LONGEST COMMON SUBSEQUENCE

Parallelizing LCS through Anti-Diagonal Approach

Name : Saema Nadim

Person# 50469138

Instructor : Dr. Russ Miller

University at Buffalo The State University of New York

CONTENT

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

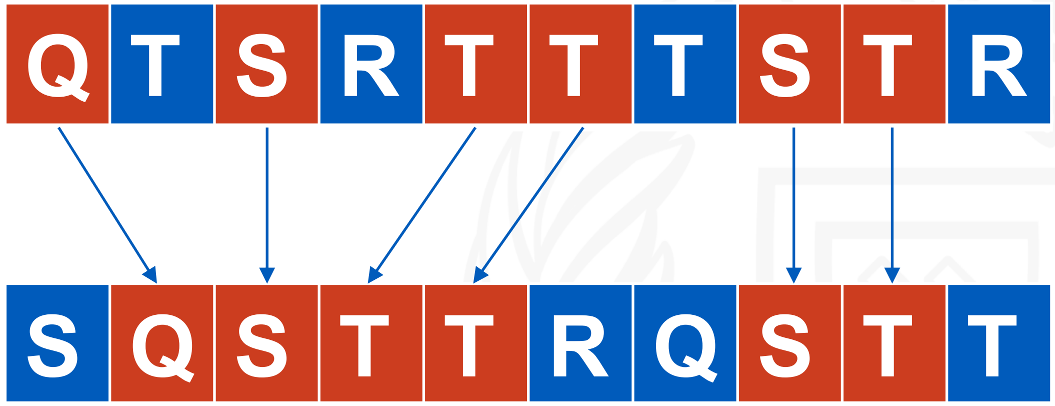
It has wide amount of real world applications:

- 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

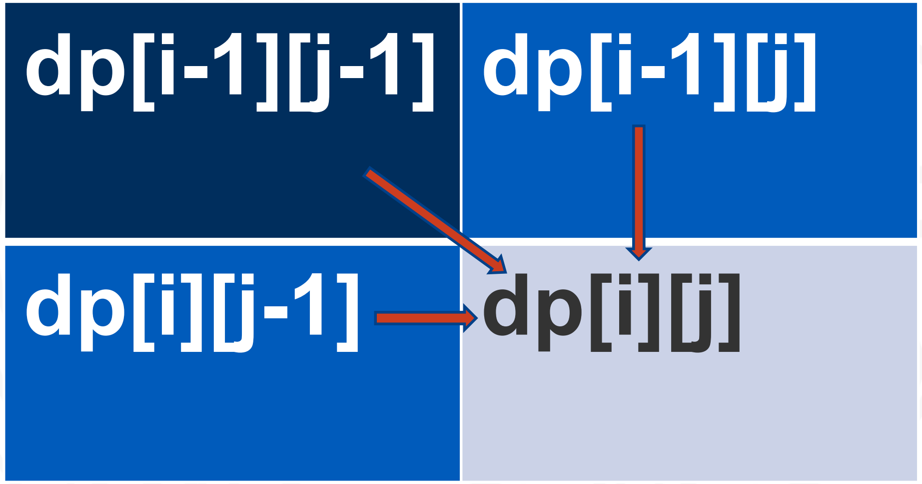
- LCS is usually solved using Dynamic Programming.
- The matrix is filled row wise under two nested for loops, where in one loop ‘i’ iterates from 0 to m(length of String1) and in the next loop ‘j’ iterates from 0 to n(length of String2).
- The time and space complexity is O(m*n).

Sequential Approach

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

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

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.

```
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-1][j], dp[i][j-1]);
            }
        }

    }
    return dp[rows-1][cols-1];
}
```

My Sequential Approach

Each black arrow represents the direction of iteration.

	0	S	Q	S	T	T	R	Q	S	T	T
0	0	0	0	0	0	0	0	0	0	0	0
Q	0	0	1	1	1	1	1	1	1	1	1
T	0	0	1	1	2	2	2	2	2	2	2
S	0	1	1	2	2	2	2	2	3	3	3
R	0	1	1	2	2	2	3	3	3	3	3
T	0	1	1	2	3	3	3	3	3	4	4
T	0	1	1	2	3	4	4	4	4	4	5
T	0	1	1	2	3	4	4	4	4	5	5
S	0	1	1	2	3	4	4	4	5	5	5
T	0	1	1	2	3	4	4	4	5	6	6
R	0	1	1	2	3	4	5	5	5	6	6

Time

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 arrow.

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

	0	S	Q	S	T	T	R	Q	S	T	T
0	P0	P0	P0	P0	P0	P0	P0	P0	P0	P0	P0
Q	P1	P1	P1	P1	P1	P0	P0	P0	P0	P0	P0
T	P2	P2	P1	P1	P1	P1	P0	P0	P0	P0	P0
S	P3	P2	P2	P1	P1	P1	P1	P1	P0	P0	P0
R	P3	P3	P2	P2	P1	P1	P1	P1	P0	P0	P0
T	P3	P3	P2	P2	P1	P1	P1	P1	P1	P0	P0
T	P3	P3	P2	P2	P2	P1	P1	P1	P1	P1	P0
T	P3	P3	P2	P2	P2	P2	P2	P1	P1	P1	P0
S	P3	P3	P2	P2	P2	P2	P2	P2	P1	P1	P0
T	P3	P3	P3	P3	P3	P3	P3	P2	P2	P1	P0
R	P3	P3	P3	P3	P3	P3	P3	P3	P2	P1	P0

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.

	0	S	Q	S	T	T	R	Q	S	T	T
0	0	0	0	0	0	0	0	0	0	0	0
Q	0	0	1	1	1	1	1	1	1	1	1
T	0	0	1	1	2	2	2	2	2	2	2
S	0	1	1	2	2	2	2	2	3	3	3
R	0	1	1	2	2	2	3	3	3	3	3
T	0	1	1	2	3	3	3	3	3	4	4
T	0	1	1	2	3	4	4	4	4	4	5
T	0	1	1	2	3	4	4	4	4	5	5
S	0	1	1	2	3	4	4	4	5	5	5
T	0	1	1	2	3	4	4	4	5	6	6
R	0	1	1	2	3	4	5	5	5	6	6

Output Screen

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

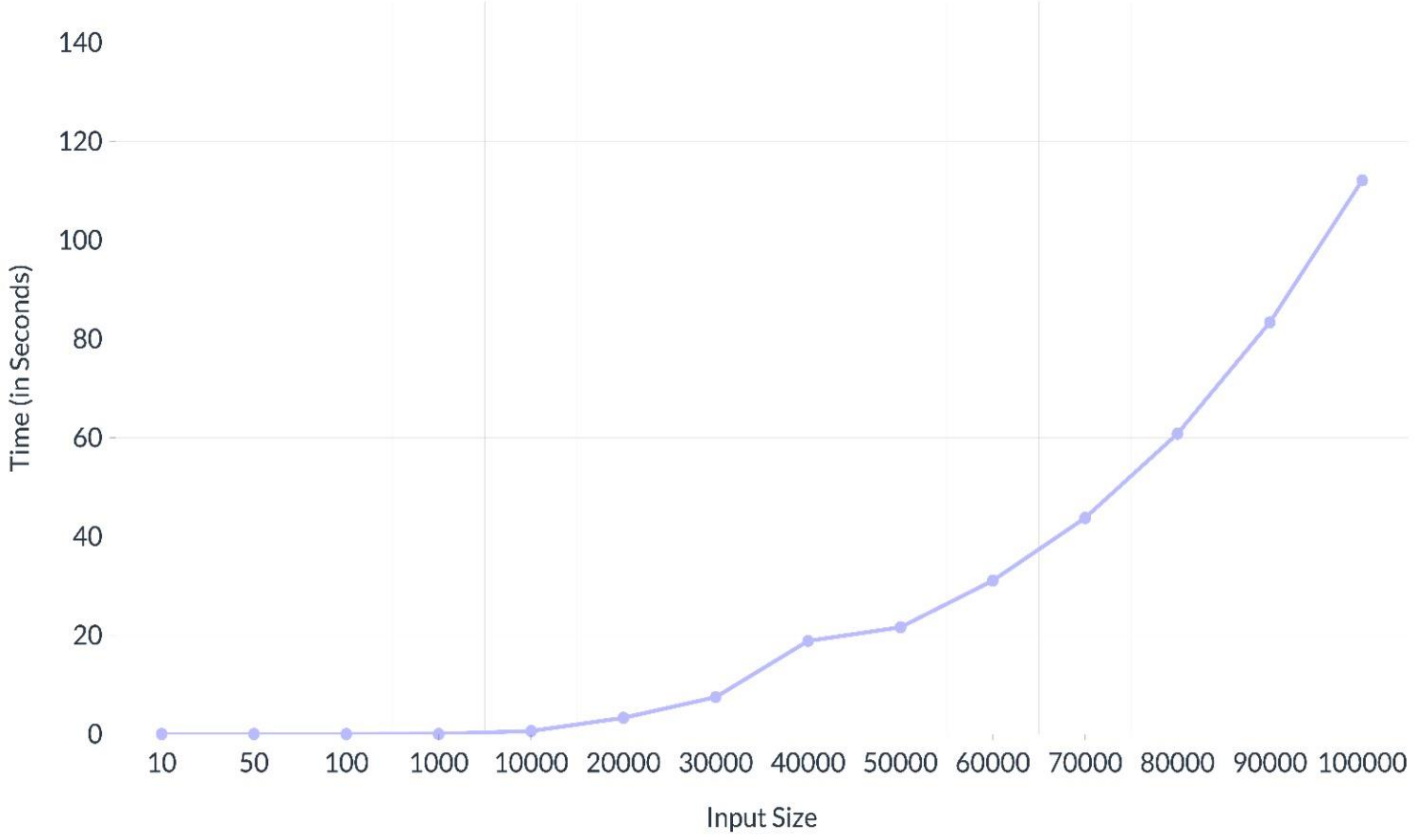
```
Length of s1: 100000 characters
Length of s2: 100000 characters


PARALLEL ALGORITHM:
Completed in : 41654.201035 ms
Length of LCS: 65315


SEQUENTIAL ALGORITHM:
Completed in : 94123.667241 ms
Length of LCS: 65315
```

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.

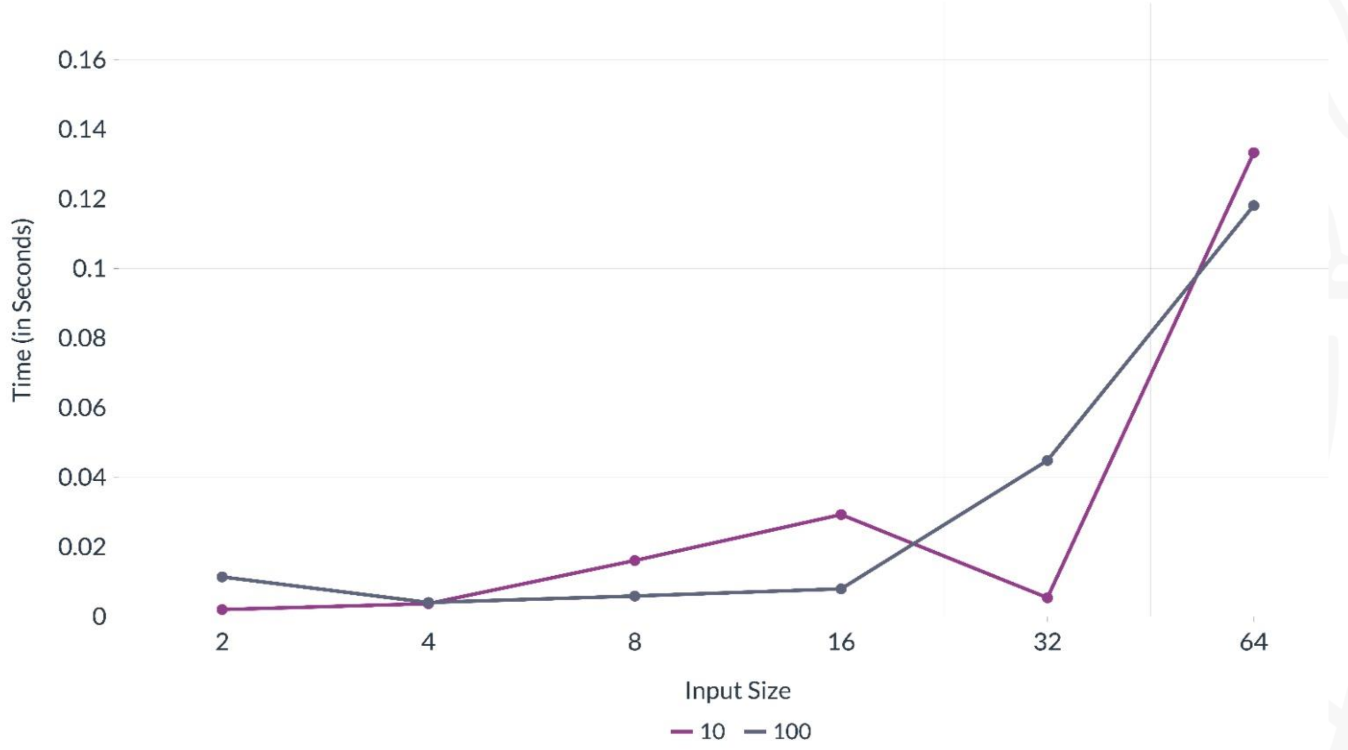
Results for Sequential Approach

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



Results for Parallel Approach (small input size)

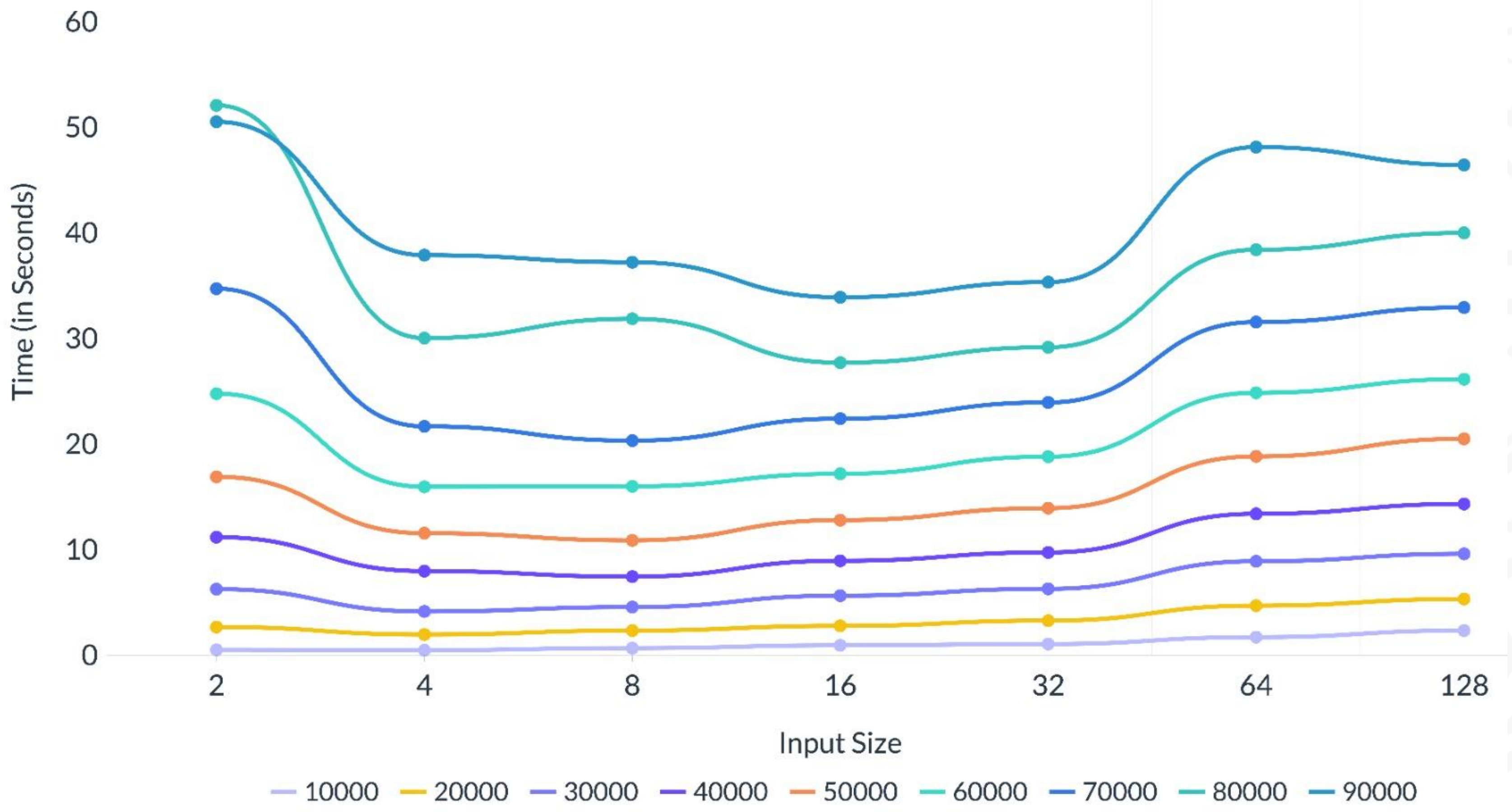
Number of Processors	Time (in s) for Input size 10	Time (in s) for Input size 100
2	0.00199635	0.01138207
4	0.00368789	0.0040272
8	0.01609147	0.00588514
16	0.02928861	0.00793963
32	0.00538751	0.04484502
64	0.13329603	0.11810985



Results for Parallel Approach (large input size)

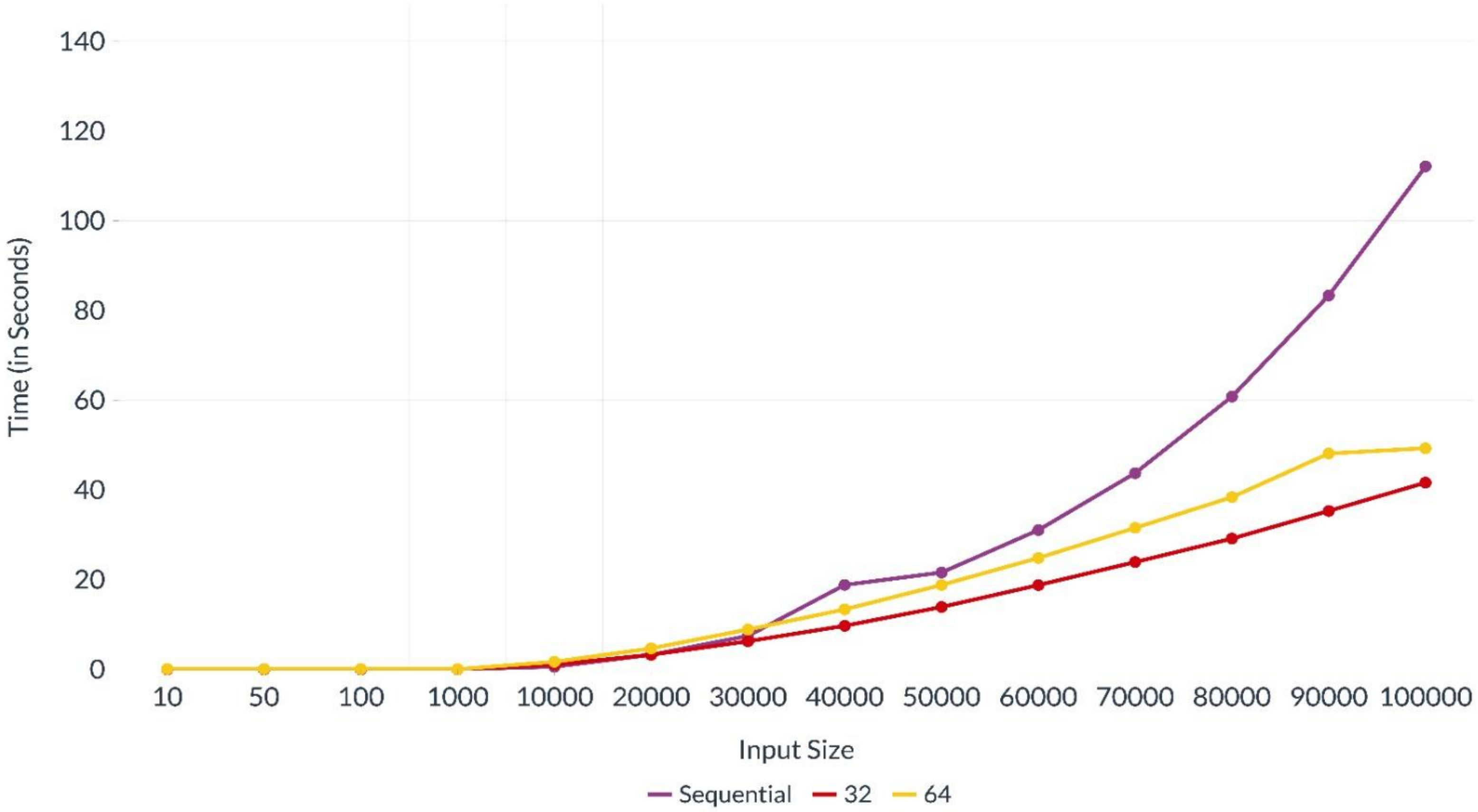
Number of Nodes	Time (in s) for Input size 10000	Time (in s) for Input size 20000	Time (in s) for Input size 30000	Time (in s) for Input size 40000	Time (in s) for Input size 50000	Time (in s) for Input size 60000	Time (in s) for Input size 70000	Time (in s) for Input size 80000	Time (in s) for Input size 90000	Time (in s) for Input size100000
2	0.52578763	2.681668937	6.268362552	11.19016013	16.90770924	24.78136885	34.72393444	52.09097353	50.5329981	107.9106947
4	0.488477363	1.973840537	4.153437258	7.979478955	11.56600832	15.96001254	21.68704922	30.04386666	37.89914124	43.24429724
8	0.685008492	2.345982661	4.569211543	7.454193071	10.88848518	16.00331206	20.3369828	31.8711065	37.23287834	37.23287834
16	0.945190889	2.7905281	5.650291473	8.953158803	12.8032219	17.19117304	22.4127304	27.71922095	33.90865398	39.99139587
32	1.072783828	3.284945664	6.297263918	9.744897762	13.93548397	18.81471166	23.94909021	29.17512206	35.3446107	41.65420104
64	1.720764667	4.705545232	8.90961194	13.40302478	18.84167263	24.86627807	31.5650644	38.41239001	48.13711712	49.30719847
128	2.54637876	5.332577586	9.61329776	14.33292146	19.75645659	26.14938322	32.94326906	40.0063902	46.44143647	53.22557095

Results for Parallel Approach (large input size)



Comparison of Sequential and Parallel Execution

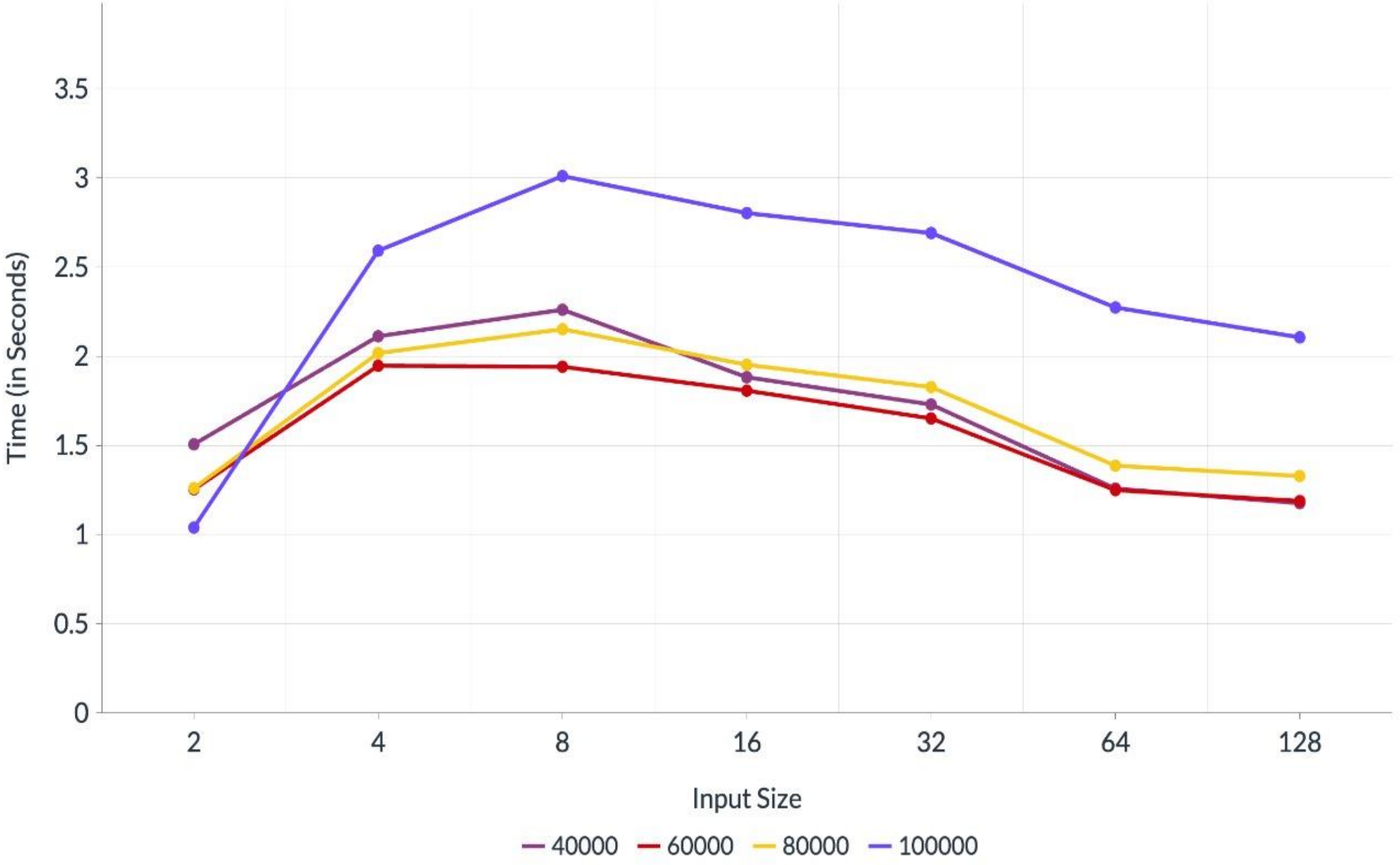
Here, I have compared sequential execution graph with graphs obtained using parallel execution on
32 Nodes and 64 Nodes.



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 = $T_{sequential} / T_{parallel}$



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

- http://personales.upv.es/thinkmind/dl/conferences/infocomp/infocomp_2011/infocomp_2011_7_20_10120.pdf
- 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!