LONGEST COMMON SUBSEQUENCE

Parallelizing LCS using OpenMP

Name : Saema Nadim

Person# 50469138

Instructor : Dr. Russ Miller

University at Buffalo The State University of New York

CONTENT

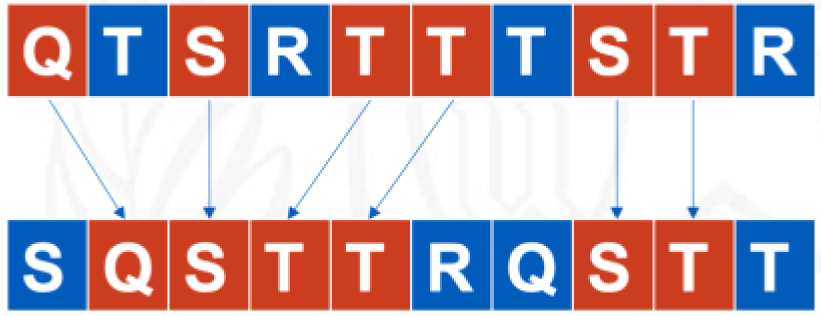
What is LCS?

- 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. 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.

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.

It's Applications

It has wide amount of real world applications:

1. Bioinformatics

   - it is used for finding similar regions of two nucleic acid sequences – like DNA

   - it helps analyse protein sequences to understand their structural and functional properties.

2. Text Comparison

   - it is essential in plagiarism detection software, helping maintain Academic Integrity.

   - it is used in version control systems like Git to track changes in code and text files.

3. Natural Language Processing (NLP):

   - It finds common text segments in multiple documents, facilitating text summarization.

   - It is also used in spell checkers.

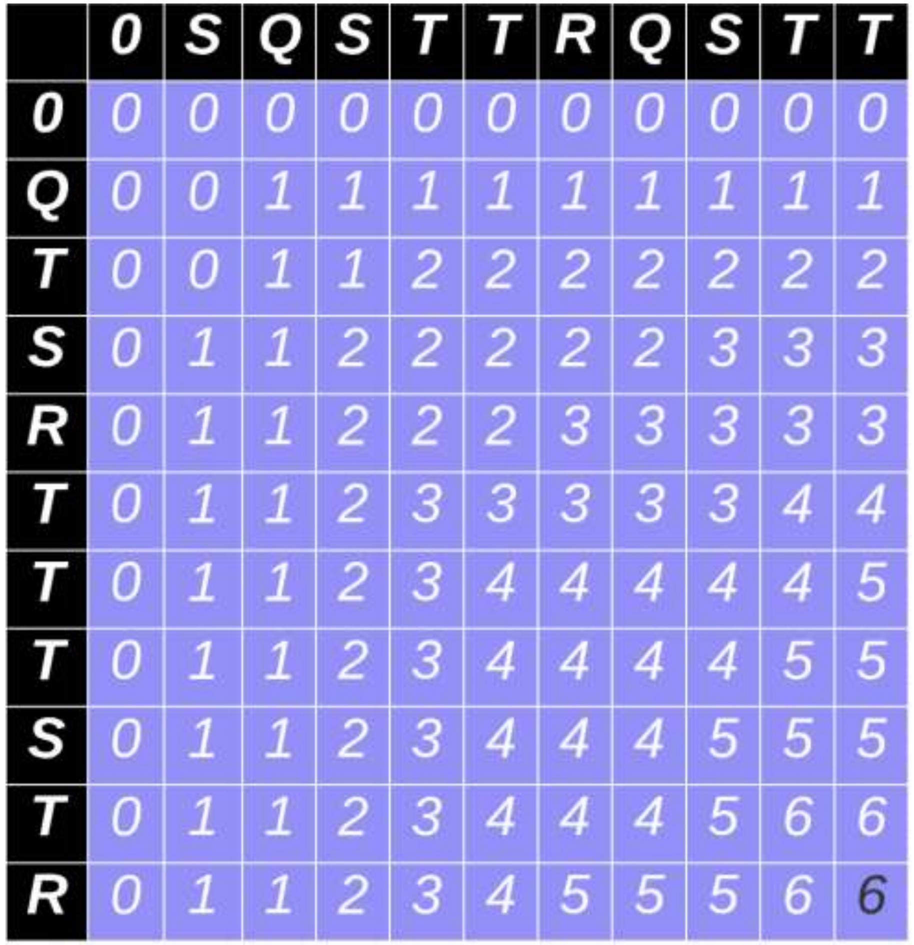
It is also used in multiple other fields like Pattern Recognition, Reinforcement Learning, Data compression, Data mining, Image comparison etc. Hence, it's a very valuable tool.

University at Buffalo The State University of New York

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.

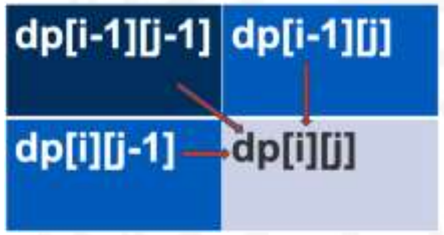
How to calculate LCS – explained (DP Approach)



	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

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

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

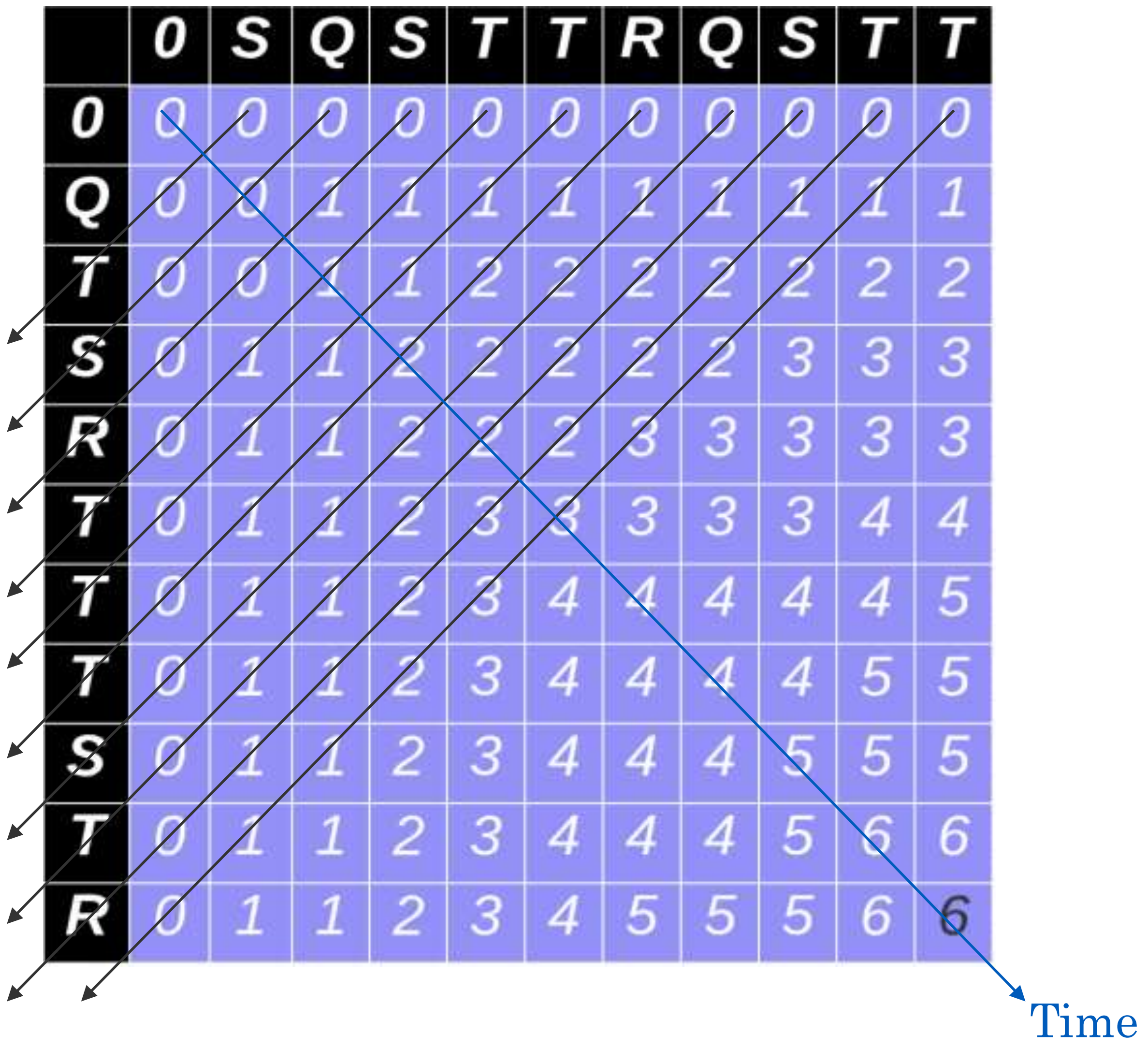


It can be seen that each element's value depends on its previous diagonals.

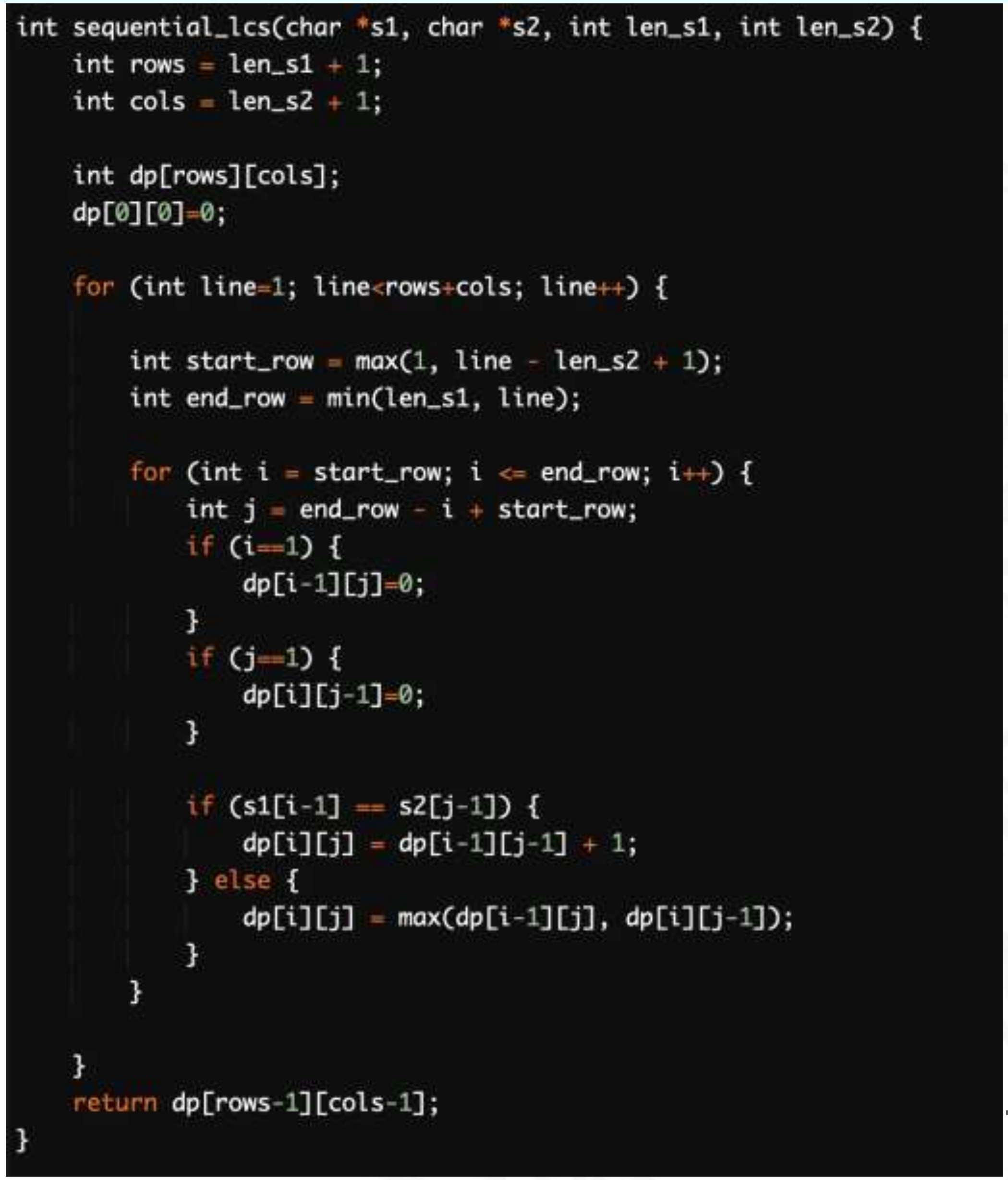
- The last bottom right value of the calculated matrix tells us the length of LCS, and the matrix can be traced back from the last element to find the required subsequence.

University at Buffalo The State University of New York

Sequential Approach



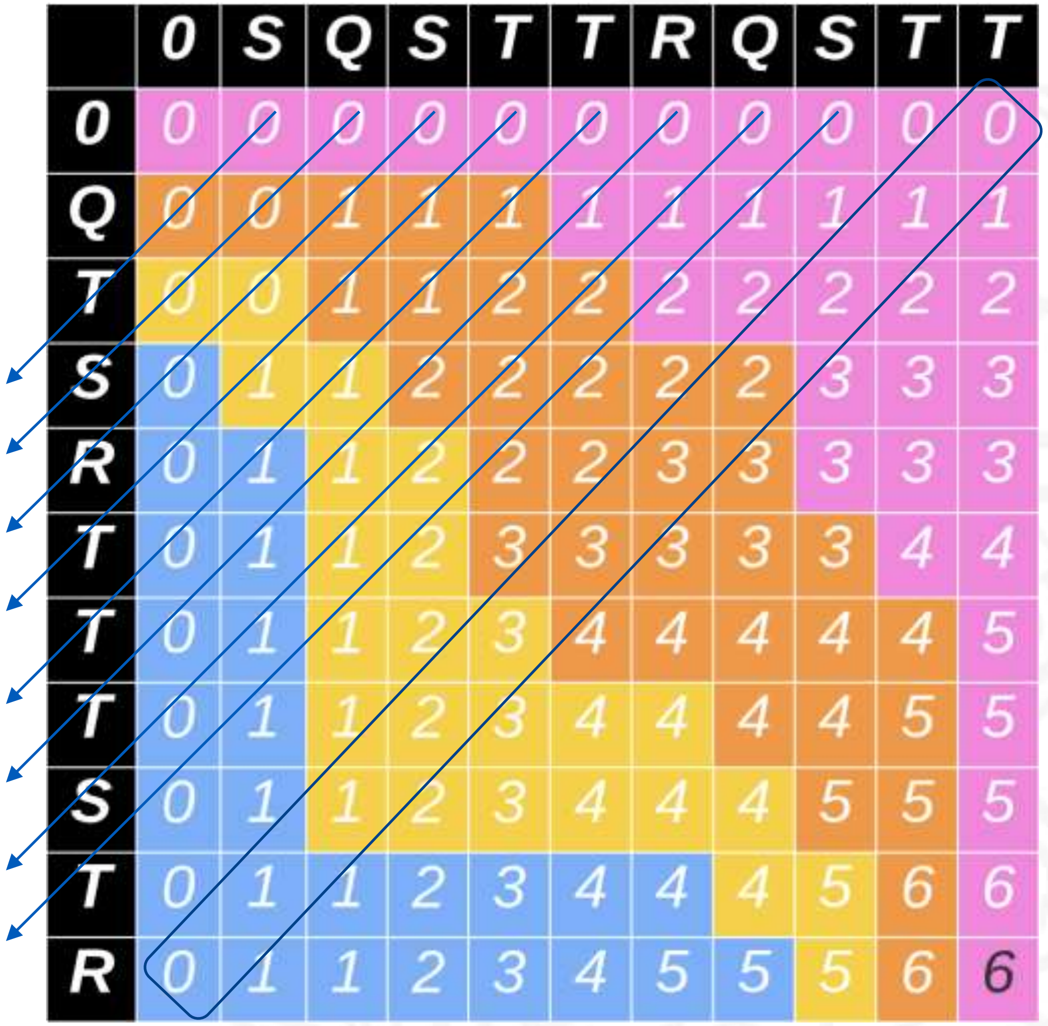
	0	S	Q	S	T	T	R	Q	S	T	T
0	0	0	0	0	0	0	0	0	0	0	0
Q	0	0	1	1	1	1	1	1	1	1	1
T	0	0	1	1	2	2	2	2	2	2	2
S	0	1	1	2	2	2	2	2	3	3	3
R	0	1	1	2	2	2	3	3	3	3	3
T	0	1	1	2	3	3	3	3	3	4	4
T	0	1	1	2	3	4	4	4	4	4	5
T	0	1	1	2	3	4	4	4	4	5	5
S	0	1	1	2	3	4	4	4	5	5	5
T	0	1	1	2	3	4	4	4	5	6	6
R	0	1	1	2	3	4	5	5	5	6	6



```
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];
}
```

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 threads using a simple formula.
- In this figure, 4 threads are used. Each color represents calculation by a single thread.
- Since threads use shared memory, a value calculated by one thread is visible to all threads, we don't need to use any send and receive functions in OpenMP



	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

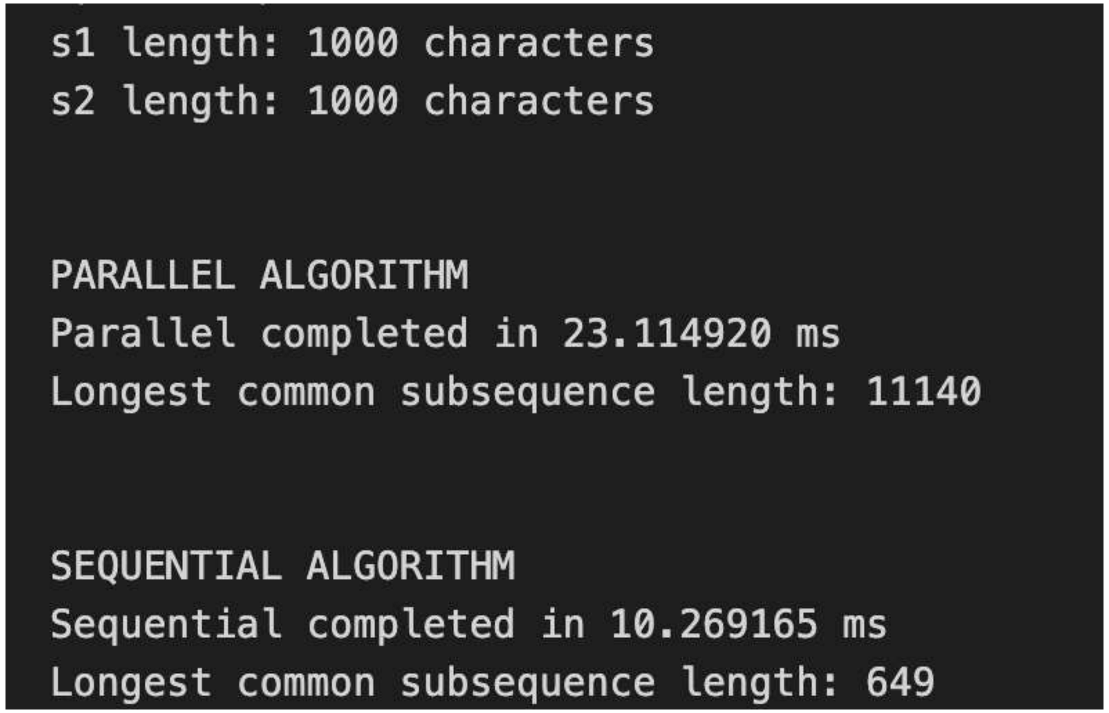
OpenMP

- OpenMP stands for Open Multi processing and it is completely different from MPI.
- In an HPC environment, MPI uses multiple nodes (distributed-memory) in the cluster and it allows processes on different nodes to communicate efficiently.
- OpenMP is well-suited for tasks that can be parallelized within a single node (shared-memory) with multiple CPU cores, where threads can easily communicate.
- It’s used for multi threaded parallelism.

University at Buffalo The State University of New York

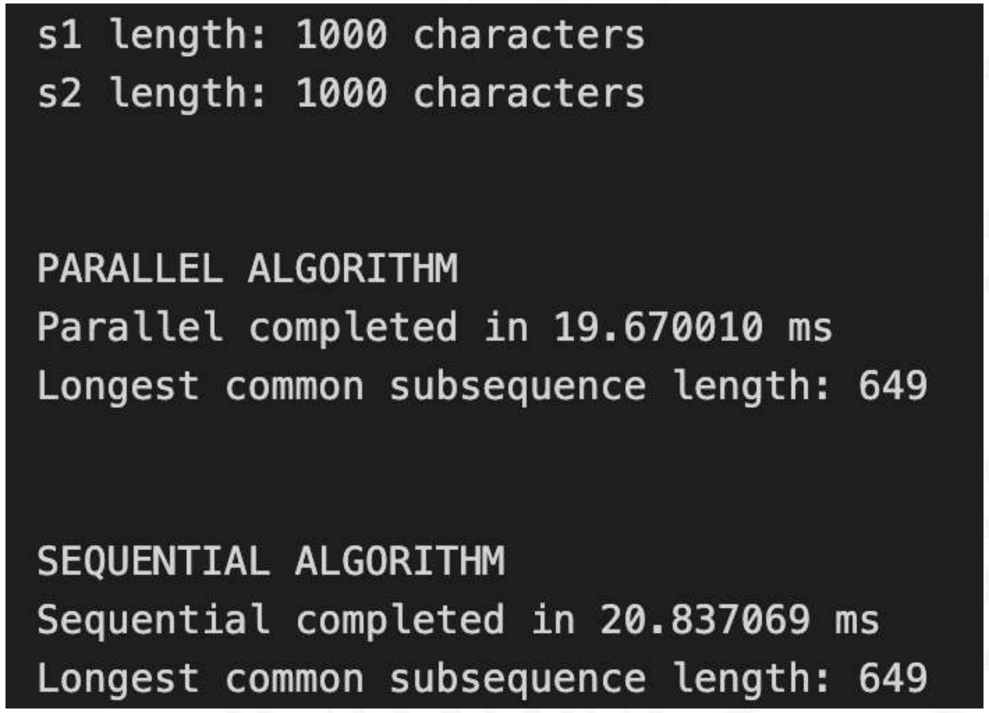
Output Screen

Did not use ‘private’:



```
s1 length: 1000 characters
s2 length: 1000 characters


PARALLEL ALGORITHM
Parallel completed in 23.114920 ms
Longest common subsequence length: 11140


SEQUENTIAL ALGORITHM
Sequential completed in 10.269165 ms
Longest common subsequence length: 649
```

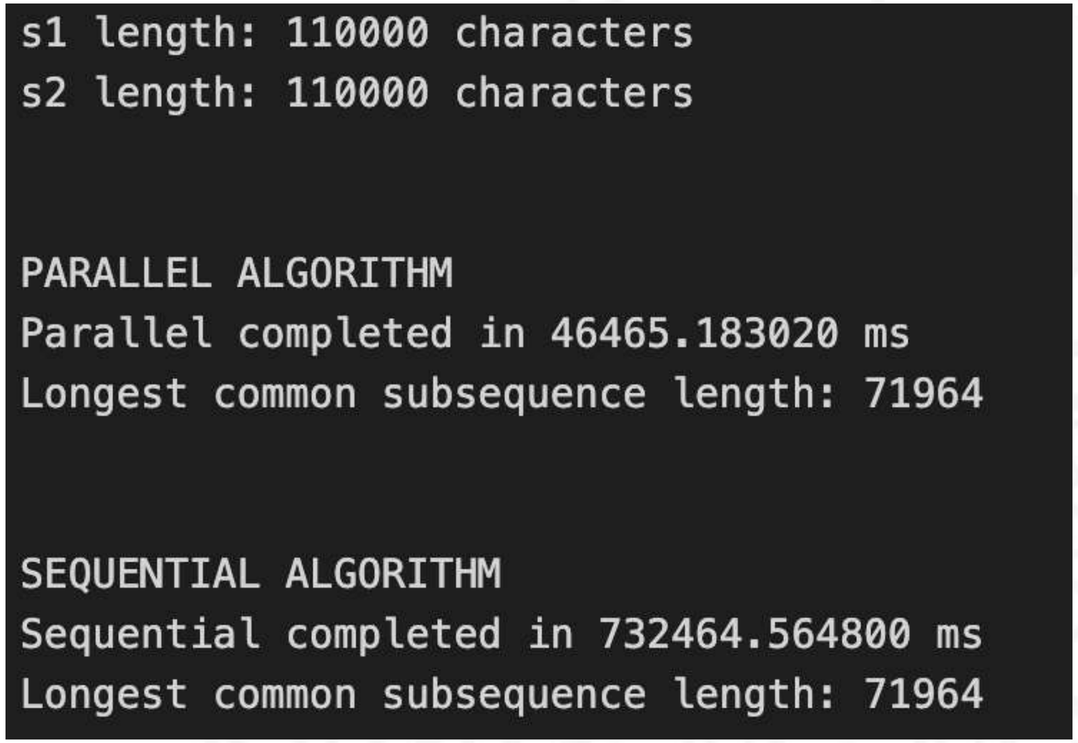
Correct Output:



```
s1 length: 1000 characters
s2 length: 1000 characters


PARALLEL ALGORITHM
Parallel completed in 19.670010 ms
Longest common subsequence length: 649


SEQUENTIAL ALGORITHM
Sequential completed in 20.837069 ms
Longest common subsequence length: 649
```

Output Screen

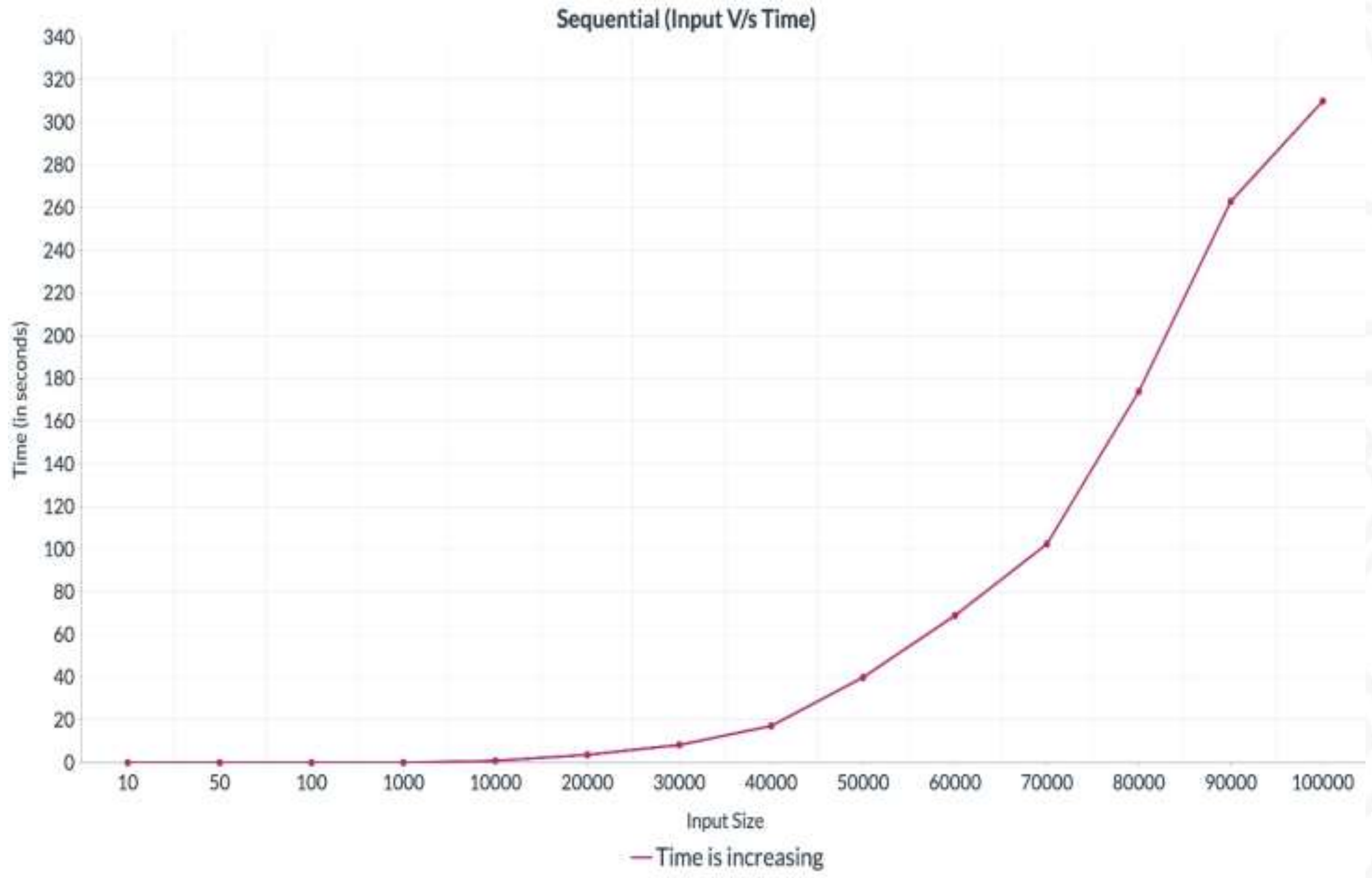
- Input length: 1,10,000
- Number of threads used: 16
- Parallel Algo time: 46 seconds (< 1 min)
- Sequential Algo time: 732 seconds (12.2 minutes)



```
s1 length: 110000 characters
s2 length: 110000 characters


PARALLEL ALGORITHM
Parallel completed in 46465.183020 ms
Longest common subsequence length: 71964


SEQUENTIAL ALGORITHM
Sequential completed in 732464.564800 ms
Longest common subsequence length: 71964
```

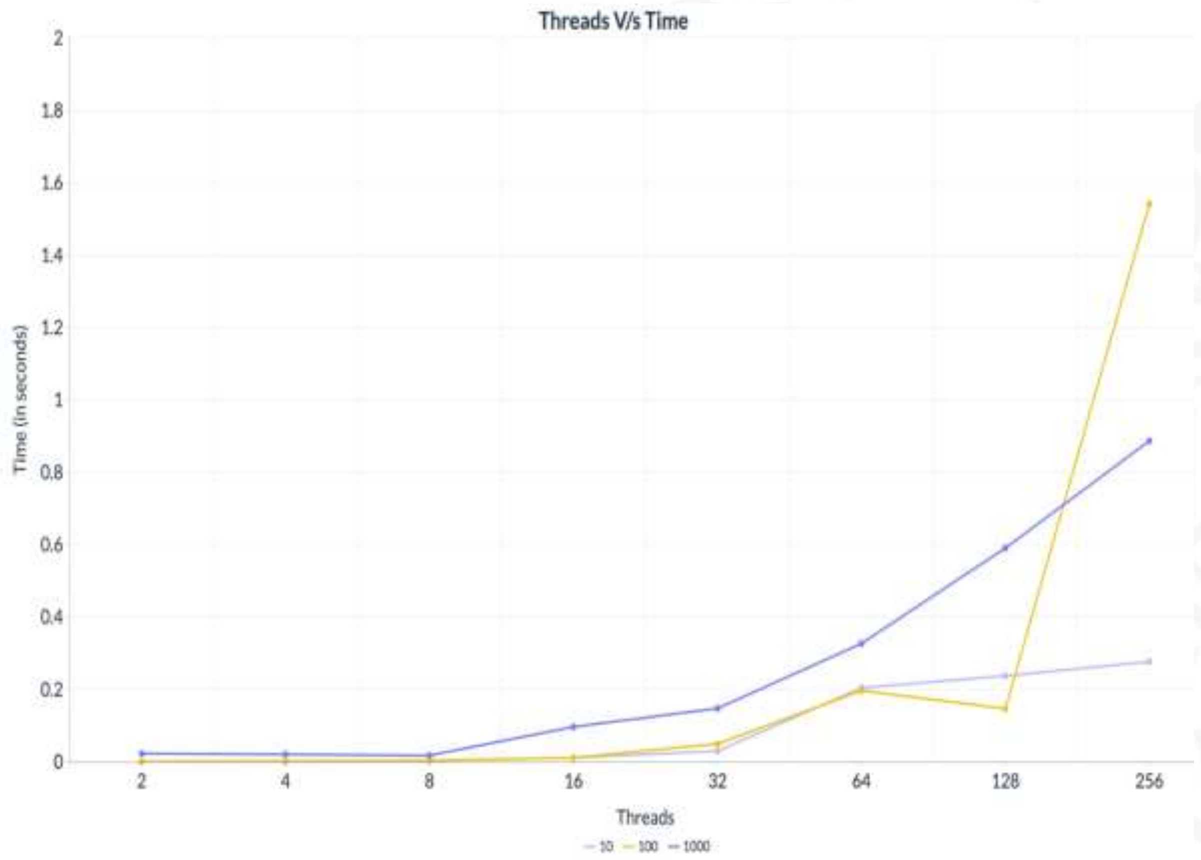
Results for Sequential Approach

Size of Input	Time (in s)
10	0.000003
50	0.00005
100	0.00015
1000	0.016
10000	0.888
20000	3.7
30000	8.34
40000	17.2
50000	40
60000	68
70000	102.5
80000	174
90000	263
100000	310



Results for Parallel Approach (small input size)

Number of Threads	Time (in s) for Input size 10	Time (in s) for Input size 100	Time (in s) for Input size 1000
2	0.0004	0.001	0.023
4	0.0009	0.0024	0.021
8	0.004	0.004	0.018
16	0.012	0.012	0.097
32	0.03	0.05	0.148
64	0.205	0.197	0.327
128	0.238	0.147	0.591

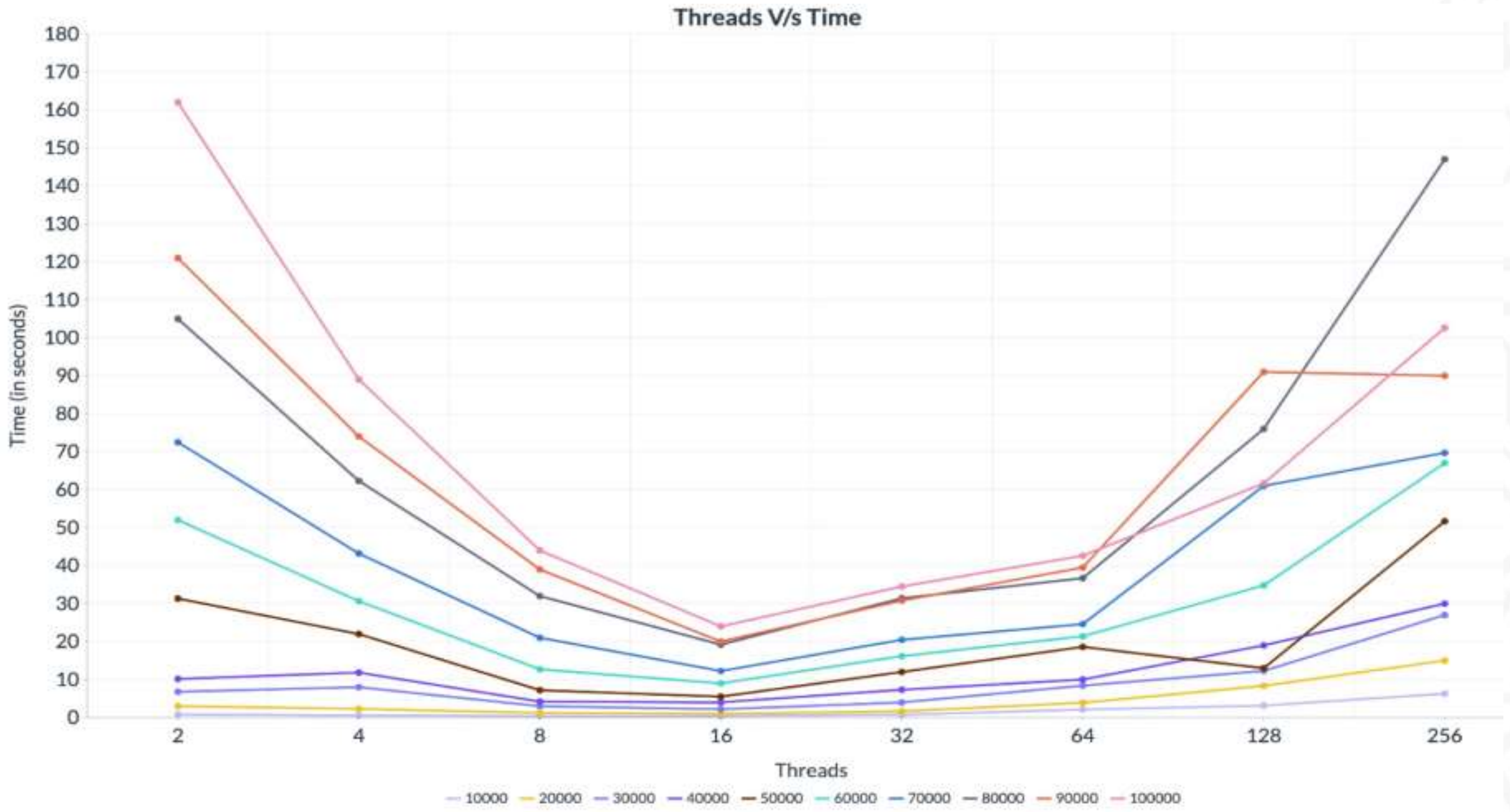


Results for Parallel Approach (large input size)

No. of Threads	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	0.827	3	6.8	10.17	31.34	52	72.5	105	121	162
4	0.565	2.28	8	11.84	22	30.65	43.2	62.3	74.03	89
8	0.4	1.2	3	4.2	7.2	12.7	21	32	39	44
16	0.4	1	2.2	4	5.5	9	12.25	19.2	20	24
32	0.754	1.651	3.982	7.331	12	16.18	20.467	31.41	30.8	34.5
64	2.126	3.9	8.4	10	18.6	21.4	24.6	36.7	39.5	42.6
128	3.15	8.345	12.28	19	13	34.8	61	76	91	61.65
256	6.3	15	27	30	51.7	67	69.7	147	90	102.6

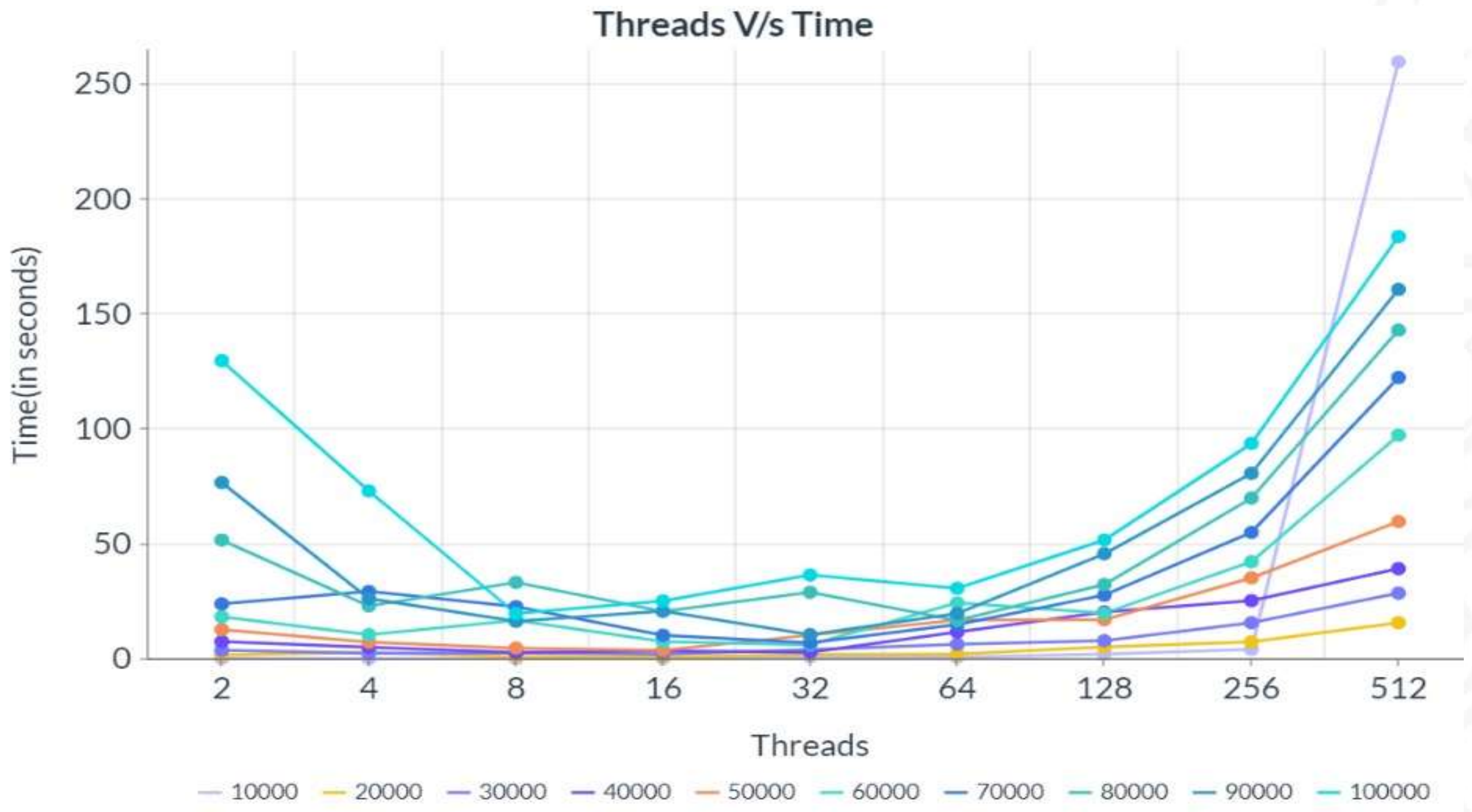
University at Buffalo The State University of New York

Results for Parallel Approach (large input size) (16 core node)



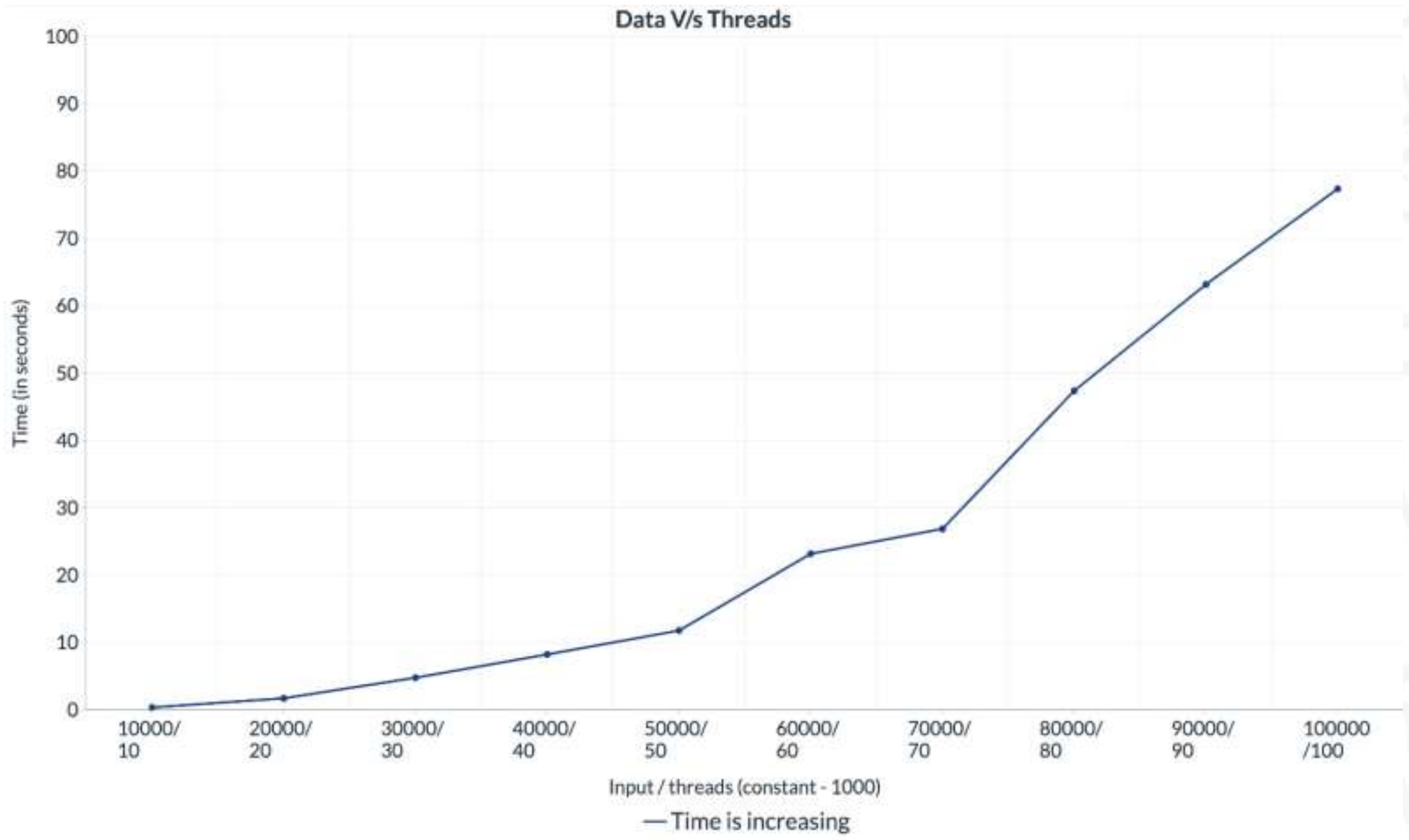
University at Buffalo The State University of New York

Results for Parallel Approach (large input size) (64 core node)



University at Buffalo The State University of New York

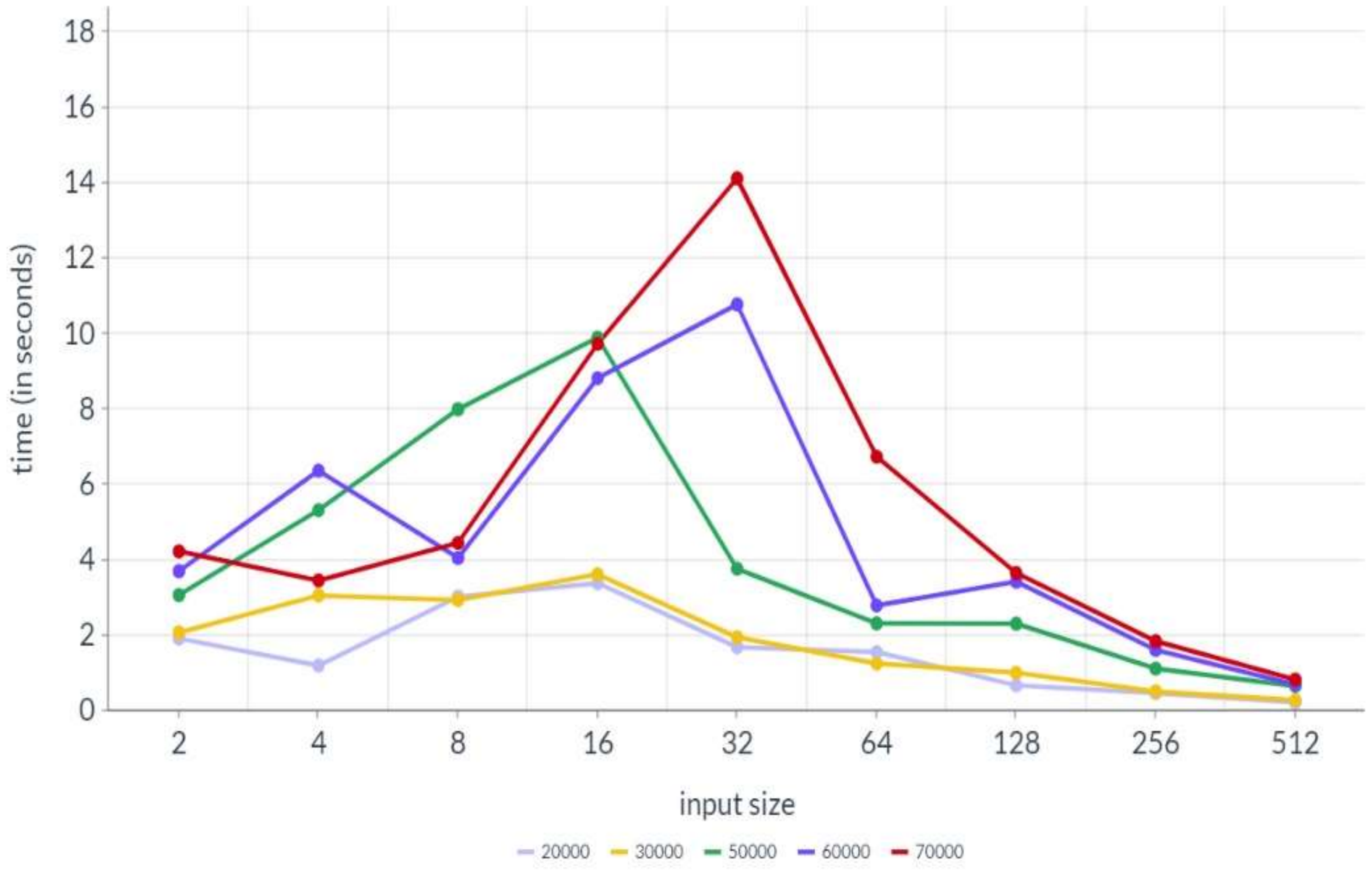
Graph where input to threads ratio is constant (1000)



Speedup Graph

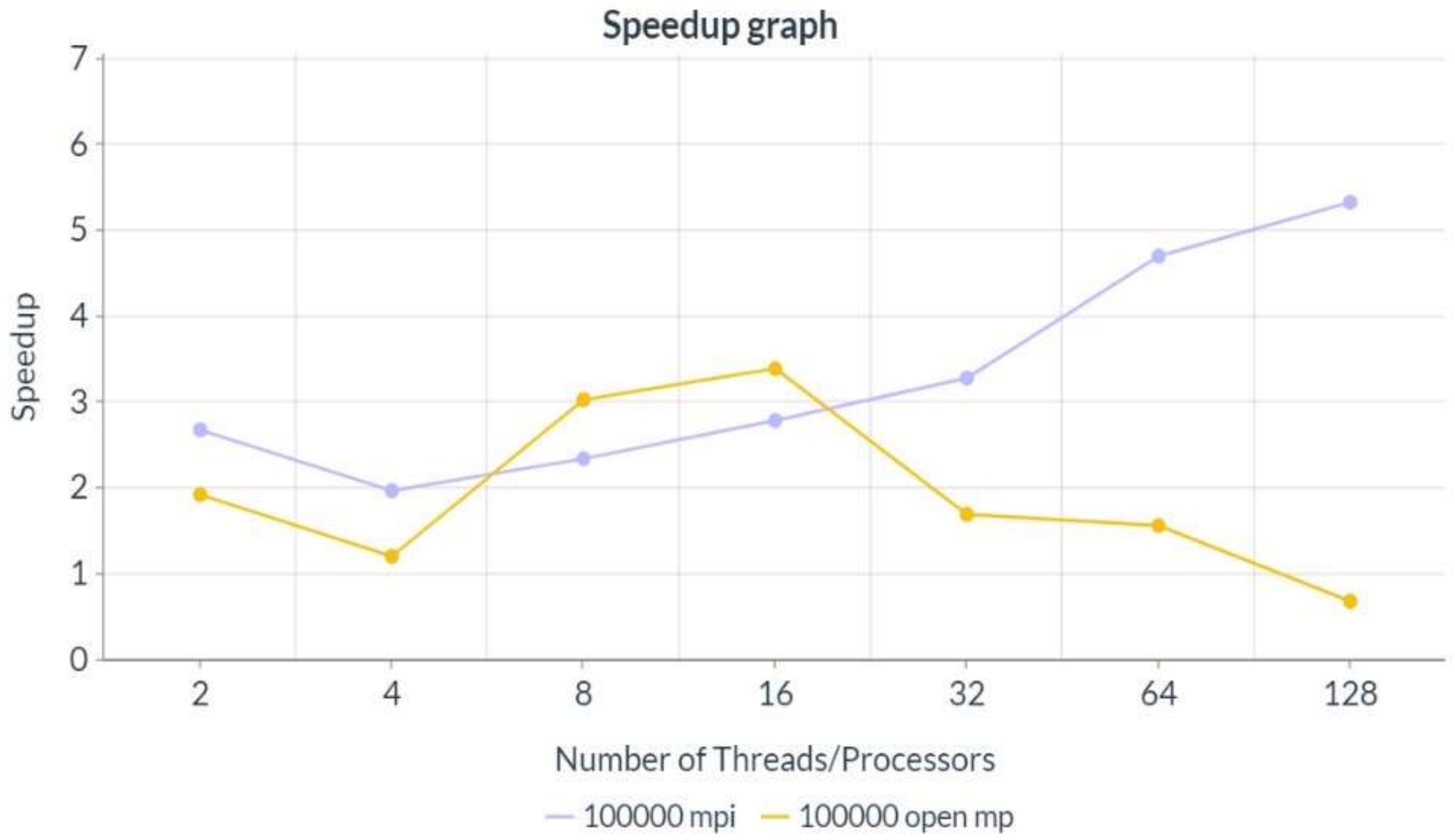
- Speedup is the execution time of a sequential program divided by the execution time of a parallel program that computes the same result.

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



University at Buffalo The State University of New York

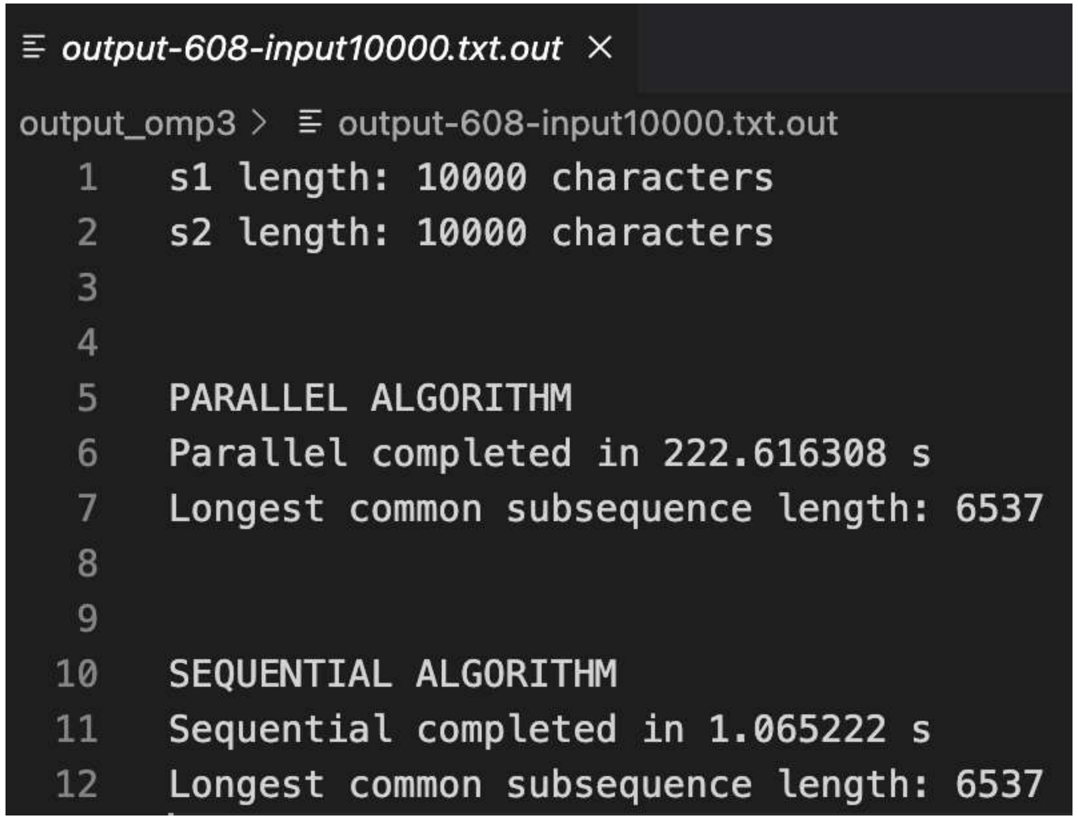
Comparison of MPI Speedup and OpenMP Speedup



Observations

- The sequential algorithm graph shows that as input increases, the time taken by sequential algorithm increases.
- It can be seen that for less input sizes, the graph pattern of parallel algorithm is similar to the sequential algorithm graph. The time increases in spite of increasing the number of threads.
- For large input sizes as the number of threads increase, the time taken by OpenMP algorithm decreases but till a certain point of time.
- After a point, time starts increasing again as the threads increase due to communication overhead between them.
- For larger inputs, the OpenMP implementation can be up to 12 times faster than the sequential algorithm.

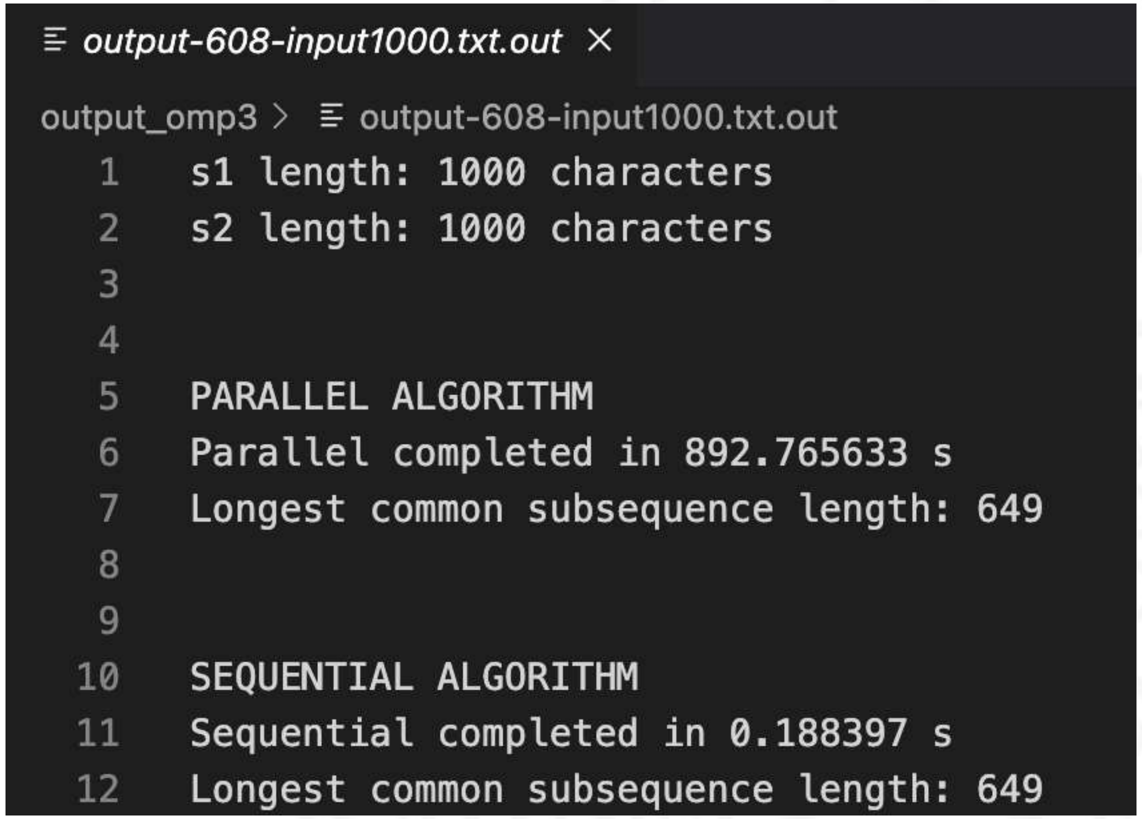
Max threads used: 608



output-608-input10000.txt.out ×

output_omp3 > output-608-input10000.txt.out

```
s1 length: 10000 characters
s2 length: 10000 characters


PARALLEL ALGORITHM
Parallel completed in 222.616308 s
Longest common subsequence length: 6537


SEQUENTIAL ALGORITHM
Sequential completed in 1.065222 s
Longest common subsequence length: 6537
```



output-608-input1000.txt.out ×

output_omp3 > output-608-input1000.txt.out

```
s1 length: 1000 characters
s2 length: 1000 characters


PARALLEL ALGORITHM
Parallel completed in 892.765633 s
Longest common subsequence length: 649


SEQUENTIAL ALGORITHM
Sequential completed in 0.188397 s
Longest common subsequence length: 649
```

Time taken: 3.7 min

Time taken: around 15 min

References

- https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf
- https://buffalo.app.box.com/s/8ynupd5rg3cl91dzbjovsxw92s4e1olu
- https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-functions?view=msvc-170
- 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

Thank You