Parallel Sequence Alignment

Ángela Garcinuño Feliciano, Lucas Massot

GitHub: https://github.com/agarfel/Parallel-Sequence-Alignment

1 Introduction

Sequence alignment is a central problem in bioinformatics. Many biological studies rely on the comparison of sequences, either DNA, RNA or protein, but to compare such sequences, we first need to align them. With recent advancements, the length of the sequences we work with has increased, reaching over 10^8 base pairs. To speed up the process, parallel versions of traditional sequence alignment algorithms have been explored.

In this project, we implement a simple time-optimal parallel version inspired from Rajko and Aluru (2004) [1], which aims to achieve space and time optimality. We then attempted to expand our algorithm to obtain space optimality, but did not obtain a fully working code due to certain issues.

2 Sequence Alignment Basics

There are several approaches to sequence alignment. In terms of alignment types, there are Global and Local alignments. In Global alignments all the characters of the sequences must be present, while local alignments contain only a subset of the characters of the sequences.

There are also different scoring schemes. We use an affine gap scheme, where we assign a score of $+1$ for matches, and a penalty of -1 for gaps. Additionally, we have a -2 penalty for gap creation.

Optimal Pairwise Biological Sequence Comparison algorithms use Dynamic Programming (DP) matrices to obtain the results. These matrices are obtained using recurrence relations on previously calculated DP values. There are three approaches to computing a DP matrix: by row, by column or antidiagonal (Fig 1).

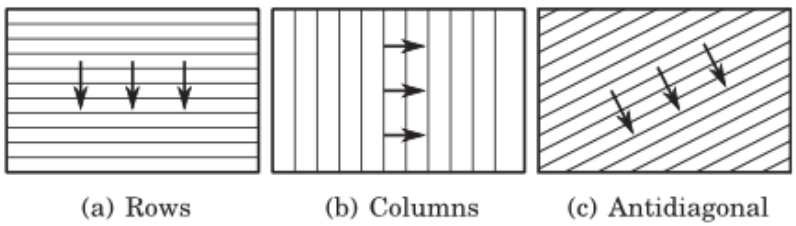

Figure 1: Approaches to DP Computation.

Once the DP is calculated, the optimal score value is stored in the bottom right cell. The alignment is obtained using a traceback algorithm on the computed DP table.

For our algorithm: we define an alignment as a list of ordered pairs $C = (i_0, j_0), (i_1, j_1), ...(i_k, j_k)$, where i_0 (resp. i_j) is either the index of a character in A (resp. B), or -1 if it is a gap. There have three types of alignment: type 1 if $i_k \geq 0$ and $j_k \geq 0$ (a_{i_k} is matched with b_{j_k}), type 2 if $b_i \geq 0$ and $a_i = -1$ (b_{j_k} is matched to a gap in A) and type 3 if $a_i \geq 0$ and $b_j = -1$ (a_{j_k} is matched to a gap in

B). We then define three DP tables: T_1, T_2 and T_3, where in T_k, the last pair of the alignment must be of type k.

3 Simple Algorithm

The simple algorithm that we have implemented focuses on computing the DP table in a concurrent manner, following the relations given in the paper we studied. Our goal is to compute three tables T_1, T_2 and T_3 as fast as possible to get the score and the alignment faster than any sequential implementation of the same relations.

We remind the reader that the T_1, T_2 and T_3 tables that we are interested in are matrices with n rows and m columns, where n is the size of the sequence A and m the size of the sequence B. The algorithm first divides the number of columns m by the amount of processors p we are using. Then each thread that we create is given $\frac{m}{p}$ columns of the tables to work on, except the last thread which is also given the reminder of the division to ensure that we consider the full sequence B.

Our algorithm takes advantage of the fact that we can compute these tables row by row. In fact, each thread is computing each of its rows (within its columns) sequentially. However some computations depends on previous cells, hence for the leftmost cells of each thread $t+1$ we need a way to pass some results from thread t to thread $t+1$. To solve that problem we use a global pointer $sharingT$ that points to a table that is $6 \times p$. This table contains 6 rows for each 6 values that we want to pass from one thread to the other (previous cell on the same row and the one above it, for each table T_1, T_2 and T_3) and within these rows the thread can select its corresponding value by going to the column corresponding to its id t. Furthermore, for every thread that isn't the last one, after having computed their part of the row they input their relevant values into the table at index $t+1$ for the next thread.

Now that we have the logic of the algorithm we need to make sure that the concurrent aspect is well implemented. First of all, to avoid any race condition, the access to $sharingT$ is performed using a lock. However, since multiple thread can technically access the $sharingT$ without looking at the same values, we implemented a vector of locks of size p, and when a thread wants to access any column t it needs to take the lock t. Secondly, to make sure that the thread $t+1$ waits for thread t to have computed its row to get the last value of the row, we implemented a $working$ vector of size p. If the thread $t+1$ is waiting for t, then $working[t+1] = 1$. We keep this thread waiting with condition variables until its state in $working$ switches to 2, which means thread t is done and has set the state of thread $t+1$ to 2, and we can now compute the row. Once we are done reading we put the thread $t+1$ to 1 again and compute our row. We input the results in $sharingT$ and set the next thread to 2. In other words, $working[id] = 1$ means that the $sharingT[id]$ has been read, and $working[id] = 2$ means that the $sharingT[id]$ has been written,

It is important to understand that we aim for time and space complexity optimality, hence when computing T_1, T_2 and T_3 we actually only keep in memory the previous and the current row. Having computed all the rows we are left with the last row, where we can find the score in the very last column of the row of the last processor, in the table T_1. Moreover, we decided to avoid traceback in this implementation, hence you can also find the alignment in this cell. In fact, we designed our tables to be filled with "Entries" which is a class containing a value and an alignment. At every computation, we take the alignment of the cell from which the current cell originates, and depending on which table we are in, we append to this alignment either a match or a gap in A or B. It only remains to return the final score and final alignment.

4 Complex Algorithm

Given two sequences A and B of lengths n and m respectively, we want to find the global optimal alignment (one of them if there are several) and return the alignment and score. We proceed in two steps: decomposition and sub-problem solving, following a divide-and-conquer approach.

The first step is to decompose the problem into several sub-problem. The main difficulty is knowing where to split the problem, since we want to be able to combine each partial solution to find the optimal alignment. To achieve this, we split the Dynamic Programming (DP) table we want to compute into special rows and special columns. We then compute the DP values for r_{mid} (the middle row), and find the optimal point. This way, we are certain that the optimal global alignment passes through that point. While computing the DP values, we keep track of the origin cell, which is a cell in the left-most (respectively right-most) special column through which the alignment passes. We can thus split our problem in 3: two parts in which we decompose recursively, and a block where we directly solve the alignment problem. Throughout this decomposition, it is important to keep track of the type of alignment each sub-problem starts and ends with. There are three types: match, gap in A and gap in B.

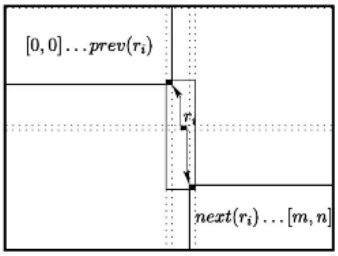

Figure 2: Caption

4.1 Decomposition

To effectively carry out the decomposition concurrently, Aluru and Rajko make each thread responsible for a different set of columns. In comparison, we have two processors for each set: one to compute T and another to compute T^R concurrently. This not only allows us to distribute the work between threads and proceed with the decomposition and solving steps without creating or killing threads, but it also reduces the memory usage, since each thread only stores a different part of the sequence B.

To find the cell to subdivide the problem, we need to compute the values for the row r_{mid}. To do so, we compute the values for T_1, T_2 and T_3 for rows r_{top},...r_{mid} with one processor and the values for T_1^R, T_2^R and T_3^R for rows $R_{mid}, ..., r_{low}$ with another. While doing this, instead of storing the whole table, we keep only the current and previous rows in memory, improving the performance in terms of memory. Since we are not computing the values of T and T^R for all rows, this approach is not slower than computing the whole matrix.

The main challenge in this project is dealing with the intra-row dependencies when computing the values. These dependencies are diagonal and sideways, where $[i, j]$ depends only on $[i-1, j]$, $[i-1, j-1]$ and $[i, j-1]$. We thus compute the matrix diagonally by rows. That is: each processor computes their row, and waits for the previous processor to compute and pass the previous value before computing that row. We use locks and conditional variables to synchronize the threads. This method allows us to compute the matrix row by row for each processor, keeping efficient memory usage, and parallelize the computation without creating or killing threads.

4.2 Synchronisation

Since we aim to only create the threads once, we need a way to synchronise them. For this, we use several locks, conditional variables and shared vectors. Given p processors, we define *update* (a conditional variable), *workers* (a vector of size p of threads), *working* (a vector of size p of integers) and *working_mutexes* (a vector of size p of mutexes). We also have several shared variables to communicate information between threads: *sharingT*, *sharingTRev*, *sharingOpt* and $Info$.

For the decomposition stage, we have a group of threads working to solve the problem. The thread with the lowest id is defined to be the *head* and coordinates the other threads in the group. Each

thread has an id between 0 and $p-1$. The value of $Working[id]$ indicates which part the thread id is working on. Values 1 and 2 are used to communicate dependencies in the DP matrices in the same way we do for the simple algorithm.

Once the odd threads have computed their respective part of T^R, they update the section of *sharingTRev* they are responsible for. Since there is never any overlap, it is not necessary to take a lock. Once they have they take the lock $working\_mutexes[id]$ and set $working[id] = 4$ and notify all with *update*, to indicate they have updated *sharedTRev* and *working*. Meanwhile, when the even threads finish computing T, they wait on update until $working[id+1] == 4$. Then, they take the updated values of *sharingTRev*, and use them to compute the optimal point within the block they are responsible for. Once this is done, they update *sharingOpt*, again without using locks since the values are never accessed concurrently, and set $working[id] = 4$, (taking $working\_mutexes[id]$ first). They then notify all with *update*.

At this point, we have computed all the partial optimal values within this decomposition problem. All non-head threads then wait until $working[id]! = 4$. The head thread uses *sharedOpt* to obtain the global optimal value within the group, splits the problem as necessary and updates $info[k]$ for each thread k within the group, sharing which problem each thread is now working on. After this, it sets $working[k] = 1$ for each of those threads and then notifies all using update so they all continue the execution.

4.3 Sub-Problem Solving

After decomposing, we need to solve the following problem: given two sequences A and B and the start and end alignment type, we want to obtain the optimal alignment and score. This sub-problem is solved by one thread since it usually ranges from one special column to the next. However, in the case where we reach through the recursion a problem with only two processors we simply ask the first processor to solve the entire sub-problem (from a special column to the next's next) and let the second one terminates, as it would be too small to decompose in three parts.

Once we have the setup we use the start type to initialize the outermost row and column of each table and compute the score by constructing the DP in a sequential manner using the recursion over T_1, T_2 and T_3 provided in the paper. While we construct T_1, T_2 and T_3 we also construct BT_1, BT_2 and BT_3 which keep track of the table of origin of each cell in their respective tables to later facilitate the trace back to obtain the alignment. Once all the tables constructed we use the end type to recover the score and trace back the construction using BT_1, BT_2 and BT_3 to create the alignment character by character. Starting by finding out from which table the rightmost bottom corner originates we trace back, using the BT tables to know in which table to look in the next iteration. Depending on if the current cell $[i, j]$ originates from T_1, T_2 or T_3 we respectively match the characters $A[i], B[j]$, match a gap with $B[j]$, or match $A[i]$ with a gap.

4.4 Joining Scores and Alignments

Before starting the threads we initialize a vector whose length is the amount of processors. Each of the threads are given the reference to an index in that vector where they can write their result. Once the processor reaches a sub-problem to solve (cf 3.3) it can then write the score and the alignment of this sub-problem in the global vector. We mentioned that some processors never solve any sub-problem : In the case where the problem to solve ranges over only two processors, the first one solves the whole problem as a sub-problem and the second one terminates, leaving the element at its index in the vector as it was initialised. This shows that we need to be careful about how the vector is initialised in the first place. When all the processors have terminated we simply add up the scores and concatenate the alignments to get the global score and the global alignment. In that sense, initializing the vector with a 0 score and an empty alignment solves the issue raised earlier.

4.5 Our Issue

As mentioned previously, our implementation of this algorithm does not work perfectly. When decomposing, we need the problems to overlap, and use s and e the start and end types to indicate the type of alignment we start and end with. If we don't make them overlap, we might ask a sub-problem to have an end type which is actually the alignment type of the next pair in the global alignment, leading to unexpected partial results if the problem is split right before or right after a change in the alignment type. While we successfully implemented this overlap, we did not have the time to get the sub-problem solver function to account for this overlap. It needs to both correctly impose the start and end types as well as exclude part of the alignment and score if there is an overlap, to avoid having the overlap to be accounted for twice in the final result.

However, the rest of this algorithm works correctly, and it only fails when a split occurs on a change of alignment type. Hence, although the results are not entirely correct, it is still interesting to compare the performance of this algorithm to the simpler one.

5 Results

Let us now look into how well the algorithm performs. We use different pairs of sequences with lengths that vary from around 100 to around 5000 to get a full range of the potential of the algorithm. We first observe the results that we get for each of these pairs and then comment on the overall performance of the algorithm. We are running these tests on the lab machine, with 24 cores.

We first test our algorithm on two sequences of insulin in the human and the bovine, that are 110 and 105 characters each.

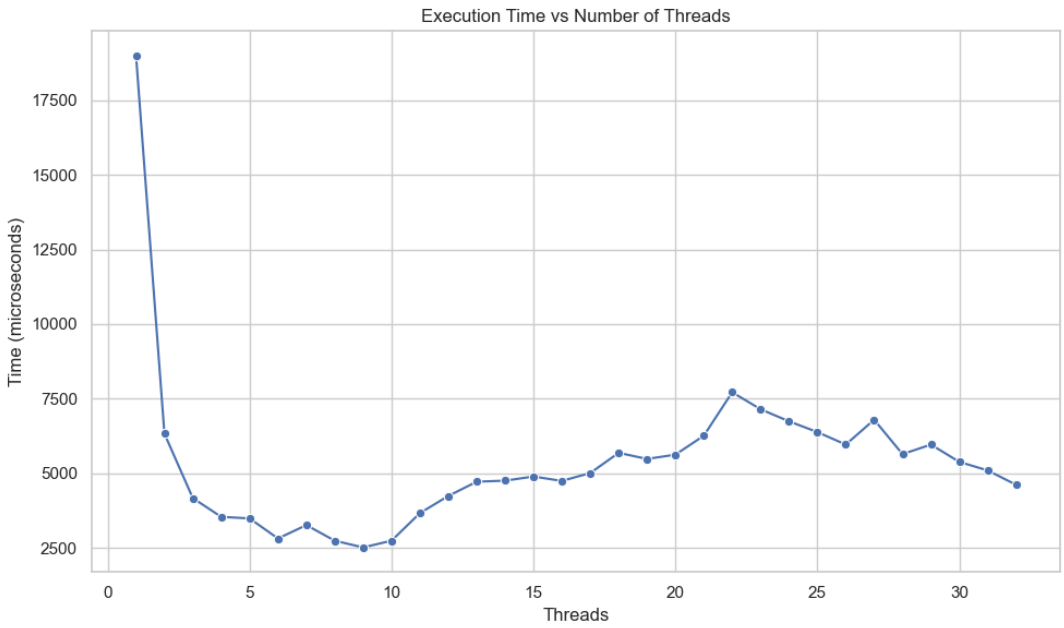

Figure 3: Computation time against number of processors

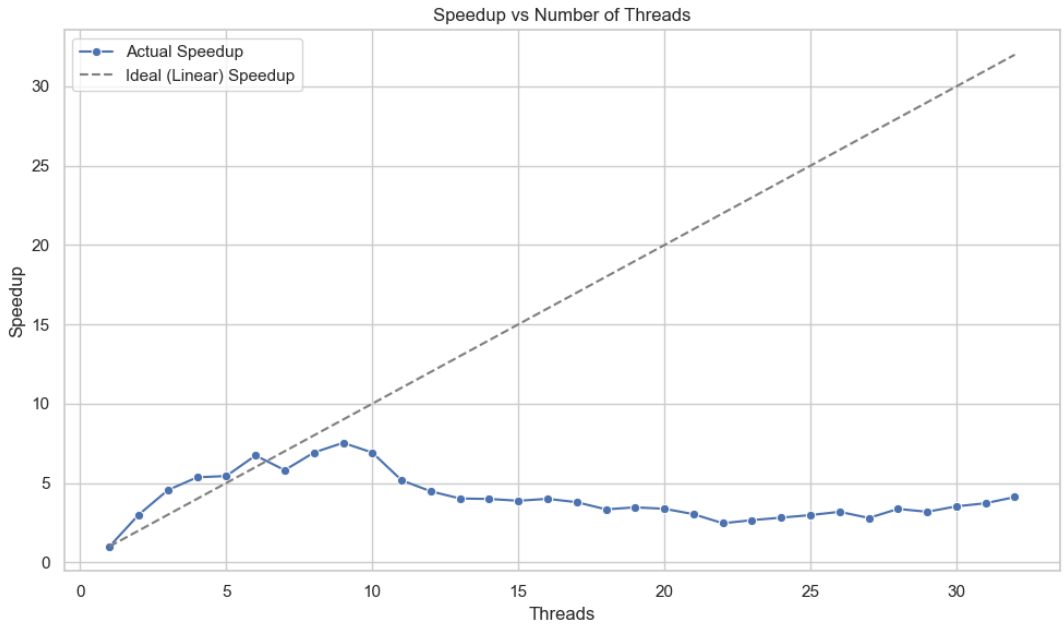

Figure 4: Speed up against number of processors

Secondly, we test our algorithm on two sequences of Tyrosine-protein kinase receptor protein from the human and Insulin-like growth factor 1 receptor protein from the mouse, around 1366 and 1373 characters respectively.

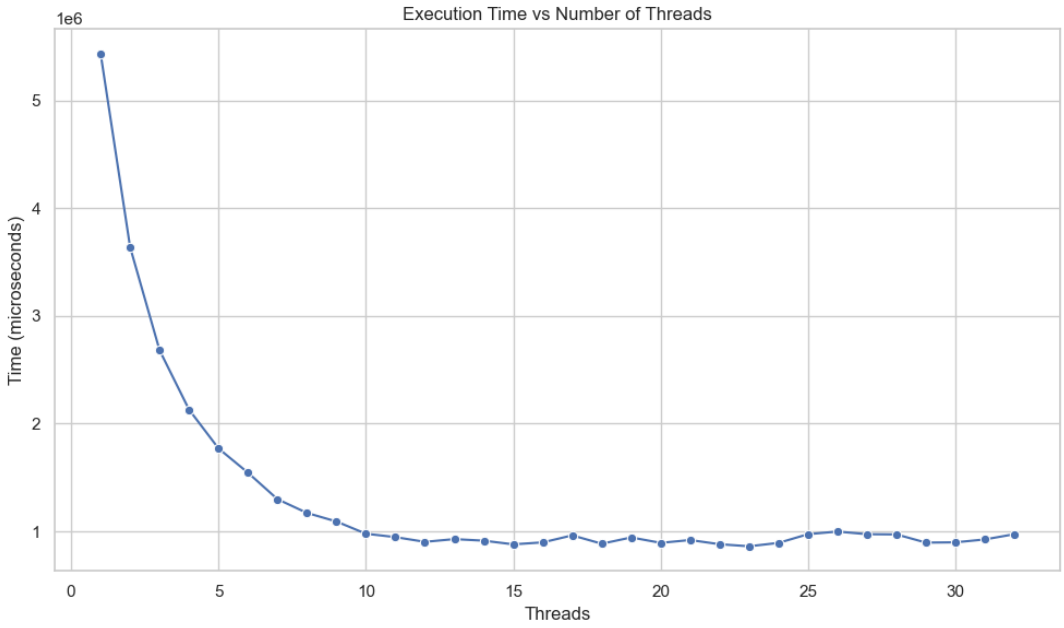

Figure 5: Computation time against number of processors

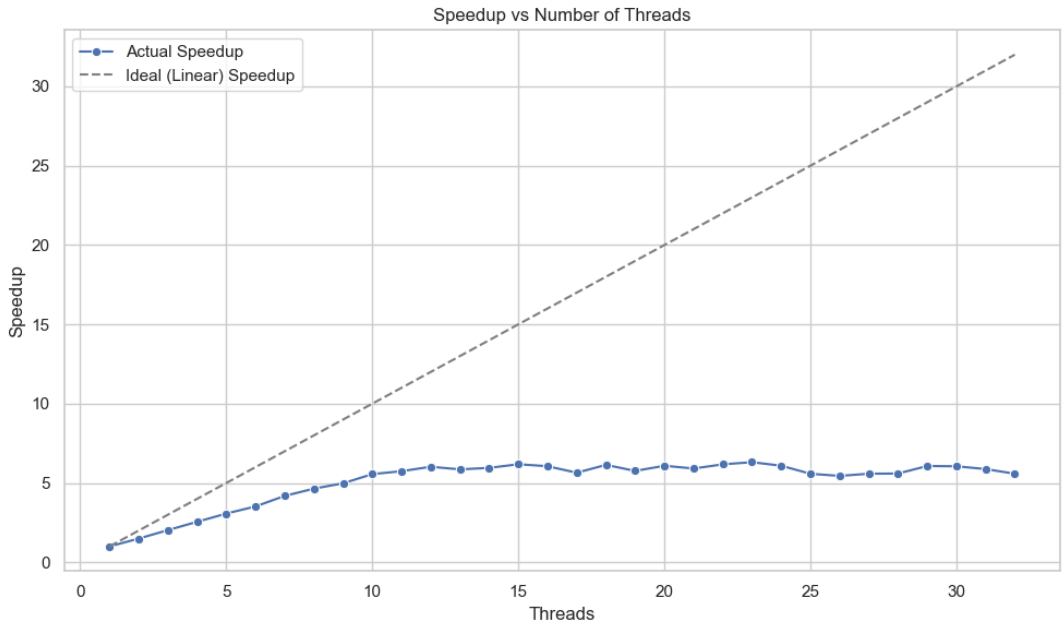

Figure 6: Speed up against number of processors

Finally, we test our algorithm on two sequences of Bridge-like lipid transfer protein from the human and from the mouse, both around 5005 characters.

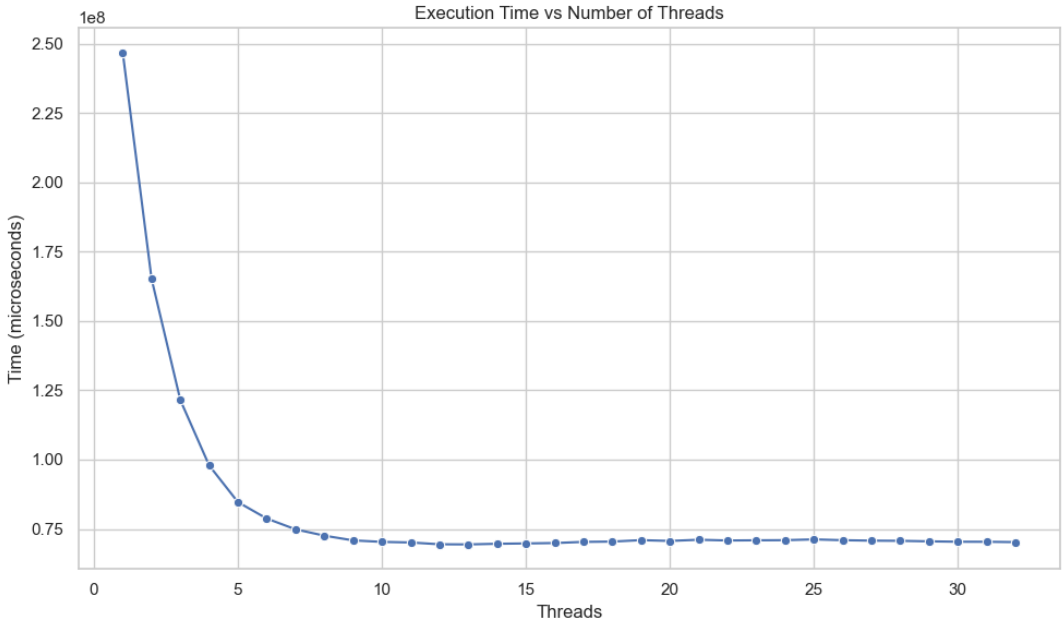

Figure 7: Computation time against number of processors

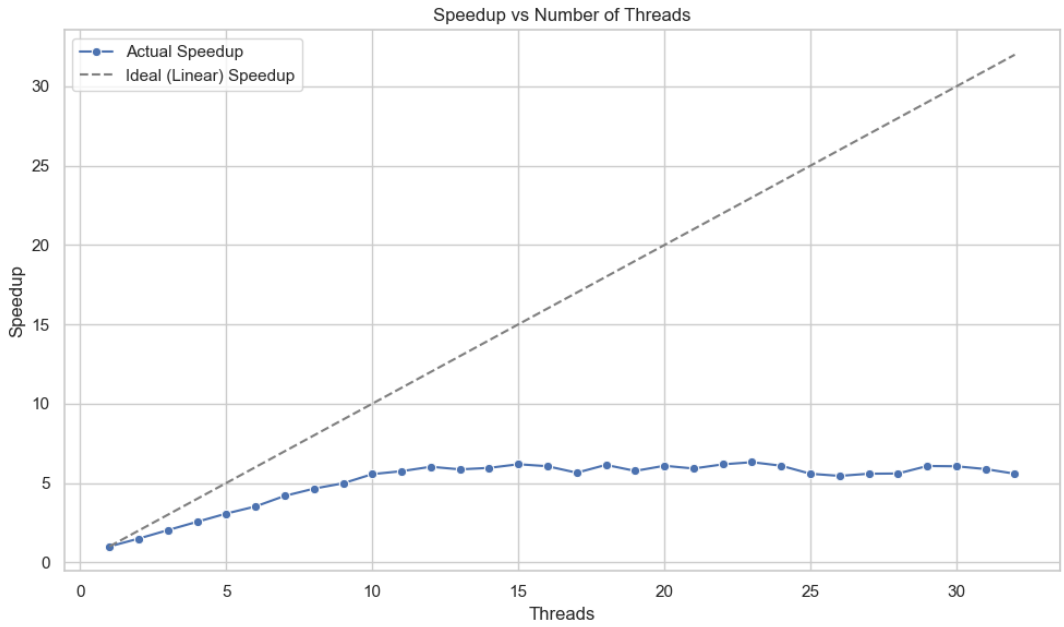

Figure 8: Speed up against number of processors

A first comment that can be made is that the overall shape of these graphs are very similar : we notice first a sharp decrease in computation time as the number of thread increases (inverse proportional decrease) which then flattens. On the other hand, the speed up starts as a linear increase but flattens really quickly. Of course the results of these tests are more relevant on the second and third pair, as the first one exhibits almost instantaneous computation, which we can see in how irregular the results for this pair is. The two others share really similar overall shapes for their graphs, which is the one described before.

While the parallel algorithm does improve performance, the speed-up is inherently bounded by inter-thread communication overhead. This leads to an eventual plateau in the speed-up. This bottleneck is particularly noticeable for short sequences, where having too many threads becomes counter productive due to this waiting time, compared to computing the DP matrix sequentially.

Threads	**Time (μs)**	**Speedup**
1	5722623	1.00
2	3762647	1.52
3	2700309	2.12
4	2111965	2.71
8	1204505	4.75
16	851548	6.72
24	811688	7.05
32	836409	6.84

Table 1: Execution Time and Speedup vs Number of Threads for Insulin-like sequences

6 Discussion

Our simple parallel implementation has very good results. We can see a significant speed-up when using more processors, until a certain amount. This is due to the necessary waiting times between threads for coordination. Our algorithm combines both the row and antidiagonal approaches to DP computation. Each processor computes row by row, but we are splitting the DP into blocks of one row and a set of columns, and these blocks are computed in an antidiagonal approach. By having each

thread to be responsible for a block of columns of the computation, we can create threads only once, improving performance compared to creating threads at each step of the computation. Additionally, it improves space complexity, since each thread only copies a portion of the sequence B. Furthermore, since we compute the DP row-by-row, we only ever store two rows of the matrix, further improving space complexity.

This algorithm can be further improved with the complex algorithm we attempted to implement. By decomposing the global problem, we eventually only compute certain blocks of the DP matrix along the global alignment path, instead of the full DP matrix. This is a significant improvement in terms of space complexity. Further improvements to the algorithm would be to use several threads to solve each sub-problem, instead of only the head thread. This approach would be the same as the simple implementation, but taking into account start and end types.

7 Responsibilities

We both worked together throughout most of the project. In general, Ángela focused more on the design and implementation of the complex algorithm while Lucas focused more on the implementation of the simple algorithm and the performance analysis.

Ángela:

- Complex algorithm: decomposition and thread communication and synchronisation
- Simple algorithm: wrote outline

Lucas:

- Complex algorithm: sub-problem solving
- Simple algorithm: completed algorithm

References

[1] Stjepan Rajko and Srinivas Aluru. Space and time optimal parallel sequence alignments. *Parallel and Distributed Systems, IEEE Transactions on*, 15:1070– 1081, 01 2005.