CSE305
Concurrent and Distributed Computing

Project Report
Name: Guruprerana Shabadi, Marie Huynh

1 Introduction : Parallel sequence alignment

Looking for similar proteins domains, finding gene homologies, identifying intron and exon positions, assess evolutionary relationships... Comparing biological sequences is at the heart of several motivations in bioinformatics. To this end, we use sequence alignment to assess similarity and build the best alignment between two sequences.

We take as input two ordered sequences of nucleotides among (A, T, G, C), or amino acids and find the optimal alignment highlighting substitutions, insertions and deletions. In this project, we will focus on pairwise parallel sequencing. We will notably focus on two algorithms based on a similar core structure: the *Needleman-Wunsch* algorithm [1] and the *Smith-Waterman* algorithm [2].

Needleman-Wunsch and the Smith-Waterman are based on the computation of a **distance** between each nucleotide (or amino acids). Both algorithms are based on dynamic programming can be naively implemented with time complexity $O(nm)$, where n and m are the lengths of the sequences to align. Since this becomes computationally significant for very long sequences, a solution can be to parallelize those algorithms.

Hence, we will see two implementations of those algorithms : a non-parallel version and a parallel one.

2 Non-Parallel Algorithms

2.1 Needleman-Wunsch — Global Alignment

The Needleman-Wunsch performs global alignment, i.e., we return the best alignment over the entire two sequences.

Given these two sequences, Needleman-Wunsch uses a score to evaluate how good an alignment is and find the *best one* (the one with the highest score). We use the idea that the best alignment can be constructed from optimal alignments of sub sequences.

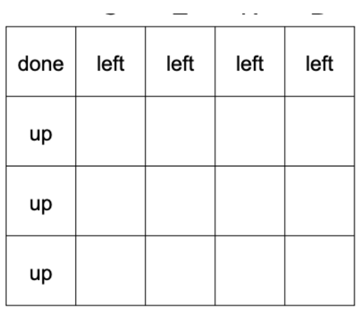
The score is computed with respect to the following variables:

- **match** (the residues are the same): 5
- **mismatch** (substitution, i.e., the residues are not the same): -3
- **gap penalty** (for insertion or deletion): 4

The number given above are examples and subject to change.

Let us consider two sequences A and B.

2.1.1 The Needleman-Wunsch Algorithm can be summarized in three steps.



done	left	left	left	left
up				
up				
up				

Figure 1: Initialization of the traceback Matrix [3]

1. First, we initialize the score matrix H of dimension $(\text{len}(A)+1) \times (\text{len}(B)+1)$. We initialize the first row and the first column to 0. We initialize the traceback matrix T of dimension $(\text{len}(A)+1) \times (\text{len}(B)+1)$ in the following way (Figure 1). The **traceback matrix** will help us build the best optimal alignment over the entire two sequences.

2. Then, we compute the rest of the cells using the following recursion.

$$H(i,j) = \max \begin{cases} H(i-1, j-1) + S(A_i, B_j) \\ H(i-1, j) + \text{gap penalty} \\ H(i, j-1) + \text{gap penalty} \end{cases}$$

   where A_i, B_j are respectively the ith and jth character of the sequences A and B and $S(A_i, B_j) =$ **match** if $A_i = B_j$ and $S(A_i, B_j) =$ **mismatch** otherwise.

   While computing each cell score, we keep track of the direction (diagonal, left, top) from the cell from where the maximum score came from. We store that direction in the traceback matrix T.

3. We start at the last cell of the Traceback matrix and reconstitue the optimal alignment until we reach the first cell of T.

2.2 Smith-Waterman Algorithm — Local Alignment

The Smith-Waterman Algorithm performs local alignment, i.e., we return the biggest region of similarity over two sequences.

The Smith-Waterman Algorithm is the same as the Needleman-Wunsch except for the recursive formula to calculate successive cells.

1. The recursion to compute H is the following:

$$H(i,j) = \max \begin{cases} 0 \\ H(i-1,j-1) + S(A_i, B_j) \\ H(i-1,j) + \text{gap penalty} \\ H(i,j-1) + \text{gap penalty} \end{cases}$$

We observe it is the same one as before, except that we set 0 as lowest value. The zero here allows to ignore any character that was before in the sequence and thus study local regions of similarity.

2. Since we are interested in the biggest region of similarity, we start the trace back at the last best score cell, instead of the last cell of the matrix. We finish the traceback as soon as we encounter a cell whose score is 0.

3 Parallel Version

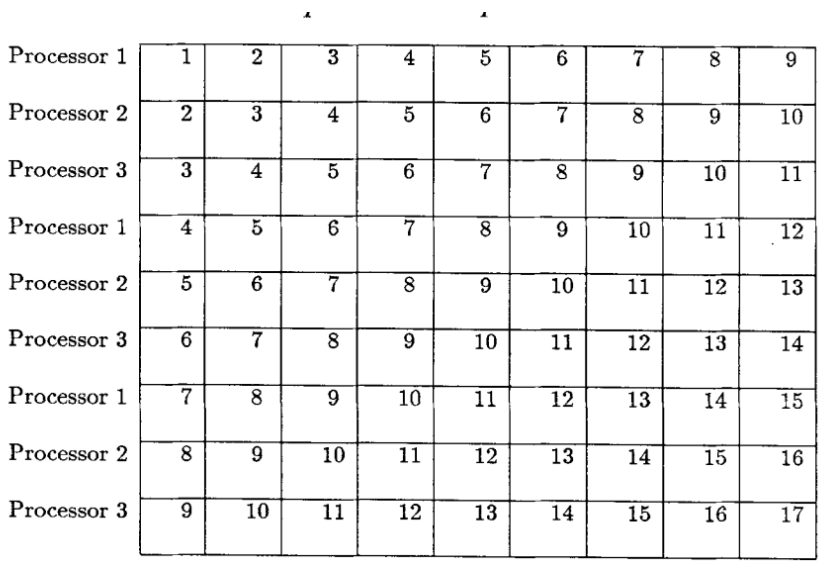
After implementing the non-parallel versions of the two above algorithms within the `SequenceAlignment_NonParallel` class, we implemented a parallel version of those algorithms following an approach presented in the following paper [4] within the `SequenceAlignmentParallel` class.

Suppose we have to align align two sequences A and B. We choose a number of threads n. Then we compute H in $\text{len}(A) + \text{len}(B) - 1$ phases. Each phase is represented by one diagonal.

In the end, each thread is responsible for several rows as we can see on Figure 2 below. But phases have to be computed in order (i.e. phase $n+1$ cannot start before phase n is over).

To synchronize the threads, we wait for all the threads to finish a phase before notifying all of them to start the next one using conditional variables.

To divide the matrix H in blocks, we have to choose `block_size_x` and `block_size_y` which will also determine the number of phases. These two parameters determine the dimensions of the blocks that we divide the matrix H into.



Processor 1	1	2	3	4	5	6	7	8	9
Processor 2	2	3	4	5	6	7	8	9	10
Processor 3	3	4	5	6	7	8	9	10	11
Processor 1	4	5	6	7	8	9	10	11	12
Processor 2	5	6	7	8	9	10	11	12	13
Processor 3	6	7	8	9	10	11	12	13	14
Processor 1	7	8	9	10	11	12	13	14	15
Processor 2	8	9	10	11	12	13	14	15	16
Processor 3	9	10	11	12	13	14	15	16	17

Fig. 1. Computational Steps for 1×1 Blocks in a 9×9 Matrix.

Figure 2: Parallelization of the computation of the H matrix with 3 threads, `block_size_x`= 1, `block_size_y` = 1. [3]

4 Performance of the algorithms depending on the inputs sizes, number of cores, and the similarity of the input sequences

First, we know that the performance of the algorithms does not depend on the similarity of the input sequence since we construct the H matrix and the traceback matrix for all cells regardless of what sequences we have. Therefore the time required to run our algorithms depends only on the length of our sequences, the number of threads, and the sizes of the blocks. Note that this would be different if we had considered more complex algorithms like the Myers-Millers algorithm which would depend on the similarity of the sequences.

In our case, the most time consuming part is to compute the H and traceback matrices, which happens within the `compute_score_matrix` function of the `SequenceAlignmentParallel` class. We only measure the time taken to run this function in the following analyses.

Concerning, the performance of the algorithms depending on the inputs size. In the following graph (Figure 3), we can observe the time in microseconds taken for the alignment of two sequences of same size using between 1 and 11 threads.

The time increases with the length of input as expected since the computations of H and the traceback matrix are more significant.

As for the number of threads, from 1 threads to 8 threads, the algorithm is generally quicker for a higher number of threads (the quickest being 8 threads) which makes sense since we make more use of parallelization. An interesting fact is that for sequences whose size is inferior to 700, the differences between 3 and 8 threads are harder to see, as for a higher number of threads a lot of time must be spent in synchronization and organization of the threads. For a number of threads higher than 8, we see that the performance decreases again as the machine we are using has 8 cores.

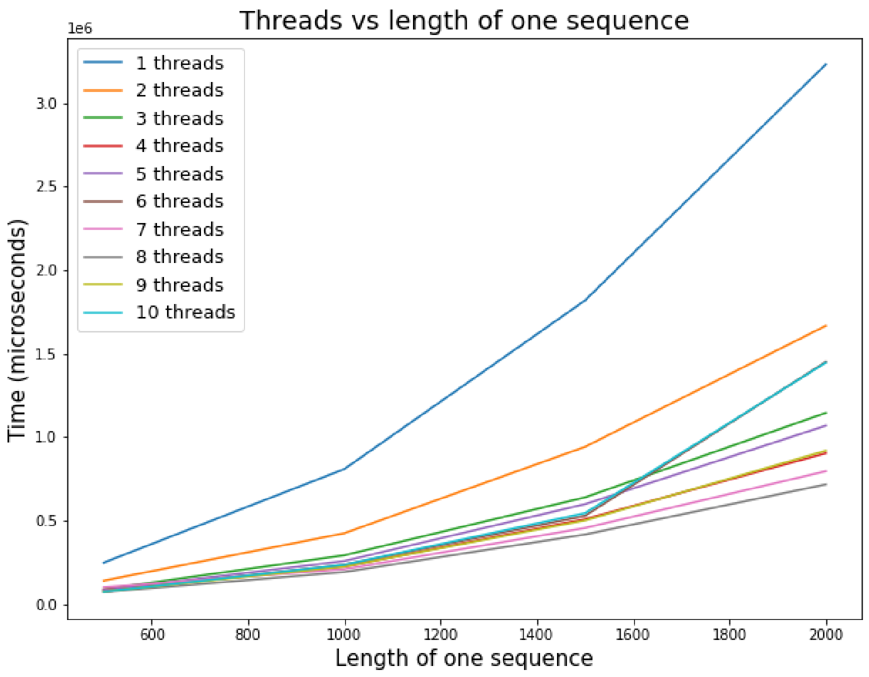


Figure 3: Graph of number of threads vs length of one sequence

In the following graph (Figure 4), we graphed the number of thread vs. the size of the matrix. We observe the same results as before except this time, the time increases linearly in function of the size of the matrix. To obtain the size of the matrix, we simply consider the square of the length of one sequence. *This is further evidence that the time complexity of our parallel algorithm only depends on the size of the matrix.*

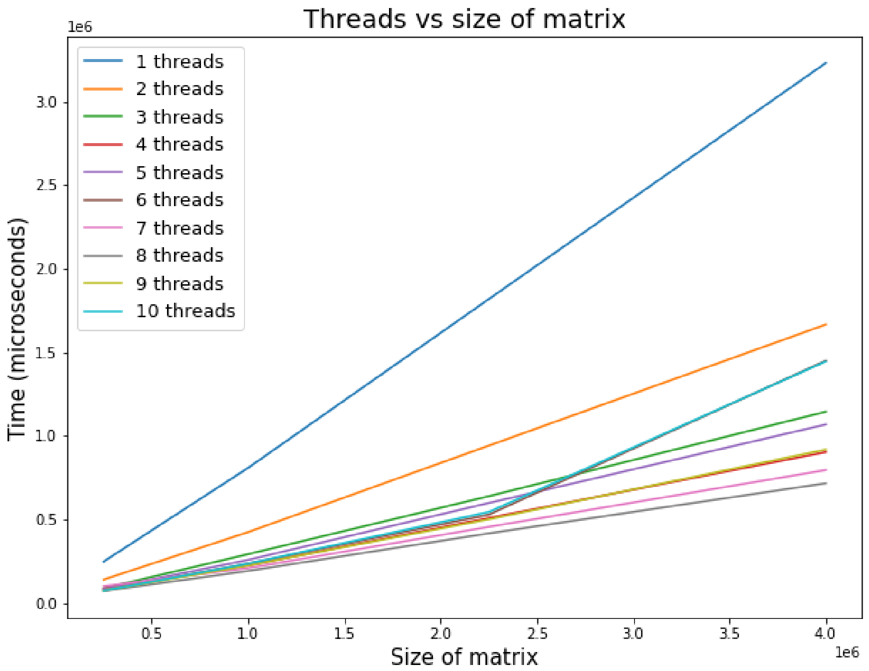


Figure 4: Graph of number of threads vs length of the sequence to align

Let us now look at the performance of the algorithms in function of the `block_size_x` and the `block_size_y` (Figure 5).

For several input sequences, there seems to be two plateaus in the evolution of time in function of the block size. The two plateaus are approximately around the same block size: one from 1 to around 40, and the other one from 40 and above.

This is probably because the optimal time, once 40 blocks is reached, the parallelization is no longer put to use. So, the overall time jumps to the second plateau. These values were computed for sequences of different lengths for 8 threads.

Conclusions

- We observe a great increase in performance going from the non-parallel implementation to the parallel implementation of the algorithm with higher number of threads.
- There is a tradeoff we can establish with the block size. Higher block sizes means lesser number of phases to compute the matrix. However, it also means that each phase takes longer to compute.

  On the other hand, a lower block size means that more time needs to be spent on synchronizations between the threads because the computations within each phase are very small in number. Thus, a compromise can be found to obtain an optimal block size.
- Theoretically, we expect a time complexity of $O(\frac{mn}{p})$ where p is the number of processors and m, n refer to the length of the sequences of A and B. We try to plot this relation in Figure 6 where we use a log-log plot of time vs. number of threads. Indeed, we observe that the log-log plots are a straight line which corresponds to our predicted time complexity of $O(\frac{mn}{p})$.

- With our parallel implementation, we fail to obtain any improvements on the space complexity. We still require $O(mn)$ space to compute alignments. Although the state of the art algorithms can achieve $O(\frac{m+n}{p})$ space complexity in the parallel case by making use of linear space alignment algorithms like the Myers-Millers algorithm [5, 6].

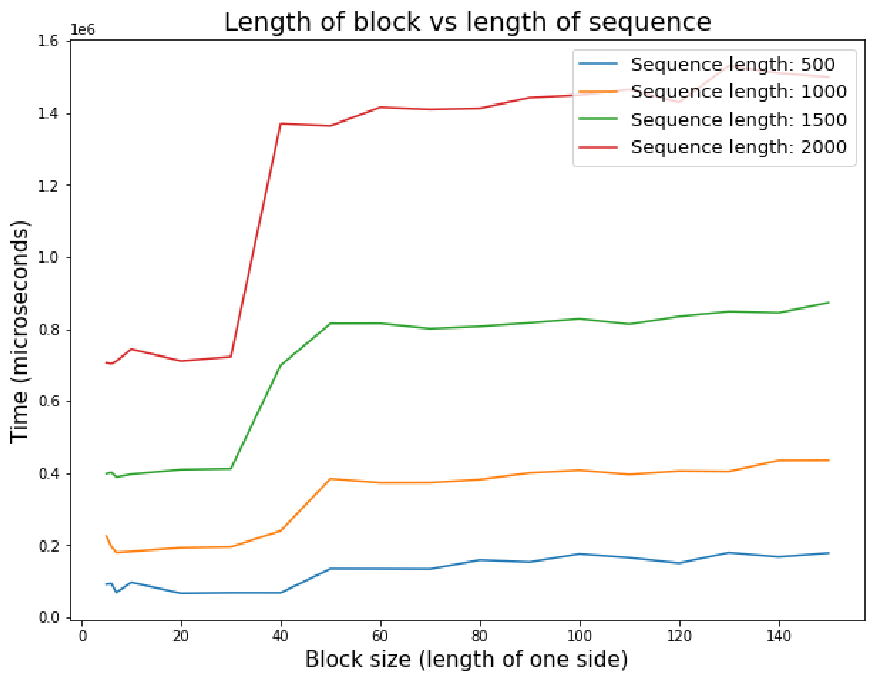


Figure 5: Variation of runtime based on block size along one direction

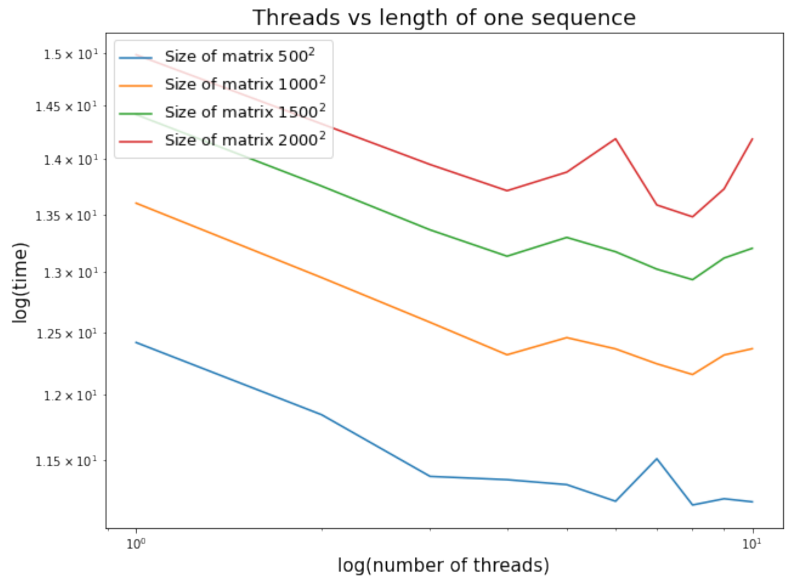


Figure 6: Number of threads vs runtime for a specific size of the matrix

References

[1] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[2] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[3] Vladimir Likic. The needleman-wunsch algorithm for sequence alignment. *Lecture given at the 7th Melbourne Bioinformatics Course, Bi021 Molecular Science and Biotechnology Institute, University of Melbourne*, pages 1–46, 2008.

[4] Elizabeth W Edmiston, Nolan G Core, Joel H Saltz, and Roger M Smith. Parallel processing of biological sequence comparison algorithms. *International Journal of Parallel Programming*, 17(3):259–275, 1988.

[5] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17, 03 1988.

[6] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, 2004.