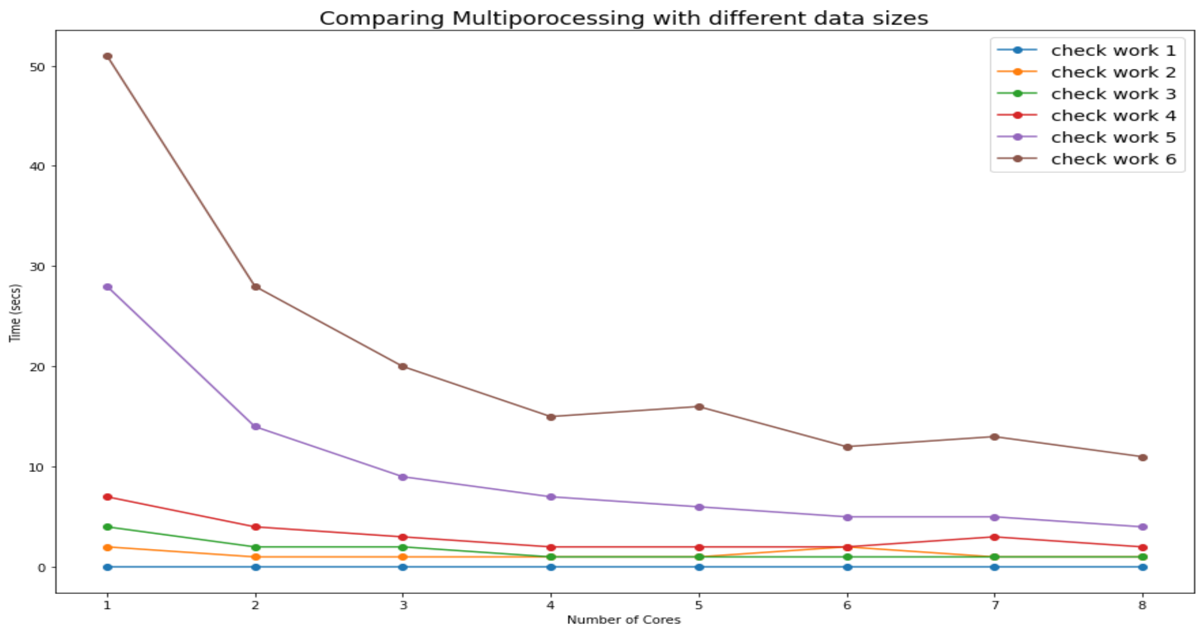
Introduction

Python language has a global interpreter lock which restricts multiple threads being sent to multiple cores we can get around this by using the multiprocessing module that allows the process to be split into subprocesses and sent to different cores for processing thus speeding up computational time. The multi processed code does not execute in the same order as its created and the first process created may not be the first to complete.

My computer is equipped with 8 cores and I will be testing how long various sets of work will take to compute using the check_prime function. There are 5 sets of work used to test this ranging between 1 element and 16 elements.

I have tested the multiprocessing module with 5 different datasets of various sizes starting from one eight digit prime up to an array with 16 eight-digit arrays. I timed each function call and how long it took using between 1- 8 cores. The y-axis is the time it took in seconds to compute the computation to see if the number was a prime number or not. The x-axis is the number of cores used for the computation.



Check work 1[1 element]

It is of no benefit to allocate more than one core to this task it cannot be completed any quicker by allocating more computational resources to it.

Check work 2[2 elements]

2 cores provide ample computational power to anymore and and the resourses are superflous.

Check work 3[4 elements]

This works much the same way as the previous two in that any more cores than elements is unnecessary as four cores can complete the task just as quickly as allocating more to the task.

Check work 4[8 elements]

Using 8 cores for 8 elements each element has its own thread to do the computaion and so each are done asynchronous and it takes 1 second to return the results a large difference from computating with just one core where it takes 7 secs to run the function. You would think that it would have taken 8 secsonds with one core based on how all 8 took one second each with 8 cores but this fails to account for the overhead in creating multiple threads which doesn't happen with one core.

Check work 5[16 elements]

Whats interesting here is that using 6 cores produce the same results as using 8 cores

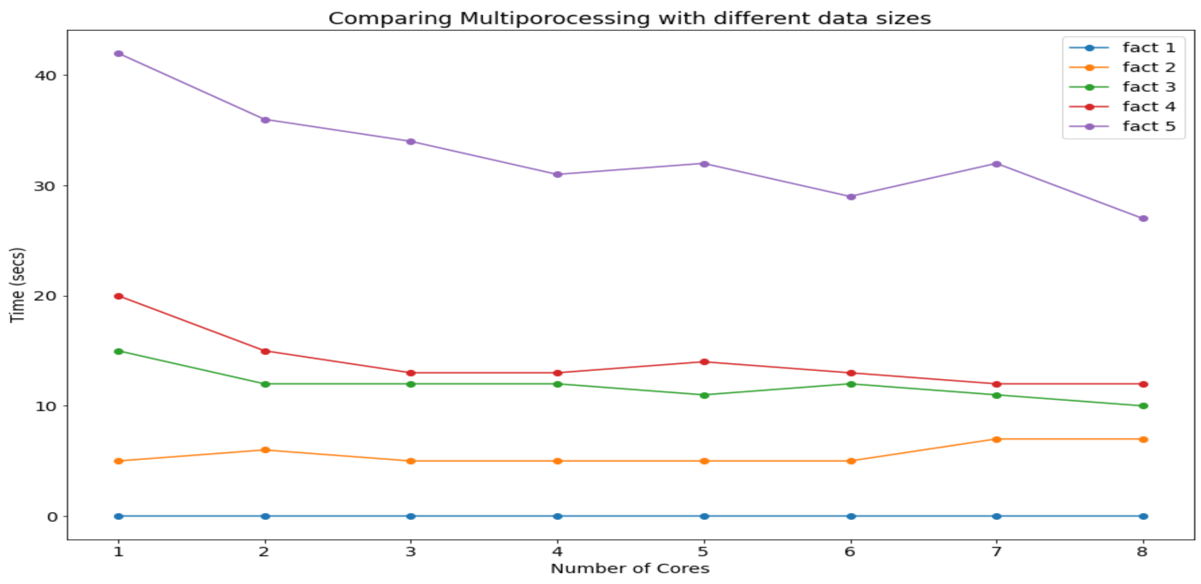
Check work 6[32 elements]

It is following much the same pattern as check work 4 and 5.

From check_work3 to check_work6 we can observe that once we hit four cores operating there is a considerable slow down of time speed up in comparison from one core to two cores.

Q2 a

We can see with implementation of the second function to caluculate the factorials of 5 didgit numbers that the increase in cores acts much the same way as with the checking_prime function. After 2 cores are utilised the increase in computation time falls with each core added. Once we engage four cores the increase in computational time is negligable. In the cases where we are adding more computational power than is need we even see the computational time go up slighlty like in fact2 and fact3, due to the overhead of the extra subprocesses being creted that we don't need.



Conclusion

I think what we can surmise form these evaluations is that it is not always beneficial to increase computational power to solve a problem as it will not always achieve the desired results. We need to make sure that the task is not dependent on any other task that is being parallelised. Each task should run independently, and it should not matter what order the processes are completed. The function should not return a value that is later relied upon further in the code. One method of assessing the benefit of parallelising code is Amdahl's law which allows us to calculate the theoretical speed up by using multiprocessing against using a serialised one core approach. Of which I have provided a sample calculation in the Jupyter notebook. This shows a theoretical speed up of 4.26 times when using eight cores over one. But we must be mindful that it is only theoretical and that many factors can affect performance for example the speed of memory, CPU cache memory, disks, and network cards. A parallel algorithm may not always be the most suitable sometimes an optimised serial algorithm may suit better as it does not have the overhead cost of splitting up the computation and then reassembling it.