Università degli studi di
Milano-Bicocca

Parallel computing systems

Final Project

Parallel longest common subsequence (LCS)

Authors:
Poveromo Marco - 830626 - m.poveromo@campus.unimib.it

September 1, 2022

Abstract

In computational biology it is still a research topic to find strategies for optimizing sequence alignment. The LCS is referred to the longest common subsequence, and is an effective method to solve this problem. Generally, the approach for the LCS is the dynamic programming, and the score table can be filled in a parallel diagonal or changing the data dependency. The goal of the following work is to present and analize the use of different models related to parallel programming in solving the LCS. A comparison between OpenMP, MPI, CUDA will be shown, on a single computer having a NVIDIA GPU and a multicore CPU. Analyzes will be carried out on the executions of two variants of the algorithm with different parameters. It will then be shown how the CUDA approach will be the most efficient.

1 Introduction

The **longest common subsequence (LCS)** problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (in this case just two sequences). Bioinformatics is an expanding area of research, and the sequence alignment problem is an effective method to analyze the similarity of different biological sequences. In this field, the sequences are represented by strings, and finding the LCS of two strings is equivalent to findind the longest common subsequence of two biological sequences.

Since genomic sequences are typically very long, sequential algorithms become slow in practice. It was necessary to introduce new techniques to optimize the computation time. Parallelizing the LCS computation is the **objective** of this work.

The most used approach for solving the LCS is the **dynamic programming**, that is a bottom-up approach that takes advantage of the optimal substructure of the problem. A score matrix is filled with a certain score, in this case the match of two characters in the strings increments the score by one. The score at the end of the computation represents the LCS, and the subsequence associated can be obtained by traceback on the table.

The parallelization approaches presented in this paper are, and will be called **v1** and **v2** for simplicity. The **v1** approach exploits the possibility to compute the matrix diagonally, so that each diagonal can be computed parallel. The **v2** approach takes advantage of the presented work [] to modify the data dependency, in order to increase the parallelism compared to v1.

For each type of parallelization (OpenMP, MPI, CUDA) the LCS execution times will be shown for both v1 and v2, and the differences and optimizations will be discussed.

2 Longest common subsequence (LCS)

The longest common subsequence (LCS) problem[1] is related to find the longest common subsequence to all sequences in a set of sequences (often just two sequences). It is not to be confused with the longest common substring, in fact in the LCS the subsequences are not required to occupy consecutive positions.

For the general case of an arbitrary number of input sequences, the problem is NP-hard. In case of two sequences, the time complexity is $O(n * m)$ where n and m are the lenghts of the sequences.

The LCS problem can be broken down into smaller, simpler subproblems, and so on, until the solution becomes trivial. Also the solutions of the high level subproblems reuse solutions of the lower levels. For this reason the dynamic programming is used, so that subproblems aloutions are memoized and reused by higher level subproblems.

The definition of the problem is the following:

"Given two sequences X and Y, of respectivly n and m length, determine one among the longest subsequences common to X and Y."

More formally, the recursive definition of the problem is as follows:

$$LCS(X,Y) = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ 1 + LCS(X_{i-1}Y_{j-1}) & \text{if } x_i = y_j \\ max(LCS(X_{i-1}, Y_j), LCS(X_i, Y_{j-1})) & else \end{cases} \quad (1)$$

The recursion can be implemented in a bottom-up fashion, filling a score matrix. The pseudocode for the serial LCS with two sequence input is shown in alhoithm 1.

The **time complexity** of the bottom-up algorithm is $\theta(m * n)$ by occupying $\Theta(m * n)$ space in memory. This is due to the fact that the algorithm access all the cells of the score table, that is n*m in size.
In the course of the project, however, memory optimization techniques will be

Algorithm 1 Length of a longer common subsequence between two X and Y sequences of length respectively (technique: bottom-up).

```
procedure LCS(X, Y)
    for i ← 0 to m do
        C[i, 0] <- 0
    for j ← 1 to n do
        C[0, j] <- 0
    for i ← 1 to m do
        for j ← 1 to n do
            if x_i = x_j then
                C[i, j] ← C[i-1, j-1]
            else
                C[i, j] ← max(C[i-1, j], C[i, j-1])
    return C[n, m]
```

used, occupying $\Theta(m)$ and losing the traceback capability of the LCS maintaining the length of the lcs.

For **example**, given X = ATCC and Y = CCGA the score matrix generated is the following:

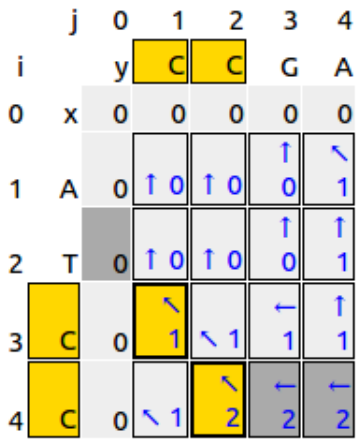


Figure 1: Example of score matrix

The arrows can be saved in a separate matrix and can be used to traceback the LCS, in this case LCS(X, Y) = CC, as in figure 1.

The final score of the highest level problem is located in the last cell of the score table, in the position (n, m), in this example the lenght of the LCS(X, Y) is 2.

2.1 [v1] Antidiagonal (wavefront) parallel LCS

In this section we will examine a first version (v1) of the parallel LCS algorithm[2], which exploits the dependency on subproblems. The parallelizable part of the algorithm is highlighted in the way in which the score table is filled. In fact what we observe is that, during the construction of the table, C[i, j] **depends on three cells** of the matrix: C[i-1 , j], C[i, j-1], C[i-1, j-1].

In other words, the computation of cell C[i, j] must have information from the top, left and top left cells to be calculated. This implies that the construction of the score table by rows or by columns is **inherently sequential** and cannot be done in parallel.

Observing the construction in figure 2 we can see how the elements on the same **antidiagonal** can be computed at the same time since each of them has

the necessary information.

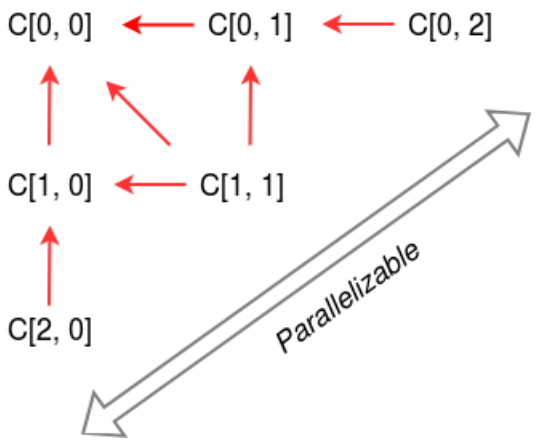


Figure 2: Filling the score table by antidiagonals

However, the computation for antidiagonals is **not embarrassingly parallel**, since one has to wait for the previous antidiagonal to finish before computing the current one.

More formally, let C be an n * m matrix, then C will have D = n + m - 1 antidiagonals d1, d2, .., dD. The parallel algorithm will have **synchronization barriers** between d_i and d_{i+1}, like in figure 3.

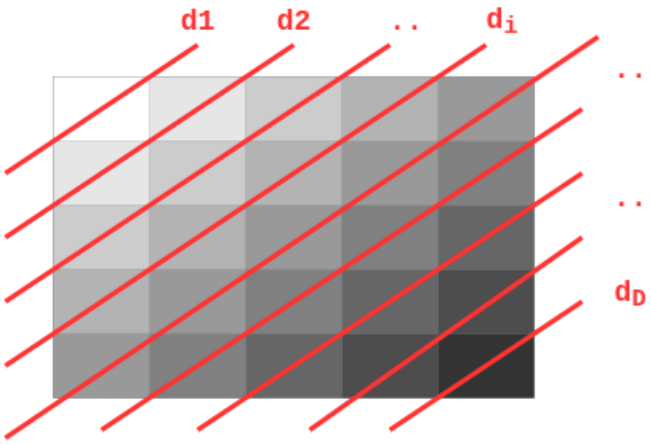


Figure 3: Antidiagonals orded

The **pseudocode** for computation of this parallel version (v1) of LCS, which

uses the ideas seen in this section, will now be presented.

The pseudocode showed in algorithm 2 presents only a high-level vision, in the next chapters it will then be implemented with the specifications of OpenMP, MPI and CUDA.

For this algorithm the three antidiagonals with dimension $1 + \|Y\|$ are initialized. It is noted that by swapping X and Y the same result is obtained without losing generality. Furthermore this dimension will be used entirely only during the computation of the main antidiagonal, while the remaining antidiagonals will **fill it only partially**. For example the diagonal d_1 will use only one entry as it has dimension 1, the diagonal d_2 will use 2 entries, etc ..

Algorithm 2 Parallel version (v1) of the LCS that computes the antidiagonals in parallel.

```
procedure LCS_v1(X, Y)
    n ← 1 + Y.length
    precPrecDiagonal = allocate(n)
    precDiagonal = allocate(n)
    currentDiagonal = allocate(n)
    for i ← 0, j ← 0 to m, n do
        diagonalSize = min(j, m - i)
        for d ← 0 to diagonalSize in parallel do
            a ← i + d
            b ← j - d
            if a == 0 or b == 0 then
                currentDiagonal[b] ← 0
            else if X[a - 1] == Y[b - 1] then
                currentDiagonal[b] ← precPrecDiagonal[b - 1] + 1
            else
                currentDiagonal[b] ← max(precDiagonal[b], precDiagonal[b - 1])
        if j == n then
            j ← j - 1
            i ← i + 1
        shiftPointersLeft(precPrecDiagonal, precDiagonal, currentDiagonal)
    return precDiagonal[n]
```

Given an n * m matrix, the algortymon computes the antidiagonals one after the other. Each antidiagonal is calculated in a parallel manner. The num-

ber of antidiagonals is $D = n + m - 1$ while the elements in them is variable 1,2,..,$min(n, m)$,..,2,1. Assuming that n = min(n, m) without losing generality, then the D antidiagonals will compute $O(n)$ elements each. The **computational complexity** T_p of the algorithm, as a function of the number of computational elements p, will therefore be:

$$T_p = O((n + m)(\frac{n}{p})) \tag{2}$$

Let p = $O(n)$ then $T_p = O(n+m)$, moreover since n is bounded above by m then $T_p = O(2m) = O(m)$.

The **cost** is calculated as $p*T_p = O(n*m+n^2)$ and $\frac{O(n*m)}{O(n^2)} = O(\frac{m}{n})$. Note that when n = m (square matrix) then $O(\frac{m}{n}) = O(1)$ that is, the algorithm becomes cost-optimal.

The **speedup S** is calculated as the ratio of the serial complexity to the parallel one $S = \frac{T_s}{T_p} = \frac{O(n*m)}{O(m)} = O(n)$. The **efficiency** is instead calculated as $E = S/p = O(n)/p$, setting $p = O(n)$ we have that $E = O(1)$.

The antidiagonals algorithm described has two main problems:

- **Limited parallelism**, this is due to the fact that the diagonals have different dimensions. Can cause a decrease in speedup as the algorithm moves away from the main antidiagonal. In this way the computing resources may not be fully exploited.
- **Hardware coalescing**, since the antidiagonals are saved in memory in distant positions, this does not favor the data transfer.

This type of algorithm is well known in the literature, and is called **irregular applications** [3], that is applications that do not properly exploit memory-access patterns, that have data-dependent control flow or that make fine-grained data transfers.

Furthermore, the score matrix that is saved has a size of n * m, which can be expensive to save in memory when the sequences become very large. For the purpose of this project, it will be shown a **memory optimized** version, that not saves all the score table but only a small constant portion of it. In particular, it will memorize only three antidiagonals.

2.2 [v2] Efficient parallel LCS

In this section we will present the efficient **version (v2)**[4] of the parallel algorithm LCS, which can be considered an **optimization** of the version (v1). It will be shown how it is possible to solve the two problems that plagued (v1), in particular by balancing the workload and increasing the parallelism, and secondly by improving memory coalescing.

Optimization is based on **changing the dependency** in the dynamic programming table so that the cells in the same row or the same column of the dynamic table can be computed in parallel, this is shown visually in figure 4. The algorithm uses O(max(m, n)) processors and its time complex is O(n). A nice property of the algorithm is that each processor has balanced workload.

We analyze data dependency in the score matrix constructed by dynamic programm. We present a new parallel algorithm based on rearranging the entries in the score matrix for greater parallelism. We analyze the time complex of the parallel algorithm and **compare it with the wavefront (v1)** parallel algorithm to show it advantages.

In order to compute the same row data or column data in parallel, the data dependence needs to be changed. Analyzing the original LCS recursion in equation (1) we have that:

- **C[i, j] = 0 if i = 0 ∨ j = 0**, in this case the data dependency is not changed.
- **C[i, j] = C[i − 1, j − 1] + 1 if** $x_i = x_j$, in this case the data dependency is not changed.
- **C[i, j] = max(C[i − 1, j], C[i, j − 1]) otherwise**, in this case the data dependence is to be changed.

The third condition **can be made indipendent** from the i-th row, as C[i, j−1] can be replaced by the following recurrence equation:

$$C[i, j-1] = \begin{cases} 0 & \text{if } i = 0 \vee j - 1 = 0 \\ 1 + C[i-1, j-2] & \text{if } x_i = y_{j-1} \\ max(C[i-1, j-1], C[i, j-2]) & else \end{cases} \quad (3)$$

Since only the third condition depends on the i-th row data, we can again reformulate S[i, j − 2] in a similar way, and so on. This process ends when j−k = 0 at the k-th step, or $x_i = b_{j-k}$ at the k-th step. Assume that the process stops at

the k-th step, and the k must be the minimum number that makes $x_i = b_{j-k}$ or j − k = 0.

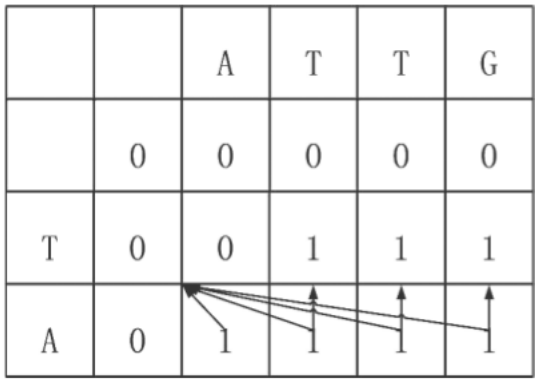


Figure 4: Reformulated data dependency

Then the recurrence equation equation (1) can be replaced by the following recurrence equation:

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ 1 + C[i-1, j-1] & \text{if } x_i = y_j \\ max(C[i-1,j], C[i-1, j-k-1] + 1) & \text{if } x_i = y_{j-k} \\ max(C[i-1,j], 0) & \text{if } j - k = 0 \end{cases} \tag{4}$$

We can further re-formulate equation (4) by replacing j − k with a preprocessed matrix P[c, j] of size l ∗ (m + 1) where l is the alphabet size. In case of biosequences gene this is $F = \{A, C, G, T\}$ so l = 4. P[i, j] represent the maximum number before j that makes $y_{P[i,j]}$ = F[i]. For further detail on this preprocessing refers to [4].

The parallel algorithm calculate the ith row data in parallel without branching as follows:

Algorithm 3 Parallel version (v2) of the LCS that computes rows in parallel.

```
procedure LCS_v2(X, Y)
    for i ← 0 to l parallel do
        for j ← 0 to m do
            Preprocess the entry P[i, j]
    for i ← 1 to n do
        for j ← 1 to m parallel do
            t = the sign bit of (0 − P [c, j])
            s = the sign bit of (0 − (S[i − 1, j] − t ∗ S[i − 1, P[c, j] − 1]))
            C[i, j] = C[i − 1, j] + t ∗ (s ⊕ 1)
    return C[n, m]
```

The **complexity** of this algorithm is calculated as follows. First the preprocess of the P matrix takes $O(m)$ time. The score table takes $O(n)$ time, since the rows have to be computed one after the other. In total the time complexity is $O(max(m, n))$.
Assuming that n = m the comlpexity became:

$$T_p = O(n * \frac{m}{p})) \tag{5}$$

Setting $p = O(m)$ then $T_p = O(n)$. The **cost** of the computation is the quantity $pT_p = p * O(n * \frac{m}{p})) = O(n * m)$. Since $\frac{O(n*m)}{O(n*m)} = O(1)$ then the algorithm is cost-optimal for square matrices. The efficiency $E = O(n)/p = O(n)/O(n) = O(1)$.

Comparing this efficient algorithm to the (v1), the data on same antidiagonal can be computed in parallel but the number of cells to compute differs from one antidiagonal to the other, and goes from 1 to max(n, m). Assuming that we have max(n, m) processors, some processors are in idle when the algorithm doesn't compute the main antidiagonal, and this cause a **reduced parallelism**.

Furthermore, the (v2) algorithm also solves the problem of branch conditions. In fact, branch divergence reduces program efficiency, especially in the case of the GPGPU/CUDA model.

3 Results and Evaluation

In this section the experimental tests of the algorithms presented above will be presented. In particular, both the v1 and v2 algorithms will be implemented with 3 parallel programming models: shared memory (with openMP), distributed memory (with openMPI) and GPGPU (with CUDA/C++), and comparisons with the serial algorithm will be made.

3.1 Hardware specification

In this section we will present the **hardware components** of the computer on which the algorithms implemented with openMPI, openMP and CUDA will be executed. The experimental results will therefore depend on the technical characteristics of this setup and may therefore be different if performed on different hardware.

Table 1: List of hardware components.

Component	**Type/version**
PC	Lenovo IdeaPad Z500
Motherboard	LENOVO 31900003WIN8 STD MLT
CPU	Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz 4 cores 8 threads
System Memory	8GiB
Host bridge	3rdGen DRAM Controller 32 bits 33MHz
GPU	GeForce GT 635M, Fermi, 2GiB

Since the aim of the project is the implementation of the algorithms presented on the message passing interface, shared memory and gpgpu models, the following software applications have been prepared. For the purposes of the experimentation was installed **ubuntu 16.04 LTS** as operating system, the following software was also installed:

- **Compiler**: gcc 5.4.0 (that includes **openMP** directives support)
- **MPI**: Open MPI 4.1.3
- **CUDA/nvcc**: NVIDIA (R) Cuda compiler driver release 8.0

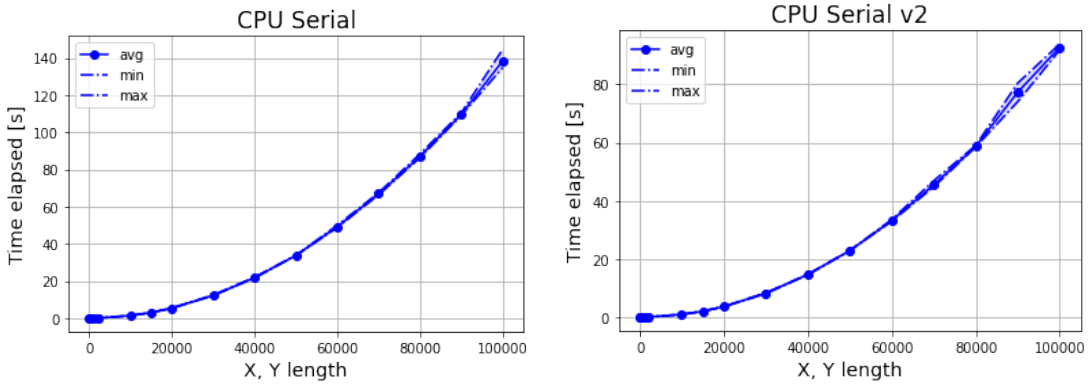
3.2 Serial

The **serial algorithm** was implemented in its original version presented in algorithm 1 and in the version v2. The time complexity of this algorithm is $\Theta(n * m)$ since it must access a table of size n * m. The memory space used is 2 * min(n * m), in fact only the last 2 rows of the table are stored.

From the **implementation** point of view, the algorithm was implemented with C++11 and executed on the machine presented in the previous chapter. The algorithm computes the score table one cell at a time row-wise, and runs on a single processor of the CPU serially.

The experiment measured the **execution times** of the serial LCS between two X and Y strings with different lengths. The strings were randomly generated in the $\Sigma = \{A, C, G, T\}$ alphabet. In order to obtain statistically significant results, 3 executions have been carried out for each pair of strings, and the minimum, maximum and average are reported.

The **graph** in figure 5a shows the evolution of the execution time as a function of the lengths of the strings.



(a) Execution time of the original LCS

(b) Execution time of the v2 LCS

Figure 5: Serial LCS executions

As can be seen from the graph, the trend seems to correspond to a polynomial n^2 where n is the length of X and Y. The computation of a string of 100,000 characters takes about 140 seconds for the original version and about 90 seconds for the v2 version.

3.3 OpenMP

OpenMP (Open multiprocessing) is an API for creating parallel applications on **shared memory** systems. It is used in this project in the C++11 programming language. OpenMP is composed of a set of compilation directives that define its operation at run-time. It uses a portable model that provides the programmer with a simple interface for developing parallel applications.
OpenMP uses multithreading to split a process into several threads that can run in parallel on the various available processors.

In order to test the performance of the LCS v1 and v2 algorithms using the shared memory model, openMP is therefore used. For each pair of randomly generated X, Y sequences, verisons v1 and v2 are performed to verify their execution times. Since the CPU supports 4 processes and 8 threads in parallel, the **benchmark** is run varying **threads from 1 to 8**.

Also in this case the graphs in figure 7 are shown with the maximum, minimum and average of the lead times.
Looking at the graphs it is noted that the best performances are obtained with 4 threads and the efficient algorithm v2 (the yellow graph), while a higher number does not lead to an increase in performance.

The experimental **speedup** is then calculated by relating the v2 version with 4 threads openMP to the efficient serial version v2. In addition, the graphs with the minimum times for both were used. As can be seen in the figure 6 the speedup can reach more than 3x in relation to the serial efficient v2 version.

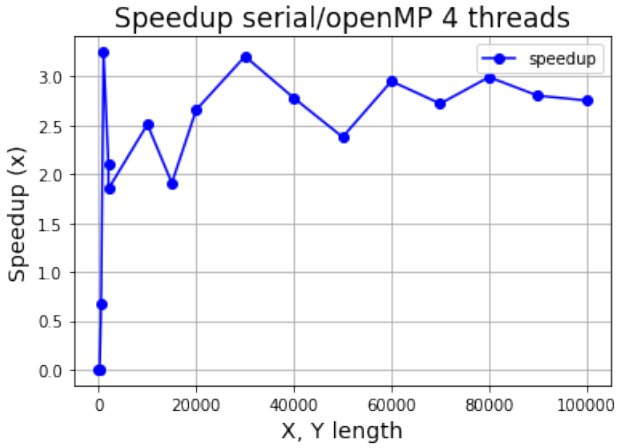
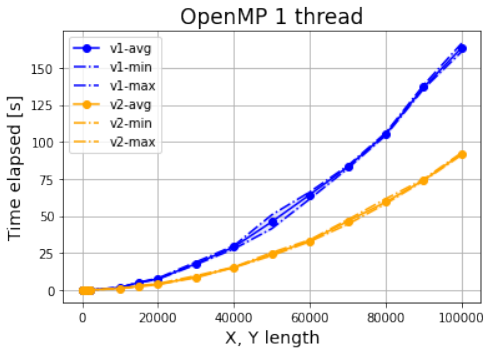
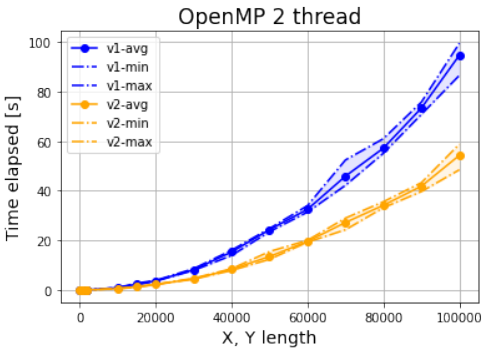


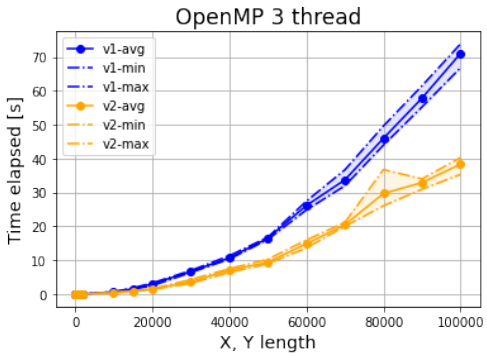
Figure 6: Speedup of the serial v2 over openMP with 4 threads



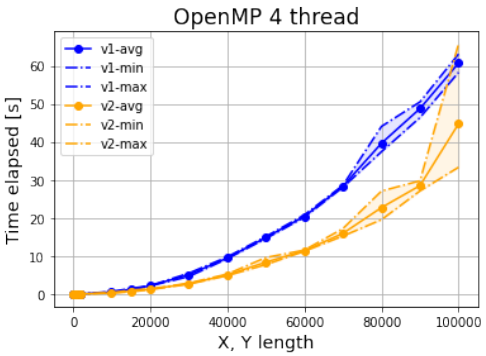
(a) 1 thread



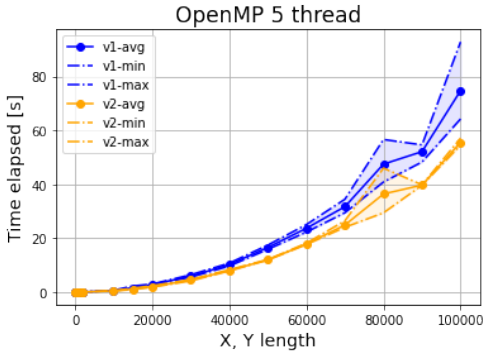
(b) 2 threads



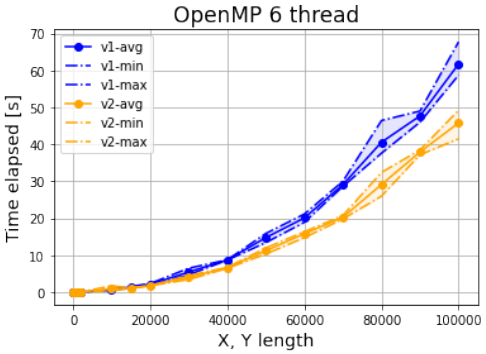
(c) 3 threads



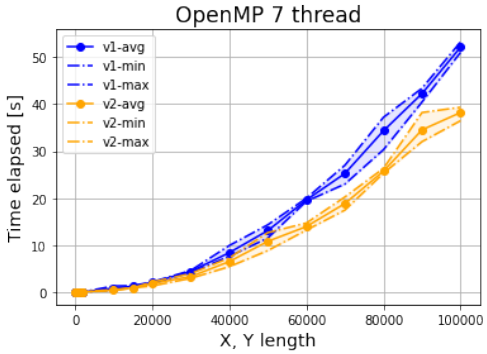
(d) 4 threads



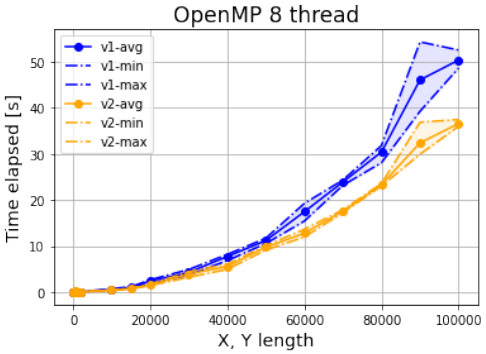
(e) 5 threads



(f) 6 threads



(g) 7 threads

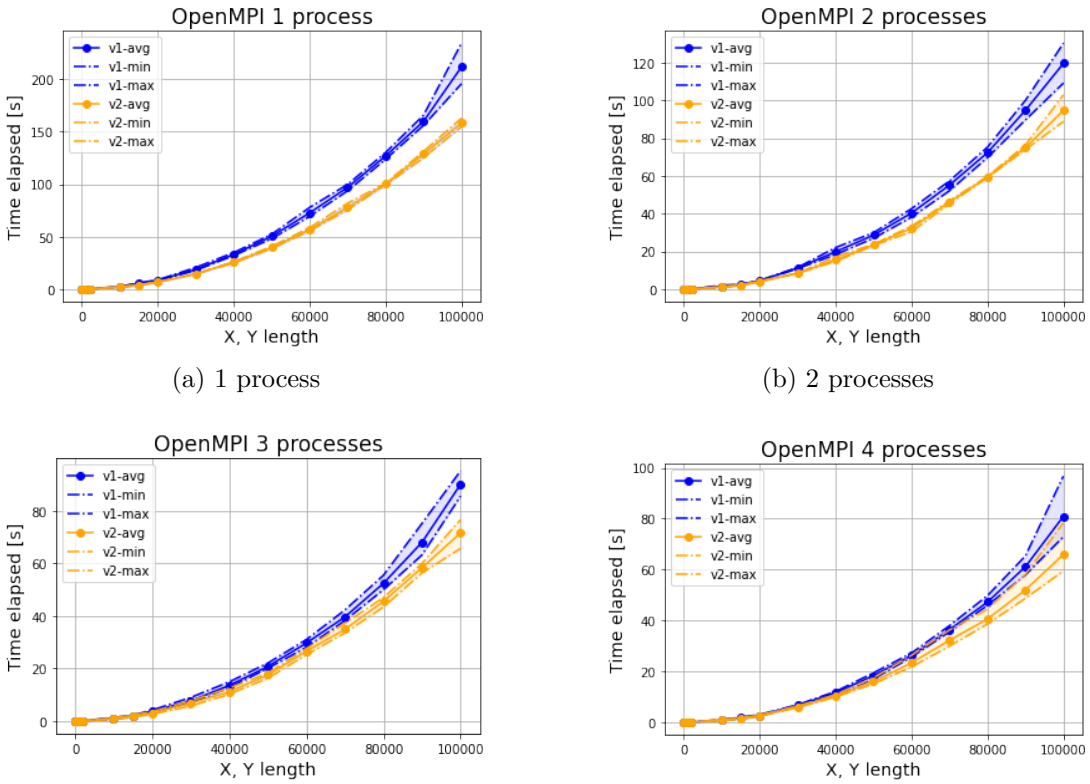


(h) 8 threads

Figure 7: OpenMP execution time of v1 and v2 parallel LCS, on 1..8 threads

3.4 OpenMPI

The experimentation of the two parallel versions of the LCS presented were also performed on the MPI model. In particular, the openMPI implementation of the **Message Passing Interface (MPI)** is used. MPI is the de facto standard for communication between nodes belonging to a cluster of computers running a parallel multi-node program.



(a) 1 process

(b) 2 processes

(c) 3 processes

(d) 4 processes

Figure 8: OpenMPI execution time of v1 and v2 parallel LCS, on 1..4 processes

Unlike the shared memory model, MPI is placed in the **distributed memory** model, in this case each process has its own address space, which is not visible from the other processes, and the only way to communicate is the exchange of messages.

In this experiment, openMPI is used on a single computer, but it could also be used in a computer network. A common use is to use OpenMPI in combination

with OpenMP in a hybrid way.

The **graphs** in figure 8 shows the execution of the two versions v1 and v2 of the LCS parallel to the variation of the size of the strings. The benchmark was done by varying the number of processes. As you can see from the graph, the fastest version is the v2 (the yellow graph) with 4 processes, in fact it takes full advantage of all the processors provided by the CPU.

Figure 9 shows the experimental **speedup** obtained by comparing serial execution v2 and openMPI v2 on 4 processes. The speedup obtained in this case reaches 1.5x.

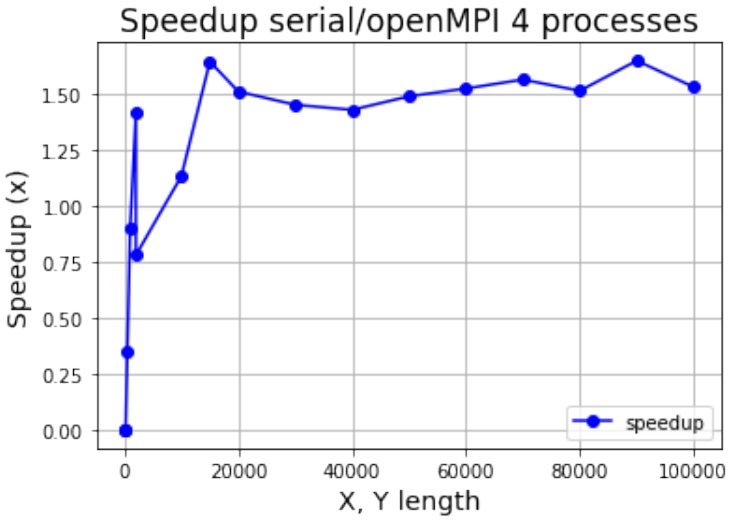


Figure 9: Speedup of the serial v2 over openMPI with 4 processes

3.5 CUDA

As the last model of this report the GPGPU model is used, and in particular the proprietary architecture of NVIDIA CUDA (Compute Unified Device Architecture). Also in this case the programming language C++ is used with the api supplied with the CUDA toolkit and with the nvcc compiler. Through this model the programmer can exploit the GPU in order to perform calculations in parallel.

As unlike CPUs, GPUs have a parallel architecture with many cores, it can deliver much higher throughput than the CPU.

The typical CUDA program expects to send data from the primary memory to the global memory of the GPU. The cpu prepares the launch configuration to start the execution of the GPU kernel, then the result of the computation is sent to the CPU. The architecture is also called single instruction stream, multiple data stream (SIMD), as a large number of identical processors execute the same sequence of instructions on different sets of data .

All threads within a block run in multiprocessor (SM) streams and can use shared memory, which is only visible within the block. Furthermore CUDA has a hierarchy of several types of memory: global, shared, local, texture/constant memory.

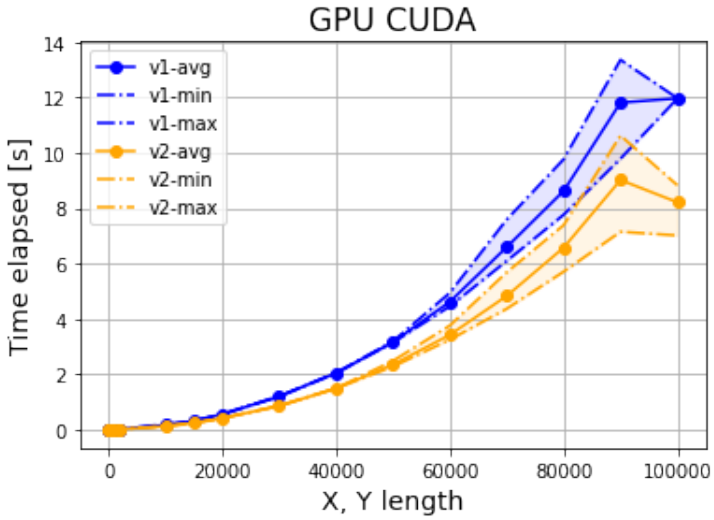


Figure 10: CUDA execution time of v1 and v2 parallel LCS

The image in figure 10 shows the maximum, minimum and average **execution times** of the two versions v1 and v2 of the parallel LCS. As you can see,

even in this case, the v2 version is more efficient, and resolves the LCS between two strings of 100,000 characters in about 7 seconds.

Comparing the execution times of the more efficient v2 version with the serial v2 version we obtain the speedup, as shown in figure 11. In this case the speedup manages to exceed 12x with very long strings.

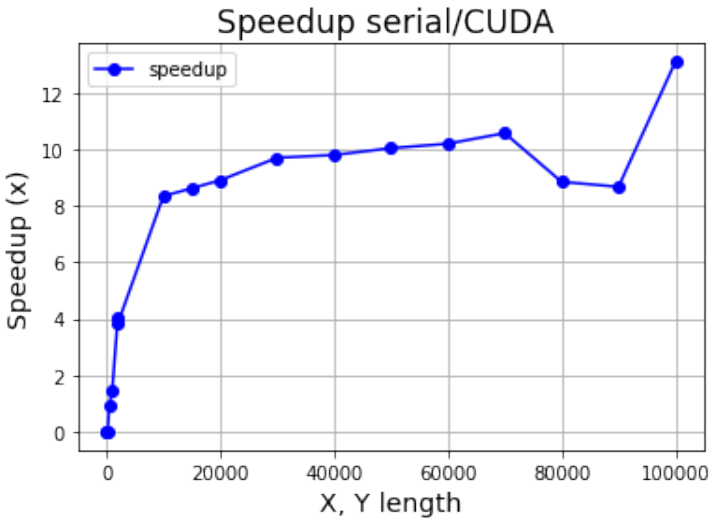
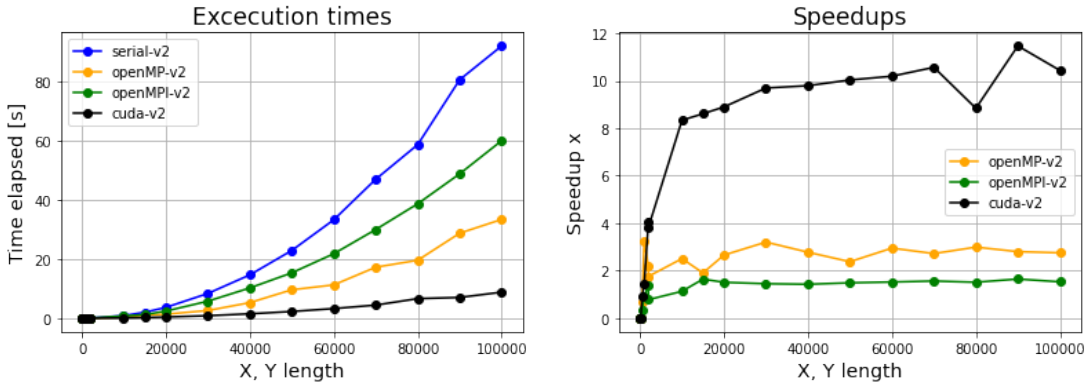


Figure 11: Speedup of the serial v2 over CUDA v2

4 Discussion

This section compares the best results obtained with the different parallelization models. In particular, these correspond to the v2 version of the parallel LCS algorithm. As you can see from the graph 12 a the best execution times of the algorithm were those of the CUDA implementation on GPU. This result can be justified by the fact that GPUs have been designed to have thousands of processors that execute the same instructions on different subsets of the data, this allows for massive parallelism. On the contrary, the CPU with 4 processors allows to have a lower degree of parallelization but with the advantage that these generally have higher clock speeds. Furthermore, sending data from the host to the GPU involves an initial overhead that can win the comparison to openMP due to the small size of the problem, in particular for strings smaller than 1000 characters.

In terms of speedup instead, the graph in figure 12b shows the comparison of the parallel algorithms in v2 with respect to the serial v2 implementation. As it is possible to see also in this case CUDA allows to reach a speedup of 12x, openMP of about 3x while openMPI of about 1.5x.



(a) Execution times

(b) Speedups

Figure 12: Comparison of openMP, openMPI and CUDA

The approach of implementing the same algorithm with the different parallelization models has allowed us to know and deepen also the **complexity that the programmer** may encounter. What we noticed is that the porting of the code from serial to openMP implementation was the simplest, in fact a few directives were enough to parallelize the main for loops. As for openMPI and CUDA, these were the most complex to implement. In particular in openMPI the programmer has to manage the passage of data between the various processes, while in CUDA the complexity derives from the efficient management of the memories and from the study of the API provided by NVIDIA.

Among the future developments one could think of optimizing the algorithms by exploiting technological advancement to have even better performance, for example the most recent GPUs have a feature called dynamic parallelism that allows the GPU to avoid the overhead of returning control to the host. Furthermore, it would also be interesting to study how to increase parallelism through openMPI in a computer network or with a hybrid approach openMP + openMPI, or multiGPU approaches in a network.

Furthermore an interesting future work can be the increase of the number of input sequences, for example using 3 sequence as input the score table became

a score cube[5], and instead of the antidiagonal computed in parallel will be a plane computed in parallel.

5 Conclusions

The implementation of the parallel LCS algorithm has allowed us to recognize how a GPGPU approach using CUDA can outperform serial, openMP and openMPI approaches in terms of temporal performance and speedup.

The work also showed how, in dynamic programming problems there can be parallelizable portions, as shown by the wavefront version (v1). It was also shown how to increase parallelism by changing the dependency of the data and moving from an "irregular" algorithm to one that makes the most of resources and eliminates branching (v2).

In conclusion, the work done can be considered successful, as it was possible to increase the performance of the serial LCS version up to 12x with CUDA. Furthermore, the dynamic programming approach that is exploited by the LCS can be easily generalized for other contexts. In particular, a variant of this problem is the Needleman–Wunsch algorithm for global sequence alignment, or the Smith–Waterman algorithm[6] for local alignment.

References

[1] Wikipedia, "Longest common subsequence problem," [Online; checked il 31/8/2022]. [Online]. Available: https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

[2] A. Dhraief, R. Aissaoui, and A. Belghith, "Parallel computing the longest common subsequence (lcs) on gpus: Efficiency and language suitability," pp. 143–148, 10 2011.

[3] A. Tumeo and J. Feo, "Irregular applications: From architectures to algorithms [guest editorsx27; introduction]," *Computer*, vol. 48, no. 08, pp. 14–16, 2015.

[4] Y. Jiaoyun, X. Yun, and Y. Shang, "An efficient parallel algorithm for longest common subsequence problem on gpus," *Lecture Notes in Engineering and Computer Science*, vol. 1, 06 2010.

[5] N. Ukiyama and H. Imai, "Parallel multiple alignments and their implementation on cm5," 08 1994.

[6] V. Manavski, S.A., "G. cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinformatics*, vol. 9 (Suppl 2), 2008.