15 Introduction to Parallel Computing

Lab Objective: *Many modern problems involve so many computations that running them on a single processor is impractical or even impossible. There has been a consistent push in the past few decades to solve such problems with parallel computing, meaning computations are distributed to multiple processors. In this lab, we explore the basic principles of parallel computing by introducing the cluster setup, standard parallel commands, and code designs that fully utilize available resources.*

Parallel Architectures

Imagine that you are in charge of constructing a very large building. You could, in theory, do all of the work yourself, but that would take so long that it simply would be impractical. Instead, you hire workers, who collectively can work on many parts of the building at once. Managing who does what task takes some effort, but the overall effect is that the building will be constructed many times faster than if only one person was working on it. This is the essential idea behind parallel computing.

A *serial* program is executed one line at a time in a single process. This is analogous to a single person creating a building. Since modern computers have multiple processor cores, serial programs only use a fraction of the computer's available resources. This is beneficial for smooth multitasking on a personal computer because multiple programs can run at once without interrupting each other.

For smaller computations, running serially is fine. However, some tasks are large enough that running serially could take days, months, or in some cases years. In these cases it is beneficial to devote all of a computer's resources (or the resources of many computers) to a single program by running it in *parallel*. Each processor can run part of the program on some of the inputs, and the results can be combined together afterwards. In theory, using N processors at once can allow the computation to run N times faster. Even though communication and coordination overhead prevents the improvement from being quite that good, the difference is still substantial.

A *computer cluster* or *supercomputer* is essentially a group of regular computers that share their processors and memory. There are several common architectures that are used for parallel computing, and each architecture has a different protocol for sharing memory, processors, and tasks between *computing nodes*, the different simultaneous processing areas. Each architecture offers unique advantages and disadvantages, but the general commands used with each are very similar.

In this lab, we will explore the usage and capabilities of parallel computing using Python's iPyParallel package. iPyParallel can be installed with either pip or conda:

```
$ pip install ipyparallel
```

```
$ conda install ipyparallel
```

The iPyParallel Architecture

There are three main parts of the iPyParallel architecture:

- *Client*: The main program that is being run.
- *Controller*: Receives directions from the client and distributes instructions and data to the computing nodes. Consists of a *hub* to manage communications and *schedulers* to assign processes to the engines.
- *Engines*: The individual processors. Each engine is like a separate Python terminal, each with its own namespace and computing resources.

Essentially, a Python program using iPyParallel creates a `Client` object connected to the cluster that allows it to send tasks to the cluster and retreive their results. The engines run the tasks, and the controller manages which engines run which tasks.

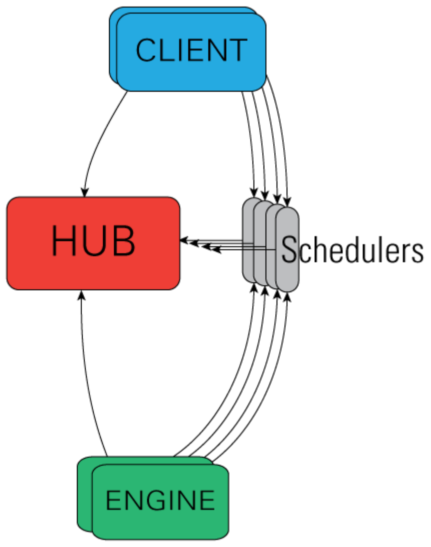

Figure 15.1: An outline of the iPyParallel architecture.

Setting up an iPyParallel Cluster

Before being able to use iPyParallel in a script or interpreter, it is necessarty to start an iPyParallel cluster. We demonstrate here how to use a single machine with multiple processor cores as a cluster. Establishing a cluster on multiple machines requires additional setup, which is detailed in the Additional Material section. The following commands initialize parts or all of a cluster when run in a terminal window:

Command	Description
`ipcontroller start`	Initialize a controller process.
`ipengine start`	Initialize an engine process.
`ipcluster start`	Initialize a controller process and several engines simultaneously.

Each of these processes can be stopped with a keyboard interrupt (`Ctrl+C`). By default, the controller uses JSON files in `UserDirectory/.ipython/profile-default/security/` to determine its settings. Once a controller is running, it acts like a server, listening connections from clients and engines. Engines will connect automatically to the controller when they start running. There is no limit to the number of engines that can be started in their own terminal windows and connected to the controller, but it is recommended to only use as many engines as there are cores to maximize efficiency.

Achtung!

The directory that the controller and engines are started from matters. To facilitate connections, navigate to the same folder as your source code before using `ipcontroller`, `ipengine`, or `ipcluster`. Otherwise, the engines may not connect to the controller or may not be able to find auxiliary code as directed by the client.

Starting a controller and engines in individual terminal windows with `ipcontroller` and `ipengine` is a little inconvenient, but having separate terminal windows for the engines allows the user to see individual errors in detail. It is also actually more convenient when starting a cluster of multiple computers. For now, we use `ipcluster` to get the entire cluster started quickly.

```
$ ipcluster start           # Assign an engine to each processor core.
$ ipcluster start --n 4     # Or, start a cluster with 4 engines.
```

Note

Jupyter notebooks also have a **Clusters** tab in which clusters can be initialized using an interactive GUI. To enable the tab, run the following command. This operation may require root permissions.

```
$ ipcluster nbextension enable
```

The iPyParallel Interface

Once a controller and its engines have been started and are connected, a cluster has successfully been established. The controller will then be able to distribute messages to each of the engines, which will compute with their own processor and memory space and return their results to the controller. The client uses the `ipyparallel` module to send instructions to the controller via a `Client` object.

```
>>> from ipyparallel import Client

>>> client = Client()       # Only works if a cluster is running.
>>> client.ids
[0, 1, 2, 3]                # Indicates that there are four engines running.
```

Once the client object has been created, it can be used to create one of two classes: a `DirectView` or a `LoadBalancedView`. These views allow for messages to be sent to collections of engines simultaneously. A `DirectView` allows for total control of task distribution while a `LoadBalancedView` automatically tries to spread out the tasks equally on all engines. The remainder of the lab will be focused on the `DirectView` class.

```
>>> dview = client[:]   # Group all engines into a DirectView.
>>> dview2 = client[:2] # Group engines 0,1, and 2 into a DirectView.
>>> dview2.targets      # See which engines are connected.
[0, 1, 2]
```

Since each engine has its own namespace, modules must be imported in every engine. There is more than one way to do this, but the easiest way is to use the `DirectView` object's `execute()` method, which accepts a string of code and executes it in each engine.

```
# Import NumPy in each engine.
>>> dview.execute("import numpy as np")
```

```
# Make sure to include client.close() after each function or else the test ↩
    driver will time out
client.close()
```

Problem 1. Write a function that initializes a `Client` object, creates a `DirectView` with all available engines, and imports `scipy.sparse` as `sparse` on all engines. Return the `DirectView`. Note: Make sure to include client.close() after EVERY function or else the test driver will time out.

Managing Engine Namespaces

Before continuing, set the `DirectView` you are using to use blocking:

```
>>> dview.block = True
```

This affects the way that functions called using the `DirectView` return their values. Using blocking makes the process simpler, so we will use it initially. What blocking is will be explained later.

Push and Pull

The `push()` and `pull()` methods of a `DirectView` object manage variable values in the engines. Use `push()` to set variable values and `pull()` to get variables. Each method also has a shortcut via indexing.

```
# Initialize the variables 'a' and 'b' on each engine.
>>> dview.push({'a':10, 'b':5})         # OR dview['a'] = 10; dview['b'] = 5
[None, None, None, None]                # Output from each engine

# Check the value of 'a' on each engine.
>>> dview.pull('a')                     # OR dview['a']
[10, 10, 10, 10]

# Put a new variable 'c' only on engines 0 and 2.
>>> dview.push({'c':12}, targets=[0, 2])
[None, None]
```

Problem 2. Write a function `variables(dx)` that accepts a dictionary of variables. Create a `Client` object and a `DirectView` and distribute the variables. Pull the variables back and make sure they haven't changed.

Scatter and Gather

Parallelization almost always involves splitting up collections and sending different pieces to each engine for processing. The process is called *scattering* and is usually used for dividing up arrays or lists. The inverse process of pasting a collection back together is called *gathering* and is usually used on the results of processing. This method of distributing a dataset and collecting the results is common for processing large data sets using parallelization.

```
>>> import numpy as np

# Send parts of an array of 8 elements to each of the 4 engines.
>>> x = np.arange(1, 9)
>>> dview.scatter("nums", x)
>>> dview["nums"]
[array([1, 2]), array([3, 4]), array([5, 6]), array([7, 8])]

# Scatter the array to only the first two engines.
>>> dview.scatter("nums_big", x, targets=[0,1])
>>> dview.pull("nums_big", targets=[0,1])
[array([1, 2, 3, 4]), array([5, 6, 7, 8])]

# Gather the array again.
>>> dview.gather("nums")
array([1, 2, 3, 4, 5, 6, 7, 8])
```

```
>>> dview.gather("nums_big", targets=[0,1])
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Executing Code on Engines

Execute

The `execute()` method is the simplest way to run commands on parallel engines. It accepts a string of code (with exact syntax) to be executed. Though simple, this method works well for small tasks.

```
# 'nums' is the scattered version of np.arange(1, 9).
>>> dview.execute("c = np.sum(nums)")   # Sum each scattered component.
<AsyncResult: execute:finished>
>>> dview['c']
[3, 7, 11, 15]
```

Apply

The `apply()` method accepts a function and arguments to plug into it, and distributes them to the engines. Unlike `execute()`, `apply()` returns the output from the engines directly.

```
>>> dview.apply(lambda x: x**2, 3)
[9, 9, 9, 9]
>>> dview.apply(lambda x,y: 2*x + 3*y, 5, 2)
[16, 16, 16, 16]
```

Note that the engines can access their local variables in either of the execution methods.

Map

The built-in `map()` function applies a function to each element of an iterable. The iPyParallel equivalent, the `map()` method of the `DirectView` class, combines `apply()` with `scatter()` and `gather()`. Simply put, it accepts a dataset, splits it between the engines, executes a function on the given elements, returns the results, and combines them into one object.

```
>>> num_list = [1, 2, 3, 4, 5, 6, 7, 8]
>>> def triple(x):                  # Map a function with a single input.
...     return 3*x
...
>>> dview.map(triple, num_list)
[3, 6, 9, 12, 15, 18, 21, 24]

>>> def add_three(x, y, z):         # Map a function with multiple inputs.
...     return x+y+z
...
>>> x_list = [1, 2, 3, 4]
>>> y_list = [2, 3, 4, 5]
>>> z_list = [3, 4, 5, 6]
>>> dview.map(add_three, x_list, y_list, z_list)
[6, 9, 12, 15]
```

Blocking vs. Non-Blocking

Parallel commands can be implemented two ways. The difference is subtle but extremely important.

- *Blocking*: The main program sends tasks to the controller, and then waits for all of the engines to finish their tasks before continuing (the controller "blocks" the program's execution). This mode is usually best for problems in which each node is performing the same task.
- *Non-Blocking*: The main program sends tasks to the controller, and then continues without waiting for responses. Instead of the results, functions return an `AsyncResult` object that can be used to check the execution status and eventually retrieve the actual result.

Whether a function uses blocking is determined by default by the `block` attribute of the `DirectView` The execution methods `execute()`, `apply()`, and `map()`, as well as `push()`, `pull()`, `scatter()`, and `gather()`, each have a keyword argument `block` that can instead be used to specify whether or not to using blocking. Alternatively, the methods `apply_sync()` and `map_sync()` always use blocking, and `apply_async()` and `map_async()` always use non-blocking.

```
>>> f = lambda n: np.sum(np.random.random(n))

# Evaluate f(n) for n=0,1,...,999 with blocking.
>>> %time block_results = [dview.apply_sync(f, n) for n in range(1000)]
CPU times: user 9.64 s, sys: 879 ms, total: 10.5 s
Wall time: 13.9 s

# Evaluate f(n) for n=0,1,...,999 with non-blocking.
>>> %time responses = [dview.apply_async(f, n) for n in range(1000)]
CPU times: user 4.19 s, sys: 294 ms, total: 4.48 s
Wall time: 7.08 s

# The non-blocking method is faster, but we still need to get its results.
# Both methods produced a list, although the contents are different
>>> block_results[10]  # This list holds actual result values from each engine.
[3.833061790352166,
4.8943956129713335,
4.268791758626886,
4.73533677711277]

>>> responses[10]           # This list holds AsyncResult objects.
<AsyncResult: <lambda>:finished>
# We can get the actual results by using the get() method of each AsyncResult
>>> %time nonblock_results = [r.get() for r in responses]
CPU times: user 3.52 ms, sys: 11 mms, total: 3.53 ms
Wall time: 3.54 ms          # Getting the responses takes little time.
```

```
>>> nonblock_results[10]    # This list also holds actual result values
[5.652608204341693,
4.984164642641558,
4.686288406810953,
5.275735658763963]
```

When non-blocking is used, commands can be continuously sent to engines before they have finished their previous task. This allows them to begin their next task without waiting to send their calculated answer and receive a new command. However, this requires a design that incorporates checkpoints to retrieve answers and enough memory to store response objects.

Class Method	Description
`wait(timeout)`	Wait until the result is available or until `timeout` seconds pass.
`ready()`	Return whether the call has completed.
`successful()`	Return whether the call completed without raising an exception. Will raise `AssertionError` if the result is not ready.
`get(timeout)`	Return the result when it arrives. If `timeout` is not `None` and the result does not arrive within `timeout` seconds then `TimeoutError` is raised.

Table 15.1: All information from `https://ipyparallel.readthedocs.io/en/latest/details.html#AsyncResult`.

Table 15.1 details the methods of the `AsyncResult` object.

There are additional magic methods supplied by `iPyParallel` that make some of these operations easier. These methods are explained in the Additional Material section. More information on `iPyParallel` architecture, interface, and methods can also be found at `https://ipyparallel.readthedocs.io/en/latest/index.html`.

Problem 3. Write a function that accepts an integer n. Instruct each engine to make n draws from the standard normal distribution, then hand back the mean, minimum, and maximum draws to the client. Return the results in three lists.

If you have four engines running, your results should resemble the following:

```
>>> means, mins, maxs = problem3(1000000)
>>> means
[0.0031776784, -0.0058112042, 0.0012574772, -0.0059655951]
>>> mins
[-4.1508589, -4.3848019, -4.1313324, -4.2826519]
>>> maxs
[4.0388107, 4.3664958, 4.2060184, 4.3391623]
```

Problem 4. Use your function from Problem 3 to compare serial and parallel execution times. For $n = 1000000, 5000000, 10000000, 15000000,$

1. Time how long it takes to run your function.
2. Time how long it takes to do the same process serially. Make n draws and then calculate and record the statistics, but use a `for` loop with N iterations, where N is the number of engines running.

Plot the execution times against n. You should notice an increase in efficiency in the parallel version as the problem size increases.

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 feasible to solve through parallel computing because solving them serially would take too long. With some of these problems, even the parallel solution could take years. Some brute-force algorithms, like those used to crack simple encryptions, are examples of this type of problem.

The problems mentioned above are well suited to parallel computing because they can be manipulated in such a way 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 a 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 be parallelized, but there is not always a significant enough improvement in run time to make it 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.

Problem 5. The *trapeziod rule* is a simple technique for numerical integration:

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

where $a = x_1 < x_2 < \ldots < x_N = b$ and $h = x_{n+1} - x_n$ for each n. See Figure 15.2.

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

Write a function that accepts a function handle to integrate, bounds of integration, and the number of points to use for the approximation. Parallelize the trapezoid rule in order to estimate the integral of f. That is, evenly divide the points among all available processors and run the trapezoid 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.

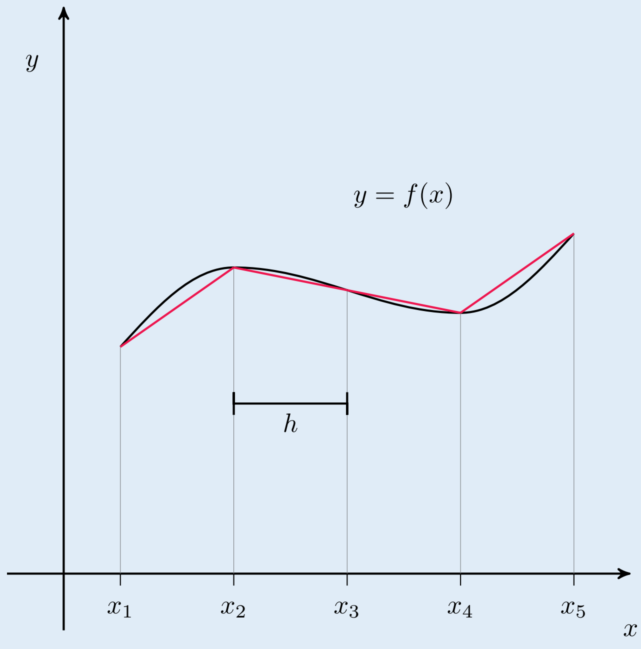

Figure 15.2: A depiction of the trapezoid rule with uniform partitioning.

Intercommunication

The phrase *parallel computing* refers to designing an architecture and code that makes the best use of computing resources for a problem. Occasionally, this will require nodes to be interdependent on each other for previous results. This contributes to a slower result because it requires a great deal of communication latency, but is sometimes the only method to parallelize a function. Although important, the ability to effectively communicate between engines has not been added to iPyParallel. It is, however, possible in an MPI framework and will be covered in the MPI lab.

Additional Material

Clusters of Multiple Machines

Though setting up a computing cluster with `iPyParallel` on multiple machines is similar to a cluster on a single computer, there are a couple of extra considerations to make. The majority of these considerations have to do with the network setup of your machines, which is unique to each situation. However, some basic steps have been taken from `https://ipyparallel.readthedocs.io/en/latest/process.html` and are outlined below.

SSH Connection

When using engines and controllers that are on separate machines, their communication will most likely be using an SSH tunnel. This *Secure Shell* allows messages to be passed over the network.

In order to enable this, an SSH user and IP address must be established when starting the controller. An example of this follows.

```
$ ipcontroller --ip=<controller IP> --user=<user of controller> --enginessh=<↩
   user of controller>@<controller IP>
```

Engines started on remote machines then follow a similar format.

```
$ ipengine --location=<controller IP> --ssh=<user of controller>@<controller IP↩
   >
```

Another way of affecting this is to alter the configuration file in `UserDirectory/.ipython/profile-default/security/ipcontroller-engine.json`. This can be modified to contain the controller IP address and SSH information.

All of this is dependent on the network feasibility of SSH connections. If there are a great deal of remote engines, this method will also require the SSH password to be entered many times. In order to avoid this, the use of SSH Keys from computer to computer is recommended.

Magic Methods & Decorators

To be more easily usable, the `iPyParallel` module has incorporated a few magic methods and decorators for use in an interactive iPython or Python terminal.

Magic Methods

The `iPyParallel` module has a few magic methods that are very useful for quick commands in iPython or in a Jupyter Notebook. The most important are as follows. Additional methods are found at `https://ipyparallel.readthedocs.io/en/latest/magics.html`.

%px - This magic method runs the corresponding Python command on the engines specified in `dview.targets`.

%autopx - This magic method enables a boolean that runs any code run on every engine until `%autopx` is run again.

Examples of these magic methods with a client and four engines are as follows.

```
# %px
In [4]: with dview.sync_imports():
   ...:     import numpy
   ...:
importing numpy on engine(s)
In [5]: \%px a = numpy.random.random(2)

In [6]: dview['a']
Out[6]:
[array([ 0.30390162,  0.14667075]),
 array([ 0.95797678,  0.59487915]),
 array([ 0.20123566,  0.57919846]),
 array([ 0.87991814,  0.31579495])]

 # %autopx
In [7]: %autopx
%autopx enabled
In [8]: max_draw = numpy.max(a)

In [9]: print('Max_Draw: {}'.format(max_draw))
[stdout:0] Max_Draw: 0.30390161663280246
[stdout:1] Max_Draw: 0.957976784975849
[stdout:2] Max_Draw: 0.5791984571339429
[stdout:3] Max_Draw: 0.8799181411958089

In [10]: %autopx
%autopx disabled
```

Decorators

The `iPyParallel` module also has a few decorators that are very useful for quick commands. The two most important are as follows:

@remote - This decorator creates methods on the remote engines.

@parallel - This decorator creates methods on remote engines that break up element wise operations and recombine results.

Examples of these decorators are as follows.

```
# Remote decorator
>>> @dview.remote(block=True)
>>> def plusone():
...     return a+1
>>> dview['a'] = 5
>>> plusone()
 [6, 6, 6, 6,]
```

```
# Parallel decorator
>>> import numpy as np

>>> @dview.parallel(block=True)
>>> def combine(A,B):
...     return A+B
>>> ex1 = np.random.random((3,3))
>>> ex2 = np.random.random((3,3))
>>> print(ex1+ex2)
 [[ 0.87361929  1.41110357  0.77616724]
 [ 1.32206426  1.48864976  1.07324298]
 [ 0.6510846   0.45323311  0.71139272]]
>>> print(combine(ex1,ex2))
 [[ 0.87361929  1.41110357  0.77616724]
 [ 1.32206426  1.48864976  1.07324298]
 [ 0.6510846   0.45323311  0.71139272]]
```