

A word of advice

Before jumping into parallelization and code optimization, try to understand what you are trying to achieve, in comparison with what you've written. Is your code not clean enough? Perhaps untangling it and using of other libraries will help. Does your code make many read/write operations? Perhaps parallelizing in threads will speed it up. Does your code perform the same task over and over again, with minimal changes? Perhaps using parallelization in processes will aid you.

Perhaps, perhaps, perhaps... think about what you have just coded.

Concurrent processes

Main benefit	Parallelize CPU-bound tasks, independent from each other.
Import class	<pre>from concurrent.futures import ProcessPoolExecutor</pre>
Run parallel tasks	<pre>with ProcessPoolExecutor() as executor: results = executor.map((func, list))</pre>
Submit a specific task to a core	<pre>with ProcessPoolExecutor() as executor: results = executor.submit(func)</pre>
Option (number of cores)	<pre>ProcessPoolExecutor(max_ worker_s=10)</pre>

Extra function arguments can be Included before mapping the function in the executor. In order to do so, we need to create a partial version of the function with `partial_func = functools.partial(func, a=a, b=b, ...)`.

Concurrent threads

Main benefit	Parallelize I/O-bound tasks, independent from each other.
Import class	<pre>from concurrent.futures import ThreadPoolExecutor</pre>
Run parallel tasks	<pre>with ThreadPoolExecutor() as executor: results = executor.map((func, list))</pre>
Submit a specific task to a core	<pre>with ThreadPoolExecutor() as executor: results = executor.submit(func)</pre>

Concurrent threads (cont)

Option (number of cores)	<pre>ThreadPoolExecutor(max_ orkers=10)</pre>
--------------------------	---

Extra function arguments can be Included before mapping the function in the executor. In order to do so, we need to create a partial version of the function with `partial_func = functools.partial(func, a=a, b=b, ...)`.

Numba library

Main benefit	Useful decorators which compile the function the first time it is used, speeding it up in subsequent runs.
Importing decorators	<pre>from numba import jit, njit</pre>
Using jit	<pre>@jit def function(...): ...</pre>
Using njit	<pre>@njit def function(...): ...</pre>

This module is very limited, in the sense that only basic and/or numpy operations and classes can speed up. Neither scipy nor pandas or networkx can be improved. If `jit` is used, then those functions are treated as usual. If `njit` is used, then the code will fail to compile as it won't know what to do with them.

Caching



By **gonz95alo**
cheatography.com/gonz95alo/

Not published yet.
Last updated 3rd November, 2022.
Page 1 of 1.

Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish
Yours!
<https://apollopad.com>