Project Report - Parallel sequence alignment

Vincenzo Politelli, Elie Huynh

May 2024

Github repo: https://github.com/vincenzo-politelli/Project

Contents

1 Introduction

One of the core challenges in computational bioinformatics is sequence alignment. The goal of this project is, given two sequences of nucleotides (A,C,G,T), to compute and find the "best" alignment, which is in a lot of ways similar to computing the dissimilarity between two strings.

A natural way to solve this problem is to use dynamic programming, which results in an algorithm with time complexity $\mathcal{O}(nm)$, where n and m are the lengths of the sequences we want to align. When parallelized, this algorithm will take time complexity $\mathcal{O}(\frac{nm}{p})$, where p is the number of processors. In this project, we will first present a sequential algorithm to solve the sequence alignment problem, and then the parallelized algorithm, followed by the experimental speedups that we observed.

In order to test our project, one can use "make" then "./main" which will run a few tests that we created.

2 Sequential Algorithm - Needleman-Wunsch

2.1 Implementation - Needleman-Wunsch

The intuition behind the following sequence alignment algorithm (Needleman-Wunsch algorithm) [Wik] is to compute and optimize for the alignment with the best score over the whole sequence we want to align. The score can be computed with the following parameters:

- the match score, the score given when two characters coincide (in our code, it is a positive integer)
- the mismatch cost, the score given when two characters do not coincide (in our code, it is a negative integer)
- the gap cost, the score given when inserting a gap ("-") (in our code, it is a negative integer)

The implementation of the sequential algorithm can be divided into 3 steps, which are the following:

- Initialize the Dynamic Programming Table/the traceback matrix (in our code, "initializeDPTable()")
- Fill the matrix (in our code, (in our code, "fillDPTable()")
- Compute the traceback, and return the optimal alignment (in our code, "traceback()")

2.2 Rundown of each step

More precisely, here is a rundown of each step.

For step 1, the Matrix is initialized with dimensions (m+1) x (n+1) where m is the length of seq1 and n is the length of seq2. The first row and the first column are initialized with the cumulative gap penalties, setting up the base cases for the alignment.

For step 2, the Matrix is filled iteratively by comparing characters from the sequences seq1 and seq2. For each cell (i, j) in the Matrix, three possible scores are considered:

- `match`: If the characters match, the score is taken from the diagonal cell $(i-1, j-1)$ with an additional match score. If they do not match, the mismatch cost is applied.
- `delete_op`: The score from the cell above $(i-1, j)$ with an additional gap cost.
- `insert_op`: The score from the cell to the left $(i, j-1)$ with an additional gap cost.

The maximum of these three scores is chosen as the value for the cell (i, j).

For step 3, The `traceback` function constructs the optimal alignment by starting from the bottom-right corner of the Matrix and tracing back to the top-left corner. The traceback process constructs the aligned sequences `align1` and `align2` by choosing the path that resulted in the current cell's score:

- Diagonal move for a match/mismatch.
- Vertical move for a deletion (gap in `seq2`).
- Horizontal move for an insertion (gap in `seq1`).

2.3 Implementation Smith-Waterman

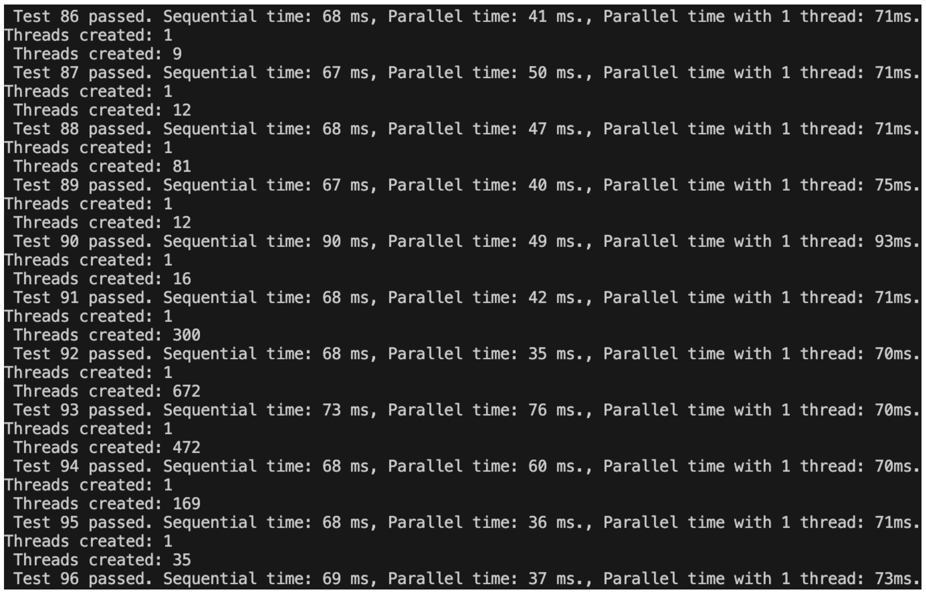
Our project also includes the Smith Waterman Algorithm for sequence alignement. The logic is essentially the same, except we don't allow for negative values in the Traceback matrix. We thus take the maximum of the three scores indicated above and 0. Moreover, we record the best score in the Traceback matrix, as we will start the traceback from this cell.

3 Parallel Algorithm

3.1 Needleman-Wunsch

The approach that we are going to present is described in [SSRV05]. In the parallel algorithm, we will aim to concurrently compute the function "fillDPTable()". Intuitively, we can separate the matrix into blocks of some size. Each thread will thus be responsible for its block. Of course, from the above, we observe that some block have dependencies, e.g. block[1][1] cannot begin unless block[0][1] and block[1][0] have been computed. We can do so by introducing the following in our *SequenceAlignmentParallel* class: a vector of vectors containing the state of the blocks, i.e. true if the block was computed, false otherwise.

The parallel algorithm calls the "fillDPTable()" which will iterate over each, creating a thread for each block. The synchronization mechanism is insured by a mutex and a condition variable. Launching a few tests for randomly generated sequences of length 1000 and length 995, for randomly generated number of threads, match score 5, mismatch cost -3, gap cost -4, we get significant speedup in most cases:



```
 Test 86 passed. Sequential time: 68 ms, Parallel time: 41 ms., Parallel time with 1 thread: 71ms.
Threads created: 1
 Threads created: 9
 Test 87 passed. Sequential time: 67 ms, Parallel time: 50 ms., Parallel time with 1 thread: 71ms.
Threads created: 1
 Threads created: 12
 Test 88 passed. Sequential time: 68 ms, Parallel time: 47 ms., Parallel time with 1 thread: 71ms.
Threads created: 1
 Threads created: 81
 Test 89 passed. Sequential time: 67 ms, Parallel time: 40 ms., Parallel time with 1 thread: 75ms.
Threads created: 1
 Threads created: 12
 Test 90 passed. Sequential time: 90 ms, Parallel time: 49 ms., Parallel time with 1 thread: 93ms.
Threads created: 1
 Threads created: 16
 Test 91 passed. Sequential time: 68 ms, Parallel time: 42 ms., Parallel time with 1 thread: 71ms.
Threads created: 1
 Threads created: 300
 Test 92 passed. Sequential time: 68 ms, Parallel time: 35 ms., Parallel time with 1 thread: 70ms.
Threads created: 1
 Threads created: 672
 Test 93 passed. Sequential time: 73 ms, Parallel time: 76 ms., Parallel time with 1 thread: 70ms.
Threads created: 1
 Threads created: 472
 Test 94 passed. Sequential time: 68 ms, Parallel time: 60 ms., Parallel time with 1 thread: 70ms.
Threads created: 1
 Threads created: 169
 Test 95 passed. Sequential time: 68 ms, Parallel time: 36 ms., Parallel time with 1 thread: 71ms.
Threads created: 1
 Threads created: 35
 Test 96 passed. Sequential time: 69 ms, Parallel time: 37 ms., Parallel time with 1 thread: 73ms.
```

Figure 1: Preliminary tests to investigate correctness (i.e. that the sequential and parallel algorithm yield the same results) and speedup

3.2 Speedup Analysis

In order to analyze the speedup, we computed the time taken for alignements of randomly generated sequences ranging from length 10 to length 999 and put them in a CSV file available on github. Overall, we observed the following:

- In order to observe significant speedups in higher length sequences, we need to use enough threads
- Not using enough threads might cause the sequential algorithm to be faster.

- Using too much threads might cause the sequential algorithm to be faster.

Here is a graph analysing the performance of the Needleman-Wunsch algorithm depending on the number of threads used.

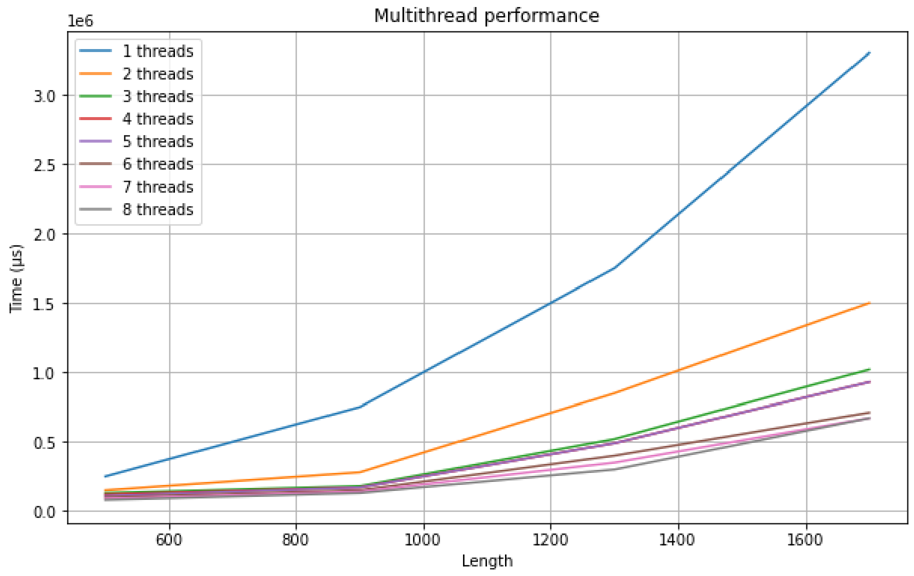


Figure 2: Plot of the performance

Since we are using a machine with eight processors, we can clearly see the speedup that is implied by the use several processors, which is a confirmation of the (expected) benefits of parallelization.

We can now look at the size of the blocks and how it impacts the performance. We took two random sequences of length 500 (in the alphabet of amminoacids) and looked at the relative performance when changing the block size. We have the following table:

10	20	30	40	50	60	70
0.725 μs	0.783 μs	0.821 μs	1.130 μs	1.135 μs	1.415 μs	1.423μs

Table 1: On top, the size (in rows and columns) of the block, on the bottom, the performance of the algorithm

We can see that a block of small size is not impacted by the possible drawback caused by the syncronization of the various threads. Moreover, the bigger the block, the worse the performance. Thus, the presence of small the blocks helps rescaling and exploits the parallelisation.

4 Conclusion

Our algorithms (Needleman-Wunsch and Smith-Waterman) are both time optimal, achieving a time complexity of $\mathcal{O}\left(\frac{mn}{p}\right)$ where m and n are the length of the sequences and p is the number of threads used.

It does not, however, achieve space optimality. Indeed, since we made use of the `C++` standard library, we could not handle different processors, and allocate memory to some of them in an independent fashion. Our algorithms have thus suboptimal space-complexity $\mathcal{O}(mn)$. More sophisticated algorithms such as [RA04] allow to achieve the space optimal bound of $\mathcal{O}(\frac{m+n}{p})$.

References

[RA04] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, 2004.

[SSRV05] F. Sanchez, E. Salami, A. Ramirez, and M. Valero. Parallel processing in biological sequence comparison using general purpose processors. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 99–108, 2005.

[Wik] Wikipedia. Needleman–Wunsch algorithm — Wikipedia, the free encyclopedia. [Online; accessed 30-May-2024].