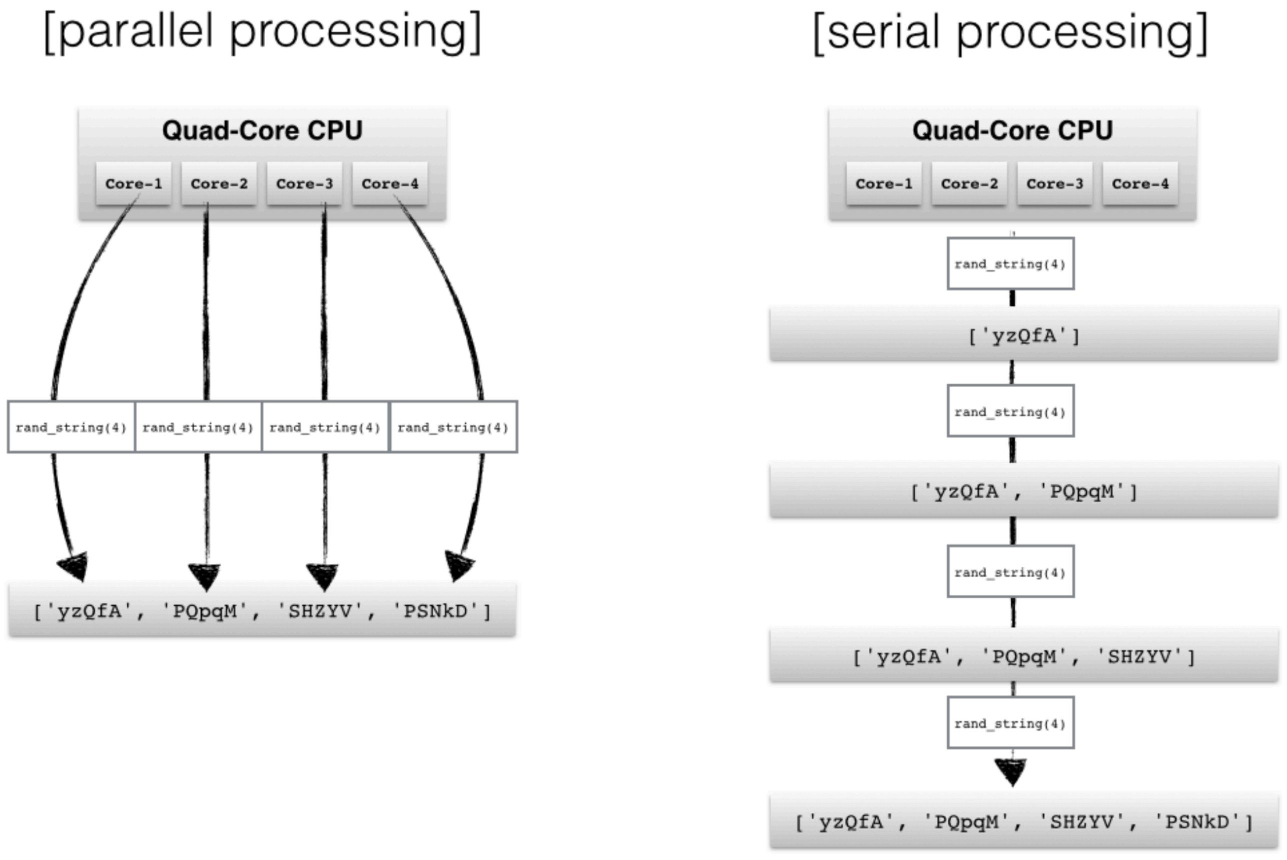
Parallel Programming in Python

CPUs with multiple cores have become the standard in the recent development of modern computer architectures and we can not only find them in supercomputer facilities but also in our desktop machines at home, and our laptops.
However, the default Python interpreter was designed with simplicity in mind and has a thread-safe mechanism, the so-called "GIL" (Global Interpreter Lock).
In order to prevent conflicts between threads, it executes only one statement at a time (so-called serial processing, or single-threading).

Multi-Threading vs. Multi-Processing

Depending on the application, two common approaches in parallel programming are either to run code via threads or multiple processes, respectively.
If we submit "jobs" to different threads, those jobs can be pictured as "sub-tasks" of a single process and those threads will usually have access to the same memory areas (i.e., shared memory).
This approach can easily lead to conflicts in case of improper synchronisation, for example, if processes are writing to the same memory location at the same time.



A safer approach (although it comes with an additional overhead due to the communication overhead between separate processes) is to submit multiple processes to completely separate memory locations (i.e., distributed memory): Every process will run completely independent from each other.
In Python's multiprocessing module and how we can use it to submit multiple processes that can run independently from each other in order to make best use of our CPU cores.

The Pool class

Another and more convenient approach for simple parallel processing tasks is provided by the Poolclass.

There are four methods that are:

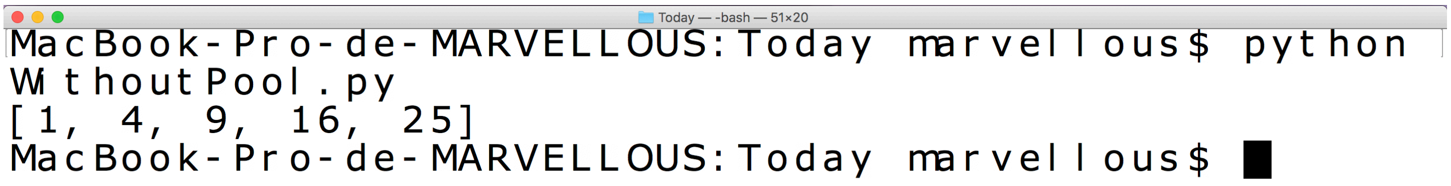
- Pool.apply
- Pool.map
- Pool.apply_async
- Pool.map_async

The Pool.apply and Pool.map methods are basically equivalents to Python's in-built apply and map functions.

Consider below program which uses serial processing approach

```
def square(n):
    return (n*n)

if __name__ == "__main__":

    # input list
    arr = [1,2,3,4,5]

    # empty list to store result
    result = []

    for num in arr:
        result.append(square(num))

    print(result)
```

Output of above application :



```
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python
WithoutPool.py
[1, 4, 9, 16, 25]
MacBook-Pro-de-MARVELLOUS:Today marvellous$ █
```

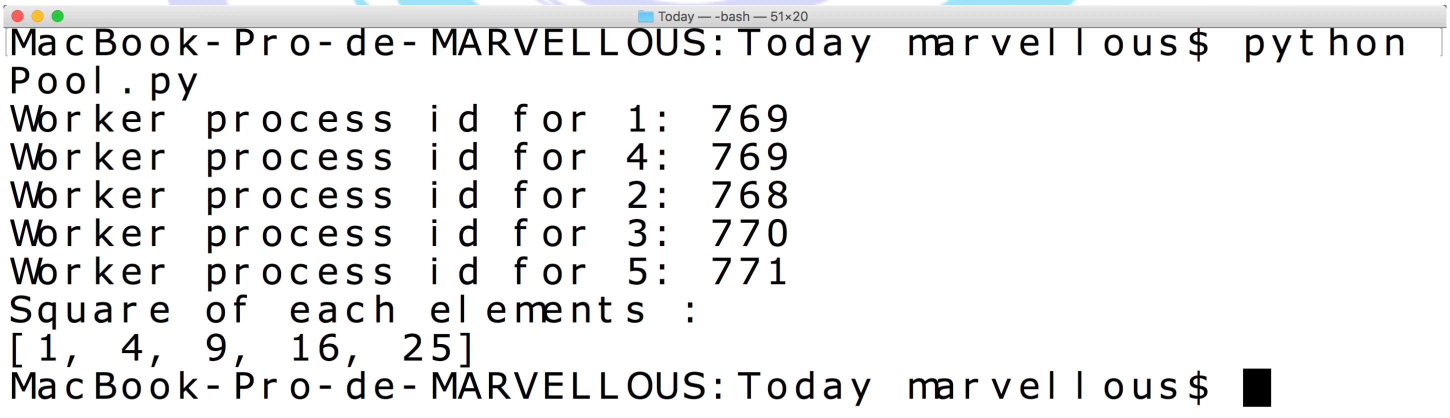
In above example for each number sequentially square function gets called.
In this application only single process gets executed on one core of our CPU.

To use power of multicore processor we use Pooling in Python.

Consider below program which uses Pooling for Parallel Programming

```
import multiprocessing
import os

def square(n):
    print("Worker process id for {0}: {1}".format(n, os.getpid()))
    return (n*n)

if __name__ == "__main__":
    # input list
    arr = [1,2,3,4,5]

    # creating a pool object
    p = multiprocessing.Pool()

    # map list to target function
    result = p.map(square, arr)

    print("Square of each elements : ")
    print(result)
```

Output of above application



```
MacBook-Pro-de-MARVELLOUS:Today marvellous$ python
Pool.py
Worker process id for 1: 769
Worker process id for 4: 769
Worker process id for 2: 768
Worker process id for 3: 770
Worker process id for 5: 771
Square of each elements :
[1, 4, 9, 16, 25]
MacBook-Pro-de-MARVELLOUS:Today marvellous$
```