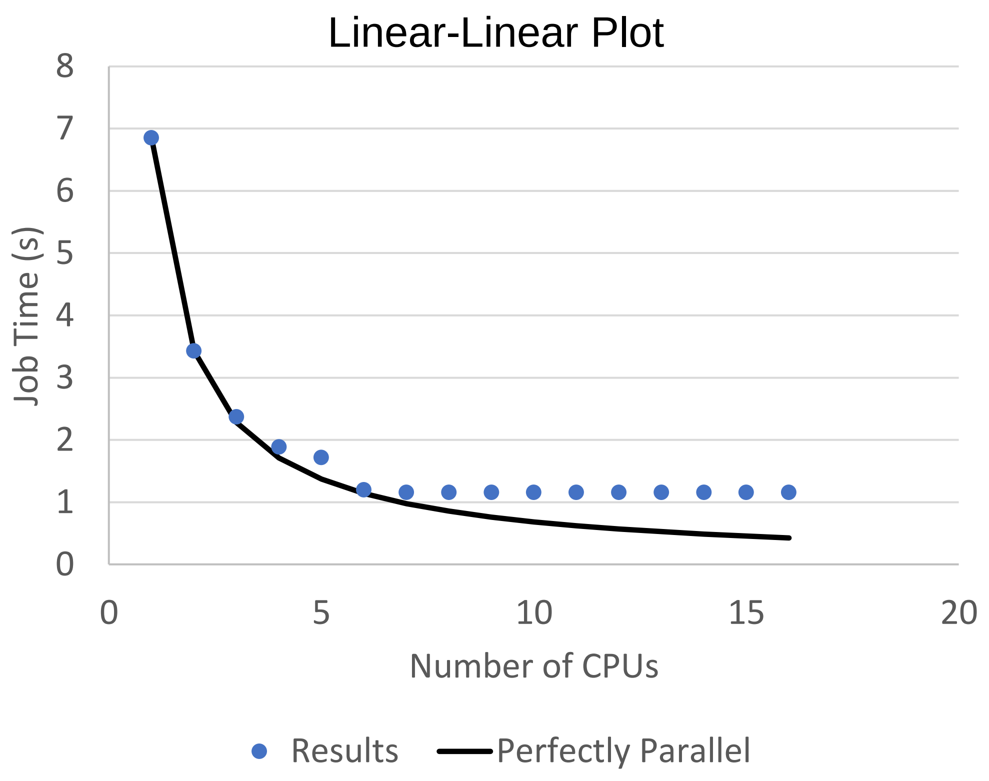
However, eventually we get diminishing returns.

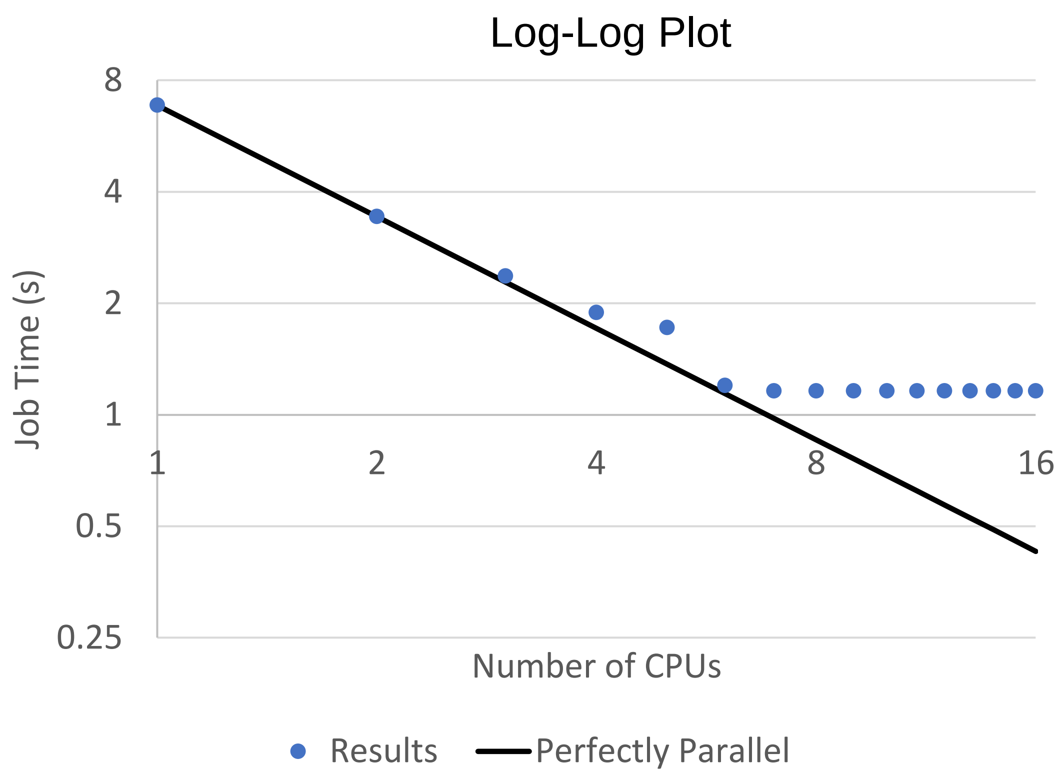
Above 6 CPUs there is no speed up. This is because we only had 6 tasks to accomplish, but we asked for resources to run more than 6 tasks. We ended up idling these extra CPUs because they were not being used. Why is this important? Computing resources are billed monthly. This will increase your computational costs with no gain in research output.

A key lesson from this exercise is **MATCH SLURM TO THE RESOURCES THAT YOUR CODE NEEDS**.

Job Time vs Number of Processors

All these features are easier to see on a Log-Log plot. On a Log-Log plot perfect parallelization appears as a straight line

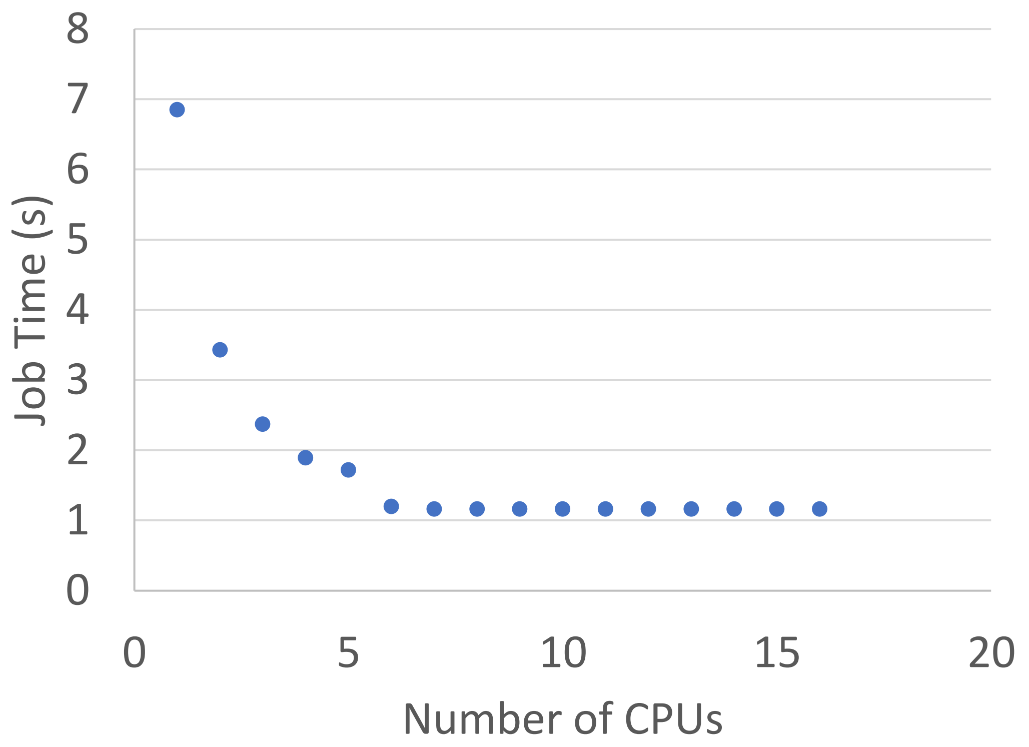




Notice that the point for 5 CPUs lies above the perfectly parallel line. In fact, we get very little speed up from going from 4 to 5 CPUs. This is because our problem naturally has six tasks. So with both 4 and 5 CPUs, it naturally takes two rounds of computation with several processors sat idle. We saw a similar concept earlier when we had people calculate the squares!

Quiz

In order to pass this class, please upload a file/image of your graph showing the decrease in time as we increase the number of cores. Hint: It should look like this:



Conclusions and What’s Next?

This class provides a very basic introduction to UNR’s High Performance Computer (Pronghorn).

As with most things, the best way to move forward increasing your knowledge at this point is to have a specific project/program/goal in mind and start working through it. There are three good ways to get support:

1. Come to the advanced Pronghorn Utilization class in 2 weeks (November 6, 2021)
2. Join the UNR RC slack channel and ask questions there — this is a community chat so you will get feedback from other researchers as well as the Pronghorn admins. To join the UNR RC slack, click this link and follow the instructions: https://unrrc.slack.com
3. Email hpc@unr.edu. This goes directly to the Pronghorn admins.
4. Attend the weekly "Hackathons" where the Pronghorn admins can give more hands-on support. Fridays from 2pm to 4pm on Slack.
5. Interact with other people using the system in your lab or department on Slack and ask questions!
6. Read the Slurm manuals: https://slurm.schedmd.com/
7. Read the manuals of your research software.

The Queue

You won't have access to cpu-s5-grad778-1 for much longer than the end of the semester (it is a partition set up for this course), so once that expires, you will be limited to 2-hour jobs, maximum of 64 CPUs on 1 node, and only 12 jobs in the queue. If your needs exceed this, please contact hpc@unr.edu for information on creating a Research/Student Research Association or for help gaining access to an existing Association.

As a campus employee/student, you have access to the free "sponsored" queue and your 50gb of home directory space. Remember the queueing limits:

Partition	Account	Max nodes	Max CPUs	Max jobs	Max runtime
`cpu-s6-test-0` (`"test"`)	`cpu-s6-test-0`	2	128	2	00:15:00
`cpu-s3-sponsored-0` (`"sponsored"`)	`cpu-s3-sponsored-0`	1	64	12	08:00:00
`cpu-s2-whitehedp-0` (`“renter”`)	`cpu-s2-core-0`	53	3392	51200	14 days

You pay per CPU-hour for this service

Service	Max Resources	Cost to user
High-Performance Storage	2000PiB	Pay-as-you-go monthly

You must pay per GB-month for this.

Next Lecture

Learning about the architecture of Pronghorn

Running larger and longer jobs

Running across multiple nodes (Distributed Storage, Message Passing Interface (MPI))

Multithreading vs Multiprocessing?

Advanced SLURM use (Jobs, Steps, and Tasks)

Installing/Compiling Software on Pronghorn (Singularity Containers)

Running Interactive Jobs