CSC2002S Assignment 3

PHLALA005

Introduction:

This assignment was to investigate the effect of parallel computing on the runtime of programs. We used a wind and cloud identification and classification program to illustrate the effects. The expected hypothesis is that for small data sets, the serial program would be faster but for larger datasets, the parallel program will be faster.

Methods:

Firstly a serial program was created. This is used as a golden measure for both accuracy and execution time. This program was tested against the sample correct outputs for correctness. A second version of the same program was created which uses the Java Fork/Join library to implement parallel threading. This allows multiple threads to be running simultaneously, utilising the computer's resources more effectively. The fork join library allowed me to split up the workload of the program into smaller chunks. This allows you to allocate these chunks to individual threads which each work on a small amount of the overall workload. This is the fork component. Then when each thread is done, the program joins the data returned from all the threads into one, creating the final output.

Each time a time was measured, the program was run a few times first to "warm the cache". This allows instructions to be loaded into memory which reduces variance in results. Then the measurement is taken 5 times and averaged to produce the final result. This result is then compared to the golden measure for the appropriate size dataset to determine a speedup. For the simple data size in the linear program, the time to execute had to be measured with the *nanoTime()* function. This function doesn't necessarily have nanosecond accuracy but averaging should remove some of that noise and speedup is not possible to calculate with a 0 duration event. All other times were measured with the *currentTimeMillis()* function. This returns the current system time in milliseconds and by recording the time before and after running, the time between these events can be determined.

To validate the output, I referred back to the provided outputs and used a simple bash file to ensure the results were identical, line by line. These tests were run on both mine and my sister's laptops. The results shown were incredibly similar and thus I simplified by averaging across them. Both laptops had 4 Intel i7 cores.

A major problem I had was properly joining the wind average results because it was difficult to weight them properly when the workload (the range of elements) each thread deals with might not be identical. This was solved by instead saving all of the pieces along with the data range which was all combined at the end of the program's runtime.

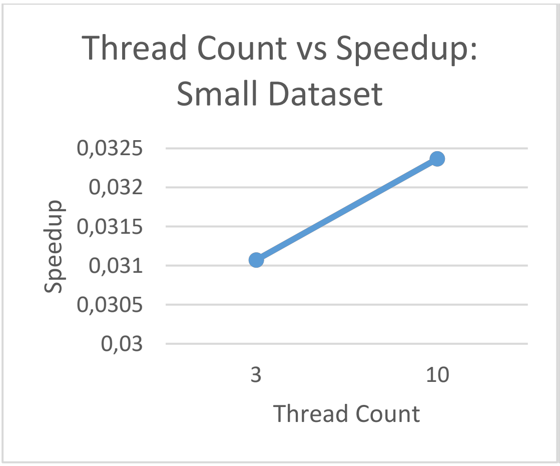
Results and Discussion:

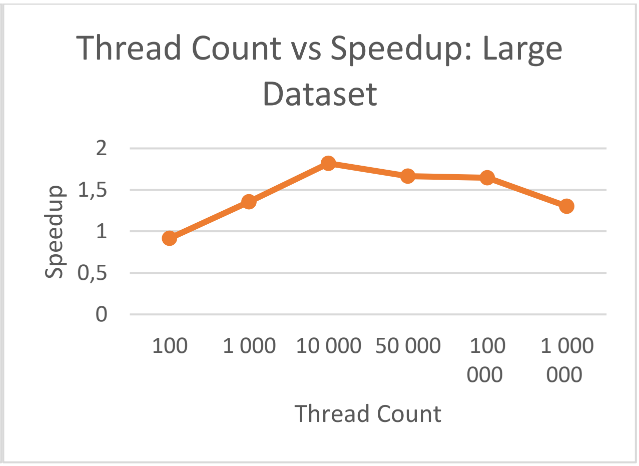
The results are tabulated below:

Serial Time: (s)		
Size:	Small	Large
	0,000146	0,459
	0,000148	0,485
	0,000179	0,49
	0,000138	0,48
	0,000166	0,495
Average	0,000155	0,4818

Parallel Time: (s)								
	Small		Large					
Threshold	3	10	100	1 000	10 000	50 000	100 000	1 000 000
	0,006	0,004	0,489	0,344	0,303	0,257	0,301	0,338
	0,005	0,004	0,555	0,371	0,301	0,28	0,286	0,444
	0,004	0,004	0,538	0,362	0,237	0,291	0,324	0,387
	0,005	0,005	0,536	0,347	0,235	0,304	0,25	0,361
	0,005	0,007	0,511	0,352	0,248	0,316	0,302	0,319
Average	0,005	0,0048	0,5258	0,3552	0,2648	0,2896	0,2926	0,3698
Speedup	0,031072	0,032367	0,916318	1,356419	1,8194864	1,663674	1,64661654	1,302866414

The serial implementation was exceedingly quick at low data sizes, running in less than a millisecond. The parallel implementation at these same data sizes, is exceedingly slow. This speedup is just over 3% of the serial speed.





Where the parallel option is more attractive is for large data sets. This test set contained over 5 million elements. For large datasets, it is clearly faster, with only a very low thread threshold resulting in a speedup of less than 1. The shape of this graph is as expected. If there are too many threads then too much work is being done switching between them and maintaining the overhead. However if there are too few threads then the full resources available might not be being used efficiently. The highest attained speedup was just over 1.8. This is close to doubling the serial implementation. With a 4 core CPU, the maximum theoretical speedup was 4 times but this is not possible in reality because the rest of the cores are being used for other things within the OS or other applications. Despite falling short of the ideal speedup, this is still a significant speedup and if it is important to whichever applications of this program there are that this program runs very quickly then implementing parallel processing is worth it. If not, you would be spending far too much time on programming for an imperceptible difference to the user.

Parallelisation is most effective for data sets with ranges greater than 1 million elements. For a data set of around 5 million (like the large dataset provided the sequential cut-off is optimal around 10 000 elements. This means that the optimal number of threads is around 525 threads.

Conclusions:

The larger the data set, the more useful it is to use parallelisation because the increased efficiency makes up for the increased overheads in managing the threads. The more cores you have in your architecture, the more efficient parallelisation is as it enables the parallel program to use more and more of the available resources proportionally to the serial program. However, for single core CPUs or lower size data sets, it is not worth the extra development time to implement parallelisation.

Github Link:

https://github.com/Lolarent000/CSC2002S-Prac-3