

# An Efficient Hybrid MPI-OpenMP Based Parallel Approach for Longest Common Subsequence

Akash Yadav

Dept. of Computer Science and Engineering  
Malaviya National Institute of Technology  
Jaipur, India  
2019rcp9150@mnit.ac.in

Alina Khan

Karaganda Medical University  
Republic of Kazakhstan  
al.khan@qmu.kz

Mushtaq Ahmed

Dept. of Computer Science and Engineering  
Malaviya National Institute of Technology  
Jaipur, India  
mahmed.cse@mnit.ac.in

**Abstract**—In the area of bio-informatics and pattern recognition, sequence alignment and pattern matching are important operations, respectively. Finding the Longest Common Subsequence(LCS) plays an essential role in sequential alignment and pattern matching operations. The longest common subsequence problem is generally solved using the dynamic programming paradigm. It takes a considerable amount of time to solve the LCS problem for the large inputs, especially when executed sequentially. Now every computer system is equipped with multi-core architectures, and this system supports parallelization. In recent years, several methods have been proposed to solve LCS, which exploits parallelism for faster execution. In this paper, we explore a hybrid parallel approach to solve the LCS problem efficiently and faster. The proposed method uses two levels of parallelism: Message Passing Interface(MPI) and OpenMP-API. The experiment results show that the proposed method outperformed the sequential method and the OpenMP-based diagonal parallelization method.

**Index Terms**—Parallel algorithms, Longest Common Subsequence(LCS), Dynamic Programming, Message Passing Interface, OpenMP

## I. INTRODUCTION

For a given pair of input strings or sequences, finding the longest subsequence common to both is a classical problem known as Longest Common Subsequence(LCS) problem. The LCS problem plays an important role in many fields of computation [1], [2]. The major application area of LCS is in the field of bio-informatics, where it is used for the sequence alignment of genomes which is a very important component in bio-informatics [3], [4].

In bio-informatics, The DNA sequence is represented as a string of four characters: A, C, G, and T, as shown in Fig. 1. DNA sequence alignment and matching are the most significant and vital operations. These operations are used

... A T T A G C C G G T ...

Fig. 1: Example of a DNA sequence

to identify the common subsequence and arrangement of the DNA sequences based on identified common subsequences to determine the similarity regions among DNA sequences [5], [6]. Sometimes it is also required to find similar sequences between more than two strings, and the problem is known as

the Multiple Longest Common Subsequence(MLCS) problem [7], [8], [9].

LCS is also used in Patter Recognition to match patterns or compare strings [10], between given texts or contents of the files. This classical problem is solved using the Dynamic Programming(DP) approach. In DP, a larger problem is divided into smaller sub-problem. These sub-problems are solved individually, and their results are stored in a matrix to be used later if a similar sub-problem occurs. Storing results in the matrix saves time and speed-up the process [11].

The dynamic programming approach saves the result for a sub-problem to be used later but still, for large inputs, it takes a considerable amount of time and space [12]. Many algorithms have been proposed to reduce running time and space for LCS problem [13], [14]. Now-a-days, every computer system is a multi-core system. They are equipped with more than one processor, which can work in parallel to complete a task faster [15], [16]. The DP-based solution can not be parallelized when the matrix elements are accessed in a row-wise manner [4]. Many algorithm have been proposed in recent years that uses the diagonal-wise accessing [17], [18] of matrix elements to exploit the parallelism of the multi-core systems to reduce execution time.

This paper proposes a hybrid parallel algorithm for the LCS problem. The proposed method accesses the LCS matrix diagonal-wise and uses two levels of parallelism to compute the value of the LCS matrix. The Message Passing Interface(MPI) introduces the first level of parallelism, which divides the task into smaller sub-tasks and distributes them between multiple processes. Afterward, each process used OpenMP Application Programming Interface(API) to apply thread-level parallelism to complete their respective sub-task faster. The proposed approach achieves a significant speed up in execution time for larger input sequences compared to the sequential approach and OpenMP-based approach to the LCS problem [19].

The rest of the paper is organized as follows: Section II explains the Longest common subsequence and dynamic programming approach to solve LCS. In Section ??, discuss the diagonal-wise manner to parallelize the LCS problem along with two mostly used parallel programming methods, MPI and OpenMP, in detail. Section IV presents the proposed

parallel hybrid parallel method to solve the LCS problem using MPI and OpenMP APIs. Section V gives the details of the experimental setup followed by the result analysis. Finally, in Section VI, we conclude the paper.

## II. LONGEST COMMON SUBSEQUENCE

The longest common subsequence problem is mostly solved using dynamic programming because it has an optimal substructure property required by the dynamic programming approach. The optimal substructure for finding LCS can be defined as follows:

Let  $A = \langle a_1, a_2, \dots, a_n \rangle$  and  $B = \langle b_1, b_2, \dots, b_m \rangle$  be the sequences of size  $n$  and  $m$  respectively, and let  $S = \langle s_1, s_2, \dots, s_k \rangle$  be any LCS of  $A$  and  $B$ .

- 1) If  $a_n = b_m$ , then  $S_k = A_n = B_m$  and  $S_{k-1}$  is LCS of  $A_{n-1}$  and  $B_{m-1}$ .
- 2) If  $a_n \neq b_m$ , then  $S_k \neq A_n \implies S$  is LCS of  $A_{n-1}$  and  $B$ .
- 3) If  $a_n \neq b_m$ , then  $S_k \neq B_m \implies S$  is LCS of  $A_n$  and  $B_{m-1}$ .

The optimal substructure can be defined recursively using the mathematical formula given in Eq. 1.

$$S(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ S(i-1, j-1) + 1 & \text{if } a_i = b_j \\ \max(S(i, j-1), S(i-1, j)) & \text{otherwise} \end{cases} \quad (1)$$

where:  $0 \leq i \leq n$  and  $0 \leq j \leq m$ .

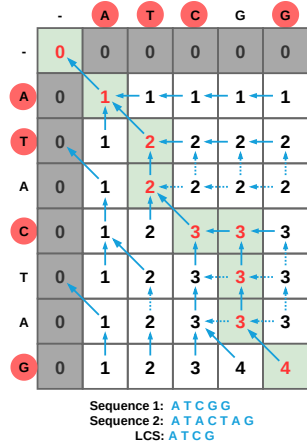


Fig. 2: Example of Longest Common Subsequence matrix

The Eq. 1 shows that value of element  $S(i, j)$  depends on three entries:  $S(i-1, j-1)$ ,  $S(i-1, j)$  and  $S(i, j-1)$ . Fig. 2 shows the example of the matrix generated for finding the LCS for two sequences, ATCGG and ATACTAG. There are many common subsequences between the two strings, such as AT, TC, ATG, ACG, etc. The longest common subsequence among those subsequences is ATCG. Depending on the input sequences, there may also exist more than one LCS between them. For example, input sequences ATCGA and ATACTAG have two longest common subsequences, ATCA and ATCG.

In Fig. 2 the solid arrows show from which neighbor (arrow-head) the current cell's value (arrow-tail) is calculated. The dashed arrows represent that the current cell's value can be computed from either of its neighbors. The value of the  $S(n, m)$  element is the length of the longest common subsequence of given strings and LCS can be found by tracing back the LCS matrix as shown in Fig. 2.

## III. PARALLELIZING MODELS

From Eq. 1, it is clear that the value of two elements either lying in same row((i,j) & (i-1,j)) or in same column ((i,j) & (i,j-1)) can not be computed in parallel because they depend on each other. Therefore, we can not parallelize the generation of the LCS matrix by row-wise computation or column-wise computation. On the other hand, elements lying on the same diagonal do not depend on each other. Hence elements lying on a diagonal can be computed in parallel. Fig. 3 shows the diagonal execution of matrix generation. OpenMP and Message Passing Interface(MPI) are two largely

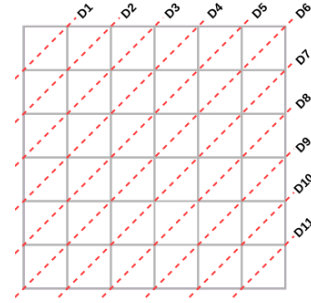


Fig. 3: Diagonal wise matrix filling for Longest Common Subsequence [20]

used programming paradigms for parallelization. OpenMP is a thread-based parallelization method in which all threads share the system's memory.

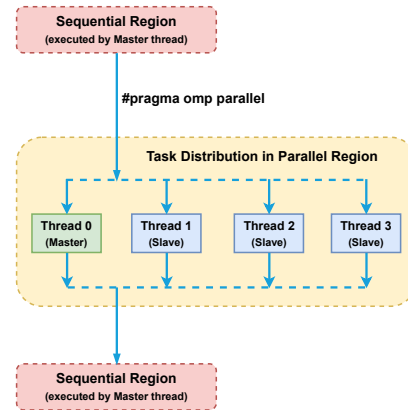


Fig. 4: Parallelization using OpenMP

In OpenMP Application Programming Interface(API), a master thread creates multiple slave threads that can work in parallel [21], [22]. The master thread divides the tasks into parts and assigns these parts to the slave threads to execute in

parallel. Fig. 4 shows the distribution of tasks between threads when a parallel region occurs.

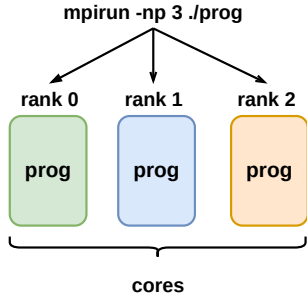


Fig. 5: Parallelization using MPI

Message Passing Interface(MPI) is also a well-known method used for parallelism. In MPI, multiple instances of the same task are created, and these instances run in parallel based on the number of available processors, as shown in Fig. 7. These instances are called processes and have their own private memory. They communicate with each other using messages with the help of the MPI API library.

MPI API library contains a rich set of functions [23]. These functions can provide point-to-point or collective (such as broadcast scatter etc.) types of communication as shown in Fig. 6(a) and Fig. 6(b) respectively. In MPI, data are distributed between the processes using these functions to parallelize the task [24].

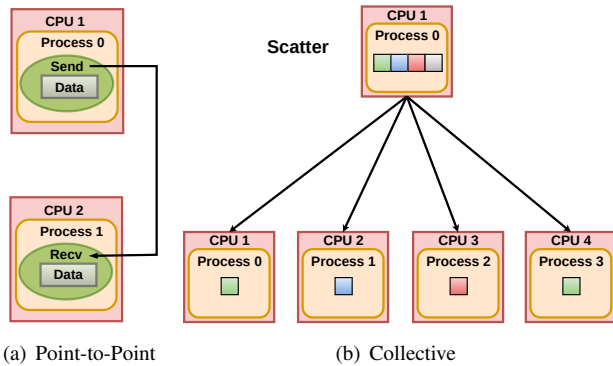


Fig. 6: Type of communication between process for data sharing using MPI API (a) Point-to-Point Communication and (b) Collective Communication

#### IV. PROPOSED METHOD

The Section presents the proposed approach with examples. In the proposed approach, the given two sequences are divided into smaller sub-sequences of the same sizes. Let  $A = \langle a_1, a_2, \dots, a_n \rangle$  and  $B = \langle b_1, b_2, \dots, b_m \rangle$  be the two sequences. Then, for a given size  $k$ ,  $A$  and  $B$

are divided into smaller sub-sequences  $(A_1, A_2, \dots, A_n)$  and  $(B_1, B_2, \dots, B_m)$ , respectively, such that

$$A = A_1 + A_2 + \dots + A_n, \quad \text{and} \\ B = B_1 + B_2 + \dots + B_m$$

where as

$$\begin{aligned} A_1 &= \langle a_1 \dots a_k \rangle, & B_1 &= \langle b_1 \dots b_k \rangle, \\ A_2 &= \langle a_{k+1} \dots a_{2k} \rangle, & B_2 &= \langle b_{k+1} \dots b_{2k} \rangle, \\ &\vdots & &\vdots \\ A_n &= \langle a_{(n-1)k+1} \dots a_n \rangle, & B_m &= \langle b_{(m-1)k+1} \dots b_m \rangle \end{aligned}$$

This divides the LCS matrix into  $\frac{n}{k} \times \frac{m}{k}$  blocks of size  $k \times k$ . The Fig. 7 shows the division of LCS matrix into smaller blocks for two input sequences,  $A$  and  $B$ , of size 16 (i.e.  $n = m = 16$ ) and sub-sequences size  $k = 4$ .

Fig. 7 shows that each block has its respective smaller sub-sequences of input sequences  $A$  and  $B$ . For example, block  $A_3B_4$  calculates the LCS values between sub-sequences  $\langle a_9a_{10}a_{11}a_{12} \rangle$  and  $\langle b_{13}b_{14}b_{15}b_{16} \rangle$  of sequences  $A$  and  $B$ , respectively. The MPI API is used to distribute blocks of a diagonal (blocks with the same color) between processes so that they can be processed in parallel. Algorithm 1 shows the steps to assign the blocks to the process. For example,

**Algorithm 1** Procedure to distributes blocks between processes

- 1:  $m$  = size of sequence  $A$
- 2:  $n$  = size of sequence  $B$
- 3:  $k$  = size of smaller sequences
- 4:  $procNum$  = total number of processes created using MPI
- 5:  $procRank$  = rank of the process assigned by MPI  
/\* number of blocks in matrix row \*/
- 6:  $BRow = m/k$   
/\* number of blocks in matrix column \*/
- 7:  $BCol = n/k$   
/\* number of diagonal in matrix of blocks \*/
- 8:  $BDia = BRow + BCol - 1$   
/\* Assign blocks to each process \*/
- 9: **for**  $i = 1, 2, \dots, BDia$  **do**
- 10:    $BCount = Block\_Count(i)$
- 11:    $BLen = BCount/procNum$
- 12:    $BStart = round(BLen * procRank)$
- 13:    $BEnd = round(BLen * (procRank + 1)) - 1$
- 14: **end for**

if MPI creates a total of four processes, each with a unique rank (starting at 0), and one diagonal of the partitioned matrix contains ten blocks with index values from 0 to 9. Then according to Algorithm 1:

$$\begin{aligned} BCount &= 10 \\ procNum &= 4, \text{ and} \\ BLen &= 2.5 \end{aligned}$$

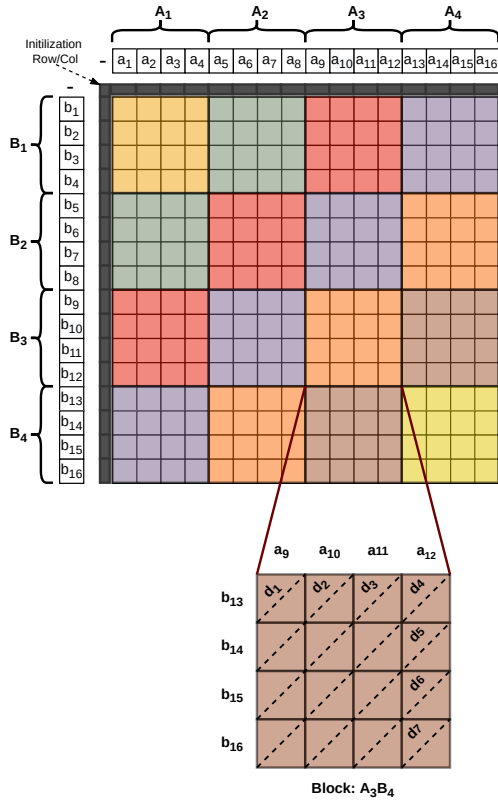


Fig. 7: Distribution of LCS matrix into rows and columns of blocks for sub-sequence size  $k = 4$

For the process with rank 0, the starting index( $BStart$ ) and ending index( $BEnd$ ) of assigned blocks will be computed as:

$$\begin{aligned}
 BStart &= \text{round}(Blen * \text{procRank}) \\
 &= \text{round}(2.5 * 0) \\
 &= 0 \\
 BEnd &= \text{round}(BLen * (\text{procRank} + 1)) - 1 \\
 &= \text{round}(2.5 * (0 + 1)) - 1 \\
 &= \text{round}(2.5 * 1) - 1 \\
 &= 3 - 1 = 2
 \end{aligned}$$

Table I shows the distribution of these ten blocks between four processes. Each process executes its respective blocks sequentially. Each process uses a procedure similar to Algorithm 1 to count the number of diagonals in each block and access the elements of the block diagonally, as shown in Fig. 7. For every diagonal, the process uses OpenMP-API to create threads to compute the value of elements of a diagonal in parallel.

The time complexity of sequential execution of diagonal-wise processing is  $O(nm)$ . In parallel processing, the processes interact with each other for data communication; hence, a parallel algorithm's execution time consists of computation time and communication time. When calculating the time complexity, we generally ignore the communication overhead. The time complexity of a parallel algorithm can be derived

TABLE I: Example of distribution of 10 blocks between 4 processes

Process	Rank	BStart	BEnd	Blocks Assigned
1	0	0	2	3
2	1	3	4	2
3	2	5	7	3
4	3	8	9	2

using Eq. 2.

$$\text{Parallel time complexity} = \frac{\text{Sequential time complexity}}{\text{Number of processors}} \quad (2)$$

In the proposed approach, the MPI creates  $P$  processes, dividing the blocks among themselves to execute them in parallel. Then each process uses OpenMP-API to create  $T$  threads to evaluate the value of elements of the blocks in parallel. Therefore, the overall time complexity for the proposed algorithm is  $O(\frac{nm}{PT})$ .

## V. EXPERIMENTAL SETUPS & RESULTS

In experiments, input sequences of size between 1K to 50K are used with an interval of size 5K. A generator program is used to create these sequences. In each test, the size of the input sequences A and B are the same. The execution time for computing the LCS matrix is measured in seconds. For the accuracy of the result, each implementation is executed five time for same input sequence pair. The final execution time is the average of execution time of these five executions. All experiments were carried out on a computer system with Intel(R) Xeon(R) processor. Table II. shows the details of the hardware and software used in experimental setups.

TABLE II: Details of hardware/software specifications used in experiments

Hardware/Software	Specification
Processor	Intel(R) Xeon(R) E5-2699 v3 (72 cores)
Clock speed	2.30GHz
RAM	64 GB
Operating System	Ubuntu Server (64-bit)
C Compiler	GCC v9.4.0
OpenMP	v4.5
MPI	v4.0.3

We have analyzed execution time for different sizes( $k$ ) of subsequences. Fig. 8 shows the comparison of execution time between the sequential method, the OpenMP-based parallelization method, and the proposed method. From Fig. 8 it is clear that the proposed approach outperforms the OpenMP-based parallelization approach. The speed-up metric is used to determine the performance of an parallel algorithm. According to Amdahl's law the speed-up that can be obtained by increasing the number of CPU cores can be defined as:

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}} \quad (3)$$

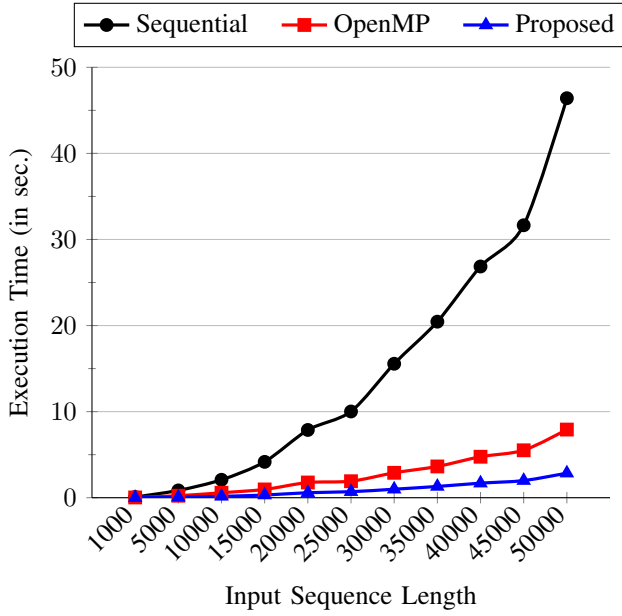


Fig. 8: Comparison of execution time between Sequential, OpenMP-based parallelization and Proposed methods

where  $S(n)$  = Speed Up

$P$  = fraction of algorithm that can be executed in parallel

$n$  = number of CPU

The Eq. 3 can be simplified as:

$$S(n) = \frac{T(1)}{T(n)} \quad (4)$$

where  $T(1)$  = Time an algorithm take to complete executed using a single core

$T(n)$  = Time an algorithm take to complete execution using  $n$  cores

Hence the speed-up difference between the sequential and parallel implementation and between the parallel OpenMP-based implementation and the proposed method's implementation can be given using the Eq. 5 and Eq. 6, respectively.

$$Speed Up = \frac{Sequential Execution Time}{Parallel Method's Execution Time} \quad (5)$$

$$Speed Up = \frac{OpenMP-based method's Execution Time}{Proposed Method's Execution Time} \quad (6)$$

The result shows that OpenMP based parallelization method is about six times faster than the sequential method. In contrast, the proposed method is about fifteen times faster than the sequential approach and approximately three times faster than the OpenMP-based approach.

Table III shows the execution time of the proposed approach for different sizes of subsequences. The bold entries show the best running time for input sequences. The larger size of sub-sequences has very less execution time compared to low size sub-sequences. The results show that dividing the smaller

TABLE III: Comparison of execution time between different sizes( $k$ ) of subsequences used to divide the input sequences

Sequence	Subsequence Block Size(k)								
Length	10	20	40	80	100	200	400	800	1000
1K	0.035	0.033	0.011	0.010	0.008	0.009	<b>0.007</b>	0.007	0.010
5K	0.518	0.316	0.162	0.104	0.081	0.077	<b>0.063</b>	0.074	0.077
10K	2.069	1.417	0.709	0.387	0.358	0.246	<b>0.159</b>	0.176	0.195
15K	5.541	2.651	1.694	0.845	0.767	0.398	<b>0.258</b>	0.297	0.332
20K	10.883	5.719	2.305	1.262	1.378	0.659	<b>0.494</b>	0.511	0.506
25K	20.748	9.388	4.099	2.637	1.999	1.284	0.869	<b>0.698</b>	0.782
30K	30.712	15.337	6.518	3.473	2.644	1.577	1.101	<b>0.988</b>	0.996
35K	43.179	22.628	9.950	5.075	4.279	2.422	1.761	<b>1.307</b>	1.317
40K	58.17	32.595	14.437	7.394	6.658	3.578	2.731	<b>1.697</b>	1.758
45K	75.163	45.656	19.967	10.312	9.866	5.095	3.965	<b>1.988</b>	2.267
50K	94.893	60.371	27.539	16.744	14.106	7.137	5.606	<b>2.853</b>	3.082

input sequences (up to 20K size) into smaller subsequences of size 400 gives the fastest execution times. When the input sequence size increases, subsequences of size 400 start performing slower. Dividing the input sequences into smaller subsequences of size 800 gives the fastest execution time for large input sizes, as shown in Fig. 9 that shows the graphical representation of a part of the Table III data.

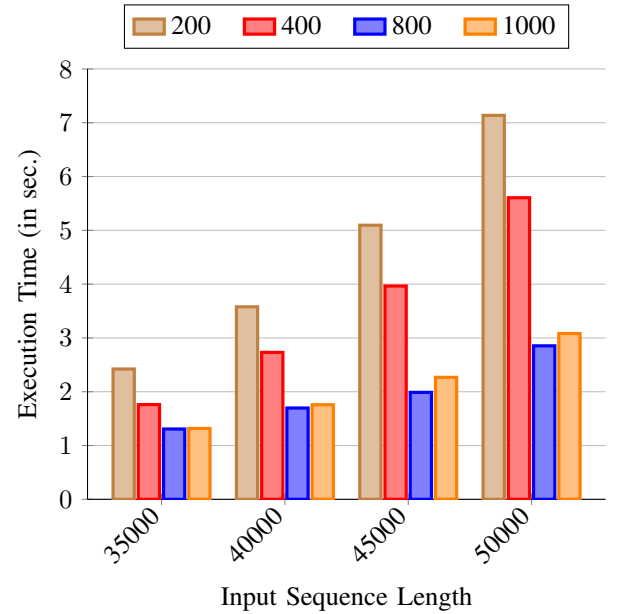


Fig. 9: Comparison of execution time between subsequences of size( $k$ ) 200, 400, 800, and 1000

When the size of sub-sequences  $k$  is very small, it divides the LCS matrix into a large number of blocks. It is similar to solving the LCS matrix sequentially. Each process created by MPI gets a large number of blocks. Because the block size ( $k \times k$ ) is small, it contains less number of diagonals. But, the overall number of diagonals(i.e., counting diagonals in

every block) increased and the processes called OpenMP-API for each diagonal to apply thread-level parallelism. Therefore, threads are created frequently which results in high overhead for creating threads for large number of diagonals. Also, when the number of blocks is high, process calls scatter and gather functions frequently to send and receive data. This also affects the running time. Hence for smaller block sizes, the running time is high compared to large block sizes.

The experiment results show that for given input sequences, the suitable size ( $k$ ) for dividing sequences into subsequences is 800, for which the proposed algorithm provides the best result for large input sequences. As mentioned above, processes are required to call scatter and gather MPI functions to share data between them. In addition to this, they also use other MPI functions to communicate with each other. When the block size increases, the size of data transferred between processes also increases, and it takes a significant amount of time to transfer the data. Hence the transmission of large data between many processes reduces the efficiency of the proposed method and results in increased running time.

## VI. CONCLUSION

The longest common subsequences problem is a classical problem used in many applications. It is solved using the dynamic programming approach. The sequential method to solve the LCS problem takes considerable time for larger input. For faster execution, many parallelization-based approaches have been proposed in recent years to solve the LCS problem. In this paper, we have presented a hybrid MPI-OpenMP based parallelization method to solve the LCS problem on multicore systems. This approach used both MPI and OpenMP-based parallelism.

The MPI-based parallelization breaks the LCS matrix into blocks, and then OpenMP uses thread-based parallelism to calculate the value of elements of each block in parallel. We have also implemented the sequential and OpenMP-based parallelization methods to solve the LCS problem and compare them with the proposed approach. We use large inputs with varying parameters in the experiment, and the results show that the proposed approach executes faster than sequential and OpenMP-based methods. The experiment results showed that the proposed algorithm runs approximately fifteen times faster than the sequential methods and outperforms the OpenMP-based parallelization method by three times. For future work, the proposed approach could be modified and implemented to parallelize the MLCS problem or other dynamic programming algorithms.

## REFERENCES

- [1] A. E. Keshk, M. Ossman, and L. F. Hussein, "Article: Fast longest common subsequences for bioinformatics dynamic programming," *International Journal of Computer Applications*, vol. 57, no. 22, pp. 12–18, November 2012, full text available.
- [2] C.-Y. Lin and F. J. Och, "Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics," in *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, Barcelona, Spain, Jul. 2004, pp. 605–612. [Online]. Available: <https://aclanthology.org/P04-1077>
- [3] C. Yu, Y. Zhao, C. Zhao, H. Ma, and G. Wang, "Diagaf: A more accurate and efficient pre-alignment filter for sequence alignment," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 19, no. 6, pp. 3404–3415, 2022.
- [4] Y. S. Lee, Y. Kim, and R. Uy, "Serial and parallel implementation of needleman-wunsch algorithm," *International Journal of Advances in Intelligent Informatics*, vol. 6, p. 97, 03 2020.
- [5] C. Notredame, D. G. Higgins, and J. Heringa, "T-coffee: A novel method for fast and accurate multiple sequence alignment," *Journal of molecular biology*, vol. 302, no. 1, pp. 205–217, 2000.
- [6] W. J. Hsu and M. Du, "Computing a longest common subsequence for a set of strings," *BIT*, vol. 24, pp. 45–59, 03 1984.
- [7] S. Wei, Y. Wang, and Y.-m. Cheung, "A branch elimination-based efficient algorithm for large-scale multiple longest common subsequence problem (extended abstract)," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 1485–1486.
- [8] Q. Aguado-Puig, S. Marco-Sola, J. C. Moure, D. Castells-Rufas, L. Alvarez, A. Espinosa, and M. Moreto, "Accelerating edit-distance sequence alignment on gpu using the wavefront algorithm," *IEEE Access*, vol. 10, pp. 63 782–63 796, 2022.
- [9] Y. Kim, M. Imani, N. Moshiri, and T. Rosing, "Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing," in *2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2020, pp. 115–120.
- [10] N. Mishin, D. Berezun, and A. Tiskin, "Efficient parallel algorithms for string comparison," in *50th International Conference on Parallel Processing*, ser. ICPP 2021. New York, NY, USA: Association for Computing Machinery, 2021.
- [11] R. E. Bellman, *Dynamic Programming*. New York: Dover Publications Inc, 2013.
- [12] J. Ullman, A. Aho, and D. Hirschberg, "Bounds on the complexity of the longest common subsequence problem," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 1–12, 1976.
- [13] E. Parvinnia, M. Taheri, and K. Ziarati, "An improved longest common subsequence algorithm for reducing memory complexity in global alignment of dna sequences," 06 2008, pp. 57–61.
- [14] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*. IEEE, 2000, pp. 39–48.
- [15] B. Barney, "Introduction to parallel computing tutorial," <https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial>.
- [16] Q. Aguado-Puig, S. Marco-Sola, J. C. Moure, D. Castells-Rufas, L. Alvarez, A. Espinosa, and M. Moreto, "Accelerating edit-distance sequence alignment on gpu using the wavefront algorithm," *IEEE Access*, vol. 10, pp. 63 782–63 796, 2022.
- [17] R. Shikder, P. Thulasiraman, P. Irani, and P. Hu, "An openmp-based tool for finding longest common subsequence in bioinformatics," *BMC Research Notes*, vol. 12, 12 2019.
- [18] O. Gupta, S. Rani, and D. C. Pant, "Impact of parallel computing on bioinformatics algorithms," in *Proceedings 5th IEEE International Conference on Advanced Computing and Communication Technologies*, 2011, pp. 206–209.
- [19] Z. Li, A. Goyal, and H. Kimm, "Parallel longest common sequence algorithm on multicore systems using openacc, openmp and openmpi," in *2017 IEEE 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*, 2017, pp. 158–165.
- [20] Y. S. Lee, Y. Kim, and R. Uy, "Serial and parallel implementation of needleman-wunsch algorithm," *International Journal of Advances in Intelligent Informatics*, vol. 6, p. 97, 03 2020.
- [21] B. Barney, "LLNL HPC Tutorials: OpenMP," <https://hpc-tutorials.llnl.gov/openmp/>.
- [22] <https://www.openmp.org/>.
- [23] <https://www.open-mpi.org/>.
- [24] B. Barney, "LLNL HPC Tutorials: Message Passing Interface(MPI)," <https://hpc-tutorials.llnl.gov/mpi/>.