Longest Common Subsequence

CS 359 Project

Course coordinator: Dr. Surya Prakash

Atche Sravya - 170001013

B Rushya Sree Reddy - 170001014

(CSE 2019)

Introduction

The LCS problem deals with comparing two sequences and finding the maximum length subsequence which is common to the two given sequences. The LCS algorithm is widely used in many areas, which includes the field of gene engineering to compare DNA of patients with that of healthy ones.

For solving the LCS problem, we resort to dynamic programming approach. Due to the growth of database sizes of biological sequences, parallel algorithms are the best solution to solve these large size problems.

Goals

1. Implementing sequential and parallel dynamic programming algorithms for Longest common subsequence problem using optimum number of processors.
2. Comparing parallel algorithm with sequential algorithm.
3. Analysing Time complexity, Speed up, Work and Cost efficiency of the parallel algorithm.

Problem Statement :

Given two sequences, find the length of the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Sequential Approach for LCS

Naive Approach : The naive approach is to generate all the subsequences of the given strings and check for the common subsequence with maximum length. This brute force approach would take exponential time complexity of **O(n * (2^n)).**

-- O(2^n) comparing the 2^n possible subsequences of each string with those of other string.

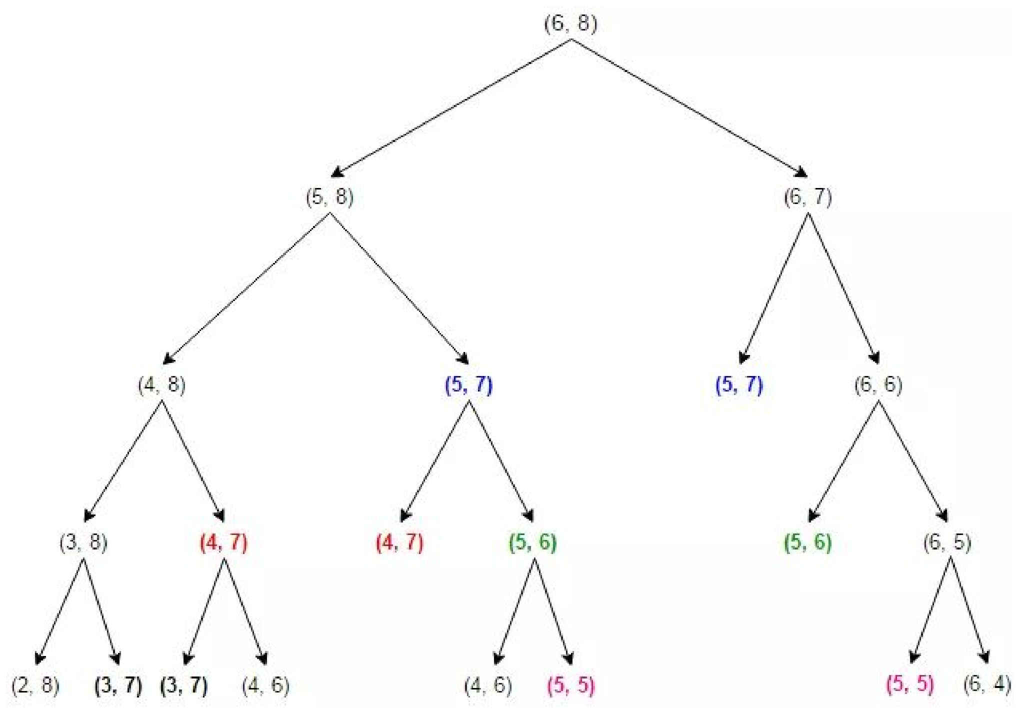
-- O(n) to check if two subsequences are equal at each comparison.

Recursive Approach : Let L[X[0,...,n-1],Y[0,...,m-1]] be the length of the longest common subsequence of strings X and Y of lengths n,m respectively.

Algorithm:

```
Lcs (X,Y,n,m)
If m == 0 OR n == 0 then
    return 0

If X[n-1] == Y[m-1]
    return 1 + Lcs(X,Y,n-1,m-1)
Else
    return max( Lcs(X,Y,n-1,m) , Lcs(X,Y,n,m-1) )
```

Here, Time complexity is O(2^n) in the worst case, when all characters of the given strings mismatch. If we draw the recursive tree, we find that this implementation has **overlapping substructure** property as shown in the below figure where two strings of length 6 and 8 whose lcs is of length 0 is taken as an example.



We can avoid this recomputation of same subproblems using memoization or tabulation using **Dynamic Programming**.

Dynamic Programming : Here, we use a bottom up approach, where we calculate the smaller values of LCS(i,j) first and then build larger values using them.

Algorithm : L[n+1,m+1] is the table storing the required Lcs values used in bottom-up approach.

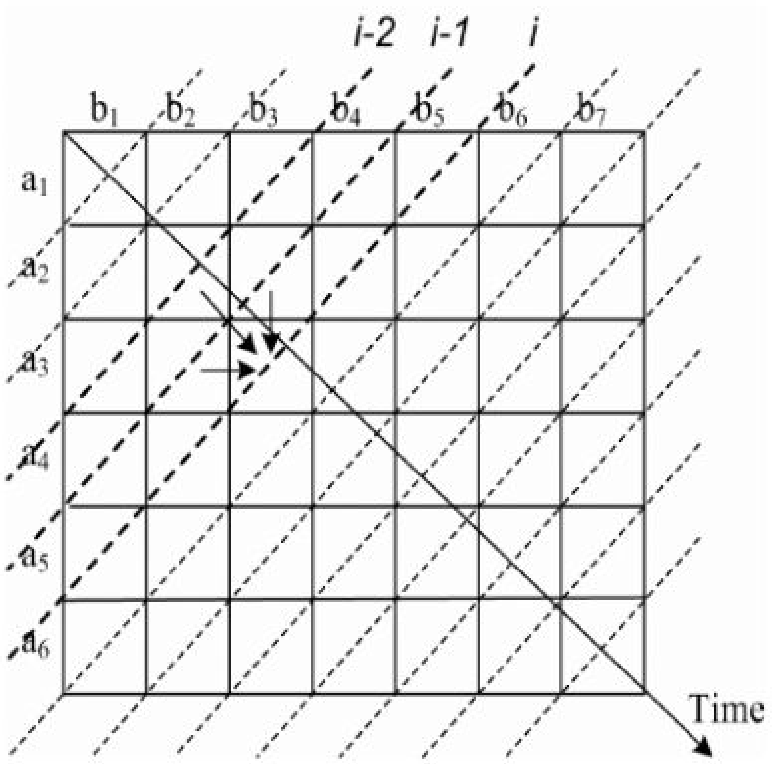
```
Lcs:   For i : 0 to n
            For j : 0 to m
                If  i == 0 OR j == 0 then
                L[i,j] = 0
                Else if X[i-1] == Y[j-1] then
                L[i,j] = 1 + L[i-1,j-1]
                Else
                L[i,j] = max( L[i-1,j] , L[i,j-1] )
           End for
        End for
        Return L[n][m]
```

Here, The time and space complexity is **O(n*m),** which is the best sequential implementation known for the longest common subsequence problem.

Parallel Approach for LCS

In the calculating the memoization table in dynamic approach for each entry L[i,j] we need L[i-1,j-1], L[i,j-1] and L[i-1,j] values. In other words L[i,j] depends on the data in the same row and same column . So, the entries of the same row, column or diagonal can't be computed in parallel.

In bottom-up approach we first compute L[1,1], then L[2,1], L[1,2] and then L[3,1], L[2,2], L[1,3] and so on.Here, we notice that entries of same anti-diagonal can be computed in parallel. So, to parallelize the dynamic programming algorithm, we have to fill memoization table in anti-diagonal direction.



Algorithm :

Parallel_lcs: Dp[i][j] stores the length of lcs of X[0,...,i] and Y[0,...,j]

```
For i=0,j=0 ; i<n,j<m ; j++                    //no of anti diagonals
   diagnoal_size = min( j , n-i )
        Par for k=0 ; k < diagonal_size ; k++          // for each anti diagonal element
                a = i+k
                b = j-k

                If (a == 0 OR b == 0)
                        Dp[a][b] = 0
                Else if ( x[a-1] == y[b-1] )
                        Dp[a][b] = Dp[a-1][b-1] + 1
                Else
                        Dp[a][b] = max( Dp[a-1][b] , Dp[a][b-1] )

        End for

        If(j >= m)
        j = m-1, i++
End for
Return Dp[n][m]
```

Here, the time complexity is O(n), since we are iterating through anti diagonals, which are max(m,n) in number.

Comparison of Sequential and Parallel Approach for LCS

Sequential Approach :

Time complexity - O(n*m)

Space complexity - O(n*m)

Parallel Approach :

Time complexity - O(n)

Space complexity - O(n*m)

Max number of processors used - O(n)

Work complexity - O(n*m)

As the work complexity of parallel approach is same as the time complexity of sequential approach, we can say that the parallel algorithm is **Work efficient**.

SpeedUp = Sequential execution time / Parallel execution time
= O(n*m) / O(n)

SpeedUp = O(n)

Cost = Parallel time complexity * no . of processors used
= O(n) * O(n)

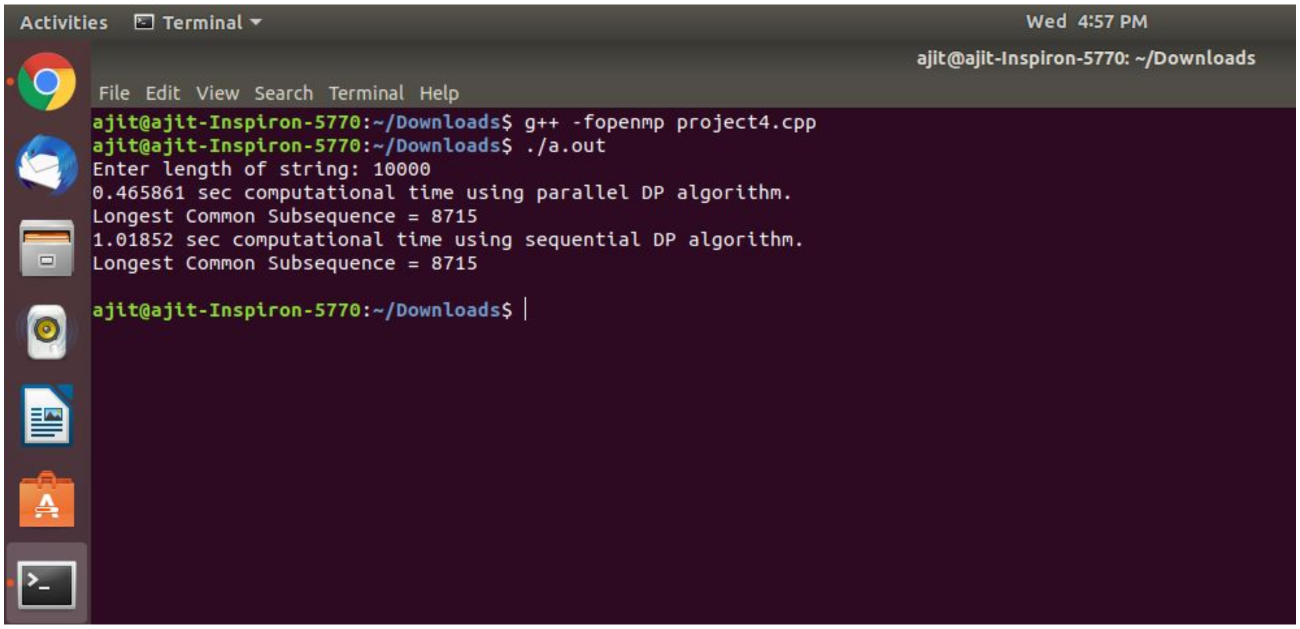
As the cost of parallel algorithm is of the same order as the time complexity of sequential algorithm , it is **Cost Efficient.**

Cost = O(n*n)

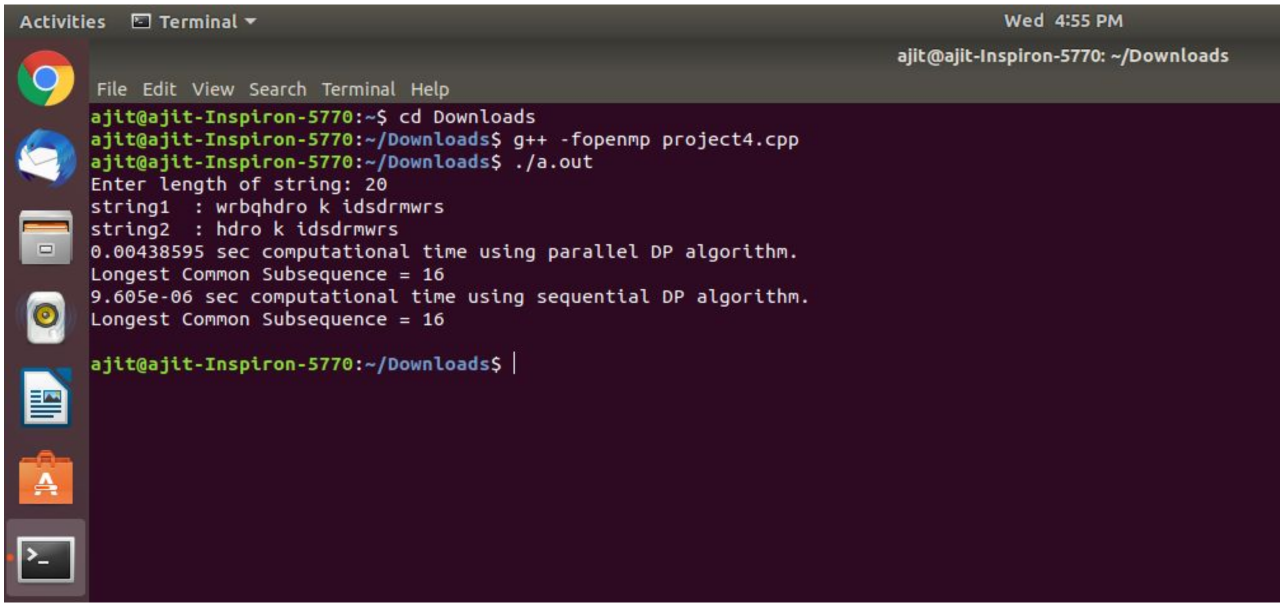
Efficiency = SpeedUp / no . of processors used.
= O(n) / O(n)

Efficiency = O(1)

Execution of Sequential and Parallel programs

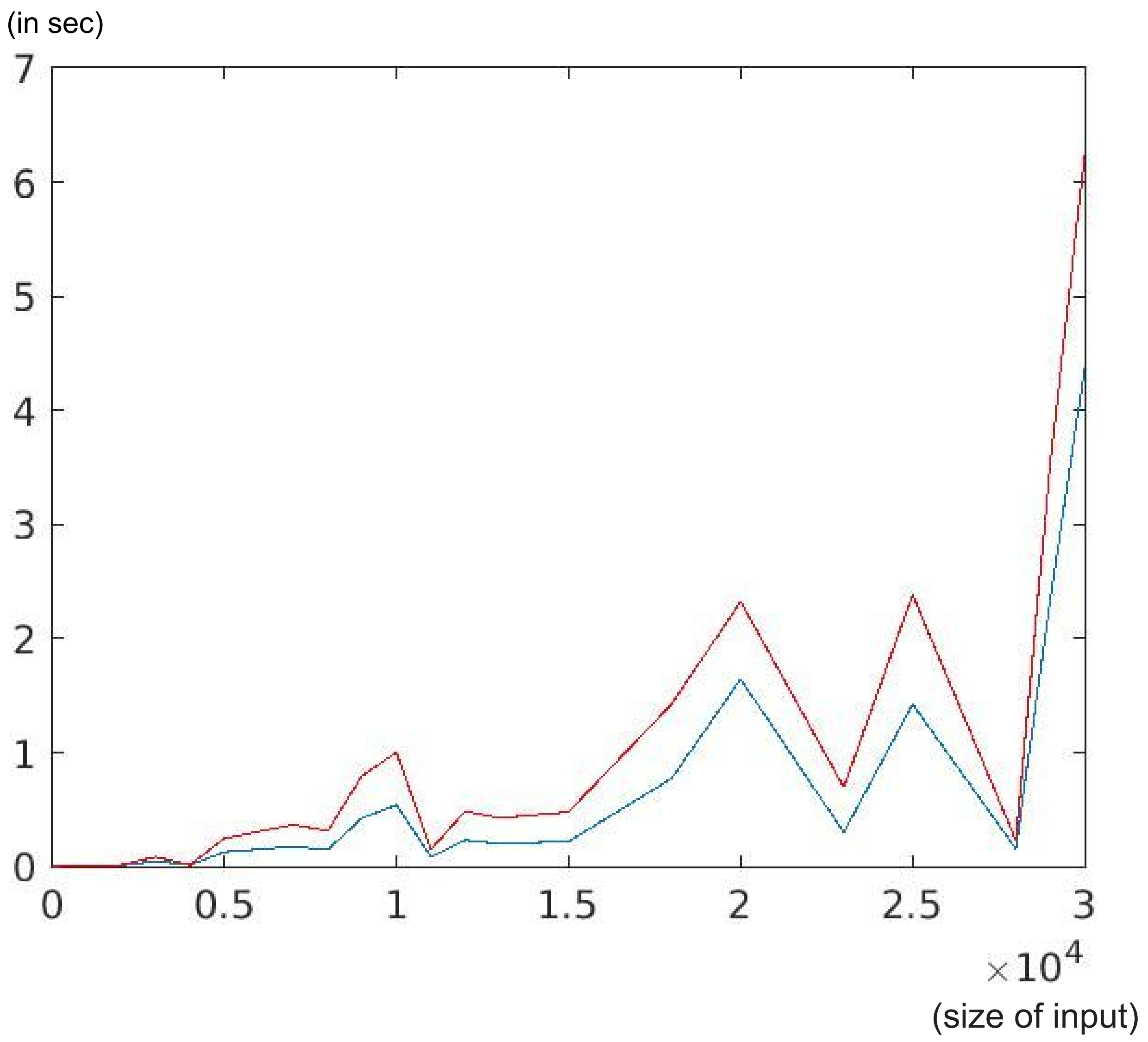

Activities
Terminal
Wed 4:57 PM
ajit@ajit-Inspiron-5770: ~/Downloads
File Edit View Search Terminal Help

```
ajit@ajit-Inspiron-5770:~/Downloads$ g++ -fopenmp project4.cpp
ajit@ajit-Inspiron-5770:~/Downloads$ ./a.out
Enter length of string: 10000
0.465861 sec computational time using parallel DP algorithm.
Longest Common Subsequence = 8715
1.01852 sec computational time using sequential DP algorithm.
Longest Common Subsequence = 8715

ajit@ajit-Inspiron-5770:~/Downloads$ 
```


Activities
Terminal
Wed 4:55 PM
ajit@ajit-Inspiron-5770: ~/Downloads
File Edit View Search Terminal Help

```
ajit@ajit-Inspiron-5770:~$ cd Downloads
ajit@ajit-Inspiron-5770:~/Downloads$ g++ -fopenmp project4.cpp
ajit@ajit-Inspiron-5770:~/Downloads$ ./a.out
Enter length of string: 20
string1  : wrbqhdro k idsdrmwrs
string2  : hdro k idsdrmwrs
0.00438595 sec computational time using parallel DP algorithm.
Longest Common Subsequence = 16
9.605e-06 sec computational time using sequential DP algorithm.
Longest Common Subsequence = 16

ajit@ajit-Inspiron-5770:~/Downloads$ 
```

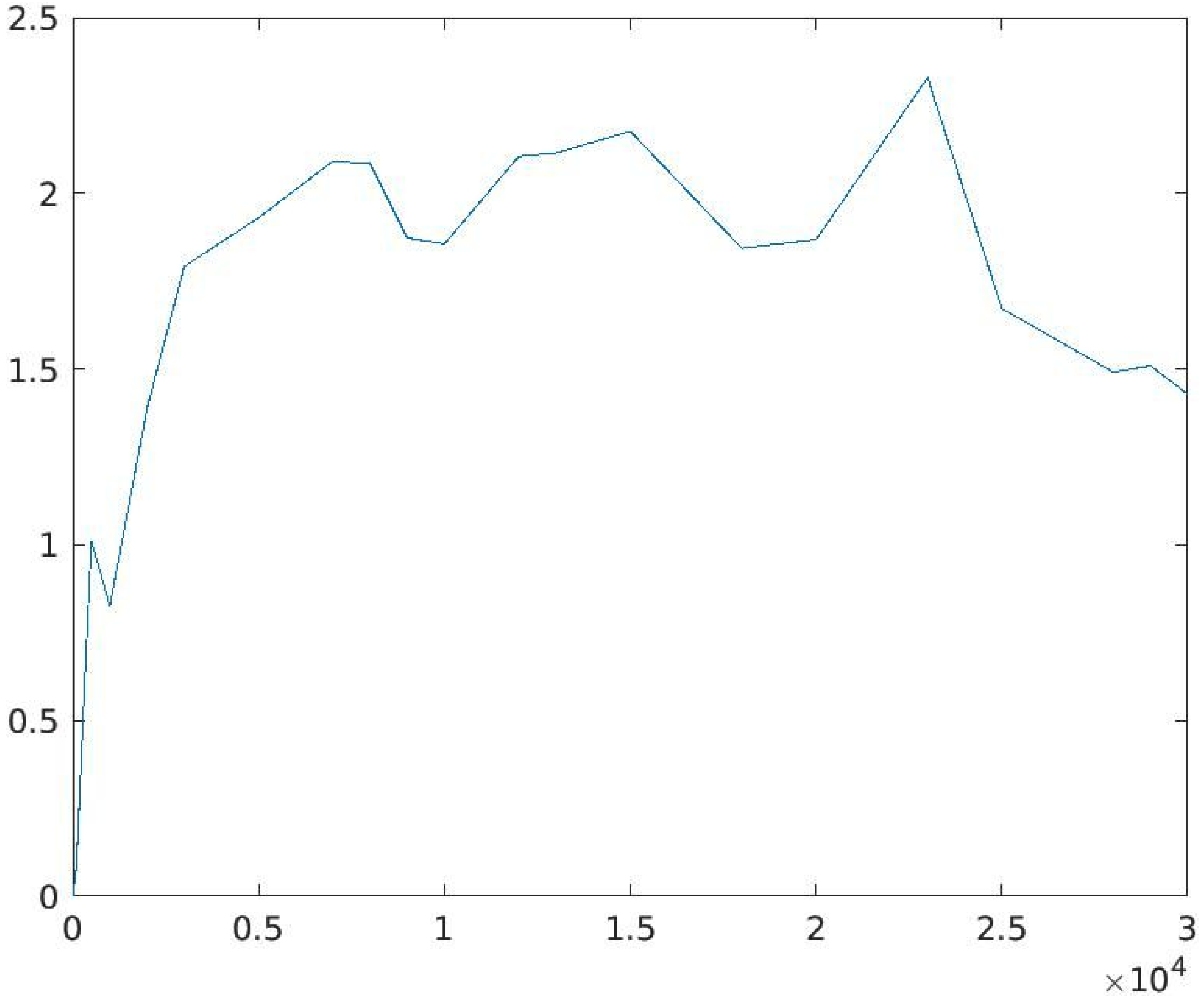
Computation-Time plots for sequential and parallel approaches :



🟥 : Computation time for Sequential Algorithm

🟦 : Computation time for Parallel Algorithm

Graph for Speed Up



(size of input)

Conclusion

We have focused on improving the time complexity of finding the Longest common subsequence using a parallel algorithm which is both work and cost efficient.

From the graph, we can observe that the speedup value is around 2, i.e., the parallel program is **twice** as fast as sequential one.

References

- https://www.irjet.net/archives/V3/i6/IRJET-V3I6183.pdf
- https://www.researchgate.net/publication/252320985_Parallel_Computing_the_Longest_Common_Subsequence_LCS_on_GPUs_Efficiency_and_Language_Suitability
- https://link.springer.com/content/pdf/10.1007%2F11424857_37.pdf
- http://www.iaeng.org/publication/WCE2010/WCE2010_pp499-504.pdf

The End