COMP30660

Computer Architecture & Organisation

Assignment 1

Multiprocessing

Hassan Albujasim, Stefan Bordei

Task1:

“You have 1 room that can fit 2 people working at once but you hired 10 to paint the room.”

We started by using `os.cpu_count()` in order to get the number of cores that our machine has. We know that we have 4 physical cores and 8 logical ones. Using `os.sched_getaffinity(pidOfProcess)` shows us that we could run python on all of the 8 logical cores (4 physical ones).

The system was tuned for benchmarking in Python by using `pyperf system tune` (on Linux).

We have decided to use a list of large prime numbers as an input for the `check_prime` function in order to get a longer runtime of the function (as instructed).

After running the `check_prime` function a couple of times we saw that the runtimes we were getting were varying so we decided to run the `check_prime` with a benchmarking function to get a running average of 10 runs to compensate for context switching and other things running in the background.

We decided to quantify the results of running the `check_prime` function with varying pool sizes by plotting the data obtained in order to visually identify the changes that occur.

In order to be able to plot the results we added a return statement to the `pool_process` function that returns the run time and we also removed all the print statements.

We ran the `prime_check` function on the `check_work` array (containing 7 large prime numbers) with different pool sizes (1 to 14) in order to better understand the impact of the pool size on the time it takes the function to complete. Due to the fact that our machine has only 4 cores we saw a direct correlation between the pool size and the running speed of the function up to a pool size of 4.

Observations:

The difference between a pool size of 1 and 2 is most noticeable and after that increasing the pool size starts to have a smaller effect on the speed of the process.

Further we decided to check the correlation between the pool size and the size of the input array in order to get a better understanding of the way the processes are parallelised.

As expected, the pool size did have a noticeable impact on the speed of execution as multiple processes could be run at, almost, the same time.

When plotting the results we discovered that the pool size and the size of the input array are directly proportional in regards to the amount of time it takes to run the function. For example, running the `check_prime` function with a pool size of 1 on multiple arrays, increasing the size of the array each time by one, showed a constant increase in the run time for each array size. Running the same function but increasing the pool size to 2 showed an increase in the run time of the process that was ½ of the increase of using a pool size of 1 as the 2 processes could be run in parallel taking almost ½ the time.

Task 2:

We decided to use a factorial function for task 2 and map it to a range of 0 to 10000 in order to force Python to use Big Numbers that would impact the total time required to run the function.

In order to cut down on the running time we did not use the benchmarking function for task 2 as we were expecting similar results as in Task 1 and we were able to identify and interpret the small variations that occurred due to our system doing other tasks.

As expected the results were similar to the results obtained from Task 1.

We also tried to use an affinity mask in order to isolate one core for the process to run on in order to check if multiprocessing in python will take this into account. The results show that the affinity mask was ignored by multiprocessing and all 4 cores were used.

Observations:

We noticed a clear increase in speed up time checking primes in the list. After a pool size of 4 the time does not change because there are only 4 physical cores to carry out the pool size of processes at once.

Running the `check_prime` on slices of the list we see a step like ladder plot, which shows how the cores are splitting the elements of the list between them when the pool size is even or odd.

For example when the pool size is set to 1 we can see an increase in time with every element added to the list. When the pool size is set to 2 we can see an increase in time with every 2 elements added to the list and so on.