Computer Architecture Assignment 1: Multiprocessing

Introduction

In this assignment I investigate the effect of multiprocessing on execution time compared to sequential processing. In order to do this, I have used a function called *check_prime* which determines if the input is a prime number. This particular function was chosen because it takes a significant amount of time to complete execution with a large enough known prime input (e.g. input 1548801 takes approx. 2 seconds to resolve). This makes execution times easily comparable. I also have a *pool_process* function that takes as arguments: another function (in our case, check_prime), the data to be passed to the latter function and the pool size. The pool size determines the number of processes to be executed at one time, i.e. how many cores to utilize. My machine has 2 cores, and so pool size was set to a maximum value of 2. Any larger value than this would make little sense because no more than 2 processes can possibly be executed at one time.

I use the multiprocessing Python library to create a Pool object - a 'pool' of processes. This pool has a map function executes the processes in the pool in *parallel,* and preventing the rest of the program from executing until our pool processes have completed.

Task 1

In order to investigate the performance and benefit of multiprocessing with 2 cores, I first looked at increasing the input dataset of the program to be executed *by 1* with each loop. Figure 1 shows the result of this experiment. We can see that the speedup varies between **1.46** and **1.76**, with a mean speedup coefficient of **1.60**. We can also notice that there are often dips in speedup when the number of processes to execute is *odd*. This is expected because we are assuming that each process takes an equal amount of time, meaning that when an odd number of processes need to be executed, the very last execution does not benefit from parallelism causing the speedup to be lower.

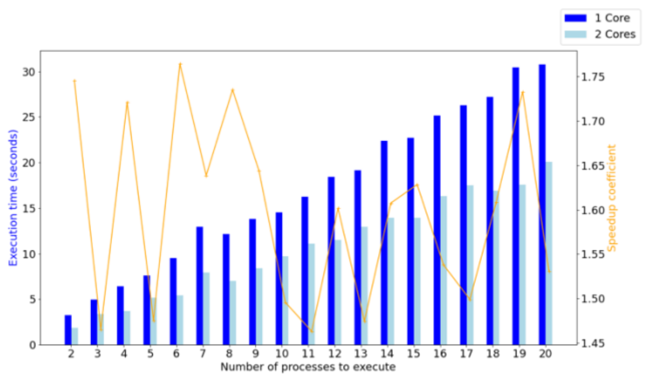

Figure 1. Running times of increasing datasets of prime numbers to be checked by the check_prime() function

Therefore, in order to optimise this, I ran a similar test, but that this time only used even datasets. This would supposedly increase overall speedup because the processes would be synchronised to run optimally on 2 cores, showing the true potential of parallelism. In Figure 2 we can see the results from this second test. We make the assumption that each check_prime function call, or each process, has equal running time based on the fact that each call uses the exact same program and dataset - the only difference is the prime number to checked. The prime numbers are large but succeed each other closely, and therefore I have assumed that they take very similar amount of CPU time to execute. However, as we can see on the graph in Figure 2, the speedup coefficient varies between **1.06** and **1.62** with different number of processes to be executed, despite them all being even. This is likely due to the fact that each check_prime call does *not* take an equal amount of time. Running time can also be affected by various factors other than parallelism that are occurring in the background which we cannot necessarily see or control. The mean speedup time for this test was found to be **1.62** over 20 different dataset sizes, which is not significantly more than the initial experiment, but we can see that there are no longer the clear dips in speedup for the odd-length datasets. This experiment suggest that although in theory the processes are fully parallelizable (they are completely independent of each other), this does not result in a speedup of 2x. However, the parallelisation of the processes has still significantly reduced the overall time of execution.

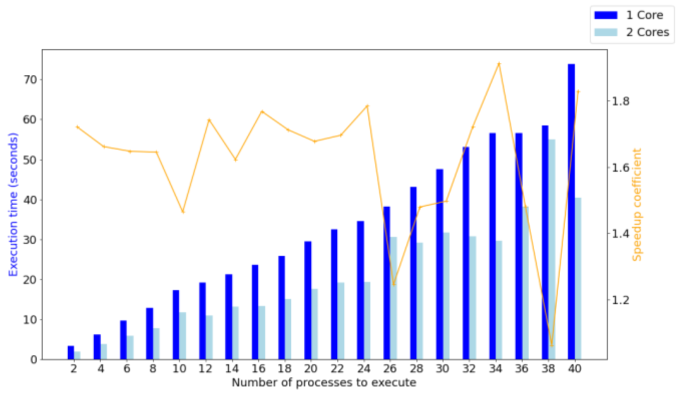

Figure 2. Running times of even numbers of function calls of check_prime() function

Task 2

In order to investigate this further, I have designed a test that controls and predefines the amount of time a function takes to run. This removes the assumption of the check_prime calls taking the same amount of time, and replaces it with a function of predetermined running time. For this, I have created a function called *sleep*, that does nothing but uses the time module to sleep for a number of seconds provided by the argument. This allows us to control the amount of time a process takes to be completed, and in turn I can design an optimal usage of multiprocessing (cf. Figure 3). I run an initial base case test in a similar way to Task 1, with increasing the dataset by 1, which would *not* be an optimal use of 2 cores. Figure 4. shows the results of this test, where the speedup times appear to plateau at each odd-sized dataset (at 9 onwards, this pattern appears to fall through). The mean speedup is found to be **1.8** times faster with 2 cores than with 1. This is significantly higher than the experiment conducted in Task 1, which suggests that the running times of the check_prime function could have been responsible for the lower-than-expected speedup coefficients.

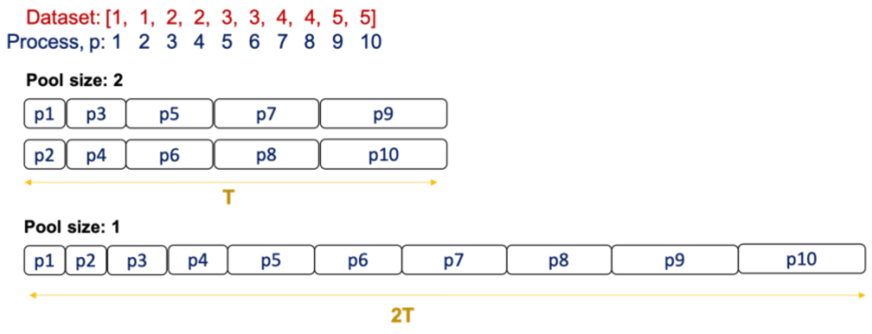
Dataset: [1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
Process, p: 1 2 3 4 5 6 7 8 9 10
Pool size: 2
p1 p3 p5 p7 p9
p2 p4 p6 p8 p10
T
Pool size: 1
p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
2T

Figure 3. Diagram showing a simplification of how the processes should run optimally with 2 cores (parallelism) vs. with 1 (sequential execution). T is the total time to execute the dataset with 2 cores.

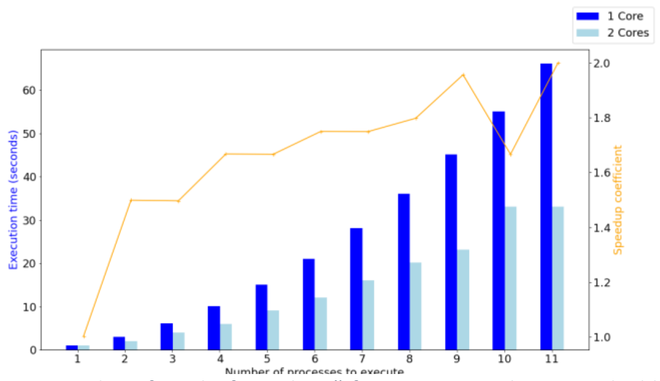

Figure 4. Speedup of results from sleep() function test with even and odd dataset lengths.

Firstly, the initial data set contains 2 equal numbers: [1, 1]. Once these are pooled and mapped, they represent 2 processes to be executed, ie. 2 instances of the sleep function being called with argument: 1 second. This is performed with a pool size of 1 and a pool size of 2 in order to compare the execution times of the dataset. Once this is completed, I continue the investigation by increasing the size of the dataset: I append 2 new *equal* numbers to the list, becoming [1, 1, 2, 2], and perform the same comparison between pool sizes. This is repeated until I have a list of arbitrary length 12, that is 5 different arguments passed to the sleep function.

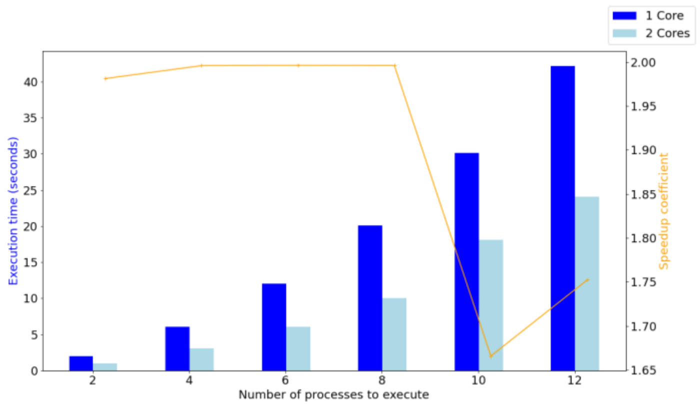

Figure 5. Speedup of results from sleep() function test with only even dataset lengths.

I can therefore anticipate/predict that the speedup should be a factor of 2 in this experiment. Figure 5 shows the result of this test, where we can see that the speedup coefficient for each dataset size is significantly higher – we find a mean speedup of **1.9**, where the minimum speedup was **1.67** and the maximum **2.0**. This mean value is very close to the theoretical speedup of 2 predicted in Figure 3. This suggests that the benefit of parallelising the execution of a program is highly dependent on the timing of each process in execution, and the way they are or aren't synchronised. A slow process can hold up one of the cores and lead to a significant decrease in the overall speedup achieved from multiprocessing the task. The sleep function shows an interesting 'optimal' use of multiprocessing, where the speedup in practice is achieved by controlling the running time and coordination of processes in execution.