A Parallel Implementation Of Longest Common Sub-sequence Using Yang's Algorithm

Farhana Zaman Glory
Student ID : 7856215

Final Project Report

Course Title : Advances in Parallel Computing

COMP 7850

Winter 2019

Department of Computer Science
University of Manitoba
Winnipeg, MB R3T 2N2, Canada
gloryfz@myumanitoba.ca

Contents

Abstract

Sequence alignment is an important problem in computational biology and finding the longest common subsequence (LCS) of multiple biological sequences is an essential and effective technique in the sequence alignment. The longest common subsequence (LCS) is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences). A major computational approach for solving the LCS problem is dynamic programming. Several dynamic programming methods have been proposed to have reduced time and space complexity. As databases of biological sequences become larger, parallel algorithms become increasingly important to tackle large size problems. In this regard, Jiaoyun Yang gave an algorithm for parallel implementation of LCS in 2010. So, as my course project for COMP 7850 course, I have implemented this algorithm in MPI, OpenMP and also the hybrid of both to solve this problem. From experiments, I have found out that this algorithm gives much faster results for input sequences than the results from the inputs using sequential algorithm.

1. Introduction

In the computational methods, biological sequences are represented as strings and finding the longest common subsequence (LCS) is a widely used method for sequence alignment. The LCS problem commonly refers to finding the longest common subsequence of two strings, whereas for three or more strings it is called multiple longest common subsequence problem (MLCS) [3]. Dynamic programming is a classical approach for solving LCS problem, in which a score matrix is filled through a scoring mechanism. The best score is the length of the LCS and the subsequence can be found by tracing back the table. Let m and n be the lengths of two strings to be compared. The time and space complexity of dynamic programming is O(mn). Many algorithms have been proposed to improve the time and space complexity. In 2010 Yang et al. [11] presented an efficient parallel algorithm for the LCS problem based on the dynamic programming algorithm. So as my term project for COMP 7850 course, I have implemented their efficient algorithm in MPI, OpenMP and also the hybrid of them. From my experiments, I am able to show that, all of them outperform the sequential approach and the hybrid outperforms the other two parallel approaches.

This report consists of five sections. Section 2 provides notation and related background along with related works. In section 3 I will briefly talk about the problem statement. Parallel architecture design, cost analysis and methodology have been discussed in Section 5 and 4 respectively. Then the evaluation of the experimental results will be discussed in section 6. In the end, I will conclude the report with some concluding remarks 7.

2. Background

In [8], Myers and Miller applied a divide-and-conquer technique [2] to solve the LCS problem and the space complexity of their algorithm is O($m + n$) while the time complex remains to be O(mn). In [7], Masek and Paterson presented some new techniques to reduce the time complexity to O($n^2/logn$). Parallel algorithms have been developed to reduce execution time through parallelization in diagonal direction [9] and bit-parallel algorithms [4]. Bit-parallel algorithms depend on machine word size and are not good for general processors. Aluru, Futamura and Mehrotra [5] proposed a parallel algorithm using prefix computation for general sequence alignment problem with time complex O($mn = p + \tau(m + p)logp + \mu mlogp$) Efficient parallel algorithms based on dominant point approaches have also been proposed for general MLCS problem [1]. However, these algorithms are usually slow for two-string LCS problems. In recent years, graphics processing units (GPUs) have become a cost-effective means in parallel computing and have been widely used in bioinformatics as hardware accelerators. Some traditional algorithms have been implemented on GPUs and achieved significant speedups. Manavski and Giorgio Valle also implemented the Smith-Waterman algorithm on the GPU [6]. Then comes Yang's algorithm [11] for solving the LCS problem. In their algorithm they have changed the data dependency in the dynamic programming table so that the cells in the same row or the same column of the dynamic table could be computed in parallel. The algorithm uses O(max(m, n)) processors and its time complexity is O(n). So, from O(mn), the time complexity has been reduced to only O(n) by Yang's algorithm.

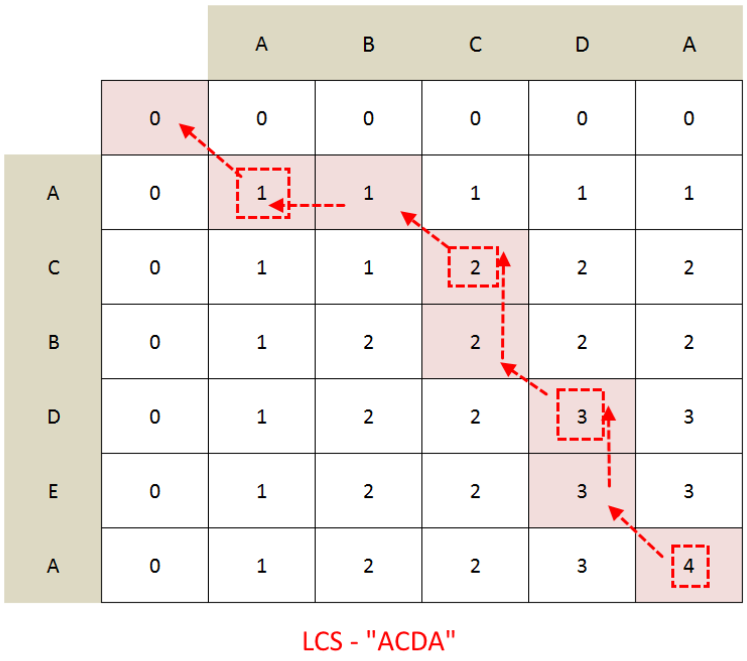
3. Problem statement and Objectives

The classical method for the LCS problem is the Smith-Waterman algorithm [10] which is based on the dynamic programming. If two input strings a_1 a_2 a_3......a_n and b_1 b_2 b_3......b_m are given, we have to construct a score matrix S of size $(n + 1)(m + 1)$, in which $S[i, j]$ $(1 \leq i \leq nand1 \leq j \leq m)$ records the length of the longest common subsequence for substrings a_1 a_2 a_3......a_i and b_1 b_2 b_3......b_j. The entries of the table will be filled by values from the following recurrence formula:

$$S[i,j] = \begin{cases} 0 & i \text{ or } j \text{ is } 0 \\ S[i-1, j-1] + 1 & a_i = b_j \\ max(S[i-1, j], S[i, j-1]) & \text{o.w.} \end{cases}$$

The LCS score table of input strings ABCDA and ACBDEA and the resultant LCS is shown in figure 1.

The main operation of the dynamic programming algorithm is filling the score table. If m



		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

Figure 1. Score table of LCS problem.

and n are the lengths of two strings to be compared, the time and space complexity of dynamic programming is O$(m * n)$. The main issue with the sequential algorithm of DP is in the score table S[i, j] is totally dependent on three entries: S$[(i-1),(j-1)]$, S$[(i-1),j]$, and S$[i,(j-1)]$, as shown in Figure 2. In other word, S[i, j] depends on the data in the same row and the same column and thus the same row or same column data cant be computed in parallel.

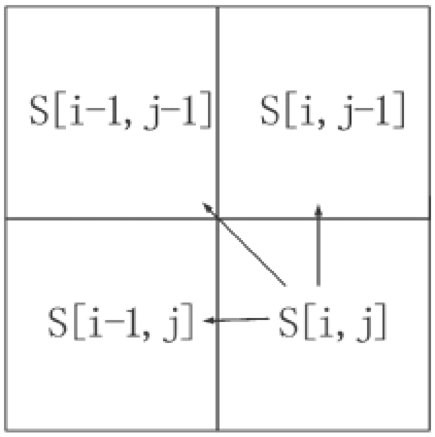


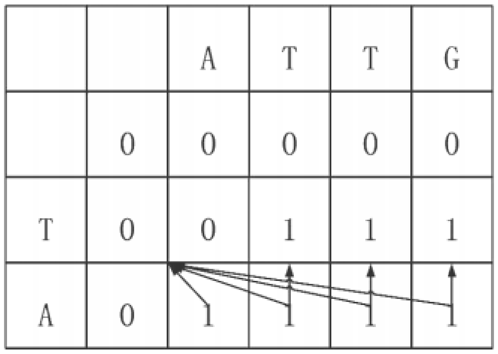
Figure 2. Dependency of the scores.

An apparent way to parallelize the dynamic programming algorithm is to compute the score table in the diagonal direction. Its disadvantage is that the workload of different processors is different and imbalanced. In order to compute the same row data or same column data in parallel, the data dependence needs to be changed. So, Yang's algorithm has removed this dependency issue in a different way and solved the LCS problem. The objective of this project is to implement and analyze the performance of the parallel version of the existing LCS problem from both practical and theoretical perspective.

4. Methodology

4.1. Yang's Method of parallel LCS:

Yang et al. [11] changed the dependency of score values in the score table. As we know that a score S[i, j] is dependent of max(S[$i-1, j$], S[$i, j-1$]) when $a_i \neq b_j$. They have decomposed the complete dependency into two parts: column level and row level. So in their algorithm, they formulated two tables instead of one. One is the pivot table and the another one is the final score table. With the inputs, they also took another sequence C in which all the common characters of the input characters were contained in. Then for one input sequence and the C sequence they calculated the pivot table and here scores used only the dependency of previous columns. Then after determining the pivot table, using its score values and the input sequences, they formulated the final score table only using the dependency of the previous row's values. Let's have a look at the final row level dependency from figure 3.



		A	T	T	G
	0	0	0	0	0
T	0	0	1	1	1
A	0	1	1	1	1

Figure 3. Final score table.

So the equations of determining the scores of both the pivot table and score table is a little bit changed from the sequential algorithm. In their changed score table's equation, there is no calculation of column values which was already handled in the pivot table's values.

$$P[i,j] = \begin{cases} 0 & \text{if } j = 0 \\ j & \text{if } b_{j-1} = C[i] \\ P[i,j-1] & \text{o.w.} \end{cases}$$

$$S[i,j] = \begin{cases} 0 & \text{if } i \text{ or } j = 0 \\ max(S[i-1,j],0) & \text{if } P[c,j] = 0 \\ max(S[i-1,j], S[i-1,P[c,j]-1]+1) & \\ & \text{o.w.} \end{cases}$$

4.2. Yang's Algorithm:

```
1. For $i = 1$ to $l$ par-do
     For $j = 1$ to $m$
       Calculate $P[i,j]$ according to
     End for
   end for

2. For $i = 1$ to $n$
     For $j = 1$ to $m$ par-do
       $t$ = the sign bit of $(0 - P[c,j])$.
       $s$ = the sign bit of $(0 - (S[i-1,j] - t * S[i -$
   $1, P[c,j]-1]))$.
       $S[i,j] = S[i-1,j] + t * (s \oplus 1)$.
     End for
   End for
```

5. Architecture Design

5.1. Design Analysis:

For implementing Yang's algorithm in the Message Passing Interface I have designed the flow of the work this way:

At first, the input sequences are read from the files. Then, another array C is taken containing all the alphabets common in both the sequences. After that, the first operation is to compute the pivot table. We know that, in MPI, block decomposition macros decomposes an input in a balanced way so that load imbalance doesn't take place. So, these macros decompose input sequence 1 into some blocks in a balanced way and send them to the processors. Then the root processor will decompose input sequence 2 into blocks and send each of them to individual processors. It means the root is sending every column of the score table or each character of input sequence 2 block-wise to the processors. Now the processors have both the blocks of input sequence 1 and 2. Then

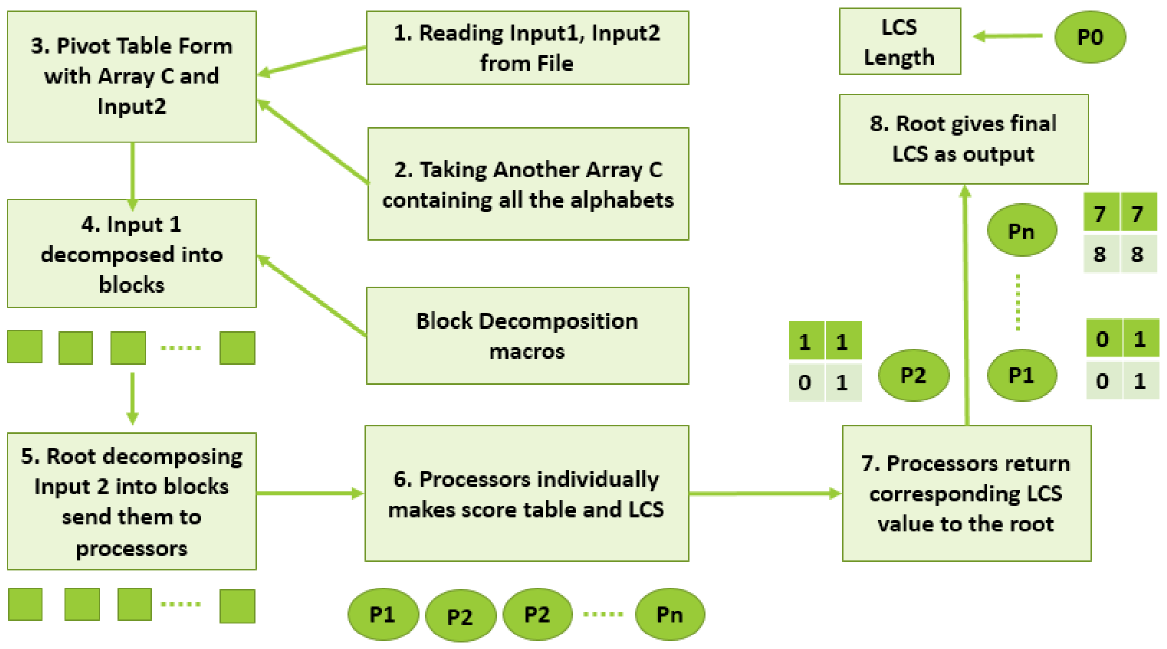

3. Pivot Table Form with Array C and Input2
1. Reading Input1, Input2 from File
2. Taking Another Array C containing all the alphabets
4. Input 1 decomposed into blocks
Block Decomposition macros
5. Root decomposing Input 2 into blocks send them to processors
6. Processors individually makes score table and LCS
7. Processors return corresponding LCS value to the root
8. Root gives final LCS as output
LCS Length
P0
Pn
P2
P1
P1
P2
P2
Pn
1 1
0 1
7 7
8 8
0 1
0 1

Figure 4. Flow of MPI Design.

the processors will calculate corresponding score table and LCS and after that they will send their corresponding results to the root processor. In the end, the root processor will show the final LCS score as output.

For OpenMP implementation, I have implemented “pragma omp parallel” at the beginning of calculating the pivot table. I have wanted to generate this table as fast as possible and so at the beginning of the second for loop I have implemented again “pragma omp parallel for”. Not only this case but also during generating the final score table I also have implemented “pragma omp parallel for” in the beginning of the second for loop. After implementing this way, my code has become much faster and is able to solve LCS problem a hundred times faster than the sequential version. My hybrid version is just the combination of these two architectures.

5.2. Cost Analysis:

For MPI implementation, the cost analysis of Yang’s LCS Algorithm will be: For normal sequential LCS problem, run time cost ts is O$(m * m)$. If in LCS problem, there are p number of processors, input strings have length m and n, then according to Yang’s algorithm every processor computes partial score table and its cost is n/p (computation depends only on rows). So, p number of processors will do this computation in $(p * (n/p))$ or n time. And also cost for computing the pivot table is also $(p * n/p)$ or n. In communication, the root processor sends some data to processors and processors give back the results to the root. So total communication cost for two way is $2(m/p + n/p)$. In this way, the total cost for parallel algorithm, tp is = O$(2 * n + 2(m/p + n/p))$

= O$((2*n*p+2*m+2*n)/p)$

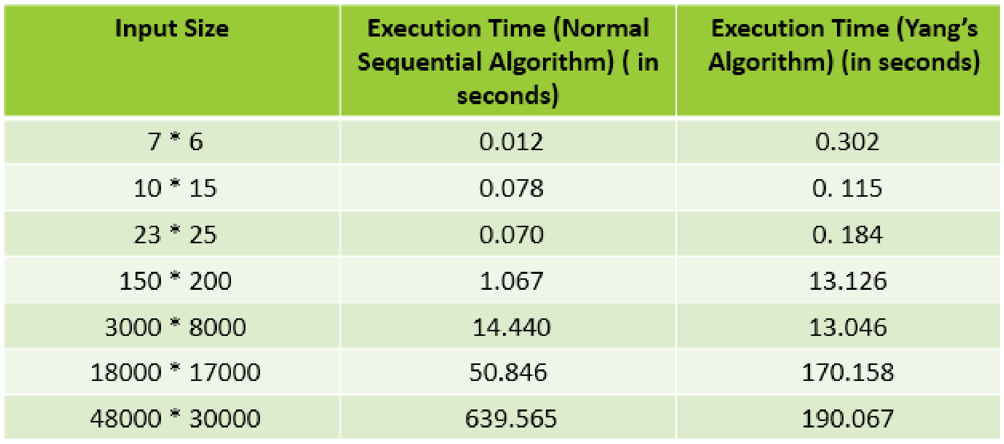
Now, speed up, S = ts/tp
= O(m*n)$/O$((2*n*p + 2*m + 2*n)/p)
= $(p*m*n)/(2(n*p+m+n))$

and efficiency E = ((p*m*n)/(2*n*p + 2*m + 2*n))$/p$
= $(m*n)/(2*(n*p+m+n))$

This parallel algorithm will be cost optimum if the one string size n is as long as $\Omega((p-2*m)/(2*p+2))$ or another string size m is as long as $\Omega(p/2-n*(p+1))$.

6. Experimental Results

I have implemented Yang's algorithms in C with MPI, OpenMP and Hybrid with the help of mercury.cs.umanitoba.ca. In my experiments, I have used 8 pairs of input sequences of various length range. I also have written code for sequential algorithm for LCS and then have compared its results with the results from Yang's algorithm without MPI, OpenMP implementation. From the comparison, it is visible that as Yang's algorithm is computing two tables, so for smaller inputs the computation overhead dominates the overall execution time, but if the input sizes get bigger this overhead may be less and then the overall execution time may supersede the time from the normal sequential algorithm.



Input Size	Execution Time (Normal Sequential Algorithm) (in seconds)	Execution Time (Yang's Algorithm) (in seconds)
7 * 6	0.012	0.302
10 * 15	0.078	0. 115
23 * 25	0.070	0. 184
150 * 200	1.067	13.126
3000 * 8000	14.440	13.046
18000 * 17000	50.846	170.158
48000 * 30000	639.565	190.067

Figure 5. Comparison between execution time of Yang's Algorithm and Sequential Algorithm.

I have compared the execution time for various input dimensions of the Yang's sequential algorithm with the parallel version implemented by MPI, OpenMP and Hybrid. These results can be found in figure 6,7 and 8. I have then experimented the execution time against a different number of processors for a fixed input size which is 48000*30000 for the three approaches. These results can be found in figure 9,10 and 11. For this fixed input size, MPI, OpenMP and Hybrid have the highest performance respectively when 20, 64 and 48 processors are used. After that, I have experimented the speed up of these three approaches for various input sizes. These result can be found in figure 12. I have also shown in a graph the execution time Vs. different input sizes for normal sequential LCS algorithm, Yang's sequential Algorithm and Parallel approaches with MPI, OpenMP and Hybrid which will be found in figure 13. From the graph, I can show that when the input size exceeds 18000 the normal sequential curve moves upward vertically, whereas in Yang's sequential algorithm the same happens when input size becomes greater than 48000. For the other three approaches for input size 18000, the three approaches result almost same but when the input size becomes bigger than 18000, the hybrid approach gives the lowest execution time than the other two approaches. From my experiment, it is visible that hybrid approach has the best speed up than the other two, and OpenMP outperforms MPI when the input sizes get to increase. MPI speed-up is good when input size is smaller but when the size gets bigger the speed-up steadily turns to decrease. This scenario can be seen in figure 14. In the end, I have experimented the efficiency of the three approaches for a various number of processors keeping fixed the input size at 48000*30000. From figure 15, I can show that efficiency gets to decrease whenever the processor sizes start to increase. MPI, OpenMP and Hybrid have efficiency greater or equal 1 up to when the system has the highest involved processor numbers 16, 32, 64 respectively. In this experiment, it is also visible that Hybrid approach has the highest efficiency than the other two approaches up to processor number 64.

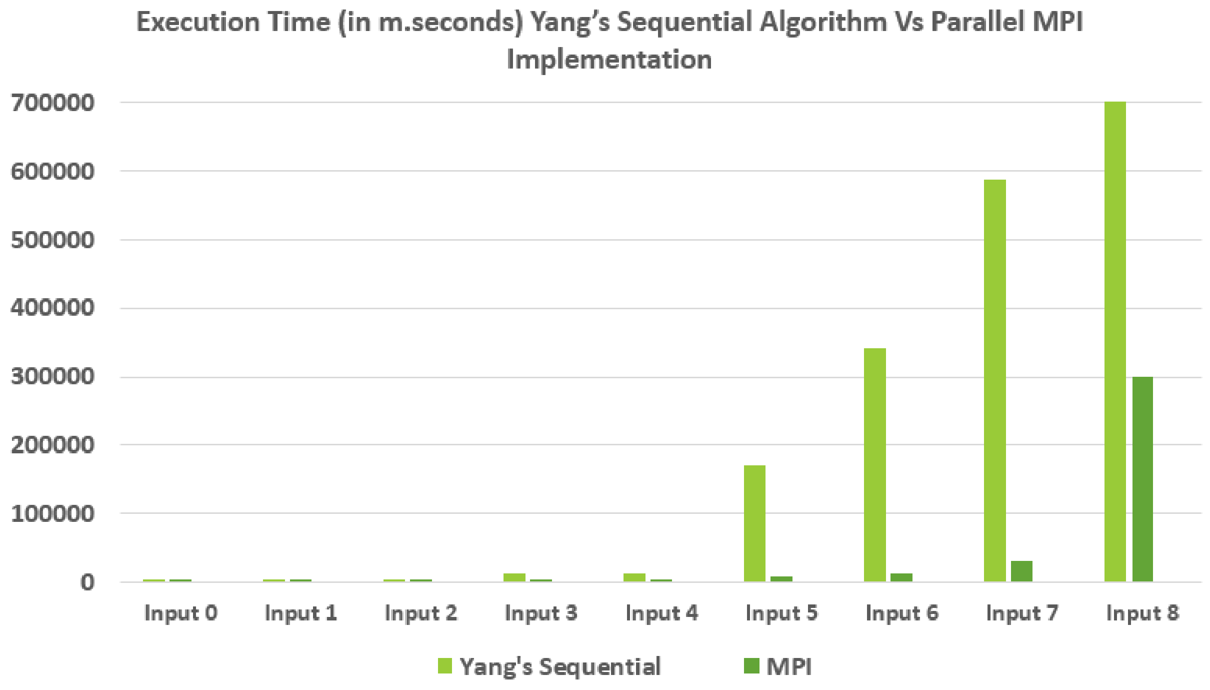


Figure 6. Comparison between execution time of Yang's Sequential Algorithm and paralleled MPI version.

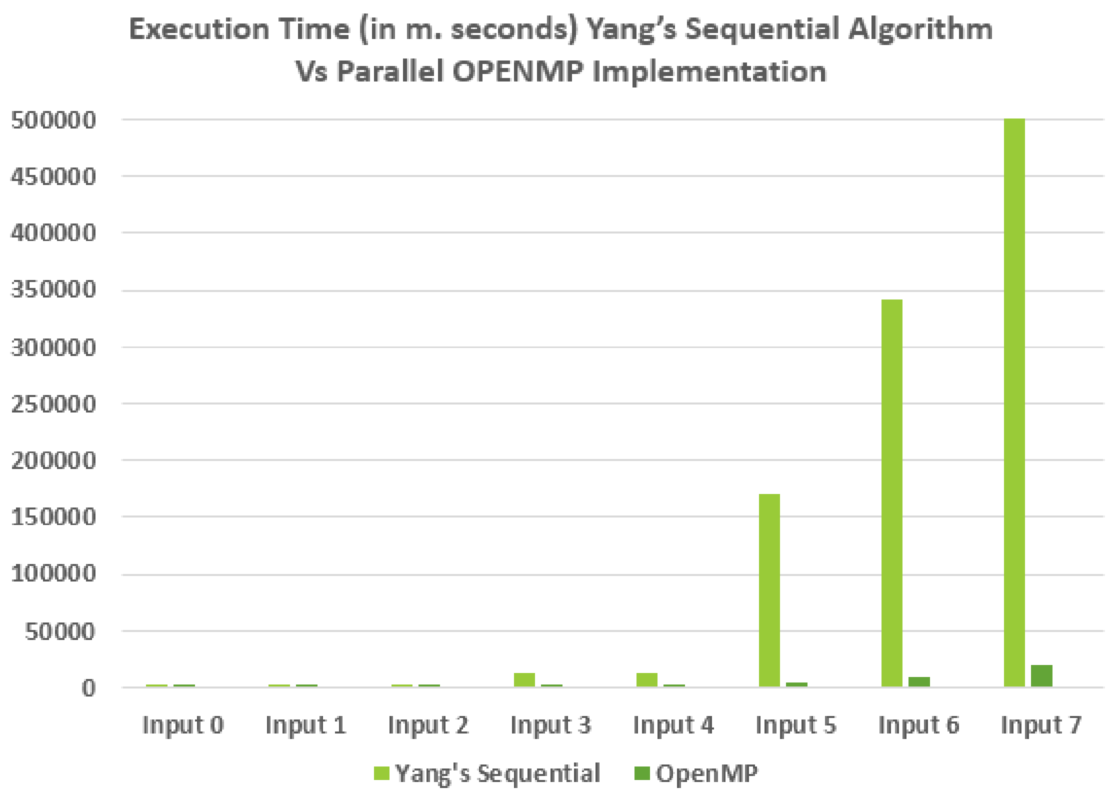


Figure 7. Comparison between execution time of Yang's Sequential Algorithm and paralleled OpenMP version.

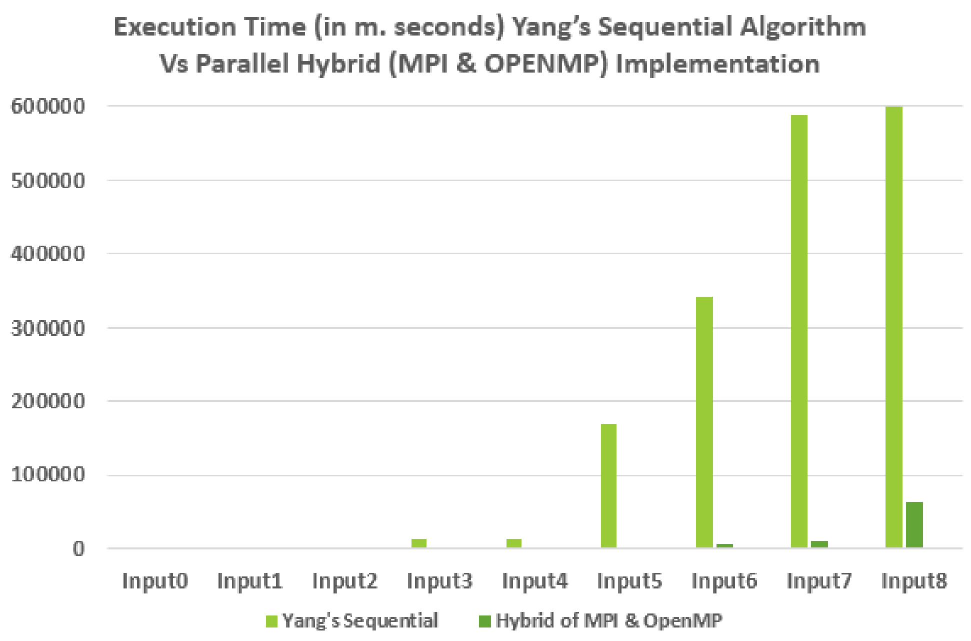


Figure 8. Comparison between execution time of Yang's Sequential Algorithm and paralleled Hybrid version.

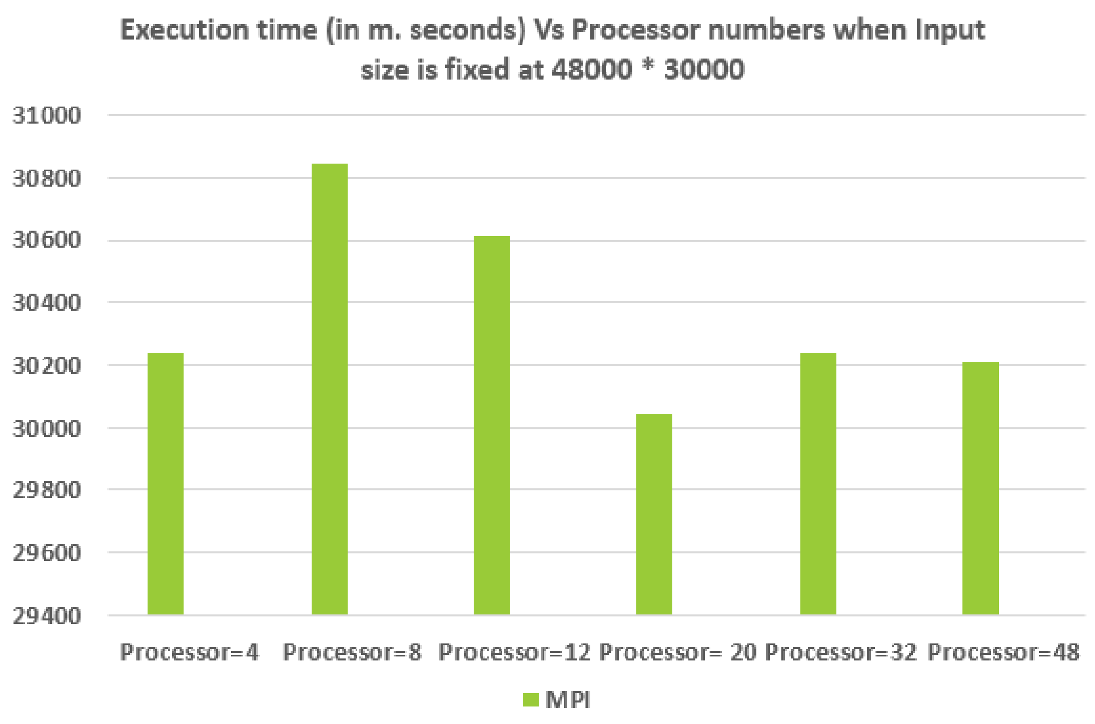


Figure 9. Comparison between execution time and different processor numbers in MPI.

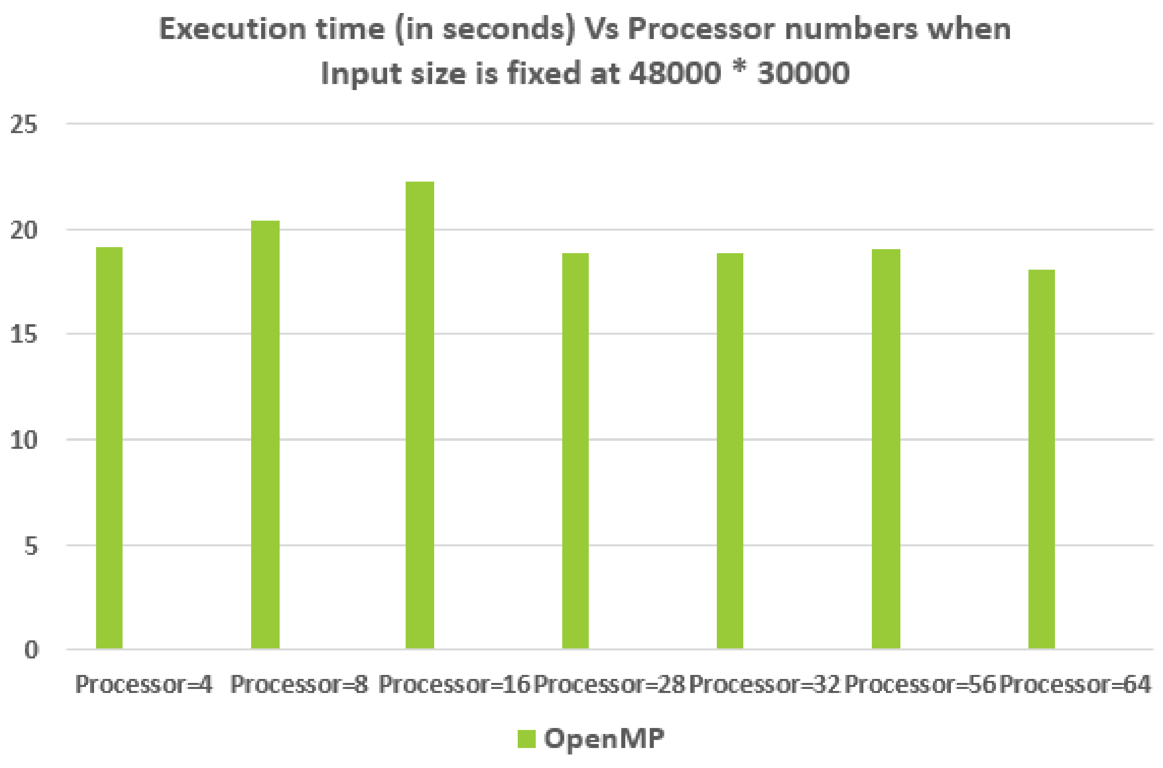


Figure 10. Comparison between execution time and different processor numbers in OpenMP.

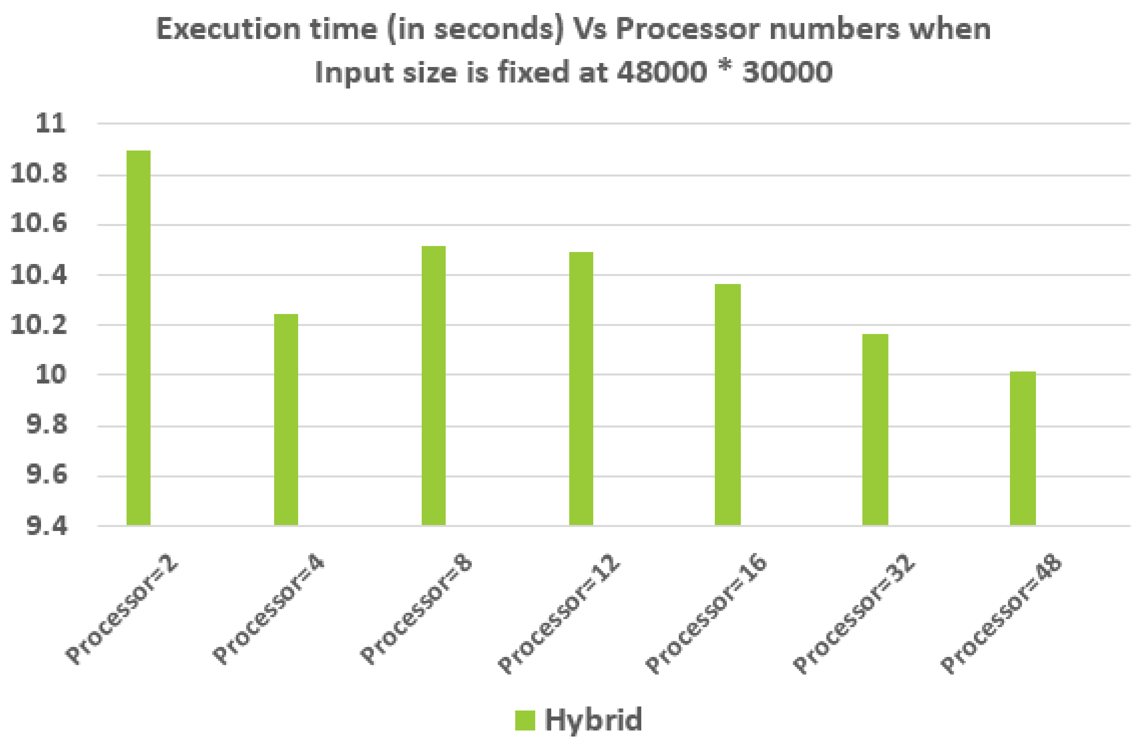


Figure 11. Comparison between execution time and different processor numbers in Hybrid version.

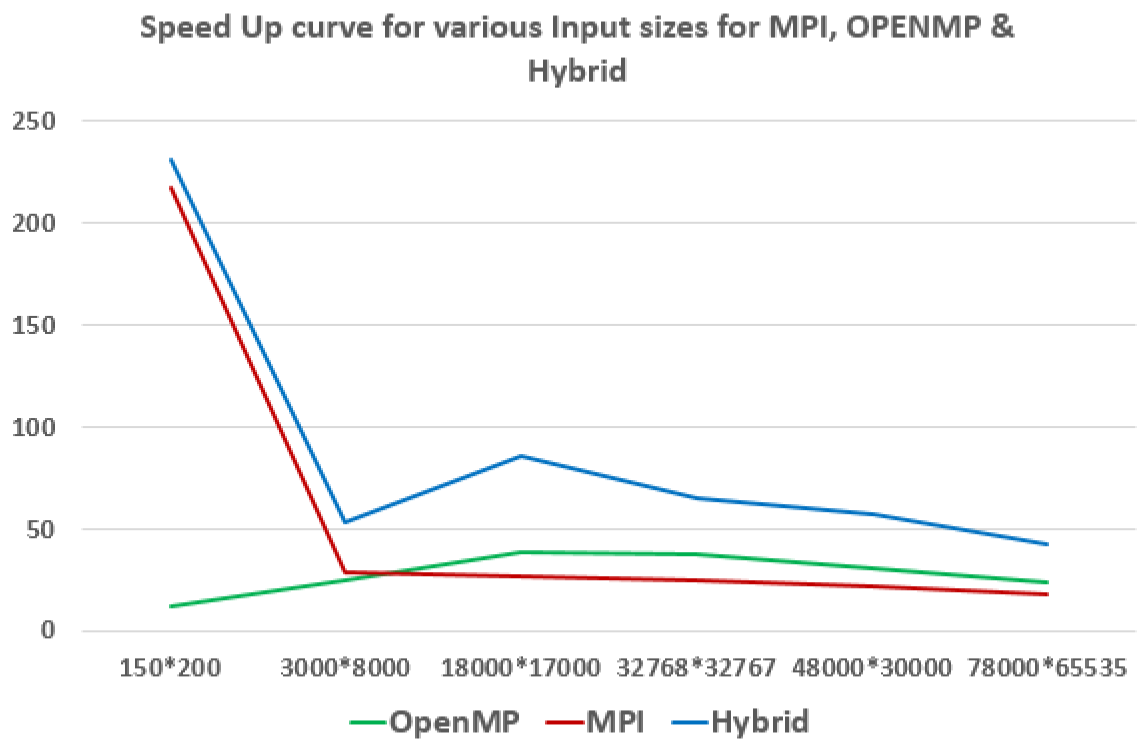


Figure 12. Speed Up curves for the three approaches.

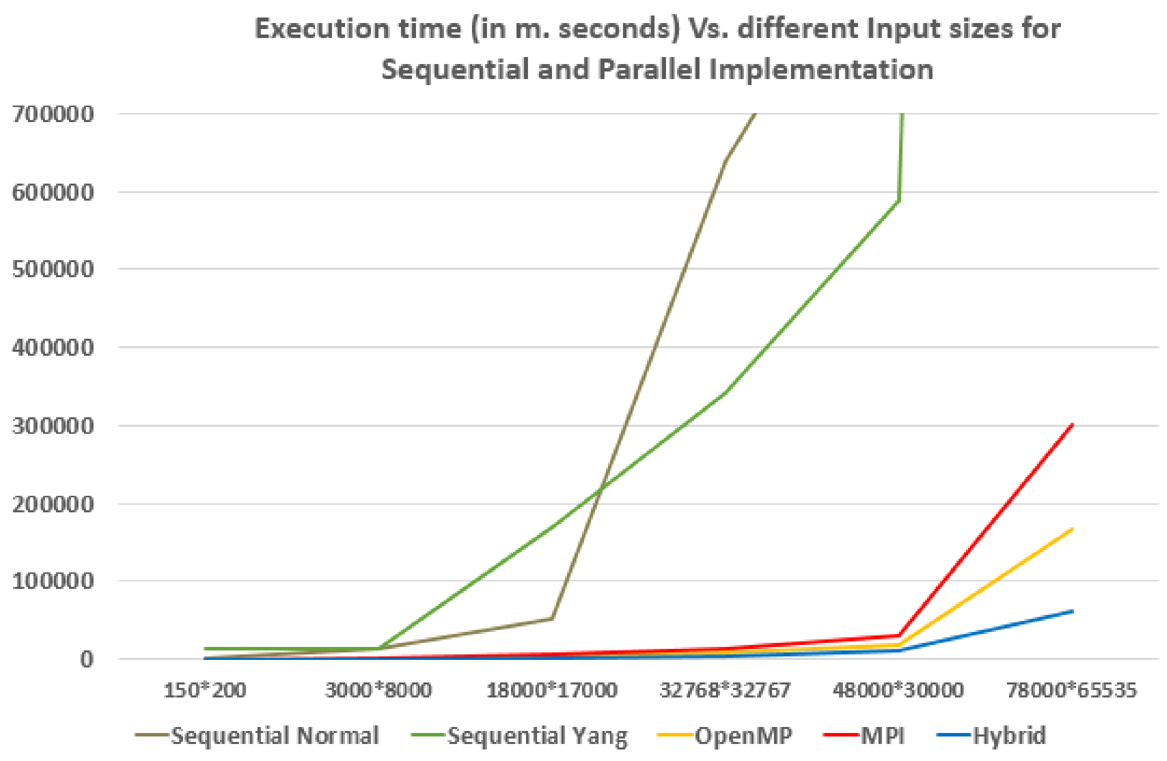


Figure 13. Comparison between execution time of sequential and different parallel approaches.

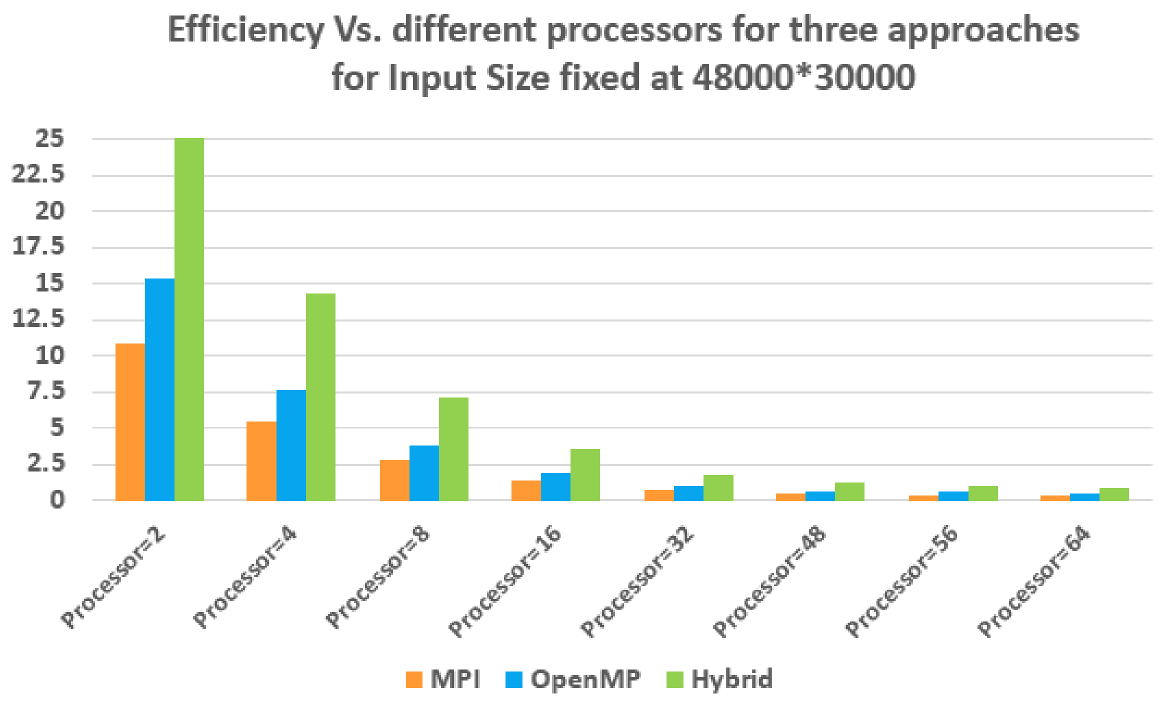
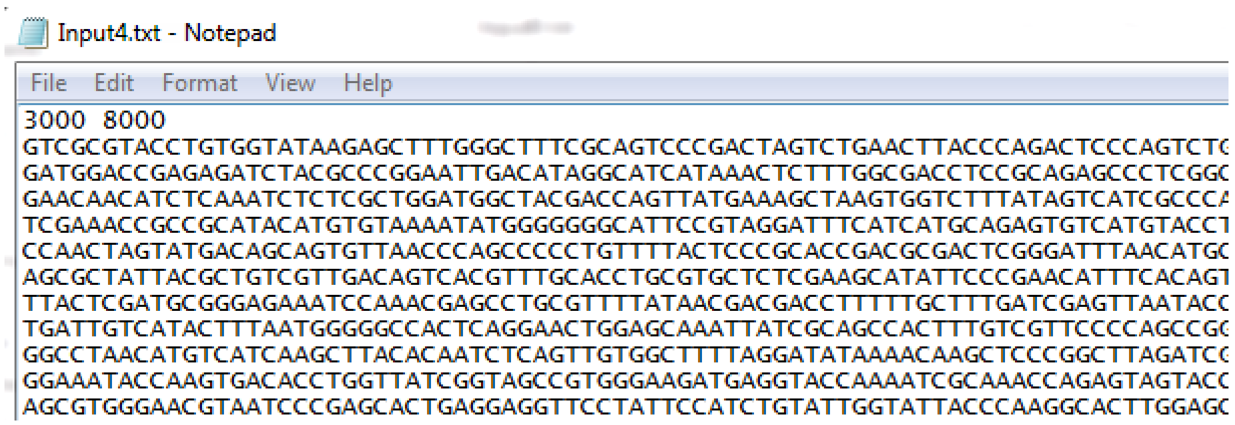
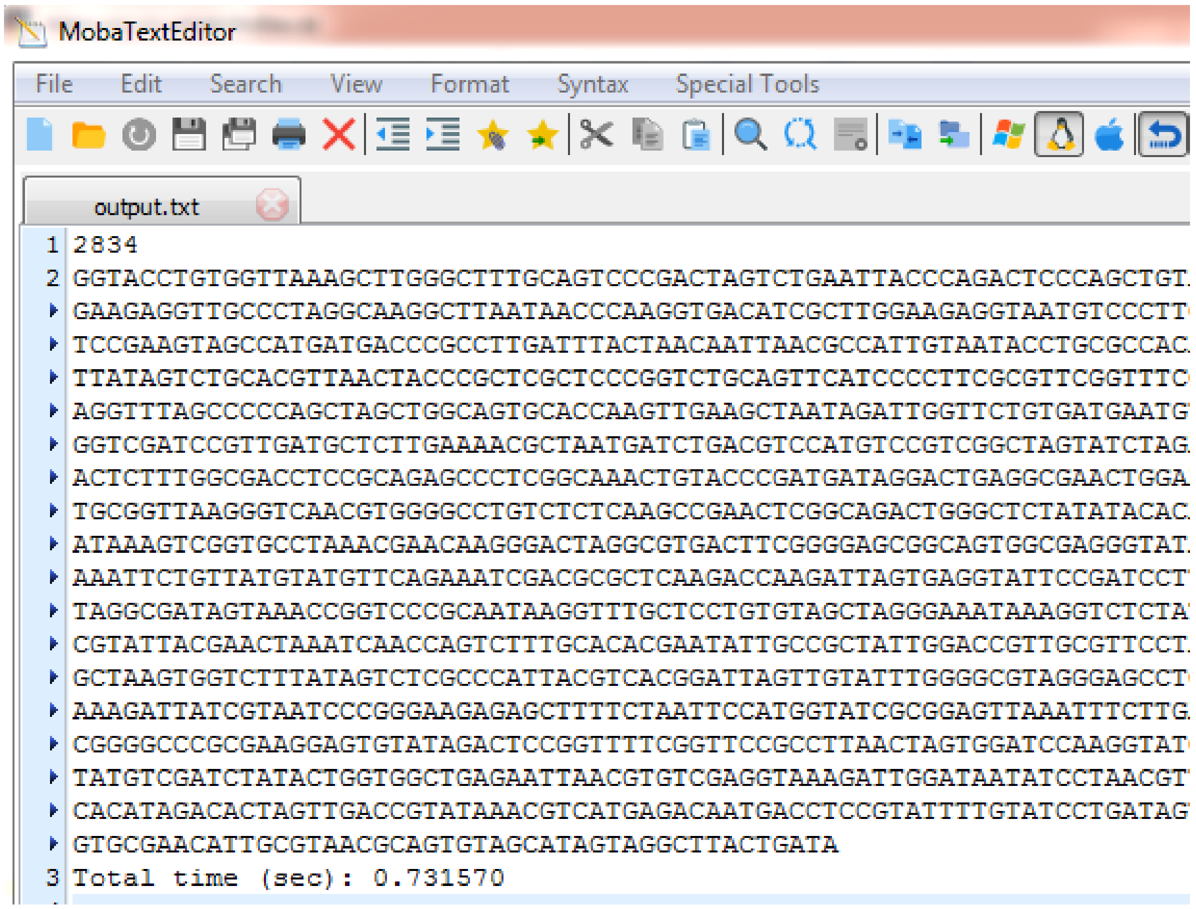


Figure 14. Comparison between efficiency and different processor numbers for the three approaches.



```
3000  8000
GTCGCGTACCTGTGGTATAAGAGCTTTGGGCTTTCGCAGTCCCGACTAGTCTGAACTTACCCAGACTCCCAGTCT
GATGGACCGAGAGATCTACGCCCGGAATTGACATAGGCATCATAAACTCTTTGGCGACCTCCGCAGAGCCCTCGG
GAACAACATCTCAAATCTCTCGCTGGATGGCTACGACCAGTTATGAAAGCTAAGTGGTCTTTATAGTCATCGCCC
TCGAAACCGCCGCATACATGTGTAAAATATGGGGGGCATTCCGTAGGATTTCATCATGCAGAGTGTCATGTACC
CCAACTAGTATGACAGCAGTGTTAACCCAGCCCCCTGTTTTACTCCCGCACCGACGCGACTCGGGATTTAACATG
AGCGCTATTACGCTGTCGTTGACAGTCACGTTTGCACCTGCGTGCTCTCGAAGCATATTCCCGAACATTTCACAG
TTACTCGATGCGGGAGAAATCCAAACGAGCCTGCGTTTTATAACGACGACCTTTTTGCTTTGATCGAGTTAATAC
TGATTGTCATACTTTAATGGGGGCCACTCAGGAACTGGAGCAAATTATCGCAGCCACTTTGTCGTTCCCCAGCCG
GGCCTAACATGTCATCAAGCTTACACAATCTCAGTTGTGGCTTTTAGGATATAAAACAAGCTCCCGGCTTAGATC
GGAAATACCAAGTGACACCTGGTTATCGGTAGCCGTGGGAAGATGAGGTACCAAAATCGCAAACCAGAGTAGTAC
AGCGTGGGAACGTAATCCCGAGCACTGAGGAGGTTCCTATTCCATCTGTATTGGTATTACCCAAGGCACTTGGAG
```

Figure 15. Sample Input



```
2834
GGTACCTGTGGTTAAAGCTTGGGCTTTGCAGTCCCGACTAGTCTGAATTACCCAGACTCCCAGCTGT
GAAGAGGTTGCCCTAGGCAAGGCTTAATAACCCAAGGTGACATCGCTTGGAAGAGGTAATGTCCCTT
TCCGAAGTAGCCATGATGACCCGCCTTGATTTACTAACAATTAACGCCATTGTAATACCTGCGCCAC
TTATAGTCTGCACGTTAACTACCCGCTCGCTCCCGGTCTGCAGTTCATCCCCTTCGCGTTCGGTTTC
AGGTTTAGCCCCCAGCTAGCTGGCAGTGCACCAAGTTGAAGCTAATAGATTGGTTCTGTGATGAATG
GGTCGATCCGTTGATGCTCTTGAAAACGCTAATGATCTGACGTCCATGTCCGTCGGCTAGTATCTAG
ACTCTTTGGCGACCTCCGCAGAGCCCTCGGCAAACTGTACCCGATGATAGGACTGAGGCGAACTGGA
TGCGGTTAAGGGTCAACGTGGGGCCTGTCTCTCAAGCCGAACTCGGCAGACTGGGCTCTATATACAC
ATAAAGTCGGTGCCTAAACGAACAAGGGACTAGGCGTGACTTCGGGGAGCGGCAGTGGCGAGGGTAT
AAATTCTGTTATGTATGTTCAGAAATCGACGCGCTCAAGACCAAGATTAGTGAGGTATTCCGATCCT
TAGGCGATAGTAAACCGGTCCCGCAATAAGGTTTGCTCCTGTGTAGCTAGGGAAATAAAGGTCTCTA
CGTATTACGAACTAAATCAACCAGTCTTTGCACACGAATATTGCCGCTATTGGACCGTTGCGTTCCT
GCTAAGTGGTCTTTATAGTCTCGCCCATTACGTCACGGATTAGTTGTATTTGGGGCGTAGGGAGCCT
AAAGATTATCGTAATCCCGGGAAGAGAGCTTTTCTAATTCCATGGTATCGCGGAGTTAAATTTCTTG
CGGGGCCCGCGAAGGAGTGTATAGACTCCGGTTTTCGGTTCCGCCTTAACTAGTGGATCCAAGGTAT
TATGTCGATCTATACTGGTGGCTGAGAATTAACGTGTCGAGGTAAAGATTGGATAATATCCTAACGT
CACATAGACACTAGTTGACCGTATAAACGTCATGAGACAATGACCTCCGTATTTTGTATCCTGATAG
GTGCGAACATTGCGTAACGCAGTGTAGCATAGTAGGCTTACTGATA
Total time (sec): 0.731570
```

Figure 16. Sample Output

7. Conclusion

Yang's Algorithm is much faster than the normal sequential algorithm of LCS problem. After implementing with MPI and OpenMP and hybrid of them, the run has become faster than the sequential one. In my experiments I haven't tried out with real DNA sequences, I have used synthetic data for this project. So in the future, I want to work with real and huge data sets for this problem.

I will also design the MPI architecture in a more efficient way so that I can more reduce the execution time of MPI implementation and if I can improve this, ultimate Hybrid implementation's speed up will also be increased.

References

[1] K. Hakata and H. Imai. Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima. *Optimization Methods and Software*, 10(2):233–260, 1998.

[2] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[3] W. Hsu and M. Du. Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics*, 24(1):45–59, 1984.

[4] H. Hyyrö. Bit-parallel lcs-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, pages 16–27. Citeseer, 2004.

[5] Y.-C. Lin and C.-Y. Su. Faster optimal parallel prefix circuits: New algorithmic construction. *Journal of Parallel and Distributed Computing*, 65(12):1585–1595, 2005.

[6] S. A. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(2):S10, 2008.

[7] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[8] E. W. Myers and W. Miller. Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17, 1988.

[9] N. Ukiyama and H. Imai. Parallel multiple alignments and their implementation on cm5. *Genome Informatics*, 4:103–108, 1993.

[10] M. S. Waterman. Efficient sequence alignment algorithms. *J. theor. Biol*, 108(3):333–337, 1984.

[11] J. Yang, Y. Xu, and Y. Shang. An efficient parallel algorithm for longest common subsequence problem on gpus. In *Proceedings of the World Congress on Engineering*, volume 1, pages 499–504, 2010.