CMPT 431 Project Report

Longest Common Subsequence

Ryan Milligan - 301 608 304
Adam Pywell - 301 335 414

Contents

1 Introduction

The Longest Common Subsequence (LCS) problem has many applications, including DNA Sequence analysis and version control systems. A subsequence is acquired by removing any number of entries from a given sequence. For example, given the sequence A = ABCD, the sequence AC would be a subsequence of A. While a substring requires that the entries of the substring be consecutive in the string from which it is derived, the entries of a subsequence do not need to be consecutive. They do, however, need to maintain their ordering. The sequence CB is not a subsequence of ABCD because the entries have been reordered.

The LCS problem is to find a sequence that is a subsequence of two input sequences and which is the longest subsequence that those two input sequences have in common. For example, given the sequences A = CGAGTAGCCT and B = TCTACTAAGG, the longest common subsequence of A and B is CATAG.

The Multiple Longest Common Subsequence (MLCS) problem involves finding a subsequence that is common to three or more input sequences. However, for this project, we will only consider the case of two input sequences.

2 Background

Notation:
For a sequence A of length n, $A = (a_1, a_2, \ldots, a_n)$
We denote the substring composed of the first i elements of A as A_i.
$A_i = (a_1, a_2, ..., a_i)$ where $i < n$.
$LCS(A, B)$ denotes the LCS of the sequences A and B.
$LCS(A_i, B_j)$ denotes the LCS of the first i elements of A and the first j elements of B.
For a matrix S, $s_{i,j}$ denotes the element of matrix S at the i-th row and j-th column.

The LCS problem can be broken down into subproblems where the LCS is computed for prefix strings of the two input sequences. Because there is overlap between these subproblems, the standard approach to solving the LCS problem is using dynamic programming. With the dynamic programming approach, a "score" matrix is constructed, where an element $s_{i,j}$ of score matrix S corresponds to the length of $LCS(Ai, Bj)$.

An individual element of the score matrix, $s_{i,j}$, can be computed from three adjacent cells: The cell above, the cell to the left, and the cell diagonally to the top-left. These three elements correspond to the lengths of $LCS(A_{i-1}, B_j)$, $LCS(A_i, B_{j-1})$, and $LCS(A_{i-1}, B_{j-1})$ respectively. For a given element $s_{i,j}$ of the matrix, if $a_i = b_j$, then they are the last element of $LCS(A_i, B_j)$ and $s_{i,j} = s_{i-1,j-1} + 1$. If $a_i \neq b_j$, then $s_{i,j} = max(s_{i-1,j}, s_{i,j-1})$. In the case where $i = 0$ or $j = 0$, then we cannot check the cells above, to the left, and to the top-left, as at least 2 of them will not exist. For this reason, it is convenient to pad the score matrix with a row of 0s at the top and a column of 0s on the left.

An important fact is that the computation of any cell depends on the solutions computed for the adjacent cells above, to the left, and to the top-left. This creates a transitive dependence on all cells above and to the left.

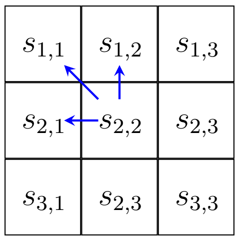


Figure 1: Dependencies for $s_{2,2}$

The serial implementation simply traverses the score matrix in row-major order and computes the score for each cell of the matrix. This ensures that for any cell, the dependencies of that cell have already been computed. As every cell of the matrix is visited and each cell is computed in constant time, the time complexity of this implementation is $O(nm)$, where n is the length of the first input sequence and m is the length of the second input sequence.

For parallel implementations, the matrix can be decomposed using a 2D column block decomposition, and each block of columns can be assigned to a thread.

		T	C	T	A	C
	0	0	0	0	0	0
T	0	T_1	T_1	T_2	T_2	T_3
G	0	T_1	T_1	T_2	T_2	T_3
C	0	T_1	T_1	T_2	T_2	T_3
A	0	T_1	T_1	T_2	T_2	T_3
A	0	T_1	T_1	T_2	T_2	T_3

Figure 2: Task decomposition with three threads, T_1, T_2, and T_3.

With this decomposition, each thread can compute its assigned block in row-major order. The major difference from the serial implementation, however, is that the threads must synchronize, so that thread T_n does not begin computing row i until thread T_{n-1} is finished computing its block of row i and has moved on to row $i + 1$. The first thread can begin immediately, as it has no neighbors to the left, but each other thread must have some way of being notified that the thread to its left is finished with the row and it is safe to proceed. This staggered traversal may at first seem inefficient, as many threads end up waiting around, unable to do any work until all of the preceding threads have finished the current row. However, this waiting stage only really occurs during the traversal of the first row. On all successive rows, the neighboring thread to the left will have been computing the next row while the thread in question was computing its previous row, and so the thread in question will not need to wait as long, if at all, before it can begin computation of the row.

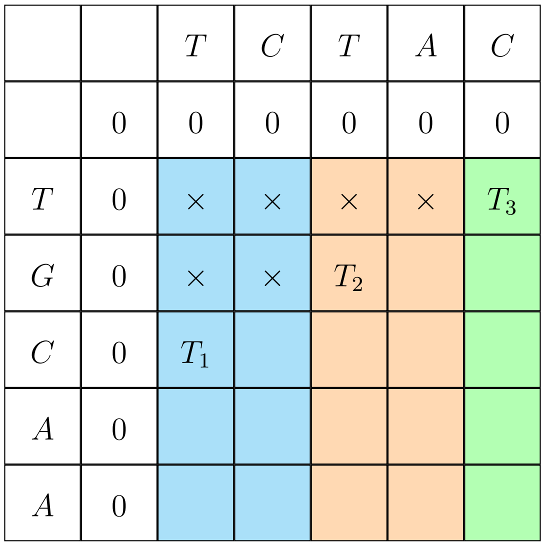
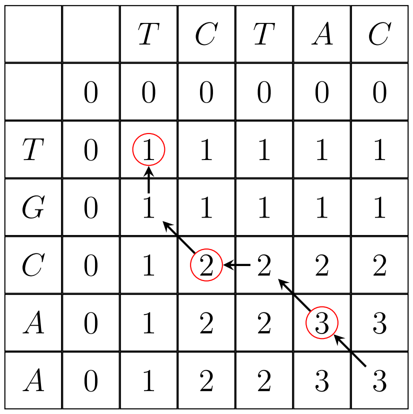


Figure 3: Each thread can only being computation of a row once the preceding thread has completed their portion of the row.

The bottom-right element of the completed score matrix corresponds to the length of the LCS. The LCS itself can be reconstructed by back-tracing through the matrix. Starting with the bottom-right element, if the elements above, to the left, and the top-left are all equal to each other, but are less than the current element, then the element of the input sequences which corresponds to the current element is an element of the LCS. When an element of the LCS is found, prepend it to the LCS string and then move to the cell diagonally to the top-left of the current cell. If the element of the input sequences corresponding to the current cell is not an element of the LCS, then, of the adjacent cells above, to the left, and the top-left, move to the cell with the highest score. In the case of a tie, move diagonally to the top left. Repeat this process until the number of elements of the LCS equals the score of the bottom-right cell.



		T	C	T	A	C
	0	0	0	0	0	0
T	0	1	1	1	1	1
G	0	1	1	1	1	1
C	0	1	2	2	2	2
A	0	1	2	2	3	3
A	0	1	2	2	3	3

Figure 4: $LCS(TGCAA, TCTAC) = TCA$

3 Implementation Details

Since much of the logic for solving the LCS problem remains the same regardless of whether the implementation is serial, parallel, or distributed, we used an abstract base class to implement the common data logic, to reduce code duplication. The rules for computing an individual cell, for instance, don't change between implementations. Each of our implementations are encapsulated in a class which inherits from this abstract base class, and each concrete class overrides the pure virtual `solve()` member function to implement the specifics of how each program solves the LCS problem.

3.1 Serial Implementation

Our serial implementation uses a dynamic programming approach to compute the length and content of the LCS between two input sequences. The algorithm operates in two phases: matrix computation and LCS reconstruction. The score matrix is represented as a 2D array where each entry `matrix[i][j]` stores the length of the LCS of the first i characters of `sequence_a` and the first j characters of `sequence_b`. The computation follows simple rules: if the characters match, the value is $1+$ `matrix[i - 1][j - 1]`, and if they do not, the value is the maximum of `matrix[i - 1][j]` and `matrix[i][j - 1]`. The matrix is traversed in row-major order, and each entry is computed one-by-one. After populating the matrix, the LCS is reconstructed by tracing back from the bottom-right corner to the top-left. The matrix is padded with a row of zeros on top and a column of zeros on the left, to handle boundary conditions. Consequently, for input sequences of length n and m, the score matrix is of dimension $(n+1) \times (m+1)$. Two timers are used, one to measure just the execution time of computing all entries of the score matrix, and a second to measure the execution time of solving the entire problem, including back-tracing through the matrix to reconstruct the LCS.

3.2 Parallel Implementation

Our parallel implementation LCS problem uses `C++11` threads to optimize performance by distributing the computation across multiple threads. The score matrix is decomposed following a column-wise block decomposition and each block of columns is assigned to a separate thread. Dependencies between cells of the score matrix are carefully managed using atomics and a condition variable to ensure that threads operate in the correct order.

A vector of atomic ints is used to record the current row index of each thread in shared memory, so that threads can check the progress of their neighboring threads. The condition variable is used to block threads from progressing until their neighbor to the left has progressed past the current row, ensuring that threads do not try to compute cells for which the dependent cells have not yet been computed. Whenever a thread completes its portion of a row, it signals the condition variable to wake up any threads that may be waiting. Spurious wake ups are prevented by passing a lambda function to the condition variable when blocking, to ensure that the thread is only awoken if its neighbor to the left has progressed far enough.

The solveParallel function assigns threads to compute their respective portions of the matrix. When a thread finishes processing a row, it signals the next thread to continue, ensuring synchronization between them. This is done using condition variables and mutexes, allowing threads to wait for others when needed. The wavefront approach used in this implementation ensures that threads are processing the matrix row by row, but they can handle different sets of columns, improving parallelism and reducing potential idle times.

This parallel LCS implementation demonstrates the effectiveness of multi-threading in solving complex problems like LCS. The workload is efficiently shared among threads, resulting in faster computation times, and the synchronization mechanisms ensure that the parallel execution remains correct. This approach shows the significant performance improvements that can be achieved for large datasets through parallelization.

3.3 Distributed Implementation

Our distributed implementation uses MPI to efficiently handle large input sequences by dividing the workload among multiple processes. It achieves this by partitioning the columns of the score matrix by splitting up the second input sequence, `sequence_b` into substrings.

Each process is assigned a portion of `sequence_b` and a full copy of `sequence_a` to construct a local score matrix. Each process computes the values for its local matrix independently, in row-major order, but inter-process communication is required for handling dependencies along the local matrix's boundaries.

To handle cell dependencies, at the start of each row, processes except for the leftmost process make a call to `MPI_Recv()` and await the data for the rightmost cell of the row from the process to their left. When the data is received, the process stores it in the cell of the 0-th column, ensuring that it will be accessible to the process when computing the next row. When a process finishes computing the last cell of a row of its local matrix, if it is not the rightmost process, it makes a call to `MPI_Send()` and sends the value computed for that cell to the neighboring process to the right.

The back-tracing process is somewhat more complicated for the distributed implementation. The back-tracing process is inherently serial, but memory is not shared in our distributed implementation, which complicates things. One approach that was considered was to gather all of the local matrices together to reconstruct a combined score matrix and then back-tracing through it with a single thread. This proved to be extremely inefficient, especially for very large matrices. Instead, we decided to perform the back-trace across the separate processes and their local matrices. Starting with the rightmost process, the back-trace is performed as usual, until the left bound of the local matrix is reached. At this point, two calls to `MPI_Send()` are made to the process to the left. The first call is to send the index of the LCS that the process was on and the index of the row that the process was on when it reached the end of its local matrix. The receiving process already knows what column it was on, since the send is only done once the left boundary is reached. From this, the receiving process can pick up the work at the correct cell of its local matrix. The second `MPI_Send()` call that the finishing process makes is to send a buffer containing the portion of the LCS that the process was able to reconstruct. This process repeats until the leftmost process completes the reconstruction of the LCS In order for all of the processes to know how much space to allocate for their local LCS buffer, before the reconstruction begins, the rightmost process broadcasts the total length of the LCS to all of the other processes. This distributed reconstruction approach avoids transmitting very large amounts of data across the network, and proves to be much more efficient.

This distributed LCS implementation showcases how parallel programming and inter-process communication can be leveraged to address computationally intensive tasks. By dividing the workload and effectively coordinating between processes, it achieves scalable performance for large input sequences, making it well-suited for distributed systems and high-performance computing environments.

4 Evaluations

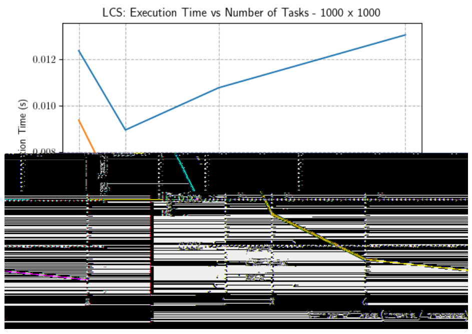


Figure 5: Execution time for sequences of length 1000.

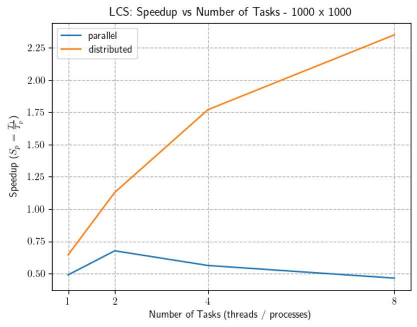


Figure 6: Speedup for sequences of length 1000.

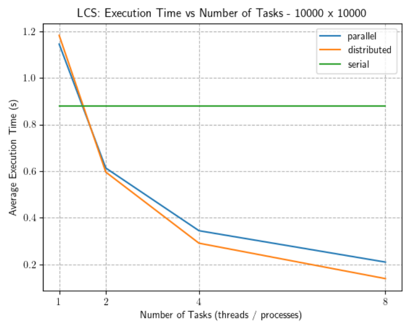


Figure 7: Execution time for sequences of length 10 000.

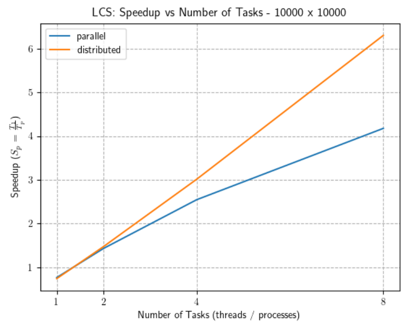


Figure 8: Speedup for sequences of length 10 000.

From the graphs, we can see that both the parallel and distributed implementations have a higher execution time when running with just one thread or process. For very large input sequences however, they both significantly improve their performance as more processing units are added. In particular, the speedup for the distributed version scales linearly as more processes are added, but the parallel version tapers off a little.

For sequence lengths of 1000, the distributed program saw less improvement of performance with increased processes. The parallel program sees an initial boost in performance when going from 1 thread to 2 threads, but the performance decreases from there when adding additional threads. Evidently, the parallel program requires much larger input sequences to see significant benefits from parallelization.

5 Conclusions

The LCS problem is one that can benefit greatly from parallelization. By taking advantage of how data dependencies are arranged, we can greatly improve performance by overlapping communication and computation. Each cell of the score matrix depends on cells above and to the left of it, so by decomposing the score matrix into blocks of columns, and mapping those blocks to different tasks, we ensure that each task depends on only the task to its left. This means that there are no global synchronization points where each task has to wait for all of the others. While there is some idle time at the very start of the computation, where processes have to wait for the rest of the first row to be computed before they may start, once they do start, the staggered nature of the synchronization means that they are unlikely to have to wait very long when they get to successive rows.

There is, however, still a significant amount of communication and synchronization overhead. As can be seen from the results of our experiments, very large sequence lengths are required to see meaningful performance gains for the parallel implementation. The distributed version doesn't require as large sequences, but does still benefit more from larger ones. When running the program with sequence lengths of 100, the results were dismal even for the distributed version. We had tried running the programs with sequence lengths of $100\,000$, but that exceeded the memory limits available to us on the Slurm cluster. Nonetheless, it is clear that the distributed algorithm for solving the LCS problem lends itself especially well towards computing the LCS of very large input sequences.