Concurrencia vs Paralelismo

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).

Concurrency is like talking and drinking, you can switch back and forth but you cannot really do both at the same time. **Parallelism** is like walking and drinking, you can actually do both at the same time.

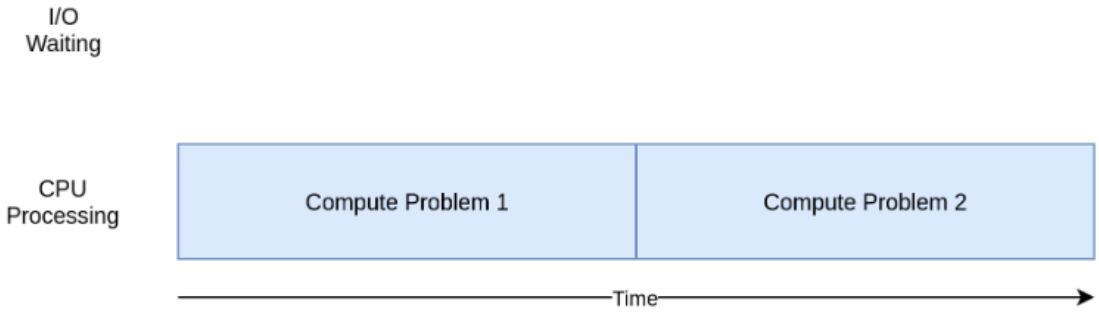
When you have only one cpu, you can only do concurrency. The overall time spent is the same as running in series. When you have multiple cpus, you can actually run things in parallel at the same time. The overall time spent can be a lot shorter.

It is important to highlight that both concurrency and parallelism could be combined during task execution.

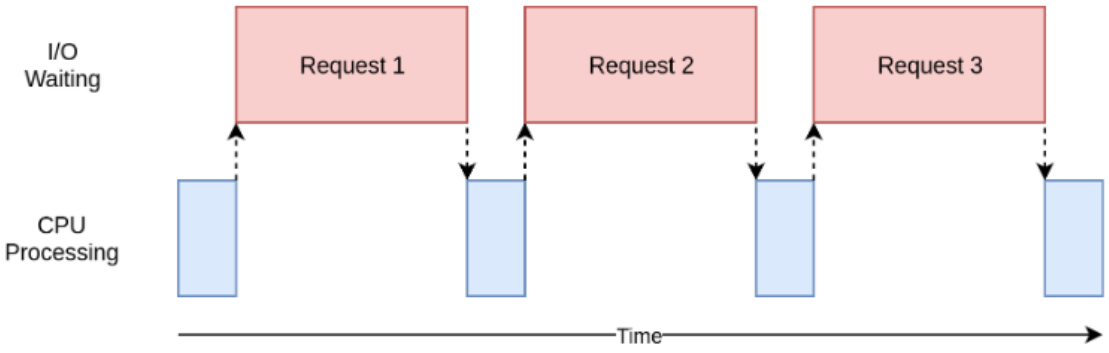
CPU-Bound VS I/O-Bound

The problems that our modern computers are trying to solve could be generally categorized as CPU-bound or I/O-bound. These concepts are universal for all programming languages.

CPU-bound refers to a condition when the time for it to complete the task is determined principally by the speed of the central processor. The faster clock-rate CPU we have, the higher performance of our program will have.



I/O bound refers to a condition when the time it takes to complete a computation is determined principally by the period spent waiting for input/output operations to be completed. This is the opposite of a task being CPU bound. Increasing CPU clock-rate will not increase the performance of our program significantly. On the contrary, if we have faster I/O, such as faster memory I/O, hard drive I/O, network I/O, etc, the performance of our program will boost.



Python Built-ins

Does Python have built-ins that facilitate us to build concurrent programs and enable them to run in parallel? *Jein*.

Why yes? Python does have built-in libraries for the most common concurrent programming constructs—multiprocessing and multithreading.

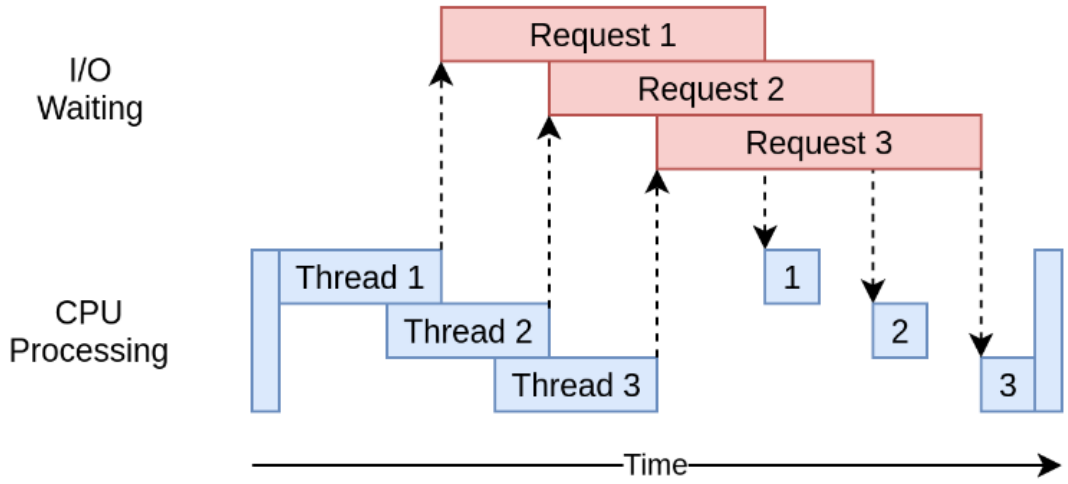
Why no? The reason is, multithreading in Python is not really multithreading, due to the **GIL** in Python.

Threading in Python

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.

A global interpreter lock (GIL) is a mechanism used in computer-language interpreters to synchronize the execution of threads so that only one native thread can execute at a time. An interpreter that uses GIL always allows exactly one native thread to execute at a time, even if run on a multi-core processor. Because Python interpreter uses GIL, a single-process Python program could only use one native thread during execution. Conventional compiled programming languages, such as C/C++, do not have interpreter and do not implement GIL. Therefore, for a single-process multi-thread C/C++ program, it could utilize many CPU cores and many native threads.

Therefore, for a CPU-bound task in Python, we would have to write multi-process Python program to maximize its performance.



However, this does not mean multi-thread is useless in Python. For a I/O-bound task in Python, multi-thread could be used to improve the program performance.

Using Python `threading`, we are able to make better use of the CPU sitting idle when waiting for the I/O. By overlapping the waiting time for requests, we are able to improve the performance.

Multiprocessing in Python

In principle, a multi-process Python program could fully utilize all the CPU cores and native threads available, by creating multiple Python interpreters on many native threads. Because all the processes are independent to each other, and they don't share memory.