Parallel Computing the Shortest Common Supersequence Problem

Zirui Zhao
Department of Computer Science Engineering
University of Michigan
Ann Arbor, US
zhaojer@umich.edu

Abstract

My term project attempts to come up with an efficient parallel algorithm for the shortest common supersequence problem, which has been understudied in the parallel computing field. This problem is not embarrassingly parallel because the current serial solution utilizes a Dynamic Programming algorithm with both row-wise and column-wise data dependencies, making parallelization impossible using a traditional iterative approach. After a reinterpretation and a substantial analysis of the problem and the serial algorithm, I was able to design and implement 2 parallel algorithms in OMP and CUDA. My parallel programs had fast runtimes and reasonable effiency/speedup, with the row-wise independent algorithm on the GPU being the best solution overall.

I. Introduction

In computer science, the shortest common supersequence (SCS) problem revolves around finding the smallest (shortest) possible sequence that encompasses two given sequences while preserving their original order. Formally, let $X = x_1...x_n, Y = y_1...y_m$ be two sequences of characters (i.e. strings) over a finite alphabet Σ; the two sequences have integer length n and m respectively. We define a sequence U as a *supersequence* of X if X can be obtained from U by deleting some (0 or more) characters from U, without changing the order of the remaining characters. Then, U is a *common supersequence* of X and Y, if U is a supersequence of X and a supersequence of Y; U is the *shortest common supersequence* of X and Y, $SCS(X, Y)$, if it is a common supersequence of X and Y with minimal length [1]. For instance, if $X = abac$ and $Y = cab$, then $cabac$ is the shortest common supersequence of X and Y. In general, the SCS is not unique.

The SCS problem can also be generalized to finding the shortest common supersequence of an arbitrary number of input sequences, commonly known as the *multiple shortest common supersequence* problem or the *generalized shortest common supersequence* problem. The core aspects and principles of the generalized SCS problem have applications in diverse domains such as data compression [2]; query optimization in database [3]; and DNA sequencing [4].

The generalized SCS problem has been proven to be NP-Hard [5]. On the other hand, when only considering a fixed number of input sequences, most commonly just two, the SCS is usually solved using a serial dynamic programming approach in $O(n^k)$ time, where n is the (maximum) length of the input sequences and k is the number of input sequences [6]. Although this problem has been around for a while, very few studies have attempted to devise an efficient parallel algorithm for it.

Thus, my term project aimed to design, implement, and analyze two novel parallel algorithms for computing the SCS of two input sequences, based on the existing serial dynamic programming algorithm for solving this problem. Parallelizing the serial dynamic programming algorithm for the SCS problem presented a worthy and adequate challenge (in the context of EECS 587) because the inherent data dependencies in a dynamic programming algorithm renders its parallelization difficult, requiring substantial changes to the serial code and a reinterpretation of the problem itself, hence indicating that the resulting parallel algorithm was not embarrassingly parallel.

In the present paper, I first discuss the existing serial dynamic programming algorithm for solving the SCS problem and the two parallel algorithms I devised based on the serial version. Then, I talk about how I implemented my algorithms using OpenMP and CUDA, specifically regarding the challenges I encountered and the optimizations I made. Finally, the timing results of the implementation of my parallel algorithms and the serial algorithm with varying input sizes (i.e. length of the two input strings) and computational resources (i.e. number of threads) are presented and analyzed.

II. Serial Algorithm for Solving the SCS Problem

Before designing the parallel algorithm, I first had to write the serial dynamic programming (DP) algorithm for finding the shortest common supersequence of two input strings. The SCS problem, like most other problems solved using dynamic programming, has two important properties. First, it has an optimal substructure, meaning that the problem can be broken down into smaller, simpler subproblems (until the solution becomes trivial), and the solution to the original problem can be found using the solution from the subproblems. Second, it also has overlapping subproblems, meaning that the problem usually solves the same subproblem over and over, or in other words, reuse the solution to the same subproblem multiple times, rather than always generating new subproblems.

These two properties are both a blessing and a curse. They are a blessing because they are what allows the problem to be solved using dynamic programming, where the solutions to the subproblems are memoized, i.e. saved for reuse, which drastically reduces the time complexity to polynomial time from exponential time of a naive brute-force or recurrence algorithm. On the other hand, they are also a curse because they introduce data dependencies, in which the current step depends on the results of the previous steps, thus making DP algorithms hard to efficiently parallelize without substantial change. The difficulty in parallelization becomes more evident later in the current section and in the next section.

Back to the serial DP algorithm, using these two properties, we could make the following observations about the SCS problem.

First, if the character a appeared in both input sequences X and Y once, then we need to include a in the supersequence exactly once, since including it twice would be redundant, consequently resulting in the supersequence not being the shortest. To put this into context of the DP algorithm, assume we have already found the SCS of strings X and Y up to but not including the last character, and the last character of X and Y are the same (i.e. both being a), then it must mean that the SCS of X and Y including the last character is just the current/existing SCS concatenated with a. Formally, this is written as the following.

Observation 1. $SCS(X^\frown a, Y^\frown a) = SCS(X, Y)^\frown a$, for all strings X, Y and all characters a, where $^\frown$ denotes string concatenation.

Second, assume the last character in strings X and Y, being a and b respectively, are different, then we know that we have to eventually include both a and b in the supersequence by the definition of *common supersequence*. However, to find the optimal way to include these two characters that would result in the shortest common supersequence, we need to break it down into 2 subproblems: (1) find the SCS between X (including a) and Y excluding its last character b, and (2) the SCS between X excluding its last character a and Y (including b). The minimum between the 2 supersequences generated from these 2 subproblems would then be used to concatenate with the character that was excluded, to generate the shortest common supersequence of the larger problem. This is possible because of the optimal substructure nature of the SCS problem. Formally, this is written as the following.

Observation 2. If a and b are two distinct characters (i.e. $a \neq b$), then $SCS(X^\frown a, Y^\frown b)$ is the minimum-length string in the set $\{SCS(X^\frown a, Y)^\frown b, SCS(X, Y^\frown b)^\frown a\}$.

Based on these two observations, we can derive the recurrence relation for finding the SCS between two input strings X and Y. Let $X = x_1...x_n$, $Y = y_1...y_m$ be two arbitrary input strings (of length n and m respectively). Let x_i denote the character in X at position i and X_i denote the first i characters in X, $1 \leq i \leq n$. Similarly, let y_j denote the character in Y at position j and Y_j denote the first j characters in Y, $1 \leq j \leq m$. Then, the SCS between X and Y, $SCS(X_n, Y_m)$, can be found using the following recurrence relation.

$$SCS(X_i, Y_j) = \begin{cases} Y_j & \text{if } i = 0 \text{ (Base Case 1)} \\ X_i & \text{if } j = 0 \text{ (Base Case 2)} \\ SCS(X_{i-1}, Y_{j-1})^\frown x_i & \text{if } x_i = y_j \text{ (Recursion Case 1)} \\ min\{SCS(X_i, Y_{j-1})^\frown y_i, SCS(X_{i-1}, Y_j)^\frown x_i\} & \text{if } x_i \neq y_j \text{ (Recursion Case 2)} \end{cases} \tag{1}$$

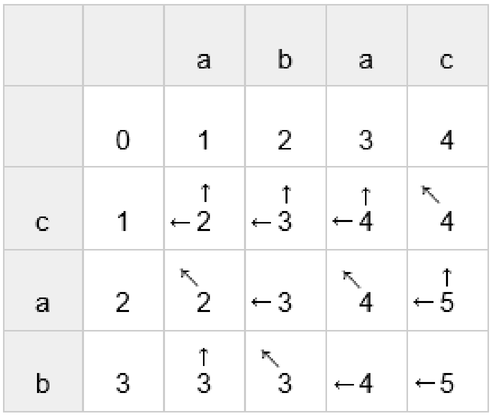
The first two lines form the base cases: If either X or Y is an empty string, then the SCS is simply the other string. The third line uses Observation 1 and the fourth line uses Observation 2, as explained previously. The time complexity of the recurrence relation if implemented naively is $O(2^{(n+m)})$. However, by employing the idea of memoization of dynamic programming, i.e. saving the already-computed SCS of the subproblems in a matrix, the time complexity is reduced to $O(nm)$.

However, saving the SCS in the memoization matrix for every subproblem can lead to large memory overhead, as storing strings is expensive. Thus, it is common to split computing the SCS into two steps. First, compute a memoization matrix that only keeps track of the current length of the SCS of each subproblem; this is the dynamic programming step, which takes $O(nm)$. Second, use the tracing-back method by starting from the bottom right of the memo trace all the way back to the top left, to construct the SCS; this step takes $O(n + m)$. My term project focused exclusively on parallelizing the former step, which is just computing the length of the SCS using dynamic programming. This is reasonable because the latter step is inherently sequential as the algorithm must determine the current character in the SCS before moving on to the next character, making it very difficult to parallelize efficiently. Moreover, its time complexity is linear, meaning that it is already pretty fast so it is hard to further speed it up with reasonable efficiency. Thus, for simplicity, although my actual implementations of the serial algorithm and my two parallel algorithms did include the second step (for finding the SCS), this step was excluded from the timing results because it was implemented the same way (i.e. serially) in all three algorithms. This paper also does not further elaborate on this tracing-back method for finding the SCS because it is just an intuitive serial algorithm, which is not the focus of this class.

Deciding to only focus on computing the length of the SCS using dynamic programming memoization, we can rewrite the recurrence relation to be the following. Let M denote the memoization matrix of size $(n + 1) \times (m + 1)$ which stores the length of the SCS of each subproblem, and $M[i][j]$ denote the length of the $SCS(X_i, Y_j)$ given where $0 \leq i \leq n$, $0 \leq j \leq m$. Then, $M[i][j]$ can be computed by using the following equation iteratively.

$$M[i][j] = \begin{cases} j & \text{if } i = 0 \text{ (Base Case 1)} \\ i & \text{if } j = 0 \text{ (Base Case 2)} \\ M[i-1][j-1] + 1 & \text{if } x_i = y_j \text{ (Case 1)} \\ min\{M[i][j-1], M[i-1][j]\} + 1 & \text{if } x_i \neq y_j \text{ (Case 2)} \end{cases} \tag{2}$$

Using this equation, $M[n][m]$ then is the length of the SCS of input strings X and Y. Figure 1 shows an example of computing the memoization matrix given $X = cab$ and $Y = abac$ by iteratively using the equation for each $0 \leq i \leq 3$ and $0 \leq j \leq 4$ (i.e. a nested for loop).



		a	b	a	c
	0	1	2	3	4
c	1	↑ ←2	↑ ←3	↑ ←4	↖ 4
a	2	↖ 2	←3	↖ 4	↑ ←5
b	3	↑ 3	↖ 3	←4	←5

Fig. 1: Memoization matrix of input sequences ”cad” and ”abac”

The matrix is of size 4×5. Each cell at index $[i][j]$ of the matrix stores the length of the SCS of strings X_i and Y_j (i.e. the SCS of a subproblem); this means that the length of the SCS of the two original input strings X and Y is stored at the bottom right cell or $M[3][4]$. The arrow(s) in each cell indicates from which other cells did the current cell use to compute the current length of SCS.

III. Parallel Algorithms for Solving the SCS Problem

A. Analysis of Data Dependencies

To determine how to parallelize the serial dynamic programming algorithm, I needed to first analyze the data dependencies of each cell in the memoization matrix, i.e. for which cells the current cell needed to wait before computing the length of the SCS in this current cell. Knowing the data dependencies in each cell, the parallel algorithm then needed to parallelize the computation of the cells that do not depend on each other. However, it was not this simple because of the nature of DP algorithm (i.e. the two properties I mentioned at the beginning of the previous section).

Analyzing the equation (2) we devised previously along with the example shown in Figure 1, we can see that for each cell that is not in the leftmost column (j = 0) nor in the top row (i = 0), computing it requires its left, top, and top-left cells to be finished first. These dependencies are depicted in Figure 2.

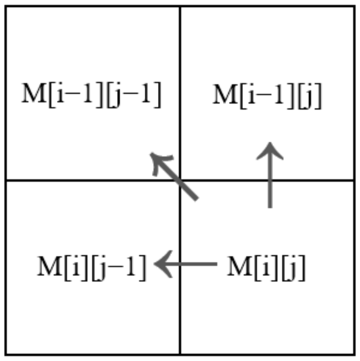


Fig. 2: Data dependencies of a cell in memoization matrix

This means that we cannot parallelize the computation of one row, because each cell in the same row needs to wait for its left cell to finish computation, consequently, the cell in the rightmost column needs to wait for all m cells before it, where m is the length of the input string Y, making this computation inherently sequential. On the other hand, we also cannot parallelize the computation of one column, because each cell in the same column also needs to wait for its top cell to finish, meaning that the cell in the bottom row needs to wait for n cells, where n is the length of the input string X, again, rendering this computation inherently sequential. If I cannot parallelize the computation in a row nor in a column, then what else can I do? At this point, I thought that I had encountered a dilemma that could not be solved. However, after pondering the problem for a considerable time, I was able to devise two parallel algorithms to the SCS problem.

B. Anti-diagonal Algorithm

After realizing that I could not parallelize the computation in the same row nor the computation in the same column, there was still one more "direction" (for the lack of a better term) in the matrix I could try: the diagonal. Although the normal diagonal (top left to bottom right) in the memoization matrix also has data dependencies as indicated by Figure 1, the anti-diagonal (top right to bottom left) of the matrix does not.

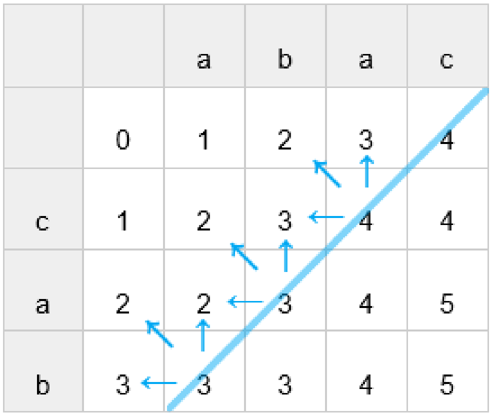


Fig. 3: Computing an anti-diagonal of memoization matrix

In Figure 3, we can see that entries on the same anti-diagonal do not have data dependencies on each other, i.e. the cells on the same anti-diagonal do not have to wait for each other to finish computation before doing their own computation. This indicates that the anti-diagonal can be in fact parallelized. Using this intuition, the first parallel algorithm, namely the *anti-diagonal parallel algorithm*, for the SCS problem was devised. A very bare-bone pseudocode is written below, mainly focusing on the overall structure and where the parallelization occurs, while abstracting the intricacies of the loop iterators and matrix indexing (discussed later in the implementation section). Figure 4 shows how this algorithm iterates through an example memo matrix.

Algorithm 1 Anti-diagonal Parallel Algorithm

```
    Input: two strings $X = x_1...x_n$ and $Y = y_1...y_m$ over a finite alphabet $\Sigma$
    Output: the length of the SCS between $X$ and $Y$
1: $M \leftarrow$ memo$[n+1][m+1]$
2: for each anti-diagonal in $M$ do
3:     for each cell in current anti-diagonal in parallel do
4:         compute length of SCS using equation (2)
5:     end for
6: end for
7: return $M[n][m]$
```

Because each entry in the same anti-diagonal can be computed independently from each other, the inner for loop can be executed completely in parallel, which theoretically takes $O(1)$ time. The outer loop takes $O(n+m)$ time since there are $n+m+1$ anti-diagonals in the matrix. Thus, the final theoretical time complexity of the anti-diagonal parallel algorithm of SCS is $O(n+m)$, which is linear with respect to input size. This presents a significant improvement from the serial DP algorithm which takes $O(nm)$ or quadratic time.

Later after researching online, it turned out that the anti-diagonal approach has also been used in other problems with similar patterns of data dependencies, such as the parallel Smith-Waterman algorithm [7]. But the use of the anti-diagonal parallel algorithm in the SCS problem has never been done previously, making my algorithm a valuable addition to the parallel computing field.

Although the inner loop of my current algorithm can be parallelized and theoretically takes constant time given enough resources, there are some clear disadvantages of this approach in practice. The first major disadvantage practically-speaking is

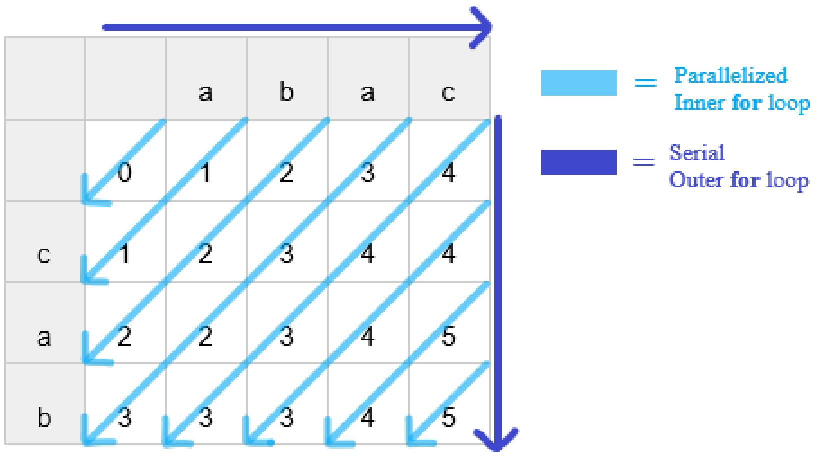


Fig. 4: Computing all anti-diagonals of memoization matrix using algorithm

that the parallelization of this inner loop is unbalanced, as the size of most anti-diagonals in the memo matrix are different, i.e. the size starts at 1, increases to n (length of input string X), and decreases to 1 again (see Figure 4 for example). This dynamic fluctuation in size of the anti-diagonals is especially true when the length of the two input strings are similar. Consequently, some computational resources allocated, e.g. threads, would be just idling for the anti-diagonals with a small size. The inability to fully utilize all computational resources would inevitably decrease the efficiency and reduce the parallelism of the algorithm.

The second major disadvantage is the potential memory access overhead, as the entries in an anti-diagonal, despite being adjacent theoretically, are definitely not close to each other in a real life memory architecture, which commonly has entries on the same row stored contiguously. Since each entry on the anti-diagonal is on a different row and a different column, accessing them likely requires constantly evicting and bringing in different memory blocks between RAM and cache (or even between RAM and disk if the input strings are large enough). This introduces potential memory access overhead that likely is not present in a serial DP implementation since the iterations are contiguous per row.

Thus, the parallel algorithm would (hypothetically) perform better if it could be computed

1) with the parallel portion being the same size for each iteration, efficiently using the computational resources, and
2) preferably contiguous in memory (i.e. in the same row), reducing memory access overhead.

C. Reducing Data Dependencies

To achieve the above two goals, I spent considerable time trying to rethink the problem in different ways, closely analyzing the data dependencies by going through many examples while drawing many diagrams. Finally, I devised an equation that allowed me to reduce the data dependencies from 3 to 2, for each cell (not on row 0 nor column 0) in the memoization matrix.

By equation (2) and Figure 2, we know that each entry, $[i][j]$, depends on

1) the entry to its left on the same row, $[i][j-1]$,
2) the entry to its top on the same column $[i-1][j]$, and
3) the entry to its top left $[i-1][j-1]$.

Data dependency (1) makes the algorithm row-wise dependent, since entries on the same row need to wait for their left entries to finish, and data dependency (2) makes the algorithm column-wise dependent, since entries on the same column need to wait for their top entries to finish. Thus, as long as I remove one of these data dependencies, I would be able to make the algorithm independent per row or per column, consequently meeting the two goals of efficient parallelization I stated previously. It turned out that I was indeed able to remove the data dependency (1), i.e. to calculate the entry $M[i][j]$ without directly using the value at $M[i][j-1]$, resulting in the algorithm being row-wise independent.

The intuition lies in finding the value at (i.e. computing) $M[i][j-1]$ "on the fly" by only using the entries from the previous row $(i-1)$ and not using any entries from the same row (i), when computing $M[i][j]$. Specifically, I made the following observations for each case that $M[i][j-1]$ can take on.

First, if the entry $M[i][j-1]$ is on the leftmost column of the memoization matrix (i.e. $j-1=0$), then the value of this entry is simply i, by the definition of Base Case 2 in equation (2). By doing so, we are able to find the value at $M[i][j-1]$ without directly using this entry (i.e. wait for this entry to be computed) nor any other entries on the same row. Consequently, this makes computing $M[i][j]$ in this case to be row-wise independent.

Second, if the entry $M[i][j-1]$ is not on the leftmost column, but $x_i = y_{j-1}$ (i.e. the character in X and the character in Y corresponding to this entry in the memoization matrix are the same), then the value of $M[i][j-1]$ is simply $M[i-1][j-2]+1$, by definition of Case 1 in equation (2). By doing so, we are able to find the value at $M[i][j-1]$ by only using the entry in the previous row ($i-1$) and not any entries in the same row (i). Just like the previous observation, this also makes computing $M[i][j]$ in this case to be row-wise independent.

Lastly, if the entry $M[i][j-1]$ is not on the leftmost column and $x_i \neq y_{j-1}$, then the value of $M[i][j-1]$ is $min\{M[i-1][j-1], M[i][j-2]\}+1$, by definition of Case 2 in equation 2. In this case, the former entry in this set (that we are taking the minimum of) is not in the same row, but the latter entry still depends on the same row i. However, to find the value at $M[i][j-2]$ without directly using this entry, we can simply reapply these three observations we made to attempt to compute $M[i][j-2]$ without using any data from the same row. Suppose we end up in the last case again where it requires $M[i][j-3]$, we simply reapply these three observations again, so on and so forth, until it either meets the first case (the entry has reached the leftmost column) or the second case (the character in X and Y corresponding to this entry is the same). Thus, we can write the following recurrence relation to capture these three observations (using the same notations as before).

$$M[i][j-1] = \begin{cases} i & \text{if } j-1=0 \text{ (Base Case 1)} \\ M[i-1][j-2]+1 & \text{if } x_i = y_{j-1} \text{ (Base Case 2)} \\ min\{M[i][j-2], M[i-1][j-1]\}+1 & \text{if } x_i \neq y_{j-1} \text{ (Recursion Case)} \end{cases} \tag{3}$$

To reiterate, calculating $M[i][j-1]$ in the first two cases does not require any entry on the ith row, thus already satisfying our goal of not using entries on the same row as $M[i][j-1]$. In other words, as soon as one of these cases are met, the relation is considered finished since we have found a way to calculate $M[i][j-1]$ without using anything on the same row, hence they are called base cases. Only in the third case does the relation depend on an entry in the same row, specifically, $M[i][j-2]$. However, we can simply keep reapplying (i.e. do recursion using) this relation to compute this value until we hit either base case, i.e. until we can calculate the value $M[i][j-k]$ without using any entries in the same row, where k is the minimum number of times this relation was applied (to meet either base case). In other words, this recursion process ends at the kth step when either $j-k=0$ (Base Case 1) or $x_i = y_{j-k}$ (Base Case 2). Thus, we can rewrite the recurrence relation (3) to be the following equation using substitution.

$$M[i][j-1] = \begin{cases} i+k-1 & \text{if } j-k=0 \text{ (Case 1)} \\ M[i-1][j-k-1]+k & \text{if } x_i = y_{j-k} \text{ (Case 2)} \end{cases} \tag{4}$$

Using this equation (4), we can then rewrite the original DP equation (2) to be the following.

$$M[i][j] = \begin{cases} j & \text{if } i=0 \text{ (Base Case 1)} \\ i & \text{if } j=0 \text{ (Base Case 2)} \\ M[i-1][j-1]+1 & \text{if } x_i = y_j \text{ (Case 1)} \\ min\{i+k-1, M[i-1][j]\}+1 & \text{if } j-k=0 \text{ (Case 2)} \\ min\{M[i-1][j-k-1]+k, M[i-1][j]\}+1 & \text{if } x_i = y_{j-k} \text{ (Case 3)} \end{cases} \tag{5}$$

From this reformulated equation 5, we can see that we have successfully removed the data dependency on the same row, as all ith row data can now be calculated from the $i-1$th row data, reducing the number of dependencies per entry from 3 to 2. As a result, entries on the same row can be computed independently in parallel, successfully creating a row-wise independent algorithm (described in the next section).

D. Row-wise Independent Algorithm

Before writing the actual row-wise independent algorithm, there are still a few issues that we need to address.

Although equation (5) does remove the data dependency in the same row, it does need to first determine k using the recurrence relation (3), which is very slow (especially when string Y is large), if implemented using a naive recursion algorithm. Instead, we can use the idea of DP here again: first compute all of the k values corresponding to each i, j in M, and store them in another memoization matrix, call it A, as such, when computing $M[i][j]$ later, the algorithm can directly use the precomputed values stored in the memo A rather than calculating them repeatedly "on the fly" with the recurrence relation (3).

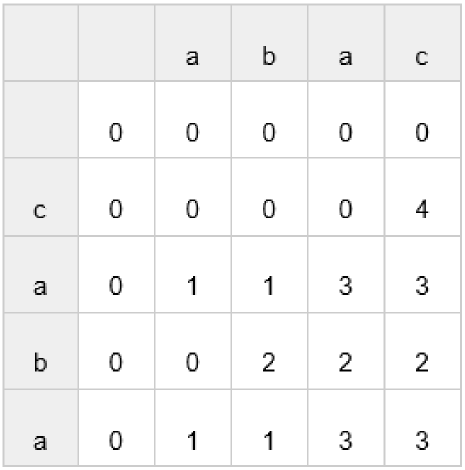
Specifically, let A be the memoization matrix of size $(n+1)\times(m+1)$, where n is the length of input string $X = x_1...x_n$ and m is the length of input string $Y = y_1...y_m$. We apply the following equation to compute each $A[i][j]$, for all $0 \leq i \leq n$ and $0 \leq j \leq m$.

$$A[i][j] = \begin{cases} 0 & \text{if } j=0 \\ j & \text{if } x_i = y_j \\ A[i][j-1] & \text{otherwise} \end{cases} \tag{6}$$

As a result, $A[i][j]$ indicates the closest index (to index j) in string Y such that

1) it either equals to 0, i.e. this index is on the leftmost column (i.e. as if we have reached the leftmost column or Base Case 1 had we used recurrence relation (3)), or
2) the character at this index in string Y, i.e. y_j, and the character at index i in string X, x_i are the same (i.e. as if we have met Base Case 1 had we used recurrence relation (3)).

In other words, $A[i][j] = j - k$, by definition.



		a	b	a	c
	0	0	0	0	0
c	0	0	0	0	4
a	0	1	1	3	3
b	0	0	2	2	2
a	0	1	1	3	3

Fig. 5: Example of a computed memoization matrix A

For example, in Figure 5, $A[2][2] = 1$ means that the closest index (to index 2) in Y that meets either of the two cases above is index 1, which specifically meets case 2 stated above as $y_1 = x_2$. Consequently, when computing $M[2][2]$ using equation (5), we can immediately know that $A[2][2] = j - k = 1$ and directly use this value (to get both $j - k$ and k), without the need to calculate k "on the fly" using recurrence relation (3). We can then rewrite the equation (5) to incorporate memo A (in Case 2 and 3 where it uses $j - k$ and k).

$$M[i][j] = \begin{cases} j & \text{if } i = 0 \text{ (Base Case 1)} \\ i & \text{if } j = 0 \text{ (Base Case 2)} \\ M[i-1][j-1] + 1 & \text{if } x_i = y_j \text{ (Case 1)} \\ min\{i + j - 1, M[i-1][j]\} + 1 & \text{if } A[i][j] = 0 \text{ (Case 2)} \\ min\{M[i-1][A[i][j] - 1] + (j - A[i][j]), M[i-1][j]\} + 1 & \text{if } x_i = y_{A[i][j]} \text{ (Case 3)} \end{cases} \quad (7)$$

Additionally, computing the memo A can also be parallelized because equation (6) is column-wise independent, as entries on the same column can be computed independently of each other. In other words, although each row in A needs to be computed serially, all of the rows can be computed in parallel.

Thus, given all of the information described above, we can write the following row-wise independent algorithm.

Algorithm 2 Row-wise Independent Parallel Algorithm (Naive)

Input: two strings $X = x_1...x_n$ and $Y = y_1...y_m$ over a finite alphabet Σ
Output: the length of the SCS between X and Y

```
1: $A \leftarrow$ memo$[n+1][m+1]$
2: $M \leftarrow$ memo$[n+1][m+1]$
3: for $i = 0$ to $n$ in parallel do
4:     for $j = 0$ to $m$ do
5:         compute $A[i][j]$ using equation (6)
6:     end for
7: end for
8: for $i = 0$ to $n$ do
9:     for $j = 0$ to $m$ in parallel do
10:        compute $M[i][j]$ using equation (7)
11:    end for
12: end for
13: return $M[n][m]$
```

This algorithm first computes memo A (storing $j-k$), with all of the rows being computed in parallel. This means that the outer loop theoretically takes $O(1)$ time (if we have enough resources to compute all rows at the same time), so the time complexity only depends on the inner loop, which is $O(m)$. The algorithm then computes memo M (storing the length of the SCS for each subproblem), with entries on the same row being computed in parallel. This means that the inner loop theoretically takes $O(1)$ time (if we have the same number of resources as the number of columns to compute all of them at the same time), so the time complexity depends on the outer loop, which is $O(n)$. Thus, the overall time complexity of the row-wise independent parallel algorithm is $O(max\{m,n\})$, which is theoretically slightly faster than $O(m+n)$ of the anti-diagonal parallel algorithm from before. Moreover, the new algorithm now embodies both goals stated previously at the end of the Anti-diagonal Algorithm section: The parallel portions all have the same size between the iterations, and the memory accesses are contiguous in memory. Thus, I have successfully created another novel and efficient parallel algorithm for solving the SCS problem.

E. *Optimized Row-wise Independent Algorithm*

Before implementing the row-wise independent algorithm in code, I realized that it could be more optimized in terms of memory usage and the number of comparisons.

First, the memo A currently has a size of $(n+1)\times(m+1)$, which is storing more than necessary. Looking at Figure 5 for instance, we can see that row 2 and row 4 have the exact same values because they both correspond to the character ‘a’ the string X. In other words, if two rows in memo A correspond to the same character in string X, then they will have the same content which is redundant to store them twice. From this observation, we can reduce the number of rows in the memo A from the size of the input string X to the size of the finite alphabet Σ which characters of the input strings belong to. This would significantly reduce the memory usage when the input string X is large.

Formally, let C be an array of all unique characters of the finite alphabet Σ and A be the memo of size $|\Sigma|\times(m+1)$, then we can rewrite equation (6) to be the following equation to compute $A[i][j]$ for all $0\le i<|\Sigma|$ and $0\le j\le m$.

$$A[i][j]=\begin{cases}0 & \text{if } j=0\\ j & \text{if } C[i]=y_j\\ A[i][j-1] & \text{otherwise}\end{cases} \tag{8}$$

Then, whenever indexing into A in equation (7), instead of naively using i, we just need to find the index which the character x_i corresponds to in the array C. Before rewriting equation (7) to implement this change, we can also simplify it a bit by reducing the comparisons. Specifically, we can observe that Case 3 essentially includes Case 1 as well, if $j-k=j$ or $k=0$, i.e. the closest index to j that has the corresponding character in string Y equal to the current character in string X is just the index j itself. Hence, Case 1 is redundant and we can simply remove it from equation (7) while still capturing all cases. Furthermore, as a minor improvement, we can also change the Case 3’s ”if” comparison to “otherwise”, since this is just the last case when all previous ”if” cases are not met. With these optimizations, we can rewrite equation (7) as the following.

$$M[i][j]=\begin{cases}j & \text{if } i=0 \text{ (Base Case 1)}\\ i & \text{if } j=0 \text{ (Base Case 2)}\\ min\{i+j-1, M[i-1][j]\}+1 & \text{if } A[c][j]=0 \text{ (Case 1)}\\ min\{M[i-1][A[c][j]-1]+(j-A[c][j]), M[i-1][j]\}+1 & \text{otherwise (Case 2)}\end{cases} \tag{9}$$

Note that c in equation (9) denotes the index of array C where $C[c]=x_i$, i.e. the character at index i of X is the same character at index c of C; because each row in memo A corresponds to a character in C, c can be used to index into the row of memo A as shown in the equation above. This index can also be easily computed in $O(1)$ time using a pre-defined hash function that converts a character to its index in C, given a finite alphabet.

Incorporating the above optimizations, we can then rewrite Algorithm 2 to improve its time and memory (see Algorithm 3). Although this new algorithm still has the same theoretical time complexity as before, it would perform much better practically, especially when the input string Y is much larger than the alphabet size, which is generally the case. Moreover, the simpler branching in equation 9 would make the implementation slightly faster as well. Note that there were even more optimizations done in the actual implementations of Algorithm 3 in code; see the next section for more information.

IV. Implementation

I first implemented the serial DP algorithm in C++. Then, I implemented both the anti-diagonal parallel algorithm 1 and the row-wise independent parallel algorithm 3 on the CPU using OpenMP/C++. Lastly, I implemented only the row-wise independent parallel algorithm on the GPU using CUDA. One major reason for not implementing the anti-diagonal parallel algorithm on the GPU was because it already performed worse on the CPU with larger input sizes; moreover, the indexing to

Algorithm 3 Row-wise Independent Parallel Algorithm (Optimized)

```
Input: two strings $X = x_1...x_n$ and $Y = y_1...y_m$ over a finite alphabet $\Sigma$
       array $C$ with all unique characters in $\Sigma$
       a hash function that takes in a character $\in \Sigma$ and returns index $c$
Output: the length of the SCS between $X$ and $Y$
 1: $A \leftarrow$ memo$[n+1][m+1]$
 2: $M \leftarrow$ memo$[n+1][m+1]$
 3: for $i = 0$ to $|\Sigma| - 1$ in parallel do
 4:     for $j = 0$ to $m$ do
 5:         compute $A[i][j]$ using equation (8)
 6:     end for
 7: end for
 8: for $i = 0$ to $n$ do
 9:     for $j = 0$ to $m$ in parallel do
10:         compute $M[i][j]$ using equation (9)
11:     end for
12: end for
13: return $M[n][m]$
```

the anti-diagonal was very complicated in the kernel and doing repeated kernel calls with different block and grid dimensions each time (since the length of the anti-diagonal varies for each iteration) was also troublesome. Thus, I decided to not do it.

The serial DP algorithm needs no further elaboration, since it was very straightforward and easy to implement. The remainder of this section is dedicated to briefly explaining how I implemented each of the parallel algorithms and how I attempted to improve their parallelism.

A. CPU/OpenMP

1) Anti-diagonal parallel algorithm: As shown in Algorithm 1, the high-level idea of the anti-diagonal algorithm is actually quite simple, as it just involves computing the memoization matrix M using the same equation as the serial algorithm. The main difficulty was actually the indexing for looping through all of the anti-diagonals in the memoization matrix. I used two indices/iterators, i corresponding to row and j corresponding to column. For the outer for loop, at the end of its each iteration, I first only increment j until it is equal to m (i.e. length of string Y; i.e. number of columns in $M - 1$) while holding i constant, then I incremented i until it is equal to n (i.e. length of string X; i.e. number of rows in $M - 1$) while holding j constant, thus iterating for a total of $n+m+1$ times. The indigo-colored rightward and downward arrows in Figure 4 illustrate the order in which the outer for loop goes through the memo. Then, for the inner loop, I use two other iterators, a for indexing into row, initially assigned to i, and b for indexing into column, initially assigned to j; at the end of each of its iteration, I increment a but decrement b, i.e. the index begins at the "top right" and moves towards the "bottom left", as shown by the light-blue-colored arrows in Figure 4. Thus, when taking both the inner and outer loop together, I would successfully loop through all of the anti-diagonals in the memoization matrix M.

I used '#pragma omp parallel for' to parallelize the inner for loop on line 3 of Algorithm 1 (i.e. going over the entries on a single anti-diagonal). This OpenMP directive is a combination of two separate directives: '#pragma omp parallel' and '#pragma omp for'. The former, as stated in the OpenMP documentations, instructs the compiler to parallelize the block of code enclosed by the curly braces; specifically, it would create a team of threads, with each thread executing all statements within the parallel region, except for work-sharing constructs, which is a general term describing the latter directive. The '#pragma omp for', an example of a work-sharing construct, instructs the compiler to distribute loop iterations among the team of threads (e.g. thread 0 executes iterations 0 to 4, thread 1 executes 5 to 9, etc.). By using the combined directive, it successfully achieves what I intended in Algorithm 1: The inner for loop is distributed among the threads, with each thread executing part of the loop in parallel. In other words, if it were possible to have the same number of threads as the number of entries in the anti-diagonal (along with enough memory resources) to execute in parallel, then all of the entries in the anti-diagonal would be computed at the same time, being what I envisioned in the previous section.

After getting my code to produce the correct results (discussed in a later section on how I verified my results), I tried to customize the OpenMP directive by playing around with its schedule parameter (i.e. how exactly does OpenMP distribute the work among the threads): Specifically, I tried "dynamic", "guided", and "static", and the preliminary timing results with input size 2000 (string X) by 2000 (string Y) did not make any difference. Thus, I just went with the default static scheduling. I also tried to see if using a varying number of threads for each anti-diagonal would improve the parallelism (i.e. more threads when the anti-diagonal is larger and less threads when the anti-diagonal is smaller, similar to weak scaling) by setting 'omp_set_dynamic(true)'. Unfortunately, it not only did not speed up my code, it actually made my code slower than if I had just kept the number of threads constant for each anti-diagonal, which was what I ended up doing. The number of threads is specified using the ‘OMP_NUM_THREADS‘ environment variable during testing.

This pretty much sums up everything I tried for implementing and improving the parallelism of this algorithm. For the actual code, please see the ”scs_anti_diagonal” function in ”parallel_omp_anti_diag_scs.cpp”.

2) Row-wise independent parallel algorithm: The input to Algorithm 3 first needs an array containing all of the unique characters in a finite alphabet and a hash function that maps each character (belonging to the alphabet) to an index in the array. For this term project, the alphabet I decided to use consists of the 26 lower-case letters of the English alphabet. Consequently, the hash function is just converting the character to its ASCII value and then subtracting 97, such that ‘a’ maps to 0, ‘b’ maps to 1, ..., ‘z’ maps to 25. Defining the two memoization matrices, A and M, as 2D arrays, I then implemented the remaining code closely following Algorithm 3. The code is quite straightforward, needing no further explanation about how it works.

For the first parallel portion, i.e. the first outer for loop on line 3 of Algorithm 3, I used the ‘#pragma omp parallel for‘ directive, along with the parameter ‘num_threads(ALPHABET_SIZE)‘ to indicate the number of threads to create for this parallel region. Note that the number of threads here is always ALPHABET_SIZE (i.e. $|\Sigma| - 1$ or 26) regardless of the size of the input strings. The reason for this is because I know that the number of iterations of the outer for loop, corresponding to the number of rows of memo A, is exactly and always equal to ALPHABET_SIZE or 26, irrespective of input size. As a result, I can always allocate this many threads such that each thread is just responsible for computing exactly one row (inner for loop), and with enough computational and memory resources in a system, they can be done completely in parallel.

For the the second parallel portion, i.e. the second inner for loop on line 9 of Algorithm 3, I also used the ‘#pragma omp parallel for‘ directive, with the number of threads specified using ‘OMP_NUM_THREADS‘ environment variable during testing, just like what I did for the anti-diagonal implementation. This concludes my initial attempt at the implementation of this algorithm.

After some preliminary testing, I realized that the runtime of my code had considerable fluctuations between runs: For instance, it would have a range of time from 5 ms to 20 ms when running the code 20 times with the same input, despite always producing the same and correct output. After searching online, reading OpenMP documentations, and reviewing the lectures to test different things, I realized that this fluctuation might have been due to the constant creation and destruction of threads as a result of the second ‘#pragma omp parallel for‘ directive. Specifically, because this directive was used for the inner for loop (line 9 of Algorithm 3), the threads for this parallel region were created at the start of the current iteration of the outer for loop, and destroyed at the end of the current iteration of the outer for loop, and then created again at the start of the next iteration, so on and so forth. Thus, I believed that the constant thread allocation and deallocation likely caused this big fluctuation in execution time between each run.

To fix this problem, I had to start the parallel region, using ‘#pragma omp parallel‘, before the outer for loop on line 8 of Algorithm 3, and instruct OpenMP to distribute the inner for loop (on line 9) among the threads in this parallel region, using ‘#pragma omp for‘. By making this change, OpenMP would create the threads before entering the outer loop and destroy them only after exiting the outer loop, i.e. the threads would be created and destroyed exactly once. This successfully eliminates the thread allocation/deallocation overhead issue from before. Additionally, I also had to change the outer loop to be a while loop checking ‘while($i \leq n$)‘, where i represents the row index and is a shared variable between threads; i is incremented by 1 at the end of the while loop’s current iteration (i.e. after the inner for loop), with a ‘#pragma omp single‘ directive placed immediately preceding it. The reason for this change is because creating the threads before entering the outer loop means that this outer loop would be executed by all threads. Therefore, I had to use a shared variable i indicating the current iteration and explicitly instruct OpenMP to only use 1 thread to increment i; doing so would make all of the threads share the iterations of the outer loop, rather than each thread doing their own iterations. As a side note, this would inevitably introduce some synchronization overhead, as all other threads have to wait for this one thread to finish incrementing i, but this was probably negligible overall. After implementing these changes, I ran my code with the same inputs as before for 20 times, and the fluctuation was only from 5 to 8 ms, being considerably smaller than before.

Fixing the fluctuation in timing issue, I then tried to further optimize the parallel portions of my code by reducing the number of branches/conditions (if-else) in them. Using Algorithm 3 as a reference, I took the first case of equation (8) (currently computed as part of line 5 of Algorithm 3) to be outside of the inner for loop on line 4 of Algorithm 3, because the 0th element of each row is always 0 by definition. Consequently, this inner for loop only goes from 1 to m and there are only one if and one else branch (corresponding to the latter two cases in equation (8)) in it, reducing the number of branches from 3 to 2.

Then, using a similar strategy, I was able to move the two Base Cases of equation (9) outside of the inner for loop on line 9 of Algorithm 3 as well, by

1) doing Base Case 1 as a separate for loop iterating through the entire 0th row before doing the loop on line 8, which then only loops from 1 to n;
    - Note that this separate for loop is placed in the same parallel region defined by the ‘#pragma omp parallel‘ directive before line 8, and the iterations of this for loop is distributed using the ‘#pragma omp for‘ directive, just like what I did for the inner for loop on line 9.

2) doing Base Case 2 outside of the inner for loop on line 9, which then only loops from 1 to m.

By making these changes, I successfully reduced the number of branches from 4 to 2, as there are only one if and one else branch (corresponding to the last two cases in equation (9)) in the inner for loop.

Furthermore, I was actually able to completely eliminate the remaining 2 branches by doing Case 1 and Case 2 solely relying on boolean logic (inside the inner for loop). This required a very in-depth analysis of the memoization matrix M (that is keeping track of the length of the SCS of each subproblem), which led me discovering an observation that the diffrence between entries $M[i-1][j]$ and $M[i][j-1]$ is at most 1. This allowed me to use only $M[i-1][j]+$some boolean to calculate $M[i][j]$. However, I would not elaborate on this any further in the paper, because not only did this change not speed up the execution time of my code, it also does not have a significant impact on the parallelism of the algorithm.

Thus, this concludes the discussion of my implementation of the row-wise independent parallel algorithm on the CPU using OpenMP. For the actual code, please see the ”scs_rowwise_independent_optimal” function in ”parallel_omp_scs.cpp”.

B. *GPU/CUDA*

I only implemented the row-wise independent parallel algorithm in CUDA. Similar to what I did for the CPU/OpenMP version, I defined an array containing the 26 lower-case letters in the English alphabet in-order (as a global device variable), and a hash function that converts a character to ASCII value and then subtracting 97, such that ‘a’ maps to index 0, ‘b’ maps to index 1, . . . , ‘z’ maps to index 25. Then, I allocated $n+1$ bytes of memory for string X and $m+1$ bytes of memory for string Y on the GPU using ‘cudaMalloc‘ such that the kernels later on would be able to access the two input strings. The two memoization matrices A and M were also allocated on the GPU using ‘cudaMalloc‘, but this time as 1D arrays unlike previously on the CPU where they were 2D arrays. This design choice was made because ‘cudaMemcpy‘ works best with 1D contiguous memory and this was also what I did in Homework 4. The only change I had to make to Algorithm 3 due to the 1D representation of the memos was the indexing into matrix A and M, for which I had to convert the separate row index and column index used for accessing a 2D array into one single index for accessing the corresponding entry in the 1D array using the following conversion formula:

$$index = rowIndex \times numColumns + columnIndex \tag{10}$$

The rowIndex and columnIndex in the equation (10) represents the original row and column index if the memo were 2D; numColumns represents the number of columns in the memo if it were 2D, which would be $m+1$ for both memo A and M. This conversion equation is used in the CUDA code whenever it indexes into memo A or M. See figure 6 for an example of a 2D array representation of the memo vs. a 1D array representation of the memo, where the number in each cell represents their corresponding indexing.

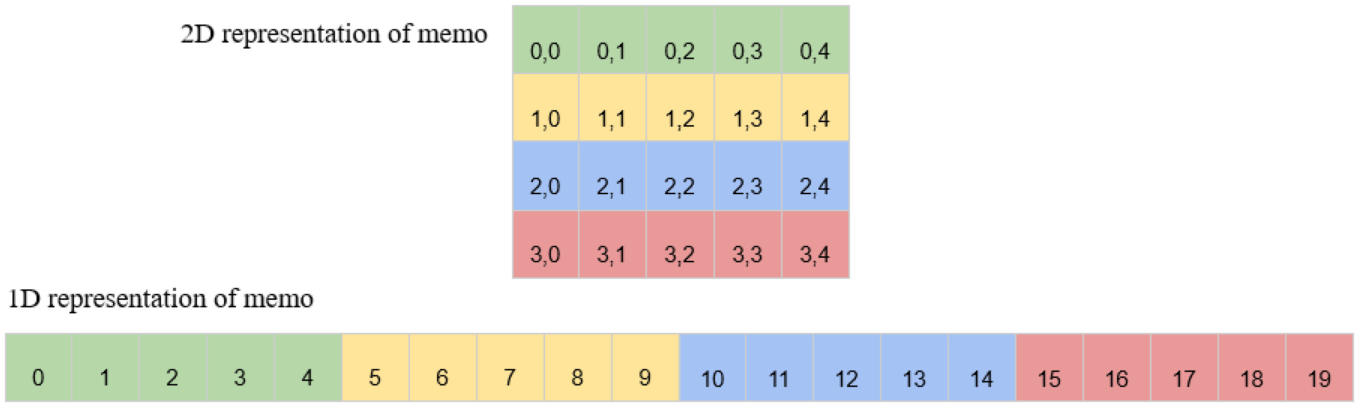


Fig. 6: Two different representations of memoization matrix

For computing memo A in parallel, my code invokes a kernel (named “compute_j_minus_k”) with ‘(ALPHABET_SIZE, 1, 1)‘ as the block dimension and ‘(1, 1, 1)‘ as the grid dimension; the kernel computes one “row” in memo A (using a equation (8) but with indexing calculated using equation (10)), where a “row” here means $m+1$ consecutive entries in memo A, and these entries would be in the same row if memo A were 2D. By doing so, I generate ALPHABET_SIZE (i.e. 26) threads in total (26 threads per block $\times$ 1 block per grid) matching the number of rows in memo A, and each thread executes the kernel code to compute a different row in memo A in parallel. See figure 7 for an illustration of what it means to have each thread compute one row (note that only 4 threads corresponding to 4 rows are shown here for simplicity). Essentially, all rows in memo A would be computed at the same time (given enough resources on the GPU), successfully achieving the same idea as line 3 to line 7 of Algorithm 3. As a side note, by only generating 1 block here, I made the assumption that ALPHABET_SIZE is less than 1024 or the maximum number of threads per block, which was true for my term project.

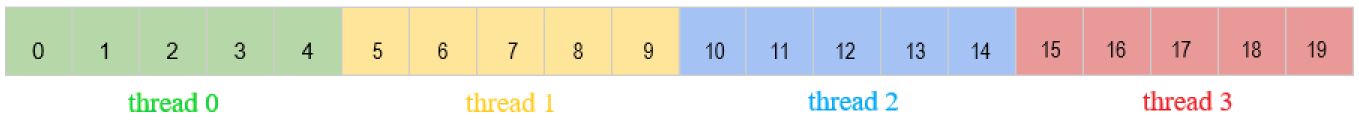


Fig. 7: Example of how memoization matrix A is parallelized in CUDA

The reason why I used 1D block and grid here is it is the most intuitive way to determine which row a thread corresponds to while in the kernel given its threadIdx.x. Specifically, since I used the same number of threads as the number of rows in memo A and each thread corresponds to exactly one row, it is easy to see that threadIdx.x just equals to row index (e.g. thread 0 is row 0, thread 1 is row 1, etc.). Then, each thread finds the start index of this row in the 1D array representation of memo A using equation (10):

$$index = rowIndex \times numColumns + columnIndex = threadIdx.x \times (m + 1)$$

For instance, in Figure 7, the number of columns (i.e. number of entries per row) is 5, so for thread 2, the start index would be $2 \times 5 = 10$. Knowing the start index and the number of entries which the thread has to compute (i.e. $m + 1$), each thread can successfully compute all of the entries it is responsible for. Thus, after all threads finish executing the kernel and return back to the CPU, all "rows" in memo A would have been computed.

Then, for computing the memo M in parallel, my code repeatedly invokes a kernel (named "compute_scs") with '($min\{1024, m+1\}$, 1, 1)' as the block dimension and '($\lceil (m+1)/1024 \rceil$, 1, 1)' as grid dimension in a loop for $n+1$ times (corresponding to the outer loop on line 8 of Algorithm 3); the kernel here only computes one entry in memo M (using equation (8), corresponding to line 10 of Algorithm 3). In other words, for each iteration, my code uses a total of $m+1$ threads (which may spread across multiple blocks when $m + 1 > 1024$) matching the number of entries per row in memo M, with each thread computing one entry on the same row in parallel. Then, after all threads finish executing the kernel in the current iteration, the CPU goes to the next iteration to invoke the kernel with the same block and grid dimension, such that each thread in the next iteration would compute one entry on the next row, so on and so forth. After $n+1$ iterations, all of the rows would have been computed. Essentially, all entries per row are computed at the same time (given enough resources on the GPU to execute all blocks in parallel), and the rows are executed serially (row i needs to wait for all entries in row $i - 1$ have been computed). This successfully achieves the same idea as line 8 to 12 in Algorithm 3. See figure 8 for an overview of how memo M is parallelized.

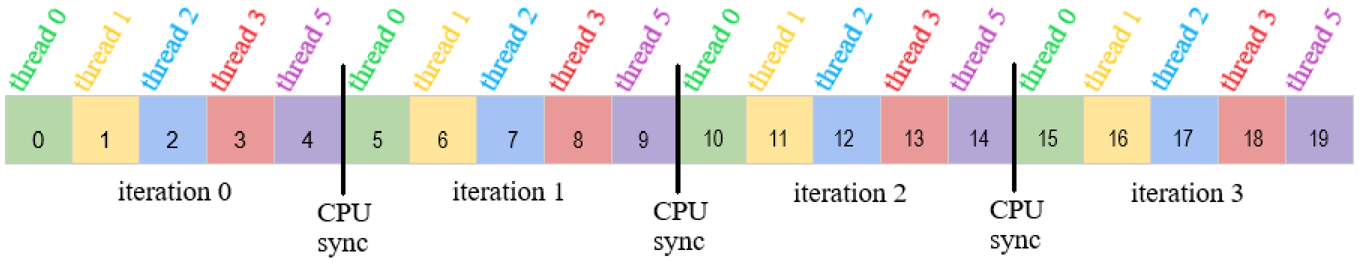


Fig. 8: Example of how memoization matrix M is parallelized in CUDA

Before moving on to discussing the kernel, I would like to elaborate on why I had to use $n + 1$ repeated kernel calls in the CPU rather than doing the for loop directly in each kernel and use '__syncthreads()' at the end of each iteration. Since I needed to guarantee that all entries on row $i - 1$ have been computed before computing any entry on row i, and the entries on the same row may be computed by threads on different blocks, I could not use '__syncthreads()' because it only synchronizes all threads in *one block* (a block-level synchronization primitive) rather than synchronizing all threads in *the grid* (i.e. synchronizing all blocks, or grid-level synchronization). The only (simple) way to achieve the latter synchronization is to use CPU synchronization, i.e. putting the loop in the CPU to do repeated kernel calls, where each kernel is only responsible for computing one entry in one row.

As for the kernel invocation, I still used 1D block and grid here because, again, it is the most intuitive way to find the index the thread corresponds to in the kernel. Although the block and grid dimensions seem a little more complicated with the min and ceiling functions, all this is doing is just to ensure that there are a total of $m + 1$ threads when considering all of the blocks. Since $m + 1$ can be greater than 1024 or the maximum number of threads per block, the combination of the number of the threads in all of the blocks specified must be greater than or equal to $m + 1$, hence the $\lceil (m + 1)/1024 \rceil$. By taking the $min\{1024, m + 1\}$, I can account for cases when $m + 1$ is less than 1024, so I would not need to use that many threads for the kernel.

Finally, the kernel code, as mentioned above, only computes one entry in memo M. Upon entering the kernel, each thread must find the index of the entry on the current row it corresponds to. It uses the following formula to first calculate the column index (index j) it corresponds to if memo M were 2D:

$$columnIdx(j) = threadIdx.x + blockIdx.x \times blockDim.x \tag{11}$$

If the column index j is greater than m, it means that the index which this thread corresponds to is out of bounds, which can happen because the ceiling function when defining the grid dimension may generate more threads in the last block than needed. In this case, the thread doing this kernel simply returns. Then, using the row index i passed into the kernel as a parameter, the thread calculates the actual index in memo M it corresponds to using equation (10).

$$index = i \times (m + 1) + j$$

The remainder of the kernel code simply follows equation (9) to compute $M[index]$ and write to it. Whenever the code needs to access an entry in A (i.e. $A[c][j]$) or another entry in M (i.e. either $M[i-1][j-k-1]$ or $M[i-1][j]$), it simply uses equation (10). This concludes the discussion of the kernel code.

To further optimize the speed and parallelism of my code. I attempted the following three strategies.

First, using the same idea as from the CPU/OpenMP implementation, I tried to reduce the number of branches in the parallel regions, i.e. the kernels. Specifically, I was able to remove the if condition checking $(j = 0)$ in the kernel that is computing memo A (i.e. corresponding to the first case in equation (8)) by setting all entries in A to 0 upon initialization using 'cudaMemset'; by doing so, I no longer need to set these entries meeting this condition (i.e. entries on the leftmost column) to 0 since they were already 0, thus, making this change successfully reduced the number of branches from 3 to 2 in this kernel. Then, I was able to remove the if condition checking $(i = 0)$ in the second kernel that is computing memo M (i.e. corresponding to Base Case 1 in equation (9)) by doing a separate kernel call in the CPU that only computes the 0th row $(i = 0)$ in M, before the loop of kernel calls which now only computes row 1 to row n. This new kernel is invoked with the same block and grid dimensions as the regular kernel, i.e., still create $m + 1$ threads (matching the m+1 entries in this row). By making this change, I successfully reduced the number of branches from 4 to 3 in the regular kernel. Lastly, I tried to remove the remaining 3 branches in this kernel (i.e. if $j = 0$, if $A[c][j] = 0$, and else), again using a similar strategy as mentioned in the CPU/OpenMP section; however, making those changes did not make any impact on timing results at all, even for large inputs. Thus, I would not like to elaborate on them any further here.

Second, I tried to reduce the number of conversions from row index and column index to the index for 1D memo M using equation (10). For instance, computing $M[i][j]$ requires $M[i-1][j]$, so originally in my code, I would calculate both $i \times (m + 1) + j$ and $(i - 1) \times (m + 1) + j$ respectively. Rather than doing the multiplication twice, I saved the value of the former calculation (i.e. the index corresponding to $[i][j]$), and made the second calculation to use the index for $[i][j] - (m+1)$. However, changes like these (where I tried to minimize the multiplication) actually made my code slower (e.g. by over 15 ms for a very large input 60000 by 60000)! I am not sure why this was the case but I switched back to my original way of converting the row and column index to the 1D index each time the code indexes into memo M.

Third, I also tried to use per-block shared memory in the kernel that is computing memo M, by having each thread first read the entry in M it corresponds to, i.e. $[i][j]$, and the entry above it, i.e. $[i-1][j]$, into a 2D __shared__ array. Then, the remaining computations simply read from this shared array whenever it needs to read from M. However, I soon realized that this would not work, because by definition of equation (9), computing $M[i][j]$ also requires $M[i-1][j-k-1]$, which may not belong to the current block of threads of the GPU, and creating ghost cells for them in the shared array may be very expensive and complex. Moreover, I realized that without saving this value at $M[i-1][j-k-1]$, this shared memory then serves no purpose because each thread does not have any overlapping accesses to entries in M other than in this case. Thus, I did not try to further explore this idea.

Lastly, I tried to experiment with different block and grid dimensions, specifically by modifying the default number of threads per block I use for the kernel that is computing memo M. Previously, I mentioned that my block dimension is '$min\{1024, m+1\}$', because I wanted to use the maximum number of threads per block (1024). But, based on my experience with CUDA from homework 4, I knew that utilizing all threads per block might not always result in the best performance. So I tested my program with 512 and 256 as the default number of threads per block. The result unfortunately showed no significant difference ($<$ 4 ms for very large inputs) between the execution time using different default number of threads per block. Thus, I still ended up using 1024.

This concludes everything I did for implementing and optimizing the parallel portions of the row-wise independent parallel algorithm on the GPU. For the actual code, please see "parallel_cuda_scs.cu".

V. Testing

A. *Ensuring Correctness*

As I was implementing the parallel algorithms, specifically, the anti-diagonal parallel algorithm with OpenMP, and the row-wise independent algorithm with OpenMP and CUDA, I had to ensure that they produced the correct results. I tested all three parallel programs using the same method. First, I created a set of test cases consisting of different input strings of various lengths and contents: some being short and simple with only 5-10 characters (for debugging mainly), some being long (e.g. with about 100 characters) and randomly generated (discussed later), and some having specific edge-case contents (e.g. one input string is empty, or both input strings have the same characters, etc.). Then, for each test, I ran my serial DP program in debug mode, by printing out everything (all values) in the final, computed memoization matrix M and saving it in an output file. After I finish implementing a parallel algorithm, I would run my parallel program against the set of test cases while also printing out the final memo M (produced by my parallel program), and do a diff check between the output of my parallel program with the saved output (of my serial program). There were a total of 15 tests, and I considered my parallel program to be correct if for all 15 tests, there was no difference in the output files.

For the anti-diagonal algorithm, I initially struggled with the indexing of the outer and inner for loops for looping through the anti-diagonal in-order, being weird and confusing. But the parallel (OpenMP) aspects did not really cause any trouble after I figured out how to do the indexing. For the row-wise independent algorithm in OpenMP, other than spending considerable time to come up with this algorithm myself, I struggled with many off-by-one errors in indexing into memoization matrix A and M vs. indexing into the input strings. Specifically, the input strings are of size n and m respectively and use 0-based indexing with $X[0]$ being the first character in string X; on the other hand, the memoization matrix M are of size $n+1$ by $m+1$, where the $+1$ is for the "ghost cells" for the two base cases stated in equation (9) so $M[1][j]$ would correspond to the first character in X. This means that for every equation previously where I used i and j to index into M, I needed to use $i-1$ and $j-1$ to index into string X and Y respectively in the code. Because I initially did not realize this off-by-one indexing issue, I thought that my equations themselves were wrong in the first place so I had to go back and analyze all of my diagrams and equations I came up with. Eventually, I was able to finally realize this stupid mistake. Fixing this bug made me pass all of the tests. Finally, implementing the row-wise independent algorithm in CUDA presented some difficulties in breaking down the memoization matrix into blocks and threads; after figuring that out, I was able to implement it and successfully pass all tests.

B. Creating and Scaling Input Strings

After ensuring that my parallel programs were indeed correct, I had to create input strings for the actual testing, which would be used for later time and efficiency analysis. Specifically, I needed a way to generate very large strings (i.e. the length of the input strings is very large). To do so, I wrote a program that can generate random strings of arbitrary length over the 26 lower-case letters in the English alphabet. I used 26 English letters as the alphabet of the input strings because it relatively accurately represents real-life English texts. Moreover, my current row-wise independent parallel implementation makes the assumption that the alphabet consists of just these 26 letters, so adding more letters to the input alphabet would result in a newly implemented array C and hash function, which despite not difficult to do, would add in additional serial computational overhead (specifically the hash function) that are not important for the parallelization aspects of the algorithm.

To generate a string of size n, the program repeatedly picks a character from the alphabet randomly (using the 'rand()' function provided in C's standard library) and appends it to the current string for n times. The final string generated was directly printed as output, and I simply redirected it to a file to save the generated string.

For the test, I initially used the following input sizes, i.e. length of the input strings: 2000, 4000, 6000, 8000, 10000, 20000, 40000, 60000, 80000, 100000; for each input size, I generated two input strings, string X and string Y, and saved them into a file. I just want to point out that both input strings for each input size are of the same length, despite the program supporting any two strings of arbitrary length; the reason for using the same length is that it allows simpler mathematical analysis of speedup and efficiency later on. I would also like to reiterate that because the SCS problem I was trying to solve only takes in 2 input strings, the input size for my problem then only considers the length of each input string, rather than the number of input strings, since that would be a different problem to solve (multiple or generalized SCS), perhaps in a future study.

After I generated the strings for each input size, I modified all my programs (serial, anti-diagonal OMP, row-wise independent OMP, row-wise independent CUDA) to read the input strings from a file using 'std::ifstream' and 'std::getline()' (where previously the input strings were just hardcoded inline), with the filename being passed into the program as an argument on the command line.

Then, I ran my programs with all 10 input sizes on the Great Lakes supercomputers (the specific submission script files are discussed later) and realized that all of my programs resulted in the correct output, i.e. the same final SCS length, for all input sizes up to and including 40000. Specifically, at 60000, the 'cudaMemcpy' in my CUDA program began to fail, as it could not copy back the final SCS length from the last cell in memo M on the GPU back to the CPU. With hours of debugging, I realized that I was (originally) using type 'int' for all of my indexings into the 1D memo M, and since the input size was 60000, the memo M had size 60001×60001, which was larger than the maximum that an int type variable could hold, so I had to change all indices for memo M to be of type 'long long', which fixed the error. But then, at 80000, the 'cudaMalloc' function actually began to fail as it could not allocated sizeof(int) $\times$ 80000 $\times$ 80000 amount of memory (at least using the submission script file I specified, which was the same file I copied over from homework 4). Thus, ultimately, I had to only use input sizes up to and including 60000, a total of 8 different input sizes, which despite being 2 input sizes less than before, could still provide sufficient insights on the time and efficiency of my parallel programs.

C. Scaling Computational Resources and Timing

For my two OpenMP programs, other than scaling the input size, I could also scale the amount of computational resources by specifying the number of threads used by the parallel regions. I did so by defining the environment variable in my submission script file using ‘export OMP_NUM_THREADS=p‘, where p indicates the number of threads I would like to use for regions in my code enclosed by ‘#pragma omp parallel‘ (see Algorithm 1 and 3, and the Implementation section on CPU/OpenMP). I ran my programs with the number of threads being 8, 16, and 32 respectively. This would reveal more insights on the efficiency and parallelism of my programs. Thus, given both the scaling of input sizes and scaling of computational resources, I wrote the script file for submitting my two OpenMP programs (anti-diagonal and row-wise independent) as a job to Great Lakes slurms, specifying the following options and content.

```
#SBATCH –job-name=term_project
#SBATCH –nodes=1
#SBATCH –ntasks-per-node=36
#SBATCH –exclusive
#SBATCH –time=00:05:00
#SBATCH –account=eecs587f23_class
#SBATCH –partition=standard
export OMP_NUM_THREADS=8 # then modify this to be 16 and then 32
./$program input/input-2000.txt > output-$program-2000-$OMP_NUM_THREADS.txt
...
./$program input/input-60000.txt > output-$program-60000-$OMP_NUM_THREADS.txt
```

Then, for my CUDA program, I could only scale input size because of the GPU’s SIMT nature and how I implemented my code: The number of threads used is always the same as the number of entries per row in M, i.e. length of input string $Y + 1$, meaning that the number of threads grows with respect to the input size (specifically length of string Y), so there is no need to further scale the number of threads manually. Thus, I wrote the script file for submitting my CUDA program as a job to Great Lakes slurms with the following specifications and content.

```
#SBATCH –job-name=term_project
#SBATCH –mail-type=BEGIN,END
#SBATCH –nodes=1
#SBATCH –gres=gpu:1
#SBATCH –time=00:05:00
#SBATCH –account=eecs587f23_class
#SBATCH –partition=gpu
#SBATCH –mem-per-gpu=16GB
./parallel_cuda_scs input/input-2000.txt > output-parallel-cuda-scs-2000.txt
...
./parallel_cuda_scs input/input-60000.txt > output-parallel-cuda-scs-60000.txt
```

For all my programs, I recorded a start immediately right after I allocated the memoization matrix M (and memoization matrix A for the row-wise independent parallel algorithm), before starting to compute any value in the memo, and I recorded an end time immediately after I finished computing all values in the memoization matrix M (i.e. computing the final length of the SCS). As such, the timing results would accurately represent the time it took to compute the SCS, ignoring the overhead of memory allocation, copying, etc. Specifically, I used ‘std::chrono‘ to record the time for my serial program; ‘omp_get_wtime()‘ to record the time for my parallel OMP programs (anti-diagonal and row-wise independent); and ‘cudaEvent‘ to record the time for my parallel CUDA program (row-wise independent).

VI. Results and Analysis

All programs were run on Great Lakes slurms using their respective script files in the morning time of the day. To account for the variations of time between runs, all programs were run multiple times on different days and the average between all runs was taken to obtain a more accurate timing result.

This section first shows a table (I) containing the execution times (i.e. runtimes) of all of my implementations, including the serial DP algorithm, anti-diagonal parallel algorithm on the CPU (OMP), row-wise independent parallel algorithm on the CPU (OMP), and the row-wise independent parallel algorithm on the GPU (CUDA). The second table (II) contains the efficiency of my parallel programs excluding the GPU implementation, calculated by $SerTime(n)/(p \times ParTime(n,p))$, where n is the input size and p is the number of threads (used by the CPU/OMP). The third table (III) shows speedup, calculated by $SerTime(n)/ParTime(n,p)$.

Afterwards, I analyze the runtimes of my programs holistically (in the first subsection); then I specifically ananlyze how the time and efficiency/speedup of my parallel programs change with respect to scaling in terms of input size and number of threads (in 4 different subsections respectively).

TABLE I: Time

String Length	Serial Time (ms)	Parallel Time Anti-diagonal CPU 8 threads (ms)	Parallel Time Anti-diagonal CPU 16 threads (ms)	Parallel Time Anti-diagonal CPU 32 threads (ms)	Parallel Time Row-wise Independent CPU 8 threads (ms)	Parallel Time Row-wise Independent CPU 16 threads (ms)	Parallel Time Row-wise Independent CPU 32 threads (ms)	Parallel Time Row-wise Independent GPU (ms)
2000	21.0650	14.9655	21.5291	51.0949	15.1931	23.6336	52.2534	5.6098
4000	83.7208	35.3459	47.2821	84.2341	34.8422	56.3370	146.9931	11.1558
6000	186.8341	63.0685	70.9093	161.0058	55.2004	89.6952	241.4106	16.9883
8000	332.9428	96.4191	105.6943	203.9953	87.3840	138.0009	311.7736	21.9110
10000	524.3038	175.2451	189.4233	284.6570	111.8855	162.3052	440.7839	27.5591
20000	2100.6904	532.5543	454.5218	633.1145	425.1172	395.2775	931.3449	58.0116
40000	8398.2480	2473.1117	1544.3785	1346.8406	1335.6912	1273.7588	1959.3540	113.1901
60000	18845.5030	5678.9880	2888.7606	2714.6061	2872.7902	2471.7683	2937.4066	185.7801

TABLE II: Efficiency

String Length	Efficiency Anti-diagonal CPU 8 threads	Efficiency Anti-diagonal CPU 16 threads	Efficiency Anti-diagonal CPU 32 threads	Efficiency Row-wise Independent CPU 8 threads	Efficiency Row-wise Independent CPU 16 threads	Efficiency Row-wise Independent CPU 32 threads
2000	0.1759	0.0612	0.0129	0.1733	0.0557	0.0126
4000	0.2961	0.1107	0.0311	0.3004	0.0929	0.0178
6000	0.3703	0.1647	0.0363	0.4231	0.1302	0.0242
8000	0.4316	0.1969	0.0510	0.4763	0.1508	0.0334
10000	0.3740	0.1730	0.0576	0.5858	0.2019	0.0372
20000	0.4931	0.2889	0.1037	0.6177	0.3322	0.0705
40000	0.4245	0.3399	0.1949	0.7859	0.4121	0.1339
60000	0.4148	0.4077	0.2169	0.8200	0.4765	0.2005

TABLE III: Speedup

String Length	Speedup Anti-diagonal CPU 8 threads	Speedup Anti-diagonal CPU 16 threads	Speedup Anti-diagonal CPU 32 threads	Speedup Row-wise Independent CPU 8 threads	Speedup Row-wise Independent CPU 16 threads	Speedup Row-wise Independent CPU 32 threads	Speedup Row-wise Independent GPU
2000	1.4076	0.9784	0.4123	1.3865	0.8913	0.4031	3.755012
4000	2.3686	1.7707	0.9939	2.4029	1.4861	0.5696	7.504663
6000	2.9624	2.6348	1.1604	3.3847	2.0830	0.7739	10.9978
8000	3.4531	3.1501	1.6321	3.8101	2.4126	1.0679	15.19521
10000	2.9918	2.7679	1.8419	4.6861	3.2304	1.1895	19.02471
20000	3.9446	4.6218	3.3180	4.9414	5.3145	2.2555	36.21156
40000	3.3958	5.4379	6.2355	6.2876	6.5933	4.2862	74.19597
60000	3.3185	6.5237	6.9423	6.5600	7.6243	6.4157	101.4398

A. Overall Performance/Time

To get an overall sense of how my parallel programs performed against the serial DP program and against each other, I graphed the runtime of my programs (in milliseconds) with respect to input size (length of string X $\times$ length of string Y) using the data from Table I. For the anti-diagonal parallel algorithm and row-wise independent algorithm implemented in OMP (CPU), I picked the fastest time out of the three different runtimes resulting from scaling the number of threads (i.e. 8, 16, 32). This imitated the idea of weak scaling, as I was scaling the computational resources (threads) and the input size together when performing the analysis, which provides a more accurate representation of how real-life programs are run and a better comparison with my GPU implementation (which also employs weak scaling by the definition of my implementation and GPU's SIMT nature). See figure 9 (excluding the serial time) and 10 (including the serial time) for an overall trend and comparison between all my programs' runtimes as the input size grows.

Looking at figure 9 first, we can easily see that the row-wise independent parallel algorithm implementation on the GPU/CUDA (blue line, circular data points) performs the fastest on all input sizes. Though initially only slightly faster than the runtime of the other parallel implementations when the input size is small, it performs way faster than the rest on large input sizes, illustrating that the runtime of the row-wise independent GPU program grows *very slowly* (much slower than the other implementations) with respect to input size.

Next, the row-wise independent parallel algorithm implementation on the CPU/OMP (green line, diamond data points) performs the second fastest. It is only slightly faster than the anti-diagonal parallel algorithm implementation (on the CPU/OMP)

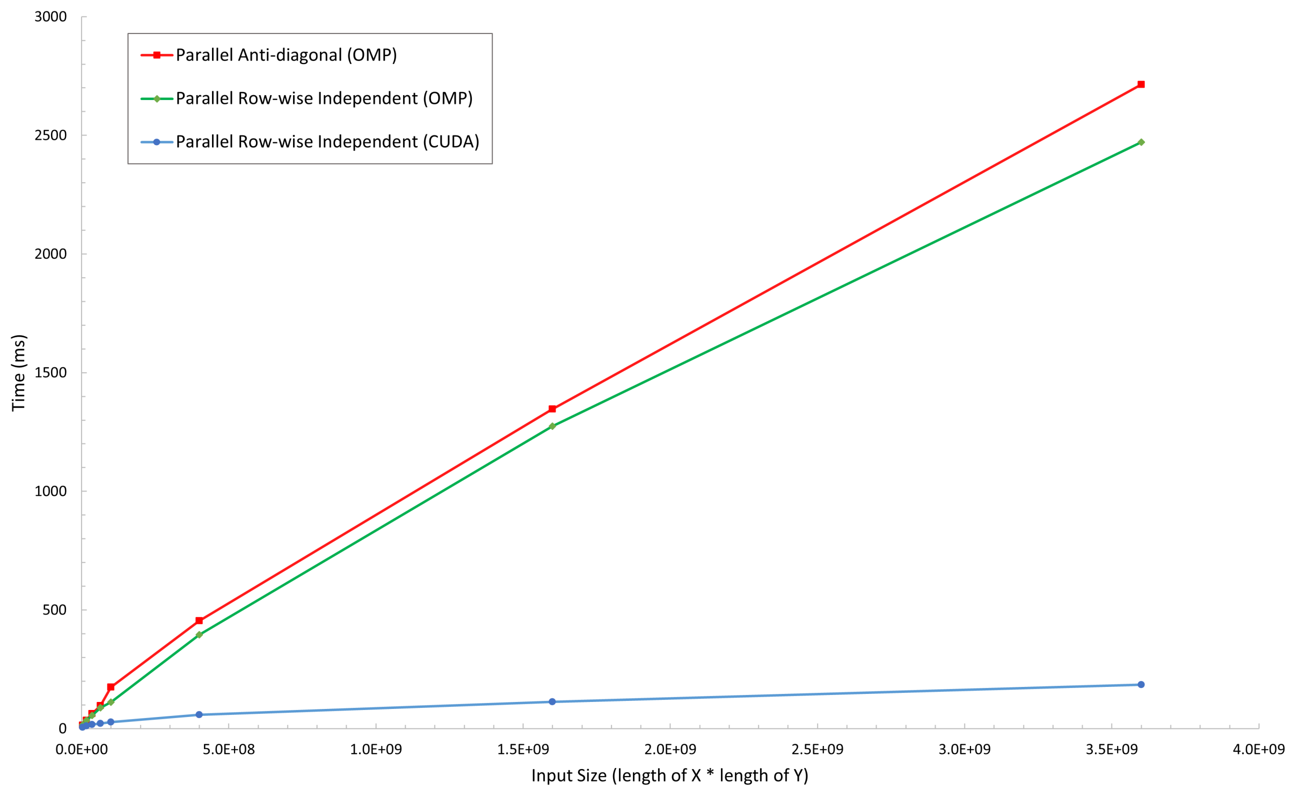


Fig. 9: Graph of Input Size vs. Execution Time for the Implemented Parallel Algorithms

when the input size is small, but the difference (the gap between the green and red line) becomes larger as the input size grows, indicating that the runtime of the row-wise independent program grows *slower* than that of the anti-diagonal program. Together, this shows that the row-wise independent algorithm performs better than the anti-diagonal algorithm, especially on large input sizes.

The anti-diagonal parallel algorithm implementation on the CPU/OMP (red line, square data points) performs the worst out of the three parallel programs (despite being still much faster than the serial DP program, as shown in figure 10). It is slightly slower than the row-wise independent parallel program on the CPU for small input sizes, but like mentioned in the previous paragraph, the difference between the runtimes of these two programs becomes larger as the input size grows, meaning that the runtime of the anti-diagonal program grows *faster* than that of the row-wise independent program with respect to input size. This implies that the anti-diagonal parallel algorithm may be better suited for small input sizes, while the row-wise independent algorithm is better for large input sizes.

Lastly, when adding the runtime of the serial DP algorithm implementation to the graph as shown in figure 10, we can see that it performs the slowest: Although the serial program performs about the same as the parallel programs on the CPU when the input size is small, as the input size increases, its runtime grows rapidly, much faster than the parallel programs, causing it to perform way slower than the rest when the input size is large.

Thus, overall, all three parallel algorithm implementations clearly perform much better than the serial DP algorithm implementation, especially when the input size is large. Out of the three parallel implementations, the row-wise independent algorithm on the GPU has the best performance, followed by the same algorithm on the CPU, followed by the anti-diagonal algorithm (on the CPU).

B. Effect of Scaling Input Size on Time

After getting the general sense of the programs' runtimes, I would like to discuss the effect of input size on time in more detail for each parallel program, using Table I as a reference. I mainly focus on analyzing and comparing the two parallel algorithms implemented on the CPU; towards the end of this section, I talk about the parallel implementation on the GPU.

Starting with the anti-diagonal algorithm and row-wise independent algorithm implemented on the CPU with a fixed number of threads = 8, we can see that initially, for string length = 2000, the runtime of the anti-diagonal program is actually slightly faster (by about 0.2 ms). But soon, starting at string length 4000, the row-wise independent program becomes faster and faster than the anti-diagonal program, as the input size increases (while fixing the number of threads = 8). This reinforces my previous observation from the graphs that the runtime of my row-wise independent program grows slower than the anti-diagonal program, as input size increases (for a fixed number of threads 8). This actually matches my analysis of the theoretical time complexity

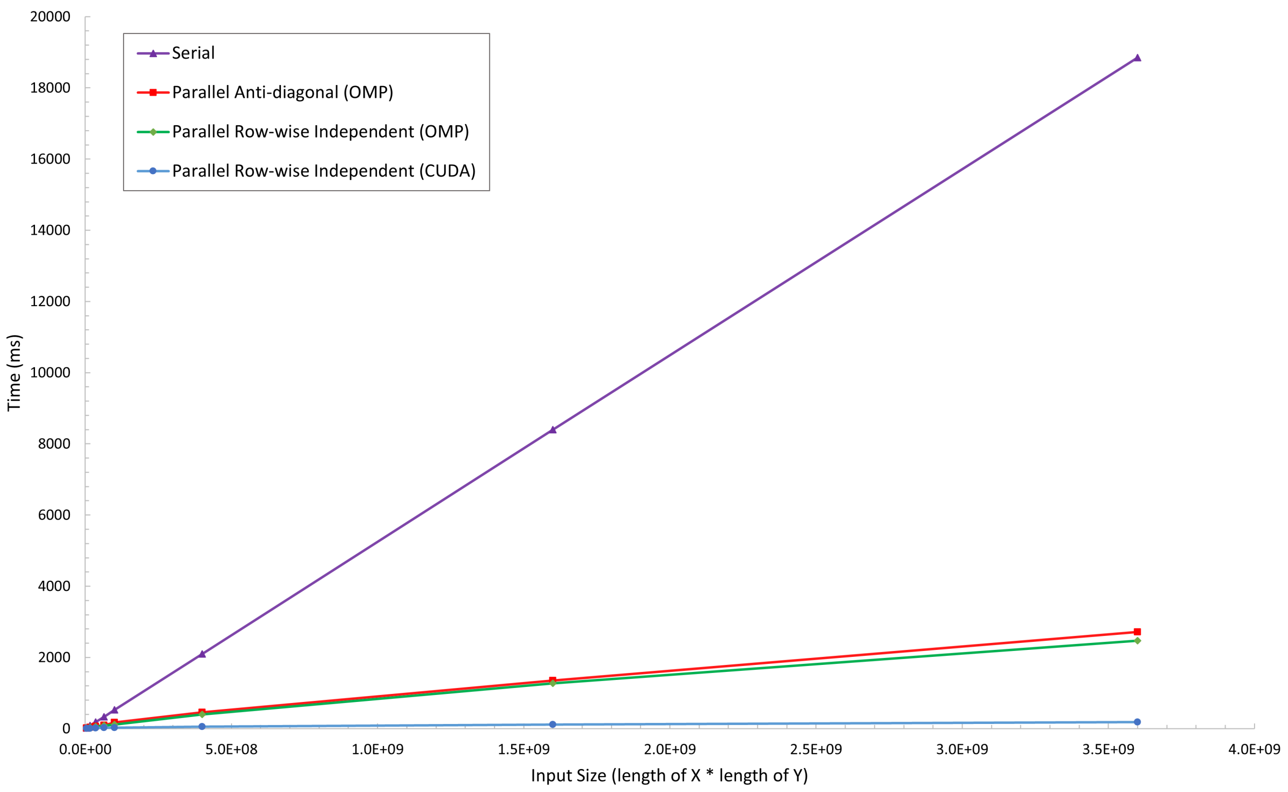


Fig. 10: Graph of Input Size vs. Execution Time for All Implemented Algorithms

of these two algorithms in Section 3: The time complexity of the row-wise independent algorithm, being $O(max\{n, m\})$ or simply $O(n)$ in this case (since length of string X and Y are the same for my testing), grows (slightly) slower as a function of input size compared to the time complexity of the anti-diagonal algorithm, being $O(n + m)$ or simply $O(2n)$. Thus, what we observed here makes sense (see figure 11 in the Appendix for an illustration of what was discussed in this paragraph).

Similarly, for a fixed number of threads 16, we can see that initially at string length = 2000, 4000, 6000, 8000, the runtime of the anti-diagonal program is actually faster than that of the row-wise independent program. But starting at string length = 10000, the row-wise independent program takes over, and becomes faster and faster than the anti-diagonal program, as input size grows larger. This further indicates that, when fixing the number of threads (e.g. 16 in this case), although the anti-diagonal algorithm may be faster initially (for smaller input sizes), the row-wise independent algorithm would eventually become faster (e.g. at string length = 10000 in this case) because it grows slower than the anti-diagonal with respect to the input size (see figure 12 in the Appendix for an illustration of what was discussed in this paragraph).

For a fixed number of threads 32, we can see that the anti-diagonal program is actually faster than the row-wise independent program for all 8 input sizes (i.e. string lengths) I used in testing. But, upon closer inspection at the difference between the runtimes of the two programs at each string length, we can see that the difference becomes smaller as the input size increases (see figure 13 in the Appendix for the illustration of this trend). Thus, this once again indicates that the runtime of the row-wise independent algorithm grows slower than that of the the anti-diagonal algorithm with respect to input size (while fixing the number of threads), meaning that eventually, at a large enough input (while fixing the number of threads), the row-wise independent algorithm would perform better than the anti-diagonal algorithm.

Furthermore, in reality, we do not exclusively scale input size while fixing the computational resources (i.e. number of threads in this case), we often scale both at the same time and use the optimal number of threads that would result in the best runtime. Putting this into context of my problem, at each input string length, we can pick the fastest runtime out of the 3 different runtimes corresponding to the 3 different number of threads used; this is exactly what I did in the previous section when I discussed the overall performance of my parallel programs (again, see Figure 9 and 10). To reiterate, when analyzing like this, the row-wise independent algorithm (implementation on the CPU) is faster than the anti-diagonal algorithm at every input size other than the very first one (where string length = 2000). This idea is elaborated in a later section where I discuss scaling in terms of number of threads.

Thus, based on the above observations, we can conclude that, when both implemented on the CPU, the row-wise independent algorithm has a better/faster time performance than the anti-diagonal algorithm as a function of input size.

Finally, the runtime of the row-wise independent algorithm implementation on the GPU is much faster and grows much slower with respect to the input size, when compared to its counterparts implemented on the CPU. This also makes sense because of GPU's SIMT nature: My GPU implementation uses as many threads as the size of the input string $Y + 1$, which is much larger than the 32 threads used in CPU, with each thread also performing a very simple/fast calculation. Thus, based on all results and observations discussed so far, we can conclude that my row-wise independent algorithm performs better (i.e. has better runtime) than my anti-diagonal algorithm, as input size increases, and it has the best performance/time when implemented on the GPU.

C. Effect of Scaling Input Size on Efficiency/Speedup

This section discusses the effect of input size on efficiency/speedup for each parallel program, using Table II and III as a reference.

First, focusing on the 2 parallel algorithm implementations on the CPU, both speedup and efficiency increase as the input size increases (for a fixed number of threads). Specifically, at a small input size (e.g. length of string = 2000), both parallel algorithm implementations on the CPU have a runtime that is only slightly faster (or even slower, when using more threads, discussed in a later section) than the serial DP algorithm implementation, thus yielding a very low speedup and efficiency initially. This makes sense since the serial algorithm is pretty fast for small input sizes to begin with, as it uses dynamic programming with a $O(n^2)$ time complexity. Moreover, the parallel algorithm implemented using OMP introduces extra threading overhead for creating and destroying the threads, synchronization overhead where some threads need to wait for other threads to finish executing, in addition to the unavoidable data dependency that must be executed serially (e.g. row i needs to wait for row $i-1$ to finish computing), all of which contribute to the slow runtime initially for small input sizes, hence the low speedup and efficiency. Thus, for a small input size (e.g. length of string $\leq$ 2000), it is not a good idea to use these two parallel algorithm implementations on the CPU due to their low speedup and efficiency.

As the input size becomes larger (while fixing the number of threads), however, both speedup and efficiency of the two parallel algorithm implementations on the CPU increase. This makes sense since as the input size grows, the runtime of the serial DP algorithm implementation grows much more rapidly than both parallel algorithm implementations, as discussed in the previous section. As a result, the serial program runs much slower for large input sizes, highlighting the faster runtime of the parallel algorithms. Moreover, as input size increases, the benefits of parallel computations in the two parallel algorithms begin to outweigh the overheads for creating the threads and for synchronization, resulting in faster runtime, hence higher speedup and efficiency. Thus, as input size increases (for a fixed number of threads), it is better to use parallel algorithm implementations as their speedup and efficiency increase.

The highest efficiency achieved by the row-wise independent program is 0.82, when string length = 60000 and number of threads = 8; the highest efficiency achieved for the anti-diagonal program is only 0.49, when string length = 20000 and number of threads = 8. Three additional observations can be made from this.

1) Efficiency, even at a large input size, is still not linear/perfect, which is reasonable because of the unavoidable data dependencies (e.g. row i depends on row $i - 1$ to be computed first).
2) Despite increasing as input size grows, efficiency would eventually reach a plateau/threshold where it can no longer increase, as demonstrated in the maximum efficiency of the anti-diagonal program being at 20000 rather than 60000 (at number of threads = 8), which also makes sense due to the data dependencies again.
3) The row-wise independent algorithm is more efficient than the anti-diagonal algorithm (this is discussed in more detail in the following paragraphs), as indicated by the efficiency of the latter already beginning to plateau where that of the former still increasing (for a fixed number of threads = 8).

Knowing the general pattern of speedup/efficiency as a function of input size for the 2 parallel algorithm implementations on the CPU, I would like to specifically compare the efficiency of these 2 parallel programs for each fixed number of threads.

For a fixed number of threads 8, we see (in Table II) that for all string lengths except 2000, the row-wise independent program has a higher efficiency than the anti-diagonal program; the difference between the two programs' efficiency becomes larger and larger as the input size increases. This is the same observation as the one about these two programs' runtimes discussed in the previous section, which is expected since the formula for calculating efficiency depends on the runtime, with a slower time resulting in a lower efficiency (while fixing the number of threads). Likewise, this same observation can be made for the fixed number of threads 16, as the row-wise independent program, despite having a lower efficiency initially for relatively smaller input sizes (string length $\leq$ 10000), becomes more and more efficient than the anti-diagonal program as input size increases. Finally, for the fixed number of threads 32, using the observations from the previous section, we know that the runtime of the row-wise independent program would eventually become faster than the anti-diagonal program at a large enough input size because the former grows much slower than the latter with respect to input size. This means that the efficiency of the row-wise independent program for the fixed number of threads 32 would eventually become higher than that of the anti-diagonal program as the input size increases. Thus, overall, we can conclude that the row-wise independent algorithm has a better efficiency than the anti-diagonal algorithm as input size increases (with a fixed number of threads).

For the row-wise independent algorithm implementation on the GPU, it does not really make sense to discuss efficiency, so I only included speedup in the results table (III). As the input size increases, the speedup of the row-wise independent algorithm on the GPU increases, like its CPU counterparts but much much faster: At string length 60000, it is able to achieve a speedup of 101.4, as compared to 7.6, the fastest speedup on the CPU, achieved by the row-wise independent algorithm, which is expected since the runtime of the algorithm on the GPU is much faster (as shown in Table I and discussed in the previous section). However, even with such a high speedup, it is still nowhere close to being a linear/perfect. This again is expected because

1) threads, despite using as many of them as the input string length + 1, are not guaranteed to run in parallel by the GPU;
2) synchronization overhead at the end of each iteration where threads that have already completed their kernels have to wait for other threads (likely in other blocks) to finish before moving on to the next iteration's kernel calls; and
3) unavoidable data dependency that must be executed serially, e.g. row i must be computed after the entire row $i-1$ in memo M,

all of which reduces parallelism of the implementation. However, it is still notable that the row-wise independent parallel algorithm implementation on the GPU has good speedup and better than that of the two parallel algorithm implementations on the CPU.

D. Effect of Scaling Number of Threads on Time

Understanding the effect of input size, I would like to direct the attention to the effect of scaling computational resources, i.e. the number of threads, on the runtime of the parallel programs on the CPU, using Table I as a reference. (Note that it does not make sense to discuss the runtime of the GPU program as a function of number of threads, because it by default increases the number of threads as input size increases, and I only used 1 GPU core for running/testing my program.)

First, for (relatively) smaller input sizes, i.e. string length $\leq$ 10000, adding more threads actually caused the runtime of both the anti-diagonal algorithm and the row-wise independent algorithm implementations on the CPU to be slower. Specifically, increasing the number of threads from 8 to 16 made the runtime slightly slower, whereas increasing from 16 to 32 made the runtime much slower (while fixing the input size). This is somewhat expected since adding more threads introduces extra overhead for creating/destroying these threads and also for synchronizing these threads before moving on to the next iteration; these additional overheads outweighed the performance benefits gained from doing the computations in parallel, leading to a slower runtime. In other words, the input size is not "large enough" to actually benefit from the utlization of that many threads. Thus, we can conclude that for smaller input sizes (e.g. string length $\leq$ 10000), it is better for both the anti-diagonal algorithm and the row-wise independent algorithm to use less threads (e.g. 8) as adding more threads actually detriments the runtime due to extra overheads.

However, looking at Figures 14 to 18 in the Appendix (graphs showing runtime as a function of number of threads at different fixed input sizes, using data from Table I), we can see that as the input size (i.e. string length) changes from 2000 to 10000, the runtime growth due to the increase in the number of threads becomes less and less steep (for both programs on the CPU). Eventually, at string length = 20000 (see Figure 19), the runtime of both programs using 16 threads becomes faster than their respective runtime using 8 threads. This indicates that at this input size, the performance benefits gained from doing the computations in parallel using 16 threads finally outweigh their overheads (discussed in the previous paragraph), resulting in the runtime of using 16 threads being faster than the runtime of using just 8 threads. Moreover, as string length becomes even larger, e.g. at 40000 and 60000 (see Figure 20, 21), this decrease in runtime (of both programs) caused by increasing the number of threads from 8 to 16 becomes more and more steep, indicating that the performance benefits gained from the parallel computation using more threads becomes more salient and more clearly outweighing their overheads.

Furthermore, at string length = 40000 (and also 60000), the runtime of the anti-diagonal program with 32 threads becomes faster than that of 16 threads. Although this is not the case for the row-wise independent program, we can still clearly see that the increase in its runtime from 16 threads to 32 threads becomes less and less steep as input size becomes larger, meaning that eventually, with a large enough input size, its runtime with 32 threads would become faster than that of 16 threads. This observation implies that the runtime of the anti-diagonal algorithm implementation benefits from using more threads at a relatively smaller input size compared to the row-wise independent algorithm implementation. In other words, the row-wise independent algorithm requires a (much) larger input size in order for its runtime to decrease by increasing the number of threads (e.g. from 16 to 32), whereas the anti-diagonal algorithm requires a relatively smaller input size for the performance benefit to show when scaling the number of threads.

Thus, based on these observations, we can make the following 2 conclusions.

1) For both algorithms (on the CPU), it is difficult to achieve any reduction in runtime from increasing the number of threads at a (fixed) relatively small input size; only at a (fixed) large input size, increasing the number of threads would result in a decrease in runtime.
2) It requires the anti-diagonal algorithm a relatively smaller input size for its runtime to decrease with respect to the number of threads, whereas the row-wise independent algorithm requires a much larger input size.

As a side note about the second conclusion, although at these input sizes where the runtime of the anti-diagonal program becomes faster as the number of threads increases, its fastest time achieved with a higher number of threads is still slower than the runtime of the row-wise independent program with a smaller number of threads. Specifically, at input string length 40000 and 60000 in my data (see Table I, or Figure 20 and 21), the runtime of the anti-diagonal algorithm with 32 threads is faster than itself with 16 threads (and 8 threads), but it is still slower than the runtime of the row-wise independent algorithm using 16 threads. This means that the row-wise independent algorithm implementation on the CPU still has a better performance than the anti-diagonal algorithm holistically.

E. Effect of Scaling Number of Threads on Efficiency/Speedup

In the final section, I would like to discuss the effect of scaling the number of threads (when fixing input size) on the efficiency of the two parallel algorithm implementations on the CPU, using Table II.

We can see that for all input sizes (i.e. string lengths), the efficiency of the two algorithm implementations actually decreases as the number of threads increases. This is reasonable and we can explain this by breaking it down into two cases.

First, for those relatively small input sizes, i.e. string length ≤ 10000, the runtime of the two algorithms actually increases as the number of threads increases (as discussed in the previous section), which leads to a decrease in efficiency by definition of the efficiency formula.

Second, for string lengths > 10000, although the runtime of the two algorithms begins to decrease as the number of threads increases, this is merely able to “slow down” the decrease in efficiency, rather than completely eliminating the reduction in efficiency with respect to the number of threads. This is because for a fixed input size, the decrease in efficiency as the number of threads increases is inevitable since there would always be a part of the program/algorithm that must be executed serially; for instance, the data dependencies between rows in my row-wise independent algorithm, where row i must wait for row $i-1$ to be computed, cannot be eliminated. Thus, as more threads are added but the input size is fixed, the runtime will not decrease significantly due to the serial parts of the algorithm, causing the efficiency to continually decrease. This analysis method (scaling computational resources but fixing the input size) by definition is just strong scaling, and this phenomenon is explained by Amdahl’s law.

The undesirably low efficiency as the number of threads increases reinforces the difficulty in parallelizing DP algorithms, as these inherent data dependencies due to their optimal substructure cannot be completely eliminated.

VII. Conclusion

All in all, all of my parallel algorithm implementations were successful, with very fast timing results compared to the serial DP implementation and reasonable (albeit far from linear/perfect) speedup/efficiency, especially for a large input size. Moreover, the row-wise independent algorithm have shown to be experimentally faster and more efficient than the anti-diagonal algorithm, especially for a large input size, when both implemented on the CPU. Using this knowledge combined with the GPU implementation of the row-wise independent algorithm having the fastest time out of all implementations, it is safe to say that currently, the best way to solve the SCS problem (with two input strings) is the row-wise independent algorithm implemented on the GPU.

In the future, it would be interesting to try to adapt and implement the algorithms to distributed memory using MPI, as in my current project, all of my implementations are on shared memory. It might also be useful to explore how (or whether it is even possible) to parallelize the process of finding the shortest common supersequence itself given a memoization matrix M that contains the length of the SCS for each subproblem. Furthermore, the row-wise independent algorithm I proposed in this project for solving the SCS problem with two input strings might be a basis for designing a solution that solves the generalized/multiple SCS problem.

Acknowledgment

Thank you Professor Stout and the EECS 587 GSIs for a great semester. Happy holidays.

References

[1] G. Nicosia and G. Oriolo, ”Solving the Shortest Common Supersequence Problem,” in B. Fleischmann, R. Lasch, U. Derigs, W. Domschke, and U. Rieder (Eds.), Operations Research Proceedings, vol. 2000. Berlin, Heidelberg: Springer, 2001, pp. 111-116. doi: 10.1007/978-3-642-56656-1_13.

[2] J. Ziv and A. Lempel, ”A universal algorithm for sequential data compression,” in IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, May 1977, doi: 10.1109/TIT.1977.1055714.

[3] S. Chaudhuri and B. Nicolas, ”Method and apparatus for generating statistics on query expressions for optimization,” U.S. Patent 7,330,848, Feb. 12, 2008.

[4] S. Rahmann, ”The shortest common supersequence problem in a microarray production setting,” Bioinformatics, vol. 19, no. Suppl 2, pp. ii156-ii161, Oct. 2003. doi: 10.1093/bioinformatics/btg1073.

[5] D. Maier, ”The Complexity of Some Problems on Subsequences and Supersequences,” J. ACM, vol. 25, no. 2, pp. 322-336, Apr. 1978. doi: 10.1145/322063.322075.

[6] V. Timkovsky, ”Complexity of common subsequence and supersequence problems and related problems,” Cybernetics and Systems Analysis, vol. 25, no. 4, pp. 565-580, 1989. doi: 10.1007/BF01075212.

[7] Z. Xia, Y. Cui, A. Zhang, T. Tang, L. Peng, C. Huang, C. Yang, and X. Liao, ”A Review of Parallel Implementations for the Smith-Waterman Algorithm,” Interdiscip Sci, vol. 14, no. 1, pp. 1-14, Mar. 2022. doi: 10.1007/s12539-021-00473-0.

Appendix

A. Graphs for the Effect of Scaling Input Size on Time

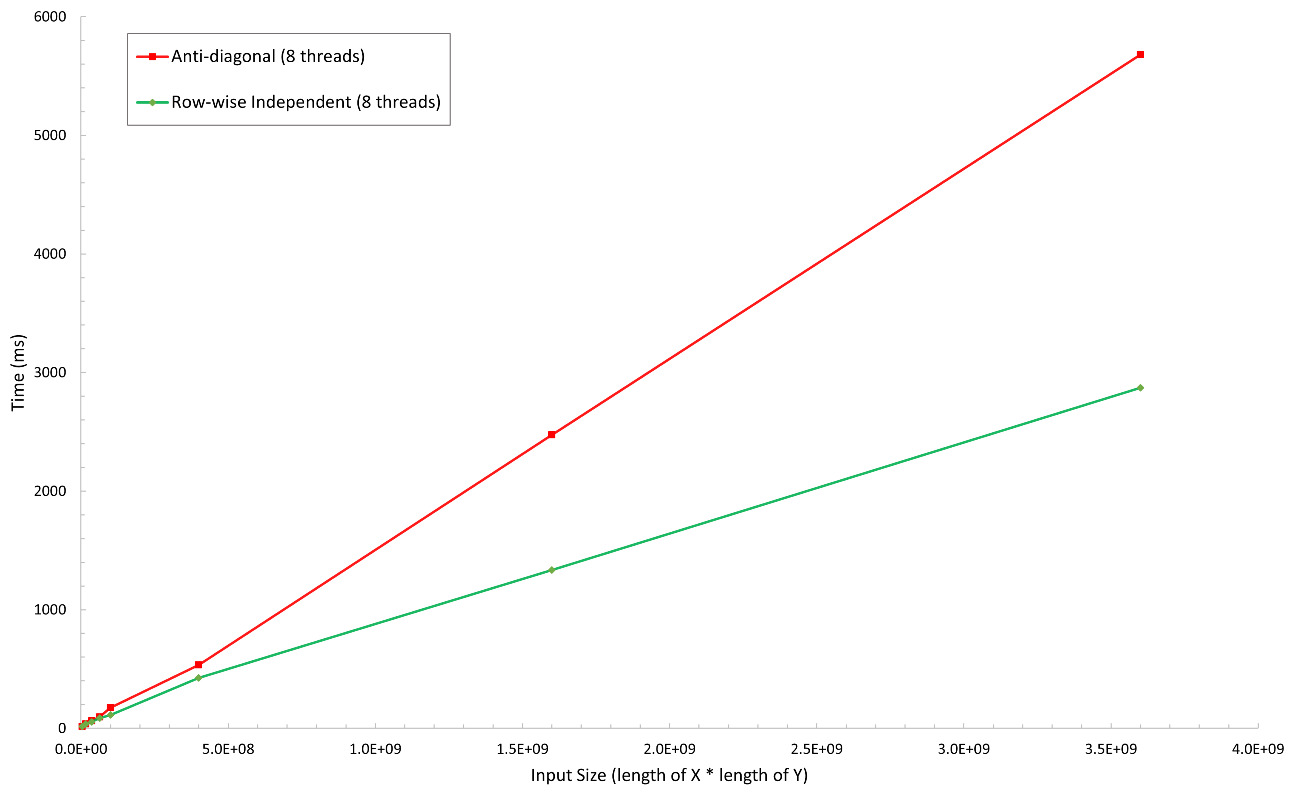


Fig. 11: Graph of Input Size vs. Execution Time for a Fixed Number of Threads 8 in OMP/CPU

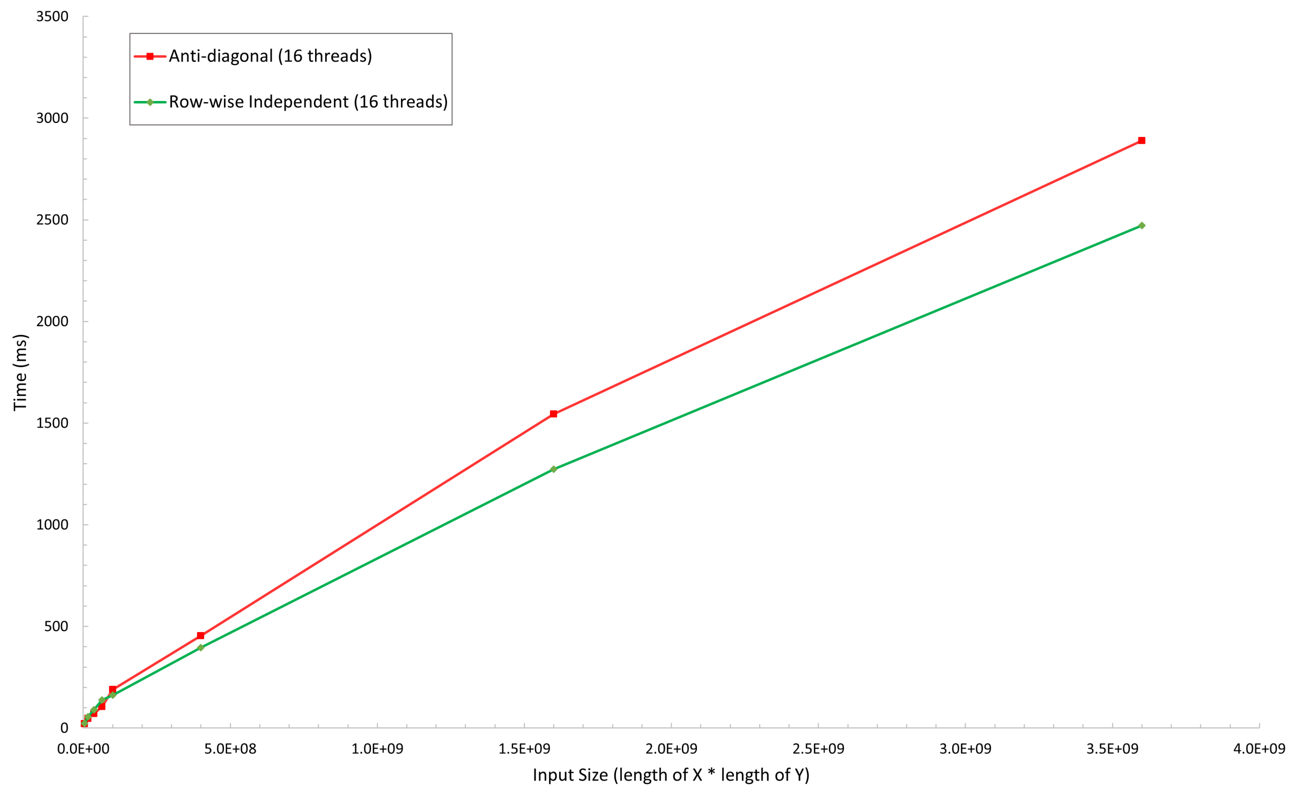


Fig. 12: Graph of Input Size vs. Execution Time for a Fixed Number of Threads 16 in OMP/CPU

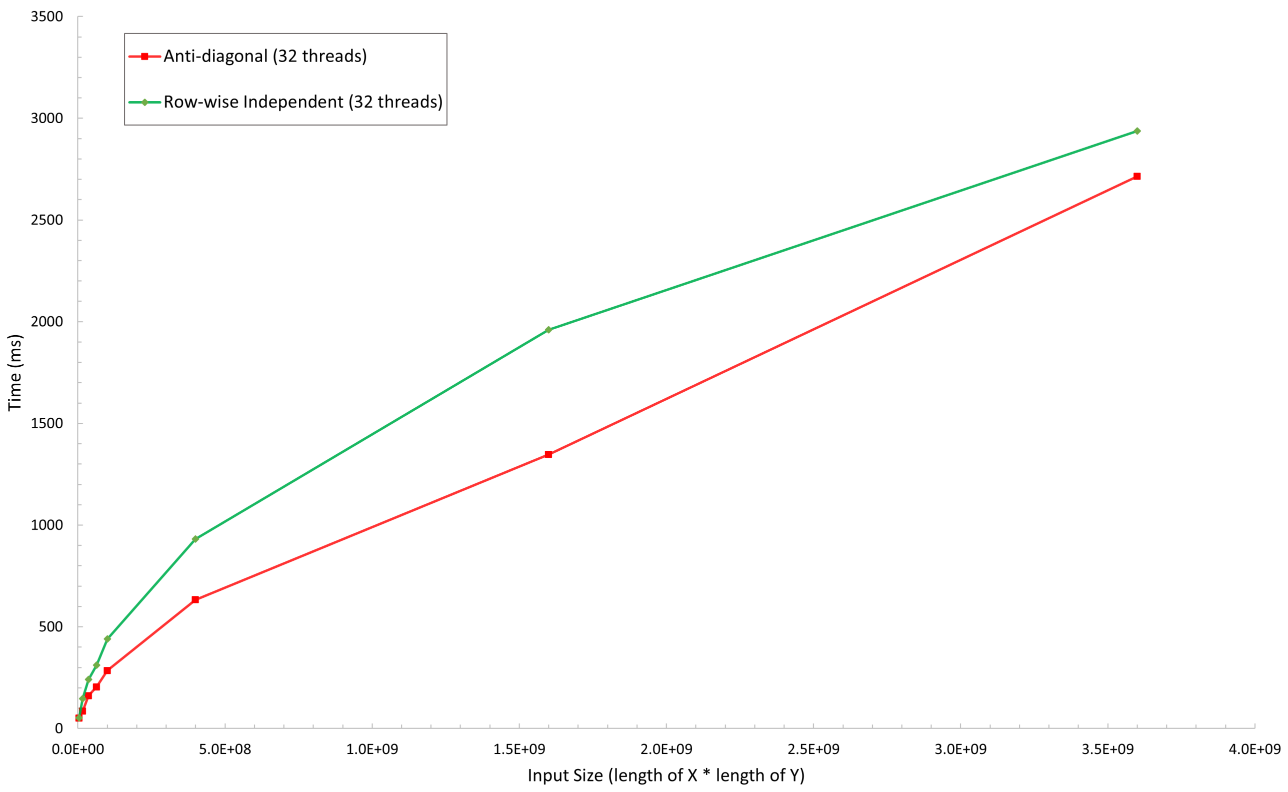


Fig. 13: Graph of Input Size vs. Execution Time for a Fixed Number of Threads 32 in OMP/CPU

B. *Graphs for the Effect of Scaling Number of Threads on Time*

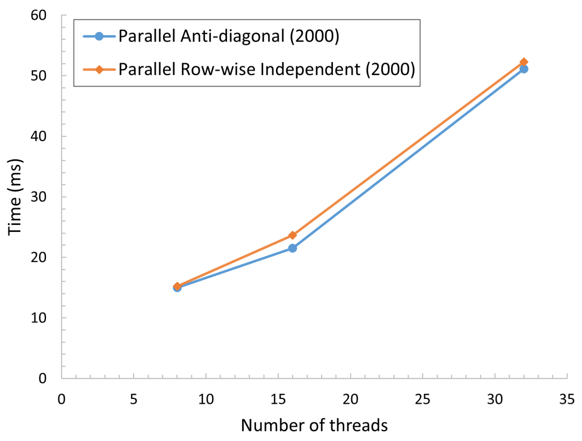


Fig. 14: Graph of Number of Threads vs. Execution Time for a Fixed String Size 2000 in OMP/CPU

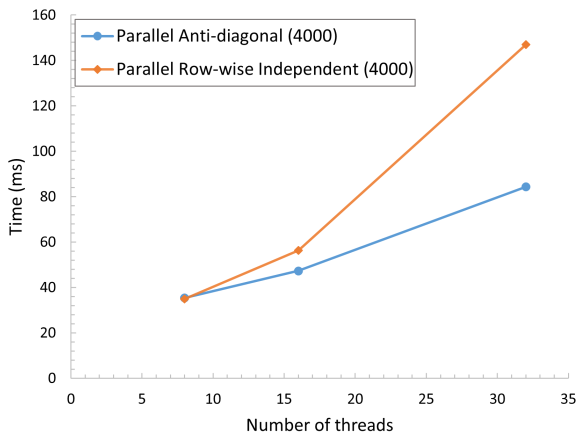


Fig. 15: Graph of Number of Threads vs. Execution Time for a Fixed String Size 4000 in OMP/CPU

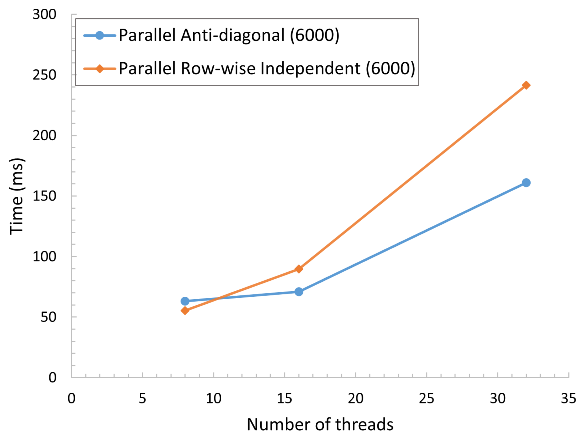


Fig. 16: Graph of Number of Threads vs. Execution Time for a Fixed String Size 6000 in OMP/CPU

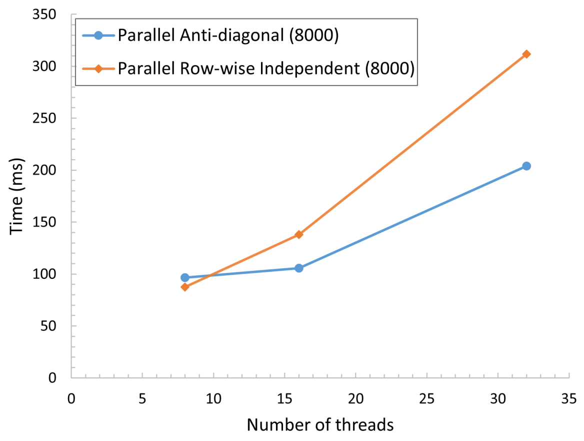


Fig. 17: Graph of Number of Threads vs. Execution Time for a Fixed String Size 8000 in OMP/CPU

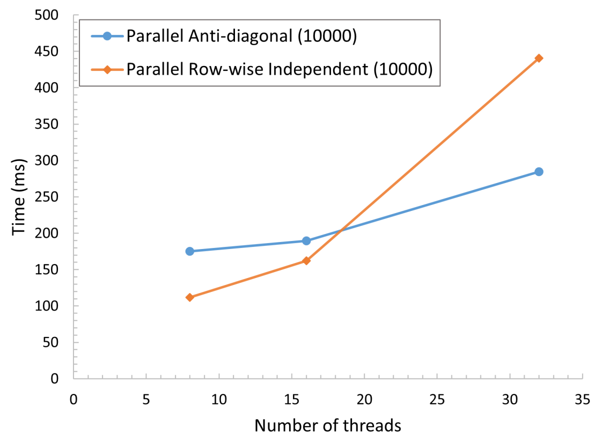


Fig. 18: Graph of Number of Threads vs. Execution Time for a Fixed String Size 10000 in OMP/CPU

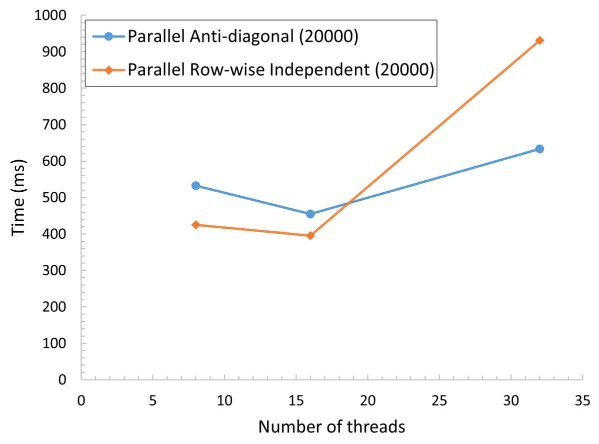


Fig. 19: Graph of Number of Threads vs. Execution Time for a Fixed String Size 20000 in OMP/CPU

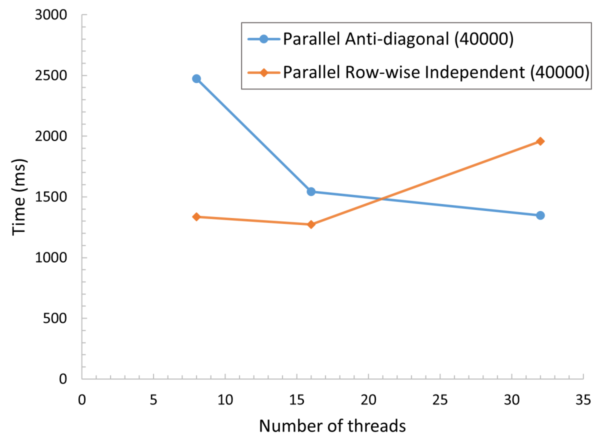


Fig. 20: Graph of Number of Threads vs. Execution Time for a Fixed String Size 40000 in OMP/CPU

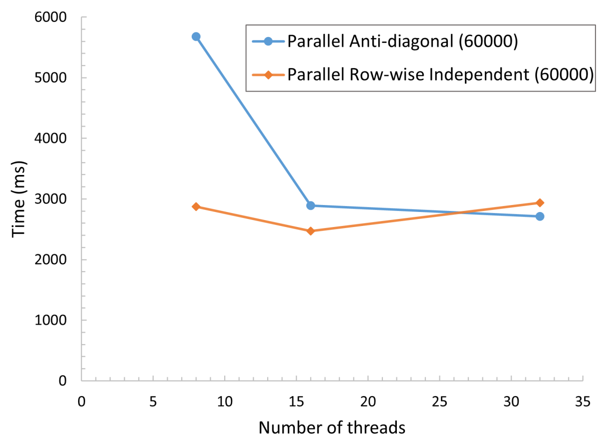


Fig. 21: Graph of Number of Threads vs. Execution Time for a Fixed String Size 60000 in OMP/CPU