**Threading vs Multiprocessing in Python: A Comprehensive Guide**

[[Arjun Prakash](https://medium.com/@arjunprakash027?source=post_page-----cae3ce0ca6c1--------------------------------)](https://medium.com/@arjunprakash027?source=post_page-----cae3ce0ca6c1--------------------------------)

[Arjun Prakash](https://medium.com/@arjunprakash027?source=post_page-----cae3ce0ca6c1--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F376a20930ef1&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40arjunprakash027%2Fthreading-vs-multiprocessing-in-python-a-comprehensive-guide-cae3ce0ca6c1&user=Arjun+Prakash&userId=376a20930ef1&source=post_page-376a20930ef1----cae3ce0ca6c1---------------------post_header-----------)

9 min read

·

Jun 29

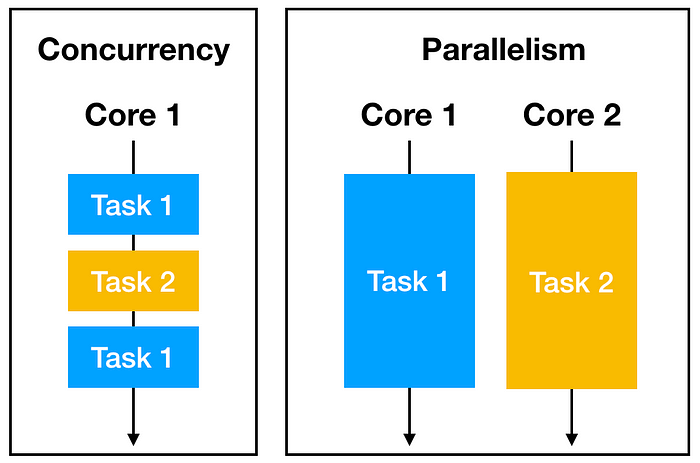
6

**Introduction**

Concurrent programming plays a vital role in optimizing performance and improving efficiency in modern applications. Python offers two primary approaches for achieving concurrency: threading and multiprocessing. In this blog, we will delve into the world of threading and multiprocessing, exploring their differences, advantages, and best use cases.

To provide a practical understanding, we will demonstrate these concepts using a comprehensive example. We will design a image downloading task and measure its performance using both threading and multiprocessing. By examining the performance metrics, we can gain insights into when to leverage threading and multiprocessing for optimal results.

**Concurrency and Parallelism**



source -> [Evolution of Concurrency in Java | LinkedIn](https://www.linkedin.com/pulse/evolution-concurrency-java-eric-kolotyluk/)

**Concurrency**

Concurrency refers to the ability of an application to handle multiple tasks concurrently. Rather than executing tasks in a strictly sequential manner, concurrency allows for overlapping and interleaving of tasks. By utilizing context switching, the application can switch rapidly between different tasks, giving the impression of parallel execution.

Concurrency is achieved using a single processing unit, such as a CPU. Unlike true parallel processing, where multiple tasks are executed simultaneously on separate cores, concurrency maximizes the utilization of a single core by rapidly switching between tasks. This approach significantly reduces the overall execution time of tasks.

When employing concurrency, tasks are divided into smaller chunks or threads. These threads are not processed in parallel, but rather scheduled and switched between rapidly. The operating system or programming language manages the scheduling and context switching, allowing each task to make progress in a time-sliced manner.

**Parallelism**

Parallelism, on the other hand, involves true simultaneous execution of tasks across multiple processing units. Unlike concurrency, which utilizes smaller subtasks, parallel processing divides the tasks into independent units that can be executed in parallel. By leveraging multiple processing units, parallelism significantly increases the speed and throughput of an application.

In parallelism, the tasks are executed simultaneously, enabling the overlapping of CPU-intensive tasks and I/O tasks across different processes. This allows for maximum utilization of resources and efficient task execution. In concurrency, the overlapping occurs between the I/O tasks of one process and the CPU tasks of another process. While concurrency provides the illusion of simultaneous execution, it is not true parallelism.

**Use Cases**

1. ***Concurrency Use Case***: Web Scraping Web scraping involves fetching data from multiple web pages simultaneously. By employing concurrency, you can make concurrent requests to different web pages using threads or asynchronous programming libraries like asyncio or gevent. This allows for efficient utilization of I/O resources and faster data retrieval.
2. ***Parallelism Use Case***: Data Processing When dealing with large datasets, parallelism can significantly enhance data processing speed. For instance, if you need to perform complex calculations on each data point independently, you can distribute the workload across multiple CPU cores using multiprocessing. Each core can handle a portion of the data, leading to faster overall processing time.

**Drawbacks**

**Concurrency**

1. Synchronization Overhead: Coordinating shared resources and managing thread safety introduces additional complexity and overhead. Ensuring proper synchronization between threads can be challenging and may lead to issues like deadlocks or race conditions.
2. Debugging Complexity: Debugging concurrent programs can be more challenging due to non-deterministic behavior caused by thread interleaving. Identifying and fixing issues related to thread synchronization or data sharing can be time-consuming.

**Parallelism**

1. Communication Overhead: Coordinating communication and data sharing between parallel processes can introduce additional complexity and overhead. Explicit mechanisms like interprocess communication (IPC) or shared memory need to be implemented, which can impact performance.
2. Load Imbalance: In parallel computing, load balancing becomes crucial to ensure that tasks are evenly distributed across processes. Load imbalance can lead to underutilization of some processes and inefficient resource utilization

**Threading and Multiprocessing in Python**

**Threading**

We learnt about concurrency in previous section, Threading in Python allows for concurrent execution of tasks by utilizing multiple threads within a single process. Threads share the same memory space and can switch rapidly between tasks, giving the illusion of parallel execution. However, due to the Global Interpreter Lock (GIL) in Python, threading is more suitable for I/O-bound tasks, where threads can wait for I/O operations without blocking the entire process. This makes threading well-suited for achieving concurrency in Python applications.

How to use threading in python:

Import the threading module in python

import threading

Define a function that must be executed using threads

def thread\_function(name):  
 print("Hello from thread", name)

Create a thread and execute the function

threading.Thread() is used to create a thread taking our desired function as arguement

thread.start() starts the thread and thread.join() stops the thread

thread = threading.Thread(target=thread\_function, args=("Thread 1",))  
thread.start()  
thread.join()

**Multiprocessing**

Multiprocessing in Python enables true parallelism by utilizing multiple processes that can run on separate CPU cores. Each process has its own memory space, allowing for independent execution of tasks. Multiprocessing is ideal for CPU-bound tasks, where the workload can be divided and executed in parallel across multiple cores. Unlike threading, multiprocessing can fully utilize multiple CPU cores and achieve significant speed improvements for parallel tasks.

How to use multiprocessing in python:

import multiprocessing module

import multiprocessing

define the function that must be executed parallely

def process\_function(name):  
 print("Hello from process", name)

start the process using multiprocessing.Process() method, it takes function name as arguement

process.start() starts the process and process.join() stops the process

process = multiprocessing.Process(target=process\_function,args=("process1",))  
process.start()  
process.join()

**Benchmarking threading and multiprocessing in python**

In this section, we will compare the download time of images using normal process, threading process, and multiprocessing process in Python. To achieve this, we will utilize the threading module for threading, the multiprocessing module for multiprocessing, and the timeit module to benchmark the execution time.

The goal is to demonstrate the performance differences between the different approaches. By measuring the time it takes to download multiple images using each method, we can observe the impact of concurrency and parallelism on the overall execution time.

**Common code for all 3 methods(normal, threading,multiprocessing)**

The provided code snippet begins by importing necessary modules (timeit, requests, threading, and multiprocessing)

import timeit  
import requests  
import threading  
import multiprocessing

create a list image\_urls that stores URLs for downloading images from the 'picsum.photos' service.

image\_urls = []  
for i in range(100,500):  
 image\_urls.append(f'https://picsum.photos/{i}')  
  
print(image\_urls)

The download\_image function is defined to handle the downloading of images. It takes an image\_url parameter and uses the requests library to retrieve the image from the URL. If the download is successful (status code 200), the image is saved to the 'images' directory with a filename derived from the URL. If there is an error downloading the image, an error message is printed.

def download\_image(image\_url):  
 response = requests.get(image\_url)  
 if response.status\_code == 200:  
 with open(f"images/{image\_url.split('/')[-1]}.png", 'wb') as f:  
 f.write(response.content)  
 else:  
 print(f'Error downloading image {image\_url}')

**Normal Execution**

The normal\_execution function is defined to download images using a normal execution approach, where each image is downloaded sequentially. The function utilizes a for loop to iterate over each image URL in the image\_urls list and calls the download\_image function to download the image.

The execution time is measured using the timeit module. The default\_timer function from timeit is used to record the start and end times of the execution. The difference between these two times represents the total execution time. The calculated execution time is then printed as "Normal Execution Time".

Remember to have the download\_image function defined and available for use when executing the normal\_execution function.

def normal\_execution():  
 start = timeit.default\_timer()  
 for image\_url in image\_urls:  
 download\_image(image\_url)  
 end = timeit.default\_timer()  
 print(f'Normal Execution Time: {end-start}')  
  
normal\_execution()

*Normal Execution Time: 265.1431192*

it takes about 265 seconds to download 400 images using normal method

**Threading**

The threading\_download function is defined to download images using threading for concurrent execution. It creates multiple threads, where each thread is responsible for downloading an image from the list of image URLs.

The function begins by measuring the start time using timeit.default\_timer(). It initializes an empty list called threads to store the created threads.

Next, a for loop iterates over each image URL in the image\_urls list. For each URL, a new thread is created using the Thread class from the threading module. The target parameter is set to the download\_image function, and the args parameter is used to pass the image URL as an argument to the function. The created thread is then appended to the threads list.

After creating all the threads, another for loop starts each thread using the start() method.

To ensure that the main program waits for all the threads to finish their execution, another for loop with join() is used.

Finally, the end time is measured using timeit.default\_timer(), and the difference between the start and end times represents the total execution time. The calculated execution time is then printed as "Threading Execution Time".

Remember to have the download\_image function defined and available for use when executing the threading\_download function.

def threading\_download():  
 start = timeit.default\_timer()  
 threads = []  
  
 for image\_url in image\_urls:  
 t = threading.Thread(target=download\_image,args=(image\_url,))  
 threads.append(t)  
   
 for thread in threads:  
 thread.start()  
  
 for thread in threads:  
 thread.join()  
   
 end = timeit.default\_timer()  
  
 print(f'Threading Execution Time: {end-start}')  
  
threading\_download()

*Threading Execution Time: 4.259069499999896*

it takes only 4.2 seconds to download using threads

**Multiprocessing**

The multiprocessing\_download function utilizes Python's multiprocessing module to achieve parallel execution of image downloads. It creates multiple processes, each responsible for downloading an image from the provided list of image URLs. By leveraging multiprocessing, the function enables concurrent execution of the download tasks, improving overall performance.

The function starts by creating processes for each image URL using the Process class. These processes are stored in a list called processes. It then starts each process using the start() method and waits for them to finish using the join() method. The execution time is measured using timeit.default\_timer() and printed as "Multiprocessing Execution Time."

Note that the download\_image function needs to be defined and available for use when running the multiprocessing\_download function.

def multiprocessing\_download():  
 start = timeit.default\_timer()  
 processes = []  
  
 for image\_url in image\_urls:  
 print('downloading image ',image\_url)  
 p = multiprocessing.Process(target=download\_image,args=(image\_url,))  
 processes.append(p)  
   
 for process in processes:  
 process.start()  
   
 for process in processes:  
 process.join()  
  
 end = timeit.default\_timer()  
  
 print(f'Multiprocessing Execution Time: {end-start}')  
  
multiprocessing\_download()

*Multiprocessing Execution Time: 3.45567756*

it takes 3.4 seconds to complete the download in multiprocess mode.

*Note: The time can fluctuate between threading and multiprocessing*

**Use Cases**

**Use cases for threading in Python:**

1. Concurrent I/O Operations: Threading is useful when performing I/O-bound tasks, such as reading from or writing to files, making API requests, or interacting with databases. By using threads, you can overlap I/O operations and improve overall efficiency.
2. GUI Applications: Threading is commonly employed in graphical user interface (GUI) applications to keep the user interface responsive while performing background tasks. For example, you can use threads to update the user interface, handle user input, and perform computations simultaneously.
3. Network Operations: Threading is beneficial for network-related tasks, such as handling multiple client connections or implementing network servers. By assigning a separate thread to each connection or request, you can handle concurrent network operations efficiently.

**Use cases for multiprocessing in Python:**

1. CPU-Intensive Tasks: Multiprocessing is ideal for computationally intensive tasks that consume a significant amount of CPU resources. By utilizing multiple processes, you can distribute the workload across multiple cores or CPUs, thereby improving performance.
2. Data Processing and Analysis: When dealing with large datasets or performing complex data processing tasks, multiprocessing can significantly accelerate the processing time. By dividing the data or computations into smaller chunks and processing them in parallel, you can achieve faster data processing and analysis.
3. Scientific Computing and Simulations: Multiprocessing is commonly used in scientific computing and simulations, such as numerical computations, simulations, and mathematical modeling. By leveraging multiple processes, you can exploit the parallelism inherent in these tasks and reduce the execution time.

**Conclusion**

In conclusion, understanding the concepts of threading and multiprocessing in Python is essential for developing efficient and concurrent applications. Threading allows for concurrent execution of tasks within a single process, making it suitable for I/O-bound operations and GUI applications. On the other hand, multiprocessing enables parallel execution of tasks across multiple processes, making it beneficial for CPU-bound tasks and data-intensive operations.

Both threading and multiprocessing have their advantages and considerations. Threading offers lightweight concurrency but can face limitations due to the Global Interpreter Lock (GIL). Multiprocessing, although requiring more system resources, provides true parallelism and is well-suited for CPU-intensive tasks.

When choosing between threading and multiprocessing, it is crucial to assess the nature of the task, system requirements, and potential synchronization challenges. By understanding the strengths and drawbacks of each approach, developers can leverage threading and multiprocessing effectively to achieve optimal performance and responsiveness in their Python applications.

**Reference**

<https://www.geeksforgeeks.org/difference-between-multithreading-vs-multiprocessing-in-python/>

**Consider following me:**

[Arjun P | LinkedIn](https://www.linkedin.com/in/arjunprakash027/)

[arjunprakash027 (Arjun P) (github.com)](https://github.com/arjunprakash027)

[Arjun Prakash (@ArjunPrakash12) / Twitter](https://twitter.com/ArjunPrakash12)

**Check out my website**: [Arjun’s Portfolio (google.com)](https://sites.google.com/view/arjunresume/home)