Sequential Exeuction, Multiprocessing, and Multithreading IO-Bound Tasks in Python

By Zac J. Szewczyk on 2020/01/25 12:40:21 EST

Python makes concurrency easy. It took less than an hour to add multiprocessing to my blog engine, First Crack, and I have used it often since. Everyone likes to call premature optimization the root of all evil, but architecting programs for concurrent execution from the start has saved me hundreds of hours in large data capture and processing projects. Color me a Knuth skeptic. This article compares sequential execution, multiprocessing, and multithreading for IO-Bound tasks in Python, with simple code samples along the way.

Terms

First, terms. Most programs work from top to bottom. The next line runs after the last one finishes. We call these **sequential**. Adding **multiprocessing** to First Crack let the script use multiple cores to run multiple lines at the same time. Where the engine used to open a file, read its contents, close it, and then repeat those steps a thousand more times, it could now handle eight at once. **Multithreading** lives somewhere in the middle. These programs use a single core, but the processor forces small blocks — threads — to take turns. By pausing a block waiting to read a file so that another can make a network connection before coming back to the first, multithreading boosts efficiency and lowers runtime. The latter approaches are examples of **concurrency**, which — to make things easy — you can just think of as anything *not* sequential.

The rest of this article starts with a simple sequential script, after the section below, before moving on to much faster concurrent versions later. Each section also includes runtime analysis, so you can see just how big an impact concurrency can have.

Imports

I excluded the import statements from the code in the following sections, for brevity's sake. For those who want to follow along at home, make sure your script starts with these lines:

```
# Imports
from sys import argv # Capture command line parameters
from multiprocessing import Pool as CorePool # Multiprocessing
from multiprocessing.pool import ThreadPool # Multithreading
from time import sleep # Sleep
from math import ceil # Rounding
from datetime import datetime # Execution time
```

I will not use most of these functions for a while, but use all of them in time. If you do decide to follow along at home, you will need Python 3.7 or later.

Sequential Execution

The first — and most common — approach to concurrency is to avoid it. Consider this simple script:

```
# Method: handle
# Purpose: Handle item.
# Parameters:
# - item: Item to handle (X)
# Return: 0 - Success, 1 - Fail (Int)
def handle(item):
    sleep(2)
    return 1 # Success

x = range(100)

t1 = datetime.now()
for each in x:
   handle(each)
t2 = datetime.now()
print("Sequential time: {}".format((t2-t1).total_seconds()))
```

The generic method `handle` does nothing — but by sleeping for two seconds, it simulates a long IO-bound task. I chose to simulate this type of task — as opposed to a CPU-bound one — because most of my recent projects have involved downloading and reading massive files. These types of jobs spend most of their time waiting on data from the network or an external hard drive, which requires little from the processor. In a real project, I might replace `sleep(2)` with code to download a web page, parse the HTML, and write to a file. For consistency across runs, I just use `sleep` here. `x = range(100)` creates a list from 0 to 99, and the `for` loop then calls `handle` one hundred times, once for each number.

We can model best-case runtime with the formula `T = h * n + o`, with `T` as total execution time, `h` as the amount of time `handle` takes to run, `n` as the number of times the method runs, and `o` as the overhead to initialize, operate, and exit the script. The script above should take just over 200 seconds to finish: `T = 2 * 100 + o = 200 + o`

The script ran for 200.17 seconds, which makes `o` — the overhead to initialize, operate, and exit the script, in seconds — equal to 0.17. Next, let's add multiprocessing and see what changes.

Multiprocessing

Consider the script below. `handle` remains unchanged. `x = range(100)` creates the same one hundred-item list from 0 to 99, but then `Core_Orchestrator` calls `handle` for each number 0 to 99.

```
# Method: handle
# Purpose: Handle item.
# Parameters:
# - item: Item to handle (X)
# Return: 0 - Success, 1 - Fail (Int)
def handle(item):
    sleep(2)
    return 1 # Success

# Method: Core_Orchestrator
# Purpose: Facilitate multiprocessing.
# Parameters:
# - input_list: List to farm out to cores (List)
# Return: True, All successful; False, At least one fail (Bool)
def Core_Orchestrator(input_list):
    pool = CorePool(processes=MAX_CORES)
    results = pool.map(handle, input_list)
    pool.close()
    pool.join()
    del pool
    return all(results)
```

```
x = range(100)

t1 = datetime.now()
Core_Orchestrator(x)
t2 = datetime.now()
print("Multiprocessing time: {}".format((t2-t1).total_seconds()))
```

Recall that multiprocessing lets the script use multiple cores to run multiple lines at the same time. My computer has eight cores, so `Core_Orchestrator` runs eight instances of `handle` at once. We can now model execution time with `T = (h * n)/c + n/c*y + o`.

Let me break this new formula down: we can represent the time to run `handle` `n` times with `(h * n)`. `(h * n)/c`, then, becomes the time to run `handle n` times on `c` cores. Multiple cores introduce some overhead, though, which we can account for with `n/c*y`: the number of times the script will have to assign `handle` to a new core, `n/c`, times the unknown amount of time that takes, `y`. `o` is, again, the overhead to initialize, operate, and exit the script.

Assuming `o` remains constant, our new formula says we can expect the code above to take at least 25 seconds: `T = (2 * 100)/8 + 100/8*y + 0.17 = 25.17 + 12.5y`. Since we don't have a value for `y`, though, we cannot say how far over. Let's find out.

The script finishes in **0:00:32.09**, which gives us a value of 0.55 for `y`. At this point, we have two extremes by which to judge performance: the sequential approach took 200.17 seconds, while the multiprocessor approach took 32.09 seconds. Let's see if we can beat it with multithreading, next.

Multithreading

In the script below, `handle` once again remains unchanged, `x = range(100)` creates the same one hundred-item list from 0 to 99, but then `Thread_Orchestrator` calls `handle` for each number 0 to 99. `Thread_Orchestrator` uses a max of eight threads.

Will this script match the performance of the sequential one, with `T = h * n + o`? It runs on a single core, after all. Or will it look more like the multiprocessed code, with execution time measured by `T = (h * n)/c + n/c*y + o`?

```
# Method: Thread_Orchestrator
# Purpose: Facilitate multithreading.
# Parameters:
# - input_list: List to farm out to threads (List)
# Return: True, All successful; False, At least one fail (Bool)
def Thread_Orchestrator(in_list):
    try:
        thread_pool = ThreadPool(8)
        results = thread_pool.map(handle, in_list)
        thread_pool.close()
        thread_pool.join()
        del thread_pool
        return all(results)
    except Exception as e:
        # print(e)
        return False

x = range(100)

t1 = datetime.now()
Thread_Orchestrator(x)
t2 = datetime.now()
print("Multithreading time: {}".format((t2-t1).total_seconds()))
```

This script finished in **32.08** seconds. Whether eight cores sleep for two seconds or a single core waits for eight threads to sleep for two seconds apiece, the same amount of time passes. As a result, the non-parallel multithreaded approach managed to match the parallel multiprocessing one. In general, the execution time for these two approaches will match for IO-bound tasks; it will not for CPU-bound tasks, though. If I had used a complex math operation that required many CPU cycles, the multiprocessing method would have split the work across eight cores, while a single one would have had to do all the work for each thread in the multithreaded code. The table below explains when to use these strategies.

Bottleneck	Example	Optimize with
IO	Network connection, file operation	Multithreading
CPU	Complex math problem, search	Multiprocessing

Given the simulated IO-bound task here, if the multiprocessing version ran just as fast as the multithreaded one, why bother with multithreading at all? Because the max number of cores a processor has is a hard physical limit, while the max number of threads is a logical one. I can never use more than eight cores, but I can use as many threads as my operating system will allow. In practice, I have found that limit hovers around 1,000 per process.

To answer my question from earlier, we can model multithreaded performance like we did multiprocessing[1], with `T = (h * n)/t + n/t*z + o` — except with `t` as the number of threads used, and `z` as the overhead of assigning `handle` to a new thread. Using the execution time `T` of the last run, 32.08 seconds, we now have a value for `z`: 0.55. This formula also tells us that we can minimize `T` by increasing `t` toward its limit around 1,000. Let's test this theory.

The script below uses 16 threads (`t=16`). According to our formula and assuming `o` and `z` remain constant, it should finish in about 16 seconds: `T = 200/16 + 100/16(0.55) + 0.17 = 16.11`

```
# Method: Thread_Orchestrator
# Purpose: Facilitate multithreading.
# Parameters:
# - input_list: List to farm out to threads (List)
# Return: True, All successful; False, At least one fail (Bool)
def Thread_Orchestrator(in_list):
    try:
        thread_pool = ThreadPool(16)
        results = thread_pool.map(handle, in_list)
        thread_pool.close()
        thread_pool.join()
        del thread_pool
        return all(results)
    except Exception as e:
        # print(e)
        return False

x = range(100)

t1 = datetime.now()
Thread_Orchestrator(x)
t2 = datetime.now()
print("Multithreading time: {}".format((t2-t1).total_seconds()))
```

The script finished in **16.07** seconds with 16 threads, and **2.15** seconds with 100. More threads over 100 could not make this faster, though, because the script only had 100 tasks to complete; more would just go unused. Is this the best we can do? No: if the multiprocessing code ran on a machine with 100 cores, each core would run `handle` once and all cores would run their instance at the

same time; execution would take about 2 seconds, since `handle` takes 2 seconds. Are 2.15 seconds *realistically* the best we can do, though? Maybe; I don't have a 100 core machine laying around — but perhaps we can get closer, by combining multiprocessing and multithreading.

Multiprocessing + Multithreading: The Blended Approach

Getting multithreading and multiprocessing to work together took some work. I'll walk you through the code first, then delve into the results.

```
# Global control variables
# MAX_CORES: Maximum number of cores
MAX_CORES = 8
```

`multiprocessing.Pool()` creates a handle through which the script delegates tasks to individual cores. This command uses `multiprocessing.cpu_count()` to define the number of available cores in the pool. In the virtual environment I wrote most of this article in, though, that function gave me incorrect results. Creating a variable `MAX_PROCESSORS` and then overriding `multiprocessing.cpu_count()` with it when instantiating the pool fixed the problem. Your mileage may vary.

```
# Method: Core_to_Thread_Orchestrator
# Purpose: Facilitate multiprocessing and multithreading.
# Parameters:
# - input_list: List to farm out to threads by core (List)
# Return: True, All successful; False, At least one fail (Bool)
def Core_to_Thread_Orchestrator(input_list):
    try:
        pool = CorePool(processes=MAX_CORES)
        n = ceil(len(x)/MAX_CORES)
        results = pool.map(Thread_Orchestrator, [list(input_list[i:i+n]) for i in range(0, len(inp
        pool.close()
        pool.join()
        del pool, n
        return all(results)
    except Exception as e:
        # print(e)
        return False
```

`Core_to_Thread_Orchestrator` accepts an input list, conveniently named `input_list`, then creates a pool of cores. The line, `results = pool.map(Thread_Orchestrator, [list(input_list[i:i+n]) for i in range(0, len(input_list), n)])`, needs some extra explaining.

1. **Divide `input_list` into even sub-lists for each core.** `n = ceil(len(x)/MAX_CORES)` uses `ceil` to make sure a list of 100 elements on an 8 core machine does not get split into 8 sub-lists with 12 elements each (`int(100/8=12.5)=12`). This would only account for 96 elements and orphan the last 4. For cases like this, `ceil` ensures 8 sub-lists are created with 13 elements each, where the last one has just 9. `[list(input_list[i:i+n]) for i in range(0, len(input_list), n)]` then splits `input_list` into even sub_lists such that each core will have about the same amount of work to do.
2. **Multithread the processing of each sub-list.** `pool.map` hands each sub-list off to a different core's multithreading function. This has two major benefits. First, this allows each core to supervise the multithreading of a fraction of `input_list`, rather than the entire thing. The system then has to create fewer threads per core, which means each core can spend less time pausing and resuming threads. In theory, this approach also multiplies the max number of possible threads: where one core might have tapped

out at 1,000, 8 cores should manage 8,000 without issue. In practice, though, most systems limit thread count by process rather than by core; on my system, that limit hovers around 1,000.

3. **Capture success or failure for all cores.** `result` becomes a list with a return value for each core.

`return all(result)` returns True if all processes succeeded, but False if *any* failed.

```
# Method: Thread_Orchestrator
# Purpose: Facilitate multithreading.
# Parameters:
# - input_list: List to farm out to threads (List)
# Return: True, All successful; False, At least one fail (Bool)
def Thread_Orchestrator(in_list):
    try:
        thread_pool = ThreadPool(len(in_list))
        results = thread_pool.map(handle, in_list)
        thread_pool.close()
        thread_pool.join()
        del thread_pool
        return all(results)
    except Exception as e:
        # print(e)
        return False
```

As we saw earlier, multithreading handles IO-bound tasks best with a thread for each task. Since `Thread_Orchestrator` now receives a variable number of tasks, it now calculates the appropriate number of threads to create with `len(in_list)`. Again, using more threads than tasks would not improve runtime. `results = thread_pool.map(handle, in_list)` then assigns `handle` to a thread for each element in the input list, captures the results in an array just like the previous method, and returns True of all threads succeed. If any fail, `Thread_Orchestrator` returns False.

```
# Read number of items to generate test data set with from parameter.
# Default to 100.
if (len(argv) == 1):
    seed = 100
else:
    seed = int(argv[1])

# Expand the range to a list of values
# seed = 100 -> 100 element list
# seed = 500 -> 500 element list
x = range(seed)

# Print seed and results
print(f"Seed: {seed}")

# # Multiprocess
# t1 = datetime.now()
# if (Core_Orchestrator(x) == False):
#     print("-- Core orchestrator failed.")
# else:
#     t2 = datetime.now()
#     print("Multiprocessing time: {}".format((t2-t1).total_seconds()))

# Multithreading
t1 = datetime.now()
if (Thread_Orchestrator(x) == False):
    print("-- Thread orchestrator failed.")
else:
    t2 = datetime.now()
    print("-- Multithreading time: {}".format((t2-t1).total_seconds()))

# Multiprocessing + multithreading
t1 = datetime.now()
if (Core_to_Thread_Orchestrator(x)  == False):
    print("-- Processor to thread orchestrator failed.")
else:
    t2 = datetime.now()
    print("-- Multiprocessing and multithreading time: {}".format((t2-t1).total_seconds()))
```

The code above accepts a parameter for the number of times `handle` must run, and then records the runtime for the multithreaded method and the blended method. Even if one fails, the test continues. The snippet below tests the limits of both approaches by feeding the script values from 100 to 1,000 in increments of 100. It does this ten times, to lessen the impact of anomalous runs.

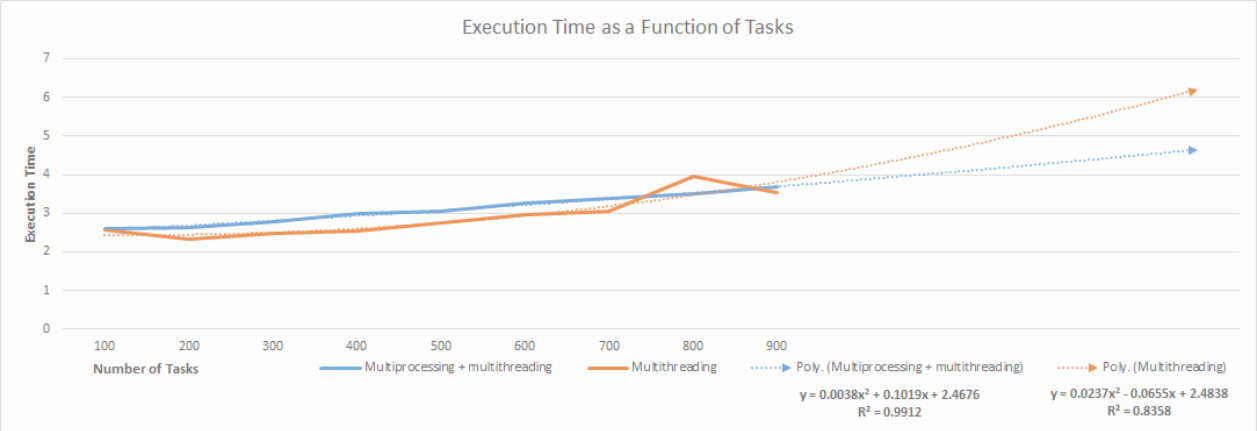
```
for k in {1..10}; do for i in {100..1000..100}; do python3 main.py $i >> $k".txt"; kill $(ps aux |
```

I could have done this all in Python, but this approach does a few things for me. For one, it forces the script to initialize, execute, and exit for each set of tasks from 100 to 1,000. This lessens the chance of cache or memory usage impacting successive runs. `kill $(ps aux | grep python | awk '{print $2}') 2> /dev/null` makes sure no Python processes stick around to interfere with those runs. Again, it also goes through this process ten times, to lessen the impact of anomalous runs. Together, these help give me as unbiased a picture of the script's runtime as possible. Check out the results, tabled below:

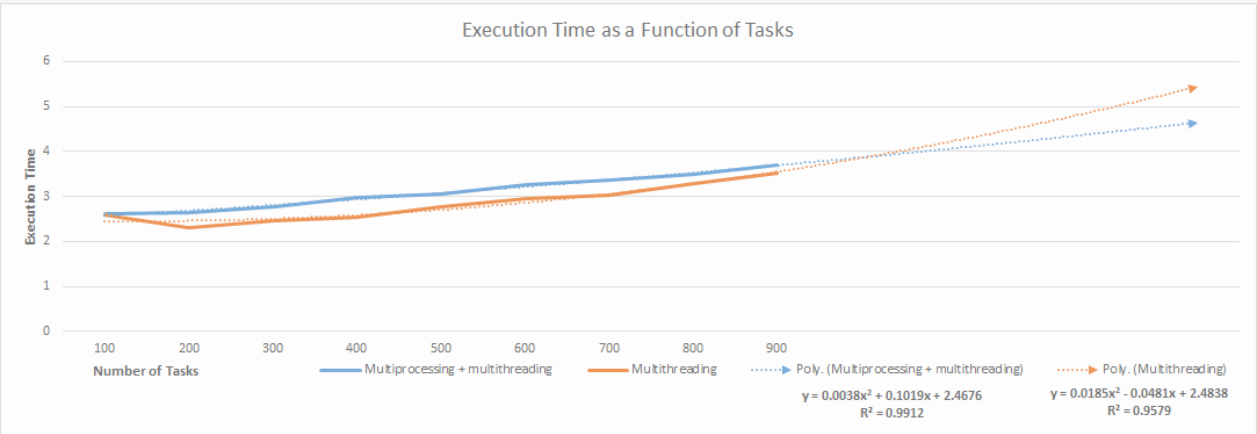
Approach	Tasks	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
Blended	100	3.567204	2.48169	2.469574	2.571985	2.495898	2.509916	2.487354	2.551873	2.520211	2.49428
Blended	200	2.62477	2.721776	2.623119	2.62792	2.666248	2.65166	2.656948	2.607229	2.581664	2.61788
Blended	300	3.084136	2.766728	2.820711	2.718879	2.71363	2.730298	2.764282	2.694365	2.71308	2.71020
Blended	400	3.834809	3.039956	3.025324	2.826441	2.831418	2.902106	2.811155	2.893164	2.824065	2.81503
Blended	500	3.412031	3.085576	3.107757	2.994405	2.970532	3.066721	2.962216	3.010268	2.919665	2.92071
Blended	600	3.920105	3.276128	3.20756	3.191376	3.24736	3.18067	3.095356	3.182228	3.177074	3.12045
Blended	700	3.893453	3.338353	3.45412	3.307333	3.269286	3.375174	3.416062	3.189951	3.309285	3.22316
Blended	800	4.396074	3.409177	3.489747	3.441516	3.416976	3.33599	3.333954	3.409126	3.298936	3.52455
Blended	900	4.138501	3.736235	3.642134	3.608156	3.699388	3.624607	3.776716	3.607106	3.60067	3.53270
Multithreaded	100	2.530468	2.678918	2.375514	2.669632	2.716273	2.433133	2.633597	2.528427	2.689116	2.52975
Multithreaded	200	2.646297	2.301428	2.220176	2.269561	2.33174	2.266194	2.251354	2.290965	2.278389	2.30764
Multithreaded	300	2.516477	2.406957	2.401755	2.533726	2.390161	2.862111	2.38876	2.384127	2.389991	2.39856
Multithreaded	400	2.482953	2.546255	2.676493	2.495154	2.58571	2.522376	2.571323	2.506227	2.574394	2.46832
Multithreaded	500	3.363701	2.85764	2.775013	2.604138	2.645682	2.601216	2.622577	2.702628	2.785129	2.66743
Multithreaded	600	2.987607	2.724781	2.882752	2.681507	2.788063	3.160047	2.780519	3.312241	3.519288	2.78833
Multithreaded	700	3.457589	3.197889	2.950593	2.88573	2.986151	2.9273	2.890827	3.018946	2.894577	3.18995
Multithreaded	800	3.291344	3.208601	3.031519	2.981974	2.979717	6.996563	3.098343	3.908249	3.083194	7.00713

Multithreaded	900	3.532445	3.201465	3.164539	3.478344	5.107173	3.211502	3.120932	3.690724	3.577692	3.14734

The graph below visualizes execution time as a function of tasks, from 100 to 900. After 900, the system refused to spawn new threads; the dotted lines predict execution time beyond that point. `y = 0.0237x2 - 0.0655x + 2.4838` models the multithreading method with a R^2 value of 0.83, and `y = 0.0038x2 + 0.1019x + 2.4676` models the blended method with a R^2 value of 0.99.



The multithreaded method's runtime consistently spikes with 800 tasks. Interestingly, normalizing the average runtime for 800 tasks to fall between 700's and 900's changes the trendline function from `y = 0.0237x2 - 0.0655x + 2.4838` to `y = 0.0185x2 - 0.0481x + 2.4838`, and causes the R^2 value to jump from 0.83 to 0.96. Check out that graph below.



Conclusions and Takeaways

To return to my question from earlier, are 2.15 seconds the best we can do? Recall that `Thread_Orchestrator` blew through 100 simulated IO-bound tasks in 2.15 seconds, using 100 threads. Over ten runs, it averaged 2.16 seconds; the blended multiprocessed + multithreaded method, on the other hand, averaged 2.44 seconds over 10 runs. To answer my question from earlier, then, 2.15 seconds are the best we can do. Multithreading wins for IO-bound tasks.

As the number of IO-bound tasks grows, that eventually changes. The multithreaded method's execution time stays below the blended method's from 100 to 900 tasks, but the former grows faster than the latter. On a system that permits a process to spawn more than

1,000 threads, the blended approach will begin to win out when processing over 1,000 IO-bound tasks. The table below summarizes when to use multithreading, multiprocessing, or a mix of both.

Bottleneck	Example	Tasks	Optimize with
CPU	Complex math problem, search	Any	Multiprocessing
IO	Network connection, file operation	< 1,000	Multithreading
IO	Network connection, file operation	> 1,000	Multiprocessing + multithreading

Use this table to choose an approach, and the scripts above to make quick work of even large jobs. Multi-core, multithreaded architectures mean no one should have to suffer through painful sequential execution anymore. Python makes concurrency easy, so take advantage of it.

↩ I understand that `T = (h * n)/t + n/t*z + o` implies simultaneous execution, which is correct when using multiprocessing but not when using multithreading. Multithreaded programs run on a single core. Although the processor pauses and resumes threads so fast that it gives the impression of parallel execution, they do not execute in parallel. In this scenario, though, this is effectively a meaningless distinction. Multithreaded IO-bound tasks are essentially indistinguishable from multiprocessed ones, given an equal number of threads and cores.