LCS Problem

Joshua Kim, **Parshan** Pourbakht, **Vincente** Buenaventura

Background

The Longest Common Subsequence (LCS) problem is about finding the longest sequence of characters that appear in the same order in two strings. For example, given "ABCDEF" and "BCZE", the LCS is "BCE". The traditional way to solve this uses a dynamic programming table, where each cell represents the length of the LCS up to certain points in the strings.

Parallel Implementation

What Is the Wavefront Approach?

The wavefront approach processes the dynamic programming (DP) table diagonally instead of row-by-row or column-by-column. This works well for problems like LCS, where the value of each cell in the DP table depends on three neighbors: the one directly above, the one directly to the left, and the one diagonally above-left.

Instead of waiting for entire rows or columns to finish, the wavefront approach capitalizes on the fact that all cells on the same diagonal (where row + col is constant) can be computed independently. This unlocks parallelism because cells on a diagonal don't depend on each other but only on the prior diagonal.

Implementation

In our implementation of the parallel LCS (Longest Common Subsequence) algorithm, we used a wavefront approach combined with multi-threading to speed up the computation. The basic idea is to divide the task into smaller subproblems, where each thread calculates the LCS for a diagonal of the table. This helps in making the process more efficient, especially for larger strings.

We begin by initializing the memoization table with zeros and adding an extra row and column for the base case, which is done in the initialize_table function.

Most of the work is done in the wavefront_worker function. Each thread calculates the LCS values for a diagonal of the table, iterating over the rows for each diagonal. The characters are compared, and the LCS value is updated either from the top, left, or diagonal cells, depending on whether the characters match.

```
void wavefront_worker(MemoizedTable& lcsTable, const std::string& str1, const std::string& str2, int numRows, int numCols, int diag) {
```

```
  int rowStart = std::max(1, diag - numCols);
  int rowEnd = std::min(numRows, diag - 1);

  for (int row = rowStart; row <= rowEnd; ++row) {
    int col = diag - row;
    if (str1[row - 1] == str2[col - 1]) {
      lcsTable[row][col] = lcsTable[row - 1][col - 1] + 1;
    } else {
      lcsTable[row][col] = std::max(lcsTable[row - 1][col], lcsTable[row][col - 1]);
    }
  }
}
```

In the lcs_parallel function, the core logic for multi-threading is implemented. We create a set of worker threads where each one handles a specific range of diagonals. The threads communicate via a mutex and condition variable to ensure that they only process a diagonal once the previous one is complete.

```
auto threadWorker = [&](int threadId) {
  for (int diag = threadId + 2; diag <= numRows + numCols; diag += threadSize) {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [&]() { return completedDiag >= diag - 1; });

    wavefront_worker(lcsTable, str1, str2, numRows, numCols, diag);

    std::lock_guard<std::mutex> guard(mtx);
    completedDiag = diag;
    cv.notify_all();
  }
};
```

Once all the threads complete their work, we backtrack through the memoization table to reconstruct the LCS string. The backtrack_lcs function traces through the table, moving up or left depending on the values, and constructs the LCS by appending matching characters.

Strengths of the Wavefront Approach

1. Parallelism: By computing diagonals independently, multiple threads can work simultaneously without interfering with one another. This makes the approach highly scalable for large strings.
2. Fine-Grained Dependencies: Unlike row-by-row or column-by-column approaches, wavefront ensures dependencies are resolved as soon as possible, minimizing idle time.

3. Load Balancing: The round-robin distribution ensures threads are evenly utilized.

Weaknesses of the Wavefront Approach

1. Synchronization Overhead: Each thread must wait for the previous diagonal to complete before proceeding. This introduces some latency, especially for small strings.
2. Thread Bottlenecks: In cases where diagonals have significantly different amounts of work (e.g., near the start or end of the table), some threads might finish early and remain idle.
3. Implementation Complexity: Compared to simpler approaches, managing diagonal indices, thread assignments, and synchronization requires more code and careful handling.

Distributed Implementation

What Is the Row-based Decomposition?

Row-based decomposition is a method of dividing the computation of the LCS table by assigning rows to different processes. The LCS table itself is a 2D matrix, where the value at cell (row, col) depends on values from (row-1, col) and (row, col-1). This dependency structure allows the table to be naturally split into rows, with each process responsible for computing a contiguous subset.

Implementation

In our distributed approach to calculating the Longest Common Subsequence (LCS), we used a row-based decomposition technique to split the LCS table among multiple processes. This method divides the table into segments, where each segment corresponds to a specific range of rows. Each process is then responsible for computing the values in its designated rows. The number of rows in the table, determined by the lengths of the two strings (str1 and str2), is distributed across the available processes. The start and end rows for each process are calculated dynamically to ensure an even distribution of work. For example, if there are 10 rows and 4 processes, each process handles approximately 2 rows. This division allows us to perform parallel computation efficiently.

Each process then computes its assigned rows by comparing characters from str1 and str2. This is done in a for loop that iterates over the rows and columns, filling the LCS table according to the standard dynamic programming approach. If the characters at the current positions match, we increment the value from the diagonal. If not, we take the maximum from the cell above or to the left. Here's how the computation looks within the lcs function:

```
void lcs(const std::string &str1, const std::string &str2,
        int startRow, int endRow, MemoizedTable &lcsTable) {
```

```
  int numRows = str1.size();
  int numCols = str2.size();

  for (int row = startRow; row < endRow; ++row) {
    for (int col = 1; col <= numCols; ++col) {
      if (str1[row - 1] == str2[col - 1]) {
        lcsTable[row][col] = lcsTable[row - 1][col - 1] + 1;
      } else {
        lcsTable[row][col] = std::max(lcsTable[row - 1][col], lcsTable[row][col - 1]);
      }
    }
  }
}
```

The process synchronization is key to ensuring that each process has the correct data to work with. After computing its assigned rows, each process must communicate with its neighbors to share information. This happens in two stages: first, a process sends the last row of its computed segment to the next process, and second, it receives the last row of the previous process's segment. The communication is achieved through MPI's MPI_Send and MPI_Recv functions. These operations ensure that no process works with incomplete data. For example, if a process is not the first, it receives data from the previous process before continuing its computation:

```
if (rank > 0) {
  MPI_Recv(lcsTable[startRow - 1].data(), numCols + 1, MPI_INT,
           rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Similarly, if a process isn't the last, it sends the last row of its segment to the next process:

```
if (rank < size - 1) {
  MPI_Send(lcsTable[endRow - 1].data(), numCols + 1, MPI_INT,
           rank + 1, 0, MPI_COMM_WORLD);
}
```

Once all processes have completed their computations, the root process (rank 0) collects the results from all other processes. This is done by receiving the computed rows from each process using MPI_Recv. The root process ensures that the entire LCS table is gathered before proceeding with the next steps. Here's the code snippet responsible for receiving the data:

```
for (int p = 1; p < size; ++p) {
  int processStartRow = p * processRows + 1;
  int processEndRow = std::min((p + 1) * processRows + 1, numRows + 1);

  for (int row = processStartRow; row < processEndRow; ++row) {
    MPI_Recv(lcsTable[row].data(), numCols + 1, MPI_INT, p, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
  }
}
```

With the full table now gathered, we can backtrack from the bottom-right corner to reconstruct the LCS string. This backtracking process traces the highest values in the table to identify the common subsequence. If the characters at the current positions in str1 and str2 match, we add that character to the result and move diagonally in the table. If the values indicate a choice between moving up or left, we move in the direction that preserves the LCS length. Once the backtracking is complete, the LCS string is reversed to give the correct result.

Strengths of the Row-based Approach

1. Scalability: The algorithm handles larger datasets well, as the workload is split across multiple processes. More processes mean faster computation times, which is especially useful for handling big inputs.
2. Efficient Resource Use: By assigning each process a separate chunk of the data, the algorithm maximizes CPU usage. Each process works independently, minimizing the need for frequent communication, which speeds up the overall execution.
3. Simple Synchronization: Synchronizing the processes is straightforward with MPI's MPI_Send and MPI_Recv. Since processes only communicate with their neighbors, the synchronization steps are simple and easy to manage.

Weaknesses of the Row-based Approach

1. Communication Overhead: The need for processes to frequently exchange data can slow things down, particularly as the number of processes grows. The communication time can become a bottleneck in some cases.
2. Uneven Load: If the number of rows isn't evenly divisible by the number of processes, some processes might end up doing more work than others, leading to inefficiency and slower performance for those processes.
3. Memory Usage: Each process needs to store its own portion of the LCS table, which increases memory usage. For large inputs, this can be a problem, especially on machines with limited memory. Additionally, the root process must gather all results, requiring enough memory to hold everything.

Analysis

Benchmarking

We evaluated the three implementations: serial, parallel using C++ threads, and distributed using MPI, using different size input strings and different number of threads(1, 2, 4, 8) for the parallel and distributed versions. These experiments were executed on the cluster.

Serial

The serial implementation served as a baseline reference point to evaluate the efficiency of the parallel and distributed versions.

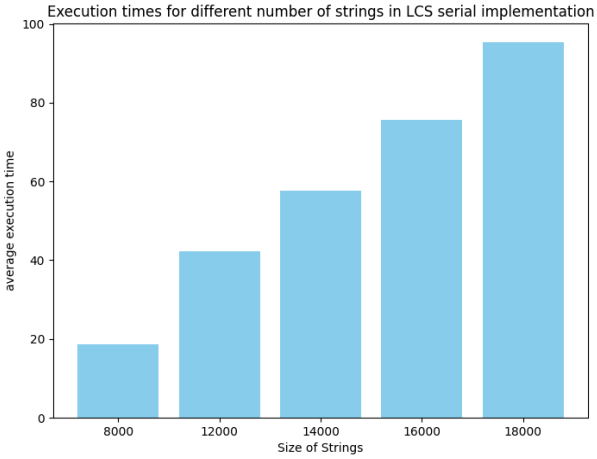
Performance

Execution Time:

String size	**Time**
8,000	18.66s
12,000	42.14s
14,000	57.53s
16,000	75.43s
18,000	95.57s

The runtime seems to increase linearly with the size of the input strings. Based on the graph we can see a linear increase in execution time from 12,000 - 18,000 length strings with a slightly greater increase from 8,000 - 12,000 length strings.

This is expected as the serial implementation has a O(m x n) complexity where m and n are the length of the two strings.



Parallel

Performance

Execution Time:

String size	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**
8,000	8.03s	14.34s	14.28s	17.89s
12,000	20.54s	31.35s	31.38s	34.12s
14,000	34.5s	42.78s	42.51s	52.53s
16,000	44.13s	55.03s	54.50s	67.14s
18,000	59.83s	69.63s	68.65s	77.75s

Computed speed-ups:
Speed-up = Execution Time with 1 thread / Execution Time with n threads

String size	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**
8,000	18.66/8.03=2.32	18.66/14.34 = 1.30x	18.66/14.28 = 1.31x	18.66/17.89=1.04 x
12,000	42.14/20.54=2.0 5	42.14/31.35=1.34x	42.14/31.38=1.34x	42.14/34.12=1.26 x
14,000	57.53/34.5=1.68	57.53/42.78=1.34x	57.53/42.51=1.35x	57.53/52.53=1.10 x

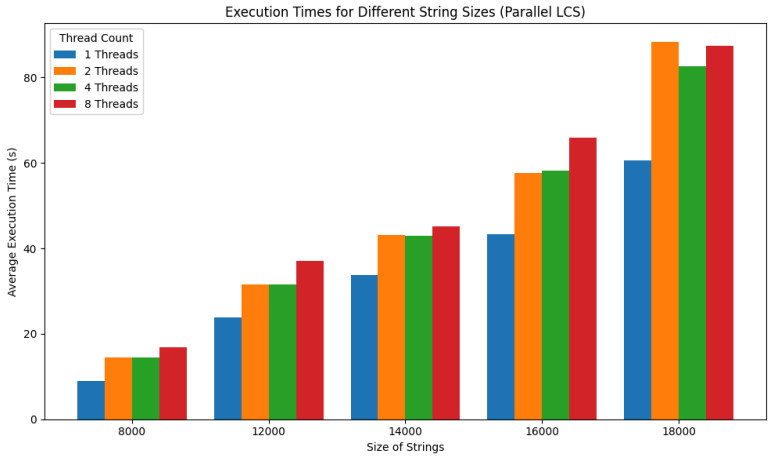
16,000	75.43/44.13=1.71	75.43/55.03=1.37x	75.43/54.50=1.38x	75.43/67.14=1.12x
18,000	95.57/59.83=1.60	95.57/69.63=1.37x	95.57/68.65=1.39x	95.57/77.75=1.23x

Compared to the serial execution times, we are seeing speed-up throughout all string sizes and number of threads. The speed-up increases the smaller the strings and the less threads are used.

Challenges

With more threads, comes more overhead for thread synchronization. As shown in the graph, the gradual increase in execution time as the string sizes increase is noticeable, especially for the experiment with string length 18,000.

This is also partly due to the uneven workload among threads where the middle diagonals have more work than
the edge diagonals, making the edge diagonal threads wait. This unevenness grows larger as string lengths increase, hence why the larger strings take longer.



Distributed (with MPI)

Performance

Execution Time:

String size	**1 Process**	**2 Processes**	**4 Processes**	**8 Processes**

8,000	5.10s	5.26s	5.16s	5.20s
12,000	11.59s	11.89s	11.73s	11.71s
14,000	15.77s	16.23s	15.93s	16.00s
16,000	20.74s	21.29s	20.97s	21.10s
18,000	26.33s	26.96s	26.38s	26.79s

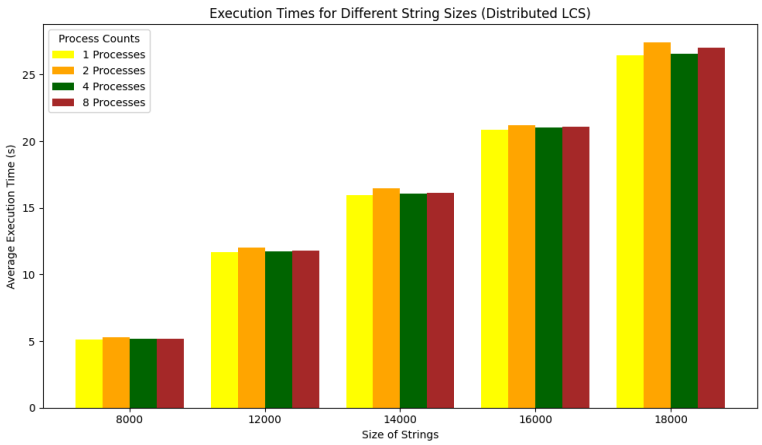
Computed speed-ups:
Speed-up = Execution Time with 1 process/ Execution Time with n processes

String size	**1 Process**	**2 Processes**	**4 Processes**	**8 Processes**
8,000	18.66/5.10=3.66	18.66/5.26 = 3.54x	18.66/5.16 = 3.62x	18.66/5.20=3.59x
12,000	42.14/11.59=3.64	42.14/11.89=3.54x	42.14/11.73=3.59x	42.14/11.71=3.60x
14,000	57.53/15.77=3.65	57.53/16.23=3.54x	57.53/15.93=3.61x	57.53/16.00=3.60x
16,000	75.43/20.74=3.64	75.43/21.29=3.54x	75.43/20.97=3.60x	75.43/21.10=3.57x
18,000	95.57/26.33=3.63	95.57/26.96=3.54x	95.57/26.38=3.62x	95.57/26.79=3.57x

The distributed version achieves an approximate speed-up of 3.6x across all string sizes and process counts. As opposed to the parallel version, this consistent speed-up across all variables suggests that MPI efficiently manages overhead issues such as synchronization and communication. But having a consistent speed-up also means that there is no increase in speed-up as process count increases. This is likely due to the frequent communication required between processes especially for the diagonal based approach.

Challenges

As the number of processes increases, the time spent on inter-process communication can outweigh the computational gains, limiting speed-up. This is something we slightly see with the speed-up from 1 to 2 processes and 4 to 8. Debugging the distributed version was also a challenge we faced as the asynchronous nature of execution introduces more complexity and also the need to manage communication explicitly.



Key Takeaways

Throughout the process of completing this project, we faced computational challenges and trade-offs for solving the Longest Common Subsequence (LCS) problem for the three different implementations, serial, parallel, and distributed. Understanding each version's strengths and weaknesses gave us a deeper understanding of the algorithm design and performance optimization.

- The serial implementation highlighted the O(m x n) complexity of the problem, serving as a baseline for evaluating the improvements through parallel and distributed designs. While the serial version is straightforward, the approach quickly becomes computationally limited by larger string sizes.
- The parallel implementation using the wavefront approach demonstrated speed-up through parallelism and fine-grained dependency attributes. Although speed-up was achieved, we had challenges with the inherent communication and load-balancing issues that come with parallelism.
- The distributed version showed significant speed-ups overall for all string sizes and process counts. However, similarly to the parallel version our implementation was bottlenecked by inter-process communication issues and data distribution.
- After implementing all three versions, we realized trade-offs between computation and communication where the balancing of these two factors decided the efficiency of the program.
- This project highlighted the importance of scalability considerations in algorithm design, especially as data sizes and computational resources grow.