26/04/2

Multiprocessing

COMPUTER ARCHITECTURE ORGANISATION

MIGUEL MARTINEZ - 00021466
RAFAEL MARTINS - 18200630

UNIVERSITY COLLEGE DUBLIN | MSc Computer Science (Conversion)

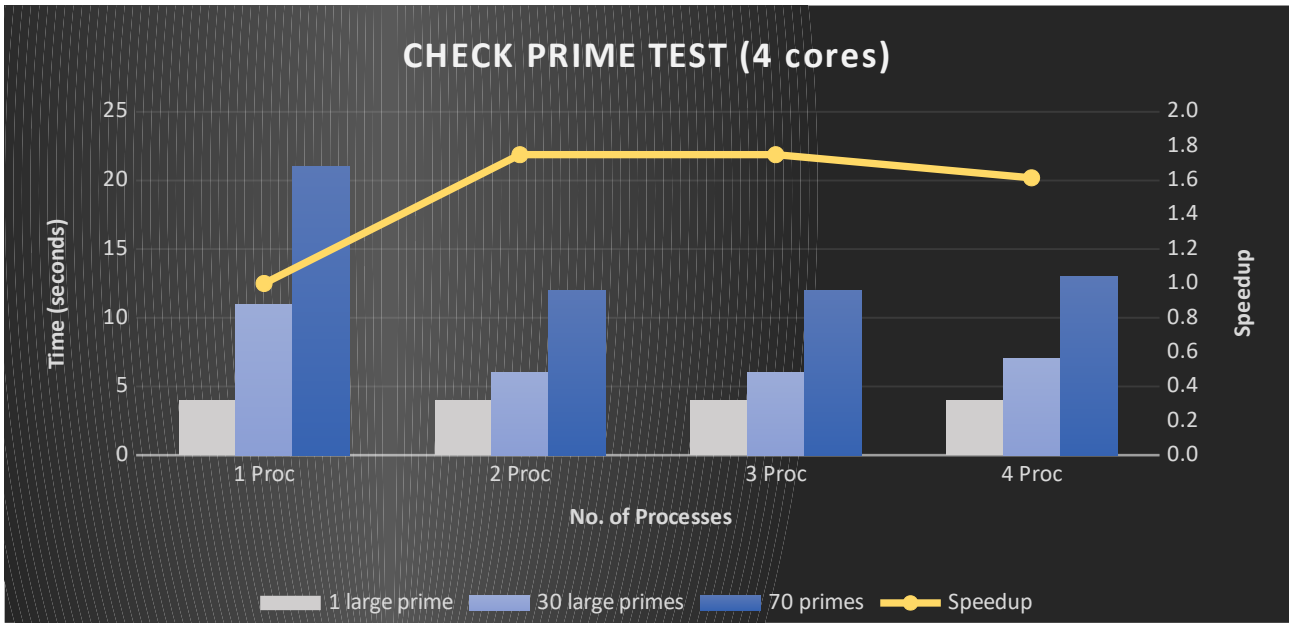
OVERVIEW

The objective of this exercise is to evaluate speedup derived from using multiple CPU cores through the multiprocessing facility in Python, which has a framework called Pool that enables the assessment of how multiprocessing impacts the runtime on a multi-core machine. In order to accomplish that, we used two different processing tasks to test the CPU as we incremented the number of cores available:

- Check Prime, a function that checks if a number is prime;
- GCD, a function that finds the greatest and smallest numbers in a list and returns the greatest common divisor between them.

CHECK PRIME

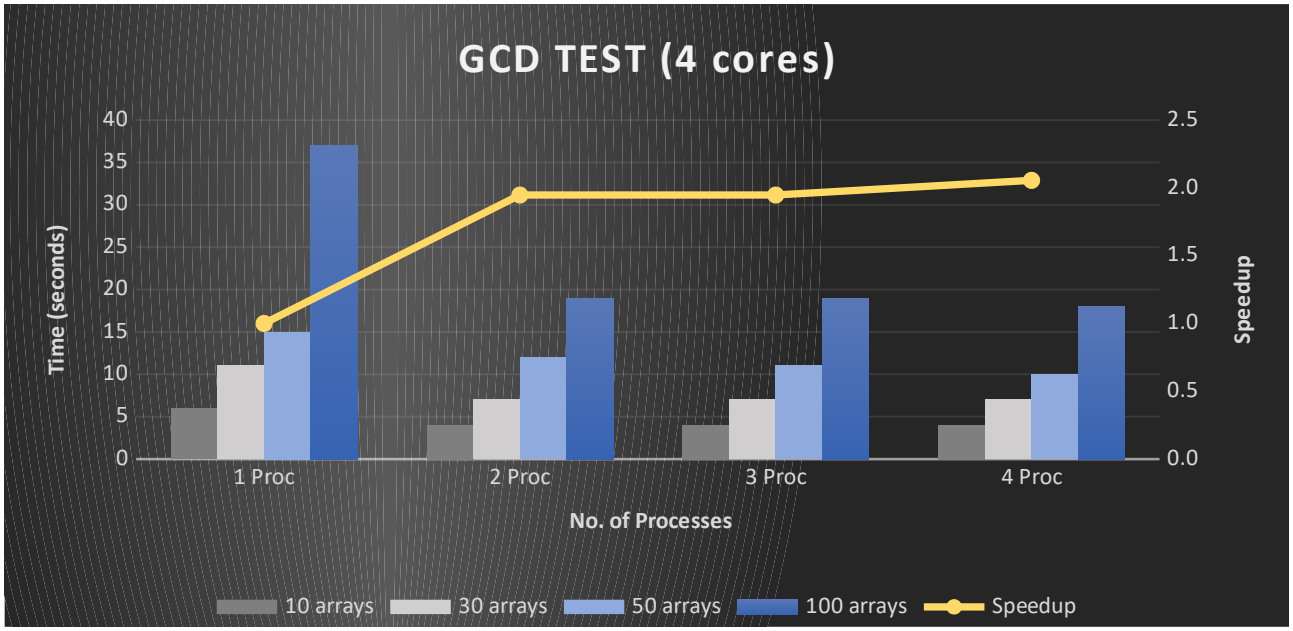
After connecting the "check prime" function to the Pool processing function, we tested it with three different input sizes to quantify the speedup achieved with multiple cores. The idea is to, for each input size, assess the performance achieved by 1, 2, 3 and 4 cores. The different inputs were 1, 30 and 70 prime numbers.



From this data, we have witnessed that by using smaller sets of data the time curve shows general improvement from 1 worker in the pool up to 4 workers. This contrasts to the use of bigger sets of data that show an initial improvement of computation time mainly from and including 1 and 2 cores. However, from 3 cores onwards, the performance seems to be averaging a similar time and, therefore, invalidating the use of more workers in order to gain time efficiency.

GCD

We then executed the same performance test using a different function related to integer numbers: the aim is to ask the machine to find the maximum, the minimum in each sequence (list) contained in an array and return the greatest common divisor between each pair. Each sequence is made by a custom number of randomly generated integers. The different inputs were 10, 30, 50 and 100 lists.



As we look into the number of lists that we pass into the function in question to compute, we could see that using a single list showed no improvement when using more workers as that list can only be used by one worker. However, if greater than one, number of lists to compute would be divided into the number of workers and, therefore, result in a time performance gain. Nevertheless, after trying multiprocessing using 3 or more cores, the speedup seems to stabilise in the same way as it happened during the first test.

CONCLUSION

After reflecting on the results, we have based a sizeable amount of time in researching the way that Python 3 multiprocessing operates. We were especially curious about the performance as the computer that we were using to test the performance of the algorithm is made up of 2 cores and 4 threads. We wanted to assess whether or not the four threads were going to be employed when adding 4 workers to the pool. In order to test the use of threading when multiprocessing we tried to use ThreadPool and then decided not to progress within this path as we felt we were deviating from the aim of the assignment.

This exercise has proven quite revealing and explanatory in nature and we believe that it has served us well in learning about the advantages of multiprocessing. We have also learned that due to the code itself, the task at hand being multi processed may or may not lend itself to be faster. This is so, as a part of an algorithm may not able to be used by multiple cores at the same time while another part of the code may be. Depending on how much of the code is able to be multi processed or not the addition of further cores to the computation of the algorithm will impact more or less the final time that it takes the algorithm to return an output. It is important to note that communication between many processes when computing small inputs or tasks can even result on worse performance.