GRAD 778 - Elements of Research Computing
Module 9 - Batch Processing on Pronghorn
November 7, 2020
Instructor: Jonathan Greenberg
Contributors: John Anderson, Theo Hartsook, Adriana Parra, Sebastian Smith

Overview: Introduction to using the Pronghorn high performance supercomputer to perform large-scale, parallel computing tasks. Students will learn basic parallel computing concepts and learn to implement them using the SLURM job submission system.

Websites:

- https://www.unr.edu/research-computing/hpc

Zoom Meeting info:

- https://unr.zoom.us/j/92493493674

PLEASE MAKE SURE YOU CAN LOGIN TO PRONGHORN USING SSH.

ssh [netid]@pronghorn.rc.unr.edu

Table of Contents

I. Login, Setup and Getting Acquainted With Pronghorn

Notes: whenever something is in `Courier font`, this is something you should enter into the command line.

Requirements:

- SSH software (to connect to pronghorn).
    - Mac users can use "Terminal" (in your Applications/Utilities (or, if you want, grab iterm2: https://www.iterm2.com/)
    - Windows users should download/install "MobaXterm" **FREE** edition (https://mobaxterm.mobatek.net/)
- SCP or SFTP software (to transfer data to and from pronghorn)
- UNR NetID (all UNR employees)
- Pronghorn account (apply for an account at https://www.unr.edu/research-computing/hpc/support)

A. Logging In

Today, we want to make sure each of you are landing on the same login node whenever you connect. Please ssh to the appropriate login node below, substituting YOUR netid for "**`[netid]`**" (no brackets)!

If you were born in an EVEN year, please use login-0 for the entire module:

```
ssh [netid]@login-0.ph.rc.unr.edu
```

If you were born in an ODD year use login-1 for the entire module:

```
ssh [netid]@login-1.ph.rc.unr.edu
```

Usually, you'll use pronghorn.rc.unr.edu which will round-robin between the login nodes.

We 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.

B. Pronghorn Storage Locations

Pronghorn has two main storage locations, your home directory (aka on Pronghorn your "research file set") and the "association" directory. 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.

Your home directory (with 50GB of quota) can be found at:

```
cd /data/gpfs/home/$USER
```

or

```
cd ~
```

The GRAD778 association storage can be found at:

```
cd /data/gpfs/assoc/grad_778-0
ls -l
```

For instance, my lab's association is:

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

(you shouldn't be able to see the contents, in theory)

We are going to create a directory in your home for use with this class. Please enter:

```
# This recursively creates some nested folder for the class:
mkdir -p ~/grad778-f20/module09
```

II.What is parallel processing and what is a high performance computer?

A. 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 10 random numbers, and add 100 to each of them. Let's further say that it takes a single processor 1 minute for each calculation. We start at 9am.

Here is our random input data:
1 2 5 8 6 7 4 0 3 9

Time	Processor 1's calculation
9:00 to 9:01	1 + 100
9:01 to 9:02	2 + 100
9:02 to 9:03	5 + 100
9:03 to 9:04	8 + 100
9:04 to 9:05	6 + 100
9:05 to 9:06	7 + 100
9:06 to 9:07	4 + 100
9:07 to 9:08	0 + 100
9:08 to 9:09	3 + 100
9:09 to 9:10	9 + 100

Ok, so in human time ("wall time" aka "wall clock time" aka "real world time") we waited 10 minutes for these to get calculated. But what if we have two processors divide the workload? Each processor takes the same amount of time:

Time	Processor 1's calculation	Processor 2's calculation
9:00 to 9:01	1 + 100	2 + 100
9:01 to 9:02	5 + 100	8 + 100
9:02 to 9:03	6 + 100	7 + 100
9:03 to 9:04	4 + 100	0 + 100
9:04 to 9:05	3 + 100	9 + 100

Whoa! We halved the amount of time! It only took 5 minutes in this hypothetical. In this case, Processor 1 and Processor 2 were running in parallel (they were working on subsets of the larger problem at the same time). Now, as you scale up with more processors, we can see how much we reduce the time. We'll throw four processors at it:

Time	Processor 1's calculation	Processor 2's calculation	Processor 3's calculation	Processor 4's calculation
9:00 to 9:01	1 + 100	2 + 100	5 + 100	8 + 100
9:01 to 9:02	6 + 100	7 + 100	4 + 100	0 + 100
9:02 to 9:03	3 + 100	9 + 100	Idle	Idle

With four processors, we see that it takes only 3 minutes to process. Processor's 3 and 4 don't do anything during their last step, since all 10 calculations were completed.

Not all processes are parallelizable. How about a problem where we want each output to be the previous step plus a new random number that is created when the new step runs. In other words, each step requires knowledge of the previous step before it executes. These sorts of problems are not parallelizable.

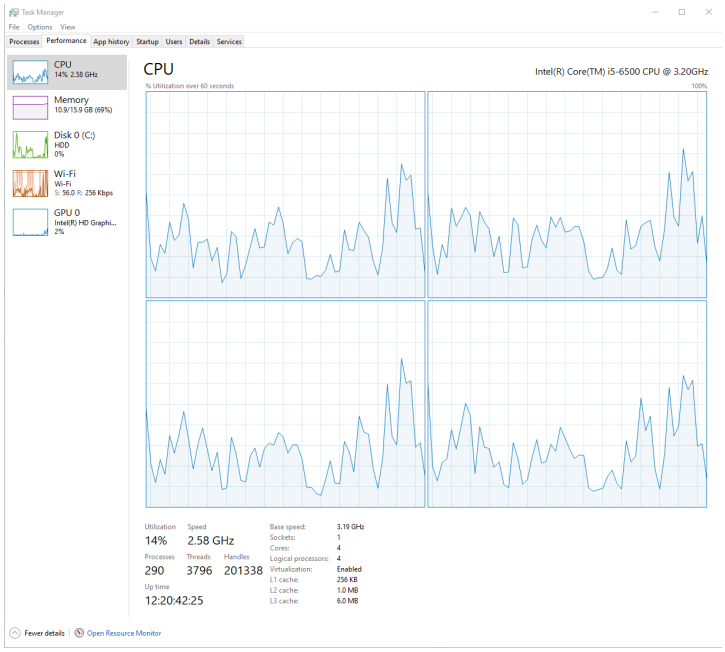
A typical first step in understanding if a problem is even conducive to parallel processing is to determine if a problem can be broken up into smaller problems, like the case above. In this case, each calculation had no reliance on the other, they were all doing the same thing, namely applying a function:

$Y = X + 100$

Where X was our input number, and Y is a single result of that calculation. Basic parallel processing is basically a big "for" loop, where we are iterating through these smaller problems and 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! Here is mine (4 cores on my Windows box):



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). Here is mine (8 cores on my Mac):

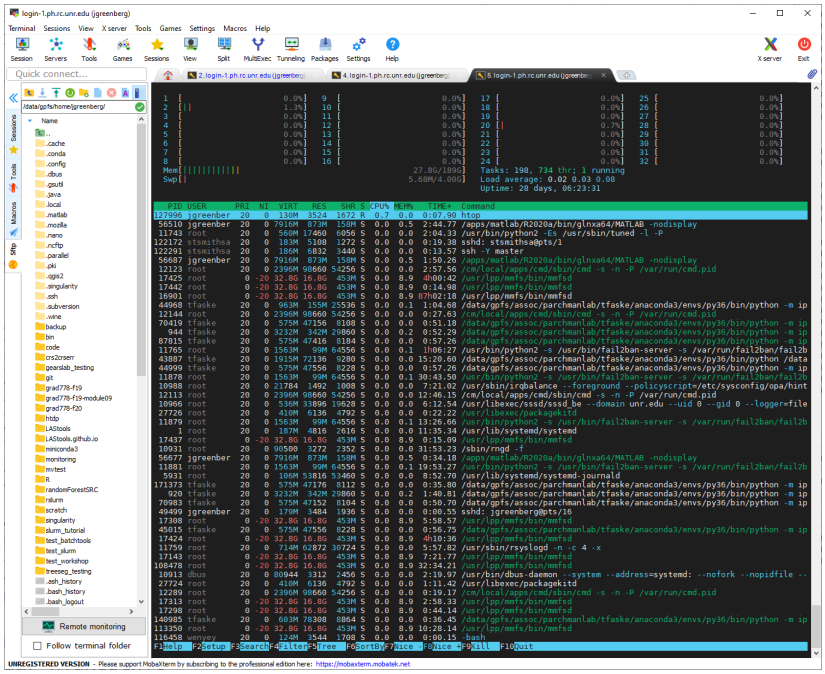
What is using the cores? Well, on Windows, in the Task Manager, click "Details" (you may need to click "More details" at the bottom. Sort by "Name", scroll down to some application you are using and notice the CPU column. On Mac, Application Monitor (All Processes), sort by Process Name and look at % CPU.

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

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. 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 32 (or 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 (Command) running, and the % CPU they are using.



Now, we've seen that for certain problems, we can 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?

B. Enter: 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 collocated at the Switch Citadel Campus located 25 miles East of the University of Nevada, Reno. Switch is the definitive leader of sustainable data center design and operation. The Switch Citadel is rated Tier 5 Platinum, and will be the largest, most advanced data center campus on the planet.

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

First up, let's talk about what a high-performance computer (HPC) is: really, it is a bunch of individual computers ("nodes"), just like the ones you are using, strung together with networking cables, with the ability to deploy "jobs" (some computational task you are trying to accomplish) across multiple nodes easily. As such, we can determine how many cores we have access to by counting the number of cores on each individual node, and summing them all up. Pronghorn has 3,456 CPU cores that (in theory) we have access to! In a perfect world (more on that later), you COULD divide the amount of time it takes to do a job by the number of cores you throw at it. With Pronghorn, you could theoretically do 10 YEARS of sequential calculations in less than one day! Put another way, Pronghorn's capabilities are 864 times faster than my Windows machine.

Your desktop or laptop is all yours, generally, so you aren't sharing the resources with anyone else. You've effectively pre-paid for ~ 5 years if computational time (warranty!) times the number of cores you have, so I've bought about 20 years of CPU-time on my Windows desktop and 40 years of CPU-time on my Mac laptop. Pronghorn, assuming a 5 year lifespan, has **17,280 years (!)** of CPU-time, all of which was purchased in advance. While you are probably ok with your laptop/desktop just sitting there idle not doing much, a research computer like Pronghorn is designed to be used at near-capacity! Also, this is a SHARED MACHINE and as such much of the process getting your programs to run on it requires some understanding of how the system shares its resources amongst all the users! Enter **SLURM.**

III. SLURM Jobs

SLURM is what is known as a workload manager. SLURM's job is to take the vast number of different jobs sent to it by all users in the system, reserve "resources" (# of nodes per job, # of cores per node, memory per job), and then execute 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. Within the job are "steps" which can be running sequentially or in parallel depending on the particulars of your workflow. A step consists of one or more "tasks". Each "task" runs on one or more "cpus" (cpu is the same as a logical core in SLURM parlance). Parallelization can occur at multiple levels: job, step, and task.

SLURM uses a "job script" written in any interpreted language that uses "#" as the comment character-- typically we'll use the "bash" language to create a job. This job script follows a very specific format that you will get familiar with. Your job script 1) tells SLURM what resources you need, and 2) once the resources are allocated, what programs to execute and how to allocate the resources to those programs.

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). While you CAN run "interactive jobs" on Pronghorn, this should be minimized wherever possible. Interactive jobs typically idle resources quite a bit.

A. First SLURM Job

We are going to try to keep each batch program neatly organized, and running in its own folder. So let's go ahead and make a new folder and cd to it:

If you missed the step to make a class folder, let's do it now:

```
mkdir -p ~/grad778-f20/module09
```

Now let's set up this module's folder.

```
cd ~/grad778-f20/module09
mkdir module09-A
cd module09-A
```

Let's first create a bash script that will run our program. All this is going to do is "sleep" (wait) for 30 seconds, and then print out the hostname of the computer the program runs on.

```
nano module09-A.sh
```

And add in this text (remember to skip the ###). Copy and paste may be a little weird in an ssh window. You will likely need to click in the window and right click to paste.

###

```
#!/bin/bash

# Spike a cpu for 15 seconds:
timeout 15s cat /dev/zero > /dev/null

# Print out the hostname
hostname

# Safely exit:
exit 0
```

###

Your nano clipboard (Ctrl+U) is not the system clipboard (right click and paste)
Use Ctrl+X to exit nano, press Y to save changes.

We are going to first test this on the login node. **AS A WARNING, DO NOT GET IN THE HABIT OF RUNNING CODE ON THE LOGIN NODES, THEY ARE NOT DESIGNED FOR COMPUTE WORKLOADS.**

You should be connected twice, so in one of the windows, please type:

```
htop -u $USER
```

How many logical cores do you see at the top? login-0 and login-1 are actually the same type of hardware, but in login-0's case, simultaneous multithreading ("SMT") has been turned on, in which each physical cpu can run two threads, thus resulting in 32 physical cpus x 2 threads = 64 logical cpus. On login-1 SMT is disabled, so you will see 32 cpus.

We are going to test our script now on the login nodes. Keep an eye on your processes in htop while in the other session start it running:

```
bash module09-A.sh
```

You'll see *cat /dev/zero* run for 15 seconds, note the CPU usage. It will probably be 100% unless the node is being overutilized. If you miss it, try running it again.

At the end, the process returns the hostname of the node it ran on!

OK. The program works! Now we want to modify this program and submit it to the queue.

Create an SBATCH script as follows:

```
nano module09-A_batch.sl
```

Edit the file as follows (notice you are just adding a bunch of #SBATCH commands), and **change the [yourEMAIL] block to your actual email address**

###

```
#!/bin/bash

#SBATCH --job-name=module09-A_job
#SBATCH --output=module09-A_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=2GB
#SBATCH --hint=nomultithread
```

```
#SBATCH --time=00:01:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

bash module09-A.sh

###
```

Control-X, Y to save it.

Ok, time to submit your job to the queue! Rather than "bash [your file]" you are going to:

```
sbatch --reservation=cpu-s5-grad_778-0_30 module09-A_batch.sl
```

You should see it say "Submitted batch job XXXXXX".

NOTE: YOU WILL TYPICALLY NOT USE THE `--reservation` FLAG. We are using that for the class, but typically you just "`sbatch batchfile.sl`".

Note the "jobid" and then check the queue status by typing (be quick, this will only show that a job is in the queue or is currently running):

```
squeue --user $USER
```

You will see something like:

2550561 cpu-core- module09 jgreenbe R 0:04 1 cpu-26

(If your status is PD the job is queued)

Type squeue in a few more times (or press the up arrow and return) and watch its progress.

Hint: you can also use this to watch your jobs in "realtime". Note it stops after ~15 seconds. We put timeout 30 to make sure you aren't abusing the squeue command. You should generally avoid watching the squeue.

```
timeout 30 watch squeue --user $USER
```

The first column is your job ID, the second is the PARTITION you submitted to, the third is the jobname you gave it, then your username, then the "Status" (ST) which likely says "R" which means its running, or "PD" if it is "pending", meaning its waiting for resources, then the number of nodes you asked for (1), then the node it is running on (cpu-XX). **This may not start running immediately!!!** You are sharing this with other people!

You can get finer details about running jobs by typing:

```
sstat [jobid]
```

And if you want to cancel the job (don't do this right now):

```
scancel [jobid]
```

Once it's done (you no longer see it in squeue) check the output file it created by:

```
more module09-A_output.txt
```

You should see it return the hostname of the computer the program ran on, cpu-XX -- not the login nodes as previously.

While this was all happening, check your email. You should have gotten a notification when your job began and, afterwards, when it ended. The email notifications are not required, but can be helpful when you are trying to plan and track jobs.

Let's break down this job script a bit more. All SBATCH directives must appear together at the top of a submission script, and should generally be formatted as "--<long option>=<value>"

--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 to this explicitly.
--error: sends standard error from a job into a filename(s) of your choice. Note you can also have your script to 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 64 logical cores, so evenly distributed we have ~ 4GB per cpu. You can ask for more, however.
--hint=nomultithread: if your process does not take advantage of 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-user: the email address to send notifications to (remove this if you don't want emails)

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

B. Process-based parallelism and resource requests

Our previous example only did one thing, 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.

We are going to need to understand where to place the "for" loop, and how this connects with resources we are going to request from SLURM, but first let's do a brute force job submission process. Instead of submitting our job once, we are going to submit it four (4) times and watch the queue.

First up, let's start monitoring our queue in ONE of our ssh sessions:

```
watch squeue --user $USER
```

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

```
sbatch --reservation=cpu-s5-grad_778-0_30 module09-A_batch.sh
sbatch --reservation=cpu-s5-grad_778-0_30 module09-A_batch.sh
sbatch --reservation=cpu-s5-grad_778-0_30 module09-A_batch.sh
sbatch --reservation=cpu-s5-grad_778-0_30 module09-A_batch.sh
```

Watch the squeue. You MIGHT see them all start running simultaneously, but maybe not if the queue is heavily utilized. After 15s of running, each should finish running.

Congrats! You (might have) parallel processed on an HPC! Each of those jobs needed 15s of processing, so you needed 1 minute 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 15s of "wall" (human) time to do 1 minute 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.

Let's make a new script, and formalize a looping that somewhat replicates this. First let's set up a new folder for this exercise:

```
cd ~/grad778-f20/module09
mkdir module09-B
cd module09-B
```

We are going to make a very simple bash program that returns the hostname it is running on, how many cores the program "sees", and then sleeps for 10 seconds before exiting the program.

```
nano module09-B.sh
```

Paste in:

###

```
#!/bin/bash

# Print out the hostname
hostname

# Prints out how many threads the job sees available:
nproc

# Prints out how many threads per core:
lscpu | grep -E '^Thread'

# Spike a cpu for 15 seconds:
timeout 15s cat /dev/zero > /dev/null

# Safely exit:
exit 0
```

###

Let's test it real quick:

```
bash module09-B.sh
```

It should return something that looks like:

login-1
32
Thread(s) per core: 1

And then it spikes a single CPU for 15 seconds before exiting.

Now, we are going to do the same thing but by submitting our job to the queue. Let's experiment with the amount of resources we are going to give our jobs. We are going to call our command "module09-B.sh" with "srun --ntasks=2" which is going to tell Slurm to create a step that is going to run one task at a time (module09-B.sh). We are also going to "trick" the system into ignoring the two threads per core by giving each process the --cpus-per-task=2 ("use two logical cores") flag. We suspect our job will take up to a minute to execute (4 x 15 seconds), so we are going to give it a bit more time to finish, so we'll adjust the time to be 2 minutes to be on the safe side.

```
nano module09-B_1cpu_batch.sl
```

###

```
#!/bin/bash

#SBATCH --job-name=jobB_1cpu
#SBATCH --output=module09-B_1cpu_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=2GB
#SBATCH --hint=nomultithread
#SBATCH --time=00:02:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

for i in $(seq 1 4)
  do
  srun --ntasks=1 --cpus-per-task=2 --exclusive bash \
  ~/grad778-f20/module09/module09-B/module09-B.sh &
done

# Add a wait:
wait
```

###

Control-X, Y.

```
sbatch --reservation=cpu-s5-grad_778-0_30 module09-B_1cpu_batch.sl
```

Keep an eye on the squeue, particularly how long it takes to finish (should be ~1 minute). Once it's done (remember how to check?) take a look at the output file:

```
more module09-B_1core_output.txt
```

This program runs four of the same steps sequentially (module09-B.sh). Each step runs for approximately 15 seconds, and the total wall time is one minute. Now let's throw more resources at the job to reduce the wall time.

There are no dependencies between the module09-B.sh processes. This means that all four processes can be run simultaneously (embarrassingly parallel). Let's modify the submission script to execute the four steps in parallel.

```
nano module09-B_4cpu_batch.sl
```

###

```
#!/bin/bash

#SBATCH --job-name=jobB_4cpu
#SBATCH --output=module09-B_4cpu_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=2GB
#SBATCH --hint=nomultithread
#SBATCH --time=00:02:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

for i in $(seq 1 4)
  do
  srun --ntasks=1 --cpus-per-task=2 --exclusive bash \
  ~/grad778-f20/module09/module09-B/module09-B.sh &
done

# Add a wait:
```

```
wait
```

###

```
sbatch --reservation=cpu-s5-grad_778-0_30 module09-B_4cpu_batch.sl
```

Keep an eye on the squeue. Note your job finished in only 15 seconds!

EXERCISE #1:

Experiment with different numbers of tasks allocated to your job to see the runtime.

Create a new SBATCH script that has the following characteristics:

- The script is named "module09-B_8cpu.sh"
- The job name is "jobB_8cpu"
- The output filename is "module09-B_8cpu_output.txt"
- The job is assigned 8 tasks.
- The job runs the same Bash as before:
  ~/grad778-f20/module09/module09-B/module09-B.sh

Submit the script, and compare the execution time to giving it ntasks=4. Did it execute any faster? Why or why not?

Answer: you only had 4 tasks to accomplish, but you asked for resources to run 8 tasks. You ended up idling 4 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.

Now, let's make a BIGGER job. Instead of looping 4 times, let's loop 16 times. Create a new sbatch script that runs module09-B.sh 16 times in parallel, and adjust the resource request to reduce the execution time as much as you can.

Hint, change the looping to:
for i in $(seq 1 16)

Now, let's REALLY scale this job. We are now going to run our job 128 times across TWO nodes and re-enable multithreading.

```
nano module09-B_128cpus_batch.sl
```

###

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

```
#SBATCH --job-name=jobB_128cpu
#SBATCH --output=module09-B_128cpu_output.txt
#SBATCH --nodes=2
#SBATCH --ntasks=128
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=2GB
#SBATCH --hint=multithread
#SBATCH --time=00:02:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

for i in $(seq 1 128)
  do
  srun --nodes=1 --ntasks=1 --cpus-per-task=1 --exclusive bash \
  ~/grad778-f20/module09/module09-B/module09-B.sh &
done

# Add a wait:
wait
```

###

```
sbatch --reservation=cpu-s5-grad_778-0_30 module09-B_128cpus_batch.sl
```

Watch the queue!

What happens if you remove the `--nodes` option?

C. Multithreaded Execution and Tuning Resource Requests

Now we move towards a more complex (but more common) problem, where each of our tasks are, themselves, multithreaded. A thread is part of a larger process, but can be run concurrently and often share memory with each other.

```
cd ~/grad778-f20/module09
mkdir module09-C
cd module09-C
```

In this case, we are going to make a python program that creates a multiprocessor "pool" that spawns four "worker" threads that each count to 300 million (so the sum total of the code counts to 1.2 billion in four (4) parallel chunks).

```
nano module09-C.py
```

```
###
# Multiprocessing is in the Python standard library
from multiprocessing import Pool
import os
from time import time

def sleep_load(x):
        counter = 0
        while counter < 300000000:
            counter = counter+1

def main():
        # Record the start time of execution
        start_time = time()

        # The number of simulated tasks to run
        tasks = 4

        # The number of CPUs requested from Slurm for this job/step
        # This should generally be matched to the --cpus-per-task
        cpus_available = 4
        print(f"available CPUs defined by python: {cpus_available}")

# Create a worker pool of size cpus_per_task, and process the
# simulated steps
```

```
        with Pool(cpus_available) as p:
                p.map(sleep_load, range(tasks))

        # Output the elapsed execution time
        print(f"Elapsed time: {time() - start_time}")

if __name__ == '__main__':
        main()
###
```

On your login node, `htop -u $USER` in one session and then in the other session test the code (which WILL run in parallel on the login node). We'll add the command "time" to figure out the execution time:

```
time python module09-C.py
```

.

You should see it take about 15 seconds, and watch four cpus light up on the login nodes.

Let's take a whack at an sbatch script:

```
nano module09-C_1task_1cpu_per_task_batch.sl
```

###

```
#!/bin/bash

#SBATCH --job-name=jobC_1task_1cpu_per_task
#SBATCH --output=module09-C_1task_1cpu_per_task_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=2GB
#SBATCH --hint=multithread
#SBATCH --time=00:02:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

srun --exclusive time python \
  ~/grad778-f20/module09/module09-C/module09-C.py &
```

```
# Add a wait:
wait

###

sbatch --reservation=cpu-s5-grad_778-0_30 \
module09-C_1task_1cpu_per_task_batch.sl
```

Since we added "time" to our script, we can check the output:

```
more module09-C_1task_1cpu_per_task_output.txt
```

Elapsed time was > 60 seconds! But our python script was supposed to use four cpus to do the job and only take 15 seconds. What happened? The SBATCH script did NOT match what your python code was tuned for. So, let's do some fixes.

One thing to ask is "Does my code use Simultaneous multithreading (SMT)?" This can sometimes be hard to know, and we'll see what happens if we get the wrong answer. How do we figure this out? Well… http://letmegooglethat.com/?q=Simultaneous+multithreading+python

If we check a few links, we'll see that for our particular problem, the answer is NO, Python's multiprocessing libraries can not take advantage of SMT. Pronghorn's cores allow for two simultaneous threads, but only with code that takes advantage of it. If we know the code doesn't use SMT, we can "disable" it by including:

#SBATCH --hint=nomultithread

We also need to adjust the cpus-per-task and to match what python "wants", which is four. Let's make a new SBATCH script:

```
nano module09-C_1task_4cpus_per_task_nosmt_batch.sl

###

#!/bin/bash

#SBATCH --job-name=jobC_1task_4cpus_per_task
#SBATCH --output=module09-C_1task_4cpus_per_task_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem-per-cpu=2GB
#SBATCH --hint=nomultithread
```

```
#SBATCH --time=00:02:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

srun --exclusive time python \
  ~/grad778-f20/module09/module09-C/module09-C.py &

# Add a wait:
wait
```

###

```
sbatch --reservation=cpu-s5-grad_778-0_30 \
module09-C_1task_4cpus_per_task_nosmt_batch.sl
```

Check the output for the elapsed time:

```
more module09-C_1task_4cpus_per_task_output.txt
```

Exercise:

Experiment with enabling Simultaneous Multithreading and seeing how it impacts execution speed:

```
#SBATCH --hint=multithread
```

Experiment with how cpus-per-task impact execution speed. If you give the python code 8 cpus-per-task, does it speed it up?

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

Now, what if we want to auto-adjust python to match the slurm cpus-per-task? That's easy! When we launch a job via SBATCH, the environment of the execution includes information on the SBATCH parameters. In the case of cpus-per-task, we can query the "SLURM_CPUS_PER_TASK" environment variable. In python, we can access this via:

```
import os
cpus_available = int(os.environ['SLURM_CPUS_PER_TASK'])
```

Let's update our python script a bit to do more work (it will count to 300 million 16 times now) but to allow us to auto-adjust the cpus_available to the multiprocessing pool:

```
nano module09-C_v2.py

###

# Multiprocessing is in the Python standard library
from multiprocessing import Pool
import os
from time import time
import threading

def sleep_load(x):
        counter = 0
        while counter < 300000000:
            counter = counter+1

def main():
        # Record the start time of execution
        start_time = time()

        # The number of simulated tasks to run
        tasks = 16
        # The number of CPUs requested from Slurm for this job/step
        cpus_available = int(os.environ['SLURM_CPUS_PER_TASK'])
        print(f"available CPUs defined by python: {cpus_available}")

        # Create a worker pool of size cpus_per_task, and process the
simulated
        # steps
        with Pool(cpus_available) as p:
                p.map(sleep_load, range(tasks))

        # Output the elapsed execution time
        print(f"Elapsed time: {time() - start_time}")

if __name__ == '__main__':
        main()

###
```

Note that if you want to test this on the login node, you will need to manually set:

```
export SLURM_CPUS_PER_TASK=16
time python module09-C_v2.py
```

This should take about 16 seconds on the login node.

Ok, now to create an SBATCH script,

```
nano module09-C_1task_Ncpu_per_task_nosmt_batch.sl
```

###

```
#!/bin/bash

#SBATCH --job-name=jobC_1task_Ncpu_per_task_nosmt
#SBATCH --output=module09-C_1task_Ncpu_per_task_nosmt_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --mem-per-cpu=2GB
#SBATCH --hint=nomultithread
#SBATCH --time=00:02:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

srun --exclusive time python \
  ~/grad778-f20/module09/module09-C/module09-C_v2.py &

# Add a wait:
wait
```

###

```
sbatch --reservation=cpu-s5-grad_778-0_30 \
module09-C_1task_Ncpu_per_task_nosmt_batch.sl
```

Exercise:

Try adjusting cpus-per-task in your SBATCH script up and down to see the impact on execution time. Python will now use whatever cpus it is handed, although it only has to do 16 tasks, so anything more than 16 cpus will be a waste.

D. Message Passing Interface (MPI)

MPI example
Message Passing Interface (MPI) is a standardized communication protocol for parallel computing. As its name indicates, MPI defines the syntax and semantics of different routines for writing message-passing programs that allow the interaction between different processes. Since it is considered a standard, MPI is supported in practically all HPC platforms, which allows portability of code between different platforms. Different implementations of MPI are available, some are open-source, and others are from vendors.

MPI is a high-performance form of interprocess communication that allows multiple processes to coordinate computation and share memory. Processes can be distributed across many computers. Due to this feature, MPI is a mechanism for large-scale computation. Pronghorn is designed for high-performance distributed computation.

Let's check the MPI libraries available on Pronghorn:

```
module avail
```

You'll find different versions of OpenMPI and Intel MPI available on Pronghorn.

We will execute a simple MPI example in C to calculate an approximation of Pi that uses Intel MPI. Information on the approximation example can be found here.

We will use two files located in the class directory: mpi_pi_reduce (the binary code), and mpi_pi_reduce.c (the source code).

First create a new subdirectory and copy the files

```
cd ~/grad778-f20/module09
mkdir module09-D
cd module09-D

cp /data/gpfs/assoc/grad_778-0/mpi_pi_reduce* \
~/grad778-f20/module09/module09-D/

ls
```

We can test this on the login node using 2 cores via:

```
module load intel/mpi
```

```
mpirun -n 2 mpi_pi_reduce
```

Let's create the SBATCH file for this example:

```
nano module09-D_program_batch.sl

###

#!/usr/bin/env bash
#SBATCH --job-name=PI_approx
#SBATCH --output=module09-F_program_%x.%j.txt
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=3500M
#SBATCH --hint=multithread
#SBATCH --hint=compute_bound
#SBATCH --time=00:05:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0

module load intel/mpi

echo "Just a quick MPI test that calculates pi."

srun --ntasks $SLURM_NTASKS --mpi=pmi2 ./mpi_pi_reduce

###
```

For this example we are defining 4 tasks, which indicates that there will be one Master and 4 workers set for this process. The master will send information to each of the workers, the workers will have a distributed load of computations and once they have finished their processes, they will send the results to the master, which will compute and print the average from all the computations made by the workers.

It is important to make sure that the modules required for the MPI process are loaded before srun is executed. Also the `--mpi` flag in `srun` must be set to identify the type of MPI to be used. More information about running MPI jobs in Slurm here.

Now let's review the source code.

```
nano mpi_pi_reduce.c
```

We will not go into details about the C code in this example, but let's review some basic ideas.

```
#define DARTS 50000     /* number of throws at dartboard */
#define ROUNDS 100      /* number of times "darts" is iterated */
```

We are defining 100 iterations each one with 50000 darts. For each of these iterations, evaluated in a loop, the number of darts will be distributed across the 4 workers, and the master will store and reduce the values resulting from each round.

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
```

In this part of the code, we are getting the initial values that are required for running the code (`homepi, pisum, pi,` etc.), and also the task id and the number of tasks, which are the environment variables defined in the sbatch script.
The `MPI_Comm_size` function will initiate the 4 workers, and the `MPI_Comm_rank` will assign a rank to each worker based on the task ID.

Submit the job to the queue and evaluate the output.

```
sbatch --reservation=cpu-s5-grad_778-0_30 module09-D_program_batch.sl
```

Let's modify the SBATCH file to request 128 tasks. We will also add a sleep function to evaluate how the queue changes.

```
nano module09-D_program_bigger.sl

###

#!/usr/bin/env bash
#SBATCH --job-name=PI_approx
#SBATCH --output=module09-F_program_bigger.txt
#SBATCH --ntasks=128
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=3500M
#SBATCH --time=00:05:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --hint=compute_bound

echo "Just a quick MPI test that calculates pi."
```

```
srun -n $SLURM_NTASKS --mpi=pmi2 ./mpi_pi_reduce
sleep 30

###

sbatch --reservation=cpu-s5-grad_778-0_30 module09-D_program_bigger.sl
```

Evaluate how the queue changes, and how the resources are allocated to the different users in the class.

```
watch squeue --reservation=cpu-s5-grad_778-0_30
```

How many nodes did your job use? Notice that your job basically used every node it could get its hands on to achieve the outcome.

How efficient is this distribution of steps? How can you increase the performance?

E. Interactive Jobs

Sometimes you just need to interact with your software on a compute node. The steps for getting an interactive job are as follows:

1. We want to connect to a specific login node, not the round-robin pronghorn: login-0.ph.rc.unr.edu or login-1.ph.rc.unr.edu
2. After we connect, use tmux to "preserve" your session from disconnects:

```
tmux
```

3. Now we ask for an interactive job with 2 cores:

```
srun --reservation=cpu-s5-grad_778-0_30 --nodes=1 --cpus-per-task=2 \
--partition cpu-core-0 --account cpu-s5-grad_778-0 \
--mail-user=[youremail] --mail-type=BEGIN --time=00:05:00 --pty bash
```

This is asking for a bash shell on four (2) cpus on one (1) node for 5 minutes. AND NOW WE WAIT. This could take a long time to get. If you add in the --mail-user and --mail-type you will receive an email that tells you the interactive job has started, allowing you to set up an alert on your phone, for instance.

If you get disconnected, reconnect to the SAME login node (this is why we don't use pronghorn.rc.unr.edu). Then we can check our "saved" sessions:

```
tmux ls
```

Note the ID of your tmux session and reattach to that session:

```
tmux attach -d -t [id]
```

What happens if we're unavailable within five minutes of our interactive session running?

What happens if we don't run a terminal multiplexer on a login node and our network connection is interrupted?

Some notes on interactive jobs:

- AVOID USING THEM IF AT ALL POSSIBLE
- Don't expect to get an immediate interactive session, especially if you aren't an investor in the system.
- REMEMBER TO LOGOUT OF THE SESSION WHEN YOU ARE DONE. Otherwise you will idle the node (and possibly get charged for it) up until the --time parameters says disconnect.

F. THE QUEUE

So now that we know how to submit jobs to Pronghorn based on different workflows, it is worth taking a minute to understand how Pronghorn chooses which jobs to run in which order. Remember, while you are submitting jobs to Pronghorn, so are other users. Pronghorn manages its workload with priority queues.

Job Priority:

The simplest job priority we can imagine is what is known as "First Come, First Serve" -- in other words, each job is placed in the order they were submitted. If there are resources currently available for a job to run, it begins executing. If the HPC is "full" (running other jobs), the system will wait until current jobs are finished until the necessary resources are free, and then begin executing the next job in line.

"First come, first serve" is easy to "cheat" -- if someone submits 10,000s of jobs that take a long time, they can totally dominate the machine. So we use what is known as the "FairShare" queuing system. Without getting too deep in the details, FairShare allows jobs from users who have NOT been using the system much in the recent past to "jump the line" in front of users who have been using the system more heavily -- it provides a fair share of the computing resources to all users.

The job prioritization also considers "backfilling", which allows smaller jobs to "jump the line" if sufficient resources are available and running the job will not delay the next job in the queue. For instance, say our HPC had only two nodes with 32 logical cores each. Four users submit 30-core jobs to the cluster, and four users submit 1-core jobs to the cluster afterwards. The first two users will see their 30-core jobs start on the two nodes, but that leaves 2 cores per node idling. Instead of waiting for the next 30-core job, Pronghorn will start the 1-core jobs on the remaining 2-cores. This maximizes the utilization of the system!

Fairshare makes Pronghorn equitable, while Backfill maximizes utilization!

Services

With a system as large as Pronghorn, we have different use cases and ways to pay for access to the system. As such, the resources of pronghorn are broken up into different partitions, which are basically isolated hardware where jobs are submitted. The job priorities described above are subdivided by the partitions. Each partition has different limitations on the number of jobs, nodes, cores, and job times. There are "accounts" associated with these partitions which are used for accounting purposes.

Pronghorn users have access to the following partitions (note these change):

Service	Partition	Account	Max nodes	Max CPUs	Max jobs RUNNING + PENDING	Max runtime	Cap on cpu-hours	Cost to user	Access limitations
Tester	cpu-s6-core-0 ("test")	cpu-s6-test-0	2	128	2	00-00:15:00	None	Free$0	Any with NetID
Sponsored	cpu-s3-core-0 ("sponsored")	cpu-s3-sponsored-0	1	64	12	00-02:00:00	None	Free$0	Any with NetID
Student Research			53	3392	51200	14-00:00:00	Yes	Free	UNR/Davidson student with NetID
Renter	cpu-s2-core-0 (“renter”)	varies	53	3392	51200	14-00:00:00 days	None	Pay-as-you-go hourly$/cpu-hour	Any with NetID*
Student Research	cpu-s5-core-0 (“renter”)		53	3392	51200	14-00:00:00	Yes	Free$0	Any student with NetID

Other services

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

*If non-NSHE, granted access by Nevada Center for Applied Research (https://www.unr.edu/ncar).

G. Monitoring Jobs

There are a number of ways to monitor job progress.

Where is my job in the queue and when will it start?
You've already learned "`squeue`" to see where your jobs are relative to the overall queue (try typing it without the -u parameter). Generally, avoid "`watch squeue`" unless you really need to do it. It can put a large load on the system.

squeue -P shows the current jobs ranked by priority
squeue --start -j <jobid> will tell you an approximation of when a specific job will begin

How do I watch a logfile my code is making?
Especially when you are just beginning to work on an HPC, try to get your code to output logfiles that you can look at. If your logfiles are recording info over time, you can "watch" the file in realtime by

```
tail -f mylogfile.txt
```

You can run this from the login nodes while your job is executing.

How do I check CPU and memory usage of my job in real time?

This is a bit more complicated, but you can do it by first asking for the names of the nodes you are running jobs on, and then ssh'ing into each one FROM the login node, and htoping. Here's an easy approach. Figure out the node names by:

```
$((squeue --user $USER --state RUNNING --Format "nodelist" --noheader
| uniq) 2>&1)
```

Then, you can:

```
ssh cpu-X -t 'htop'
```

Change "X" to match one of the nodes from above. That will launch htop on a node you are running a job on. You can open other sessions and repeat the process if you want to look at other node cpu usage. Remember you may be sharing this node with other folks! Once the job ends, your ssh connection/htop will be disconnected.

IV. Being a Good HPC Citizen

You are sharing this machine with many other folks on campus, so you need to play nice with others both within your "association" (basically who is sharing the paid resources), and across the machine. There are "rules" in place in the queueing that are or can be set up to prevent bad behavior, but these don't cover all possible situations. Let's cover a few of them now.

A. Login node processing

The login nodes are where everyone connects to Pronghorn, and can run software just like any of the compute nodes. In theory, you CAN run software on them BUT as a general rule, **avoid doing any significant processing on the login nodes** except for short tests. The login nodes should be used for transferring and organizing files and submitting jobs. Overuse of the login nodes can result in the admins contacting you.

B. Shared queue

You should not get into the habit of thinking your jobs will start immediately. During peak usage your jobs could take hours or DAYS to get started. Let's simulate this on the class reservation:

```
cd ~/grad778-f20/module09
mkdir module09-heavyqueue
cd module09-heavyqueue

nano module09-heavyqueue.sl

###

#!/bin/bash

#SBATCH --job-name=module09-heavyqueue_job
#SBATCH --output=module09-heavyqueue_job.out
#SBATCH --nodes=2
#SBATCH --ntasks=32
#SBATCH --time=00:03:00
#SBATCH --cpus-per-task=1
#SBATCH --hint=compute_bound
#SBATCH --mem-per-cpu=2GB
```

```
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-0
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

sleep 120
hostname

###

sbatch --reservation=cpu-s5-grad_778-0_30 module09-heavyqueue.sl
watch squeue --reservation=cpu-s5-grad_778-0_30
```

If you were fast to submit, you'll see your jobs start running, but if you are slower, you will see your jobs waiting to start (check the "ST" column). Our class "partition" (we are using a reservation which is a bit different) only has 4 nodes allocated to it. Keep an eye on the queue for a few minutes — notice as the jobs finish, you will see the next set up jobs start up!

C. Shared storage

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. This is stored in:

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

We have a storage association named "grad778-0" for this course, let's change into that directory:

```
cd /data/gpfs/assoc/grad_778-0
```

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 "module9_student" and am asking everyone to make a subfolder of that with their username. We can do this easily with:

```
mkdir /data/gpfs/assoc/grad_778-0/module9_students/$USER
```

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

Now, let's change into that directory…

```
cd /data/gpfs/assoc/grad_778-0/module9_students/$USER
```

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

```
fallocate -l 100G \
/data/gpfs/assoc/grad_778-0/module9_students/$USER/foo
```

Type "success!" in the chat if you successfully made the file "foo" 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_778-0/module9_students/* --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_778-0/module9_students/$USER/foo
```

V. Conclusions and What Next???

First off, everyone who is enrolled for credit should complete the CANVAS quiz in which the answer is, of course. "Batman". Please do this now.

This introduction should jumpstart getting work done on UNR's HPC. 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. 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
2. Email hpc@unr.edu. This goes directly to the Pronghorn admins.
3. Attend the weekly "Hackathons" where the Pronghorn admins can give more hands-on support. Fridays from 2pm to 4pm on Slack.
4. Interact with other people using the system in your lab or department on Slack and ask questions!
5. Read the Slurm manuals: https://slurm.schedmd.com/
6. Read the manuals of your research software.

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-core-0 ("test")`	`cpu-s6-test-0`	2	128	2	00:15:00
`cpu-s3-core-0 ("sponsored")`	`cpu-s3-sponsored-0`	1	64	12	02:00:00
`cpu-s2-core-0 ("renter")`	`cpu-s2-grad778-0`	53	3392	51200	14 days

You won't have access to cpu-s2-grad778-0 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.