Longest Common Subsequence

by Jesse Huang, Osman Saleem, Seong Hyeon (Kevin) Park

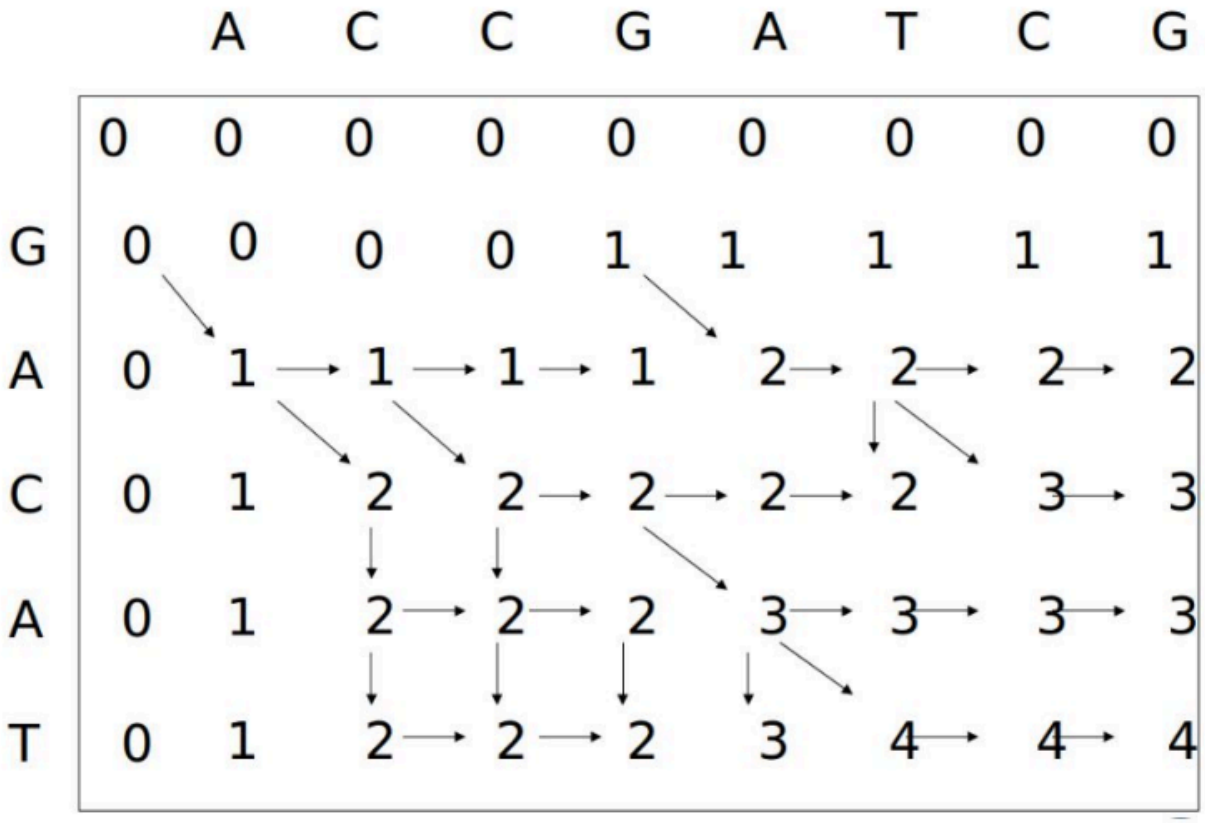
Introduction

The longest common subsequence (LCS) is a problem where given two sequences $X = <x_1, x_2, \ldots, x_n>$ and $Y = <y_1, y_2, \ldots, y_n>$, we are to find the longest subsequence L that is shared between both X and Y. A subsequence L_1 is a subset of another sequence L_2 where 0 or more deletions of elements have occurred but the remaining elements continue to maintain the order from L_2. For instance, the longest common subsequences found between subsequence X = <a,b,c,d,e,f> and subsequence Y = <d,b,a,c,e> are length 3 subsequences L_3 = <b,c,e> and L_4 = <a,c,e>. The longest common subsequence has a wide variety of applications including but not limited to bioinformatics, data compression and text comparison, such as DNA sequence analysis and the diff tool[1].

Background

The current state-of-the-art serial implementations of finding the Longest Common Subsequence revolve around the dynamic programming approach. The dynamic programming approach is where a recursive problem is broken down into subproblems and the results of the subproblems are calculated once and cached for future use. This is done using n * m matrix where each element represents the length of the longest subsequence that could be made at the certain index. The length of the longest common subsequence can be found at the bottom right of the matrix, and the subsequence itself can be found by backtracking through the table. The time and space complexities are both O(n * m). There are implementations of this where space complexity can be reduced such as https://www.geeksforgeeks.org/space-optimized-solution-lcs/.

An example of a filled-in DP matrix from [2]:



The current state-of-the-art parallel implementations of finding the longest common subsequence revolve around an anti-diagonal approach. Similar to the serial version, it computes an n * m matrix, however, because of a data dependency for each cell in the matrix between c[i,j] and its adjacent neighbors c[i-1,j], c[i,j-1] and c[i-1,j-1], the filling of the matrix must be done diagonally. This allows elements in the diagonal to be done independently so long as the previous diagonal has been computed [2].

Implementation Details

The serial implementation was created with the help of ChatGPT (for logic to handle multiple LCSs) and chapter 12 of the IPC textbook. This serial implementation is a classic dynamic programming (bottom-up) solution is where a 2D table DP (size of n (length of X) * m (length of Y)) is iteratively populated with integers. DP[i][j] stores the length of the longest common subsequence of the first i characters of X and the first j characters of Y.

Initially, DP[0][j] = 0 for all j and DP[i][0] = 0 for all i. This is because the LCS between a string and an empty string is 0.

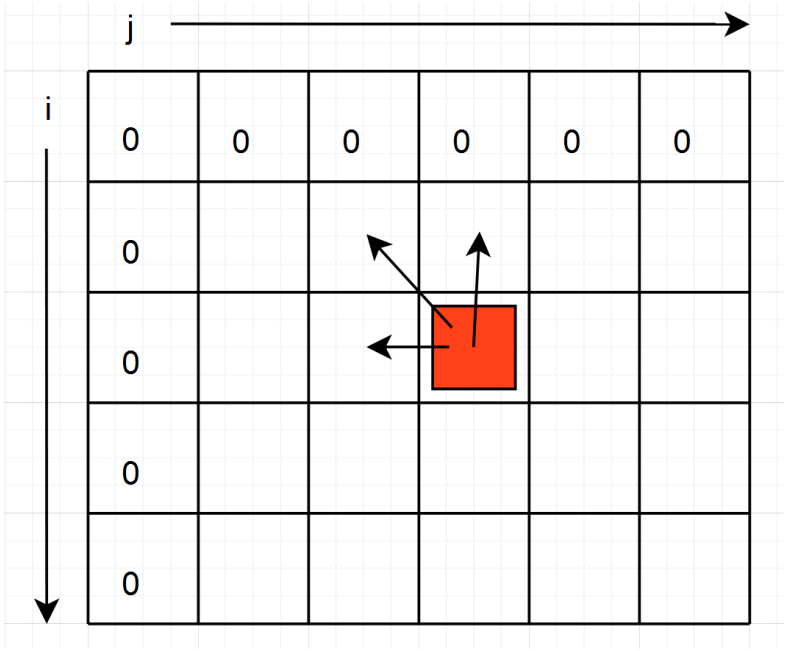
Each cell [i][j] is populated by comparing X[i - 1] and Y[j - 1], if they are equal, the cell is set to DP[i - 1][j - 1] + 1, otherwise the cell is set to max(DP[i - 1][j], DP[i][j - 1]).

If they are equal, it means that X[i - 1] and Y[j - 1] are part of the LCS so the LCS of the previous subsequence (DP[i - 1][j - 1]) + 1 (for the current match) is the value for DP[i][j].

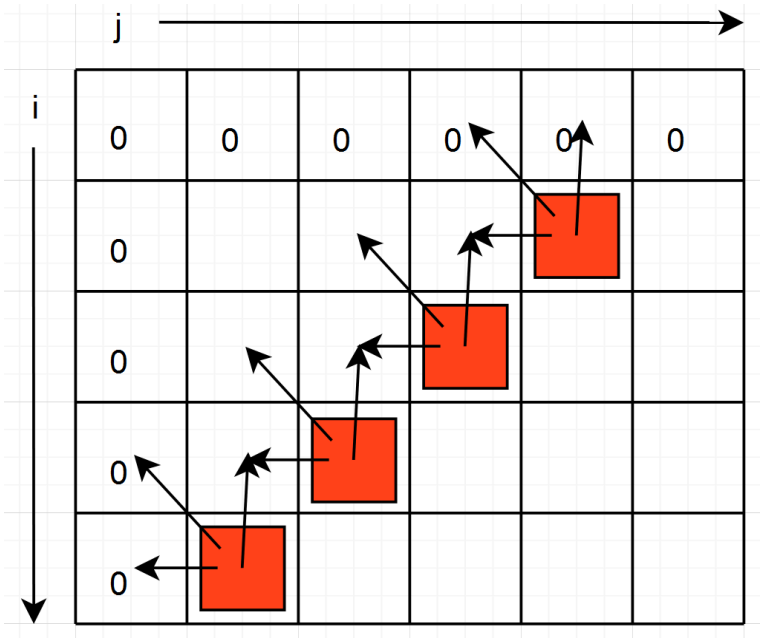
If they are not equal, the LCS is the maximum length without the current character of X or the current character of Y.

The parallel implementation requires the problem to be broken up differently while still following the dynamic programming approach. In the serial version, we advance through the DP table iteratively row by row, which is not possible in the parallel version because of the fact that the calculation at each cell depends on values of some of the neighboring cells, meaning one thread may be trying to read a value of a cell in the same row that another thread may not have calculated yet.

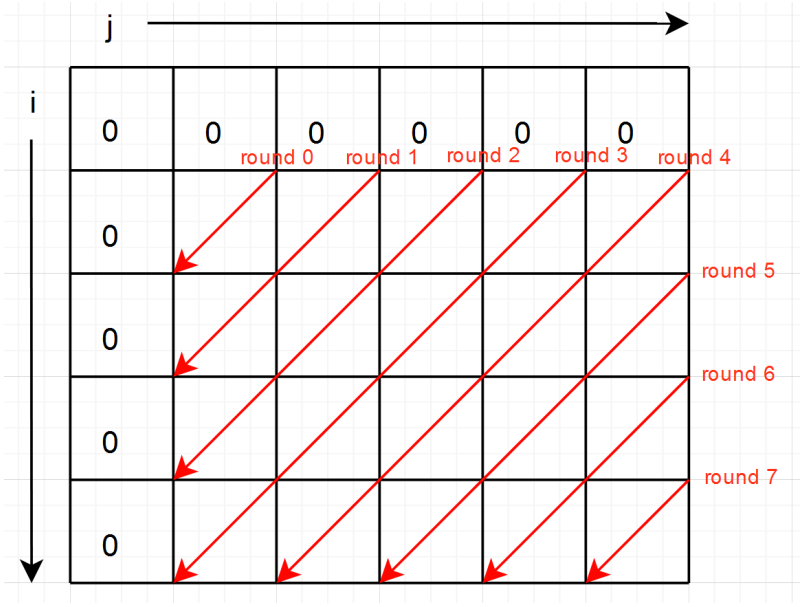
This problem can be solved by instead advancing through the DP table in diagonal lines. Consider the DP table shown in the figure below



The cell at the location of the red square depends on the values of neighboring cells pointed to by arrows. If each cell along a diagonal line was computed by a different thread, and we know that all cells' values from the previous diagonal line have already been computed, then it is possible to parallelize this algorithm.



If the algorithm progresses at one diagonal line of cells at a time, all arrows point to cells with values that are known to have been computed.



The distributed version is done using Message Passing Interface (MPI). The DP matrix is decomposed in a fine-grained manner and processed in nearly the same way as the parallel implementation, however, extra communication is needed after performing the calculations to ensure that calculations for the next diagonal possesses the necessary data in order to satisfy the data dependency of each cell and allow the next diagonal cells to be processed independently. Only the first and last cells are sent in order to reduce the size of communication between processes. After calculating the last element of the DP matrix, the root process reconstructs the DP table by iterating through the table and receiving incoming rows of data from other processes

to fill out the rest of the matrix. Backtracking is then done on the root process after reconstructing the full DP matrix in order to find the longest common subsequences.

Evaluations

To evaluate our implementations, we've run our test scripts 3 times each and took the average. There are 2 graphs each for serial, parallel and distributed versions. The first graphs only show the largest time of all threads/processes taken to fill the DP table. The second graphs include the time it takes for backtracking. Input size refers to the lengths of the strings.

We used 3 pairs of strings as input:

(Case 10x10)

X = selvyjg0he, Y = lx3oeh8yso

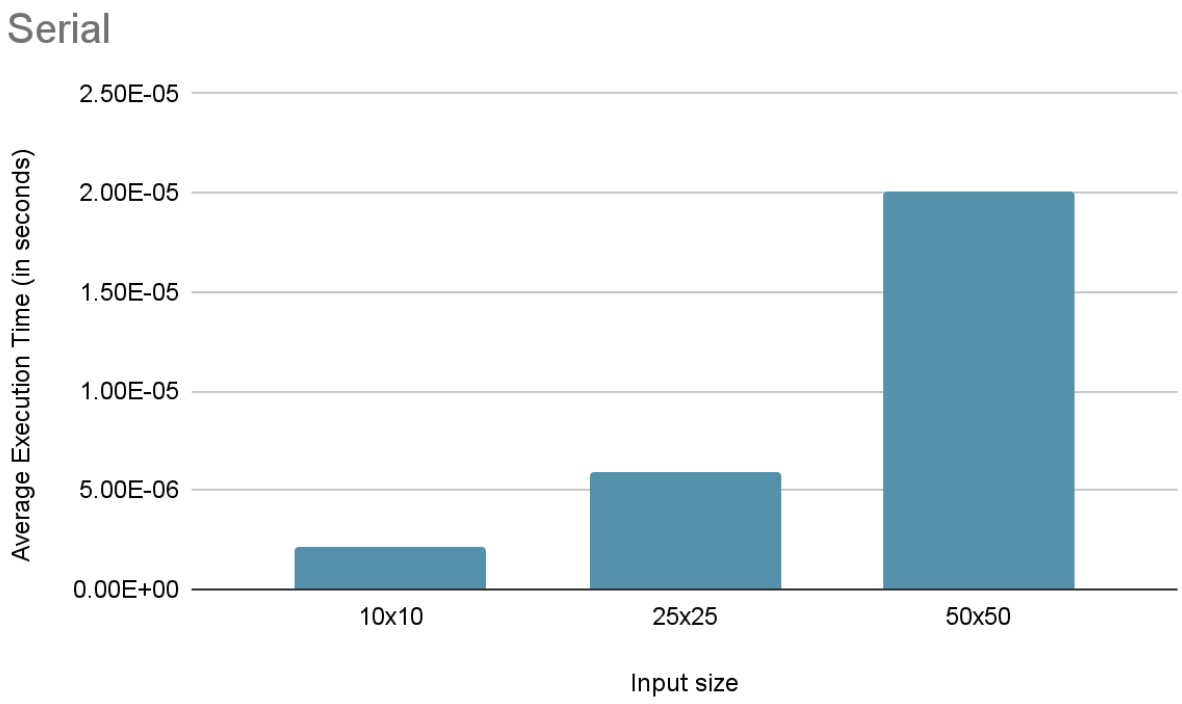
(Case 25x25)

X = qqz96admjoiekgvywb7n55qr0, Y- u8dn8wkxiovwa382m77xp137m

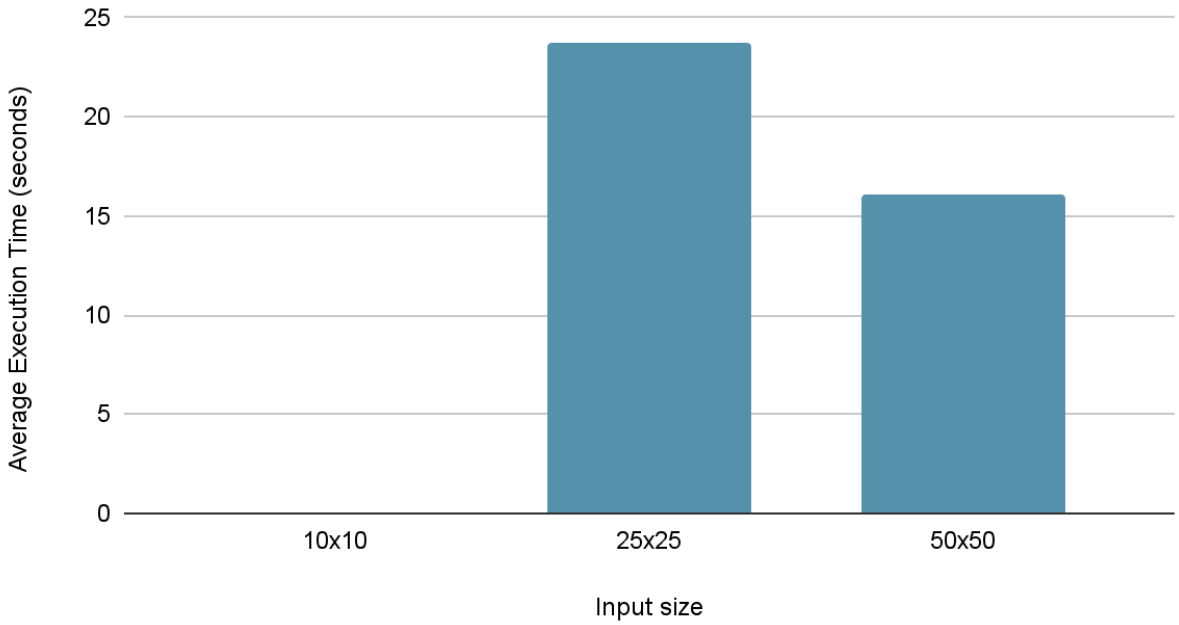
(Case 50x50)

X = 82t43c8ndm1lbsebl4jfuutw224s9ubzi78k65tidd7b2p9l10

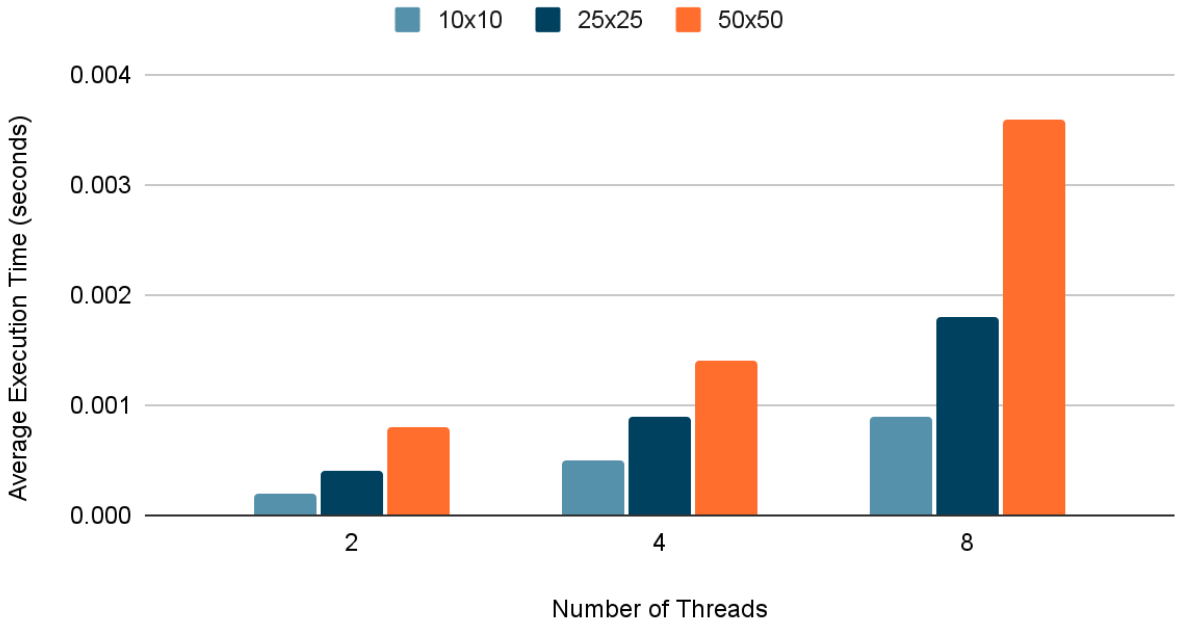
Y = ikdugd5uq3rox5lwz63adyvihu2cuo4dkvi1m7p9f7w1nk4za0



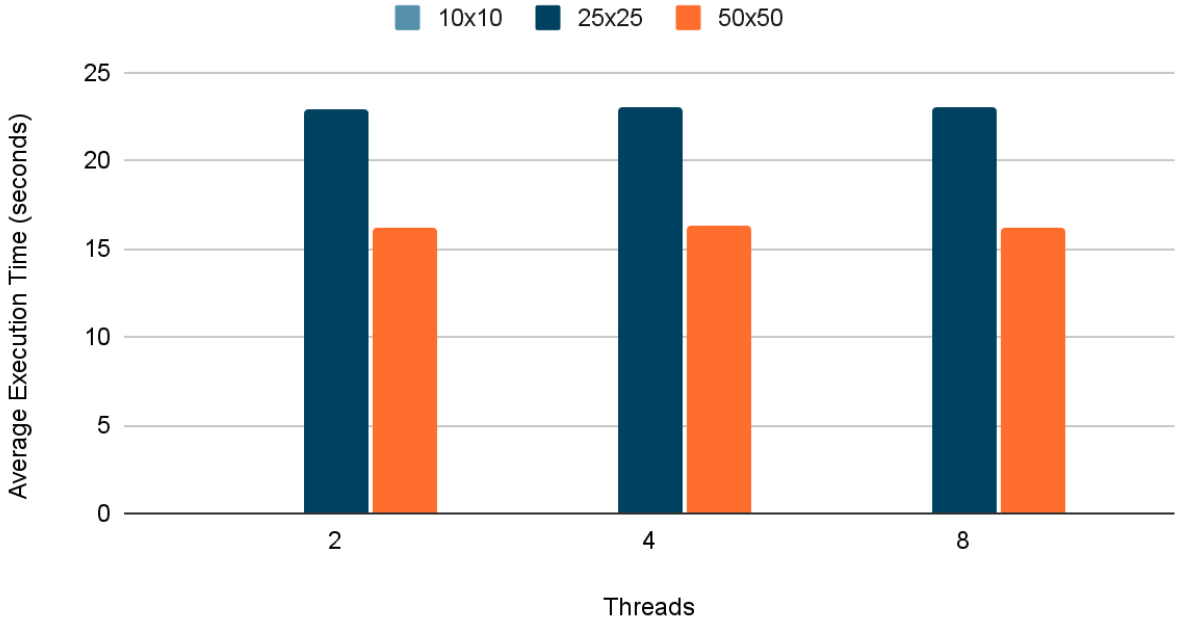
Serial + Backtrack



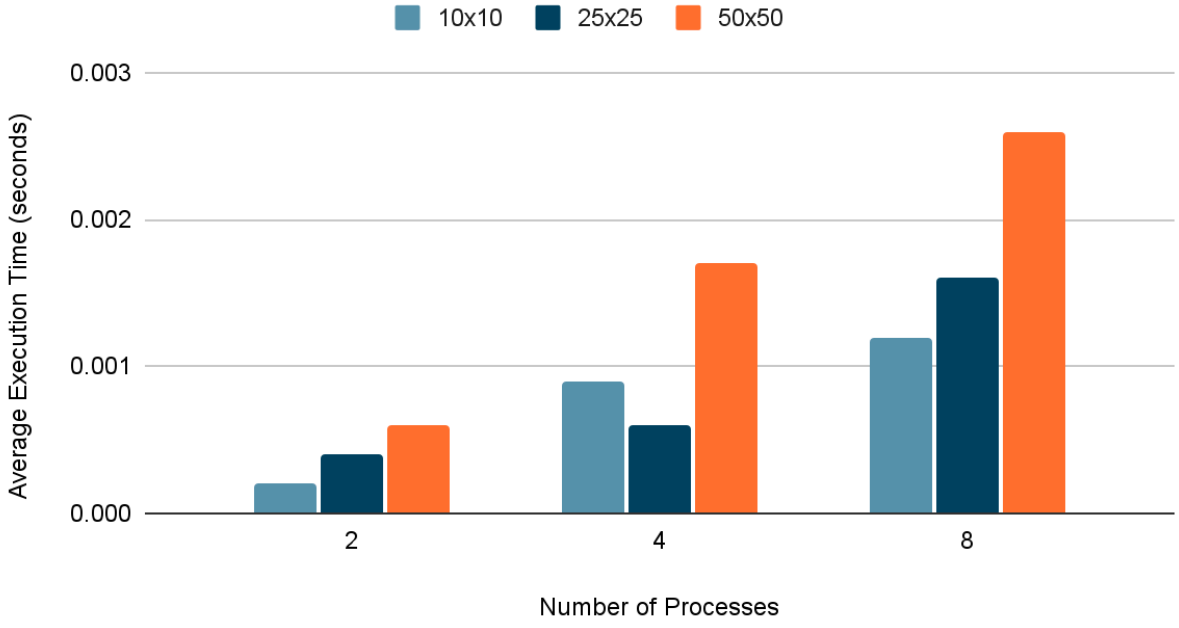
Threads

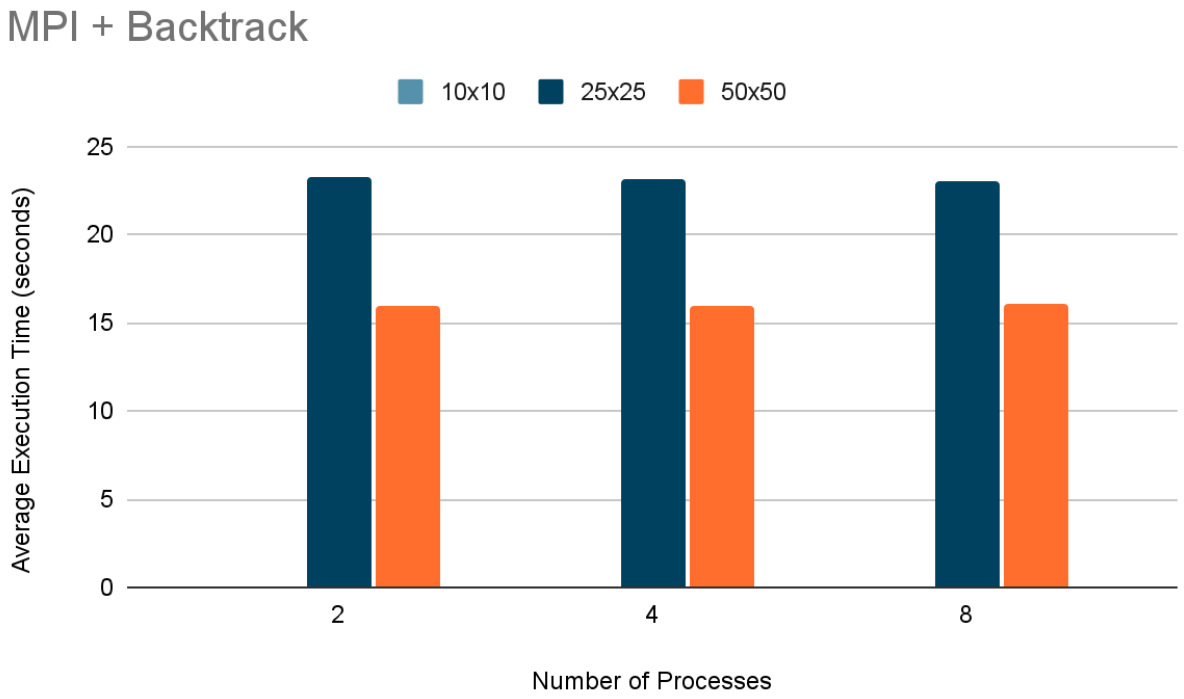


Threads + Backtrack



MPI





Conclusions

From the results shown above our parallel and distributed solutions are unfortunately slower than the serial versions. Serial is faster than 2 threads/processes by roughly 10, 70, and 40 times for 10x10, 25x25, and 50x50 respectively. Serial is faster than 4 threads/processes by 230, 150, and 70 times for 10x10, 25x25, and 50x50. Serial is faster than 8 threads/processes by 420, 300, 180 times for 10x10, 25x25 and 50x50 respectively.

These results show that implementing parallel solutions is challenging because of the data dependencies found in the LCS problem. The majority of the execution time is dedicated to communication, especially in regards to barriers which can take up to 97% of execution time for a single worker. Because this is done diagonally, there are some instances where there are not enough elements to distribute across all the available workers as well. Further optimization could be done, such as having a producer-consumer structure where producers assign elements of the DP matrix to consumers to perform calculations.

Additionally, we've learned that backtracking causes time to vary greatly depending on what the sequences are. We've used random sequences of lowercase letters and numbers. This is one of the reasons why we chose to have 2 graphs per implementation, and have the first graph represent the timing for just filling in the DP matrix. Some sequences are much easier to navigate through, as seen with how the 25x25 input sequences cause the program overall to take much longer than the 50x50 input sequences despite taking less time filling in the DP matrix. In this regard we could have also optimized the backtrack function, which is currently a sequential process unlike the calculation of the DP matrix.

References

[1] P. Karsh, “The Longest Common Subsequence Problem: An Introduction and Solution in Ruby,” October, 2023. [Online]. Available: https://patrickkarsh.medium.com/the-longest-common-subsequence-problem-an-introduction-and-solution-in-ruby-d7076e277e5b

[2] L. Nadanasabapathi. CSE633 Parallel Algorithm: Longest Common Subsequence in Parallel. [PowerPoint slides]. Available: https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Lavanya-Nadanasabapathi-Spring-2022.pdf