Advanced Utilization of Pronghorn

Dr Thomas White
University of Nevada, Reno

Module 8: "Advanced Utilization of Pronghorn"
November 6 2021
Lead: *Thomas White (Physics)*

This workshop will build on the principles developed in module 7. It will provide students with a solid understanding of the differences in the types/classes of parallel computing. We will cover multi-threading and multi-processing, shared and distributed memory, job scaling and profiling, interactive jobs, and containerization. All examples will utilize the Pronghorn High-Performance Computing (HPC) system.

Pre-reqs: Module 3 (Introduction to Linux), Module 7 (Introduction to Batch Processing in Pronghorn).

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!)

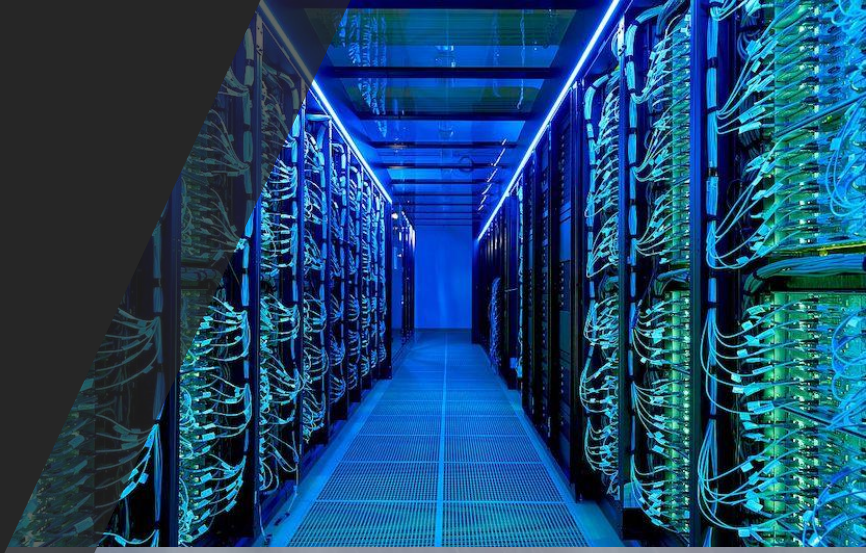
At the end of this module, I will ask you to run a parallelized python script on the pronghorn high performance computer which will generate a number. You will need to enter this number into the webcampus quiz in order to pass. This number will be unique to you, so make sure you are following along.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

Help Available in the Room

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

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).

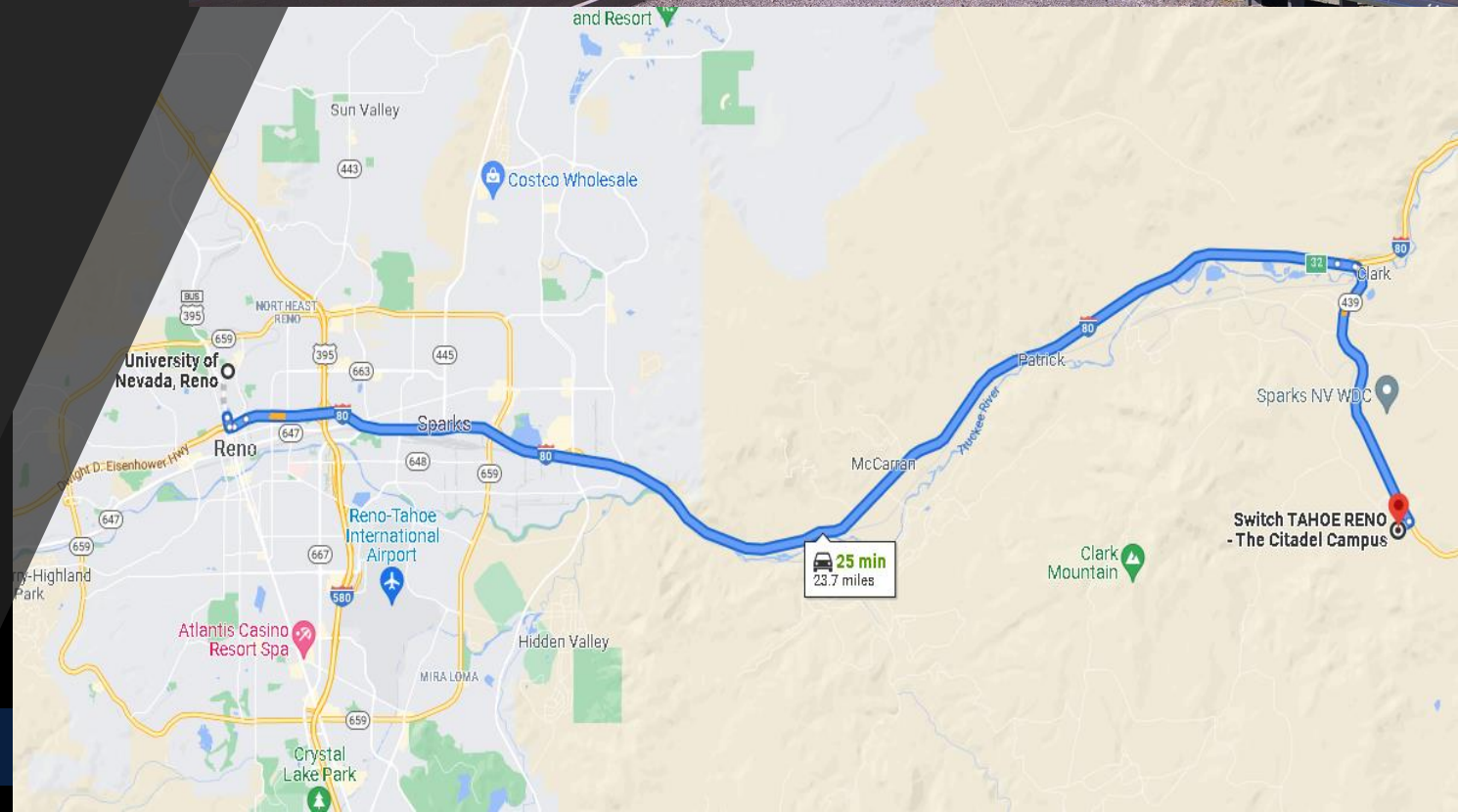
Pronghorn is composed of CPU, GPU, Visualization, and Storage subsystems interconnected by a 100Gb/s non-blocking Intel Omni-Path fabric. The CPU partition features **108 nodes**, **3,456 CPU cores**, and **24.8TiB of memory**. The GPU partition features **44 NVIDIA Tesla P100 GPUs**, **352 CPU cores**, 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



RECAP (Logging in)

Just like last time, open your SSH software (MobaXterm (win), Putty (win), or the Terminal (mac/linux)) and log in to Pronghorn twice.

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

Consider ONE of the logins to be your "monitoring" session, and the other where you'll do the bulk of the work.
You can remind yourself which node you are logged into with: `$hostname`

Just like last time, there are some files you will need for today's lecture. I have placed these files in our shared association directory.

Make a folder for this class

```
$mkdir -p ~/grad778/class2
```

Use the cd command to go into the folder

```
$cd ~/grad778/class2
```

Copy the files needed for today's class into that folder

```
$cp /data/gpfs/assoc/grad_778-1/class2/* .
```

You should now have 8 files in your class2 folder.

```
$ls
```

```
AddSquaresMultiThreadingMany.py
AddSquaresMultiThreading.py
heavyQueue.sl
input.lammps
mpiBatch.sl
multiprocessing.sl
SquareMyNumber.py
taskPar.sl
```

RECAP

- Last time we introduced a python script that squared six numbers and summed the results.
- We used two version of the code. The first added the numbers in serial.
- The second used the multi-processing package in python to run in parallel.
- The serial version of the code was taking approximately six seconds to run.

AddSquaresSerial.py (Serial)

```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)
```

Multiprocessing Package

AddSquaresMultiThreading.py (Parallel)

```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()

    #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)
```

RECAP: The SLURM Scheduler

multiprocessing.sl or exampleBatchScript.sl

```
#!/bin/bash

#SBATCH --job-name=myFirstJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=6
#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=YOUREMAILHERE
#SBATCH --mail-type=ALL

module load python3

srun python AddSquaresMultiThreading.py
```

We used the SLURM scheduler to run our programs on the Pronghorn supercomputer. Once we had written the SBACTH script, we submitted our job to SLURM using the sbatch command. For example:

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

In the SBATCH script we included all the information the scheduler needed to reserve the correct resources for our job. Remember, it is important to ask for an appropriate number of resources for our job.

We were also able to enter our email address to receive an email when our job began and when it finished.

Once the job was running, we were able to follow the progress of our job with these two commands:

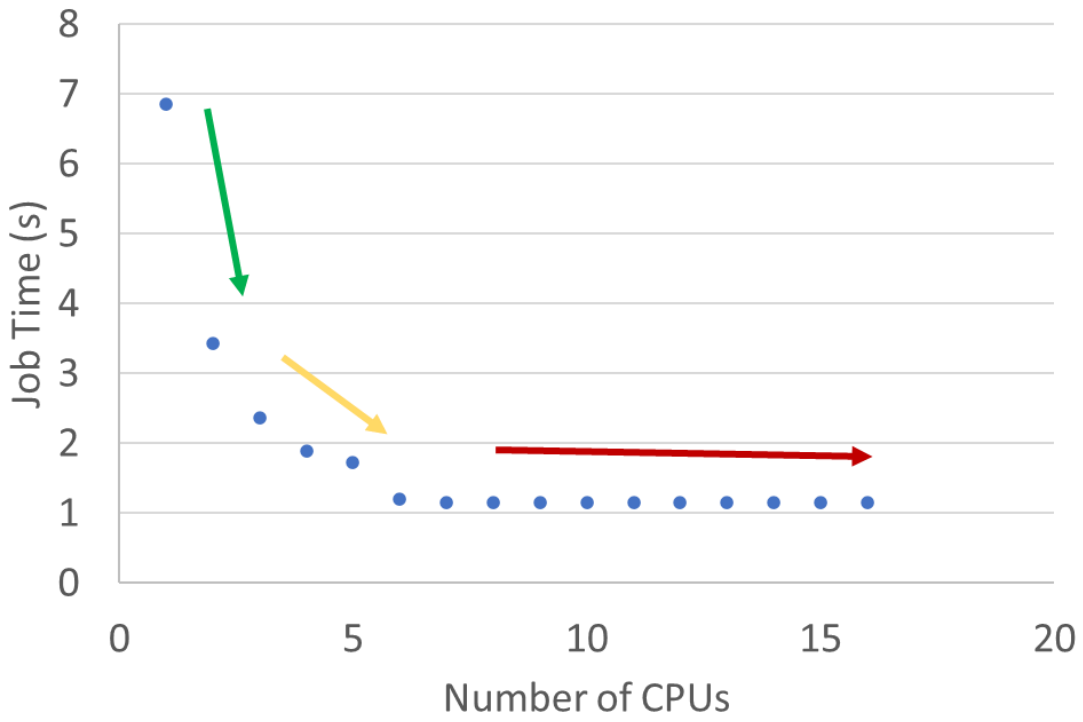
```
$squeue -u <netid>
```

```
$scontrol show job <jobID>
```

Remember, the output of the code or calculation is not printed to the screen, but instead saved to a file that we specified in the SBATCH script.

RECAP: Job Time vs Number of Processors

The final task for the last module was to see how fast we could make our calculation by running on more and more CPUs. You should all have ended up with a curve like this:



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

However, eventually we get diminishing returns.

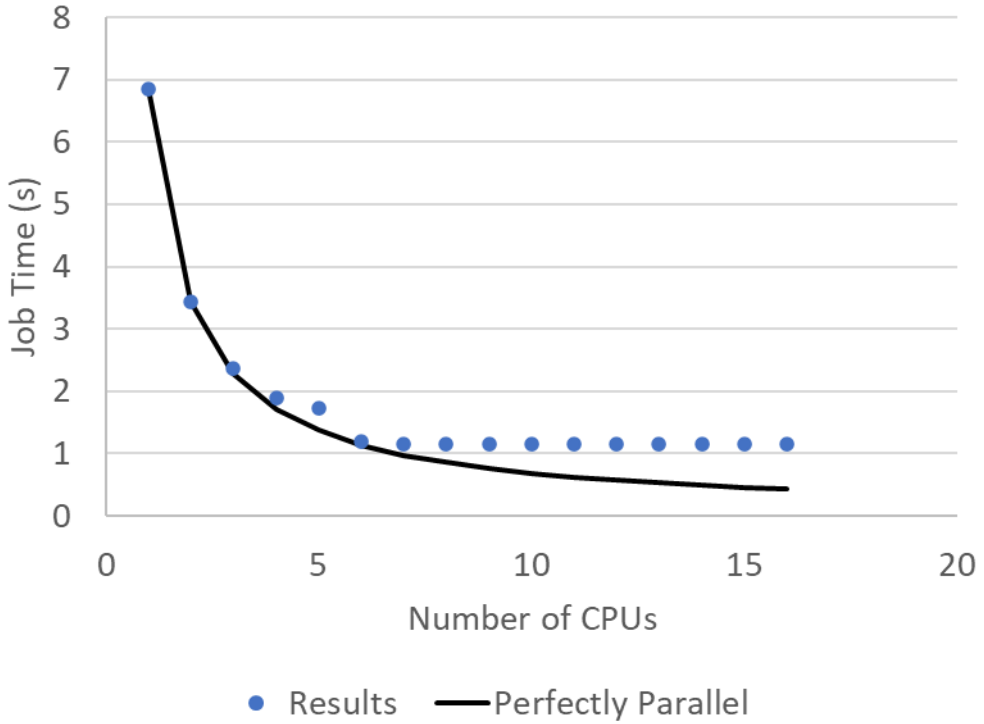
Above 6 CPUs there is no speed up. This is because we only had 6 tasks to accomplish, but we asked for resources to run more than 6 tasks. We ended up idling these extra CPUs because they were not being used. Why is this important? Computing resources are billed monthly. This will increase your computational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

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

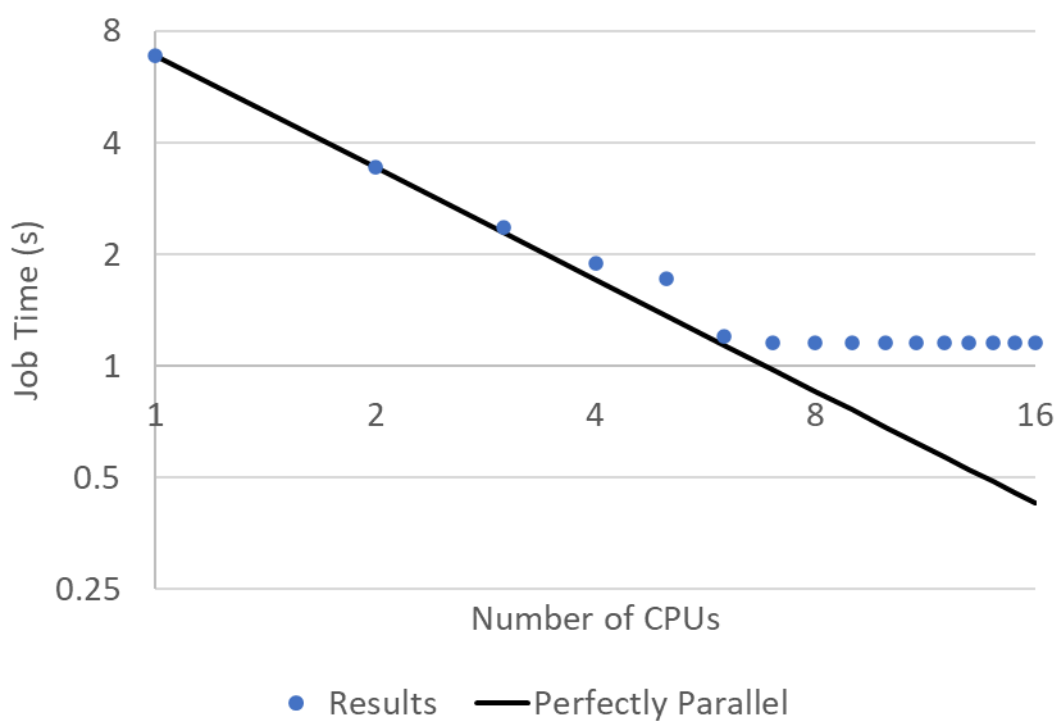
RECAP: Job Time vs Number of Processors

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

Linear-Linear Plot



Log-Log Plot



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!

Exercise 2: Remember how to run jobs

RECAP: Running Jobs

From last week, it seems like 2 CPUs was the good number to use for this calculation. We had significant speed up and didn't waste any resources i.e., we have matched our requested resources to our problem.

Either edit your batch script from last time to use 2 CPUs or use the multiprocessing.sl I provide youw ith.

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

Use `$ls -lh` to confirm that you have produced a new output file and that it is from today's session and not the previous one.

```
$squeue -u <netid>
```

How long did the job take according to the output file?

```
$scontrol show job <jobID>
```

How long did the job take according to the SLURM scheduler?

```
$scontrol show job <jobID>
```

```
#!/bin/bash

#SBATCH --job-name=myParallelJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#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=YOUREMAILHERE
#SBATCH --mail-type=ALL

module load python3

srun python AddSquaresMultiThreading.py
```

RECAP: Running Jobs

From last week, it seems like 2 CPUs was the good number to use for this calculation. We had significant speed up and didn't waste any resources i.e., we have matched our requested resources to our problem.

Either edit your batch script from last time to use 2 CPUs or use the multiprocessing.sl I provide youw ith.

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

Use `$ls -lh` to confirm that you have produced a new output file and that it is from today's session and not the previous one.

```
$squeue -u <netid>
```

How long did the job take according to the output file?

```
$scontrol show job <jobID>
```

```
Time taken in seconds - 3.0376791954040527
```

How long did the job take according to the SLURM scheduler?

```
$scontrol show job <jobID>
```

```
StartTime=2021-10-29T14:52:54 EndTime=2021-10-29T14:52:57
```

Notice that the SLURM scheduler is only accurate to the second. If you had a rough looking graph at the end of the last lecture, this could be the reason.

```
#!/bin/bash

#SBATCH --job-name=myParallelJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#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=YOUREMAILHERE
#SBATCH --mail-type=ALL

module load python3

srun python AddSquaresMultiThreading.py
```

More SLURM Commands

We should already be familiar with three SLURM commands - `squeue`, `scontrol` and `sbatch.`

Sometimes you will need to cancel currently running jobs. For example, you realize you have a bug in your code. You can do this using the scancel command:

```
$scancel <jobID>
```

To get information on currently running jobs including CPU usage, task information, node information we can use sstat. Note, as our job finishes in about 3 seconds, so you will need to be VERY fast to see this ;)

```
$sstat --jobs=<jobID>
```

To get information on past jobs we can use sacct

```
$sacct --jobs=<jobID>
```

```
$sacct --jobs=<jobID> --format=var_1,var_2, ... ,var_N
```

Use the sacct command now to display the jobid, jobname, ncpus, and cputime for the job you just ran.

```
       JobID    JobName      NCPUS     CPUTime
------------ ---------- ---------- ----------
2831304      myFirstJob          4   00:00:12
2831304.bat+      batch          4   00:00:12
2831304.ext+     extern          4   00:00:12
2831304.0        python          4   00:00:12
```

According to SLURM we used four CPUs and our job took 3 seconds. So, the total amount of CPU time was 12 seconds. This number will determine how much we are charged for our job.

CPU Time = Wall Time x Number of CPUs

Variable	Description
account	Account the job ran under.
avecpu	Average CPU time of all tasks in job.
averss	Average resident set size of all tasks in the job.
cputime	Formatted (Elapsed time * CPU) count used by a job or step.
elapsed	Jobs elapsed time formated as DD-HH:MM:SS.
exitcode	The exit code returned by the job script or salloc.
jobid	The id of the Job.
jobname	The name of the Job.
maxdiskread	Maximum number of bytes read by all tasks in the job.
maxdiskwrite	Maximum number of bytes written by all tasks in the job.
maxrss	Maximum resident set size of all tasks in the job.
ncpus	Amount of allocated CPUs.
nnodes	The number of nodes used in a job.
ntasks	Number of tasks in a job.
priority	Slurm priority.
qos	Quality of service.
reqcpu	Required number of CPUs
reqmem	Required amount of memory for a job.
user	Username of the person who ran the job.

Architecture of Pronghorn

Why does SLURM think we used four CPUs when we only asked for two?
Use either sacct or scontrol to find which NODE your job ran on. Then use control to view the details of that node.

```
$scontrol show nodes cpu-X
```

Don’t forget to replace X with the number of your node

```
CPUAlloc=0 CPUTot=64 CPULoad=0.01
AvailableFeatures=intelv4
ActiveFeatures=intelv4
Gres=(null)
NodeAddr=cpu-0 NodeHostName=cpu-0 Version=20.11.7
OS=Linux 3.10.0-1160.31.1.el7.x86_64 #1 SMP Thu Jun 10 13:32:12 UTC 2021
RealMemory=256000 AllocMem=0 FreeMem=223828 Sockets=2 Boards=1
MemSpecLimit=24000
State=RESERVED ThreadsPerCore=2 TmpDisk=0 Weight=200 Owner=N/A MCS_label=N/A
Partitions=cpu-s2-core-0,cpu-core-0,cpu-s3-sponsored-0
BootTime=2021-09-23T15:43:31 SlurmdStartTime=2021-09-23T16:54:02
CfgTRES=cpu=64,mem=250G,billing=314
AllocTRES=
CapWatts=n/a
CurrentWatts=0 AveWatts=0
ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
Comment=(null)
```

How many CPUs does this node have?

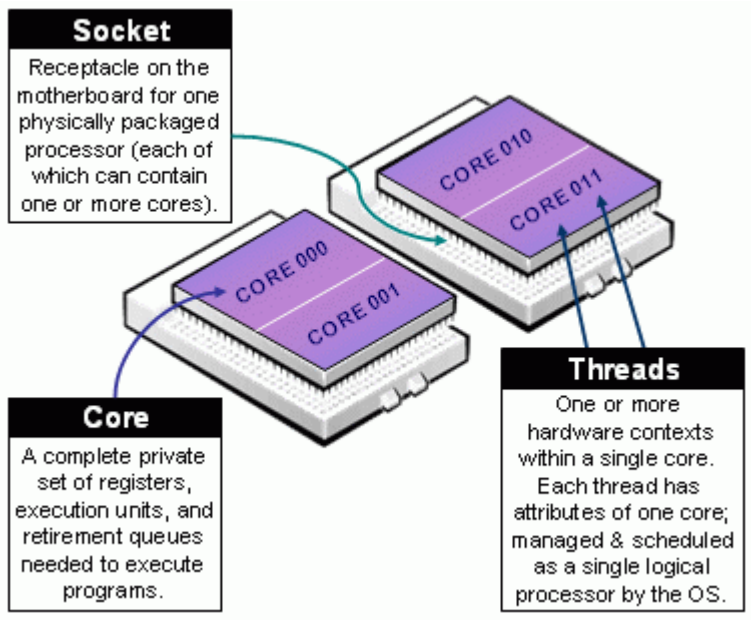
Architecture of Pronghorn

Why does SLURM think we used four CPUs when we only asked for two?
Use either sacct or scontrol to find which NODE your job ran on. Then use control to view the details of that node.

```
$scontrol show nodes cpu-X
```

Don't forget to replace X with the number of your node

```
CPUAlloc=0 CPUTot=64 CPULoad=0.01
AvailableFeatures=intelv4
ActiveFeatures=intelv4
Gres=(null)
NodeAddr=cpu-0 NodeHostName=cpu-0 Version=20.11.7
OS=Linux 3.10.0-1160.31.1.el7.x86_64 #1 SMP Thu Jun 10 13:32:12 UTC 2021
RealMemory=256000 AllocMem=0 FreeMem=223828 Sockets=2 Boards=1
MemSpecLimit=24000
State=RESERVED ThreadsPerCore=2 TmpDisk=0 Weight=200 Owner=N/A MCS_label=N/A
Partitions=cpu-s2-core-0,cpu-core-0,cpu-s3-sponsored-0
BootTime=2021-09-23T15:43:31 SlurmdStartTime=2021-09-23T16:54:02
CfgTRES=cpu=64,mem=250G,billing=314
AllocTRES=
CapWatts=n/a
CurrentWatts=0 AveWatts=0
ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
Comment=(null)
```



How many CPUs does this node have?

This node has 64 CPUS = 2 Sockets x 16 Cores Per Socket x 2 Threads Per Core

The SLURM scheduler gave us two threads (and charged us for two threads) even though we didn't use them. Simultaneous multithreading (SMT) is a technique for improving the overall efficiency of superscalar CPUs with hardware multithreading. In certain cases*, it lets the CPU do multiple calculations at once. However, writing code that utilizes SMT can be difficult.

*SMT typically allows the CPU where the type of calculation frequently changes. For example, if the program is a mix for floating point or integer calculations.

What is this multithreading thing?

Can we use multithreading to speed up our code?

In the job script there is the nomultithread directive which we used to disable SMT in our last class. Try changing this to --hint=multithread (which allows for multithreading) and perform the same speed test as last time. Try out 1,2,3,4,5,6,8,16 CPUs. Did we gain a speed up with multithreading compared with last time?

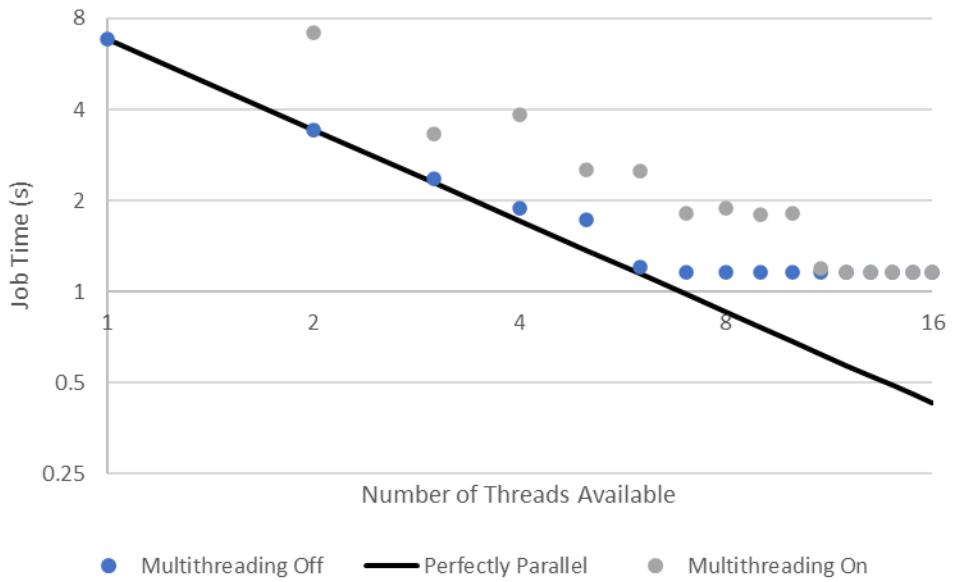
```
#!/bin/bash

#SBATCH --job-name=myParallelJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --mem-per-cpu=10M
#SBATCH --hint=multithread
#SBATCH --time=00:01:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-1
#SBATCH --mail-user=YOUREMAILHERE
#SBATCH --mail-type=ALL

module load python3

srun python AddSquaresMultiThreading.py
```

What is this multithreading thing?

Can we use multithreading to speed up our code?

In the job script there is the nomultithread directive which we used to disable SMT in our last class. Try changing this to --hint=multithread (which allows for multithreading) and perform the same speed test as last time. Try out 1,2,3,4,5,6,8,16 CPUs. Did we gain a speed up with multithreading compared with last time?

```
#!/bin/bash

#SBATCH --job-name=myParallelJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --mem-per-cpu=10M
#SBATCH --hint=multithread
#SBATCH --time=00:01:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-1
#SBATCH --mail-user=YOUREMAILHERE
#SBATCH --mail-type=ALL

module load python3

srun python AddSquaresMultiThreading.py
```



It looks like turning multithreading on made the code slower. However, this is not really the case. Remember, each processor has two threads (or two logical CPUs) for each physical CPU. In order to have a fair comparison, we should plot the number of threads available to the calculation on the x-axis of our graph. This means doubling the x-axis for the points with multithreading off.

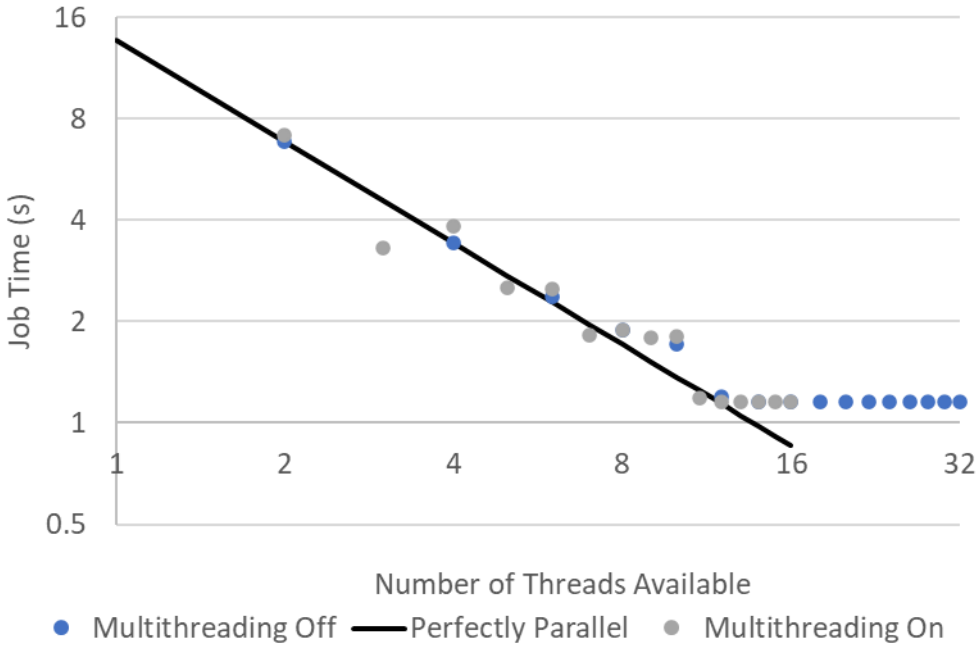
What is this multithreading thing?

Okay! So now it looks like the multithreading option gave essentially the same speed as the nomultithreading option.

Multithreading didn't make our job slower, but it didn't make it faster either. Simultaneous multithreading uses multiple threads to run different instructions in the same clock cycle (i.e., concurrently). It is supposed to be a way to speed up our job, what happened?

One thing to ask is "Does my code use Simultaneous multithreading (SMT)?" 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 do allow for two simultaneous threads, but only with code that takes advantage of it.



Writing code that utilizes SMT can be dangerous as different threads can access the same memory space. Therefore, python implements the global interpreter lock (or GIL) to prevent two threads from accessing the same data.

Always check if your software or code correctly utilizes multithreading.

To avoid confusion, in this module we have chosen to disable SMT by including the nomultithread directive.

Being a Good HPC Citizen

RECAP

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 memory

The computer system is shared with many other users. Last time we saw that we should try to avoid making a mess of shared directories or storing too much data in them!

C. 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. We will be running some big jobs after the break so the system will be heavily utilized – you might have to wait a few minutes.

Let's simulate this on the class reservation. Take a look at the heavyQueue.sl batch script. It is essentially the same as before, except, once running, it just waits 2 minutes. Let's all submit this script at once.

Don't press submit until the entire class is ready

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

$watch squeue --reservation=cpu-s5-grad_778-1_45
```

If you were fast to submit, you'll see your job 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 2 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!

```
#!/bin/bash
#SBATCH --job-name=heavyqueue_job
#SBATCH --output=heavyqueue_job.out
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --time=00:03:00
#SBATCH --cpus-per-task=16
#SBATCH --hint=compute_bound
#SBATCH --mem-per-cpu=2GB
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-1
#SBATCH --mail-user=[yourEMAIL]
#SBATCH --mail-type=ALL

sleep 120
hostname
```

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. This maximizes the utilization of the system!

Fairshare makes Pronghorn equitable, while Backfill maximizes utilization!

Advanced Job Monitoring?

There are several 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).

`$squeue -P`	shows the current jobs ranked by priority
`$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 with tail –f. You can run this from the login nodes while your job is executing.

```
$tail -f myLogfile.txt
```

Hint: you can also use this to watch your jobs in "realtime". We put timeout 30 to make sure you aren't abusing the squeue command. Generally, avoid "watch squeue" unless you really need to do it. It can put a large load on the system.

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

Advanced Job Monitoring?

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.

Break

Varying the number of calculations

Our last code calculated the square of 6 numbers. Let's increase this number now. Find the python script: **AddSquaresMultiThreadingMany.py**

This code takes the square of 8 randomly generated numbers, sums the results and displays them. It essentially works the same as the previous code, but this time, with however many numbers we specify.

Modify your SBATCH script to run this code instead. Set the script to use just 4 cores. Don't forget to put back the --hint=nomultithread

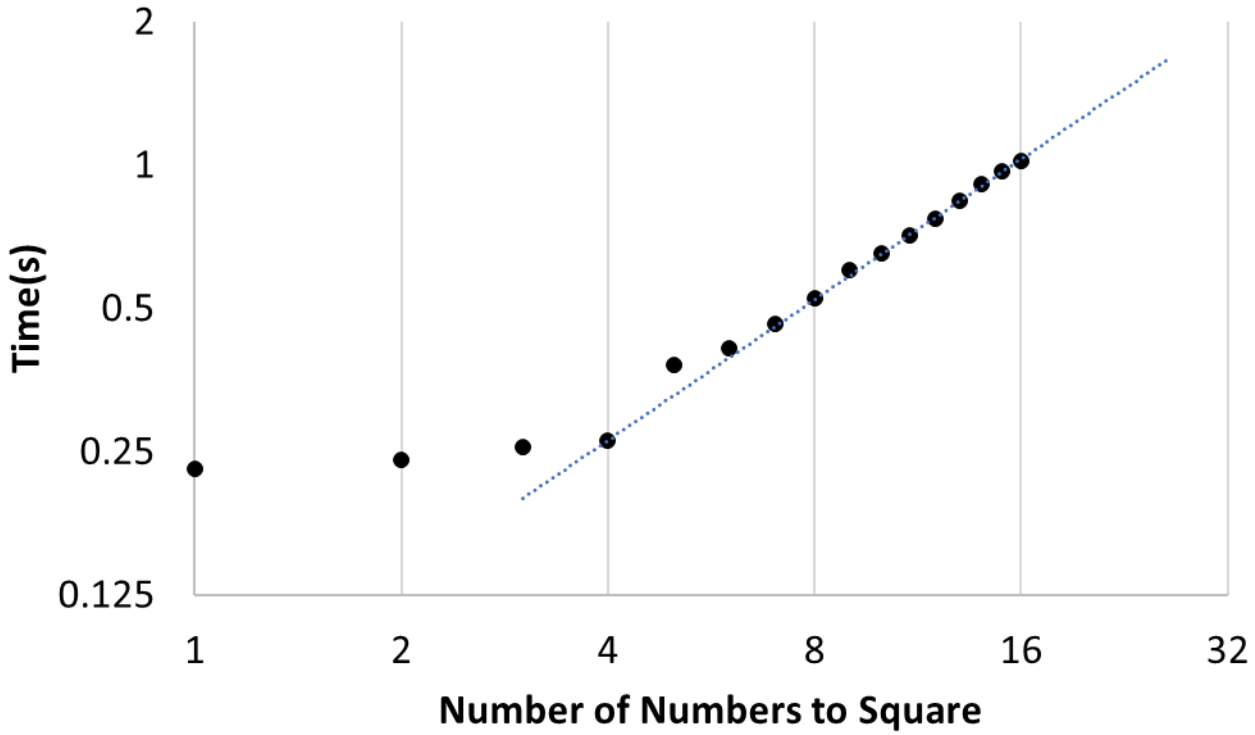
```
#!/bin/bash

#SBATCH --job-name=myMultiProcessingJob
#SBATCH --output=myMultiProcessingJob_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#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 AddSquaresMultiThreadingMany.py
```

Edit the python code to calculate the square of between 1 and 16 numbers. Create a graph showing the time each takes. Plot your results. What do you notice?

If you are getting a memory error, try increasing the memory to 1GB per cpu!

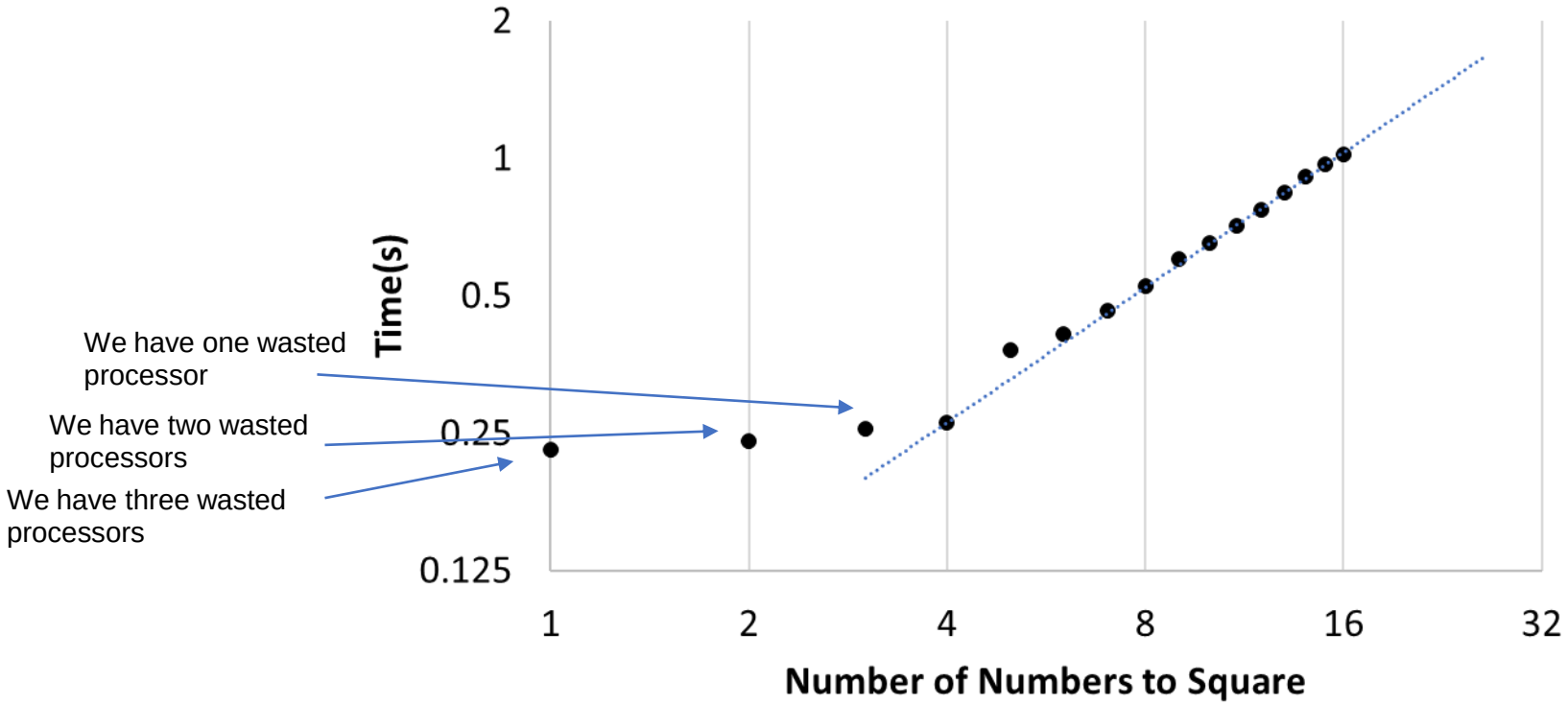
```
import time, os
import threading as th
import multiprocessing as mp
import random

#Create 18 random numbers
NUMBEROFNUMBERS = 8

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=5000000
    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()

    #Numebr List To Proces
    NUMBERLIST = []

    #Create random number to process
    for i in range(0,NUMBEROFNUMBERS):
        n = random.randint(1,10)
        NUMBERLIST.append(n)


    #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)
```

Varying the number of calculations



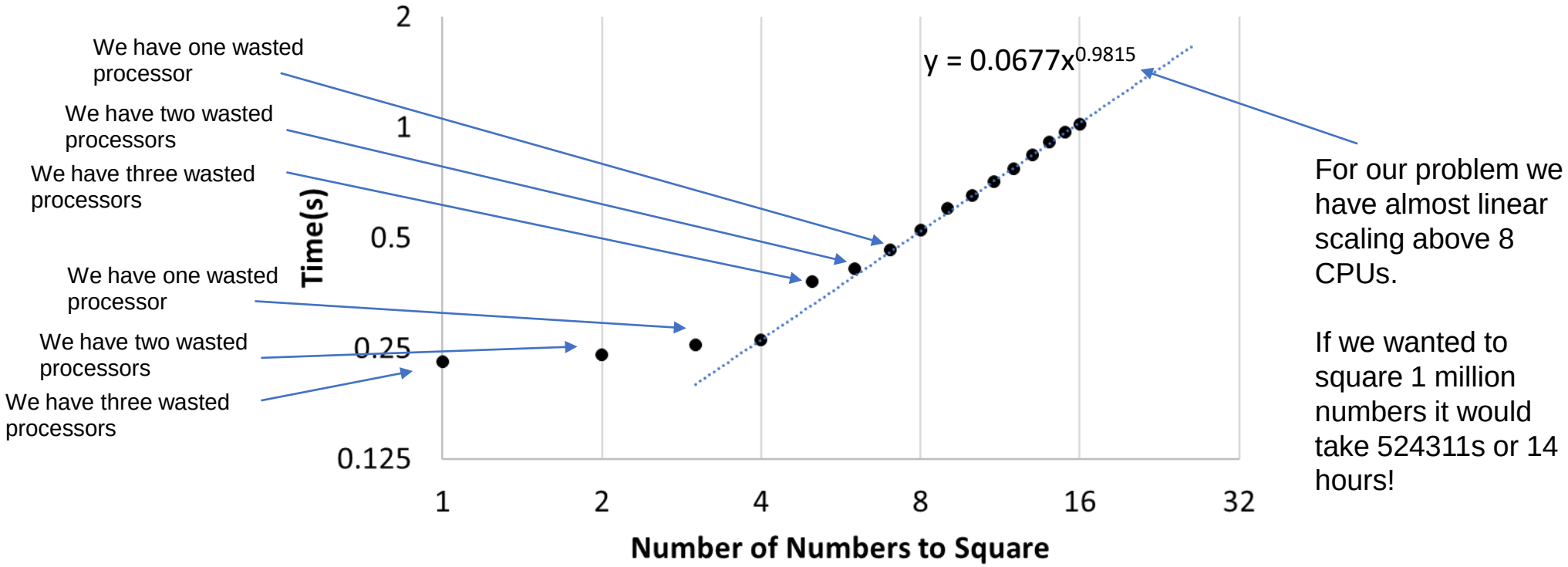
Clearly, performing more calculations takes longer. But there is a huge jump in time once each processor needs to perform multiple calculations. We must always be aware of the overhead in our calculations.

Varying the number of calculations



Clearly, performing more calculations takes longer. But there is a huge jump in time once each processor needs to perform multiple calculations. We must always be aware of the overhead in our calculations.

Varying the number of calculations



For our problem we have almost linear scaling above 8 CPUs.

If we wanted to square 1 million numbers it would take 524311s or 14 hours!

Clearly, performing more calculations takes longer. But there is a huge jump in time once each processor needs to perform multiple calculations. We must always be aware of the overhead in our calculations.

Once you understand the scaling in your problem, you can work out how long it would take for hundreds or even thousands of calculations.

Running Even Bigger Jobs

Now, let's make an even BIGGER job. Edit the python script to run over 128 numbers. **AddSquaresMultiThreadingMany.py**

```
#!/bin/bash

#SBATCH --job-name=myMultiProcessingJob
#SBATCH --output=myMultiProcessingJob_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#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 AddSquaresMultiThreadingMany.py
```

Try timing this larger code with between 1 and 64 CPUs. Plot the speed up. What do you notice?

If you are getting a memory error (or no time output) try increasing the memory to 1000M per cpu!

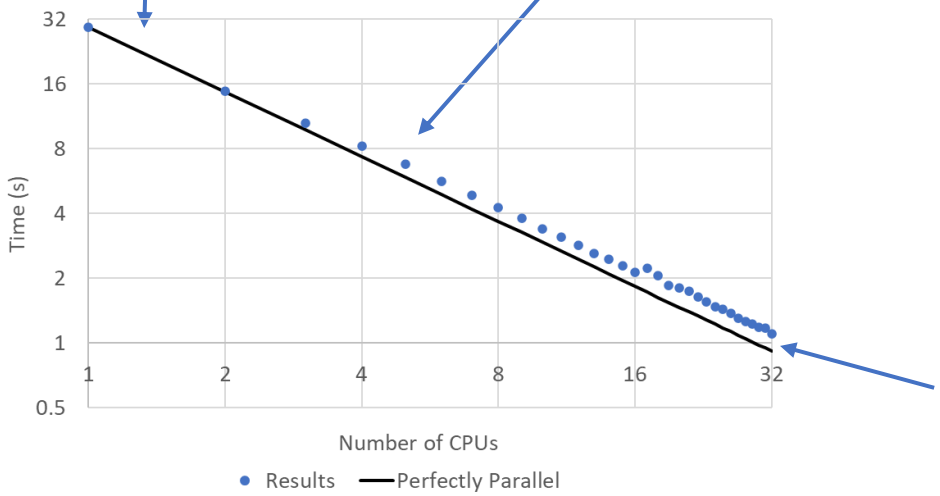
Exercise 5: Larger Parallel Jobs

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

#Create 128 random numbers

NUMBEROFNUMBERS = 128

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=5000000
    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()

    #Numebr List To Proces
    NUMBERLIST = []

    #Create random number to process
    for i in range(0,NUMBEROFNUMBERS):
        n = random.randint(1,10)
        NUMBERLIST.append(n)


    #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)
```

Running Bigger Jobs

In the beginning we have almost perfect parallelization

As we increase the number of CPUs we see a small deviation. This is probably due to overhead in our code (i.e., printing the results to the file)



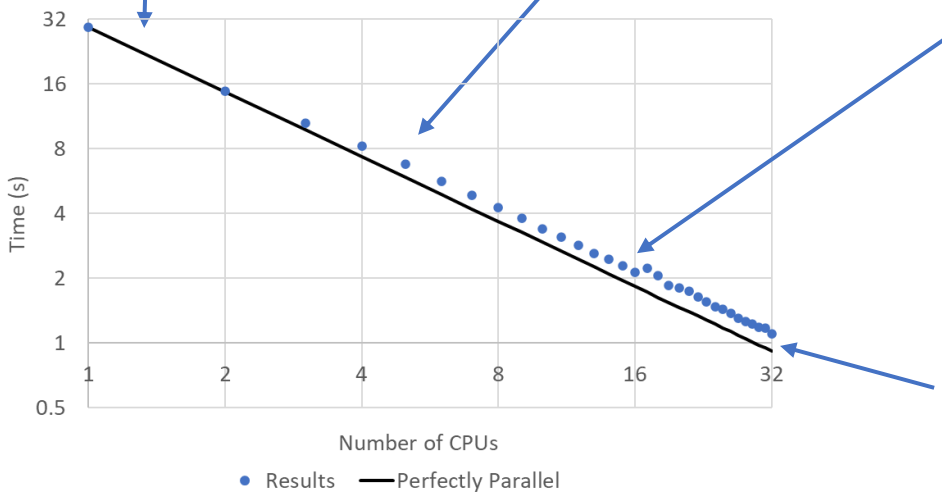
We are not able to run jobs that utilize more than 32 CPUs as each node only has 32 physical cores. To perform calculations on more than one core requires some additional thought (MPI is one way we can achieve this)

Running Bigger Jobs

In the beginning we have almost perfect parallelization

As we increase the number of CPUs we see a small deviation. This is probably due to overhead in our code (i.e., printing the results to the file)

At 17 CPUs we get a decrease in speed. What do you think is happening here?



We are not able to run jobs that utilize more than 32 CPUs as each node only has 32 physical cores. To perform calculations on more than one core requires some additional thought (MPI is one way we can achieve this)

Running Bigger Jobs

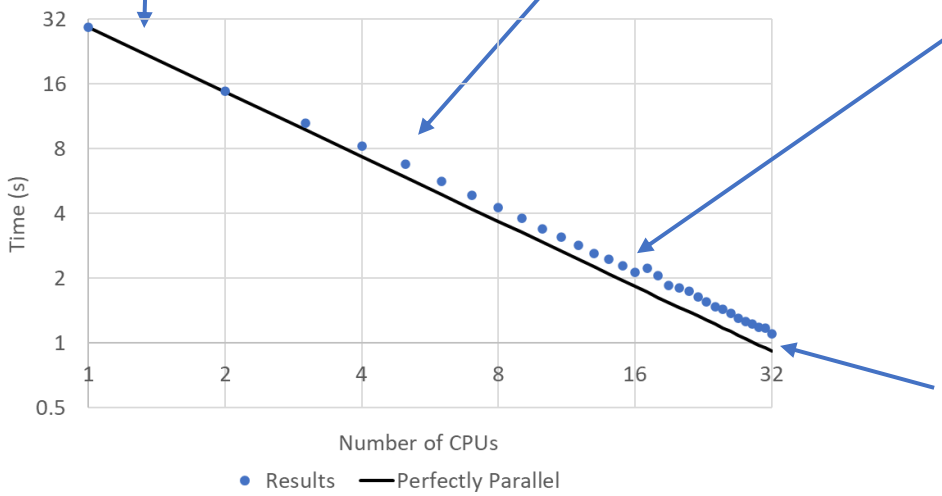
```
NodeName=cpu-77 Arch=x86_64 CoresPerSocket=16
   CPUAlloc=16 CPUTot=64 CPULoad=4.41
   AvailableFeatures=intelv5
   ActiveFeatures=intelv5
   Gres=(null)
   NodeAddr=cpu-77 NodeHostName=cpu-77 Version=20.11.7
   OS=Linux 3.10.0-1160.31.1.el7.x86_64 #1 SMP Thu Jun 10 13:32:12 UTC 2021
   RealMemory=191000 AllocMem=167000 FreeMem=156667 Sockets=2 Boards=1
   MemSpecLimit=24000
   State=MIXED ThreadsPerCore=2 TmpDisk=0 Weight=100 Owner=N/A MCS_label=N/A
   Partitions=cpu-s2-core-0,cpu-core-0,cpu-s3-sponsored-0
   BootTime=2021-09-23T14:52:10 SlurmdStartTime=2021-09-23T15:06:40
   CfgTRES=cpu=64,mem=191000M,billing=250
   AllocTRES=cpu=16,mem=167000M
   CapWatts=n/a
   CurrentWatts=0 AveWatts=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s
   Comment=(null)
```

This node has 64 CPUS = 2 Sockets x 16 Cores Per Socket x 2 Threads Per Core

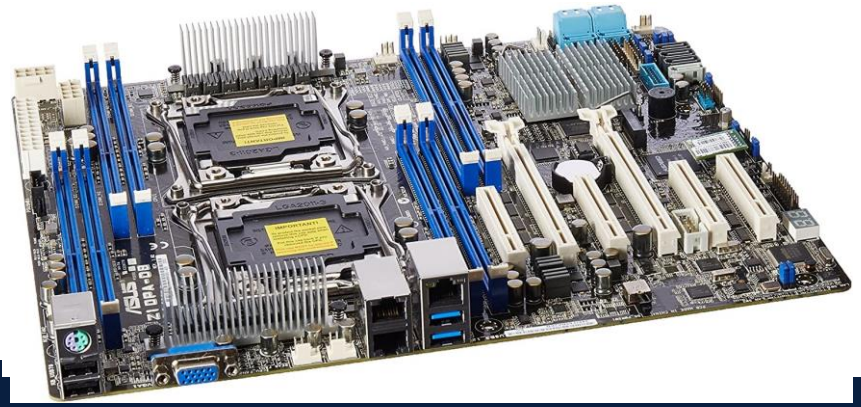
In the beginning we have almost perfect parallelization

As we increase the number of CPUs we see a small deviation. This is probably due to overhead in our code (i.e., printing the results to the file)

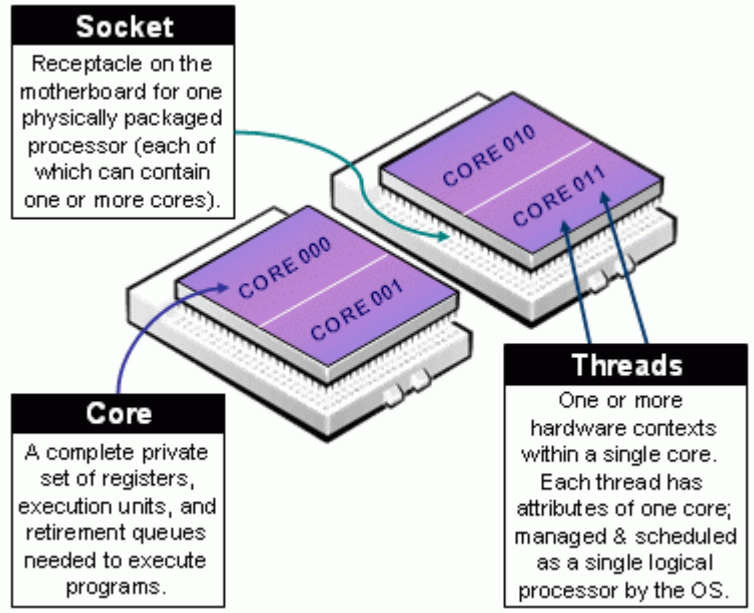
At 17 CPUs we get a decrease in speed. What do you think is happening here?



We are not able to run jobs that utilize more than 32 CPUs as each node only has 32 physical cores. To perform calculations on more than one core requires some additional thought (MPI is one way we can achieve this)



Conclusion: You must have a basic understanding of computer hardware to run jobs efficiently on HPC systems.



Running over multiple nodes

What happens when we ask for more than 32 CPUs for our job?

Running over multiple nodes

What happens when we ask for more than 32 CPUs for our job?

```
-bash-4.2$ sbatch --reservation=cpu-s5-grad_778-1_45 multiprocessing.sl
sbatch: error: Batch job submission failed: Requested node configuration is not available
```

It errors because we asked for 32 CPUs and just one node. Each node only has 32 CPUs!

What if we edit our SBATCH script to use two nodes and ask for 64 CPUs?

Running over multiple nodes

What happens when we ask for more than 32 CPUs for our job?

```
-bash-4.2$ sbatch --reservation=cpu-s5-grad_778-1_45 multiprocessing.sl
sbatch: error: Batch job submission failed: Requested node configuration is not available
```

It errors because we asked for 32 CPUs and just one node. Each node only has 32 CPUs!

What if we edit our SBATCH script to use two nodes and ask for 64 CPUs?

```
-bash-4.2$ sbatch --reservation=cpu-s5-grad_778-1_45 multiprocessing.sl
sbatch: Warning: can't run 1 processes on 2 nodes, setting nnodes to 1
sbatch: error: Batch job submission failed: Requested node configuration is not available
```

It still doesn't work! SLURM won't let us run our code over multiple nodes. This is because our python script uses the multiprocessing package, which does not work across multiple nodes.

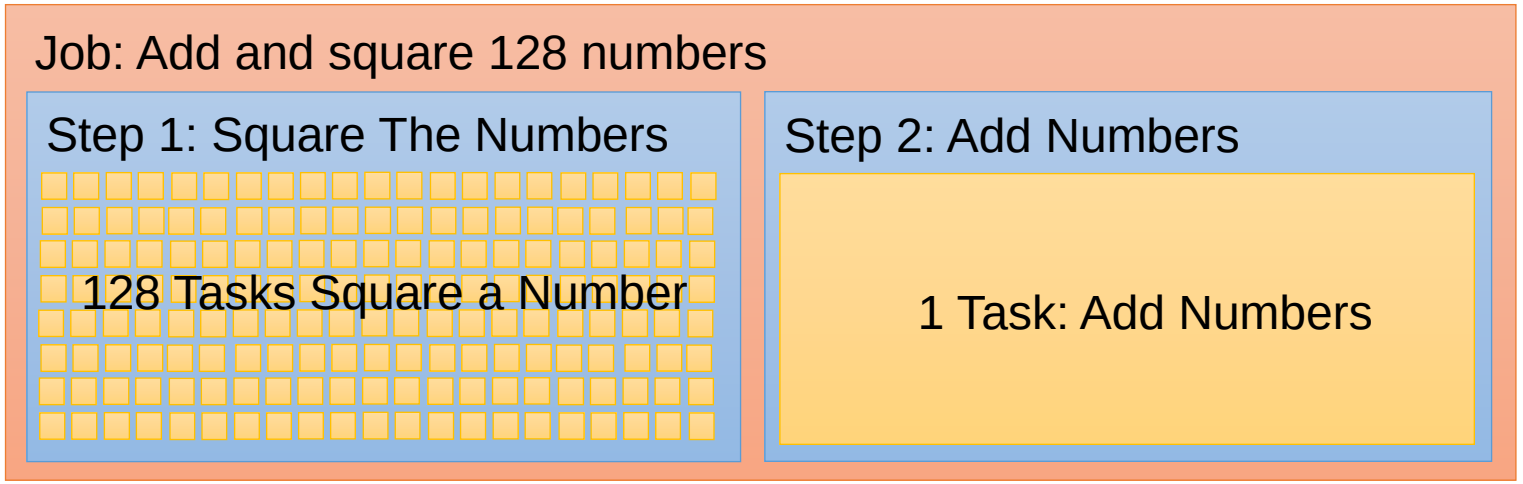
In general, getting code to work across multiple nodes is extremely difficult. However, there is one quite easy to parallelize over more than one node.

This is to parallelize over tasks!

Parallelizing over tasks

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.



Let's return to our SBATCH script. Up to now we have always set the ntasks value to one. i.e., our job has consisted of performing a single task with multiple CPUs assigned to that task.

Now we will look at using multiple tasks, but only assigning one CPU to each task. However, to do this, each task must be independent from each other.

```
#!/bin/bash

#SBATCH --job-name=myMultiProcessingJob
#SBATCH --output=myMultiProcessingJob_output.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#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 AddSquaresMultiThreadingMany.py
```

Break

Parallelizing over tasks

A simple use case for this type of parallelization would be if you had a large dataset and you wanted to perform an analysis of each piece of data.

First take a look at the SquareMyNumber.py code

```
$more SquareMyNumber.py
```

Try running the code on the login node

```
$python SquareMyNumber.py 5
```

```
-bash-4.2$ python SquareMyNumber.py 5
Square is:  25  Time taken is:  0.1957864761352539
```

It should take about 0.2 seconds to square our number.

Up to now we have been running code that is designed to take advantage of multiple CPUs. Notice that this time our code is purely serial. It does not contain either the multiprocessing or multithreading packages!

SquareMyNumber.py

```python
import time, os
import random
import sys

#Start Timer
start = time.time()

#Create Empty Array For Results
number = int(sys.argv[1])

#Calculate the Square of Number
result = number * number

#Waste time counting to a very large number ;)
n=13000000
while n>0:
    n -= 1

#Stop Timer
end = time.time()

#Print Squre of Number and Time Taken
print(f"Square is: ", str(result), " Time taken is: ", end-start)
```

Parallelizing over tasks

Take a look in the taskPar.sl SBATCH script.

```
$more taskPar.sl
```

This is the SBATCH script which will be able to run multiple copies of our python code. Notice that we have included a loop here which loops through the numbers 1 to 64 and passes these numbers into the python script. Try submitting the SBATCH script:

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

Once it's running, watch the output in real time. Notice that it's completing the tasks one at a time.

```
$tail -f myOutput.txt
```

Q. How long did it take to run?

A. Around 20 seconds. Does this make sense?

64 numbers * 0.2s per number = 12.8s

taskPar.sl

```
#!/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=100M
#SBATCH --hint=nomultithread
#SBATCH --time=00:01:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-1
#SBATCH --mail-user=YOUREMAILHERE
#SBATCH --mail-type=ALL

module load python3

start=$(date +%s.%N)

for i in $(seq 1 64)
do
    srun --ntasks=1 --nodes=1 --cpus-per-task=1 --exclusive python SquareMyNumber.py $i
done



end=$(date +%s.%N)

runtime=$(python -c "print(${end} - ${start})")

echo "Runtime was $runtime"
```

The reason the code is taking so long is because for each task there is a considerable overhead. Each task needed to start up python, run the code, then close python. The longer our job, the less important this overhead would be.

Parallelizing over tasks

Now try increasing the number of tasks in the job.
Try ntasks=2. What do you notice? Is it any faster?

Parallelizing over tasks

Now try increasing the number of tasks in the job. Try ntasks=2. What do you notice? Is it any faster?

In order to get the tasks to start in parallel we need to end the srun line with an ampersand (i.e., &).

In addition, we need to add the wait command to tell the script to wait here until all the tasks started with an & have finished.

Add the & and the wait command to the SBATCH script.

Now try increasing the number of tasks and see if you can observe any speed up. How fast can you make it?

Can you get it to run on 64 CPUs across two nodes?

Is your file getting filled up too much output? Try using the grep command to extract the lines we are interested in:

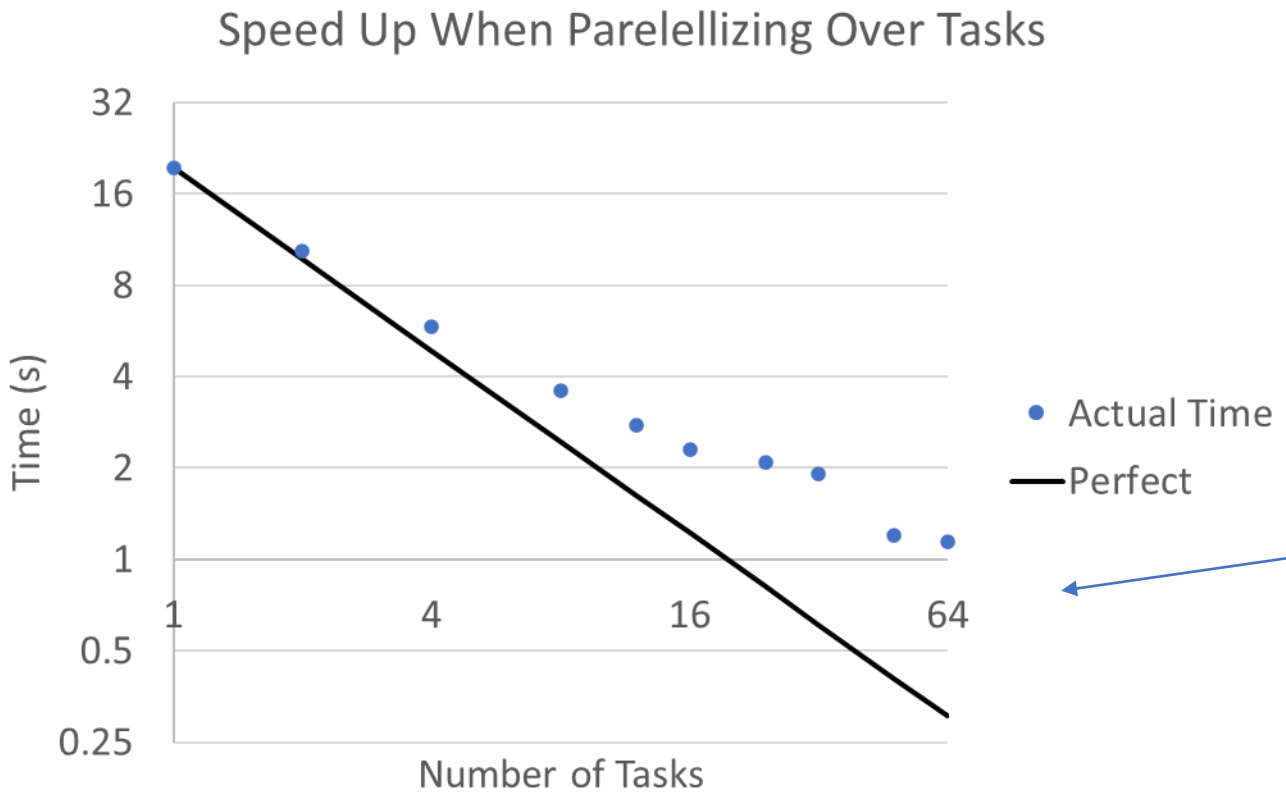
```
$grep 'Runtime' myOutput.txt
```

taskPar.sl

```
#!/bin/bash

#SBATCH --job-name=myFirstJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=2
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=100M
#SBATCH --hint=nomultithread
#SBATCH --time=00:01:00
#SBATCH --partition=cpu-core-0
#SBATCH --account=cpu-s5-grad_778-1
#SBATCH --mail-user=YOUREMAILHERE
#SBATCH --mail-type=ALL

module load python3

start=$(date +%s.%N)

for i in $(seq 1 64)
do
    srun --ntasks=1 --nodes=1 --cpus-per-task=1 --exclusive python SquareMyNumber.py $i &
done

wait

end=$(date +%s.%N)

runtime=$(python -c "print(${end} - ${start})")

echo "Runtime was $runtime"
```

WARNING: We are chaging ntasks NOT cpus-per-task

What does our speed-up plot look like?



For the first time, we were able to get the program to use more than 32 CPUs

Look how much faster we made it!

The curve is certainly not perfect. The computer hardware and divisibility of our program both affect the shape of the curve.

This exercise demonstrates how important it is to profile your code on the machine you are using.

What is this message we are getting?

Some of you will have noticed a message being printed in our output. It says something about step creation being temporarily disabled. What does this mean?

In this example I have given SLURM 5 numbers to square but just one task allowed at a time. After the first task starts, the remaining four tasks must wait their turn. That is what we are seeing here.

If the number of jobs is more than the number of tasks it has to wait until a CPU is available to run more tasks.

If the number of tasks to perform is matched to the number of tasks allocated by the SLURM scheduler, we would not see this error message.

```
-bash-4.2$ more myOutput.txt
Time is Tue Nov  2 23:05:25 PDT 2021
Square is:  9  Time taken is:  1.0625548362731934
srun: Job 2833586 step creation temporarily disabled, retrying (Requested nodes are busy)
srun: Job 2833586 step creation temporarily disabled, retrying (Requested nodes are busy)
srun: Job 2833586 step creation temporarily disabled, retrying (Requested nodes are busy)
srun: Job 2833586 step creation temporarily disabled, retrying (Requested nodes are busy)
srun: Step created for job 2833586
Square is:  1  Time taken is:  1.495072364807129
srun: Job 2833586 step creation still disabled, retrying (Requested nodes are busy)
srun: Job 2833586 step creation still disabled, retrying (Requested nodes are busy)
srun: Step created for job 2833586
srun: Job 2833586 step creation still disabled, retrying (Requested nodes are busy)
Square is:  4  Time taken is:  1.016106128692627
srun: Job 2833586 step creation still disabled, retrying (Requested nodes are busy)
srun: Job 2833586 step creation still disabled, retrying (Requested nodes are busy)
srun: Step created for job 2833586
Square is:  16  Time taken is:  1.0165317058563232
srun: Job 2833586 step creation still disabled, retrying (Requested nodes are busy)
srun: Step created for job 2833586
Square is:  25  Time taken is:  1.19154953956604
Time is Tue Nov  2 23:05:31 PDT 2021
```

The best of both worlds?

In our first example, we used a multiprocessing python code and told the SLURM scheduler to run a single task. However, we could make multiple CPUs (on a single node) available to that task. This type of parallelization is what is called Shared-Memory Parallelism (SMP).

In SMP, the work is divided between multiple threads or processes running on a single machine.

In SMP, each CPU has access to the same shared main memory (think RAM). These CPUs can share information through this shared memory. For example, that is how we could add up all the squared numbers at the end of our program – they were all stored in the shared memory.

However, 'main memory' is only available to the machine/node that it is connected to. On Pronghorn, this means that we are limited to running on a single node. Nodes can't access each other's main memory.

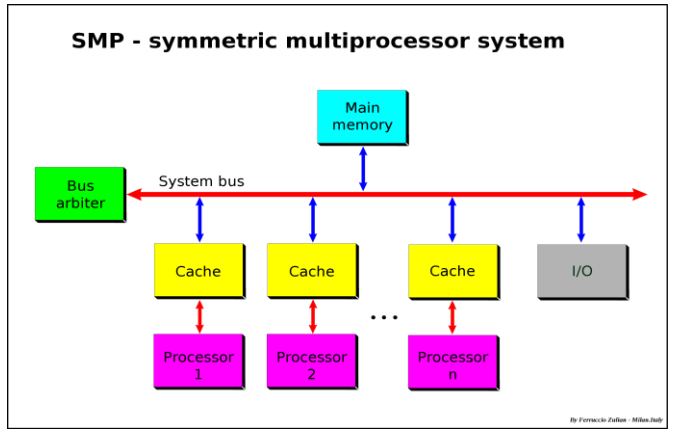


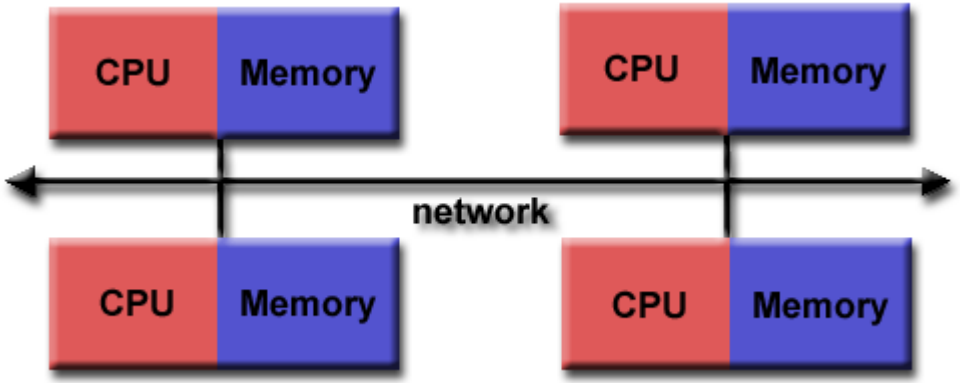
Diagram of a shared memory multiprocessing system from Wikipedia.

In our second example, we used a serial python code and told the SLURM scheduler to run multiple tasks. This allowed us to run over multiple nodes, but there was no way for each task to communicate with each other!

What is we had a problem where each process needed to communicate with other processes (i.e., share information), but we also want to be able to run on more than one node?

The answer is distributed memory!

Distributed Memory + MPI



In a distributed memory system, each process has its own memory allocation.

Therefore, information cannot be shared via the memory. Instead, information is shared over a network. This is known as inter-process communication (IPC).

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.

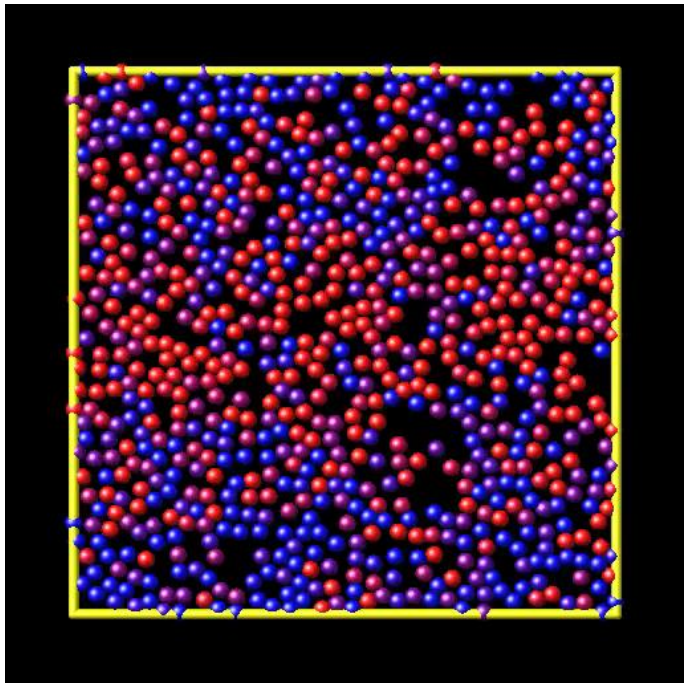
Let's check the MPI libraries available on Pronghorn:

```
$module avail
```

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

Coding in MPI is quite difficult and will certainly require years of practice and experience. If you are interested in coding using MPI then I recommend: https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

Getting Software on Pronghorn



The most typical use case that you will encounter is probably running software that someone else has written.

For example, in my own research, I regularly use the software package LAMMPS (www.lammps.org).

LAMMPS is a very common molecular dynamics software used by thousands of researchers every day. It is used to simulate the motions of atoms interacting through some predefined potential.

The details of the software do not matter here. What is important is that it is extremely well parallelized through the MPI formalism.

Let's say we want to run a simulation using this software package. How do we get this software onto Pronghorn?

You either have to 1) ask the Pronghorn admins if they'd install it for you (hint: the answer will be NO), or 2) compile it from scratch including all required libraries (never a fun approach).

There is a third option, use a container!

What is a container?

A "package" (file) that contains all the software and its dependencies INCLUDING the operating system it runs on. This allows you to package-up an application (and data!) in a single file and share that file with other people allowing an identical execution environment on other machines.

Singularity vs. Docker

The most common container system is Docker (https://www.docker.com/). We are using Singularity (https://sylabs.io/).

One of the main reasons we use Singularity is that, unlike Docker, Singularity does not require superuser (root) privileges to run. On an HPC, we don't want each user to be able to nuke the entire system. Second, Docker containers do not (really) support MPI, whereas Singularity has built-in support for MPI. HOWEVER, as we will see, we can easily convert from a Docker container to a Singularity container.

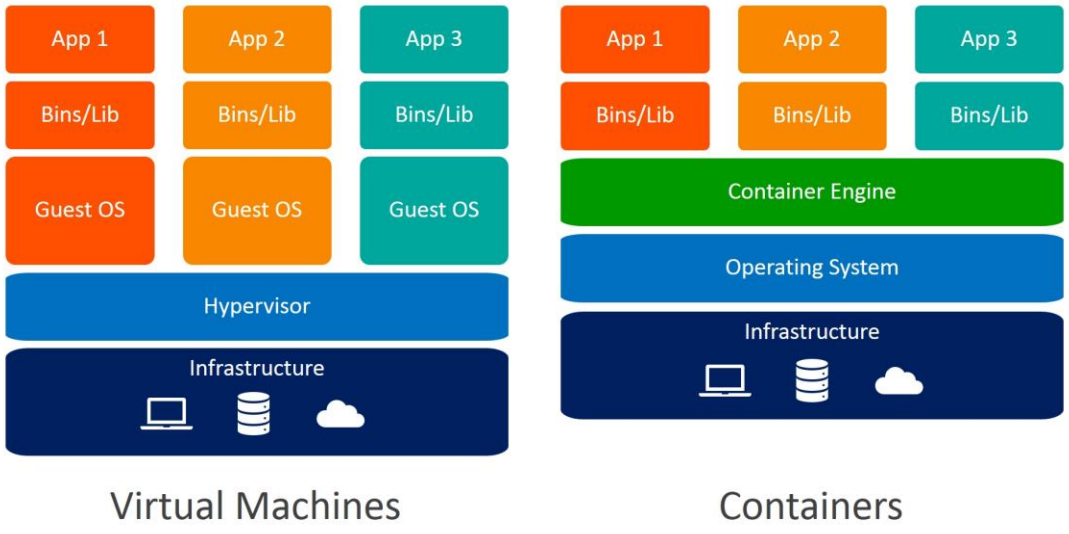
How is this different from a virtual machine?

A virtual machine connects a "guest operating system" with the apps you want to run to the true hardware via a "hypervisor". These are full operating systems, each VM needs its own reserved resources (e.g. CPU, RAM). They have to "boot" just like any computer.

A container does not have a hypervisor, and each container contains the minimum dependencies needed to run (bins and libs).

Containers start in milliseconds and save disk, RAM and CPU resources. VMs take time to boot.

For more detail check out: https://www.electronicdesign.com/dev-tools/what-s-difference-between-containers-and-virtual-machines



Virtual Machines Containers

Exercise 8: Investigate the docker website

How do I get a container?

We can build containers in three basic ways:

1) Download/bootstrap from an existing container (e.g. from Docker)
2) Using a definition file
3) Using a sandbox

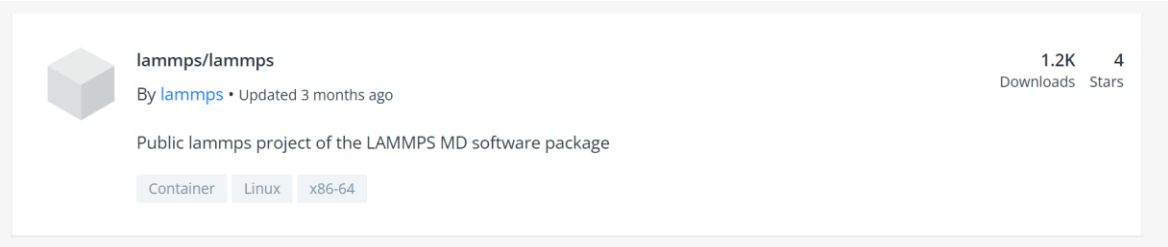
We are going to use the first method, because it is the easiest. However, method two will also work on Pronghorn. The third method (Sandbox) requires a more complex setup -- you will need to install singularity locally on your own computer, which is beyond the scope of this class.

Singularity has an easy-to-use "conversion" from Docker containers, which are far more common, that can be downloaded and built on Pronghorn.

First, let's take a look at Docker Hub at https://hub.docker.com/

Perform a search for ‘lammps’.

Find the “Public lammps project of the LAMMPS MD software package” and click on it.



Make note of the name of the file “lammps/lammps” and also the contents of the file.

Contents

- `lmp_serial` - Serial executable of LAMMPS
- `lmp_mpi` - MPI executable of LAMMPS
- `lammps-shell` - Interactive serial executable of LAMMPS
- `liblammps_serial.so` - serial shared library of LAMMPS
- `liblammps_mpi.so` - MPI shared library of LAMMPS
- `lammps` Python module

Container Example

Go back to Pronghorn and use the singularity `pull` command to download the LAMMPS container.

```
$singularity pull docker://lammps/lammps
```

If you do not have access to singularity, you may need to install it.

```
$module load singularity
```

After a while, you should see the container in your directory. You can identify it with the '.sif' extension.

It will be called something like 'lammps_latest.sif'.

Inside this container are the binary files to run the program. The contents list from the docker website gives us the names of these files.

In order to run the files that are inside the container we use the singularity exec command. For example, in order to run the MPI executable of LAMMPS we would type:

```
$singularity exec lammps_latest.sif lmp_mpi
```

The first thing LAMMPS does is print out the date this version of LAMMPS was written. Even though there are lots of errors, we are able to see that the program did run!

```
-bash-4.2$ singularity exec lammps_latest.sif lmp_mpi
libibverbs: Warning: couldn't open config directory '/etc/libibverbs.d'.
LAMMPS (30 Jul 2021)
OMP_NUM_THREADS environment is not set. Defaulting to 1 thread. (src/comm.cpp:98)
  using 1 OpenMP thread(s) per MPI task
```

Contents

- `lmp_serial` - Serial executable of LAMMPS
- `lmp_mpi` - MPI executable of LAMMPS
- `lammps-shell` - Interactive serial executable of LAMMPS
- `liblammps_serial.so` - serial shared library of LAMMPS
- `liblammps_mpi.so` - MPI shared library of LAMMPS
- `lammps` Python module

Running Our Program Using MPI

I have provided you with an SBATCH script (mpiBatch.sl) which runs the LAMMPS software. The SBATCH script has a few interesting features we have not seen before.

1. We turn on simultaneous multithreading (SMT) as LAMMPS is able to utilize multiple threads.
2. Each task is assigned two CPUs which are our two threads.
3. We must tell LAMMPS how many threads to use.
4. We must specify which version of MPI was used to compile LAMMPS and load that compiler (this gives us access to the mpirun command).
5. We always load singularity

mpiBatch.sl

```
#!/bin/bash

#SBATCH --job-name=myFirstMPIJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#SBATCH --hint=multithread
#SBATCH --mem-per-cpu=100M
#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

export OMP_NUM_THREADS=2

module load openmpi/gcc/4.0.4
module load singularity

mpirun singularity exec lammps_latest.sif lmp_mpi < input.lammps
```

Try running the LAMMPS executable that is inside the singularity container.

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

While the software is running, use the tail –f command to watch the output.

```
$tail –f myOutput.txt
```

After about a minute you should see the program finish and write the total amount of time to the screen. Mine took around 49 seconds to run.

```
Total wall time: 0:00:49
```

Running Our First Program with MPI

Edit the SBATCH script and increase the number of tasks (ntasks).

Try using 1, 2, 4, 8, 16, 32 and 64.

Note, when requesting more than 32 tasks you will need to request two nodes.

What is the fastest you can you make the program?

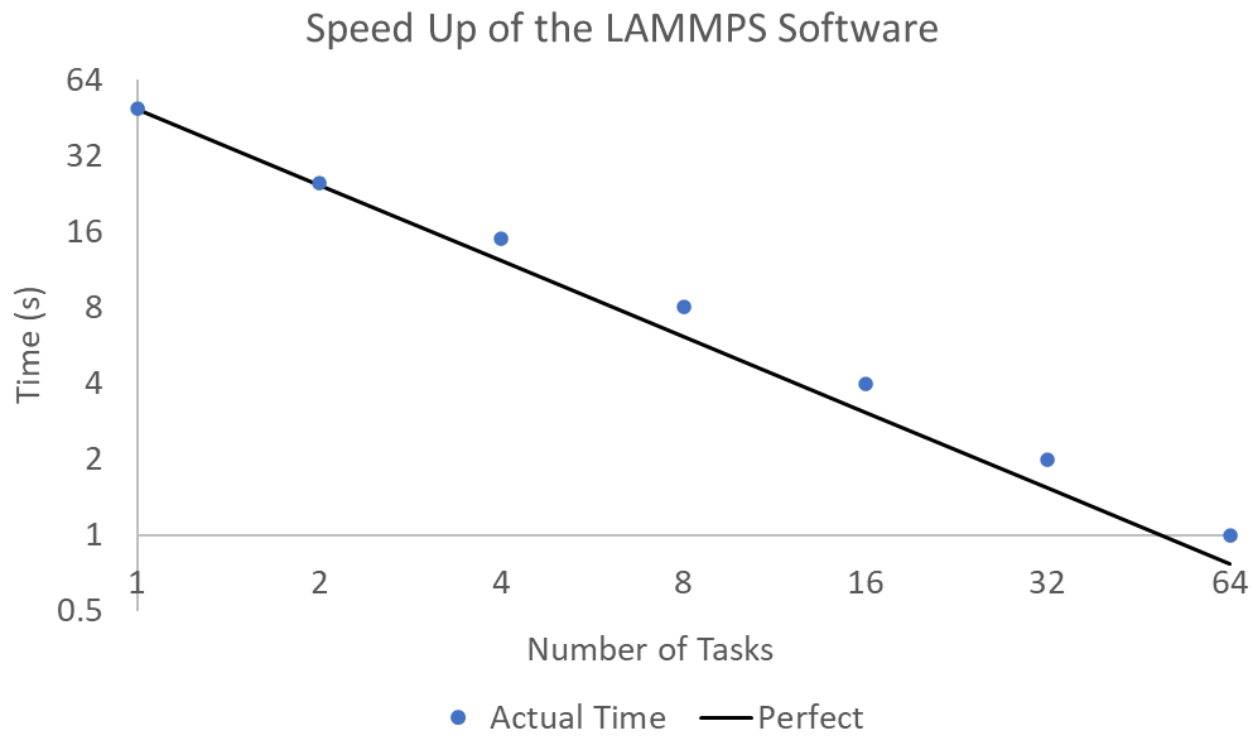
Quiz: In order to pass this class, please upload a file/image of your graph showing the decrease in time it takes the LAMMPS software to run as we increase the number of cores.

mpiBatch.sl

```
#!/bin/bash

#SBATCH --job-name=myFirstMPIJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#SBATCH --hint=multithread
#SBATCH --mem-per-cpu=100M
#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

export OMP_NUM_THREADS=2

module load openmpi/gcc/4.0.4
module load singularity

mpirun singularity exec lammps_latest.sif lmp_mpi < input.lammps
```

Quiz

Quiz: In order to pass this class, please upload a file/image of your graph showing the decrease in time it takes the LAMMPS software to run as we increase the number of cores.



Which of these do you think will run faster?

These two scripts both utilize 8 logical processors / threads.
Which do you think would run the fastest?

```
#!/bin/bash

#SBATCH --job-name=myFirstMPIJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=2
#SBATCH --hint=multithread
#SBATCH --mem-per-cpu=100M
#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

export OMP_NUM_THREADS=2

module load openmpi/gcc/4.0.4
module load singularity

mpirun singularity exec lammps_latest.sif lmp_mpi < input.lammps
```

```
#!/bin/bash

#SBATCH --job-name=myFirstMPIJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --hint=nomultithread
#SBATCH --mem-per-cpu=100M
#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

export OMP_NUM_THREADS=1

module load openmpi/gcc/4.0.4
module load singularity

mpirun singularity exec lammps_latest.sif lmp_mpi < input.lammps
```

Which of these do you think will run faster?

These two scripts both utilize 8 logical processors / threads.
Which do you think would run the fastest?

```
#!/bin/bash

#SBATCH --job-name=myFirstMPIJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=2
#SBATCH --hint=multithread
#SBATCH --mem-per-cpu=100M
#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

export OMP_NUM_THREADS=2

module load openmpi/gcc/4.0.4
module load singularity

mpirun singularity exec lammps_latest.sif lmp_mpi < input.lammps
```

15 s

```
#!/bin/bash

#SBATCH --job-name=myFirstMPIJob
#SBATCH --output=myOutput.txt
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --hint=nomultithread
#SBATCH --mem-per-cpu=100M
#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

export OMP_NUM_THREADS=1

module load openmpi/gcc/4.0.4
module load singularity

mpirun singularity exec lammps_latest.sif lmp_mpi < input.lammps
```

13 s

Even though LAMMPS allows for multithreading, it doesn't necessarily mean it will run faster.
In this case, the overhead associated with multithreading makes this version run slower!

End