Multiprocessing

Assignment 1

for COMP30660 – Computer Architecture & Organization

Prepared by **Ednalva Oliveira – Student Number 19732291**

and

Ester Nunes McGarry – Student Number 19205519

School of Computer Science – UCD

May 2020

Part I

The aim of this project is to analyse how parallel computing works with phyton. In python, the multiprocessing module allowed us to divide a task into parts and solve each part in parallel, so the time to complete the task could be reduced. As phyton tends to use only one processor, we used Pool class to go around the Global Interpreter Lock and perform a computational task using 12 processors (within a single computer) working simultaneously on the same job.

Python multiprocessing was tested by connecting the given check_prime function to the Pool processing function and quantify the speedup achieved with multiple cores. As seen in the code and graph (on notebook file), we run the code and achieved the expected result of finish the task in less time that with only one processor. The following considerations were observed:

The efficiency: Observing the graph we were able to visualize the improvement by using multiple processes to run one single task. As the number of processors used to run the program increased, the time spent to perform the sets of work was reduced. So, the programme could take advantage of all the cores in the computer. Another important observation is that because each task was entirely independent of each other, we could run the programme and gradually improve the speedup rate until up to 12 processors.

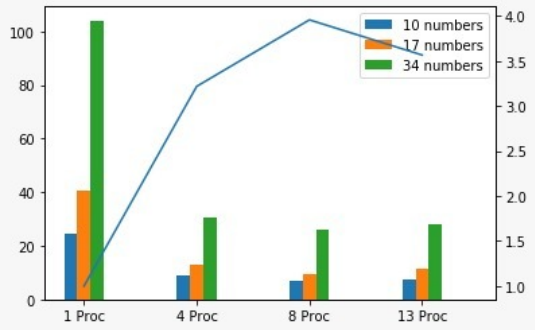
Hardware limitation: After running with 12 processors we noticed that any other processor added to the pool would not bring any benefits to the programme in terms completion time or performance. Considering the overhead of creating an extra processor, the effect generated is actually the opposite, the overall performance of the code was reduced. This shows that the highest performance achieved is running with maximum of 12 processors. Therefore, in order to successfully achieve an efficient parallel computing, the hardware limitations and the independence of each task had to be considered in this case.

Another relevant aspect was brought into question while working on this project. We wanted to check if running the same function with the same input multiple times, would occur any difference in the speedup overtime. As we though the performance of the programme could be affected by other applications using the CPU while we ran the code.

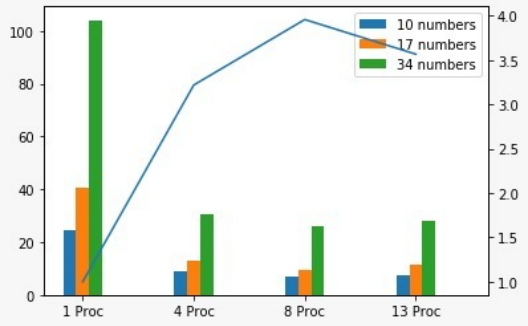
So, we decide to run the Pool processing function 15 times and take the average of the overall time and then compute the speedup. However, it was verified that there was no difference if running once or multiple times, the graph remained identical as running the function only once.

Result:

Running 1x



Running 15 x and plotting the average time



Although the second graph did not show any considerable improvement or inefficiency. It was significant to experiment a different approach to test if we would get different results

Part II

For this part of the project, we chose to modify the *check_prime* function to process the task in a different way. We separated the work the function itself is doing. We found interesting results running this program.

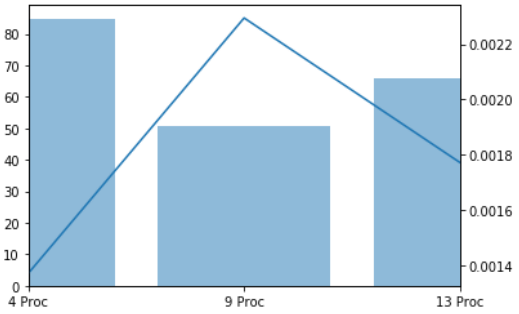
In the first part of this project one processor is checking one of the numbers in an array, which makes the task faster to conclude.

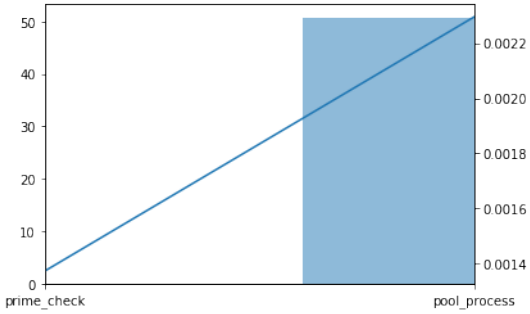
In this section of our project, we created a function called *check_prime_multi*, which is an adaptation of *check_prime*. This function will allow us to send only one number to be checked if it is prime or not, but the work of checking will be divided into different processors. The program will take a number, divide the number in different ranges depending on the number of processors. Each processor will check a different range. As an example, let's use the number 17 as an example and let's say we are using 4 processors.

- The range (2,17) will be divided into 4 ranges (as we are using 4 processors):
  - (2,4) – processor 1 will check if 17 modulo each number of this range is 0.
  - (4,8) – processor 2 will check if 17 modulo each number of this range is 0.
  - (8,13) – processor 3 will check if 17 modulo each number of this range is 0.
  - (13,17) – processor 4 will check if 17 modulo each number of this range is 0.

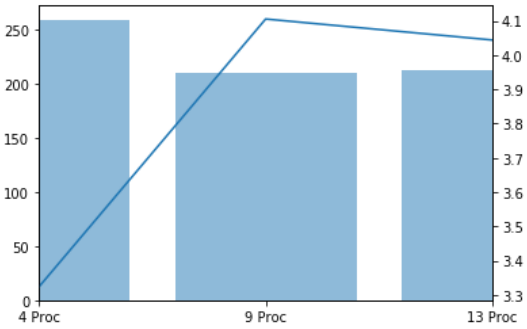
We ran the program with a prime number and then we ran it again with a non-prime number.

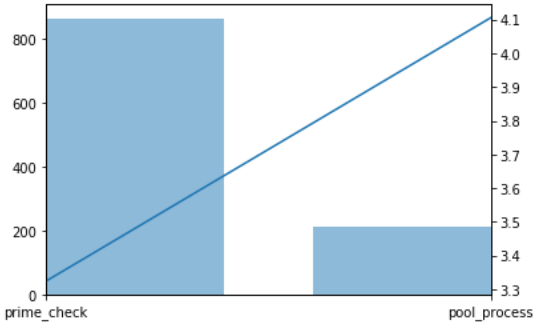
- 10-digit prime number:
  - The first graph shows improvement in speed as the number of processors gets higher. When the number of processors in our *pool_process* is higher than the number of processors in our computer, then the speed either plateaus or reduces. This means it takes less time to find out if the number is prime when we have multiple processors.
  - The second graph shows the running time when using only *check_prime* function by itself vs the running time using the *pool_process*. We can see that the *pool_process* runs a lot faster than the *check_prime* by itself, which is expected judging by the results we get from the first graph.





- 10-digit non-prime number:
  - The first graph shows the same improvements as we see when checking a non-prime number – the higher the number of processors in the *pool_process* (taking into consideration hardware limitation as mentioned above), then faster the program runs.
  - However, in the second graph we can see that running the function *check_prime* by itself is a lot faster than running the *pool_proce*ss, even though within the *pool_process* we get faster running times for multiple processors.





To summarise, running a prime number in multiple processors is faster than running it in one processor and running a non-prime number in one processor is faster than running it in multiple processors. This is to be expected, since when working with a non-prime, the program will stop running as soon as it finds a number that is divisible by the initial non-prime number when using one processor, but this is not true if using multiple processors.

We can then conclude that the decision of running a program in one processor or in multiple processors will depend on what the program was designed to do:

- If we need a program that runs an array of numbers to find out if the each number is prime or not, then using multiple processors will return the results faster.
- If we need a program that runs one number to find if that number is prime or not, then there is further analysis required. There are 78498 prime numbers that are smaller than 1 million, meaning less than 8% of numbers under a million are prime.
  - If we know there is a high probability that the number we are running is prime, then we should use multiple processors, as this will be quicker.
  - If we know there is a low probability that the number we are running is not prime, then we should use one processor, as this will be quicker.

To finish, we can outline some of the main lessons learned about parallel computing through this programme. To use multiprocessing, the solution to a problem must be designed with tasks that can run independently from another parallelised task. Another relevant aspect is the limit of parallel computing reached by hardware limitation of the machine.

In conclusion, this projects makes it clear that even though some programs might run faster if run in multiple processor, that is not true for every programme and before making the decision of using one or multiple processors, there needs to be further analysis of the program as well as testing similar to what we have done in this project.