LONGEST COMMON SUBSEQUENCE
Parallelizing LCS through Anti-Diagonal Approach

Name: Saema Nadim
Person#: 50469138
Instructor: Dr. Russ Miller
CONTENT

➢ What is LCS?
➢ It’s Applications
➢ Example of LCS
➢ Sequential Approach
➢ Need for parallelization
➢ Parallel Approach
➢ Changes after last presentation
➢ Results and Graphs (Sequential, Parallel, Comparison, Speedup)
➢ Observations
➢ References
What is LCS?

- As the name suggests, this algorithm is used to find Longest Common Subsequence among two or more strings.
- It uses a dynamic programming approach to do so. It can also use recursion but DP is faster and more efficient.
- The solution for each comparison depends on the solution of previous comparisons.
- It is an NP-Hard problem if arbitrary number of sequences are provided as input, but for constant number of sequences it can be solved in polynomial time.
It’s Applications

It has wide amount of real world applications:

- finding similar regions of two nucleic acid sequences – like DNA
- in the Computer Science field to compare two codes in git while merging.
- in Computational Linguistics
- even in algorithms to detect AI, since it can detect similar texts!
Example

Consider two strings of length 10 –

1. String1: QTSRTTTSTR
2. String2: SQSTTRQSTT

Their **Longest Common Subsequence** is highlighted with red. It will be QSTTST.
Sequential Approach

• LCS is usually solved using Dynamic Programming.

• The matrix is filled row wise under two nested for loops, where in one loop ‘i’ iterates from 0 to m(length of String1) and in the next loop ‘j’ iterates from 0 to n(length of String2).

• The time and space complexity is O(m*n).
Sequential Approach

• The value of each element is calculated using following formula-

\[
dp[i][j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
\text{dp}[i-1][j-1] + 1 & \text{if String1}[i] = \text{String2}[j] \\
\max(dp[i-1][j], dp[i][j-1]) & \text{if String1}[i] \neq \text{String2}[j] 
\end{cases}
\]

It can be seen that each element’s value depends on its previous diagonals.

• The last bottom right value of the calculated matrix tells us the length of LCS, and the matrix can be traced back from the last element to find the required subsequence.
My Sequential Approach

• As seen earlier, we fill the matrix row wise and it makes it difficult to parallelize the algorithm.

• I have just changed the way we fill the matrix, the formula used is the same.

• In my algorithm, we are iterating through each diagonal of the matrix represented by ‘line’. For each diagonal, its start_row and end_row is calculated.

• We need to iterate through the rows and fill the elements of the diagonal using the formula of previous slide.

```cpp
int sequential_lcs(char *s1, char *s2, int len_s1, int len_s2) {
    int rows = len_s1 + 1;
    int cols = len_s2 + 1;

    int dp[rows][cols];
    dp[0][0]=0;

    for (int line=1; line-rows<cols; line++) {
        int start_row = max(1, line - len_s2 + 1);
        int end_row = min(len_s1, line);

        for (int i = start_row; i <= end_row; i++) {
            int j = end_row - i + start_row;
            if (i==1) {
                dp[i-1][j]=0;
            }
            if (j==1) {
                dp[i][j-1]=0;
            }
            if (s1[i-1] == s2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
            }
        }
    }
    return dp[rows-1][cols-1];
}
```
My Sequential Approach

Each black arrow represents the direction of iteration.
Need for parallelization

- **Reduced computation time:** The computation of the LCS is a computationally expensive task, especially for long input sequences. Parallelizing the computation can help reduce the computation time by distributing the workload across multiple processors or computing nodes.

- **Better resource utilization:** Parallelization allows better utilization of available computing resources, such as multi-core processors or clusters.

- **Scalability:** As the size of the input increases, parallelization allows us to handle larger inputs while still achieving reasonable computation times.

- **Improved efficiency:** Parallel algorithms can reduce the time to solution, and allow researchers to perform larger or more complex analyses in the same amount of time.
Parallel Approach

- Parallel Approach is similar to previous sequential approach such that each element of every diagonal is iterated in the direction of the arrow.

- Each diagonal is divided into all available processes using a simple formula.
Parallel Approach

- The boundary values are exchanged through MPI_Send and MPI_Recv to the adjacent processes.

- In the given example, 4 processes are used:
  - Process 0 is represented by Pink,
  - Process 1 is represented by Orange,
  - Process 2 is represented by Yellow, and
  - Process 3 is represented by Blue.

- When the process is calculating it’s part of the matrix (eg. Process 1), it receives the last boundary value of previous process (eg. Process 0) and first boundary value of next process(eg. Process 2). It also sends it’s own boundary values to those processes.
Output Screen

- This is my output screen for 32 Nodes.
- The length of each string is 100000 characters.
Changes after last presentation

• Earlier, I was using 1 node and multiple processors. I could run my algorithm till 64 processors. Now, I have used 1 Node per processor and I could go till 128 Nodes.
• Earlier I took max input length of 2000, now I have taken the max input length of 100,000.
• Used a slurm script.
• Compared Sequential and Parallel execution.
• Calculated Speedup.
# Results for Sequential Approach

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>Time (in s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.00000408</td>
</tr>
<tr>
<td>50</td>
<td>0.000054614</td>
</tr>
<tr>
<td>100</td>
<td>0.00010246</td>
</tr>
<tr>
<td>1000</td>
<td>0.02062343</td>
</tr>
<tr>
<td>10000</td>
<td>0.61874786</td>
</tr>
<tr>
<td>20000</td>
<td>3.284738392</td>
</tr>
<tr>
<td>30000</td>
<td>7.485478848</td>
</tr>
<tr>
<td>40000</td>
<td>16.8582492</td>
</tr>
<tr>
<td>50000</td>
<td>21.6216583</td>
</tr>
<tr>
<td>60000</td>
<td>31.07484096</td>
</tr>
<tr>
<td>70000</td>
<td>43.769907</td>
</tr>
<tr>
<td>80000</td>
<td>60.822275</td>
</tr>
<tr>
<td>90000</td>
<td>83.3521512</td>
</tr>
<tr>
<td>100000</td>
<td>112.123667</td>
</tr>
</tbody>
</table>

![Graph showing the relationship between input size and time](image-url)
### Results for Parallel Approach (small input size)

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time (in s) for Input size 10</th>
<th>Time (in s) for Input size 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00199635</td>
<td>0.01138207</td>
</tr>
<tr>
<td>4</td>
<td>0.00368789</td>
<td>0.0040272</td>
</tr>
<tr>
<td>8</td>
<td>0.01609147</td>
<td>0.00588514</td>
</tr>
<tr>
<td>16</td>
<td>0.02928861</td>
<td>0.00793963</td>
</tr>
<tr>
<td>32</td>
<td>0.00538751</td>
<td>0.04484502</td>
</tr>
<tr>
<td>64</td>
<td>0.13329603</td>
<td>0.11810985</td>
</tr>
</tbody>
</table>
## Results for Parallel Approach (large input size)

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Time (in s) for Input size 10000</th>
<th>Time (in s) for Input size 20000</th>
<th>Time (in s) for Input size 30000</th>
<th>Time (in s) for Input size 40000</th>
<th>Time (in s) for Input size 50000</th>
<th>Time (in s) for Input size 60000</th>
<th>Time (in s) for Input size 70000</th>
<th>Time (in s) for Input size 80000</th>
<th>Time (in s) for Input size 90000</th>
<th>Time (in s) for Input size 100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.52578763</td>
<td>2.681668937</td>
<td>6.268362552</td>
<td>11.19016013</td>
<td>16.90770924</td>
<td>24.78136885</td>
<td>34.72393444</td>
<td>52.09097353</td>
<td>50.5329981</td>
<td>107.9106947</td>
</tr>
<tr>
<td>4</td>
<td>0.488477363</td>
<td>1.973840537</td>
<td>4.153437258</td>
<td>7.979478955</td>
<td>11.56600832</td>
<td>15.96001254</td>
<td>21.68704922</td>
<td>30.04386666</td>
<td>37.89914124</td>
<td>43.24429724</td>
</tr>
<tr>
<td>8</td>
<td>0.685008492</td>
<td>2.345982661</td>
<td>4.569211543</td>
<td>7.45193071</td>
<td>10.8848518</td>
<td>16.00331206</td>
<td>20.3369828</td>
<td>31.8711065</td>
<td>37.23287834</td>
<td>37.23287834</td>
</tr>
<tr>
<td>16</td>
<td>0.945190889</td>
<td>2.7905281</td>
<td>5.650291473</td>
<td>8.953158803</td>
<td>12.8032219</td>
<td>17.19117304</td>
<td>22.4127304</td>
<td>27.71922095</td>
<td>33.90865398</td>
<td>39.99139587</td>
</tr>
</tbody>
</table>
Results for Parallel Approach (large input size)
Comparison of Sequential and Parallel Execution

Here, I have compared sequential execution graph with graphs obtained using parallel execution on 32 Nodes and 64 Nodes.
Speedup Graph

- Speedup is the execution time of a sequential program divided by the execution time of a parallel program that computes the same result.

- Speedup = $T_{\text{sequential}} / T_{\text{parallel}}$
Observations

• The graph of the sequential algorithm keeps increasing.
• It can be seen that for less number of processors, the graph of the time taken by the parallel algorithm is similar to the sequential algorithm graph.
• As the processor increases, the time taken decreases but till a certain point of time.
• After a point, time starts increasing again due to communication overhead between processes.
References


• https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6458724/#CR14

• https://www.researchgate.net/publication/332352052_An_OpenMP-based_tool_for_finding_longest_common_subsequence_in_bioinformatics

• https://ieeexplore.ieee.org/document/8326619
Thank You!