Multi-threading and Multi-processing in Python

A deep dive into multi-threading and multi-processing with Python and how they are related to concurrency and parallelism

Introduction

Threading and multi-processing are two of the most fundamental concepts in programming. If you have been coding for a while, you should have already come across with use-cases where you'd want to speed up specific operations in some parts of your code. Python supports various mechanisms that enable various tasks to be executed at (almost) the same time.

In this tutorial we will grasp an understanding of **multi-threading** and **multi-processing** and see in practice how these techniques can be implemented in Python. We'll also discuss about which technique to use based on whether the application is *I/O* or *CPU bound*.

Before discussing about threading and multi-processing it's important to understand two terms that are often used interchangeably. **Concurrency** and **parallelism** are closely *related* but *distinct* concepts.

Concurrency and Parallelism

In many occasions we may need to speed up a few operations in our code base in order to boost the performance of the execution. This can normally be achieved by executing multiple tasks in parallel or concurrently (i.e. by interleaving between multiple tasks). Whether you could take advantage of concurrency or parallelism really depends on your code but on the machine that is running it, too.

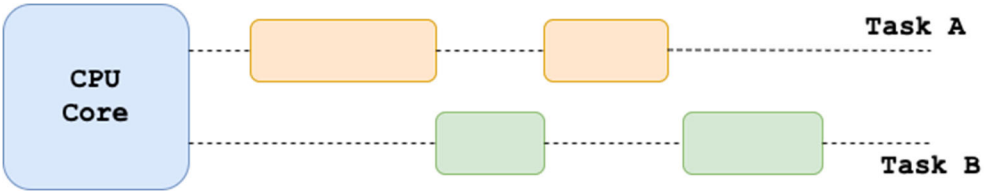
In **concurrent execution,** two or more tasks can start, execute and complete in overlapping time periods. Therefore, these tasks don't necessarily have to run simultaneously — they just need to make progress in an overlapping manner.

> Concurrency: A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.
>
> — Sun's Multithreaded Programming Guide

Now let's consider a use-case where we have a computer with one single-core CPU. This means that the tasks that need to be executed as part of an application cannot make progress at exactly the same time since the processor is only capable of working on just a single task at a time. Running multiple tasks concurrently, means that the processor performs context switching so that multiple tasks can be in-progress at the same time.

One of the main goals of concurrency is to prevent tasks from blocking each other by switching back and forth, when one of the tasks is forced to wait (say for a response from an external resource). For example, Task A progresses till a certain point, the the CPU stops working on Task A, switches to Task B and starts working on it for a while and then it could switch back to Task A to finish it, and finally return back to Task B till it finishes this task, too.

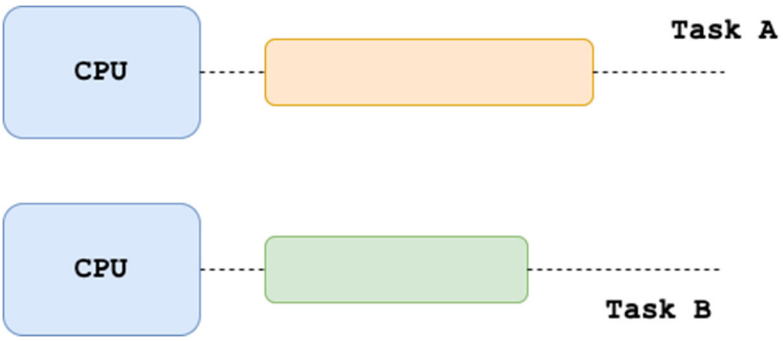
An application consisting of two tasks that are being executed concurrently in a single core is illustrated in the diagram below.



On the other hand, in **parallelism** multiple tasks (or even several components of one task) **can literally run at the same time** (e.g on a multi-core processor or on a machine with multiple CPUs). Therefore, it is not possible to have parallelism on machines with a single processor and single core.

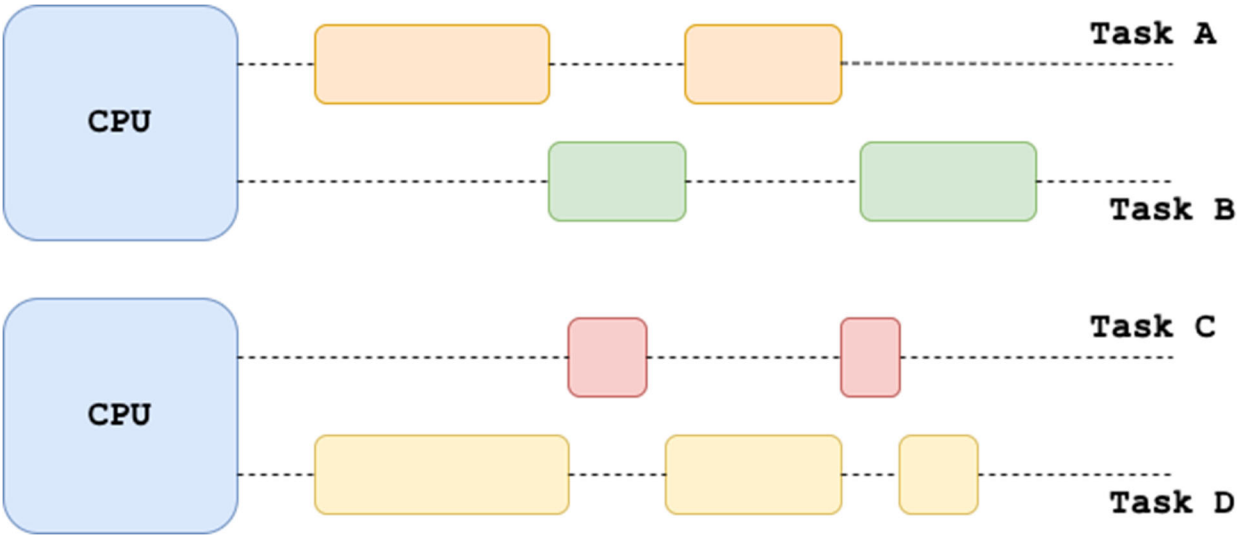
> Parallelism: A condition that arises when at least two threads are executing simultaneously.
>
> — Sun's Multithreaded Programming Guide

With parallelism, we are able to maximise the use of hardware resources. Consider the scenario where we have 16 CPU cores — it would be probably smarter to launch multiple processes or threads in order to make use of all these cores rather than relying on just a single core while the remaining 15 are idle. In multi-core environments, each core can execute one task at exactly the same time, as illustrated in the diagram below:



To recap, concurrency can be seen as a **property** of a system or program and refers to how a single CPU (core) can make progress on multiple tasks **seemingly** at the same time (i.e. concurrently) while parallelism is the actual **run-time behavior** of executing at least two tasks literally at the same time, in parallel. Additionally, it is important to highlight that **both concurrency and parallelism could be combined during task execution**. In fact, we could have all sort of combinations;

- **Neither concurrent, nor parallel**: This is also known as *sequential execution* where tasks are executed strictly one after the other.
- **Concurrent, but not parallel**: This means that the tasks make progress *seemingly* at the same time, but in fact the system switches between various tasks that are concurrently in progress, until all of them are executed. Therefore, there is no true parallelism and thus no two tasks are being executed at exactly the same time.
- **Parallel, but not concurrent**: This is a fairly rare scenario where only one task is being executed at any given time, but the task itself is broken down into sub-tasks that are being processed in parallel. Every task though, must be completed before the next task is picked up and executed.
- **Concurrent and parallel**: This could happen basically in two ways; The first is the simple parallel and concurrent execution where the application fires up multiple threads that are being executed on multiple CPUs and/or cores. The second way that this can be achieved is when the application is able to work on multiple tasks concurrently but at the same time it also breaks down each individual task into sub-tasks so these sub-tasks can eventually be executed in parallel.



Now that we have a basic understanding about how concurrency and parallelism work let's explore multi-processing and multi-threading using some examples in Python.

Threading in Python

A thread is a sequence of instructions that are being executed within the context of a process. One process can spawn multiple threads but all of them will be sharing the same memory.

When experimenting with multi-threading in Python on CPU-bound tasks, you'll eventually notice that the execution is not optimised and it may even run slower when multiple threads are used. Normally, the expectation would be that the use of a multi-threaded code on a multi-core machine should take advantage of the cores available and thus increase the overall performance.

In fact, a Python process cannot run threads in parallel but it can run them concurrently through context switching during I/O bound operations.

This limitation is actually enforced by GIL. The **Python Global Interpreter Lock** (GIL) prevents threads within the same process to be executed at the same time.

> GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once — Python Wiki

The GIL is necessary because Python's interpreter is **not thread-safe**. This global lock is enforced every time we attempt to access Python objects within threads. At any given time, only one thread can acquire the lock for a specific object. Therefore, CPU-bound code will have no performance gain with Python multi-threading.

CPython implementation detail:

In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation).
If you want your application to make better use of the computational resources of multi-core machines, you are advised to use `multiprocessing` *or* `concurrent.futures.ProcessPoolExecutor`.
- Python Docs

Now that we have an understanding about how multi-threaded applications work in Python, let's write some code and take advantage of this technique.

In Python, threads can be implemented with the use of **threading** module. Now let's consider a function that is used to download an image — this is clearly a I/O-bound task:

```
import requests


def download_img(img_url: str):
        """
        Download image from img_url in curent directory
        """
        res = requests.get(img_url, stream=True)
        filename = f"{img_url.split('/')[-1]}.jpg"

        with open(filename, 'wb') as f:
                for block in res.iter_content(1024):
                        f.write(block)
```

view rawdownload_image.py hosted with ♡ by GitHub

Then let's try to download a few images from Unsplash using the code snippet below. Note that for to demonstrate the effect of threading more clearly, we intentionally attempt to download these images 5 times (see the `for` loop):

```python
import requests


def download_img(img_url: str):
        """
        Download image from img_url in curent directory
        """
        res = requests.get(img_url, stream=True)
        filename = f"{img_url.split('/')[-1]}.jpg"

        with open(filename, 'wb') as f:
                for block in res.iter_content(1024):
                        f.write(block)


if __name__ == '__main__':
    images = [
        # Photo credits: https://unsplash.com/photos/IKUYGCFmfw4
        'https://images.unsplash.com/photo-1509718443690-d8e2fb3474b7',

        # Photo credits: https://unsplash.com/photos/vpOeXr5wmR4
        'https://images.unsplash.com/photo-1587620962725-abab7fe55159',

        # Photo credits: https://unsplash.com/photos/iacpoKgpBAM
        'https://images.unsplash.com/photo-1493119508027-2b584f234d6c',

        # Photo credits: https://unsplash.com/photos/b18TRXc8UPQ
        'https://images.unsplash.com/photo-1482062364825-616fd23b8fc1',

        # Photo credits: https://unsplash.com/photos/XMFZqrGyV-Q
        'https://images.unsplash.com/photo-1521185496955-15097b20c5fe',

        # Photo credits: https://unsplash.com/photos/9SoCnyQmkzI
        'https://images.unsplash.com/photo-1510915228340-29c85a43dcfe',

    ]

    for img in images * 5:
         download_img(img)
```


So our tiny app works fine but we can definitely do better and optimize the code by taking advantage of threads (don't forget that downloading -multiple- images is a I/O-bound task).

```python
import requests
from queue import Queue
from threading import Thread


NUM_THREADS = 5
q = Queue()


def download_img():
        """
        Download image from img_url in curent directory
        """
        global q

        while True:
                img_url = q.get()

                res = requests.get(img_url, stream=True)
                filename = f"{img_url.split('/')[-1]}.jpg"

                with open(filename, 'wb') as f:
                        for block in res.iter_content(1024):
                                f.write(block)
                q.task_done()


if __name__ == '__main__':
    images = [
        # Photo credits: https://unsplash.com/photos/IKUYGCFmfw4
        'https://images.unsplash.com/photo-1509718443690-d8e2fb3474b7',

        # Photo credits: https://unsplash.com/photos/vpOeXr5wmR4
        'https://images.unsplash.com/photo-1587620962725-abab7fe55159',

        # Photo credits: https://unsplash.com/photos/iacpoKgpBAM
        'https://images.unsplash.com/photo-1493119508027-2b584f234d6c',

        # Photo credits: https://unsplash.com/photos/b18TRXc8UPQ
        'https://images.unsplash.com/photo-1482062364825-616fd23b8fc1',

        # Photo credits: https://unsplash.com/photos/XMFZqrGyV-Q
        'https://images.unsplash.com/photo-1521185496955-15097b20c5fe',
```

```
44
45          # Photo credits: https://unsplash.com/photos/9SoCnyQmkzI
46          'https://images.unsplash.com/photo-1510915228340-29c85a43dcfe',
47      ]
48
49      for img_url in images * 5:
50          q.put(img_url)
51
52      for t in range(NUM_THREADS):
53
54          worker = Thread(target=download_img)
55          worker.daemon = True
56          worker.start()
57
58      q.join()
```

view rawdownload_unsplash_threaded.py hosted with ♡ by GitHub

To recap, threading in Python allows multiple threads to be created within a single process, but due to GIL, none of them will ever run at the exact same time. **Threading is still a very good option when it comes to running multiple I/O bound tasks concurrently**. Now if you want to take advantage of computational resources on multi-core machines, then multi-processing is the way to go. You should also note that threading comes with the overhead associated with managing the threads and therefore you should avoid using them for basic tasks. Additionally, they also increase the complexity of the program which means that debugging may become a little bit tricky. Therefore, use threading only wherever there's a clear value when doing so.

Multi-processing in Python

Now if we want to take advantage of multi-core systems and eventually run tasks in a truly parallelised context, we need to perform multi-processing instead of multi-threading.

In Python, multi-processing can be implemented using the **multiprocessing** module (*or* `concurrent.futures.ProcessPoolExecutor`) that can be used in order to spawn multiple OS processes. Therefore, **multi-processing in Python side-steps the GIL** and the limitations that arise from it since every process will now have its own interpreter and thus own GIL.

`multiprocessing` *is a package that supports spawning processes using an API similar to the* `threading` *module.*

The `multiprocessing` package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads.
Due to this, the `multiprocessing` module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.
- Python Docs

In the previous section we talked about Threading and we saw that threading does not improve CPU-bound tasks at all. This can be achieved with the use of multiprocessing. Let's use the same function `append_to_list()` we used in the previous section, but this time instead of `threading` we are going to use `multiprocessing` in order to take advantage of our multi-core machine.

Now let's **consider a CPU-bound operation** that involves a function which appends multiple random integers to a list.

```
import random

def append_to_list(lst, num_items):
        """
        Appends num_items integers within the range [0-20000000) to the input lst
        """
        for n in random.sample(range(20000000), num_items):
                lst.append(n)
```

view rawmultiprocessing_example.py hosted with ♡ by GitHub

Now let's assume we want to run this function twice, as illustrated below:

```
def append_to_list(lst, num_items):
        """
        Appends num_items integers within the range [0-20000000) to the input lst
        """
        for n in random.sample(range(20000000), num_items):
                lst.append(n)


if __name__ == "__main__":
    for i in range(2):
        append_to_list([], 10000000)
```

view rawmultiprocessing_example.py hosted with ♡ by GitHub

And let's `time` this execution and inspect the results.

```
$ time python3 test.py

real    0m35.087s
```

```
4   user    0m34.288s
5   sys     0m0.621s
```

view rawmultiprocessing_example_results.txt hosted with ♡ by GitHub

Now let's refactor our code a little bit and now use two different processes so that each call to the function is executed in its own process:

```
1   import random
2   import multiprocessing
3
4
5   NUM_PROC = 2
6
7
8   def append_to_list(lst, num_items):
9           """
10          Appends num_items integers within the range [0-20000000) to the input lst
11          """
12          for n in random.sample(range(20000000), num_items):
13                  lst.append(n)
14
15
16  if __name__ == "__main__":
17          jobs = []
18
19          for i in range(NUM_PROC):
20                  process = multiprocessing.Process(
21                          target=append_to_list,
22                      args=([], 10000000)
23                  )
24                  jobs.append(process)
25
26          for j in jobs:
27                  j.start()
28
29          for j in jobs:
30                  j.join()
```

view rawmultiprocessing_example.py hosted with ♡ by GitHub

And finally let's `time` and execution and inspect results:

```
1   $ time python3 test.py
2
3   real    0m15.251s
4   user    0m29.599s
5   sys     0m0.659s
```

We can clearly see that (even though `user` and `sys` times remained approximately the same), the `real` time has dropped by a factor even greater than two (and this is expected since we essentially distributed the load to two distinct processes so that they can run in parallel).

To recap, multi-processing in Python can be used when we need to take advantage of the computational power from a multi-core system. In fact, multiprocessing module lets you run multiple tasks and processes in **parallel**. In contrast to threading, `multiprocessing` side-steps the GIL by using subprocesses instead of threads and thus multiple processes can run literally at the same time. This technique is mostly suitable for CPU-bound tasks.

Final Thoughts

In today's article we introduced two of the most fundamental concepts in programming, namely concurrency and parallelism and how they differ or even be combined when it comes to execution. Furthermore, we discussed about threading and multi-processing and explored their main advantages and disadvantages as well as a few use-cases that could eventually help you understand when to use one over the other. Finally, we showcased how to implement threaded or multi-processing applications with Python. To recap,

Threading

- Threads share the same memory and can write to and read from shared variables
- Due to Python Global Interpreter Lock, two threads won't be executed at the same time, but concurrently (for example with context switching)
- Effective for I/O-bound tasks
- Can be implemented with `threading` module

Multi-processing

- Every process has is own memory space
- Every process can contain one ore more subprocesses/threads
- Can be used to achieve parallelism by taking advantage of multi-core machines since processes can run on different CPU cores
- Effective for CPU-bound tasks
- Can be implemented with `multiprocessing` module (or `concurrent.futures.ProcessPoolExecutor`)