**Computer Architecture Assignment 1: Multiprocessing**

**Introduction**

In this assignment we investigate the effect of multiprocessing on execution time compared to sequential/serial processing. In order to do this, we 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 with a large enough known prime input (eg. input 1548801 takes roughly 2 seconds to resolve). This makes execution times easily comparable.

We 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, ie. 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.

We 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 are completed.

**Task 1**

In order to investigate the speedup achieved with multiple cores, we can run the pool\_process function with pool size set to 1. This ensures that the functions are being timed in the exact same way, except that 1 process is in execution at any one time rather than 2. A data range of length 20 is used (ie. check\_prime function is executed 20 times), with input a the constant value of 15488801. We can then compare the total execution time with 1 core and with 2 cores.

By increasing the number of prime numbers in the data set to be checked, we increase the number of processes to be executed. With a max pool size of 2, we can look at how the speedup time performs with a range of data set lengths. Figure 1 is a graph that allows us to see the variation in speedup time with different dataset lengths. We find that the max speedup time is 2 times faster with 2 cores than with 1. The smallest speedup was 1.3 times faster with 2 cores than with 1. We can see in this graph that all speedup values (calculated by dividing the 2-core time by the 1-core time) fall within a range of 0.68 and have a standard deviation of 0.17. This latter value is low, meaning that the speedup times calculated are fairly clustered around the mean of 1.64. This is visible on the graph, but we can also plot the linear regression (Figure 2) to find that the fitted line is has a gradient very close to zero (-0.0016, to 3 significant figures).

Chart, line chart

Description automatically generatedThis indicates that the time taken to execute N processes speeds up by a constant factor of approximately 1.6 when using 2 cores as opposed to 1.

However, these results could have been affected by [XXX].

**Task 2**

As an extension/alternative to this test, we have designed a test that controls and predefines the amount of time a function takes to run. For this, we 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 we can design an optimal usage of multiprocessing. Firstly, the initial data set contains 2 equal numbers: [1, 1]. Once these are pooled and mapped, they represent 2 processes to 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, we continue the investigation by increasing the size of the dataset: we 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 we have a list of arbitrary length 10, that is 5 different arguments passed to the sleep function. This progression is visualised for both pool sizes in figure 2 below.

Figure 1. Speedup with increasing data set size (number of prime numbers to be checked by check\_prime function), and regression line plotted using SciPy library.

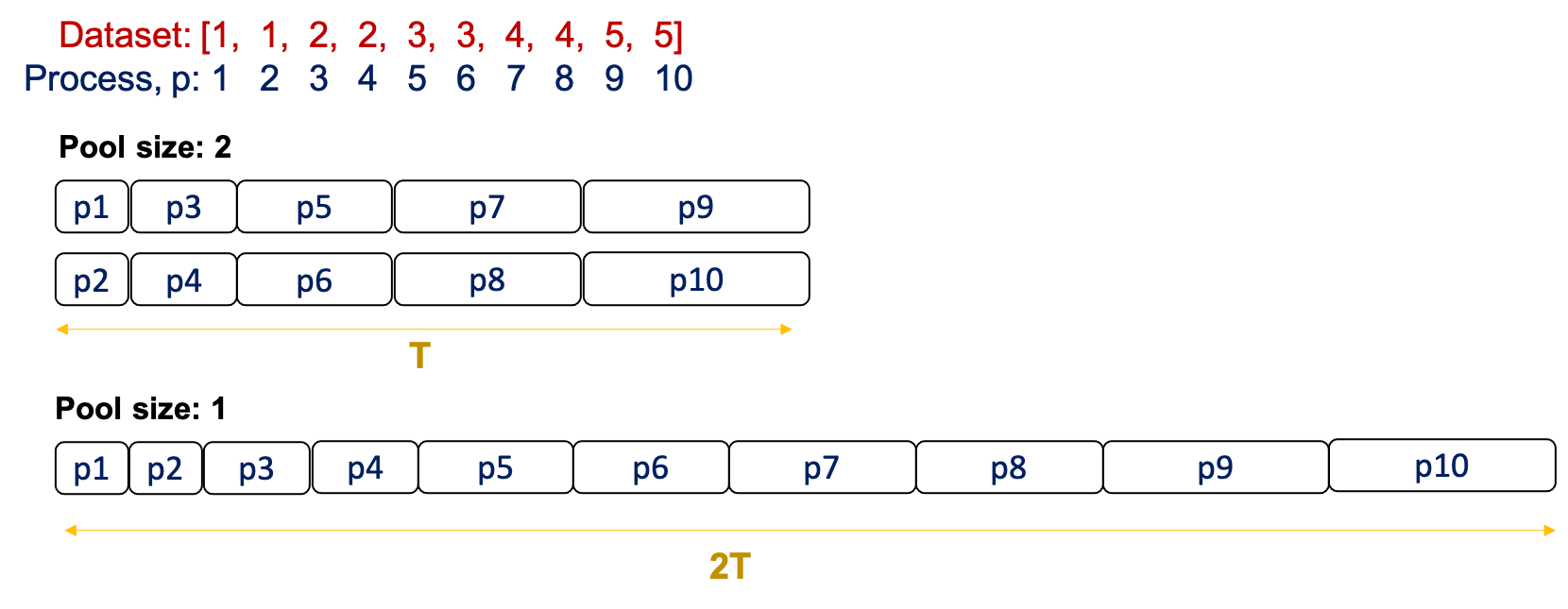


Figure 2. Visualisation of the processes being executed for both pool sizes. Pool size 2 shows the effect of parallelism, while pool size 1 shows the sequential execution. T is a the total time of execution of the dataset.

This prediction implies that the speedup should be a factor of 2 in this experiment. Figure 3 concurs with this prediction, showing a constant (gradient approx. 0) relationship with a speedup of approximately 1.98.

Chart, line chart

Description automatically generated