CS3210 Assignment 1

Parallel Implementation of Longest Common Subsequence

Jervis Chan Jun Yong

A0134191M

1. Program Design

As described in the assignment outline, an approach to solving the Longest Common Subsequence (LCS) problem is to use a sequential dynamic programming approach. Below, we see the recurrence formula used to construct a score matrix.

$$LCS[i,j] = \begin{cases} 0, & i \text{ or } j \text{ is } 0 \\ LCS[i-1, j-1] + 1, & x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]), & \text{otherwise} \end{cases} \quad (1)$$

According to Eq. (1), for all i and j ($1 \leq i \leq n$, $1 \leq j \leq m$), LCS[i, j] depends on three entries, LCS[i − 1, j − 1], LCS[i−1,j] and LCS[i,j −1]. This shows that LCS[I,j] is dependent on data in the same row and the same column. Thus, the same row or same column data can't be computed in parallel.

To compute the same row or column data in parallel, we need to change the data dependence for LCS[i, j]. According to Yang et al., this can be done by using a table, DP, and a finite set of possible letter values, A. In our problem statement, A will take the possible values of "ATGC". The score matrix is then calculated based on the DP table. [1]

2. Parallel Implementation

In the implementation with processes and threads, I used the fork-join paradigm. The parent process/thread will perform the initialisation and pre-processing of table DP. It will then fork child processes/threads which will do perform the work on the score table. The work for each child is split evenly by the total number of columns, with any remainders given to the last child. Once complete, the child will exit before the parent prints the final answer and cleans up resources.

Synchronisation is required for the rows, and to do this, I used a barrier from the pthread library. At each row, the workers will be blocked at the barrier until all the other workers have reached the barrier before they can move on to the next row. In the case for processes, a shared memory was required for the score table and the DP table. In the OpenMP implementation, we will use the pragma directive to calculate the score table, and there was no need to explicit synchronize across threads or processes.

3. Results

I performed the experiment with different nodes and input size. I repeated the experiment, took 3 measurements with the same settings and showed the best results in my analysis. The best result happens when the run is the least impacted by other tasks running at the same time in the system.

In the case for my implementation with processes and threads, I also performed the experiment with varying numbers of threads from 1 to 256.

Lab Machines Used: 1) soctf-pdc-004 (Xeon), 2) soctf-pdc-009 (i7)

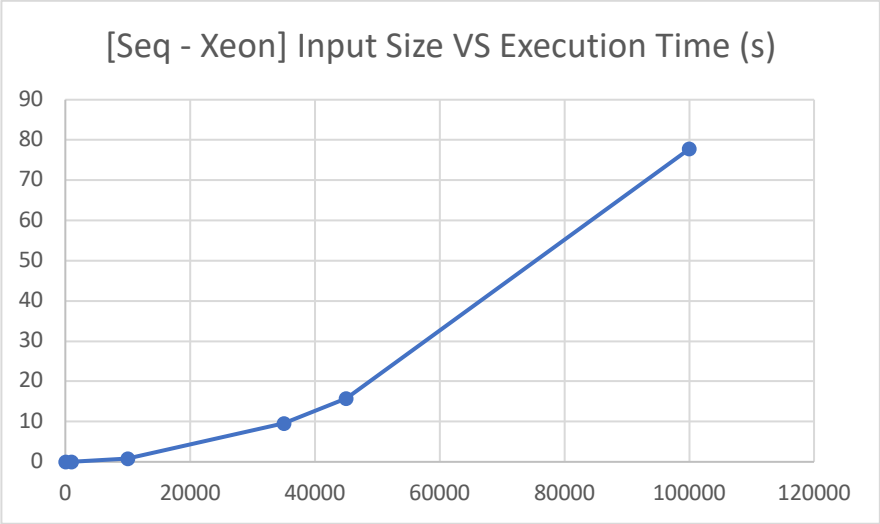

Figure 1: Input Size VS Execution Time(s) of Sequential Algorithm on Xeon Silver 4114 Processor

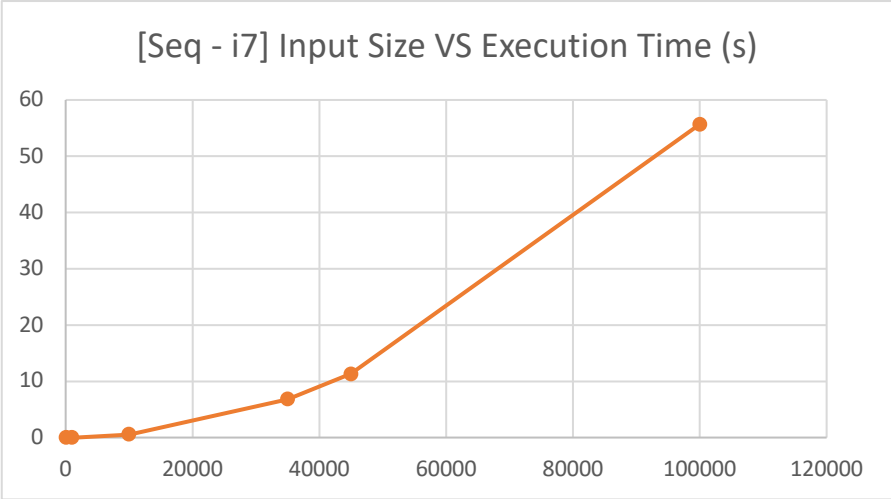

Figure 2: Input Size VS Execution Time(s) of Sequential Algorithm on i7-7700 Processor

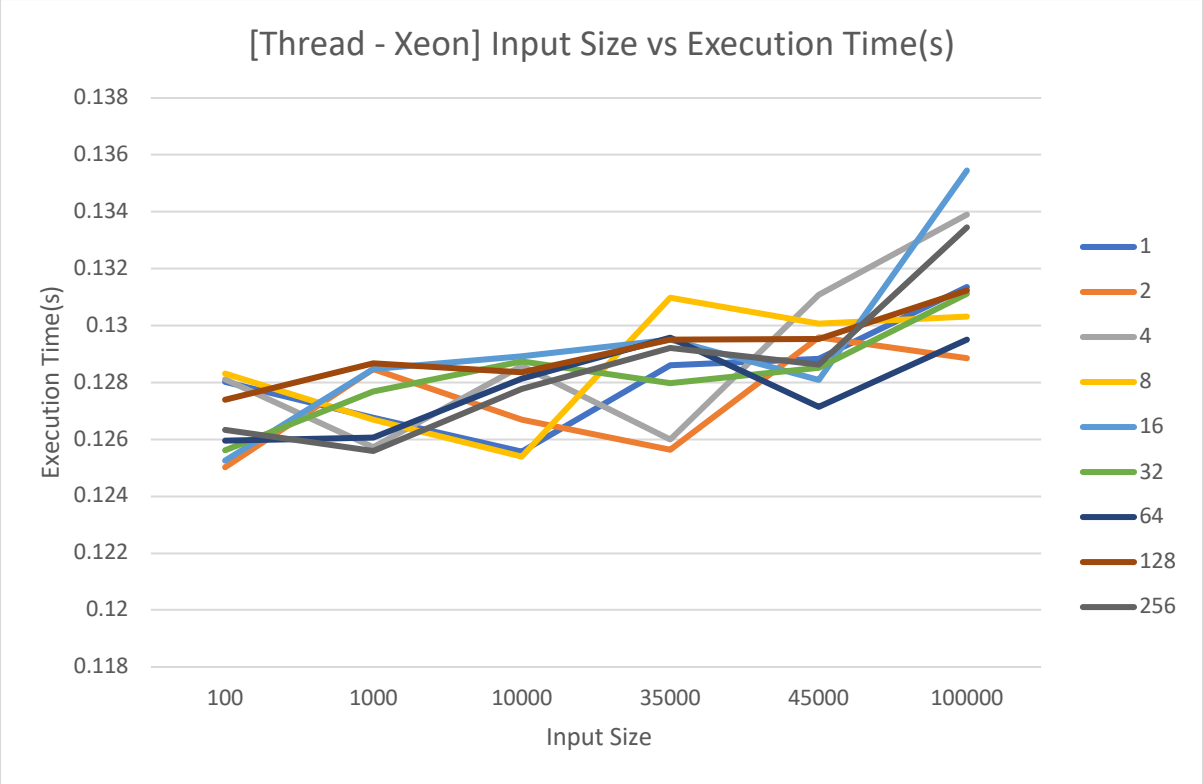

Figure 3: Input Size VS Execution Time(s) of Thread Implementation on Xeon Silver 4114 Processor

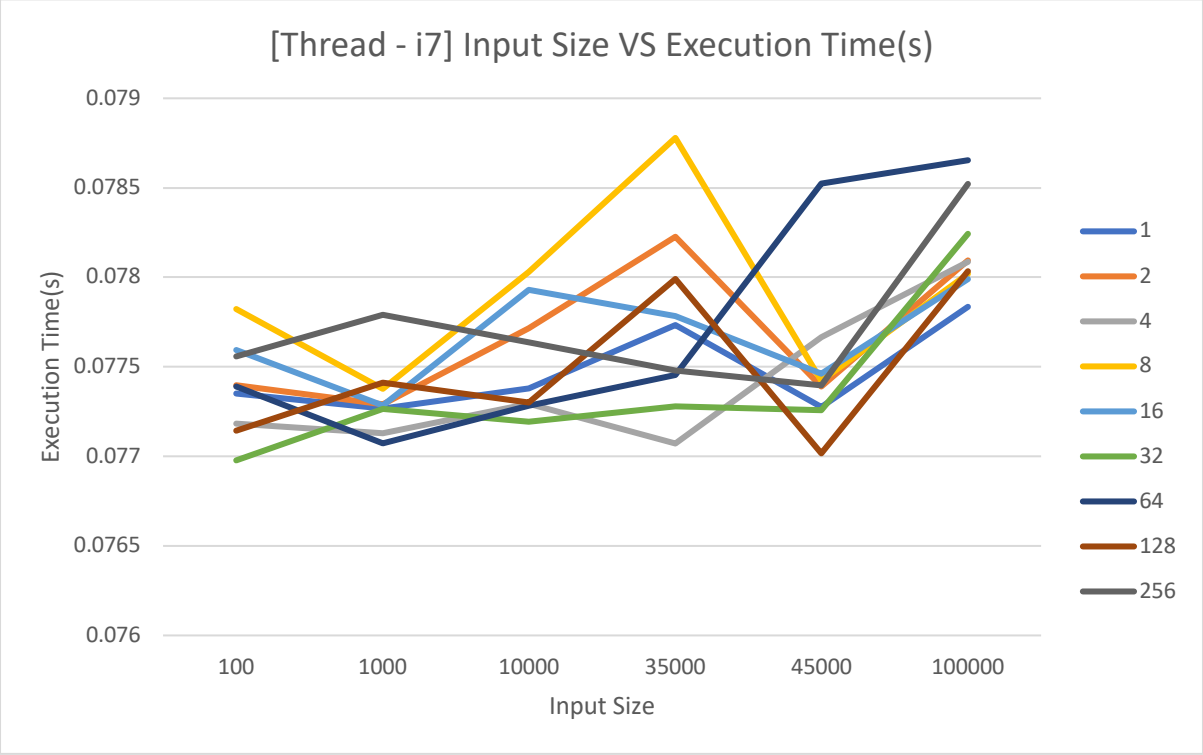

Figure 4: Input Size VS Execution Time(s) of Thread Implementation on i7-7700 Processor

Figure 5: Input Size VS Execution Time(s) of Process Implementation on Xeon Silver 4114 Processor

Figure 6: Input Size VS Execution Time(s) of Process Implementation on i7-7700 Processor

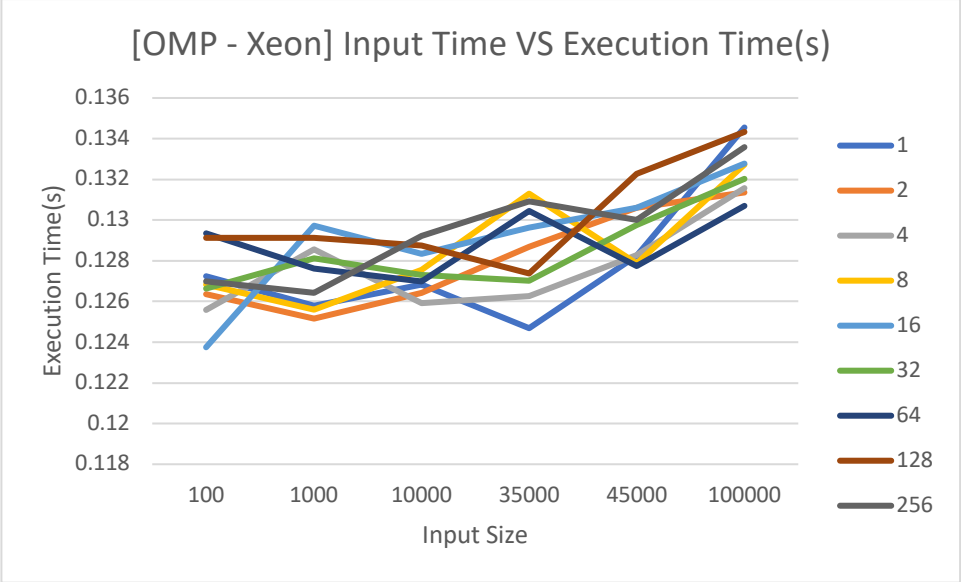

Figure 7: Input Size VS Execution Time(s) of OpenMP Implementation on Xeon Silver 4114 Processor

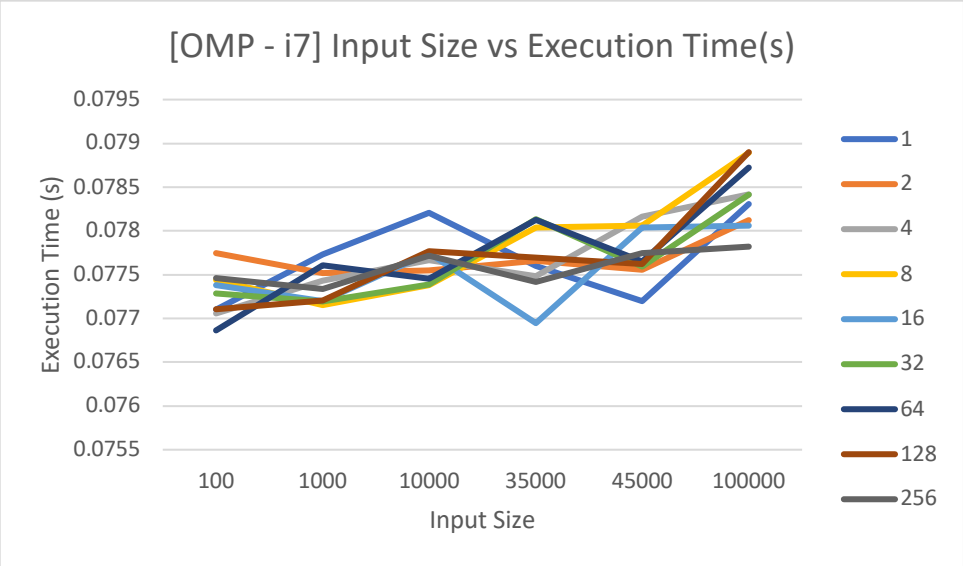

Figure 8: Input Size VS Execution Time(s) of OpenMP Implementation on i7-7700 Processor

4. Analysis

4.1 Speedup

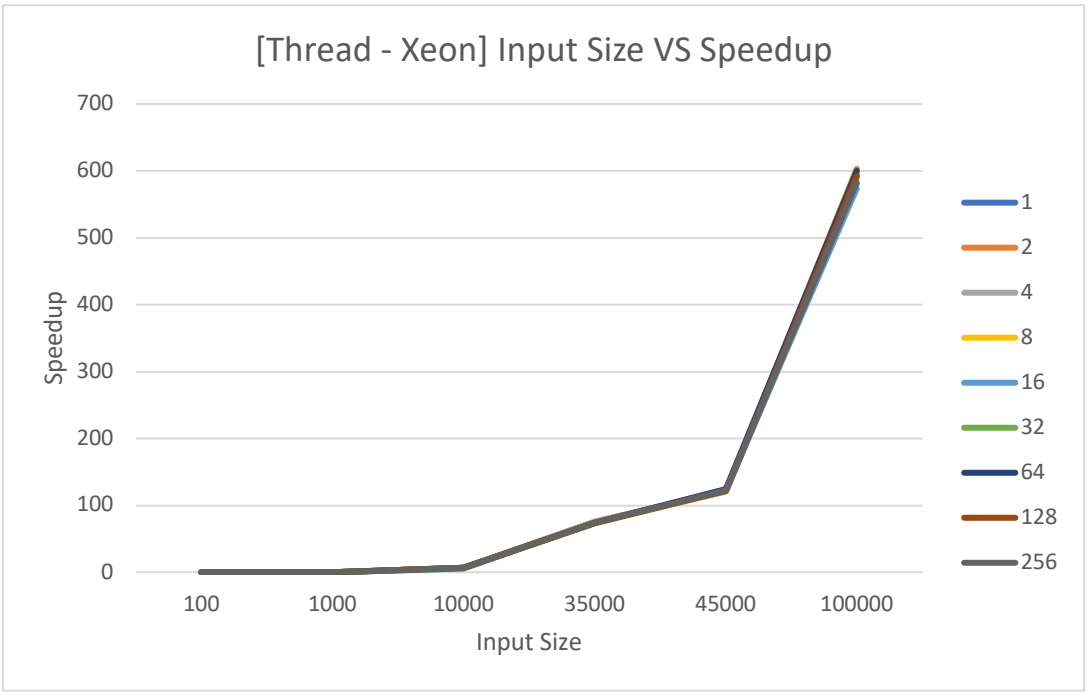

Figure 9: Input Size VS Speedup of Thread Implementation on Xeon Silver 4114 Processor

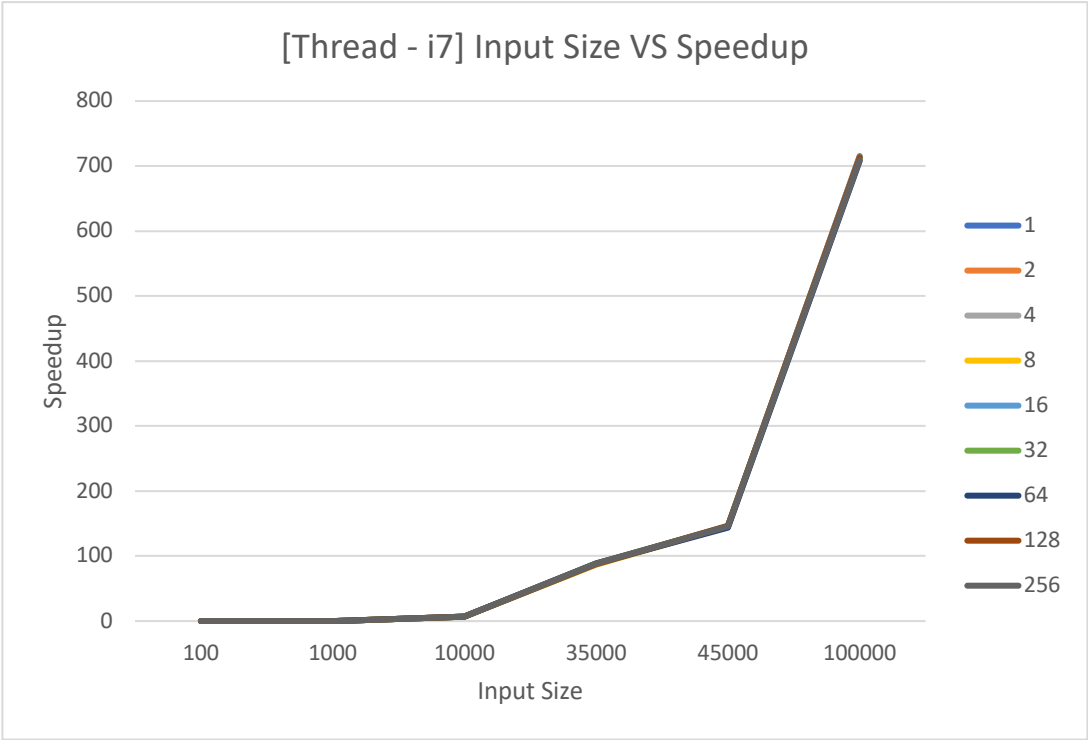

Figure 10: Input Size VS Speedup of Process Implementation on i7-7700 Processor

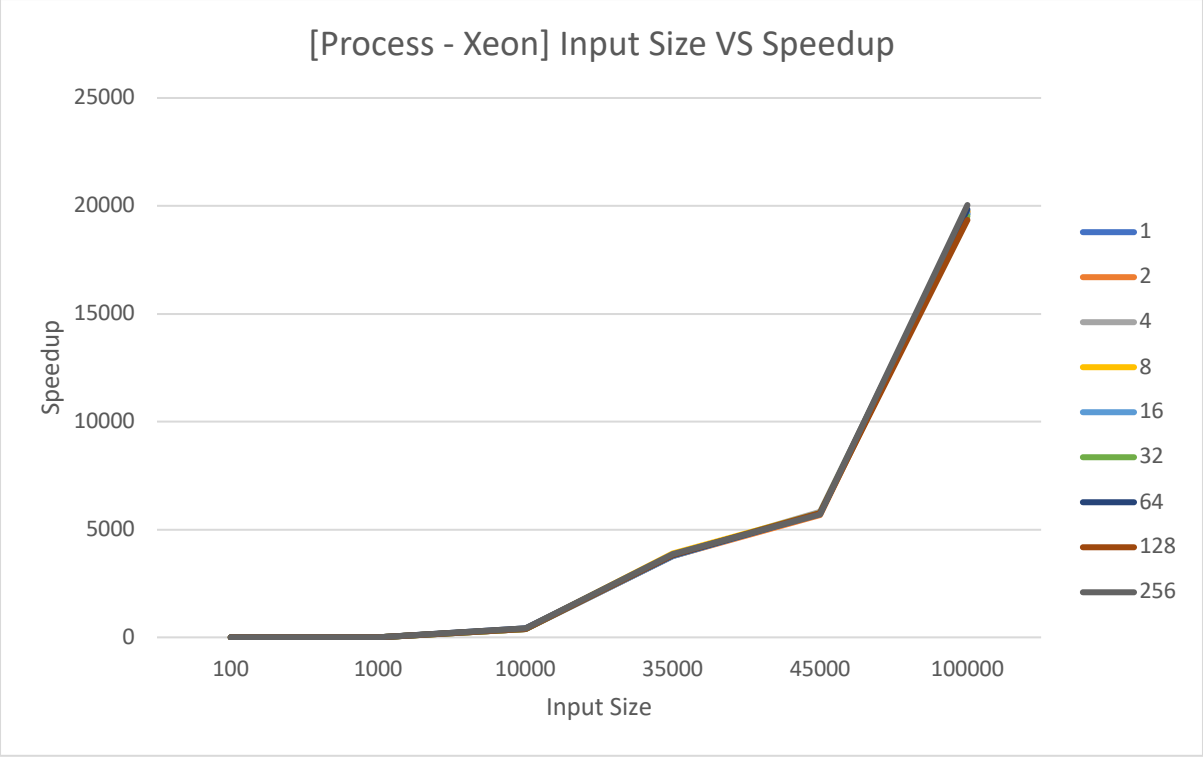

Figure 11: Input Size VS Speedup of Process Implementation on Xeon Silver 4114 Processor

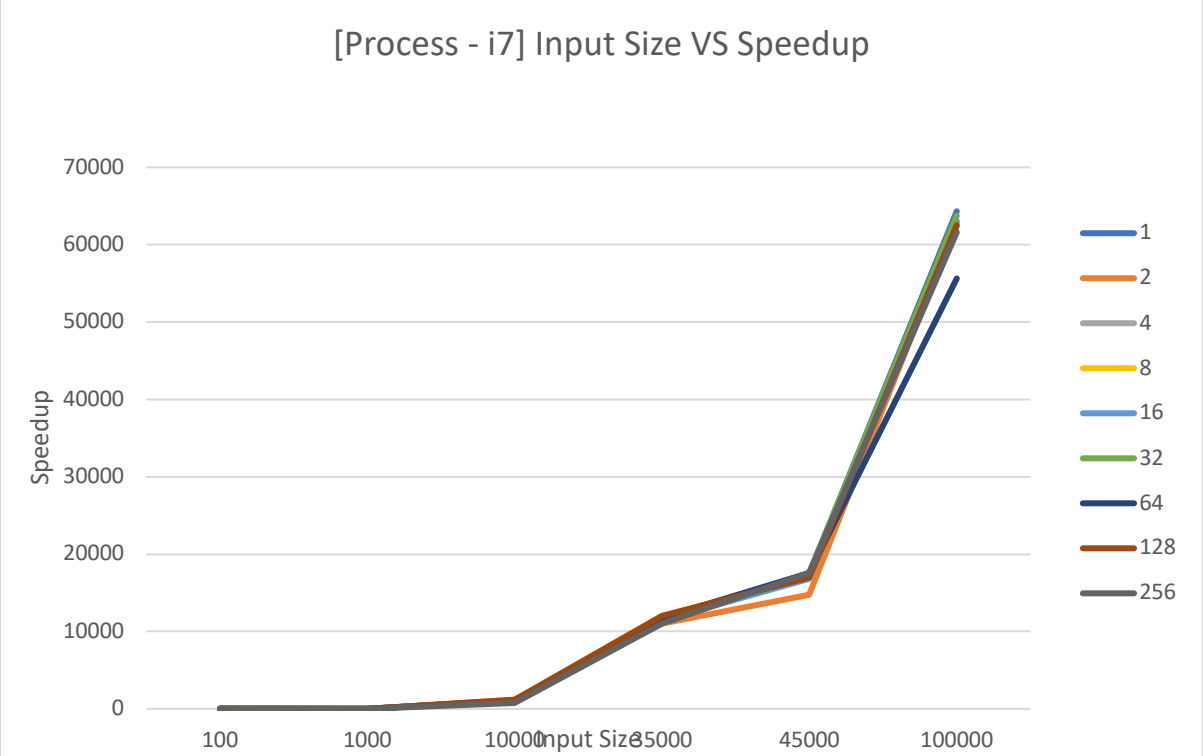

Figure 12: Input Size VS Speedup of Process Implementation on i7-7700 Processor

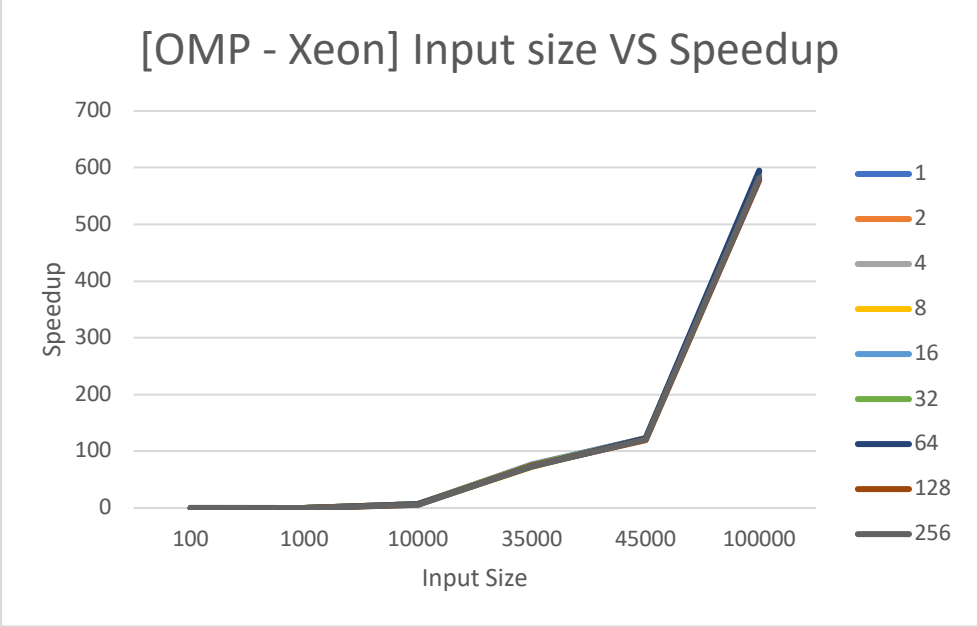

Figure 13 : Input Size VS Speedup of OpenMP Implementation on Xeon Silver 4114 Processor

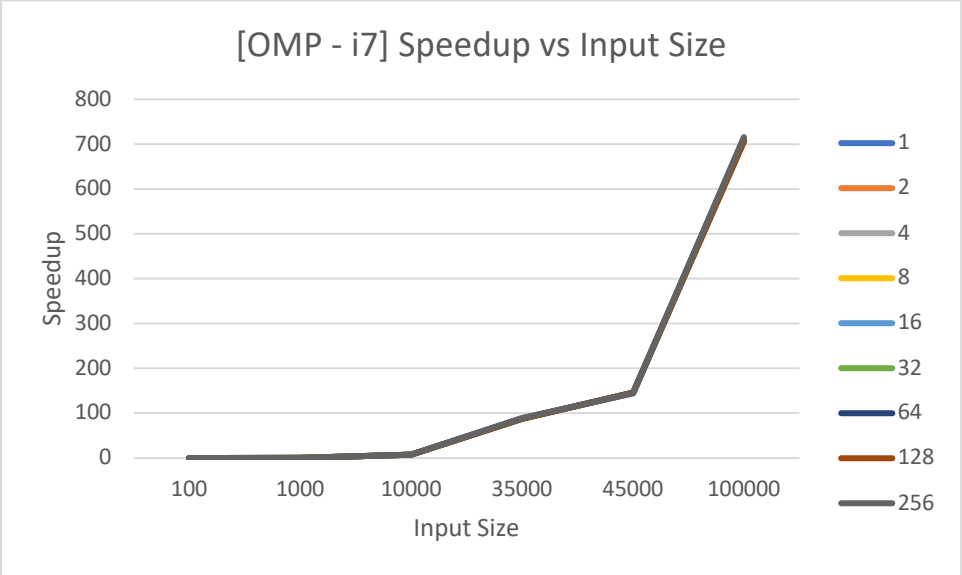

Figure 14: Input Size VS Speedup of OpenMP Implementation on i7-7700 Processor

From the figures above, we can see that the parallel implementation for the LCS algorithm definitely results in a faster speed-up for large input sizes, and this speedup scales with increasing input size. This is because in the sequential algorithm has a time complexity of $O(n^2)$ as we calculate the entries for an n X n dynamic programming array, causing the execution time to increase more than proportionately. On the other hand, the execution time for the parallel implementation remains largely constant, resulting in a more than proportionate scaling of the speedup as well. It must be noted that due to the overhead, the

parallel implementation was slower than the sequential implementation for the input size of 100 and 1000, but once the input size got larger, the parallel implementation was a lot faster.

When experimenting between different nodes, the i7-7700 Processor was faster and had a better speed-up compared to the Xeon Silver 4114 Processor.

4.2 Performance Comparison

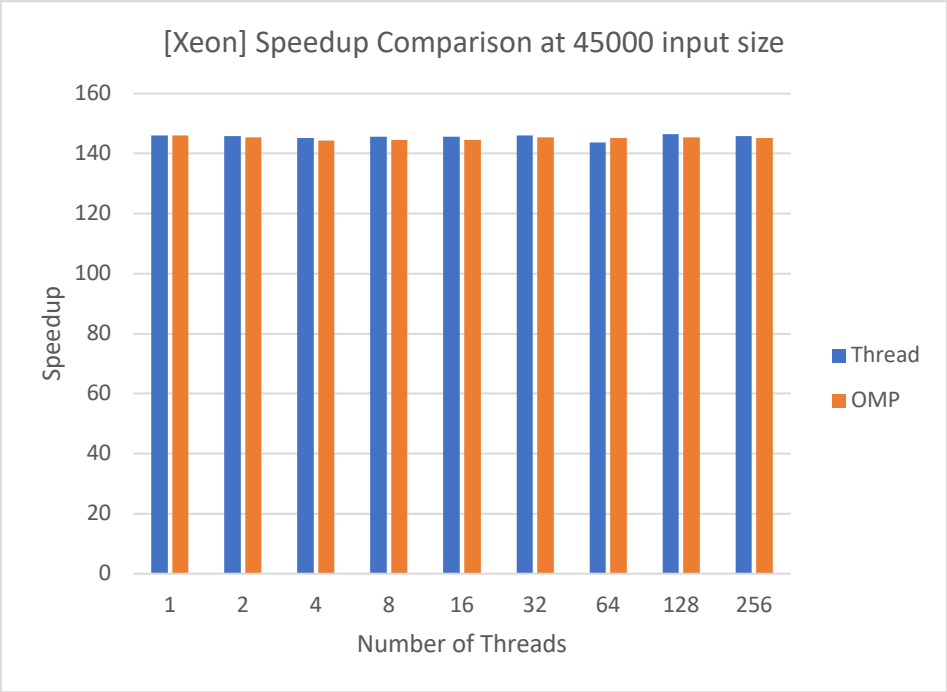

Figure 15: Speedup Comparison (Thread VS OMP) on Xeon Silver 4114 Processor

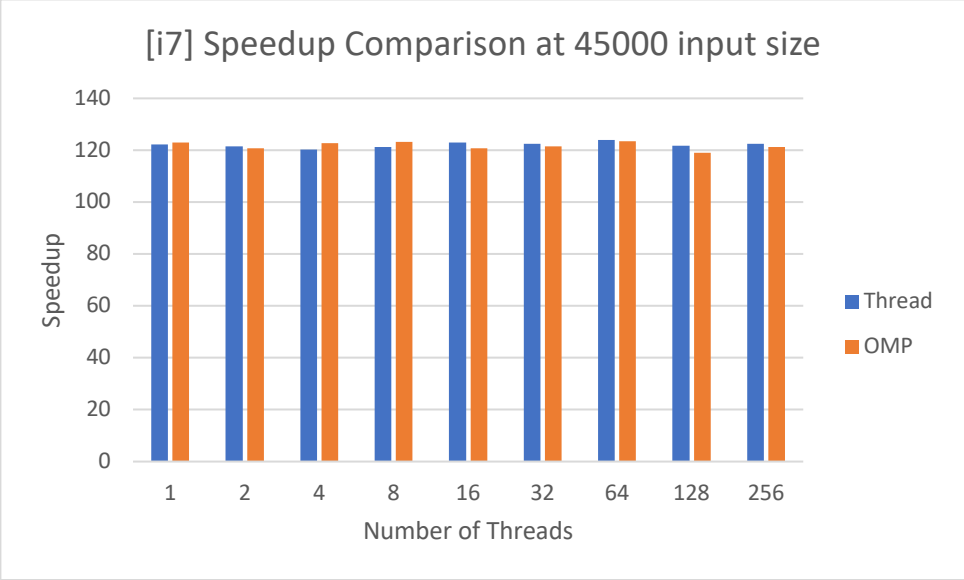

Figure 16: Speedup Comparison (Thread VS OMP) on i7-7700 Processor

From the figures above, we can see that the OpenMP implementation performed roughly similar to the thread implementation of the parallel algorithm. This could be due to the fact that the OpenMP library is a high level library built on top of threads. However, while the performance was the same, using the OpenMP library allowed for much greater convenience as there was no need to explicit code synchronization methods for the threads.

5. Conclusion

In this experiment, I tried to investigate ways to implement a parallel algorithm to solve the Longest Common Subsequence (LCS) problem. In the original dynamic programming algorithm, we see high data dependence for each entry on both previous rows and columns. In order to parallelise this algorithm, we changed the data dependence according to the method explained by Yang et al. We implemented this new parallel algorithm with threads, processes and OpenMP, which demonstrated a clear speed-up as compared to the sequential algorithm.

6. References

[1] Yang J, Xu Y, Shang Y. An efficient parallel algorithm for longest common subsequence problem on GPUs; Proceedings of the World Congress on Engineering. 2010 Vol I. p. 499–504.