Speeding Code Execution with the Econ Cluster

For most tasks, it is often enough to execute code on your own computer. And for most tasks, using a single core of a processor is also enough. However, there are compute intensive tasks that can greatly benefit from a computer with several cores and access to a relevant amount of memory. For these tasks, your own computer might not suffice. Fortunately, the Economics Department at Duke has its own cluster of computers that Master's and PhD students can use.

1 Requesting Access to the Cluster

To access the cluster, you will need a username and password. The username for the Econ Cluster is the same as your University NetID username, but the password is different. If you do not have a password yet, or do not remember it (if you are a PhD student you were probably assigned a password during your 1st year at Duke), you can request a password by emailing help@econ.duke.edu.

2 Connecting to the Cluster

You can think of the Econ Cluster as a collection of several computers, which are managed by some centralized software. One of these computers is responsible for handling log in, and is known as the front-end node (or the login node). This node is also responsible for receiving and scheduling tasks on the other computers in the cluster.

To log in the Econ Cluster we will use the SSH protocol. This protocol was designed to allow two computers to securely communicate over an insecure network. If you are using a Mac or Linux-based operating system (like Ubuntu), then you already have what is required to use SSH. If you are on Windows, then you will need to install an SSH client, like PuTTY or any of the alternatives (they are all similar and straightforward).

2.1 Mac, Ubuntu or Linux

Open the Terminal program. You should see a window similar to the one depicted in Figure 1.

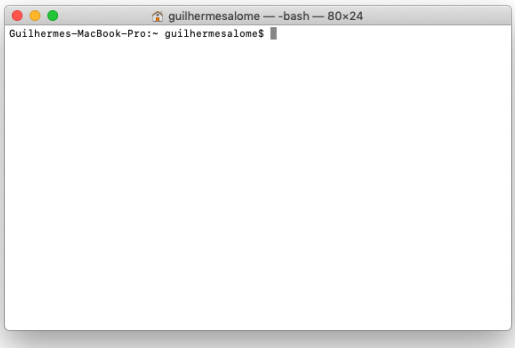

Figure 1: Terminal on a Mac.

You can type commands after the dollar sign, and the terminal will interpret the commands and execute them (REPL). We will discuss how to use the terminal in the next section.

To log in the Econ Cluster, we will the the `ssh` command. The `ssh` command uses the syntax:

```
ssh username@hostname
```

Where `username` is the username you will use to log in, and `hostname` is the address of the computer host that you will connect to. If you are currently inside the Duke network, then the `hostname` is `login.econ.duke.edu`. In my case, I would execute:

```
ssh gfs8@login.econ.duke.edu
```

Then, I am asked for my password to finish logging in (see Figure 2).

Figure 2: SSH Asking for Password.

After typing your password, you should be greeted with a welcome message and are now connected to the cluster (see Figure 3). You are connected to a bash terminal, which can take commands and will execute them. We will discuss these commands in the next section.

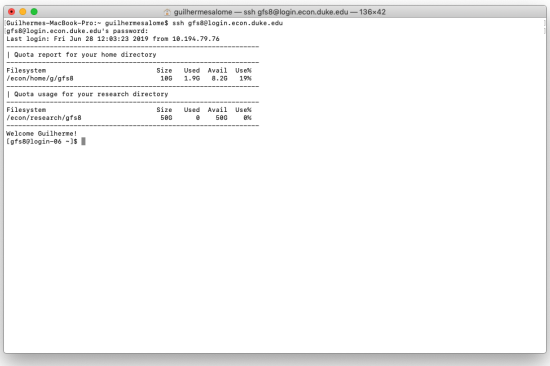

Figure 3: SSH Welcome Message.

If you are outside the Duke network, then you first the need to `ssh` into the Duke network, and then `ssh` again into the Econ Cluster. To `ssh` into the Duke network, you should use `login.oit.duke.edu` as the `hostname`, but now the username and password are the same you use for logging into Duke websites (like Dukehub). After that, you can execute `ssh` again to log in the Econ Cluster (see Figure 4).

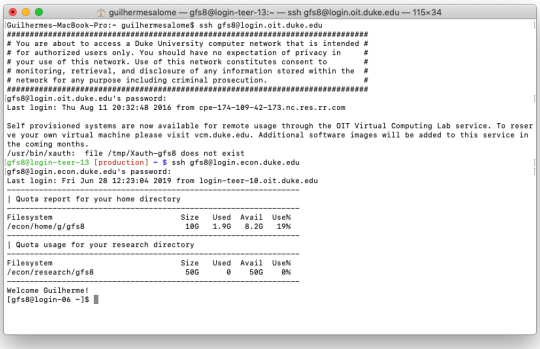

Figure 4: SSH From Outside Duke Network.

2.2 Windows

To connect to the cluster when working in the Windows operating system, you will need to install an SSH client. I recommend installing PuTTY due to its simplicity and convenience (it is also free). After installing the software, you will use `ssh` but with a graphical user interface.

Open PuTTY and type `login.econ.duke.edu` on the `Host Name` window (see Figure 5). You can now click `Open` to connect to the cluster. You will be prompted for your username and your password.

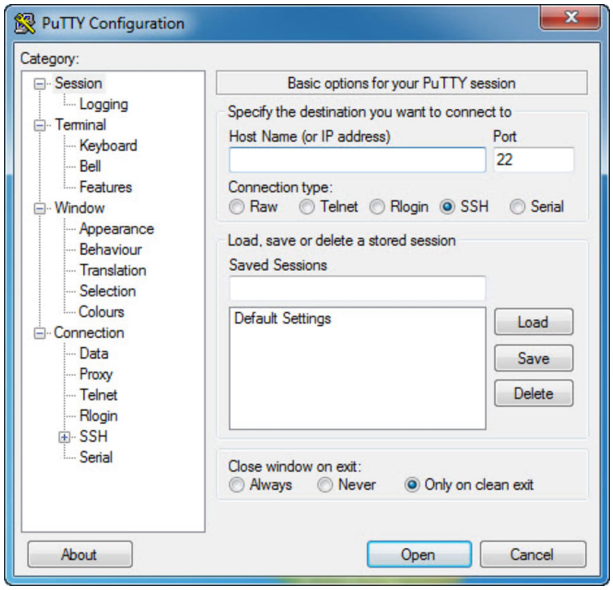

Figure 5: PuTTY SSH Client.

After typing your password, you should be greeted with a welcome message and are now connected to the cluster. You are connected to a bash terminal, which can take commands and will execute them. We will discuss these commands in the next section.

3 Using a Bash Terminal

Now that we are connected to the cluster, we can discuss how to use the terminal. The terminal window you see is a bash shell. A shell is just a user interface to the underlying operating system, and bash refers to the type of interface.

We can interact with the shell either by typing commands and executing them line by line, or by creating script files. We will cover some basic commands of the bash shell.

3.1 Working Directory

The command `pwd` prints the current working directory:

```
pwd
```

You can change the working directory with the `cd` command:

```
# syntax: cd folder
cd /econ
pwd
cd /econ/home
pwd
cd /econ/home/g/gfs8
pwd
cd ..                              # .. refers to the parent
   folder
pwd
```

While `..` represents the parent folder, there is a shortcut for your home folder as well:

```
cd ~
# ~ represents the home folder
# alternatively
cd $HOME
```

3.2 Creating Folders

You have permission to change things around only in your home folder. In my case, my home folder is the folder `/econ/home/g/gfs8`. We can create a new folder with the command `mkdir`:

```
# syntax: mkdir folder_name
mkdir Matlab
mkdir Test
# create multiple folders
mkdir A B C
```

3.3 Listing Files and Folders

We can list all files and folders inside a folder with the command `ls`.

```
# syntax: ls
ls
# display one file or folder per line
ls -1
```

You can get a full list of the options a command accepts by reading the manual page of the command. You can access the manual page of a command using `man`:

```
# syntax: man command_name
man ls
```

3.4 Deleting Files and Folders

We can delete an empty folder with the command `rmdir`.

```
# syntax: rmdir folder_name
rmdir Test
# remove multiple folders
rmdir A B C
```

To remove a non-empty folder we need to use the more versatile command `rm` with the option `-r`:

```
# create a non-empty folder
mkdir Test Test/A Test/B
# check it is non-empty
ls -1 Test
# equivalent to
```

```
cd Test
ls -1
cd ..
# try to remove with rmdir
rmdir Test                          # error
# use rm -r
rm -r Test                          # works
```

The command `rm` can also be used to remove files.

3.5 Creating Files

We can create empty files with the `touch` command:

```
# syntax: touch file_name
touch test.txt
# create multiple files
touch a.csv b.jpg
# remove files
rm test.txt
# remove multiple files
rm a.csv b.jpg
```

We can also create and edit files. To edit a file we need an editor. Some of the editors available inside the bash terminal are: nano, vim and emacs. You can create or edit a file with these editors by typing the name of the editor followed by the name of the file. The editor `nano` is the most straightforward to use, however, the editors `vim` and `emacs` are extremely powerful and might be worth to learn if you often use your computer to type text into files.

```
# create a new file with nano
nano data.csv
# nano will now open with an empty file
# type in:
# 1,21,0
# 2,28,25000
# 3,35,70000
# then use Ctrl-O to save the file, and then Ctrl-X to exit
   nano
# create another file
nano description.txt
# type in:
# id,age,income
```

3.6 Inspecting Files

You can quickly inspect the contents of a file with the `cat` command.

```
# check name of files
ls -1
```

```
# see contents of data.csv
cat data.csv
# see contents of description.txt and data.csv
cat description.txt data.csv
```

If the file is too big and you do not want to display its entirety on the screen, you can use the `head` and `tail` commands. The `head` command displays the first few lines of a file, while the `tail` command displays the last lines of a file.

```
# see first lines of the file
head data.csv
# see only first two lines of the file
head -n 2 data.csv
# see only the first line of the file
head -n 1 data.csv
# see last lines of the file
tail data.csv
# see only very last line of the file
tail -n 1 data.csv
# display first line of multiple files
head -n 1 description.txt data.csv
```

3.7 Copying and Moving Files

To copy files use the `cp` command.

```
# syntax: cp source_file target_file
# create a copy of data.csv
cp data.csv data_copy.csv
# create a copy of the data inside a data folder
mkdir Data
cp data.csv Data/
ls Data
```

Files can be moved with the `mv` command.

```
# remove the copy of data.csv from the Data folder
rm Data/data.csv
ls Data
# move the original data file to the Data folder
mv data.csv Data/
# check the file changed folder
ls
ls Data
```

4 Bash scripts

A bash script is just like a Matlab script: it is a text file containing commands that should be executed line by line. We can create a bash script by creating a file with the

extension `.sh`. Let's use the command `echo` to print strings to the terminal window and save them in a script file.

```
# syntax: echo string
echo "Hello there!"
echo "How are you?"
```

Save it in a file (`nano greetings.sh`):

```
# greetings.sh
echo "Hello there!"
echo "This is a bash terminal."
echo "Welcome."
head data.csv
```

We can execute this file with the command `source`. The `source` command takes a script file and executes it line by line in the current shell.

```
# syntax: source script_name
source greetings.sh
```

And you should see the three messages displayed in the shell.

We can use a shell script to execute several programs, including programs in other languages. To do so, the shell must be able to find other programs. For example, when we typed `nano` before, the shell searched for the program `nano` and then executed it with the parameters we passed it. We can check whether the terminal can find a program with the command `which`.

```
# syntax: which program_name
# if the program can be found, then the shell
# displays the path to the program binaries
which nano
which emacs
# if the program cannot be found, then nothing is displayed
which foo123
```

If the program cannot be found, then it could either not be available in the machine, or it could be outside of the path of the terminal. The path is a list of folders where the terminal searches for programs. If the program cannot be found in those folders, then the terminal does not return anything. However, if you know in which folder the program lives, then you can specify the full path to it. Alternatively, you can add its folder to the search path (this is left for another time, as it introduces some complications). For example, in my machine `matlab` is not in the search path, but it can be found by specifying the full path:

```
# on local machine
which matlab
# returns nothing
# but we can specify the full path
which /Applications/MATLAB_R2019a.app/bin/matlab
```

On the cluster, however, `matlab` should be in the search path:

```
ssh gfs8@login.econ.duke.edu
which matlab
# returns /usr/local/bin/matlab
```

Let's create a Matlab script that will perform some computation and save the results.

```
% matlab_computation.m
% Do some computation
x = rand(200, 1);
y = rand(200, 1);
res = x.*y.^2 + 1;
% Save results
save('results.mat', 'res');
```

We can now write a bash script that will interact with the operating system and then execute the Matlab script. To execute the Matlab script we would call `matlab test.m`. However, `matlab` also takes some options that can speed up its execution. We can pass a script to be executed with `matlab` by using the option `-batch` and specifying the script name, without the extension `.m`. The complete set of options `matlab` takes is described in this reference page.

```
# matlab_from_bash.sh
echo "Executes the Matlab script: matlab_computation.m"
# execute Matlab script
matlab -batch "matlab_computation"
```

We can now execute this script with `source matlab_from_bash.sh`, and the script will be executed. Notice that at the end of the script, Matlab saves the results in the file `results.mat`, which later we can import into our computer to analyze.

5 Slurm: Scheduling Tasks

Bash scripts are important because they are how we can interact with the Econ Cluster. Remember, the Cluster is simply a collection of computers being managed by some software. The software that manages the Econ Cluster is the Slurm Workload Manager.

In the previous section, we logged in the log-in node of the cluster. At that node, we can interact with the cluster via the terminal, and even execute some Matlab code via a `bash` script. However, Slurm will not allow us to execute a lot of code, or code that takes too long to run. Instead, Slurm allows us to submit tasks for it to run. That is, we can give Slurm a `bash` script, and it will allocate the script to some computer in the cluster, run it, and save the results in your home folder. In doing so, Slurm allows all users of the Cluster access to powerful computers, but there may be a queue.

5.1 Shebang

The script `matlab_from_bash.sh` is almost ready to be submitted for execution with Slurm. It is only missing a shebang line. The shebang is the very first line of a text file used as a script, which specifies the interpreter that should be used when executing the

file. While we could execute our script with `source`, by adding a shebang to the file, we can execute it as an executable file. Create the following test script:

```
#!/bin/bash
# test_shebang.sh
echo "Hello!"
```

We can still execute it with `source`:

```
source test_shebang.sh
```

But now, we can execute it as an executable file:

```
# make the file executable
chmod +x test_shebang.sh
# execute it
./test_shebang.sh
```

The first line of the script tells the terminal that it should use `bash` to interpret its contents. Slurm needs this shebang to correctly submit the script.

5.2 Submitting to the Cluster

Modify the `matlab_from_bash.sh` so that its first line is `#!/bin/bash`. Delete the `results.mat` file from the folder.

We can submit the `matlab_from_bash.sh` script to Slurm with the command `sbatch`.

```
# submit script to Slurm
sbatch matlab_from_bash.sh
```

If there is no immediate issue with the script, Slurm will schedule the script for execution in one of the computers (nodes) of the cluster. The scheduled script is called a job. Slurm will print the job number on the screen.

5.3 The Queue

After submitting a script for execution, you should have its job number. You can use this number to check what is the state of the job. We can use the command `squeue` to see all jobs currently scheduled and being executed. See Figure 6 for the output of `squeue`

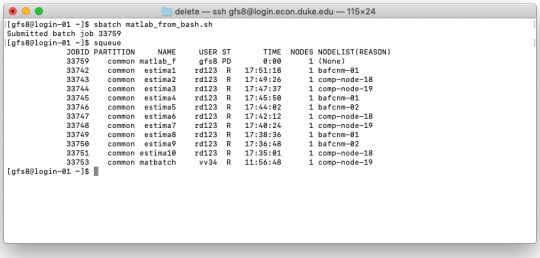

Figure 6: Slurm queue after submitting a job.

The job we just submitted has the number `33759`. You can see in the second line after the command `squeue` that the job has been scheduled. The `NAME` shows the name of the script submitted, `USER` shows the user who submitted the job (net id), `ST` shows the state of the job (PD stands for pending, while R stands for running). The `TIME` shows for how long the job has been running. In the case of our job it has not started yet, so `TIME` is `0:00`. Notice that some other users have jobs running for more than 17 hours! The `NODELIST` describes the reason why the job is in the queue. In the case of the job we submitted, `(None)` means the job will be executed next. Usually, if there are not enough resources available to execute your script, you will see a `(Resources)`, which indicates Slurm is waiting for other scripts to finish before executing yours. When the script is being executed, `NODELIST` will show the node (computer) where the script is running.

If we have multiple jobs running, then we can check the status of the jobs submitted only by us (not by everyone). We can do so with the option `-u username`, which displays `squeue` but only for the specified username.

```
squeue -u gfs8
```

See Figure 7 for an example.

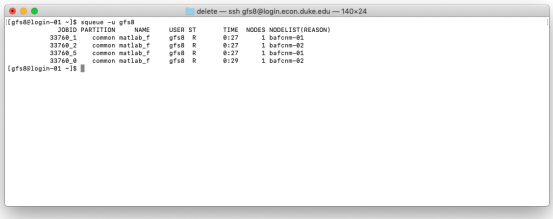

Figure 7: View of Slurm queue for a specified user.

In this case I submitted several jobs. The queue shows they are all running, some are running on different nodes, and they have been running for a few seconds already. If you run `squeue -u gfs8` again, then you might see fewer jobs. This is because when a job is completed, it does not show up in the queue anymore.

5.4 Details on a Job

We can use the `sacct` to get information on running and completed jobs we have submitted to Slurm. Running `sacct` will display details for all jobs submitted by you on a certain period of time. If you have the job number, then you can use pass it with the option `-j` to get details about that specific job (see Figure 8):

```
sacct -j 33759
```

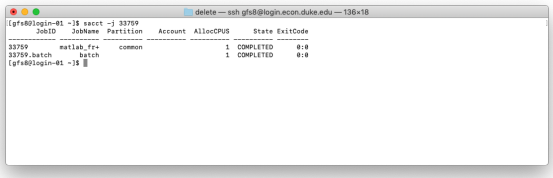

Figure 8: View details of a submitted job.

If you want even more information about a specific job, then you can use the command `sacct_ec`:

```
sacct_ec -j 33759
```

Figure 9 displays the output of `sacct_ec`. Notice it shows the node where the script was executed, and even the start and end time of the script.

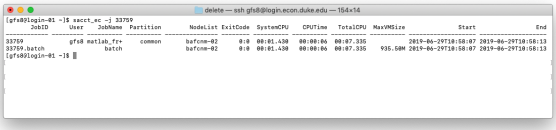

Figure 9: View even more details of a submitted job.

These commands are useful when you submit jobs that might take a long time to execute, or when debugging scripts that are failing to execute.

5.5 Canceling a Job

You can cancel a job using the `scancel` command. For example, if the job number is 33759, then `scancel 33759` will cancel the job. You can only cancel jobs you started yourself. If your script is taking longer than expected to complete, then you might have some bug in your code that is causing it to hang. In this case, canceling the job might be required.

5.6 Outputs of the Job

When a job is submitted and starts running, a file with the name `slurm-XXXXX.out` is created. The `XXXXX` represents the number of the corresponding job. This file contains whatever your script prints to the screen. For example, when we executed the `matlab_from_bash.sh` script, it printed a couple of lines on the terminal. If we submit this script to Slurm, then the lines that would be printed on the terminal are saved on the `slurm-XXXXX.out` file.

The `.out` file can be used as a log of what is happening in your script. You can use it to debug your program, since debugging in the cluster is not as straightforward as debugging in your local machine (you cannot stop the execution of the code and inspect variables, for example).

The Matlab script we submitted also created a file containing the results. This file is also saved in our home folder. When we submit a job with Slurm, our home folder is directly accessible by the job, and behaves as a local drive.

6 Slurm: Partitions and Nodes

Slurm organizes the cluster in `partitions`, where each `partition` is a set of `compute nodes` (computers used to run code). We can get an overview of the partitions available in the Econ Cluster with the `sinfo` command. Figure 10 displays the output of `sinfo`.

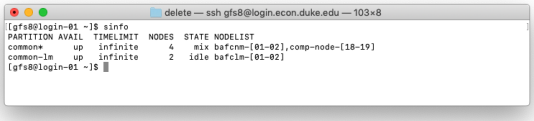

Figure 10: Overview of cluster partitions.

Notice there are two partitions, `common*` and `common-lm`. Different partitions might have different purposes. For example, there could be a partition for debugging code, and another for actually submitting tasks. In this case, the two partitions are for computing. The default partition is marked with an asterisk. That is, the partition `common` is where jobs are submitted to by default.

There are four compute nodes in the default partition. Some of them are in use, other are idle, leading to the `mix` state. In the `common-lm` partition there are two nodes, which are idle. The `NODELIST` column gives the names of the nodes. The nodes in the `common` partition are `bafcnm-01`, `bafcnm-02`, `comp-node-18` and `comp-node-19`. The nodes in the `common-lm` partition are `bafclnm-01` and `bafcnm-02`.

Notice that the column `TIMELIMIT` says `infinite`. This is the time limit to which jobs are subject. In this case, there is no time limit.

We can display information organized by node instead of partition with the option `-N`. We can also use the option `-l` to display extra information.

```
1 sinfo -N -l
```

Figure 11 displays the output of `sinfo -N -l`.

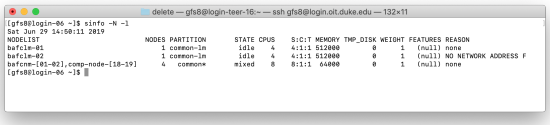

Figure 11: Overview of cluster nodes.

Notice that some of the nodes have 512 GB of memory, and the nodes in the default partition have 64 GB of memory. The nodes in the common partition have more CPUS than the nodes in the `common-lm` partition. The `S:C:T` column displays the number of sockets in each motherboard[1], the number of cores in each processor, and the number of threads in each core. If you multiple these numbers you get the number in the `CPUS` column. In terms of parallel computing, the number of `CPUS` is more or less equivalent to the number of parallel process that can be used with `parfor`.

It is important to know that when you submit a job to Slurm, it does not mean the job will use an entire compute node to execute. That is, a single compute node can execute multiple jobs at the same time. For example, a single job might use just one of the CPUS of a compute node, so that the other CPUS can be allocated to other jobs. It is also possible to have a job use multiple CPUS, including CPUS from more than one

[1]The socket is the physical space in a computer's motherboard where the physical CPU is placed. Most common motherboards only have one socket. However, motherboards for cluster computers usually have more than one socket. This is the case for the computers in the Econ Cluster. Some of them have 4 sockets, while others have 8 sockets. That is, a single computer can have 4 or 8 different physical CPUs.

compute node. On the next section we will see how to request CPUS and memory when submitting jobs to Slurm.

7 Slurm: Requirements and Directives

When it comes to speeding up code execution, we saw that Matlab allows us to use more than one processor core to speed up loops with `parfor`. With the Econ Cluster, we can submit a job and request a certain number of CPUS to be available. Additionally, we can also request a certain amount of memory, so that we do not run into memory issues when using `parfor`. We will see how to specify these requirements using directives.

7.1 Directives

When we submit a script with `sbatch`, Slurm checks the script for directives. Directives are lines that start with `#SBATCH`. We can add directives that tell Slurm what resources are required to run the script. Slurm then processes these directives and waits until the resources are available to start executing your script. This allows us to specify, for example, how many CPUS and memory we need to run the script.

7.2 Requesting Memory and CPUs

Let's modify the `matlab_from_bash.sh` script to add some directives.

```
1 #!/bin/bash
2 # matlab_from_bash.sh
3 # require 12 GB of memory for this job
4 #SBATCH --mem=12G
5 # require 4 CPUS for this job
6 #SBATCH --cpus-per-task=4
7 echo "Executes the Matlab script: matlab_computation.m"
8 # execute Matlab script
9 matlab -batch "matlab_computation"
```

The first directive requires a large amount of memory (12 GB). Remember that the more workers you have, the more memory you need, since Matlab needs to distribute all information in your workspace to all of the workers. If you run into problems executing a parallel code in the cluster, it might be because you requested too little memory. If this is the case, either request more memory or reduce the number of workers you require. By using the `--mem` directive, we know that when our code is executed, at least 12 GB of memory will be available.

There is a caveat with the memory directive in Slurm. If at some point our code requires more than 12 GB of memory, Slurm will kill the execution of the job. This happens even if more memory is available at the node. That is, if the node has 64 GB of free memory, but your code uses more than the 12 GB that were requested, then Slurm will kill the job. This means that the `--mem` directive is binding. You must make sure that your program does not use more memory than what was requested.

The second directive requires a node that has at least four available CPUS. Contrary to the memory directive, the `--cpus-per-task` directive will allow your code to use more CPUS if available. That is, if Slurm starts executing your job and all CPUS of the node

are not in use, then your code may receive more CPUS than were requested. This allows us to use `parfor` to speed the execution of the code. If more resources are available, then we might even get more CPUS than requested.

Let's modify the `matlab_computation.m` script to run a `parfor` loop.

```
% matlab_computation.m
% Check how many cores were allocated to the job
feature('numcores')
% Start pool of workers
ppool = parpool();
disp(sprintf('Total Workers: %d', ppool.NumWorkers));
% Run parfor with increasingly more workers
for w = 1:ppool.NumWorkers
    disp(sprintf('Workers: %d', w));
    tic;
    parfor (i=1:10, w)
        pause(1)
    end
    toc;
end
```

The first line tells Matlab to check how many cores are available. It prints out information on how many cores are available in total, and how many are actually being used by Matlab. The number of cores will determine how many workers can be used with `parfor`.

The second line initializes the pool of workers. By calling `parpool()` we create the pool of workers with the default number of workers. If you are worried about memory limitations (remember: more workers require more memory), then you can call `parpool` with the number of workers you require. For example, `parpool(4)` would initialize a pool of workers with only four workers, even if more are available.

The third line of code displays how many workers are available in the pool. The foor-loop iterates over the number of available workers. It starts with a single worker, and times the execution of a parfor-loop. Observe that if there is only one worker, then the parfor-loop is a regular loop. It will iterate over the variable `i`, pausing for one second at every iteration. When `w` is one, this parfor-loop should take about 10 seconds to complete.

On the next iteration in the outermost loop, the variable `w` becomes two. Now, we run the parfor-loop with two workers. This means that the command `pause(1)` will be assigned twice, once to each worker, so the loop will execute faster. Indeed, it should take about 5 seconds to execute, since we two workers paused at each time.

Since the number of workers in the cluster can be high, the time to execute the parfor-loop code above will get progressively smaller. After submitting the code with `sbatch`, we can inspect the output file. Figure 12 displays the output.

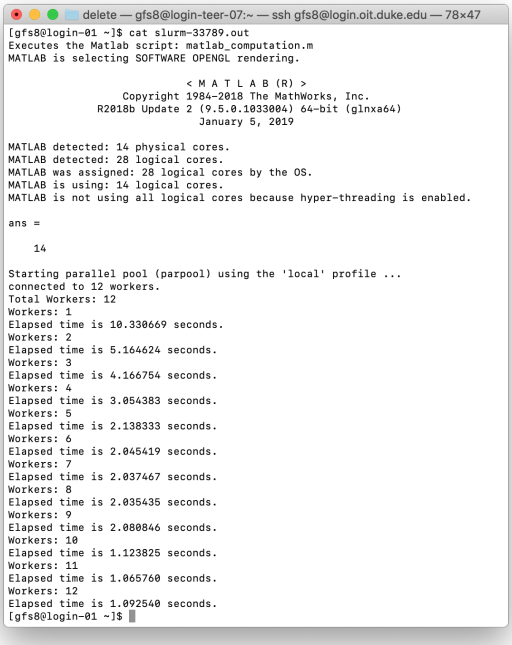

Figure 12: Output of Matlab script using `parfor` on the cluster.

In this case, even though we only requested four CPUS, many more were available. Indeed, we could create a pool of workers with twelve workers! Notice that there were actually fourteen cores available to Matlab, but the default number of workers from the `parpool` command is twelve. We could have created a pool with the fourteen workers by calling `parpool(14)`. Alternatively, we could have called `parpool(feature('numcores'))` to use all available cores.

7.3 Multiple Jobs and Transferring Files

We have discussed how to submit a single job that uses many cores to speed up computation. An alternative paradigm is submitting several different jobs that use a single core, and then collecting all of their results.

For example, consider you have a statistic that you are interested in studying via bootstrap, but that takes a long time to be computed (potentially because of a large sample, or because the statistic is complicated). In this case, we could launch several different jobs, where each computes a bootstrap sample and the statistic just once. We can then accumulate the results to study the distribution of the statistic.

Let's create some fake data to bootstrap.

```
% generate_fake_data.m
mu = 20.5;
sigma = 5.8;
sample_size = 1000000;
sample = normrnd(mu, sigma, sample_size, 1);
save('fake_data.mat', 'sample');
```

We will use this data to compute the confidence interval for the mean via bootstrap. We need to get the data from our computer into the cluster. To do so, we will use the

command `scp`, which is similar to the `ssh` command, but is used to copy files securely. If you are transferring a lot of files, or want a graphical user interface, then you could use the program Cyberduck, for example.

The syntax for `scp` is the following:

```
scp file_to_copy username@host:~
```

The command will take the file named `file_to_copy` and copy it to the folder `~` in the host. Remember that `~` represents your home folder.

At the time of writing, I am outside of the Duke network, so there are two steps to copy the data from my laptop to the Econ Cluster. First, I need to transfer the data from my laptop to the Duke login node. Then, I `ssh` into the Duke login node and transfer the data from there to the Econ Cluster.

```
# Locate the file on my laptop
ls -1
# fake_data.mat is in the current working directory
scp fake_data.mat gfs8@login.oit.duke.edu:~
# will ask for your password and then start transferring the file
# Now, ssh into the login node of Duke
ssh gfs8@login.oit.duke.edu
# check that fake_data.mat is in the current working directory
ls -1
# transfer to the Econ Cluster
scp fake_data.mat gfs8@login.econ.duke.edu:~
# ssh into the Econ Cluster
ssh gfs8@login.econ.duke.edu
# verify that the file was transferred
ls -1
```

Now, let's create the bootstrapping script.

```
% bootstrap_mean_fake_data.m
% Load data
load('fake_data.mat', 'sample');
% Bootstrap a new sample
ssize = length(sample);
new_sample = sample(unidrnd(ssize, ssize, 1));
% Compute statistic
statistic = mean(new_sample);
% Save result to csv
% Question: but what should be the name of the file?
```

The script loads the fake data, generates a bootstrap sample, computes the statistic and saves it in a `.csv` file. However, we have a problem. What should be the name of the `.csv` file? Since we are launching multiple jobs at the same time, how do we set the name of the file programmatically? Before answering these questions, let's create the `bash` script that will launch several jobs, where each job will execute the Matlab script above once.

```
#!/bin/bash
# multiple_jobs.sh
#SBATCH --cpus-per-task=1
#SBATCH --mem=1GB
#SBATCH --array=1-10
matlab -batch "bootstrap_mean_fake_data"
```

We can use the `--array` directive to submit multiple jobs with Slurm. The directive `--array=1-10` will launch ten different jobs, numbered from 1 to 10. Each job will execute our Matlab script. Since each job has a different number, each job will create a separate `.out` output file. However, this output file is just what is displayed in the terminal, and is mainly used for debugging. What we want to recover is the file containing the statistic we computed.

When we submit a job with `--array`, a variable is created with the number of the number of the job. We can pass this variable to the Matlab script and use it to save the statistic value. The variable name that holds the job number is `SLURM_ARRAY_TASK_ID`. We can access its value in the `bash` script using the command `$SLURM_ARRAY_TASK_ID`. We can then pass this value to the matlab script:

```
#!/bin/bash
# multiple_jobs.sh
#SBATCH --cpus-per-task=1
#SBATCH --mem=1GB
#SBATCH --array=1-10
matlab -batch "job_number=$SLURM_ARRAY_TASK_ID;
   bootstrap_mean_fake_data"
```

Now, inside the `boostrap_mean_fake_data.m` script we can access the variable `job_number`:

```
% bootstrap_mean_fake_data.m
% Set seed
rng(job_number);
% Load data
load('fake_data.mat', 'sample');
% Bootstrap a new sample
ssize = length(sample);
new_sample = sample(unidrnd(ssize, ssize, 1));
% Compute statistic
statistic = mean(new_sample);
% Save result to csv
filename = strcat(num2str(job_number), '.csv');
csvwrite(filename, statistic);
```

The variable `job_number` is initialized by the `bash` script. We use it to create the `.csv` file, so that each job, which has its own unique number, will create a unique `.csv` file. This also has the advantage that if some job fails (a bug, or takes too much resource, or bad weather), then we can quickly find it. We also used the `job_number` to set the seed for random number generation. This way when we generate the bootstrap samples they

are not all going to be the same, and the output will be repeatable if executed with the same `job_number`.

We can submit the script to Slurm and verify that indeed 10 jobs are created:

```
# Submit script for execution
sbatch multiple_jobs.sh
# A job id is assigned.
# Look at all jobs that were created:
squeue -u gfs8
# Notice that JOBID is xxxxxx_y, where xxxxxx is the job id, and
# y is the job number specified in the --array directive
# Observe that many slurm-xxxxxx_y.out files were created:
ls -1 slurm-*
# The * above expands slurm- to all existing filenames that
# begin with slurm-
# Jobs should be done now:
squeue -u gfs8
# Notice that the .csv files were created
ls -1 *.csv
# We can sort the names numerically with the option -v
ls -1 -v *.csv
# We can print them on the screen with cat
cat *.csv
```

The directive `--array` takes a range of numbers, like `1-10`, and launches jobs using those numbers to create the job ids. However, the range of numbers can be discontinuous. For example, the directive `--array=1-3,5,7-10` would launch jobs with ids 1, 2, 3, 5, 7, 8, 9 and 10, skipping the ids 4 and 6. It is also possible to use `--array=7` to launch a single job with that id number. The single value or discontinuous `--array` directives are useful for resubmitting a specific job that failed.

8 Assignment

Problem 1 *Modify the scripts* ***`multiple_jobs.sh`*** *and* ***`bootstrap_mean_fake_data.m`*** *so that a new folder is created to store the resulting* ***`.csv`*** *files. If the folder already exists, the files inside it should be deleted before the script runs.*

Problem 2 *(Optional) Create a* ***`bash`*** *script that takes all of the* ***`.csv`*** *files outputted from* ***`multiple_jobs.sh`*** *and creates a single* ***`.csv`*** *file with all of the results.*

Problem 3 *Create a Matlab script that takes all of the* ***`.csv`*** *files outputted from* ***`multiple_jobs.sh`*** *and creates a single* ***`.csv`*** *file with all of the results.*

Problem 4 *Compare the advantages and disadvantages of the following takes on parallel computing with the cluster:*

- *Launching a single job that uses multiple cores;*
- *Launching several jobs that use a single core.*

Problem 5 *Assume you mistakenly launched several jobs with the* `--array=1-500000` *directive. Instead of typing* `scancel job_id` *for each job, how would you use* `scancel` *to cancel all of the jobs you submitted at once?*

Problem 6 *Consider a deterministic growth model, where an agent decides between consumption (c_t) and investment in capital (k_t), while maximizing his utility. We can write this problem as:*

$$\max \sum_{t=0}^{\infty} \beta^t U(c_t)$$
$$\text{subject to } \begin{cases} k_{t+1} = k_t^{\alpha} - c_t + (1-\delta)k_t, \forall t >= 0 \\ k_0 > 0 \end{cases}$$

Write the problem as a Bellman equation. Let $U(c;\sigma) = \frac{c^{1-\sigma}-1}{1-\sigma}$. Obtain the Euler equation for this problem in terms of the consumption c. Solve the problem by Value Function Iteration. Consider $\sigma = 2$, $\beta = 0.95$, $\delta = 0.1$ and $\alpha = 0.33$. Use the steady state value of k to create a grid for the possible values of k, say 100 points between $0.25k^*$ and $1.75k^*$. Start with a guess for V over the grid, for example $V(k) = 0$ for all k in the grid. Use the Matlab minimization function to solve for k. You may want to add the constraint that c should always be positive.

Use `parfor` *and the Econ Cluster to speed up the solution of this problem. Compare the speed gain with solving the problem using your local computer.*