# Economics Department Cluster, and Your Own Cluster

Now that we know how to run code in multiple cores, we can take advantage of the availability of powerful computers. The Economics Department at Duke has its own cluster of computers that Master's and PhD students can use. We will discuss how to set up Python, create scripts and submit them for execution in the cluster. After we learn how to use the Economics cluster, we will learn how to use DigitalOcean to gain access to very powerful computers for a limited time.

# 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. A good way to get one is by installing Git. After installing Git, you should have a program named Git Bash, which is similar to the Terminal program used on Mac or Ubuntu.

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

• • •	😭 guilhermesalome — -bash — 80×24	
Guilhermes-MacB	ook-Pro:~ guilhermesalome\$	

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

1

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.

Guilhermes-MacBook-Pro:~ gui	🛞 guil	ilherm	esalome	- ssh
afs80login.econ.duke.edu's p Last login: Fri Jun 28 12:03	hermesalome\$ ssh ssword: 23 2019 from 10.:	gfs8	0login. 9.76	econ.du
Quota report for your home	directory			
Filesystem /econ/home/g/gfs8	Size U 105 1	Used 1.9G	Avai1 8.20	Use% 19%
Quota usage for your resea	ch directory			
Filesystem /econ/research/gfs8	Size U 500	Used Ø	Avail 500	Use% 8%
an a				

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).

# You are about to access a Duke # for authorized users only. You # your use of this network. Use # monitoring, retrieval, and dis metwork for any purpose inclus ####################################	<ul> <li>University</li> <li>should have of this network</li> <li>closure of this ding criminal</li> <li>ding criminal</li> <li>ding criminal</li> <li>ding criminal</li> <li>ding criminal</li> <li>ding criminal</li> <li>ding criminal</li> <li>available f</li> </ul>	comput o no ex work co iny inf l prose ######## be-174-	er netw pectati nstitut ormatio cution. NBNBNBN 189-42-	k that is intended # of privacy in # consent to # stored within the # # # # # # # # # # # # # # # # # # #	
# for authorized users only. You # your use of this network. Use # monitoring, retrieval, and dis network for any purpose inclus ####################################	should have of this network closure of a ding criminal communication ord: 2016 from cp available f	o no ex work co iny inf l prose www.ww ce-174-	pectati nstitut ormatio cution. NUNUNUNU 189-42-	of privacy in # consent to # stored within the # ###################################	
# your use of this network. Use # monitoring, retrieval, and dis # network for any purpose includ memeratematematematematematematematematematem	of this netw closure of a ding criminal communication ord: 2016 from cp available f	ork co ny inf prose nanana ne-174-	ormatitut ormatio cution. NUNUNUNU 189-42-	consent to # stored within the # # NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN	
# monitoring, retrieval, and dis W network for any purpose includ ####################################	sclosure of a ding criminal second ord: 2016 from cp s available f	iny inf prose IBNBNBB De-174-	ormation. cution. NBNBNBNB 189-42-	Stored within the # # NewNewSeawseawseawseaw 3.nc.res.rr.com	
gfs@login.oit.duke.edu's passwo Last login: Thu Aug 11 20:32:48 Self provisioned systems are now ve your own virtual machine plea	rd: 2016 from cp	e-174-	-189-42-	Nannenananananananan 3.nc.res.rr.com	
gfs8@login.oit.duke.edu's passwc Last login: Thu Aug 11 20:32:48 Self provisioned systems are now ve your own virtual machine plea	ord: 2016 from cp / available f	oe-174-	189-42-	3.nc.res.rr.com	
Last login: Thu Aug 11 20:32:48 Self provisioned systems are now ve your own virtual machine plea	2016 from cp available f	oe-174-	189-42-	3.nc.res.rr.com	
Self provisioned systems are now ve your own virtual machine plea	r available f				
gfs8@login.econ.duke.edu's passm Last login: Fri Jun 28 12:23:84   Quota report for your home dir	2019 from lo	∋gin-te	er-10.0	. duke . edu	
Filesystem /econ/home/g/gfs8	Size 100	Used 1.96	Avail 8.20	se% 19%	
Quota usage for your research	directory				
Quota usage for your research	directory Size	Used	Avail	 seX	

Figure 4: SSH From Outside Duke Network.

# 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

1

You can change the working directory with the cd command:

```
# syntax: cd folder
1
2
  cd /econ
3
  pwd
  cd /econ/home
4
5
  pwd
6
  cd /econ/home/g/gfs8
7
  pwd
8
  cd ..
                                      # .. refers to the parent
     folder
9
  pwd
```

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

```
1 cd ~
2 # ~ represents the home folder
3 # alternatively
4 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:

```
1 # syntax: mkdir folder_name
2 mkdir Python
3 mkdir Test
4 # create multiple folders
5 mkdir A B C
```

# 3.3 Listing Files and Folders

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

```
1 # syntax: ls
2 ls
3 # display one file or folder per line
4 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:

```
1 # syntax: man command_name
2 man ls
```

# 3.4 Deleting Files and Folders

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

```
1 # syntax: rmdir folder_name
2 rmdir Test
3 # remove multiple folders
4 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
1
2
   mkdir Test Test/A Test/B
   # check it is non-empty
3
4
   ls -1 Test
   # equivalent to
5
6
   cd Test
7
   ls -1
   cd ..
8
9
   # try to remove with rmdir
10
   rmdir Test
                                      # error
11
   # use rm -r
12
   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
1
2
  touch test.txt
  # create multiple files
3
  touch a.csv b.jpg
4
  # remove files
5
6
  rm test.txt
7
  # remove multiple files
8
  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.

```
1 # create a new file with nano
2 nano data.csv
3 # nano will now open with an empty file
4 # type in:
5 # 1,21,0
6 # 2,28,25000
7 # 3,35,70000
```

```
8 # then use Ctrl-O to save the file, and then Ctrl-X to exit
nano
9 # create another file
10 nano description.txt
11 # type in:
12 # id,age,income
```

#### 3.6 Inspecting Files

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

```
1 # check name of files
2 ls -1
3 # see contents of data.csv
4 cat data.csv
5 # see contents of description.txt and data.csv
6 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
1
2
  head data.csv
3
  # see only first two lines of the file
  head -n 2 data.csv
4
  # see only the first line of the file
5
6
  head -n 1 data.csv
7
  # see last lines of the file
  tail data.csv
8
9
  # see only very last line of the file
10
  tail -n 1 data.csv
  # display first line of multiple files
11
  head -n 1 description.txt data.csv
12
```

### 3.7 Copying and Moving Files

To copy files use the cp command.

```
1 # syntax: cp source_file target_file
2 # create a copy of data.csv
3 cp data.csv data_copy.csv
4 # create a copy of the data inside a data folder
5 mkdir Data
6 cp data.csv Data/
7 ls Data
```

Files can be moved with the mv command.

```
# remove the copy of data.csv from the Data folder
1
2
  rm Data/data.csv
  ls Data
3
4
  # move the original data file to the Data folder
5
  mv data.csv Data/
  # check the file changed folder
6
7
  ls
8
  ls Data
```

# 4 Bash scripts

A bash script is just like a Python 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.

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

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

```
1 # greetings.sh
2 echo "Hello there!"
3 echo "This is a bash terminal."
4 echo "Welcome."
5 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.

```
1 # syntax: source script_name
2 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**.

```
1 # syntax: which program_name
2 # if the program can be found, then the shell
3 # displays the path to the program binaries
4 which nano
5 which emacs
6 # if the program cannot be found, then nothing is displayed
7 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).

On the cluster, we have Python installed, but it is a super old version that is quite different from the Python we have been using:

```
1
  which python
2
  python --version
```

Python3 was launched in 2008, and offers several improvements over the Python2 version of the language. These improvements also meant that Python3 is backwards incompatible. That is, Python2 code usually does not work with the Python3 interpreter. It has been more than a decade since the introduction of Python3, and Python3 is now the standard. Python2 still receives minor updates, but will be officially retired soon.

On the cluster we still have Python2 installed, and I could not find Python3 installed anywhere. The solution will be to install Python3 and the packages we want ourselves.

#### Installing Python 3 and Packages on the Cluster 5

To install Python, we will use a tool called pyenv. It is similar to the virtualenv we used in the very first class, but it can automatically install several different versions of Python. To install **pyenv**, run the following on bash:

```
# connect to the cluster
1
2
  ssh gfs8@login.econ.duke.edu
3
  # change to ~ folder
  cd ~
4
5
  # install pyenv
6
  curl https://pyenv.run | bash
```

After the installation is done, create the file .bashrc on your home folder:

nano ~/.bashrc

1

Type the following inside your .bashrc file:

```
export PATH="/econ/home/g/gfs8/.pyenv/bin:$PATH"
1
2
  eval "$(pyenv init -)"
3
```

```
eval "$(pyenv virtualenv-init -)"(miniconda3-latest)
```

On the first line, instead of /g/gfs8/, change it to reflect your own home folder.

Whenever you want to work with Python, you should first execute the .bashrc file with the source command. It is similar to what we do with virtualenv.

```
source ~/.bashrc
```

We will now install a Python version that comes with a package manager called conda, which work just like pip. We are using conda because it can install some packages with less issues than pip, when in a restricted environment.

1 2 pyenv install miniconda3-latest
conda install numpy scipy matplotlib pandas

We are now ready to begin.

# 6 Submitting Python Code

Let's create a Python script that will perform some computation and save the results. Create a file named pyscript.py and type the following:

```
import numpy as np
x = np.random.random(200)
y = np.random.random(200)
res = x*y*np.exp(y)
np.savetxt('results.csv', res, delimiter=',')
```

We can now write a bash script that will interact with the operating system and tell Python to execute the script. To execute a Python script we would simply call python pyscript.py. However, we need to make sure that the correct Python version is available and is being used. To do so, we activate our "environment" by calling source ~/.bashrc. We then verify the Python version, and finally execute the script. Create the script named send\_python.sh and type the following:

```
1 pwd
2 date
3 source ~/.bashrc
4 python --version
5 python pyscript.py
```

We can now run this script:

1 source send\_python.sh

It will execute the script and display what is happening on the screen. After it is done, a file named **results.csv** should be on your working directory. You can check it out with:

```
cat results.csv
```

# 7 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 Python 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 and manage. 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.

# 7.1 Shebang

The script send\_python.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:

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

We can still execute it with source:

```
1 source test_shebang.sh
```

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

```
1 # make the file executable
2 chmod +x test_shebang.sh
3 # execute it
4 ./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.

# 7.2 Submitting to the Cluster

Modify the send\_python.sh so that its first line is #!/bin/bash. Delete the results.csv file from the folder.

We can submit the send\_python.sh script to Slurm with the command sbatch.

```
1 # submit script to Slurm
2 sbatch send_python.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.

# 7.3 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 5 for the output of squeue



Figure 5: 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.

1 squeue -u gfs8

See Figure 6 for an example.



Figure 6: 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.

## 7.4 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 7):

```
1 sacct -j 33759
```



Figure 7: View details of a submitted job.

If you want even more information about a specific job, then you can use the command  $\verb+sacct_ec:$ 

1 sacct\_ec -j 33759

Figure 8 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 8: 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.

## 7.5 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.

## 7.6 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 send\_python.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 Python script we submitted also created a file containing the results. This file is also saved in our working directory (but this can be changed within the send\_python.sh script with cd). When we submit a job with Slurm, our home folder is directly accessible by the job, and behaves as a local drive.

# 8 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 9 displays the output of sinfo.



Figure 9: 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 -1 to display extra information.

1 sinfo -N -1

Figure 10 displays the output of sinfo -N -1.



Figure 10: 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<sup>1</sup>, 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

<sup>&</sup>lt;sup>1</sup>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.

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

# 9 Slurm: Requirements and Directives

When it comes to speeding up code execution, we saw that Python allows us to use more than one processor core to speed up loops using the concurrent.futures module. 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 launching multiple Python processes on different cores. We will see how to specify these requirements using directives.

### 9.1 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.

### 9.2 Requesting Memory and CPUs

Let's modify the send\_python.sh script to add some directives.

```
#!/bin/bash
1
2
   # require 12 GB of memory for this job
3
   #SBATCH --mem=12G
   # require 4 CPUS for this job
4
5
   #SBATCH --cpus-per-task=4
6
   pwd
7
   date
8
   source ~/.bashrc
9
   python --version
10
   python pyscript.py
```

The first directive requires a large amount of memory (12 GB). Remember that the more processes you launch, the more memory you need, since Python needs to distribute all of the variables to each of the processes. 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. We will use the built-in module os to obtain the number of physical cores available.

Let's modify the pyscript.py script to run code in parallel:

```
from concurrent.futures import ProcessPoolExecutor
import time
from os import cpu_count
# Count how many cores were allocated to this job
total_cpus = cpu_count()
print(f'Total CPUs Available: {total_cpus}')
def slow_computation(x):
    time.sleep(2)
    return x
# Time code execution as number of CPUS increases
for cpus in range(1, total_cpus + 1):
    start = time.perf_counter()
    with ProcessPoolExecutor(max_workers=cpus) as executor:
        results = executor.map(slow_computation, range(10))
    print(f'CPUs: {cpus} | Time: {time.perf_counter() - start
       :.2f}')
```

The line where cpu\_count is called recovers how many cores are available. The number of cores is then used with ProcessPoolExecutor input max\_workers. This limits how many cores are used to executor the slow\_computation. When max\_workers is one, we are using a single core. The max\_workers increases one by one, until all available cores are being used. We time the execution of the code for each iteration, and print the findings on the screen. The print output is saved in the slurm-XXXXX.out file, which we can later inspect with cat.

Observe that when there is only one core being used, the map becomes a regular for-loop (although the order of execution is random). Since we are calling the slow\_computation function 10 times, and each call takes about 2 seconds, then the total time of execution with a single core should be about 20 seconds.

On the next iteration in the for-loop, the variable cpus becomes two. Now, we run map using two cores. This means that the command sleep(2) will be executed twice at the same time, so the loop will execute faster. Indeed, it should take about 5 seconds to execute, since we have two processes (one at each core) paused at the same time.

Since the number of workers in the cluster can be high, the time to execute our script will get progressively smaller. After submitting the code with sbatch send\_python.sh, we can inspect the output file. Figure 11 displays the output.

n890 Python - 2019 — -bash	90 Python - 2019 — -bash	@dcc-slogin.oit.duke.edu	
<pre>(miniconds&gt;-latest) [gf+80[or] /econ/hose/gr/9768/PT1MONCLASS Mon Jul 15 14:81:37 EDT 2819 Python 3.7.3 Total CPUs Available: 12 CPUs: 1   Time: 20.04 CPUs: 1   Time: 20.04 CPUs: 3   Time: 6.02 CPUs: 3   Time: 6.02 CPUs: 3   Time: 4.02 CPUs: 6   Time: 4.02 CPUs: 6   Time: 4.02 CPUs: 6   Time: 2.03 CPUs: 1   Time: 2.03</pre>	n-06 PYTHONCLASS]\$ cat slurm- n-06 PYTHONCLASS]\$	34137.out	

Figure 11: Output of Python script using multiple cores 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!

### 9.3 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. We will consider the example we have been working with on the lecture about concurrency.

### 9.3.1 Financial Data

We were working with stock data, computing a statistic and bootstrapping its confidence interval. Let's create a script containing several functions. We need functions for:

- Loading and cleaning the data;
- Computing the statistic of interest;
- Obtaining a bootstrap sample;
- Combining results to obtain the confidence interval.

```
import pandas as pd
import numpy as np
import os

def load_stocks(tickers):
    """Computes panel of geometric intraday-returns for stocks.
    """
    folder = ('/Users/guilhermesalome/'
```

```
'Teaching/Duke/Econ890 Python - 2019/supporting/
                 data/StocksHF/')
    # Obtain the data and times from the first file
    date_times = pd.read_csv(f'{folder}{tickers[0]}.csv',
                             skiprows=0, header=None, usecols
                                = [0, 1])
    dt = pd.to_datetime(date_times[0]*10**4+date_times[1],
       format = '%Y%m%d%M%S')
    dt.name = 'Time'
    # Create extractor to simplify list inside pd.concat
    def extract(folder, ticker):
        df = pd.read_csv(f'{folder}{ticker}.csv',
                         skiprows=0,
                         header=None)
        df = df.iloc[:, 2]
        df.name = ticker
        return df
    panel = pd.concat([extract(folder, t) for t in tickers],
       axis=1)
    panel.index = dt
    # Compute geometric returns, but only intraday
    returns = np.log(panel).groupby(panel.index.date).diff()
    returns = returns.dropna()
    return returns
def computeRV(returns):
    return returns.groupby(returns.index.date).apply(lambda x:
       (x**2).sum())
def bootstrap_returns(returns):
    groups = returns.groupby(returns.index.date, group_keys=
       False)
    return groups.apply(lambda group: group.sample(n=group.
       shape[0], replace=True))
def getCI(df_iterable):
    """Computes 99% confidence intervals from a list of pandas
       data frames."""
    all_stats = np.array([df.values for df in df_iterable])
    # Obtain column names
    cols = df_iterable[0].columns.values
    # Create multi index indicate this is the lower bound
    cols_lower = pd.MultiIndex.from_tuples(zip(cols, ['lower']*
       len(cols)))
    # Create data frame with lower bound of confidence interval
```

1

We have most of the tools we need to speed the code execution using the cluster. Before we can actually execute it, we need to first get the stock data from our computer into the cluster.

#### 9.3.2 Transferring Files to 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. 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.

```
1
   # Locate the file on my laptop
2
   ls -1
3
   # fake_data.mat is in the current working directory
   scp fake data.csv gfs8@login.oit.duke.edu:~
4
   # will ask for your password and then start transferring
5
      the file
6
   # Now, ssh into the login node of Duke
   ssh gfs8@login.oit.duke.edu
7
8
   # check that fake data.mat is in the current working
      directory
9
  ls -1
10
  # transfer to the Econ Cluster
```

```
11 scp fake_data.csv gfs8@login.econ.duke.edu:~
12 # ssh into the Econ Cluster
13 ssh gfs8@login.econ.duke.edu
14 # verify that the file was transferred
15 ls -1
```

If you are transferring a lot of files, or want a graphical user interface, then you could use the program Cyberduck, for example.

Another option to get files is the command wget. If you have your file hosted somewhere online, like on Github. Then you can get the url to the file, and use wget url to download the file:

```
1 mkdir PYTHONCLASS
2 cd PYTHONCLASS
3 wget https://raw.githubusercontent.com/python-for-
economists/lecture-notes/master/supporting/data/StocksHF
.zip
4 unzip StocksHF.zip
5 rm StocksHF.zip
```

#### 9.3.3 Modifying the Code to Work on the Cluster

We created a folder named PYTHONCLASS that will have our main python script and our bash script to submit the job to Slurm. Inside this folder, we created another folder named StocksHF when we unzipped the StocksHF.zip file. The financial data is inside this folder, so we need to update the function load\_stocks to reflect that:

```
def load_stocks(tickers):
    """Computes panel of geometric intraday-returns for stocks.
    """
    folder = '/econ/home/g/gfs8/PYTHONCLASS/'
    # rest of the code ...
```

We now need to decide how to break the execution of the code. We can split it by stock, for example, letting a single core work on getting the confidence intervals for a single stock. Or, we can split the problem even further, by computing the confidence interval for each stock, month by month, and so on. Let's try the first approach: split execution by stock.

Create a .py file named rv\_conf\_intervals.py:

```
import pandas as pd
import numpy as np
import os
def load_stocks(tickers):
```

```
"""Computes panel of geometric intraday-returns for stocks.
       0.0.0
    folder = '/econ/home/g/gfs8/PYTHONCLASS/StocksHF'
    # Obtain the data and times from the first file
    date_times = pd.read_csv(f'{folder}{tickers[0]}.csv',
                              skiprows=0, header=None, usecols
                                = [0, 1])
    dt = pd.to_datetime(date_times[0]*10**4+date_times[1],
       format = '%Y\%m\%d\%M\%S')
    dt.name = 'Time'
    # Create extractor to simplify list inside pd.concat
    def extract(folder, ticker):
        df = pd.read_csv(f'{folder}{ticker}.csv',
                         skiprows=0,
                         header=None)
        df = df.iloc[:, 2]
        df.name = ticker
        return df
    panel = pd.concat([extract(folder, t) for t in tickers],
       axis=1)
    panel.index = dt
    # Compute geometric returns, but only intraday
    returns = np.log(panel).groupby(panel.index.date).diff()
    returns = returns.dropna()
    return returns
def computeRV(returns):
    return returns.groupby(returns.index.date).apply(lambda x:
       (x**2).sum())
def bootstrap_returns(returns):
    groups = returns.groupby(returns.index.date, group_keys=
       False)
    return groups.apply(lambda group: group.sample(n=group.
       shape[0], replace=True))
def getCI(df_iterable):
    """Computes 99% confidence intervals from a list of pandas
       data frames."""
    all_stats = np.array([df.values for df in df_iterable])
    # Obtain column names
    cols = df_iterable[0].columns.values
    # Obtain indices
    indices = df_iterable[0].index
    # Create multi index indicate this is the lower bound
```

```
cols_lower = pd.MultiIndex.from_tuples(zip(cols, ['lower']*
   len(cols)))
# Create data frame with lower bound of confidence interval
lower = pd.DataFrame(data=np.quantile(all_stats, 0.005,
   axis=0),
                     index=indices, columns=cols_lower)
# Create multi index indicate this is the upper bound
cols_upper = pd.MultiIndex.from_tuples(zip(cols, ['upper']*
   len(cols)))
# Create data frame with upper bound of confidence interval
upper = pd.DataFrame(data=np.quantile(all_stats, 0.995,
   axis=0),
                     index=indices, columns=cols_upper)
# Merge both frames
ci = pd.merge(left=lower, right=upper, left_index=True,
   right index=True)
# Do a sort on the columns first hierarchy
return ci.sort_index(axis=1, level=0)
```

We have a couple of things missing:

- 1. Need to get the tickers (AAPL, BA, BAC) from the csv files;
- 2. Choose one of the tickers to run the code;
- 3. Obtain the confidence intervals;
- 4. Save the results with the appropriate name.

How do we choose one of the tickers to run the code? Since we will be launching several jobs, each responsible for one stock, we need to have a way to change which stock will be used at each job. The solution to this problem is a Slurm directive called **--array**. Consider the following bash script:

```
1 #!/bin/bash
2 #SBATCH --cpus-per-task=1
3 #SBATCH --mem=4G
4 #SBATCH --array=1-12
5 python rv_conf_intervals.py
```

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 Python 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.

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 Python script and use it to select which stock should be used. The variable name that holds the job number is SLURM\_ARRAY\_TASK\_ID. We can access its value in the bash script with \$SLURM\_ARRAY\_TASK\_ID. We can then pass this value to the Python script:

```
1 #!/bin/bash
2 #SBATCH --cpus-per-task=1
3 #SBATCH --mem=4G
4 #SBATCH --array=1-12
5 python rv_conf_intervals.py $SLURM_ARRAY_TASK_ID
```

Now, inside the rv\_conf\_intervals.py script, we can access the value of the variable SLURM\_ARRAY\_TASK\_ID:

```
import sys
print(f'Script Name: {sys.argv[0]'}
print(f'SLURM_ARRAY_TASK_ID = {sys.argv[1]}')
# the variables are passed as strings, so we need to conver to
    the appropriate format
job_number = int(sys.argv[1])
```

We can use this number to decide which stock to use.

After obtaining the confidence intervals, we need to store the resulting data frame in a file. We can do so, with the data frame method to\_csv.

Let's modify the rv\_conf\_intervals.py script to incorporate these changes:

```
import pandas as pd
import numpy as np
import os
import sys
import time
folder = '/econ/home/g/gfs8/PYTHONCLASS/StocksHF/'
# Create a main function that will connect all the pieces
def main(folder=folder):
    # Obtain job number
    # Remember that try blocks do not create a new scope
    try:
        job_number = int(sys.argv[1])
    except IndexError:
        message = ("Expected a job number to be passed to the
           script. "
                   "This script should be executed via the CLI.
                   "Specifically: python rv_conf_intervals.py
                      $SLURM_ARRAY_TASK_ID")
        raise IndexError(message)
    except ValueError:
        message = (f"Expected an integer, but got a {type(sys.
           argv[1])}.")
```

```
# Obtain tickers
    tickers = sorted([f.strip('.csv') for f in os.listdir(
       folder) if f.endswith('.csv')])
   print(tickers)
   # Load data
   # we need to pass an iterable, even if it only has one
       element
   print(f"Loading returns: {tickers[job_number]}")
   ret = load_stocks([tickers[job_number]])
    total_bsamples = 10
    print(f"Bootstrap loop with {total_bsamples} samples")
    df_iterable = [computeRV(bootstrap_returns(ret)) for _ in
       range(total_bsamples)]
   print("Computing confidence intervals")
    ci = getCI(df_iterable)
    print("Storing results")
    results_file = f'RV_CI_{tickers[job_number]}.csv'
    ci.to_csv(results_file)
    print("Done!")
def load_stocks(tickers, folder=folder):
    """Computes panel of geometric intraday-returns for stocks.
       0.0.0
    # Obtain the data and times from the first file
    date_times = pd.read_csv(f'{folder}{tickers[0]}.csv',
                              skiprows=0, header=None, usecols
                                =[0, 1])
    dt = pd.to_datetime(date_times[0]*10**4+date_times[1],
       format = '%Y\%m\%d\%M\%S')
    dt.name = 'Time'
   # Create extractor to simplify list inside pd.concat
    def extract(folder, ticker):
        df = pd.read_csv(f'{folder}{ticker}.csv',
                         skiprows=0,
                         header=None)
        df = df.iloc[:, 2]
        df.name = ticker
        return df
    panel = pd.concat([extract(folder, t) for t in tickers],
       axis=1)
   panel.index = dt
   # Compute geometric returns, but only intraday
   returns = np.log(panel).groupby(panel.index.date).diff()
   returns = returns.dropna()
    return returns
```

```
def computeRV(returns):
    return returns.groupby(returns.index.date).apply(lambda x:
       (x**2).sum())
def bootstrap_returns(returns):
    groups = returns.groupby(returns.index.date, group_keys=
       False)
    return groups.apply(lambda group: group.sample(n=group.
       shape[0], replace=True))
def getCI(df_iterable):
    """Computes 99% confidence intervals from a list of pandas
       data frames."""
    all_stats = np.array([df.values for df in df_iterable])
    # Obtain column names
    cols = df_iterable[0].columns.values
    # Obtain indices
    indices = df_iterable[0].index
    # Create multi index indicate this is the lower bound
    cols_lower = pd.MultiIndex.from_tuples(zip(cols, ['lower']*
       len(cols)))
    # Create data frame with lower bound of confidence interval
    lower = pd.DataFrame(data=np.quantile(all_stats, 0.005,
       axis=0),
                         index=indices, columns=cols_lower)
    # Create multi index indicate this is the upper bound
    cols_upper = pd.MultiIndex.from_tuples(zip(cols, ['upper']*
       len(cols)))
    # Create data frame with upper bound of confidence interval
    upper = pd.DataFrame(data=np.quantile(all_stats, 0.995,
       axis=0),
                         index=indices, columns=cols_upper)
    # Merge both frames
    ci = pd.merge(left=lower, right=upper, left_index=True,
       right_index=True)
    # Do a sort on the columns first hierarchy
    return ci.sort_index(axis=1, level=0)
# Call the main function after everything is defined
start = time.perf_counter()
main()
print(f"Total Time: {time.perf_counter() - start} seconds")
```

The variable job\_number is initialized by the bash script. We use it to select the stock ticker, which in turns defines the name for the .csv file which holds the results. Thus, 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 can also use the job\_number to set the seed

for random number generation, for example.

Create the bash script send\_rv\_conf\_intervals.sh:

```
1
  #!/bin/bash
2
  #SBATCH --cpus-per-task=1
3
  #SBATCH --mem-per-cpu=4G
  #SBATCH --array=0-11
4
   echo $SLURM_ARRAY_TASK_ID
5
  source ~/.bashrc
6
7
   pwd
8
  date
9
  python --version
10
  python rv_conf_intervals.py $SLURM_ARRAY_TASK_ID
```

We use the directive --mem-per-cpu to specify how much memory should be available to each CPU. Since we are launching multiple jobs with a single CPU each, this defines how much memory should be available per job. Notice we change the values passed by --array so that they match how many stocks we have, and also the way Python indexes lists (starting at 0).

We can submit the script to Slurm and verify that indeed 12 jobs were created:

```
1
   # Submit script for execution
   sbatch send rv conf intervals.sh
2
   #+END_SRC bash
3
4
   A job id is assigned.
5
   Look at all jobs that were created:
6
   #+ATTR LATEX: :options style=bash
7
   #+BEGIN SRC bash -n
8
   squeue -u gfs8
9
   #+END SRC bash
10
11
12
   Notice that =JOBID= follows the fomrat =xxxxxx_y=, where =
      xxxxxx= is the job id, and =y= is the job number
      specified in the =--array= directive.
   Also notice that many =slurm-xxxxxx_y.out= files were
13
      created:
14
   #+ATTR_LATEX: :options style=bash
   #+BEGIN_SRC bash -n
15
16
   ls -1 slurm-*
17
   # The * above expands slurm- to all existing filenames that
18
   # begin with slurm-
   # To print in the correct order (sorted numerically instead
19
       of alphabetically):
20
  ls -1 -v slurm*
```

```
# To see the output of all the .out files, we need to pass
21
      the results of the
22
   # ls command as the input of the cat command:
   ls -1 -v slurm* | xargs cat
23
24
   #+END SRC bash
25
26
   The jobs should be done now:
27
   #+ATTR LATEX: :options style=bash
28
   #+BEGIN SRC bash -n
29
   squeue -u gfs8
```

Notice that the .csv files were created:

```
1 ls -1 *.csv
2 # We can sort the names numerically with the option -v
3 ls -1 -v *.csv
```

We can print them on the screen with cat:

```
1 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.

#### 9.3.4 Observations

When we submitted the code to the cluster, we set the number of bootstrap samples to a low number (10). In practice, however, we would need that number to be much higher, at least 1000. When the number of bootstrap samples is much higher, the code will take a long time to run, and the .out file will be updated very infrequently. To force the file to be updated, we need to modify the print commands to print(..., flush=True).

We can get a sense for how much memory our code needs, by computing how many bytes of memory the data frame objects use. The method **getsizeof** of the **sys** module returns the size (in bytes) of any Python object:

```
rv = computeRV(ret)
print(f"Size of Data Frame: {sys.getsizeof(rv)*10**-6} MB")
```

It does not seem our script will use a lot of memory. In our bash script, we requested for 4 gigabytes of memory, which should be more than enough.

# 10 Digital Ocean

The Economics Cluster can be used to speed up computation, but is limited to current Duke students and faculty, and, at times, can be difficult to use due to large queues. Companies that offer infrastructure as a service (IAAS) come to the rescue when we need to use a cluster (or one powerful computer), but can no longer access the department cluster. These companies basically offer compute power by the hour. For example, we can start a computer with however many cores and memory we require, install the software we need, run our code, extract the results, and then shut off the computer. This setup can be very efficient, since you only need to pay for the hours you use, eliminating the high fixed cost of buying and setting up a high-performance computer.

We will use DigitalOcean due to its simplicity, but other companies also offer similar services (Google Compute Engine and Amazon EC2). The service itself is not free, but if you sign up for DigitalOcean following this link, then you will get a \$50 credit to spend over 30 days. You can also get more credit for being a student by signing up for Github's education package.

## 10.1 Modifying Code to Work on a Single Computer with Several Cores

We will use DigitalOcean to run our bootstrapping code on a single computer with several cores. To do so, we will use the **concurrent** module. The idea is to have each core execute the **main** function for a different stock, saving the results on a folder as the function finishes running. Let's modify the code to account for the new architecture:

```
import pandas as pd
import numpy as np
import os
import sys
import time
from concurrent import futures
# Create a main function that will connect all the pieces
def main(job_number, folder, total_bsamples=10):
    # Load data
    # we need to pass an iterable, even if it only has one
       element
    ret = load_stocks([tickers[job_number]], folder)
    df_iterable = [computeRV(bootstrap_returns(ret)) for _ in
       range(total_bsamples)]
    ci = getCI(df_iterable)
    results_file = f'RV_CI_{tickers[job_number]}.csv'
    ci.to_csv(results_file)
```

```
return True
def load_stocks(tickers, folder):
    """Computes panel of geometric intraday-returns for stocks.
       .....
    # Obtain the data and times from the first file
    date_times = pd.read_csv(f'{folder}{tickers[0]}.csv',
                             skiprows=0, header=None, usecols
                                =[0, 1])
    dt = pd.to_datetime(date_times[0]*10**4+date_times[1],
       format = '%Y\%m\%d\%M\%S')
    dt.name = 'Time'
    # Create extractor to simplify list inside pd.concat
    def extract(folder, ticker):
        df = pd.read_csv(f'{folder}{ticker}.csv',
                         skiprows=0,
                         header=None)
        df = df.iloc[:, 2]
        df.name = ticker
        return df
    panel = pd.concat([extract(folder, t) for t in tickers],
      axis=1)
    panel.index = dt
    # Compute geometric returns, but only intraday
    returns = np.log(panel).groupby(panel.index.date).diff()
    returns = returns.dropna()
    return returns
def computeRV(returns):
    return returns.groupby(returns.index.date).apply(lambda x:
       (x**2).sum())
def bootstrap_returns(returns):
    groups = returns.groupby(returns.index.date, group_keys=
       False)
    return groups.apply(lambda group: group.sample(n=group.
       shape[0], replace=True))
def getCI(df_iterable):
    """Computes 99% confidence intervals from a list of pandas
      data frames."""
    all_stats = np.array([df.values for df in df_iterable])
    # Obtain column names
    cols = df_iterable[0].columns.values
    # Obtain indices
```

```
indices = df_iterable[0].index
    # Create MultiIndex for the lower bound
    cols_lower = pd.MultiIndex.from_tuples(zip(cols, ['lower']*
       len(cols)))
    # Create df for the lower bound of confidence interval
    lower = pd.DataFrame(data=np.quantile(all_stats, 0.005,
       axis=0),
                         index=indices, columns=cols_lower)
    # Create MultiIndex for the upper bound
    cols_upper = pd.MultiIndex.from_tuples(zip(cols, ['upper']*
       len(cols)))
    # Create df for the upper bound of confidence interval
    upper = pd.DataFrame(data=np.quantile(all_stats, 0.995,
       axis=0),
                         index=indices, columns=cols_upper)
    # Merge both frames
    ci = pd.merge(left=lower, right=upper, left_index=True,
       right_index=True)
    # Do a sort on the columns first hierarchy
    return ci.sort_index(axis=1, level=0)
# Execute main() for the different stock tickers
total_cpus = os.cpu_count()
print(f"Total CPUs: {total_cpus}")
folder = '/Users/guilhermesalome/Teaching/Duke/Econ890 Python -
    2019/supporting/data/StocksHF/'
tickers = sorted([f.strip('.csv') for f in os.listdir(folder)
   if f.endswith('.csv')])
start = time.perf_counter()
with futures.ProcessPoolExecutor(max_workers=total_cpus) as
   executor:
    to_do = \{\}
    for i in range(len(tickers)):
        future = executor.submit(main, job_number=i, folder=
           folder, total_bsamples=10)
        to_do[future] = i
        print(f"Scheduled {tickers[i]} at {future}")
    results = {}
    for future in futures.as_completed(to_do.keys()):
        i = to_do[future]
        value = future.result()
        results[tickers[i]] = value
        print(f'{tickers[i]}: {value}')
stop = time.perf_counter()
print(f"Total Time: {stop - start} seconds")
```

Test the code works with a few bootstrap samples and two cores. The only change necessary to make this code work on a different computer is to **change the folder variable** to the appropriate value.

# 10.2 Launching a Droplet

DigitalOcean uses the word droplet to refer to their compute nodes. To launch a droplet, first log in with your account, then go on the menu Manage and click on Droplets (see Figure 12).



Figure 12: Droplets menu when no Droplets exist.

Click on Create Droplet, and a new page should open. On the new page, choose the following options:

- Distributions: Ubuntu 18 x64
- Choose a plan: CPU Optimized, option with 4 GB of memory 2 CPUs
- Choose a datacenter region: New York or San Francisco
- Authentication: One-time password

Then click on the button to create the droplet. At the time of writing, this system costs 0.06 dollars per hour. You will be charge this amount per hour, and the charge will only stop after you shut down and delete your droplet.

The droplet will be initialized, and you will receive an email with the password to log in on the droplet. After the droplet is done initializing, you should see a page similar to the one depicted in Figure 13.

••• <> ••	Sing 1 A Could digitaloreen.com	• @ @
	Q         E         #         E	eet Create~ (8) 🚇 USAGE (19)
PROJECTS     A     guilhermesalome     + New Project	guilhermesalome www. Update your project information under Settings	→ Move Resources
MANAGE ^ Droplets Kubernetes	Resources         Arthvity         Settings           DROPLETS (1)         159.65.211.37	
Databases Spaces Images	Create something new Leas	rn more
Networking Monitoring API	Create a Managed Database Worry-free database monagement Worry-free database monagement Storage and the scalable object Tech storage and	fuct Docs nnical overviews, how-tos, release notes, support material
DISCOVER ^	Spin up a Load Balancer Dow Distributes traffic between multiple Droplets Ran	rials Ops and development guidelines Docs your resources programmatically
manacapiace	Build on what you have Ask	a question

Figure 13: Droplet was initialized and is now available for usage.

You can use **ssh** to connect to the droplet. The username is **root**, and the host name is the ip address of the droplet, which was sent to your email and is also shown on the website.

1

ssh root@159.65.231.37

You should answer **yes** to the security question. After connecting, you will be immediately prompted for changing your password. After that, we will need to install Python and all the packages we require. Ubuntu, which is the operating system of our droplet, has an application for downloading packages and installing them: **apt-get**. You can think of it as an "App Store" that you use via the command line interface (CLI).)

```
1
   # update the packages available in the "app store"
2
   apt-get update
3
   # install python 3 and the package manager pip
   apt-get install python3 python3-pip
4
   # in some systems, python is installed as python3
5
   python3 --version
6
7
   # the same thing goes for pip
   pip3 --version
8
   # install virtualenv
9
   pip3 install virtualenv
10
   # create new virtual environment
11
   virtualenv py36 -p python3
12
   # activeate environment
13
   source py36/bin/activate
14
15
  # update pip
   pip install -U pip
16
   # install required packages
17
   pip install numpy pandas
18
```

### 10.3 Getting the Code on the Droplet

Get the data on the droplet with wget:

```
1
  wget https://raw.githubusercontent.com/python-for-
     economists/lecture-notes/master/supporting/data/StocksHF
     .zip
2
  # install unzip
  apt-get install unzip
3
  # unzip data
4
  unzip StocksHF.zip
5
  rm StocksHF.zip
6
7
  # copy the script to the computer
```

```
8 nano rv_conf_intervals.py
```

Remember to update the variable folder.

### 10.4 Executing and Extracting the Results

We can now run the code by simply typing python rv\_conf\_intervals.py. Always test the code with a smaller sample and a small number of cores. This is important to iron out any other issues that might appear, and to keep costs low while we are testing. To extract the results from the computer you can use scp, Git and Github, Cyberduck, or any other method you prefer.

After you test your code on a smaller droplet, you may choose to increase the number of CPUs available to the script. It is easy to do that in DigitalOcean. Go back to the droplet page, and look for the menu option "More" and then choose "Resize". On the "Resize" page (see Figure 14), you can increase the number of CPUs and memory available to your droplet. This requires the droplet to temporarily shut down. When it turns on again, all your files and packages are still going to be in the same place, and you can just run the code.

	12 % # #	🔒 cloud digitalocean.com	c	000
al 🙃 🕭 🦛 I	🗄 O 👌 🖬 🥖 🦛	-V- ③ DigitalOcea	n - sbuntu-c-2-4gib-nyc1-01	
ລ	Q, Search by resource nam	e or IP (Cmd+B)	Create V 🛞 🔔	USAGE \$0.00
BOJECTS A	<b>ubuntu-c</b> -	2-4gib-nyc1-01 esalome / 4 GB Memory / 25 GB Disk / NYC1 - Ubuntu 18.04	1 ×64	
	ipv4: 159.65.231.37	ipv6: Enable now Private IP: Enable now Floatin	ng IP: Enable now	Console: 🗗
roplets	Graphs			
bernetes	Access	Resize Droplet		
lumes	Power	Resize Droplet		
tabarce	Volumes	This Droplet is on a cpu intensive plan. You must power	ar off your Droplet to resize it.	
	Nesize			
aces	Backups	CPU and RAM only	Disk, CPU and RAM	
ages	Snapshots	<ul> <li>This will only increase or decrease the CPU and RAM of your Droplet not disk</li> </ul>	This will increase the disk size, CPI     RAM of your Droplet. This is a nem	Jand
tworking	Kernel	size. This can be reversed.	change and cannot be reversed.	
	History			
	Destroy			
	Tags			
	Recovery	Standard Droplets	General Purpose	
		Standard virtual machines with a mix of memory and	High performance virtual machines with a	good
arketplace		compute resources. Best for small projects that can handle variable levels of CPU performance, like blogs, web apps and dev/test environments.	balance of memory and dedicated hyper- from best in class Intel processors. A grea a wide range of mainstream, production w	threads it choice for vorkloads,

Figure 14: Droplet resize page.

When the droplet finishes resizing, you need to turn it on again (by clicking on the turn on button at the top of the page). Then, use **ssh** to connect back to the droplet. You can now execute your code with far more bootstrap samples:

```
1
  # connect to droplet
  ssh root@159.65.231.37
2
3
  # activate environment
  source py36/bin/activate
4
  # update number of bootstrap samples in the script to 1000
5
6
  emacs rv_conf_intervals.py
7
  # total bsamples=1000
  # run the code
8
9
  python3 rv_conf_intervals.py
```

Notice that we are only launching 12 new Python processes, and our code does not make use of more than 13 cores. If you get a computer with more than 13 cores, you should try to adapt your code to make use of all of the compute power. Otherwise, use a computer with less cores to keep costs low.

After the code is done running, retrieve its results. Then, if you are done, destroy the droplet to stop charges to your account (see Figure 15). When the droplet is destroyed, all of the files are permanently deleted.



Figure 15: Destroy the Droplet to limit costs.

# 11 Assignment

### Problem 1 (Optional)

Modify the scripts send\_rv\_conf\_intervals.sh and rv\_conf\_intervals.py 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 Python script that takes all of the .csv files generated by the script rv\_conf\_intervals.py and creates a single pandas data frame containing all of the results. Then save the results in a .csv file.

**Problem 3** What is the Hierarchical Data Format useful for? Does pandas offer any API to deal with HDF files? If so, describe how to use this API (how to save a data frame in this format, how to load a data frame saved in the format).

### Problem 4 (Optional)

Modify  $rv_conf_intervals.py$  to use the HDF file format. Save the results in this format. Compare the file sizes with the .csv file sizes. What are the pros and cons of using the HDF file format when compared to the .csv format?

### Problem 5 (Optional)

 $Compare \ the \ advantages \ and \ disadvantages \ of \ the \ following \ takes \ on \ parallel \ computing \ with \ the \ cluster:$ 

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

• Launching several jobs that use several cores.

### Problem 6 (Optional)

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 7 (Optional)

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:

$$\max \sum_{t=0}^{\infty} \beta^{t} U(c_{t})$$
  
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 a minimization function to solve for k. You may want to add the constraint that c should always be positive.

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