


7 SEPTEMBER 2019 / DATA SCIENCE

Multiprocessing vs. Threading in Python: What Every Data Scientist Needs to Know



Sooner or later, every data science project faces an inevitable challenge: speed. Working with larger data sets leads to slower processing thereof, so you'll eventually have to think about optimizing your algorithm's run time. As most of you already know, parallelization is a necessary step of this optimization. Python offers two built-in libraries for parallelization: multiprocessing and threading. In this article, we'll explore how data scientists can go about choosing between the two and which factors should be kept in mind while doing so.

Parallel Computing and Data Science

As you all know, data science is the science of dealing with large amounts of data and extracting useful insights from them. More often than not, the operations we perform on the data are easily parallelizable, meaning that different processing agents can run the operation on the data one piece at a time, then combine the results at the end to get the complete result.

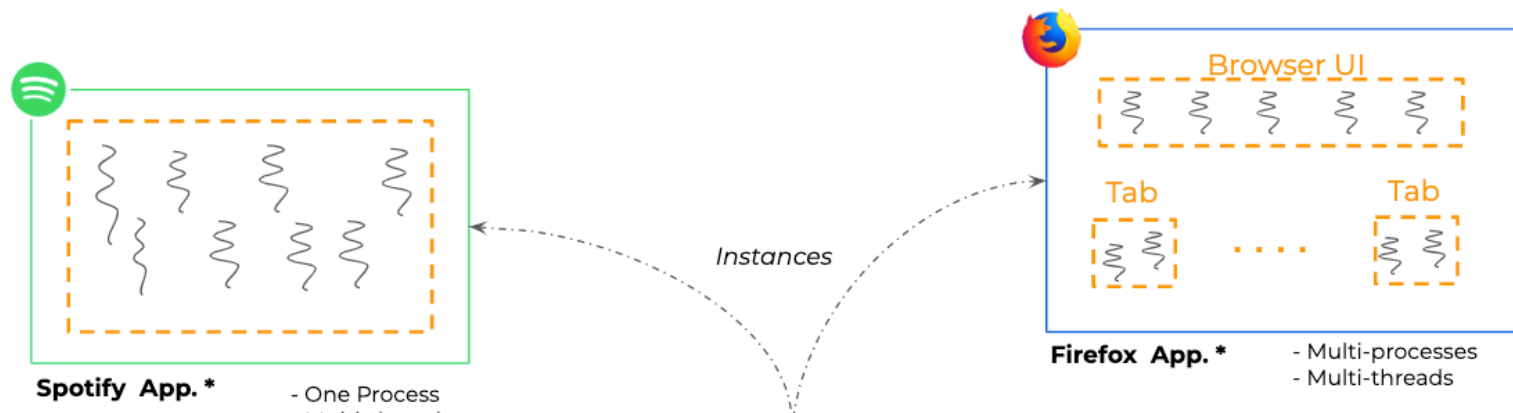
To better visualize parallelizability, let's consider a real world analogy. Suppose you need to clean three rooms in your home. You can either do it all by yourself, cleaning the rooms one after the other, or you can ask your two siblings to help you out, with each of you cleaning a single room. In the latter approach, each of you are working parallelly on a part of the whole task, thus reducing the total time required to complete it. This is parallelizability in action.

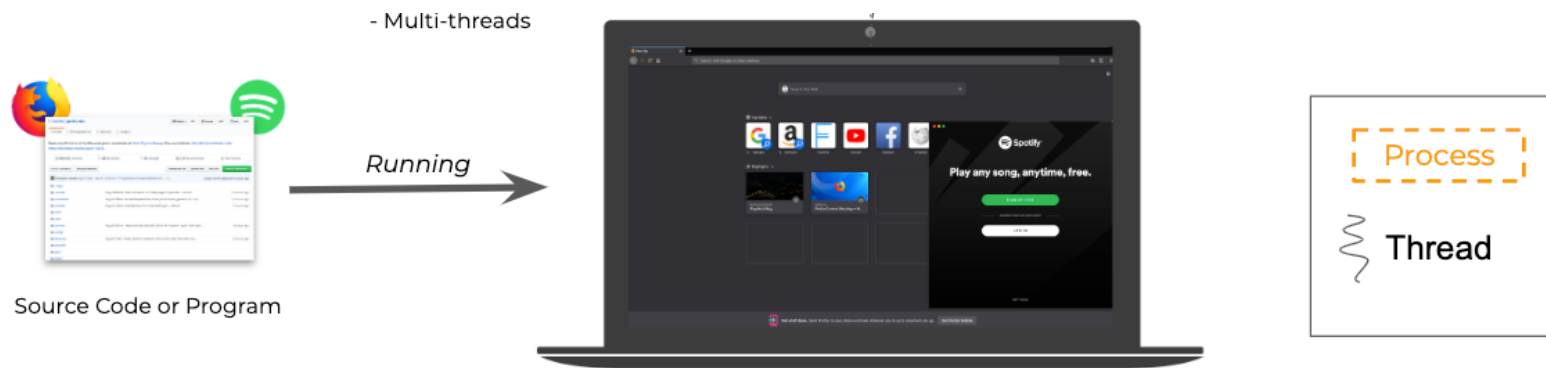
Parallel processing can be achieved in Python in two different ways: multiprocessing and threading.

Multiprocessing and Threading: Theory

Fundamentally, multiprocessing and threading are two ways to achieve parallel computing, using processes and threads, respectively, as the processing agents. To understand how these work, we have to clarify what processes and threads are.

Programs, Apps, Processes & Threads





** this image may not reflect the reality for the show-cased apps*

Process

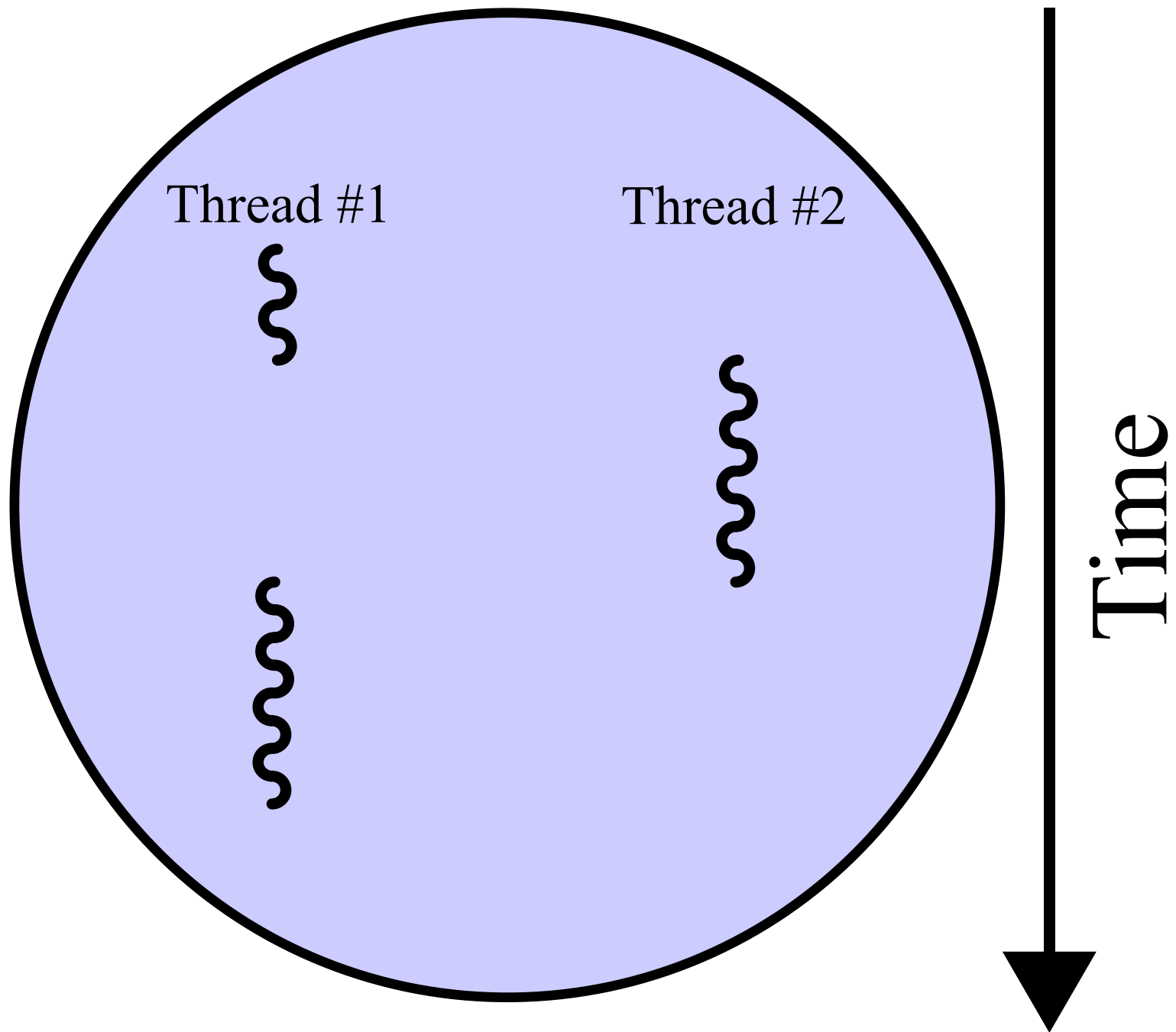
A process is an instance of a computer program being executed. Each process has its own memory space it uses to store the instructions being run, as well as any data it needs to store and access to execute.

Threads

Threads are components of a process, which can run parallelly. There can be multiple threads in a process, and they share the same memory space, i.e. the memory space of the parent process.

This would mean the code to be executed as well as all the variables declared in the program would be shared by all threads.

Process



Process and Threads, By I, [Cburnett](#), [CC BY-SA 3.0](#), [Link](#)

For example, let us consider the programs being run on your computer right now. You're probably reading this article in a browser, which probably has multiple tabs open. You might also be listening to music through the Spotify desktop app at the same time. The browser and the Spotify application are different processes; each of them can use multiple processes or threads to achieve parallelism. Different tabs in your browser might be run in different threads. Spotify can play music in one thread, download music from the internet in another, and use a third to display the GUI. This would be called multithreading. The same can be done with multiprocessing—multiple processes—too. In fact, most modern browsers like Chrome and Firefox use multiprocessing, not multithreading, to handle multiple tabs.

Technical details

- All the threads of a process live in the same memory space, whereas processes have their separate memory space.
- Threads are more lightweight and have lower overhead compared to processes. Spawning processes is a bit slower than spawning threads.
- Sharing objects between threads is easier, as they share the same memory space. To achieve the same between process, we have to use some kind of IPC (inter-process

communication) model, typically provided by the OS.

Pitfalls of Parallel Computing

Introducing parallelism to a program is not always a positive-sum game; there are some pitfalls to be aware of. The most important ones are as follows.

- **Race Condition:** As we already discussed, threads have a shared memory space, and therefore they can have access to shared variables. A race condition occurs when multiple threads try to change the same variable simultaneously. The thread scheduler can arbitrarily swap between threads, so we have no way of knowing the order in which the threads will try to change the data. This can result in incorrect behavior in either of the threads, particularly if the threads decide to do something based on the value of the variable. To prevent this from happening, a mutual exclusion (or mutex) *lock* can be placed around the piece of the code that modifies the variable so that only one thread can write to the variable at a time.
- **Starvation:** Starvation occurs when a thread is denied access to a particular resource for longer periods of time, and as a result, the overall program slows down. This can happen as an unintended side effect of a poorly designed thread-scheduling algorithm.
- **Deadlock:** Overusing mutex locks also has a downside - it can introduce deadlocks in the program. A deadlock is a state when a thread is waiting for another thread to release a lock, but that other thread needs a resource to finish that the first thread is holding onto.

This way, both of the threads come to a standstill and the program halts. Deadlock can be thought of as an extreme case of starvation. To avoid this, we have to be careful not to introduce too many locks that are interdependent.

- **Livelock** : Livelock is when threads keep running in a loop but don't make any progress. This also arises out of poor design and improper use of mutex locks.

Multiprocessing and Threading in Python

The Global Interpreter Lock

When it comes to Python, there are some oddities to keep in mind. We know that threads share the same memory space, so special precautions must be taken so that two threads don't write to the same memory location. The CPython interpreter handles this using a mechanism called `GIL`, or the Global Interpreter Lock.

From the Python [wiki](#):

*In CPython, the **global interpreter lock**, or **GIL**, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe.*

Check the slides [here](#) for a more detailed look at the Python GIL.

The GIL gets its job done, but at a cost. It effectively serializes the instructions at the interpreter level. How this works is as follows: for any thread to perform any function, it must acquire a global lock. Only a single thread can acquire that lock at a time, which means **the interpreter ultimately runs the instructions serially**. This design makes memory management thread-safe, but as a consequence, it can't utilize multiple CPU cores at all. In single-core CPUs, which is what the designers had in mind while developing CPython, it's not much of a problem. But this global lock ends up being a bottleneck if you're using multi-core CPUs.

This bottleneck, however, becomes irrelevant if your program has a more severe bottleneck elsewhere, for example in network, IO, or user interaction. In those cases, threading is an entirely effective method of parallelization. But for programs that are CPU bound, threading ends up making the program slower. Let's explore this with some example use cases.

Use Cases for Threading

GUI programs use threading all the time to make applications responsive. For example, in a text editing program, one thread can take care of recording the user inputs, another can be responsible for displaying the text, a third can do spell-checking, and so on. Here, the program has to wait for user interaction, which is the biggest bottleneck. Using multiprocessing won't make the program any faster.

Another use case for threading is programs that are IO bound or network bound, such as [web-scrapers](#). In this case, multiple threads can take care of scraping multiple webpages in parallel. The threads have to download the webpages from the Internet, and that will be the biggest bottleneck, so threading is a perfect solution here. Web servers, being network bound, work similarly; with them, multiprocessing doesn't have any edge over threading. Another relevant example is [Tensorflow](#), which uses a thread pool to transform data in parallel.

Use Cases for Multiprocessing

Multiprocessing outshines threading in cases where the program is CPU intensive and doesn't have to do any IO or user interaction. For example, any program that just crunches numbers will see a massive speedup from multiprocessing; in fact, threading will probably slow it down. An interesting real world example is [Pytorch Dataloader](#), which uses multiple subprocesses to load the data into GPU.

Parallelization in Python, in Action

Python offers two libraries - `multiprocessing` and `threading` - for the eponymous parallelization methods. Despite the fundamental difference between them, the two libraries offer a very similar API (as of Python 3.7). Let's see them in action.

```
import threading
import random
```

```

import random
from functools import reduce

def func(number):
    random_list = random.sample(range(1000000), number)
    return reduce(lambda x, y: x*y, random_list)

number = 50000
thread1 = threading.Thread(target=func, args=(number,))
thread2 = threading.Thread(target=func, args=(number,))

thread1.start()

thread2.start()

thread1.join()
thread2.join()

```

You can see that I've created a function `func` that creates a list of random numbers and then multiplies all the elements of it sequentially. This can be a fairly heavy process if the number of items is large enough, say 50k or 100k.

Then, I've created two threads that will execute the same function. The thread objects have a `start` method that starts the thread asynchronously. If we want to wait for them to terminate and return, we have to call the `join` method, and that's what we have done above.

As you can see, the API for spinning up a new thread to a task in the background is pretty

straightforward. What's great is that the API for multiprocessing is almost the exact same as well; let's check it out.

```
import multiprocessing
import random
from functools import reduce

def func(number):
    random_list = random.sample(range(1000000), number)
    return reduce(lambda x, y: x*y, random_list)

number = 50000
process1 = multiprocessing.Process(target=func, args=(number,))
process2 = multiprocessing.Process(target=func, args=(number,))

process1.start()
process2.start()

process1.join()
process2.join()
```

There it is—just swap `threading.Thread` with `multiprocessing.Process` and you have the exact same program implemented using multiprocessing.

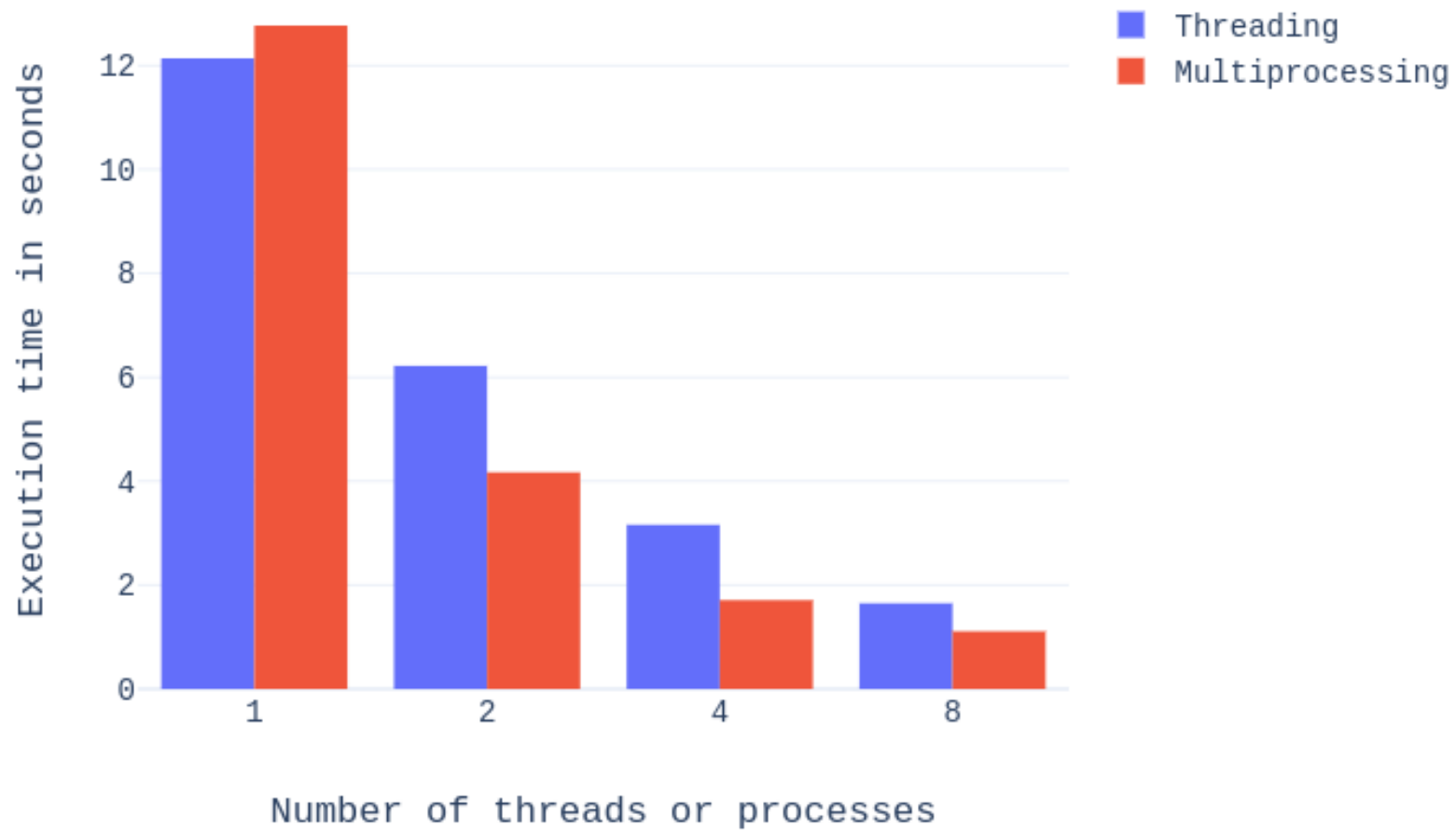
There's obviously a lot more you can do with this, but that's not within the scope of this article, so we won't go into it here. Check out the docs [here](#) and [here](#) if you're interested in learning more.

Benchmarks

Now that we have an idea of how the code implementing parallelization looks like, let's get back to the performance issues. As we've noted before, threading is not suitable for CPU bound tasks; in those cases it ends up being a bottleneck. We can validate this using some simple benchmarks.

Firstly, let's see how threading compares against multiprocessing for the code sample I showed you above. Keep in mind that this task does not involve any kind of IO, so it's a pure CPU bound task.

Threading vs Multiprocessing for CPU Bound Tasks



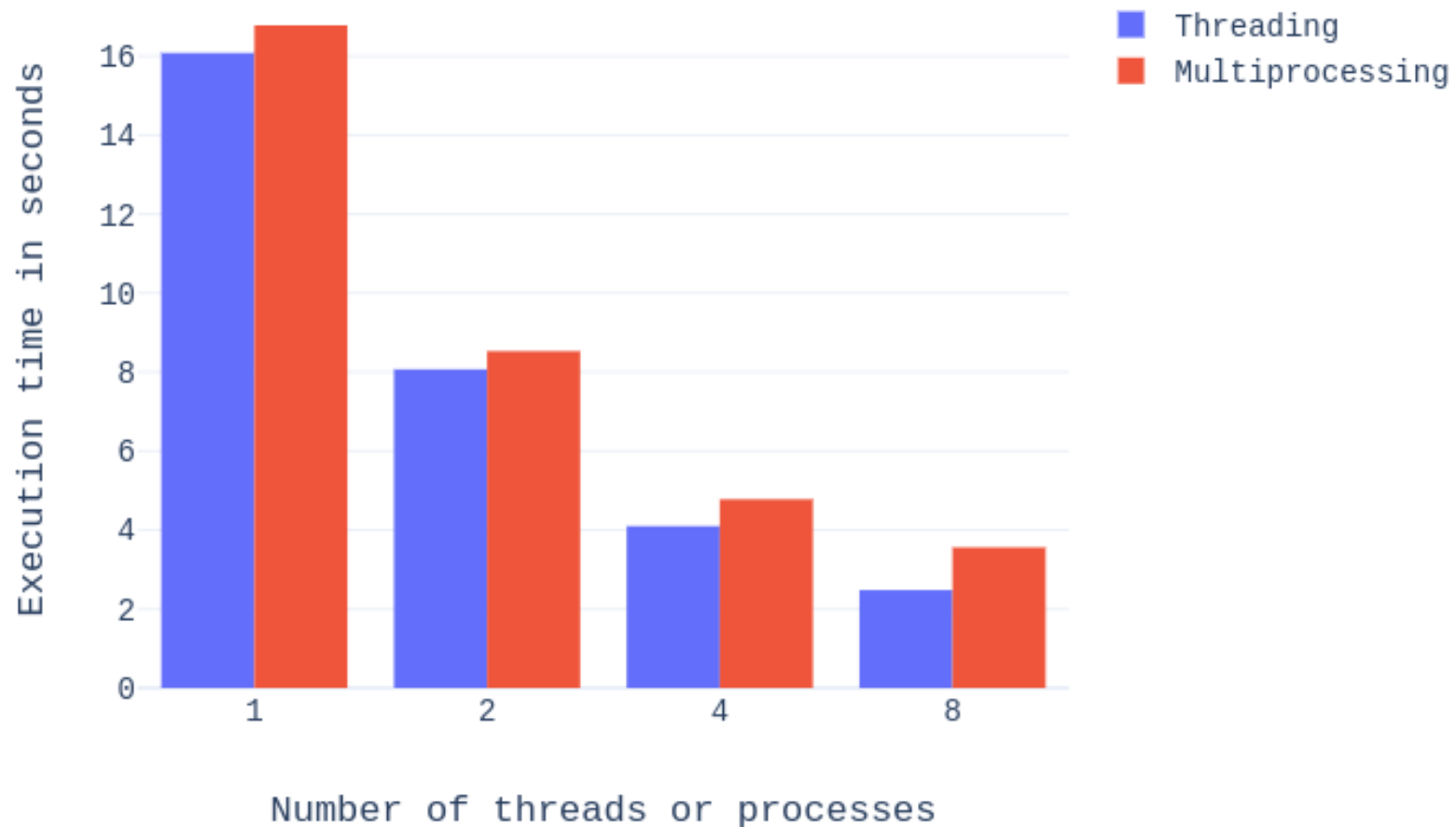
And let's see a similar benchmark for an IO bound task. For example, the following function —

```
import requests

def func(number):
    url = 'http://example.com/'
    for i in range(number):
        response = requests.get(url)
        with open('example.com.txt', 'w') as output:
            output.write(response.text)
```

The function is simply fetching a webpage and saving that to a local file, multiple times in a loop. Useless but straightforward and thus a good fit for demonstration. Let's look at the benchmark.

Threading vs Multiprocessing for IO Bound Tasks



Now there are a few things to note from these two charts:

- In both cases, a single process took more execution time than a single thread. Evidently, processes have more overhead than threads.

processes have more overhead than threads.

- For the CPU bound task, multiple processes perform way better than multiple threads. However, this difference becomes slightly less prominent when we're using 8x parallelization. As the processor in my laptop is quad-core, up to four processes can use the multiple cores effectively. So when I'm using more processes, it doesn't scale that well. But still, it outperforms threading by a lot because threading can't utilize the multiple cores at all.
- For the IO-bound task, the bottleneck is not CPU. So the usual limitations due to GIL don't apply here, and multiprocessing doesn't have an advantage. Not only that, the light overhead of threads actually makes them faster than multiprocessing, and threading ends up outperforming multiprocessing consistently.

Differences, Merits and Drawbacks

- Threads run in the same memory space; processes have separate memory.
- Following from the previous point: sharing objects between threads is easier, but the flip side of the coin is that you have to take extra measure for object synchronization to make sure that two threads don't write to the same object at the same time and that a race condition does not occur.
- Because of the added programming overhead of object synchronization, multi-threaded programming is more bug-prone. On the other hand, multi-processes programming is

programming is more complex than the other two, but process programming is easy to get right.

- Threads have a lower overhead compared to processes; spawning processes take more time than threads.
- Due to limitations put in place by the GIL in Python, threads can't achieve *true* parallelism utilizing multiple CPU cores. Multiprocessing does not have any such restrictions.
- Process scheduling is handled by the OS, whereas thread scheduling is done by the Python interpreter.
- Child processes are interruptible and killable, whereas child threads are not. You have to wait for the threads to terminate or `join`.

From all this discussion, we can conclude the following —

- Threading should be used for programs involving IO or user interaction.
- Multiprocessing should be used for CPU bound, computation-intensive programs.

From the Perspective of a Data Scientist

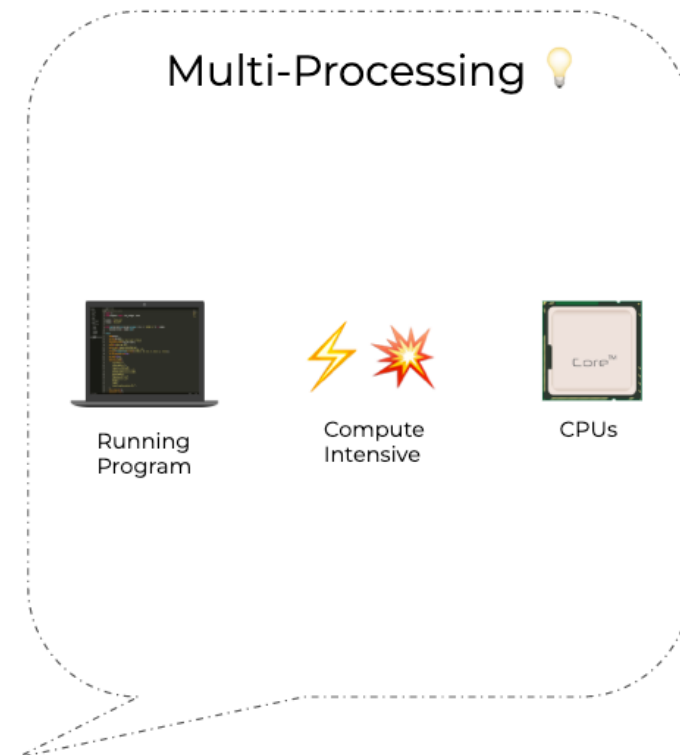
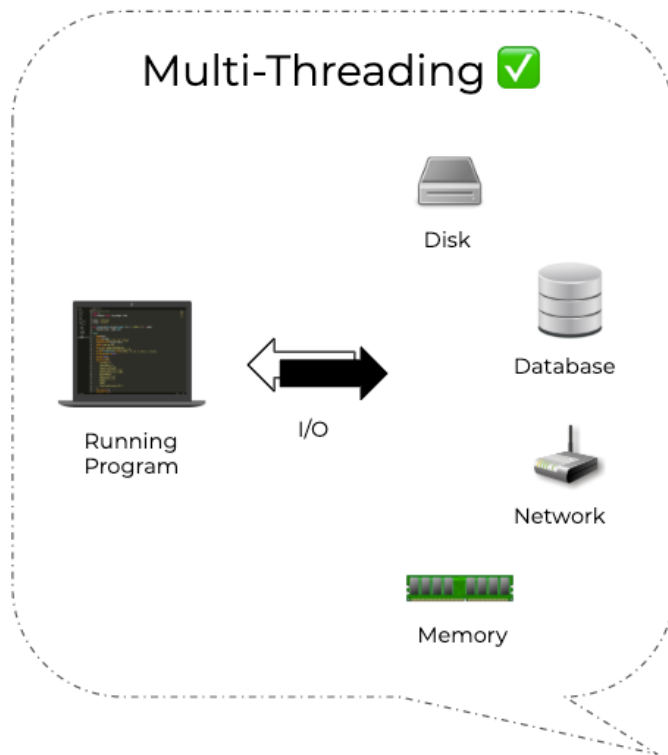
A typical data processing pipeline can be divided into the following steps:

1. Reading raw data and storing into main memory or GPU
2. Doing computation, using either CPU or GPU
3. Storing the mined information in a database or disk.

Let's explore how we could introduce parallelism in these tasks so that they can be sped up.

Step 1 involves reading data from disk, so clearly disk IO is going to be the bottleneck for this step. As we've discussed, threads are the best option for parallelizing this kind of operation. Similarly, step 3 is also an ideal candidate for the introduction of threading.

However, step 2 consists of computations that involve the CPU or a GPU. If it's a CPU based task, using threading will be of no use; instead, we have to go for multiprocessing. Only then we'll be able to exploit the multiple cores of the CPU and achieve parallelism. If it's a GPU based task, since GPU already implements a massively parallelized architecture at the hardware level, using the correct interface (libraries and drivers) to interact with the GPU should take care of the rest.



Now you may be thinking, “My data pipeline looks a bit different to this; I have some tasks that

don't really fit into this general framework." Still, you should be able to observe the methodology used here to decide between threading and multiprocessing. The factors you should consider are:

- Whether your task has any form of IO
- Whether IO is the bottleneck of your program
- Whether your task depends upon a large amount of computation by the CPU

With these factors in mind, together with the takeaways above, you should be able to make the decision. Also, keep in mind that you don't have to use a single form of parallelism throughout your program. You should use one or the other for different parts of your program, whichever is suitable for that particular part.

Now we'll look at two example scenarios a data scientist might face and how you can use parallel computing to speed them up.

Scenario: Downloading Emails

Let's say you want to analyze all the emails in the inbox of your own home-grown startup and understand the trends: who are the most frequent senders, what are the most common keywords appearing in the emails, which day of the week or which hour of the day do you receive most emails, and so on. The first step of this project would be, of course, downloading the emails to

your computer.

At first, let's do it sequentially without using any parallelization. The code to use is below and it should be pretty self-explanatory. There is a function `download_emails` which takes a list of email ids as input and downloads them sequentially. This calls this function with a list of ids of 100 email at once.

```
import imaplib
import time

IMAP_SERVER = 'imap.gmail.com'
USERNAME = 'username@gmail.com'
PASSWORD = 'password'

def download_emails(ids):
    client = imaplib.IMAP4_SSL(IMAP_SERVER)
    client.login(USERNAME, PASSWORD)

    client.select()
    for i in ids:
        print(f'Downloading mail id: {i.decode()}')
        _, data = client.fetch(i, '(RFC822)')
        with open(f'emails/{i.decode()}.eml', 'wb') as f:
            f.write(data[0][1])
    client.close()
    print(f'Downloaded {len(ids)} mails!')

start = time.time()
```



```
client = imaplib.IMAP4_SSL(IMAP_SERVER)
client.login(USERNAME, PASSWORD)
client.select()
_, ids = client.search(None, 'ALL')
ids = ids[0].split()
ids = ids[:100]
client.close()

download_emails(ids)
print('Time:', time.time() - start)
```

Time taken :: 35.65300488471985 seconds.

Now let's introduce some parallelizability into this task to speed things up. Before we dive into writing the code, we have to decide between threading and multiprocessing. As you've learned so far, threads are the best option when it comes to tasks that have some IO as the bottleneck. The

task at hand obviously belongs to this category, as it is accessing an IMAP server over the internet. So we'll be going with `threading`.

Much of the code we're going to use is going to be the same as the one we used in the sequential case. The only difference is that we will split the list of 100 email ids into 10 smaller chunks, each chunk containing 10 ids, then create 10 threads and call the `download_emails` function with a different chunk from each of them. I'm using the `concurrent.futures.ThreadPoolExecutor` class from the Python standard library for threading.

```

import imaplib
import time
from concurrent.futures import ThreadPoolExecutor

IMAP_SERVER = 'imap.gmail.com'
USERNAME = 'username@gmail.com'
PASSWORD = 'password'

def download_emails(ids):
    client = imaplib.IMAP4_SSL(IMAP_SERVER)
    client.login(USERNAME, PASSWORD)
    client.select()
    for i in ids:
        print(f'Downloading mail id: {i.decode()}')
        _, data = client.fetch(i, '(RFC822)')
        with open(f'emails/{i.decode()}.eml', 'wb') as f:
            f.write(data[0][1])
    client.close()
    print(f'Downloaded {len(ids)} mails!')

start = time.time()

client = imaplib.IMAP4_SSL(IMAP_SERVER)
client.login(USERNAME, PASSWORD)
client.select()
_, ids = client.search(None, 'ALL')
ids = ids[0].split()
ids = ids[:100]
client.close()

```

```
number_of_chunks = 10
chunk_size = 10
executor = ThreadPoolExecutor(max_workers=number_of_chunks)
futures = []
for i in range(number_of_chunks):
    chunk = ids[i*chunk_size:(i+1)*chunk_size]
    futures.append(executor.submit(download_emails, chunk))

for future in concurrent.futures.as_completed(futures):
    pass
print('Time:', time.time() - start)
```

Time taken :: 9.841094255447388 seconds.

As you can see, threading, sped it up considerably.

Scenario: Classification Using Scikit-Learn

Let's say you have a classification problem, and you want to use a *random forest* classifier for this. As it's a standard and well-known machine learning algorithm, let's not [reinvent the wheel](#) and just use `sklearn.ensemble.RandomForestClassifier`.

The below code serves demonstration purposes. I have created a classification dataset using the helper function `sklearn.datasets.make_classification`, then trained a `RandomForestClassifier` on that. Also, I'm timing the part of the code that does the core work of fitting the model.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
import time

X, y = datasets.make_classification(n_samples=10000, n_features=50, n_informative=20, n_classes=2)

start = time.time()
model = RandomForestClassifier(n_estimators=500)
model.fit(X, y)
print('Time:', time.time()-start)
```

Time taken :: 34.17733192443848 seconds.

Now we'll look into how we can reduce the running time of this algorithm. We know that this algorithm can be parallelized to some extent, but what kind of parallelization would be suitable? It does not have any IO bottleneck; on the contrary, it's a very CPU intensive task. So multiprocessing would be the logical choice.

Fortunately, `sklearn` has already implemented multiprocessing into this algorithm and we won't have to write it from scratch. As you can see in the code below, we just have to provide a parameter `n_jobs`—the number of processes it should use—to enable multiprocessing.

```
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn import datasets
import time

X, y = datasets.make_classification(n_samples=10000, n_features=50, n_informative=20, n_classes

start = time.time()
model = RandomForestClassifier(n_estimators=500, n_jobs=4)
model.fit(X, y)
print('Time:', time.time()-start)
```

Time taken :: 14.576200723648071 seconds.

As expected, multiprocessing made it quite a bit faster.

Conclusion

Most if not all data science projects will see a massive increase in speed with parallel computing. In fact, many of the popular data science libraries already have parallelism built into them, *you just have to enable it*. So before trying to implement it on your own, look through the documentation of the library you're using and check if it supports parallelism (by the way, I definitely recommend you to check out [dask](#)). In case it doesn't, this article should assist you in implementing it on your own.

Ready to build, train, and deploy AI?

Get started with FloydHub's collaborative AI platform for free

Try FloydHub for free

About the Author

Sumit is a computer enthusiast who started programming at an early age; he's currently finishing his master's degree in computer science at IIT Delhi. Whenever he isn't programming, you can probably find him either reading philosophy, playing the guitar, taking photos, or blogging. You can connect with Sumit on [Twitter](#), [LinkedIn](#), [Github](#), and [his website](#).

Subscribe to FloydHub Blog

Get the latest posts delivered right to your inbox

Subscribe



Sumit Ghosh

Sumit is a computer enthusiast who started programming at an early age; he's currently finishing his master's degree in computer science at IIT Delhi

Read More

— FloydHub Blog —
Data Science



FloydHub Cloud Setup Challenge: Jupyter + TensorFlow in 44 seconds [WR]

Naïve Bayes for Machine Learning – From Zero to Hero

A Pirate's Guide to Accuracy, Precision, Recall, and Other Scores

See all 8 posts →



DEEP LEARNING

Attention Mechanism

What is Attention, and why is it used in state-of-the-art models? This article discusses the types of Attention and walks you through their implementations.



19 MIN READ



DEEP LEARNING

When Not to Choose the Best NLP Model

The world of NLP already contains an assortment of pre-trained models and techniques. This article discusses how to best discern which model will work for your goals.



15 MIN READ

FloydHub Blog © 2020

[Latest Posts](#) · [Facebook](#) · [Twitter](#) · [Ghost](#)