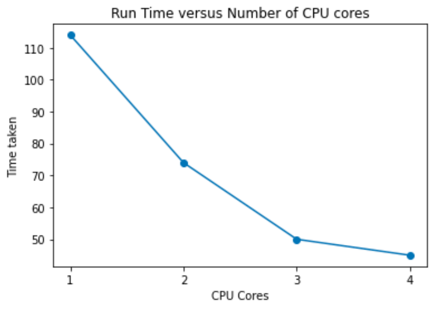
Conor Kennedy – 16722649

The objective of this assignment is to investigate the run times of different processing tasks on a different number of CPU cores by importing the multiprocessing module in Python. The first section of this report looks at the given `check_prime()` and the second part examines the results from the implemented `factorial_iterative()` function. Conclusions and observations have been made in section 3 of this report.

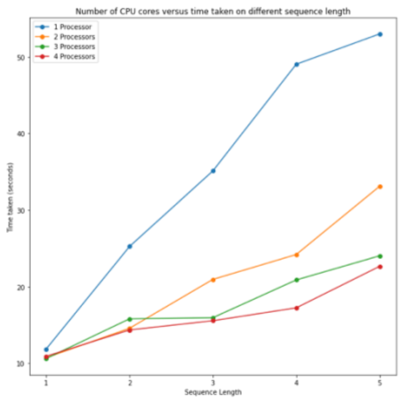
Section 1 – Check Prime Function Results

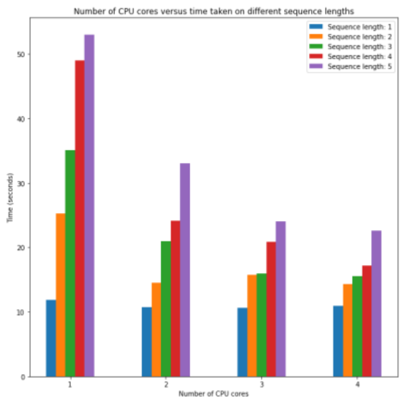
Firstly, the `check_prime()` function was carried out on the variable `numbers_list` which contained 10 eight digit prime numbers. The times taken were as follows:

Number of Cores	Time taken
1	114 seconds
2	74 seconds
3	50 seconds
4	45 seconds



Further analysis was carried out to see the differences in time taken when the sequence length changes. For this analysis, a shortened list of 5 prime numbers was used.

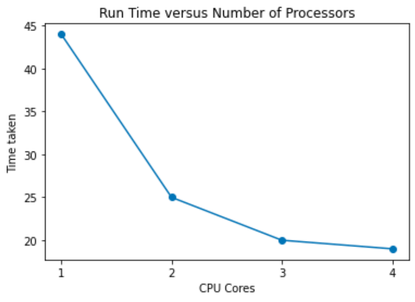




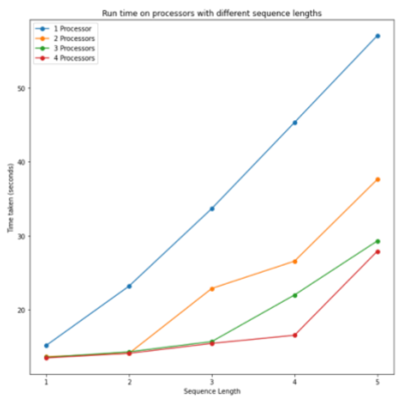
Section 2 – Factorial Function Results

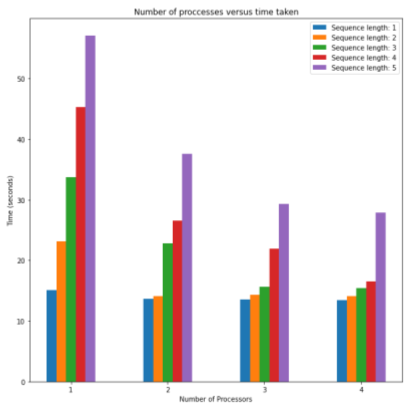
In order to analyse the time taken for the factorial to be computed with the `factorial_iterative()` function, five random numbers in the range 10,000 to 99999 have been used. The times taken using different number of cores are shown below:

Number of Cores	Time taken
1	44 seconds
2	25 seconds
3	20 seconds
4	19 seconds



Again, similar to section 1, analysis was carried out on the processing times on different sequence lengths. The following graphs show the results:





<u>**Section 3 – Analysis of Results**</u>

The Global Interpreter Lock mechanism used in Python assures that only one thread executes a Python bytecode at a time. By importing the Multiprocessing package which allows programmes to leverage multiple processors, it is clear that as the number of cores increases, the time taken for the process to complete reduces at a diminishing rate. As the results/graphs for both functions are very similar, analysis will only be completed on the `check_prime()` function but also applies to Section 2's analysis of the `iterative_function()` runtimes.

As we can see from the first table and graph in Section 1, the time taken has reduced as the number of CPU cores involved in the process increases. It is important to note that there is not an absolute linear speed-up with the number of cores used. This is shown in the graph where it is clear that the line is going down more gradually being cores 3 & 4 than the steep decrease between cores 1 & 2. In other words, doubling the number of cores does not double a computer's speed. This could be down to the CPUs having to communicate with one another that it uses up the speed.

The second part looked at varying the sequence length for the `check_prime()` function to work on. It is worth noting that when the sequence length was one digit, the process time appears to have been the same regardless of the number of cores being used. This aligns with Amdahl's law which says that in a program with parallel processing, adding more processors may not make a program run faster if there are only a few instructions to be performed in sequence. As the sequence increases in length, there is a clear difference in the time-taken for the process to complete between each of the cores - the time taken again decreases drastically and then levels off as the number of cores increase.

Further analysis could be carried out by increasing the sequence lengths used or finding the average over a number of iterations. These were not carried out during this analysis due to issues with processing times. Overall, this analysis was insightful and showed that as the processes used more CPU cores, the time taken decreased exponentially.