UNIVERSITY OF WASHINGTON
Paul G. Allen School of Computer Science and Engineering

CSE P 524 **Assigned:** 3/8/2022
Problem Set #10 (Prelim) **Due:** 3/18/2022

1 Preliminaries

1.1 Shared Computing Resources and the Hyak Compute Cluster

For the final assignment this quarter, we will be using the UW Hyak compute cluster.

Compute clsuters are a fairly standard operational model for high-performance computing, typically consisting of a set of front-end resources (aka front-end nodes aka login nodes) and then a (much larger) numer of dedicated back-end resources (aka compute nodes). The front-end resources are shared – multiple users can login and use them. The compute resources, where the real computing is done, however are batch scheduled via a queueing system. Only one job will be run on a particular resource at a time.

Compute resources are often heterogeneous, as is the case with Hyak, with different capabilities being available on different installed hardware. When a user wishes to use a particular class of resource, that resource is requested specifically when the job is submitted to the queueing system.

Administratively, Hyak is a "condo cluster", meaning different resources have been purchased under the auspices of different research projects, but the day-to-day operation and maintenance is amortized by collecting them all into a single administrative domain. The resources associated with a particular project (or research group) are available for that project (group) on demand. However, idle resources are put into a pool for general use.

We will be using Hyak to submit distributed memory jobs for this assignment. For this course we have 80 CPU cores and 370GB of RAM as dedicated resources. Students at UW are also eligible to use resources that are part of the student technology fund by joining the Research Computing Club (I encourage everyone to join the RCC). The RCC web pages have additional information on how to access and use Hyak.

If you already have access to Hyak via your research group, you can indicate which group's resources to use when submitting your job. You can find the resources available to you at any time by issuing the command "hyakalloc".

Note that the main login node for hyak is klone.hyak.uw.edu. When we say "connect to Hyak" we mean connect to (log in to) klone.hyak.uw.edu.

1.2 Account Activation

An account has been created for you on Hyak for this course. To activate your account, including setting up two-factor authentication (2FA), follow the instructions at https://hyak.uw.edu/docs/account-activation. (See also the RCC instructions.)

1.3 Connect with VS Code

The easiest (and best) way to work on Hyak is to connect with VS Code. This will provide a fairly seamless transition—you will be able to edit your files and work with the command line in pretty much the same way you have been so far.

- Install the Remote SSH extension for vs code
- Use the Remote SSH extension to connect to klone.hyak.uw.edu

You will be able to edit files on Hyak directly on your laptop and interact with Hyak through the vs code terminal.

1.4 Some Rules

The Hyak cluster is a shared compute resource, which sharing includes the compute resources themselves, as well as the file system and the front-end (login) node. When you are logged in to the head node, other students will also be logged in at the same time, so be considerate of your usage. In particular,

- Use the cluster only for this assignment (no mining bitcoins or other activities for which a cluster might seem useful),
- Do not run anything compute-intensive on the head node (all compute-intensive jobs should be batch scheduled),
- Do not put any really large files on the cluster, and
- Do not attempt to look at other students' work.

1.5 Sanity Check

Once you are connected to the head node, there are a few quick sanity checks you can do to check out (and learn more about) the cluster environment.

To find out more about the head node itself, try

```
$ uname -a
$ lscpu
$ more /proc/cpuinfo
```

These will return various (and copious) information about the OS and cores *on the head node*. Of interest are how many cores there are.

You can see the resources that are available to you on Hyak with the 'hyakalloc' command

```
$ hyakalloc
```

You should get a response back similar to

```
Account-Partition - Resource            Total           Used           Free
---------------------------             -----           ----           ----
niac
   - CPU cores                             80              0             80
   - Memory (GB)                          370              0            370
```

("niac" is the account group that you are assigned to as part of this course. If you belong to other groups you may see their resources as well.)

1.6 Batch Resources

We will be using the "SLURM" batch queuing system for this assignment, which we will discuss in more detail below. To get some information about it, you can issue the command

```
$ sinfo
```

You can get an organized summary of nodes in the cluster with the command

```
$ sinfo -o "%20N  %10c  %10m  %25f  %10G"
```

This will print a summary of the compute nodes along with some basic details about their capabilities.

You should see a list that looks something like this

```
NODELIST              CPUS        MEMORY      AVAIL_FEATURES              GRES
n[3008-3011,3064,306  40          386048      bigmem,cascadelake          (null)
z3000                 40          191488      2080ti,cascadelake,gpu,gp   gpu:2080ti
n[3000-3007,3016-302  40          773120      cascadelake,hugemem         (null)
z3001                 40          191488      2080ti,cascadelake,gpu,gp   gpu:2080ti
n[3012-3015,3024-306  40          176128+     cascadelake                 (null)
```

(Use "man sinfo" to access documentation to explain the different columns.) The command "squeue" on the other hand will provide information about the jobs that are currently running (or waiting to be run) on the nodes.

```
$ squeue
$ squeue -arl
```

("man squeue" to access information about the squeue command.)

The basic paradigm in using a queuing system is that you request the queueing system to run something for you. The thing that is run can be a program, or it can be a command script (with special commands / comments to control the queueing system itself). Command line options (or commands / directives in the script file) are used to specify the configuration of the environment for the job to run, as well as to tell the queueing system some things about the job (how much time it will need to run for instance). When the job has run, the queueing system will also provide its output in some files for you – the output from cout and the output from cerr.

A reasonable tutorial on the basics of using slurm can be found here: https://www.slurm.schedmd.com/tutorials.html. The most important commands are: "srun", "sbatch", "sinfo", and "squeue". The most important bits to pay attention to in that tutorial are about those commands.

A brief overview of slurm on Hyak can be found https://hyak.uw.edu/docs/compute/scheduling-jobs

1.7 srun

There are two modes of getting SLURM to run your jobs for you – pseudo-interactive (using "srun") and batch (using "sbatch"). The former ("srun") should only be used for short-running jobs (testing, debugging, sanity checks). Using it for verifying your programs for the first part of this assignment is fine. Note that you are still using a slot in the queue even with srun – so if your job hangs, it will block other jobs from running (at least until your timeslot expires).

The "look and feel" of srun is alot like just running a job locally. Instead of executing the job locally with "./norm_parfor.exe" (for example), you simply use "srun":

```
$ srun ./norm_parfor.exe
         N  Sequential    1 thread   2 threads   4 threads   8 threads      1 thread     2 threa
   1048576     5.76266     5.72983     11.4924     22.8542     50.2792             0   5.19496e-
   2097152     5.47874     5.46393      10.811     21.7382     42.1178             0   6.80033e-
   4194304     4.05311     4.06923     7.84222     15.6246     31.9816             0   4.80808e-
   8388608     3.29741      3.3026     6.39376     12.9454     24.6724             0   1.11505e-
```

(Note that the performance is significantly better than what you may have been seeing so far on your laptop.)

Try the following with "srun"

```
$ hostname
$ srun hostname
```

The former will give you the name of the front-end node while the latter should give you the hostname of the compute node where SLURM ran the job – and they should be different. (Had better be different!)

Note that "srun" is also the command we will use in upcoming assignments to launch distributed-memory (MPI) jobs. You can pass it an argument to tell it how many separate jobs to run.

```
$ srun -n 4 hostname
```

You shouldn't use this option until we start doing MPI programs.

The srun command takes numerous other options to specify how it should be run (specific resources required, time limits, what account to use, etc.) We will cover these as needed but you can

1.8 sbatch

"srun" basically just takes what you give it and runs it, in a pseudo-interactive mode – results are printed back to your screen. However, jobs invoked with "srun" are still put into the queue – if there are jobs ahead of you, you will have to wait until the job runs to see your output.

It is often more productive to use a queueing system asynchronously – especially if there are many long-running jobs that have to be scheduled. In this mode of operation, you tell SLURM what you want to do inside of a script – and to submit the script to the queuing system.

For example, you could create a script hello_script.bash as follows:

```
#!/bin/bash
echo "Hello from Slurm"
hostname
echo "Goodbye"
```

This file is in your repo as warmup/hello_script.bash. Submit it to the queuing system with the following:

```
% sbatch hello_script.bash
```

NB: Files that are submitted to "sbatch" must be actual shell scripts – meaning they must specify a shell in addition to having executable commands. The shell is specified at the top of every script, starting with "#!". A bash script therefore has "#!/bin/bash" at the top of the file.

If you submit a file that is not a script, you will get an error like the following:

```
sbatch: error: This does not look like a batch script.  The first
sbatch: error: line must start with #! followed by the path to an interpreter.
sbatch: error: For instance: #!/bin/sh
```

("#!" is pronounced in various ways – I usually pronounce it as "crunch-bang", though "pound-bang" or "hash-bang" are also used. "#!" serves as a *magic number* (that is the technical term) to indicates scripts in Unix / Linux systems.

Output from your batch submission will be captured in a file 'slurm-¡some number¿.out'. The script above would produce something like

```
Hello from SLURM!
n3164
Goodbye
```

Job options can be incorporated as part of your batch script (instead of being put on the command line) by prefixing ach option with #SBATCH.

For example the following are typical options to use in this course

```
#!/bin/bash

#SBATCH --account=niac
#SBATCH --partition=niac
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
#SBATCH --time=00:05:00
```

Here, we request from the niac account and niac partition, 1 core on 1 node, and a time limit of 5 minutes. (Another partition that can be used is the ckpt partition, which is the resource pool of available idle resources.)

If you see an error such as

```
slurmstepd: error: execve(): hello_script.bash: Permission denied
```

you may need to set the executable bit for the script.

```
$ chmod +x hello_script.bash
```

You can verify that the executable bit is set by doing a long listing of the file:

```
$ ls -l hello_script.bash
-rw-r--r-- 1 al75 al75 62 May 30 17:25 hello_script.bash
```

The script is not executable. So we issue use "chmod +x"

```
$ chmod +x hello_script.bash
$ ls -l hello_script.bash
-rwxr-xr-x 1 al75 al75 62 May 30 17:25 hello_script.bash
```

1.9 Program Output

When you use sbatch, rather than sending your output to a screen (since sbatch is not interactive), standard out (cout) and standard error (cerr) from the program will instead be saved in a file named "slurm-xx.out", where xx is the job number that was run. The file will be saved in the directory where you launched sbatch.

Try the following

```
$ sbatch hello_script.bash
```

You should see a new file named "slurm-¡some number¿.out" in your directory.

1.10 Investigating the Cluster

One program that is interesting to run across the cluster is the "hostname" command because it will do something different on different nodes (namely, print out the unique hostname of that machine). This can be very useful for quickly diagnosing issues with a cluster – and for just getting a feeling for how the queuing system works.

As an example, try the following:

```
% srun hostname
```

This will run the hostname command on one of the cluster nodes and return the results to your screen. You should see a different node name than the node you are logged into.

To run the same thing on multiple different nodes in the cluster, try the following:

```
% srun -A niac --tasks=8 hostname
```

The `--tasks=8` argument indicates that srun should run the indicated program – in this case, `hostname`, as 8 tasks.

Based on what is printed as a result, are the tasks run on one host or multiple hosts?

Specifying the number of tasks is different from specifying the number of nodes:

```
% srun --nodes=8 hostname
```

The `--nodes=8` argument indicates that `srun` should run the indicated program on 8 nodes.

Based on what is printed as a result, are the tasks run on one host or multiple hosts?

How many tasks are launched, and how many nodes are used, when you combine 'tasks' and 'nodes'?

```
% srun --nodes=8 --tasks=2 hostname
```

The nodes in Hyak have multiple CPU cores per nodes (how many?) and we can specify tasks per node to make best use of the cores (and other resources). Note also that we can still specify cpus per task for multi-threaded tasks. (You might try this with omp_info.exe):

```
% srun --nodes=4 --tasks=2 --cpus-per-task=8 ./omp_info.exe
```

Try running srun with hostname a few times. Experiment with various numbers of nodes, tasks, cpus.

NB: The niac partition will allow a total of 80 cores to be allocated. If you request more total cores (cpus) within the niac account, slurm will reject your request. You can however, submit larger jobs to the ckpt partition.

```
$ srun -p ckpt --nodes=8 --tasks=4 --cpus-per-task=4 ./omp_info.exe
```

This will require waiting until the requested number of resources become available in the ckpt pool, however.

1.11 sbatch vs srun

Above we looked at basic use of srun and sbatch and ran the following:

```
$ sbatch hello_script.bash
```

Look at the contents of the output file from this run. How many instances of the script were run (how many "hello" messages were printed)?

We can implement multiple instances with sbatch:

```
$ sbatch --nodes=4 hello_script.bash
```

How many instances were run?

This is different than what you might have expected. Run the script with "srun"

```
% srun --nodes=4 hello_script.bash
```

How many "hello" messages were printed?

The difference between the sbatch and the 'srun' command is that 'sbatch' just puts your job into the queuing system and requests resources to potentially run a parallel job (the `--nodes` option). But it doesn't run the job in parallel. Running a job in parallel is the performed by 'srun' (or, from a job in the queue, with 'mpirun').

Consider the following script (hello_script_p.bash), slightly modified from hello_script.bash:

```
#!/bin/bash
echo "Hello from Slurm"
srun hostname
echo "Goodbye"
```

Here, we run "hostname" with "srun" – note that we have not specified how many nodes we want. Launch a few trials of the second script

```
% sbatch hello_script_p.bash
% sbatch --nodes=2 hello_script_p.bash
% sbatch --nodes=4 hello_script_p.bash
```

Look at the .out files for each job. How many instances of hostname were run in each case? How many "hello" messages were printed?

We can also use "mpirun" to launch parallel jobs (and is what we will need to run our MPI jobs):

```
#!/bin/bash
echo "Hello from Slurm"
mpirun hostname
echo "Goodbye"
```

(Note, you will need to have the ompi module loaded to be able to invoke mpirun. See below.)

Again, run this script ("hello_script_m.bash") a few different ways and see if the results are what you expect.

```
% sbatch hello_script_m.bash
% sbatch --nodes=2 hello_script_m.bash
% sbatch --nodes=4 hello_script_m.bash
```

NB: If you launch mpirun directly on the front-end node – it will run – but will launch all of your jobs on the front-end node, where everyone else will be working. This is not a way to win friends. Or influence people. You also won't get speed-up since you are using only one task on one node (the front-end node).

Warning: Don't ever run your MPI programs by invoking "mpirun" manually.

2 Warm Up: PS8 Reprise

For this problem set warm-up will consist of revisiting ps8 and executing some of the problems we did on multicore nodes of Hyak.

2.1 Setting up your environment

Different development environments on Hyak are supported via the modules system. There are two modules we need to load for this asignment: gcc/10.2.0 and niac/ompi/4.1.1. To load these modules issue the commands

```
$ module load gcc/10.2.0
$ niac/ompi/4.1.1
```

You will need to load these every time you connect to Hyak in order to use the right version of gcc and to use MPI for this assignment. You can add these statements to the end of your .bashrc file so that they are executed automatically whenever you login.

2.2 hello_omp

From the hello_omp directory, build ompi_info.exe

```
$ make omp_info.exe
```

Now we are ready to run our first job on Hyak.

```
$ srun --time 5:00 -A niac ./omp_info.exe
```

You should get back an output similar to the following

```
OMP_NUM_THREADS         =
hardware_concurrency() = 40
omp_get_max_threads()  = 1
omp_get_num_threads()  = 1
```

Note what this is telling us about the environment into which omp_info.exe was launched. Although there are 40 cores available, te maximum number of Open MP threads that will be available is just 1 – not very much potential parallelism.

Fortunately, srun provides options for enabling more concurrency.

Try the following

```
$ srun --time 5:00 -A niac --cpus-per-task 2 ./omp_info.exe
```

Question 1: How many omp threads are reported as being available? Try increasing the number of cpus-per-task. Do you always get a corresponding number of omp threads? Is there a limit to how many omp threads you can request?

Question 2: What is the reported hardware concurrency and available omp threads if you execute omp_info.exe on the login node?

We are explicitly specifying the maximum time allowed for each job as 5 minutes, using the `--time` option. The default is higher, but since the niac allocation is being shared by the entire class, we want to protect against any jobs accidentally running amok. We will be setting time limits on the command line whenever we use srun and within the batch files when we use sbatch.

2.3 norm

Now let's revisit some of the computational tasks we parallelized in previous assignments. Before we run these programs we want to compile them for the compute nodes they will be running on.

Recall that one of the arguments we have been passing to the compiler for maximum optimization effort has been "-march=native", that means to use as many of the available instructions that might be available, assuming that the executable will be run on the same machine as it is compiled on. To make sure we do this, we need to compile on the cluster nodes as well as execute on them.

Let's build and run norm_parfor.exe

```
$ srun --time 5:00 -A niac make norm_parfor.exe
```

To see that there is a different architectural difference between the compute node and the login node, try

```
$ ./norm_parfor.exe
```

You should get an error about an illegal instruction – generally a sign that the code you are trying to run was built for a more advanced architecture than the one you are trying to run on.

To run norm_parfor.exe, try the following first

```
$ srun --time 5:00 -A niac norm_parfor.exe
```

How much speedup do you get?

We have seen how to get parallel resources above, to launch norm_parfor.exe with 8 cores available, run:

```
$ srun --time 5:00 -A niac --cpus-per-task 8  ./norm_parfor.exe
```

Question 3: What are the max Gflop/s reported when you run norm_parfor.exe with 8 cores? How much speedup is that over 1 core? How does that compare to what you had achieved with your laptop?

3 Problems

Coming soon.