EG3205 First Assignment Report

Task 1

This for loop runs N times. Values are stored in the A array, in each iteration, elements of the array are square-rooted and stored in the variable y. The value of y is then added to the value of the element in the A array and divided by the square of the value of x. The values of the evaluated expression are stored in the D array. To run this parallelly, only one clause is necessary.

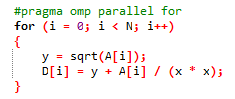
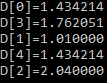


Figure 1 - Task 1 Code Figure 2 - Example Parallel Execution

Task 2

This block of code is running two for loops N number of times. The first for loop is evaluating an expression which will be stored in the D array and the second for loop is evaluating an expression to be stored in the C array using a value stored in the D array.

To parallelise this code, a nowait clause is added to the first for loop so that if the loop is executed then using that D[i] value it can go onto the next for loop and evaluate the expression for C[i]. Without the nowait, there is a barrier at the end of the for loop, thus stopping the program from moving onto the next for loop even though it has evaluated the expression for D[i] which can be used to evaluate C[i]. Thus, without the nowait clause, the program will run the first for loop N times and then run the second for loop N times. So, with the nowait clause, threads will not synchronize/wait at the end of that particular construct.

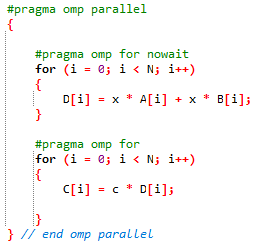
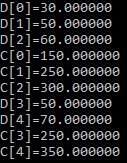


Figure 3 - Task 2 Code Figure 4 – Example Parallel Execution

Task 3

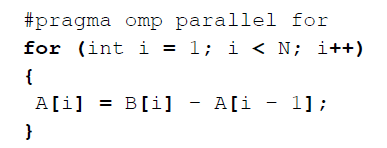


Figure 5 - Task 3 code

This loop cannot be parallelised. This is because the values of A[i] depend on the previous value and thus even by using #pragma omp critical clauses which ensures mutual exclusion, the final values of the array may differ when executed because of the number of possible thread interleavings. So, if this code was run sequentially, then this would be the order of execution.

A[1] = B[1] - A[0];

A[2] = B[2] - A[1];

And so on and so forth until

A[N-1] = B[N-1] - A[N-2];

Hence, every time the A array would always be populated by the same values. So, if we run parallelly we will either execute in the order above (and get the same result as the sequential

case), or we could execute in this order for example:

A[2] = B[2] - A[1];  
A[1] = B[1] - A[0];

Thus, in this case, the final value of A[2] is calculated using the value of A[1] that was originally in the array so this would give a different result for A[2] from the sequential execution.

The greater the value of N there is an exponential increase in the number of possible thread interleavings, and so a wider range of possible final values stored in the array A.

Task 4

This function implements Matrix times Vector (MxV) operation. To run this function in parallel, the variables i and j have to be private because each thread needs a copy of those variables. Otherwise, multiple threads may over increment the for loops if they were shared. The other variables and arrays are shared so that the data in those will be accessible to all threads in order to execute the calculation in parallel. Thus, all threads access the same address space.

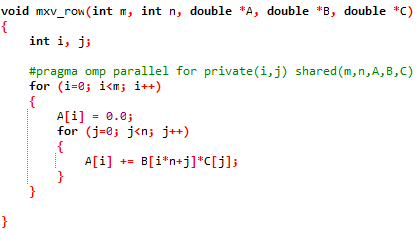


Figure 6 - Code for Task 4

Part 2

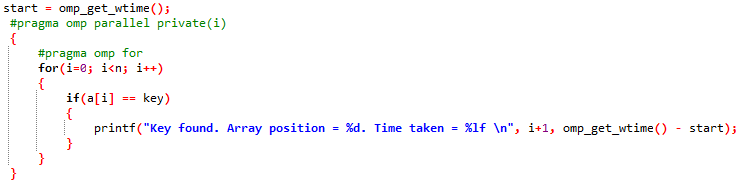


Figure 7 - Extract of Linear Search Code

The linear search algorithm that has been implemented (using an array of size 99000000) is executed fastest when 5 threads are used (takes 0.43483 seconds). The array is deliberately made to be very large so that speed up can be compared. The variable i was made private so that each thread has its own copy of it. The simplest way to implement this search was using the parallel for construct.

In general, the execution time is fairly constant from 2 threads onwards. There is little fluctuation (by a few microseconds) as can be seen from the chart (below in Figure 8) and in Table 1. The linear search was executed on a dual core laptop, which had 2 processors, hence when running the program parallelly, the execution time did not reduce by much when the threads were increased after 2 threads. Generally, the program took a minimum of 0.4 seconds to run on average. When the number of threads were not specified using Openmp statements the computer used 2 threads. Hence, the optimum number of threads is approximately equal to the number of cores/logical processors.

Table 1 - Table Showing Average Time Taken for Execution and Average Speed Up of Program



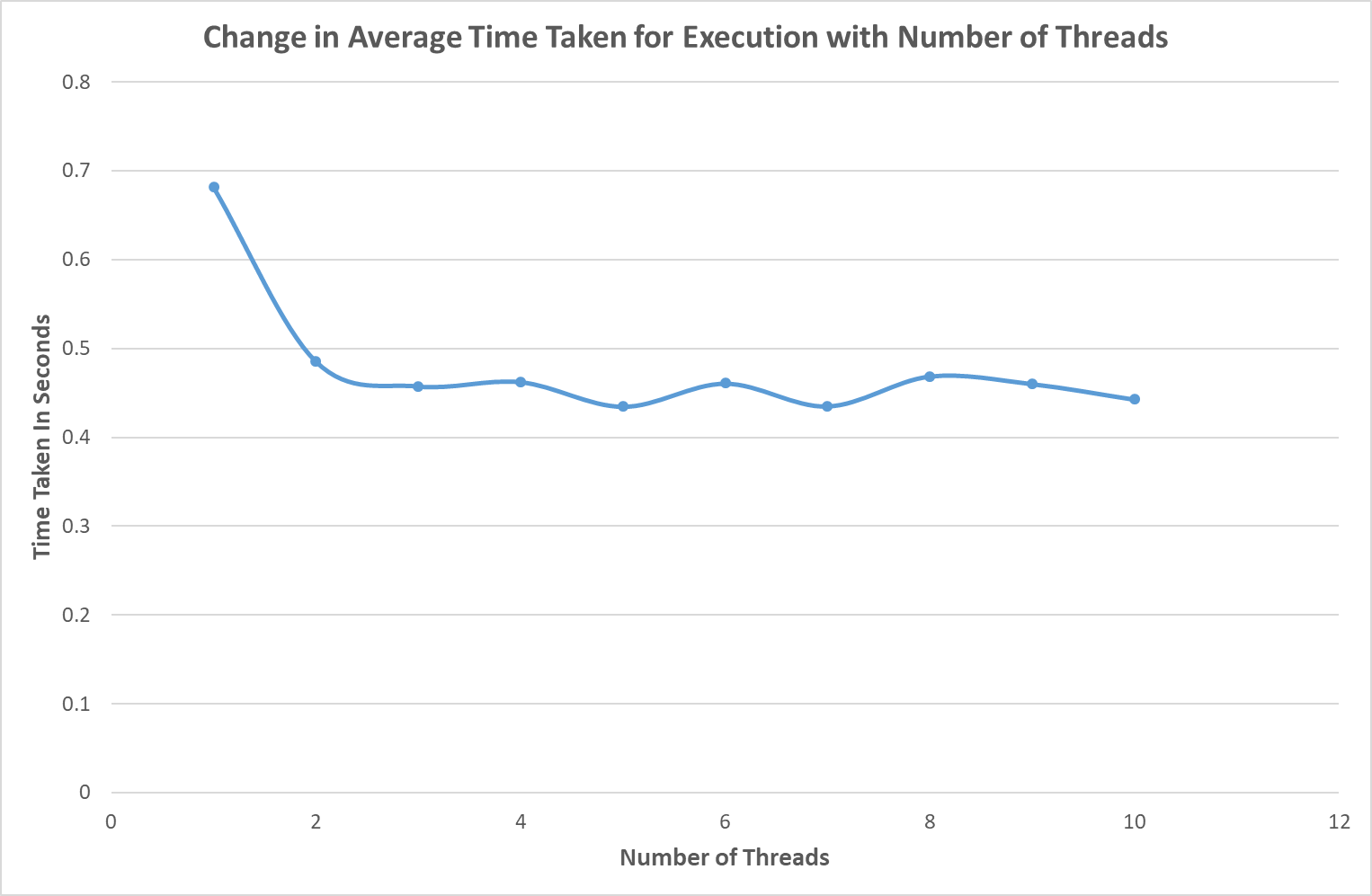


Figure 8 - Chart to Show Change in Average Execution Time with Number of Threads

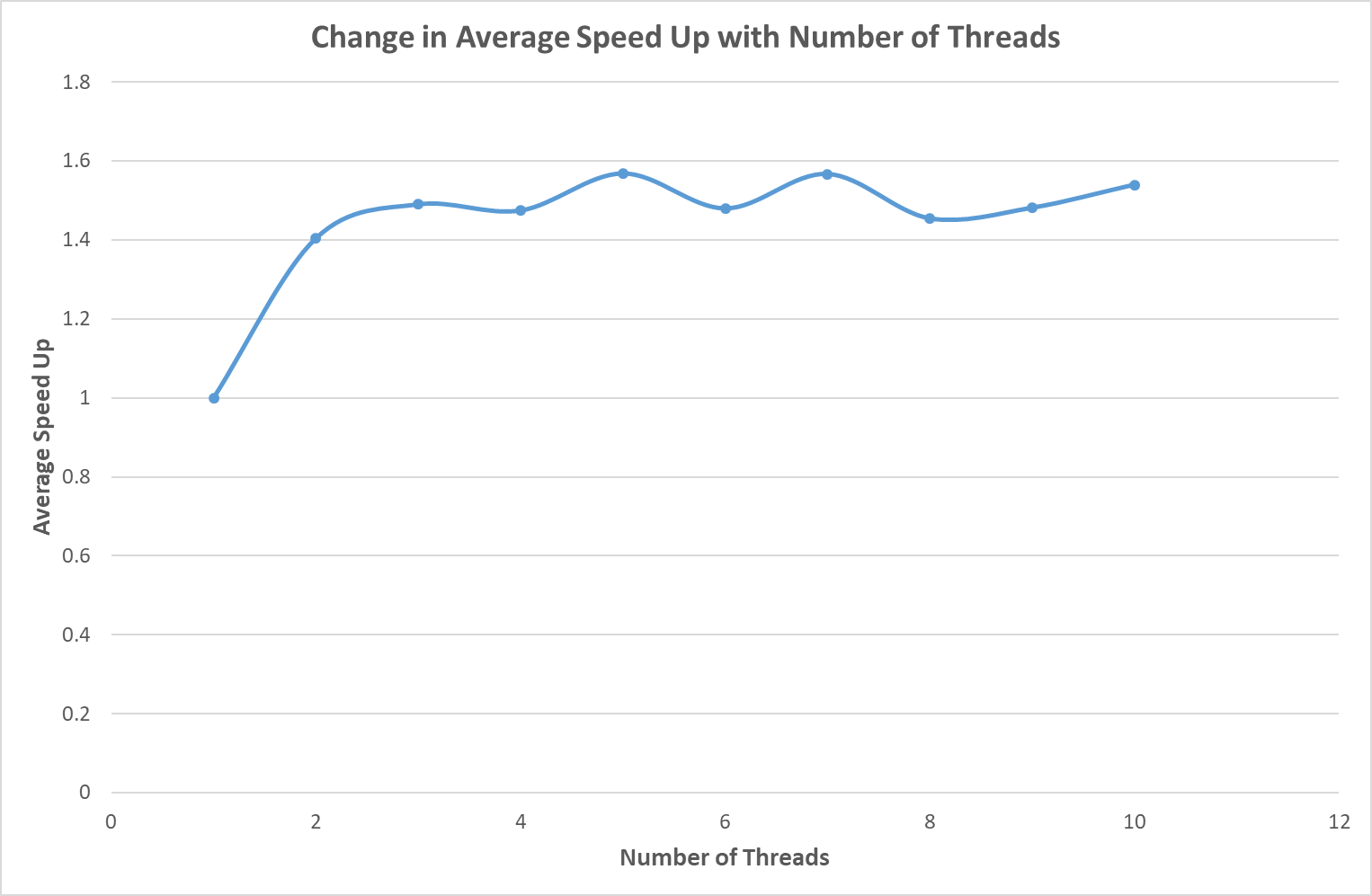


Figure 9 - Chart to Show Change in Average Speed Up with Number of Threads

The maximum speed up is calculated using Amdahl’s law:

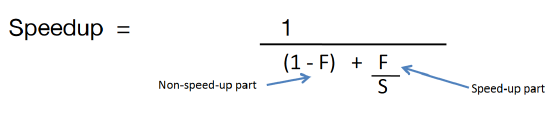


Figure 10 - Amdahl's Law

Where S is the enhancement factor and F is the fraction of the task that can be accelerated.

From the results that can be seen in Figure 9 the average speed up fluctuates between approximately 1.40 and 1.57 from 2 threads onwards. Thus, even with 100 threads the speed up will not increase beyond 1.57 on the dual core computer. Thus, it behaves like it has reached a saturation point. There is an increase in speed up when multiple threads are used compared to the serial version which took on average 0.68 seconds to run. The increase in speed up is approximately 57% [calculation: ((1.57-1)/1)\*100 ] when we consider the maximum speed up from Table 1 which occurred for 5 and 7 threads respectively.

In conclusion, if the code was run on a computer with more processors/cores, then the maximum speed up would have increased. Without the application of Amdahl’s law, the maximum speed up that can be expected is approximately 1.57 to 2 decimal places on the dual processor computer.