Needleman-Wunsch Algorithm

Team 25
Srikar Bhavesh Desu (2020101003)
Sambasai Reddy Andem (2020101014)

Needleman–Wunsch algorithm

The Needleman–Wunsch algorithm is an algorithm used in bioinformatics to align protein or nucleotide sequences. It was one of the first applications of dynamic programming to compare biological sequences. The algorithm was developed by Saul B. Needleman and Christian D. Wunsch and published in 1970.

First we make the initial matrix which has (A+1) columns and (B+1) rows, where A and B are the lengths of the two sequences given. Then we make the dynamic programming matrix following three basic steps:

First, we initialise both the first row and column of the scoring matrix with the gap penalty. Each cell is filled with their respective gap penalty.

The second and crucial step of the algorithm is matrix filling starting from the upper left hand corner of the matrix. To find the maximum score of each cell, it is required to know the neighbouring scores (diagonal, left and right) of the current position. From the assumed values, add the match or mismatch (assumed) score to the diagonal value.

Similarly add the gap score to the other neighbouring values. Thus, we can obtain three different values, from that take the maximum among them and fill the ith and jth position with the score obtained. We use this method to fill all the remaining rows and columns till the whole matrix is filled. We place back pointers to the cell where the maximum score is obtained from the predecessors of the current cell. These pointers (also called back pointers) point back to the predecessor with the highest score.

The final step in the algorithm is the trace back for the best alignment. The important point to be noted here is that there may be two or more alignments possible between any two sequences.

At every cell we check the maximum predecessor score using the pointers we placed in step2. If there are two or more values which points back, suggests that there can be two or more possible alignments.

By continuing the trace back step by the above defined method, one would reach to the 0th row, 0th column. Following the above described steps, alignment of any two sequences can be found.

Compute Configuration

- Model name: Intel(R) Core(TM) i7-9750H
- CPU MHz: 2600
- CPU max MHz: 4500
- CPU min MHz: 800
- Number of threads: 12
- Number of Cores: 6
- Thread(s) per core: 2
- Core(s) per socket: 6
- Socket(s): 1
- Processor Base Frequency: 2600 MHz
- Hyper-Threading Availability: Yes
- Architecture: x86_64
- RAM: 16137308 kB
- Max Memory Bandwidth: 41.8 GB/s
- RAM Clock Speed: DDR4 and 2667MT/s Synchronous

Correctness Evaluation

We initially made our own small scale test cases on which we manually evaluated the correctness of our algorithm.

After we ensured that the base implementation was correct, we compared all further optimisations' output with the baseline program.

We first used a random sequence generator function and passed it to both the baseline function and the optimised function respectively.

Finally we used the datasets provided on moodle for a final check. (sequence length =10,000)

Performance Evaluation Framework

We mainly judged the performance of our optimised program on the execution time of the relevant function.

We also calculated the GIPS (Giga Instructions per second), the memory bandwidth and the compute throughput for further assessment of the performance.

We also varied the DNA sequence size to check how each of the optimised functions perform with respect to the brute force dp solution without any optimization.

Datasets Used

- We first used random small sequences that we came up on our own to test our codes with a brute force solution.
- We then made a generator function to generate random sequences, given the length of the sequence and tried aligning the 2 random sequences with our algorithms.
- Finally we used real time datasets that were provided as fasta files via moodle.

Baseline Performance

Program-1 : We used column traversal while iterating through the row-major matrix, to calculate the dp matrix.

Execution Time - 740 ms(g++), 713ms(icc)
Throughput - 0.865 GIPS(g++)

Program-2: We used regular row traversal while iterating through the row-major matrix to calculate the dp matrix.

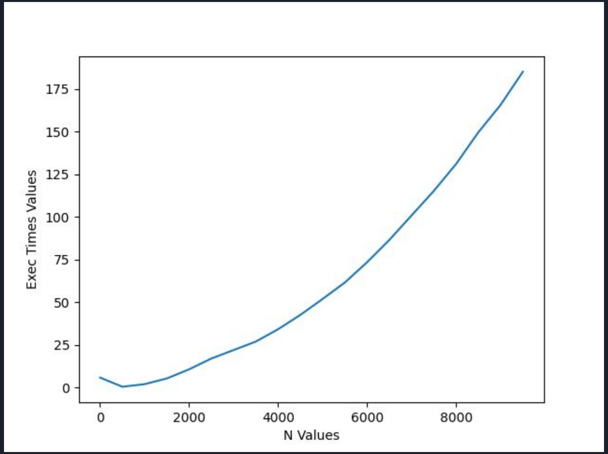
Execution Time - 251 ms(g++), 252ms(icc)
Throughput - 2.07 GIPS(g++)

(Note: all evaluations are done on the sample dataset that has sequence length of 10,000)

Optimization Techniques

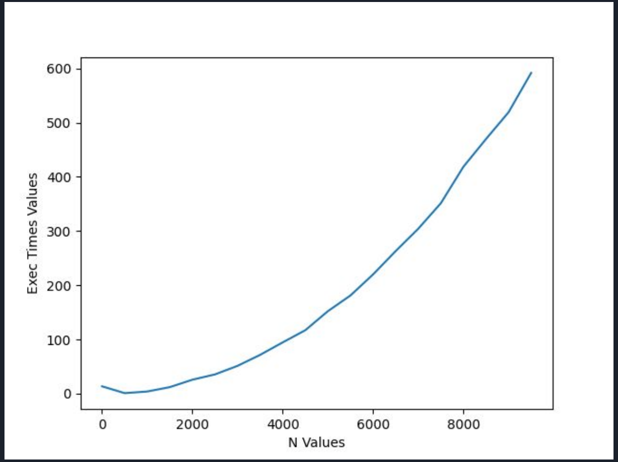
1. Firstly, we made the outer loop traverse the rows as the matrix is row major. This sped up our program as it had more spatial locality, leading to more cache hits.

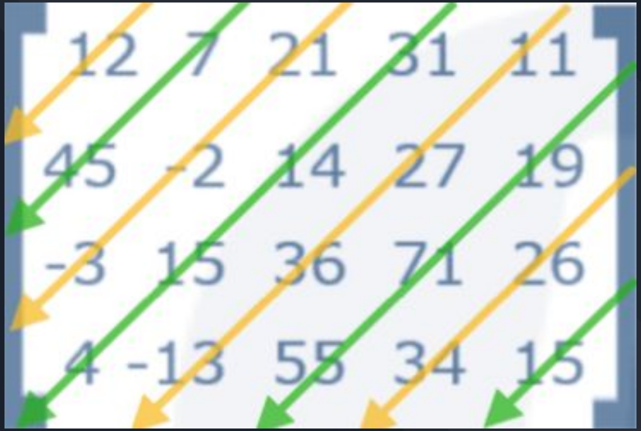
   Execution Time : 251 ms(g++), 252ms(icc)
   Throughput : 2.391 GIPS
   Graph :



2. We then calculated the scoring matrix in an anti-diagonal manner, i.e, we iterated through diagonals which stretch from the top of the matrix to the left, as shown in the figure below. Note that, this is without parallelising, thus due to lack of spatial locality, it gives bad execution times.

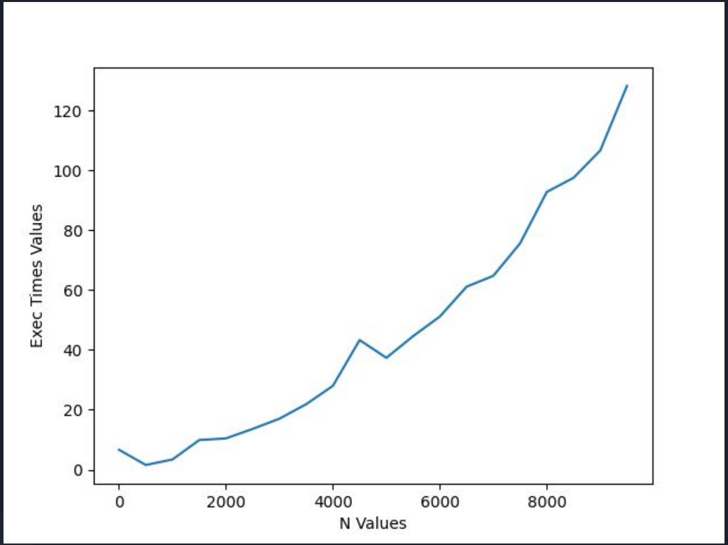
Execution Time : 683 ms(g++), 753.3 ms(icc)
Throughput : 0.879 GIPS
Graph :





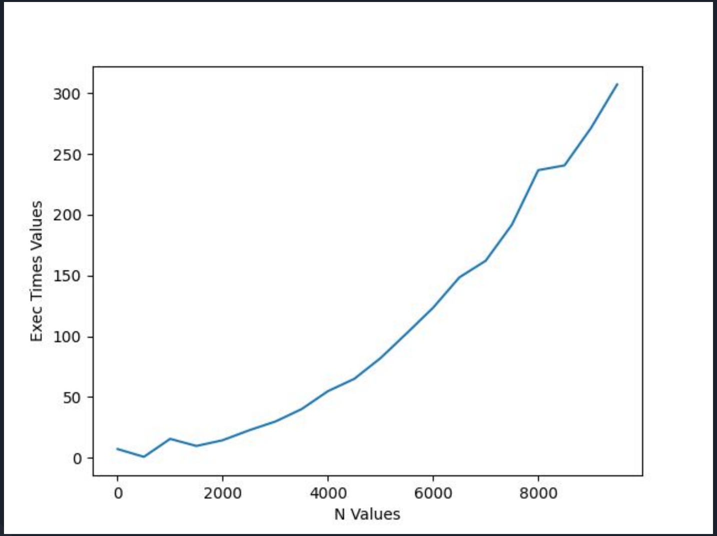
3. We then parallelised this as all the elements belonging to a certain anti-diagonal have no dependencies other than the previously computed anti-diagonals. This led to a major speedup in execution time.

Execution Time : 142 ms(g++), 168ms(icc)
Throughput : 4.225 GIPS
Graph :



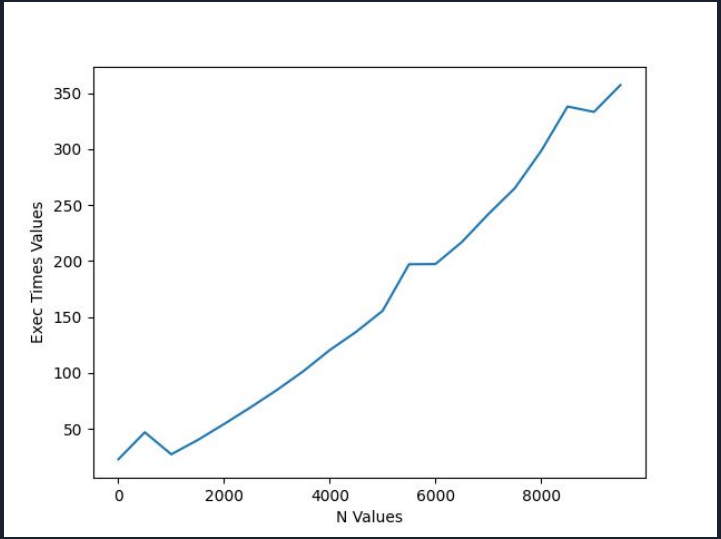
4. Next, we explored the idea of tiling the matrix for better spatial locality thereby increasing the cache hits and reducing the memory access time.

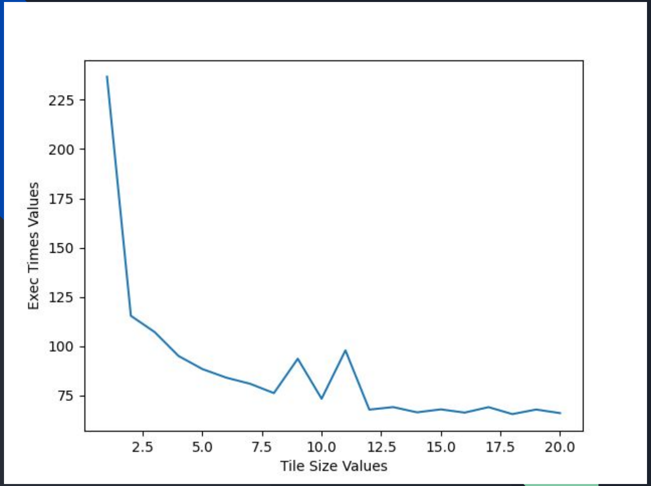
Execution Time : 420 ms(g++), 484(icc)
Throughput : 1.432 GIPS
Graph :



5. Then, we applied the anti-diagonal concept of computing the dp matrix inside every single tile that we obtain from tiling the entire matrix. Here we also parallelised this anti - diagonal computation as we did in the 3rd optimisation.

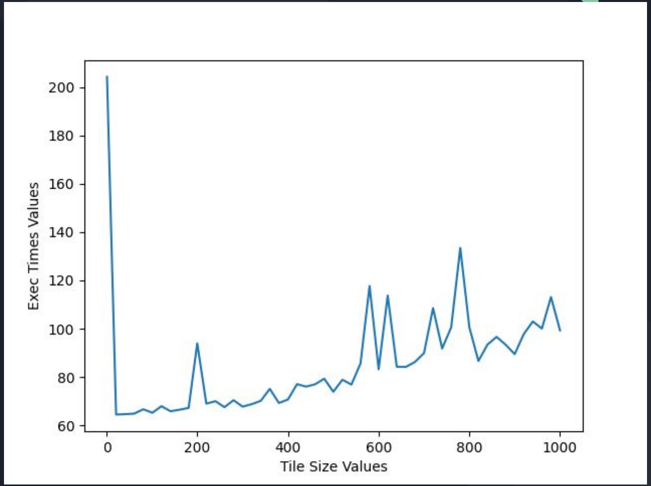
Execution Time : 363 ms(g++), 384ms(icc)
Throughput : 1.652 GIPS
Graph :

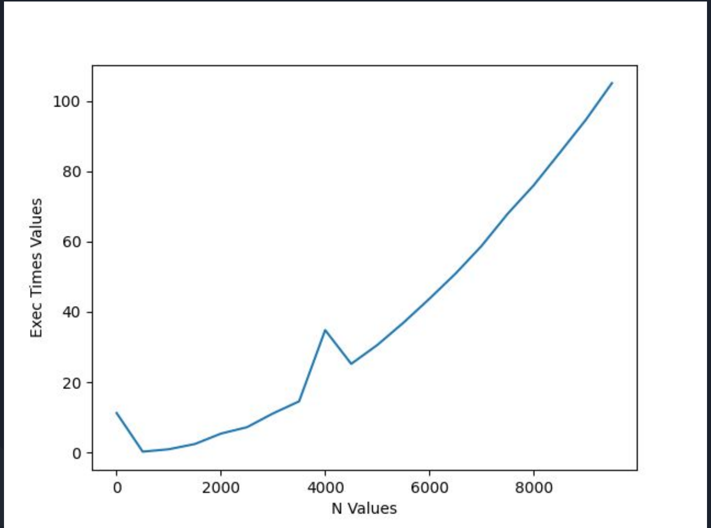




6. Next, we applied the idea of anti-diagonal computation on the tiles themselves as a whole. So for every tile we compute the dp matrix using the naive row major transversal method, but the tiles belonging to a specific anti-diagonal were parallelised. This gave us our best result yet.

Execution Time : 112 ms(g++), 111(icc) using tile size 1000 and 64.5 ms(g++), 72ms(icc) using tile size of around 15-30.
Throughput : 5.358 GIPS and 9.302 GIPS
Graphs:(the graphs on the left are exec time vs tilesize)





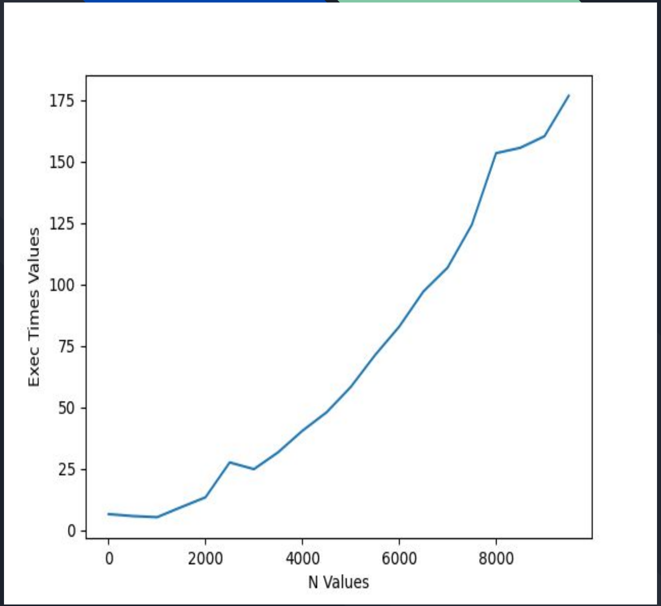
7. Finally, we applied the anti-diagonal approach to both the tiles themselves as a whole and to the cells inside each tile as well. So every anti -diagonal of tiles in the matrix were parallelised and every anti-diagonal of cells in each tile were also parallelised. We expected this to give the best performance, but unfortunately, despite a good improvement from the baseline performance, it did not give us the best performance.

Exec time: 162 ms(g++), 137 ms(icc) using a tile size of around 500.
Throughput : 3.704

Best speed up: 740ms/64.5ms = approximately 12.

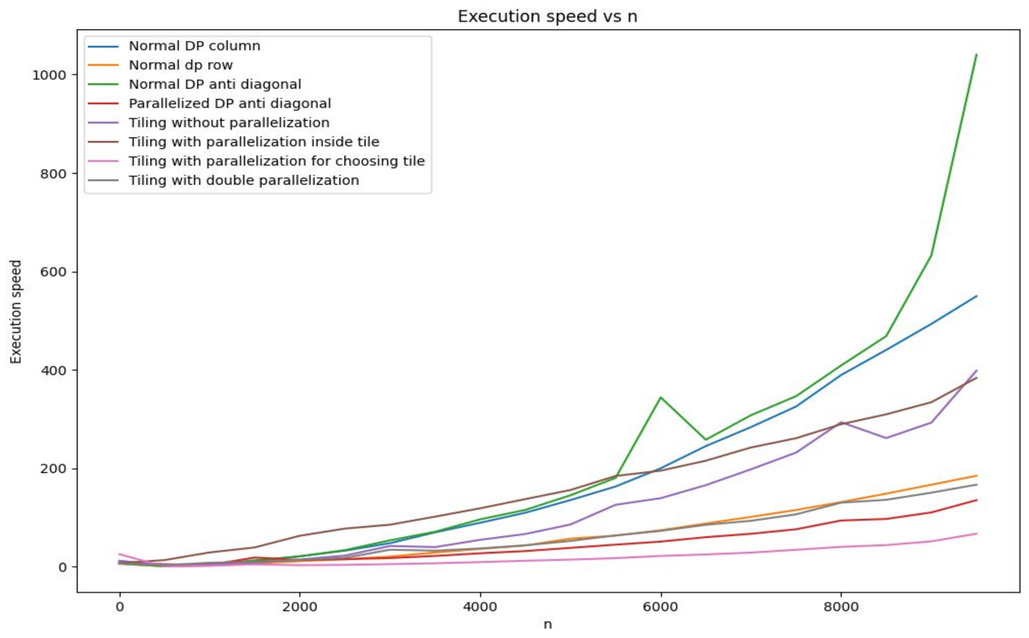
We noticed that a tile size of around 400-600 was giving the best performance for 1e4 sized strings.

The plot is plotted using a tile size of 1000.



Insights and Future Work

Overall graph for all the algorithms implemented is as follows



Insights

- We observed that, on increasing the size of the sequence, the speed up also increased.
  For sequence size of 1e4, we achieved a speedup close to 12. Whereas for sequences of size 3e4, we achieved a speedup close to 18 (10.2 sec/ 0.55sec).
- We also observed that the graph between execution time and tile size usually has a minima and is close to a downwards parabola.
- On just increasing and decreasing the tile size, we achieved considerable difference in execution times.
- Another insight that we found which was not consistent with what we expected is that, tiling with tiles in parallelisation actually performed better than when both the tiles and the filling of tiles was parallelised.
- There was also a considerable speedup just by changing how we traversed through the DP Table, row wise, column wise and anti diagonal wise for example. This is due to the spatial locality exhibited.

Future Work

As we had observed varying performances on varying the tile size in optimizations 4,5,6 and 7, we might look into optimizing the tile size for a given pair of sequences.
One idea for this might be performing binary search on the tilesize to find which tile size gives us the most optimal execution time and speedup.

Another idea we could explore is optimizing the space complexity, as reducing the space required would enhance the performance manifold. For this, we might look into the Hirschberg's algorithm, which has a space complexity of O(n), where n is the length of the shorter sequence.
This algorithm is basically the space-efficient version of the Needleman–Wunsch algorithm that uses the concept of divide and conquer.

THANK YOU