Lab 1

Parallel Computing with `ipyparallel`

Lab Objective: *Most computers today have multiple processors or multiple processor cores which allow various processes to run simultaneously. To perform enormous computations, "supercomputers" or computer clusters that combine many processors and a great deal of memory are commonly used. In this lab, we will explore the basic principles of designing code that fully utilizes available resources for parallel computing using the* `iPyParallel` *python package.*

Why Parallel Computing?

When a single processor takes too long to perform a computationally intensive task, there are two simple solutions: build a faster processor or use multiple processors to work on the same task. Unfortunately, as processors have become smaller, it has become increasingly difficult to dissipate the heat they produce. This problem is called the "heat wall" and has presented a currently insurmountable barrier. Therefore, the second solution has taken precedence and seen incredible growth in the past two decades. Though there are many different architectures for parallel computing, essentially, a 'supercomputer' is made up of many normal computers which share or use their own memory.

In the majority of circumstances, these processors communicate with each other and coordinate their tasks with a message passing system. The details of this message passing system, MPI, will be the topic of the next lab.

In this lab, we will become familiar with some of the basic ideas behind parallel computing.

Serial Execution vs. Parallel Execution

Up to this point, all the programs you have written are executed one line at a time, or in *serial*. The following exercise will help visualize the serial process of a program.

Problem 1. If you are working on a Linux computer, open a terminal and execute the `htop` command. (If `htop` is not on your system, install it using your default package manager). When opening this program, your terminal should see an interface similar to Figure 1.1. The numbered bars at the top represent each of the cores of your processor and the workload on each of these cores.

Now, run the following python code with your terminal running `htop` still visible. The sole purpose of the following code is to create a computationally intensive function that runs for about 15 seconds.

```
import numpy as np
for i in xrange(10000):
    np.random.random(100000)
```

You should have seen one of the cores get maxed out at 100%. It is also possible that you saw the load-carrying core switch midway through the execution of the file. This is evidence one indicator that our script is being executed in serial – one line at a time, one core at a time.

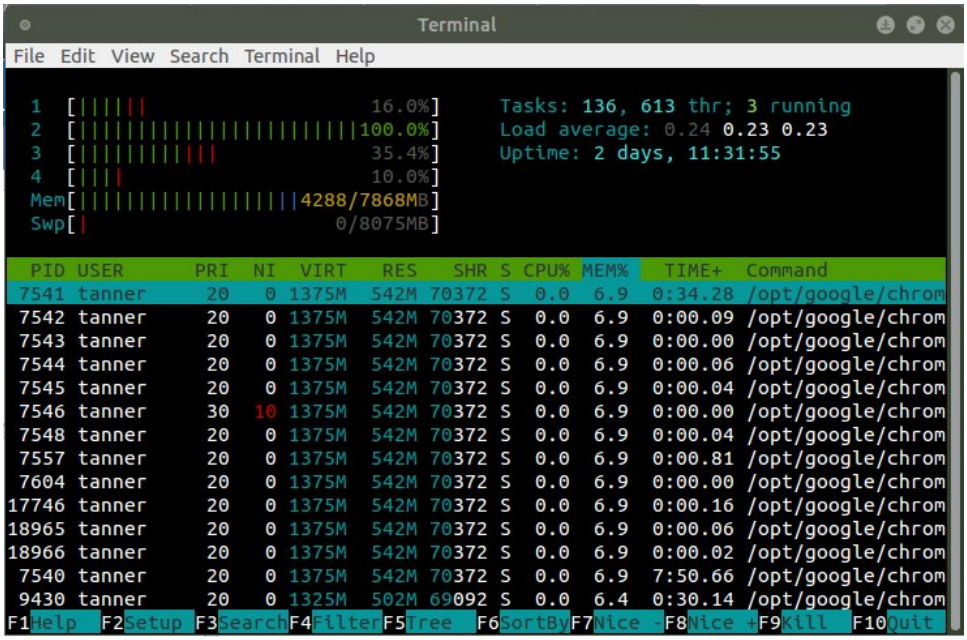

Figure 1.1: An example of `htop` with a computationally intense python script running.

As you saw in the exercise above, only one of the cores was carrying the load at a time (if it was more than one, your computer has default code to try distributing some of the process). This means that we are only using a fraction of the computer's resources. When working on a personal computer, this would often be to your

benefit because dividing jobs among multiple cores is part of what makes smooth multitasking possible. However, in the event you wish to devote all the computer's resources to executing a certain program, we can employ the help of the `ipyparallel` module. In theory, you can make your code run N times faster when executing in parallel where N is the number of cores you have access to.

The `ipyparallel` Module

We will begin our discussion on parallel computing by learning about the `ipyparallel` module. Since Python is a relatively slow scripting language, and since the main purpose of parallel computing is to speed up run time, most parallel computing in application is done in a language other than Python. Though this may not be the fastest parallel computing framework available, it is still fairly easy to take advantage of all the cores available to speed up run time and to test parallel code logic. This is done by specifying what happens on each core, which is core principle of parallel computing.

Installation and Initialization of `ipyparallel`

If you have not already installed `ipyparallel`, you may do so using the conda package manager.

```
$ conda update conda
$ conda update anaconda
$ conda install ipyparallel
```

With `ipyparallel` installed, we can now initialize an IPython cluster that is comprised of iPyParallel *engines*. By default, an engine will be started on each of your machines processor cores and a controller will be started to communicate with each of the engines. The controller can be accessed through the `Client` object, which has two classes, `DirectView`, and `LoadBalancedView`. We will discuss these classes in further detail later.

We won't go too much into the architecture of the IPython cluster, but if you are interested in learning more, visit `https://ipyparallel.readthedocs.io/en/latest/intro.html#architecture-overview`.

Now to initialize an IPython cluster, run the following code:

```
$ ipcluster start
```

If you would like to specify the number of engines to initialize, run the following:

```
# Start a cluster with 8 engines.
$ ipcluster start --n 8
```

If you choose to explicitly specify the number of engines, it is not optimal to initialize more engines than you have processors. Doing so would require multitasking on each processor instead of having each processor dedicated to one task.

If you are more accustomed to using Jupyter Notebooks, you may have noticed the "Clusters" tab. You can start an IPython cluster in this tab after enabling the `ipcluster` notebook extension.

```
$ ipcluster nbextension enable
```

Problem 2. Initialize an IPython cluster with an engine for each processor. As you did in the previous problem, open `htop`. Run the following code and examine what happens in htop.

```
from ipyparallel import Client
client = Client()
dview = client[:]

dview.execute("""
import numpy as np
for i in xrange(10000):
    np.random.random(100000)
""")
```

The output of `htop` should appear similar to Figure 1.2. Notice that all of the processors are being utilized to run the script.

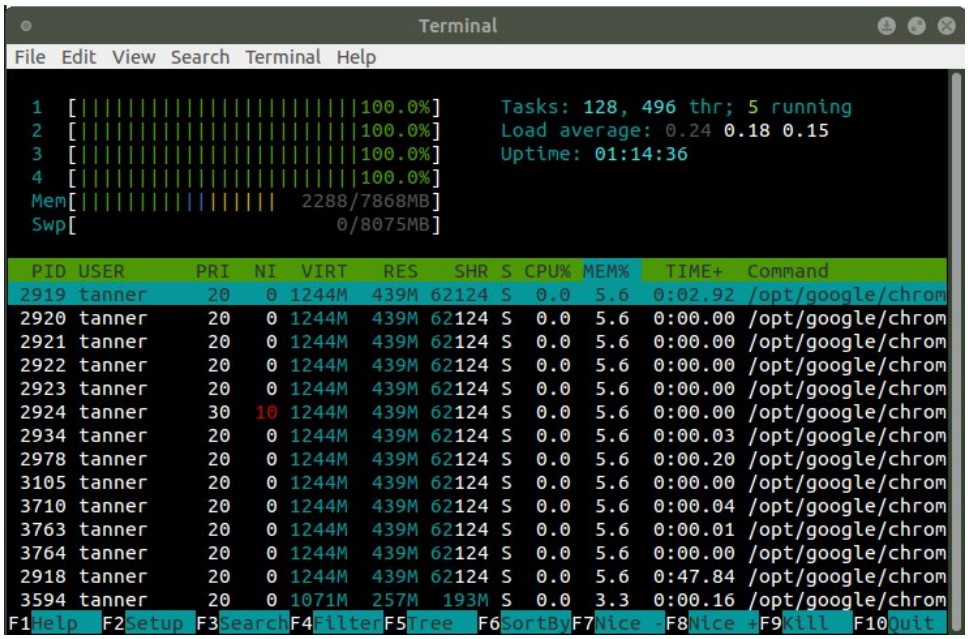

Figure 1.2: An example of `htop` with a computationally intense python script running in parallel.

This result ensures that our IPython cluster is successfully executing on all engines. We can now dive into the details of what syntax we can use to utilize the cluster.

Syntax for `ipyparallel`

The basic framework for `ipyparallel` revolves around a `DirectView` or a `LoadBalancedView`. A `DirectView` is the object through which we can communicate with each of the engines individually and gives us control over which variables are pushed to each engine and what functions are performed. A `LoadBalancedView` takes the commands that are being executed and does its best to distribute the load evenly across all engines.

For the purposes of learning how each engine works, we will focus on the `DirectView` in this lab. To initialize a `DirectView`, run the following code:

```
>>> from ipyparallel import Client
>>> client = Client()

# Verify the dview has been generated correctly.
# If you had four processors, the output would be as follows.
>>> client.ids
[0, 1, 2, 3]

# Initialize DirectView
>>> dview = client[:]
```

Variables on Different Engines

When using multiple processors, you can imagine each engine running its own iPython terminal with its own namespace. This implies that any variable we want to use must be initialized on each engine. There are a few different ways to do this.

```
# To share the variable 'a' across all engines
>>> a = 10
>>> b = 5
>>> dview["a"] = a
>>> dview["b"] = b

# Or alternatively,
>>> dview.push({'a':a, 'b':b})

# To ensure the variables are on engine 0
>>> client[0]['a']
10

>>> client[0]['b']
5
```

The code you just ran is the easiest way to get individual values on each of the engines. We will discuss a couple of other methods further on. We will now move on to simple computations in parallel.

The `apply()` and `apply_sync()` Methods

To execute functions on each of the engines, we can use the `apply()` or the `apply_sync()` methods which are differentiated by the presence of *blocking*. If a function is

executed with blocking, you will be unable to send any other commands to the engine until the function has finished. If a function is run without blocking, you can still execute additional commands on that engine while the function is still computing its result. Though seemingly unimportant, blocking is an important principle if your computing processors or *nodes* are communicating one with another during a parallel process. The `apply()` methods executes without blocking and the `apply_sync()` method executes with blocking. The following code box describes how this is done:

```
>>> def add():
...     return a+b

# Runs add() without blocking (in background)
>>> result = dview.apply(add)

# Checks to see if the output is ready
>>> result.ready()
True

# Retrieves output
>>> result.get()
[15, 15, 15, 15]

# Runs add() with blocking
>>> dview.apply_sync(add)
[15, 15, 15, 15]
```

You can also pass variables into your function as part of call to `apply_sync()`. Consider the following example:

```
>>> def add(x, y):
...    return x+y

>>> dview.apply_sync(add, 3, 6)
[9, 9, 9, 9]
```

To this point, the examples of what you can do with parallel computing may not seem too interesting since each engine is producing the same result. There are, however, circumstances in which the engines return different results. In these situations, parallel computing can drastically speed up the result.

```
# A function that draws four samples from a standard normal distribution
>>> def draw():
...    import numpy as np
...    a = np.random.randn(4)
...    return a

# Runs draw() on all engines simultaneously and returns the results
>>> dview.apply_sync(draw)
[array([-0.7277754 , -0.39273127,  0.05636817, -0.26855806]),
 array([ 0.46569263,  0.63911368, -0.02812979,  1.63223456]),
 array([ 0.92278649, -1.42868485,  0.32370856, -0.2386319 ]),
 array([-0.93787564,  1.16286507, -0.0388443 , -1.10649599])]
```

Note

In the code above, notice that NumPy is imported within the function. Since each engine has its own namespace, we must ensure that the desired modules are imported in each engine. There is more than one way to do this.

For example, the following command imports NumPy to all engines simultaneously.

```
>>> dview.execute("import numpy as np")
```

Problem 3. Using `apply_sync()`, draw n samples from a standard normal distribution where n is default to 1,000,000. Print the mean, max, and min for draws on each individual engine. For example if you have four engines running, your output should look like:

```
means = [0.0031776784, -0.0058112042, 0.0012574772, -0.0059655951]
maxs = [4.0388107, 4.3664958, 4.2060184, 4.3391623]
mins = [-4.1508589, -4.3848019, -4.1313324, -4.2826519]
```

In theory, using parallel computing for this problem should be approximately N times faster where N is the number of engines you are using. In practice, however, the scaling is not quite linear. This is due in part to the controller running on one of the engines, your computers standard processes still running, and the overhead of communication from the controller with the engines. We test this in the following problem.

Problem 4. Using the function you wrote and passed into `apply_sync()` in the previous problem, compare the time it takes to run the function with parallel computing to the time it takes to run the function serially. That is, time how long it takes to run the function on all of your machine's engines simultaneously using `apply_sync`, and how long it takes to run the function in a `for` loop n times, where n is the number of engines on your machine. Print the results for 1,000,000, 5,000,000, 10,000,000, and 15,000,000 samples. You should notice an increase in efficiency as the problem size increases.

Problem 5. Now let's do a problem that is a bit more computationally intensive. Define the random variable X to be the maximum out of N draws from the standard normal distribution. For example, one draw from X when $N = 10$ would be the maximum out of 10 draws from the normal

distribution. Write a function that accepts an integer N, takes 500,000 draws from this distribution (X), and plots the draws in a histogram. The resulting histogram will approximate the p.d.f. of X.

Write your function in such a way that each engine will carry an equal load. Also write your function in such a way that it is flexible to the number of engines that are running. HINT: Remember that you can get a list of all available engines using `clients.ids`.

The `scatter()` and `gather()` Methods

There are many situations where we would want to spread a dataset across all the available engines. This way, we can have a function work on each of these portions of the dataset. In its simplest form, this is the basis of the MapReduce program which will address in more detail in a future lab.

We will first introduce an example of `scatter()` and explain the proper usage in more detail throughout the example.

```
# Initialize the dataset to scatter
>>> a = np.arange(10)

# Scatter the data. The pieces of the data will be
#   named "a_partition" on each of the engines.
>>> dview.scatter("a_partition", a)

# Verify that the data has been successfully scattered.
#    Notice that the data has been scattered as
#    equally as possible.
>>> client[0]["a_partition"]
array([0, 1, 2])

>>> client[1]["a_partition"]
array([3, 4, 5])

>>> client[2]["a_partition"]
array([6, 7])

>>> client[3]["a_partition"]
array([8, 9])
```

Now that the `a_partition` variable has been initialized on each engine, we can now execute functions that depend on this variable. Consider the following simple example using the `execute()` method.

```
# Pass a string with the Python code that we wish to run
#   on each engine. This code will simply sum the entries
#   in "a_partition"
>>> dview.execute("""
... b = a_partition.sum()
... """)
```

> **Note**
>
> If you are using a Jupyter Notebook, there is a built in magic function that is analagous to `dview.execute()`. If you put the `%%px` magic at the beginning of a cell of code, that cell of code will be executed on each engine. This tool is very useful for designing and debugging parallel algorithms.

We have now computed the sum of each of these pieces of the data and stored the result in the variable *b*. To gain access to these results, we use the `gather()` method.

```
# Gather all the 'b' values into a list.
>>> b_list = dview.gather("b", block=True)
[3, 12, 13, 17]

>>> sum(b_list)
45
```

To summarize, this example has taken a piece of data, scattered it to all available engines, performed a computation on each of these pieces of data, then gathered the results back to the controller.

Applications

Parallel computing, when used correctly, is one of the best ways to speed up the run time of an algorithm. As a result, it is very commonly used today and has many applications, such as the following:

- Graphic rendering
- Facial recognition with large databases
- Numerical integration
- Calculating Discrete Fourier Transforms
- Simulation of various natural processes (weather, genetics, etc.)
- Natural language processing

In fact, there are many problems that are only possible to solve through parallel computing because solving them serially would take too long. In these types of problems, even the parallel solution could take years. Some brute-force algorithms, like those used to crack well designed encryptions, are examples of this type of problem.

The problems mentioned above are well suited to parallel computing because they can be manipulated in a way such that running them on multiple processors results in a significant run time improvement. Manipulating an algorithm to be run with parallel computing is called *parallelizing* the algorithm. When an problem only requires very minor manipulations to parallelize, it is often called *embarrassingly*

parallel. Typically, an algorithm is embarrassingly parallel when there is little to no dependency between results. Algorithms that do not meet this criteria can still technically be parallelized, but there is not a significant enough improvement in run time to make this worthwhile. For example, calculating the Fibonacci sequence using the usual formula, $\mathrm{F}(n) = \mathrm{F}(n-1) + \mathrm{F}(n-2)$, is poorly suited to parallel computing because each element of the sequence is dependent on the previous two elements.

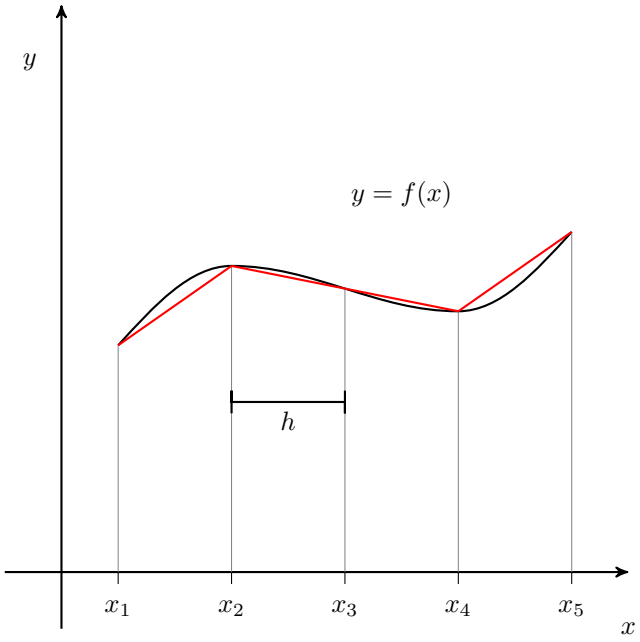

Figure 1.3: A depiction of the trapezoidal rule with uniform partitioning.

Problem 6. Consider the problem of numerical integration using the trapezoidal rule, depicted in Figure **??**. Recall the following formula for estimating an integral using the trapezoidal rule,

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{k=1}^{N}(f(x_{k+1}) + f(x_k)),$$

where x_k is the kth point, and h is the distance between any two points (note they are evenly spaced).

Note that estimation of the area of each interval is independent of all other intervals. As a result, this problem is considered to be embarrassingly parallel.

Write a function called `parallel_trapezoidal_rule()` that accepts a function handle to integrate, bounds of integration, and the number of points to use for

the approximation. Utilize what you have learned about parallel computing to parallelize the trapezoidal rule in order to estimate the integral of f. That is, evenly divide the points among all available processors and run the trapezoidal rule on each portion simultaneously. The sum of the results of all the processors will be the estimation of the integral over the entire interval of integration. Return this sum.