International Journal of Computing and Digital Systems
ISSN (2210-142X)
Int. J. Com. Dig. Sys.14, No.1 (Jul-23)

http://dx.doi.org/10.12785/ijcds/140119

Performance Analysis of Parallel Implementations of Global Sequence Alignment Methods for DNA Sequencing

Akash Yadav[1], Mushtaq Ahmed[2] and Alina Khan[3]

[1,2]*Department of Computer Science and Engineering, Malaviya National Institute of Technology Jaipur, Rajasthan, India*
[3]*Karaganda Medical University, Republic of Kazakhstan*

Received 20 Dec. 2022, Revised 6 May. 2023, Accepted 10 May. 2023, Published 1 Jul. 2023

Abstract: Global sequence alignment is an essential process in bio-informatics for determining the degree of similarity between two DNA sequences. It is performed using the Dynamic Programming (DP) approach, in which the problem is broken down into smaller ones, and these sub-problems are solved individually. There are various dynamic programming-based approaches to performing global sequence alignments, e.g.,Needleman-Wunsch (NW), Longest Common Subsequences (LCS). It is computationally difficult and complex to perform a global sequence alignment operation when the given DNA sequences are very large because programs based on dynamic programming are generally written to run sequentially. On the other hand, A well-written parallelized code executes many times faster than its equivalent serial code. This paper first explains the NW and LCS methods. Then the parallelization of these methods for shared memory and distributed memory architecture using OpenMP and MPI parallel paradigms is discussed in detail and compares the sequential and parallelized version of the Needleman–Wunsch and longest common subsequences. The detailed experiments for sequential and parallelized implementation of both methods were carried out by varying the DNA sequence length and number of threads in the case of OpenMP-based parallelization and the number of processes in the case of MPI-based parallelization. The obtained experimental results helped to perform a comparative performance evaluation based on the execution time of the sequential and parallel implementation of the methods. The speedup analysis of results showed that the parallelized implementations are many times faster than the equivalent sequential implementation, and MPI-based parallel implementation is approximately three times faster than the OpenMP-based parallel implementation.

Keywords: Parallel Computing, Global Sequence Alignment, Needleman–Wunsch(NW), Longest Common Subsequence, OpenMP, MPI, Sequence Alignment.

1. Introduction

Deoxyribonucleic Acid (DNA) is one of the essential elements of life presented in every organism in the world. It is unique to each human being and living organism. This particular property of DNA enables it to use in various fields [1]. DNA sequence matching is an important application used in forensics for identification purposes and in medicine and agriculture for exploring possible disease and abnormality diagnoses [2], [3]. A DNA sequence is composed of four letters: A, C, G, and T, as shown in Figure 1. In bio-

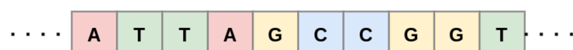


Figure 1. Example of a DNA sequence

informatics, DNA sequence alignment and matching are the most significant and vital operations. These operations are used to identify the common subsequence and arrangement of the DNA sequences based on identified common subsequence to determine the similarity regions among DNA sequences [4]. The sequence alignment can be classified into the following two types [5]:

1) **Global Alignment:** Global alignment is the end-to-end alignment of two DNA sequences to determine the best optimal alignment between them. Closely related sequences of the same lengths are best suited for global alignment (shown in Figure 2).

Figure 2. Example of global sequence alignment

2) **Local Alignment:** Local alignment of two DNA sequences is an alignment of local regions (sub-strings) with a high level of similarity. The local alignment

E-mail address: 2019rcp9150@mnit.ac.in, mahmed.cse@mnit.ac.in, al.khan@qmu.kz

method can also be used to compare even dissimilar DNA sequences (shown in Figure 2).

A C G T G
| |
T C G A T

Figure 3. Example of local sequence alignment

DNA sequence alignment is a fundamental method that aims to find the similarity level between two given DNA sequences or between a given query sequence and different DNA sequences in a DNA database. Many sequence alignment algorithms have been presented for this purpose [6], [7], [8], [9]. Many approaches are also proposed to perform Multiple Sequence Alignment (MSA) in which instead of comparing two DNA sequences, multiple sequences are compared [10], [11]. Some sequence alignment methods also use hardware-based or data-centric approaches that use special hardware or specific data structures such as FM-Index for faster processing [12], [13], [14]. However, most of the methods depend on algorithm-centric approaches such as edit distance, estimation algorithms, dynamic programming, etc., for fast processing [2], [15]. In the dynamic programming-based approach, a larger problem is divided into smaller independent sub-problems, and these sub-problems are solved individually [16]. Generally, DP-based global sequential alignment algorithms are sequential in nature and dividing a larger problem into sub-problems and computing them takes a considerable amount of time. Therefore, high-speed pattern matching is needed to perform a faster operation which requires more computing power.

Today every computer system is a multi-core system. These different cores in the system allow for performing parallel processing for fast computing of complex and large tasks. In parallel processing, a problem is divided into different parts, and these parts are distributed among multiple cores to speedup the processing [17]. As discussed earlier, programs to solve a problem using DP are generally written for sequential execution and are not suitable for parallel computing. Hence efficiently written parallelizable programs are required to perform parallel execution and produce results faster [18], [19], [20], [21].

Our paper addresses different approaches for the parallel implementation of NW and LCS. Our main contributions are:

- Overview of the various method for sequence alignment;
- Review of the two primarily used algorithms (NW and LCS) for global sequence alignment;
- Discussion of the parallel paradigms used to parallelize the above algorithms;
- Comparative performance analysis of the parallel implementation of the above algorithms based on the execution time for various sizes of inputs.

The rest of the paper is organized as follows: Section 2 presents various global sequence alignment techniques. In Section 3, we address shared memory and distributed memory-based parallelization paradigms for implementing global sequence alignment techniques discussed in Section 2. In Section 4, the experimental setup and results were evaluated. Finally, we conclude the paper in Section 5.

2. Global Sequence Alignment Methods

For computational purposes, DNA sequences are represented as strings of characters A, G, T, and C, as shown in Figure 1. Many dynamic programming-based global sequence alignment algorithms exist in the literature [5]. In dynamic programming, a problem is broken down into simpler sub-problems. These sub-problems are solved only once, and their solutions are stored in a matrix-like table. This matrix is used to get the solution for similar sub-problems without solving them again. [22].

In DP-based sequence alignment methods, this matrix is known as the score matrix. Each cell of the score matrix contains a score and a direction pointer indicating from which neighbouring elements (diagonal, left, or top) the current cell's value is calculated. These directional pointers are used in the traceback step to obtain optimally aligned sequences for given DNA sequences. Needleman-Wunsch (NW) and Longest Common Subsequence (LCS) are the widely used methods for sequence alignment [23]. Both of these methods work on the concept of dynamic programming.

A. Needleman–Wunsch(NW) Method

Needleman–Wunsch(NW) algorithm is a prominent dynamic programming-based method used for the global alignment of DNA sequences [24]. For any two given DNA sequences, many possible alignments exist between them. The NW algorithm is used to find the most optimal alignments among those alignments [25]. The NW algorithm assigns a score to each possible alignment, and the alignment with the maximum score is the optimal sequence alignment between two given sequences.

The scoring schema of the NW algorithm used the substitution score and a gap score to calculate the score of each matrix cell. The substitution score can be user-defined (e.g., 1 for match and -1 for mismatch), or it can be obtained from the predefined substitution score matrices like BLOcks SUbstitution Matrix (BLOSUM) and Percent Accepted Mutation (PAM). The gap score defines a gap penalty given to alignment when we have insertion or deletion. It is a user-defined value that is either 0 or a negative integer value(e.g.,-1 or -2).

Given two DNA sequences Seq_A=$< a_1, a_2, .., a_n >$ and Seq_B=$< b_1, b_2, .., b_m >$, The recursive equation for NW

algorithm is defined as follows:

$$NW_Mat(x,y) = max \begin{cases} NW_Mat(x-1,y-1) + S(a_x,b_y), \\ NW_Mat(x,y-1) + GP, \\ NW_Mat(x-1,y) + GP \end{cases} \quad (1)$$

where:
- $S(a_x, b_y)$ is the function to compute the substitution score, and
- GP is the gap penalty

The Eq. 1 shows that the value of the current cell depends on its diagonal (top-left), left, and top neighbors. Figure 4

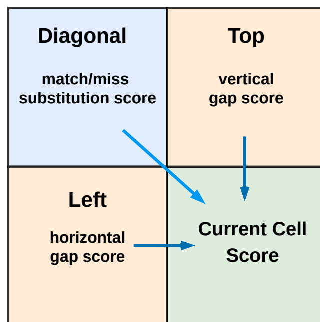


Figure 4. Data-dependency for calculation of cell score in Needleman–Wunsch method

shows that all three neighboring cells are used to compute three different scores (match/miss, vertical gap, and horizontal gap), and the maximum score is the score of the current cell. Figure 5 shows the example of score matrix filling using the NW algorithm.

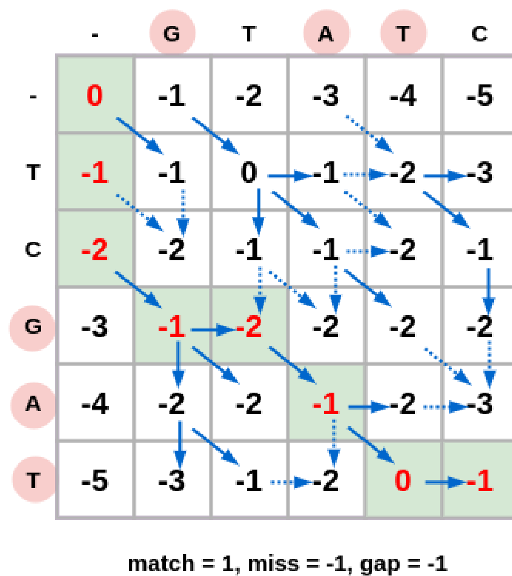


Figure 5. Example of score-table filling in Needleman–Wunsch method

Figure 5 shows the example of score matrix filling using the NW algorithm. In the example, the substitution score is calculated using the substitution function defined in Eq. 2. The substitution score for match is 1, miss is −1, and the *gap penalty* (*GP*) is −1.

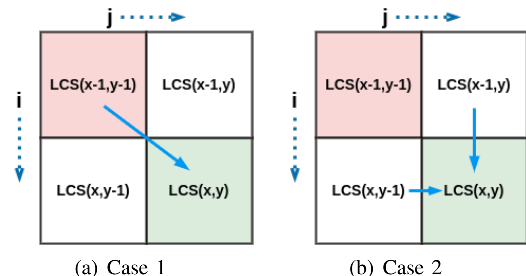
$$S(a_x, b_y) = \begin{cases} 1 & if\ a_x = b_y \\ -1 & if\ a_x \neq b_y \end{cases} \quad (2)$$

B. Longest Common Subsequence

Longest common Subsequence is a special case of global sequence alignment [26], [27]. LCS is used to find the largest subsequence that is common to both sequences [28], [29]. Let $\text{Seq}_\text{A} = < a_1, a_2, .., a_n >$ and $\text{Seq}_\text{B} = < b_1, b_2, .., b_m >$ are the two DNA sequences with the length n and m respectively to be compared and determine the length of a maximal common subsequence in them. The mathematical formula to compute the score of each element of the score matrix in LCS is defined in Eq. 3.

$$LCS(x,y) = \begin{cases} 0 & \text{if } x = 0 \text{ or } y = 0 \\ LCS(x-1,y-1) + 1 & \text{if } a_x = b_y \\ max(LCS(x,y-1), LCS(x-1,y)) & \text{otherwise} \end{cases} \quad (3)$$

where $0 \leq x \leq n$ and $0 \leq y \leq m$. The Eq. 3 states that the score value of matrix element $LCS(x,y)$ depends on three entries: $LCS(x-1,y-1)$, $LCS(x-1,y)$ and $LCS(x,y-1)$ as shown in Figure 6.



(a) Case 1 (b) Case 2

Figure 6. Data-dependency for calculation of cell score in Longest Common subsequence method (a) Case 1: when $a_x = b_y$ (b) Case 2: when $(a_x \neq b_y)$ and $(x \neq 0$ and $y \neq 0)$

Figure 7 shows the example of a score matrix generated using LCS for two DNA sequences. The solid arrows show from which neighbour (arrow-tail) the value of the current cell (arrow-head) is calculated. The dashed arrows represent that the value of the current cell can be computed from either of its neighbours neighbors. The value of the $LCS(n,m)$ element is the length of the longest common sub-sequence of given strings and can be found by tracing back the score matrix.

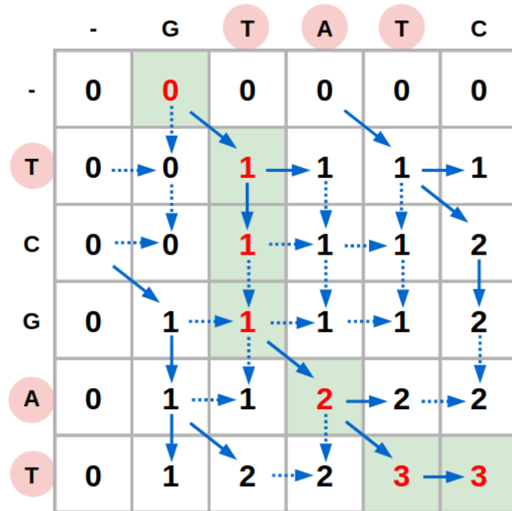
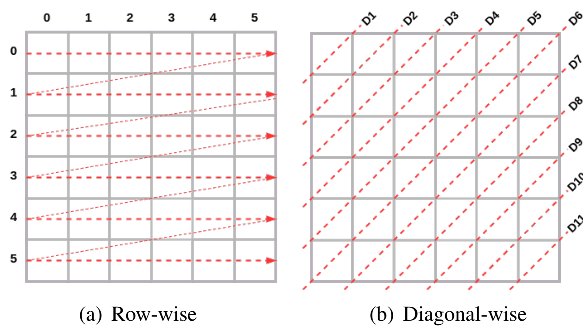


Figure 7. Example of score-table filling in Longest Common Subsequence method

C. Parallel Approach

The score matrix for both LCS and NW methods can be generated row-wise [30]. In the row-wise approach, the score of each element of the score matrix is computed one row at a time, as shown in Figure 8(a). However, in this approach score value of two elements can not be computed simultaneously because every element's score is dependent on its top, left, and diagonal neighbors in both LCS and NW methods, as shown in Figures 4 & 6. Hence, the scores of two cells, either in the same row or in different rows, can not be computed simultaneously. Thus, this row-wise score matrix generation is a sequential approach and can not be parallelized.



(a) Row-wise (b) Diagonal-wise

Figure 8. Cell accessing order for filling the score table in (a) sequential implementation and (b) parallel implementations of NW and LCS methods

In the parallel approach, the elements of the score matrix are accessed diagonally, as shown in Figure 8(b). Figure 6 and Figure 4 show that in both LCS and NW methods, the score value of one element does not depend on the value of any other element in the same diagonal [24]. Therefore, the score of elements in the same diagonal can be computed simultaneously. However, similar to the row-wise manner, elements of two diagonals can not be computed in parallel because every i^{th} diagonal's element's score depends on the $(i-1)^{th}$ diagonal's cell score. Hence in the parallel approach of score matrix generation, the score of cells of the same diagonal can be computed in parallel. Algorithm 1 and Algorithm 2 show the steps of generating the score matrix in a diagonal-wise manner for NW and LCS, respectively.

Algorithm 1 Needleman-Wunsch (A, B)

1: $m \leftarrow$ size of sequence A
2: $n \leftarrow$ size of sequence B
3: $match \leftarrow$ match score
4: $miss \leftarrow$ mismatch score
5: $gap \leftarrow$ gap penalty
6: $score_mat \leftarrow$ A 2D-array of size $(m+1, n+1)$
//Initialize the 1st row of the score matrix
7: **for** $row \leftarrow 0$ to $m+1$ **do**
8: $score_mat[row, 0] \leftarrow gap * row$
9: **end for**
//Initialize the 1st column of the score matrix
10: **for** $col \leftarrow 0$ to $n+1$ **do**
11: $score_mat[0, col] \leftarrow gap * col$
12: **end for**
//Compute scores
13: **for** $dia \leftarrow 2$ to $m+n$ **do**
14: **for** $i \leftarrow max(1, dia-n)$ to $min(m, dia-1)$ **do**
15: $j \leftarrow dia - i$
16: **if** $A[i] = B[j]$ **then**
17: $sub_value \leftarrow score_mat[i-1, j-1] + match$
18: **else**
19: $sub_val \leftarrow score_mat[i-1, j-1] + miss$
20: **end if**
21: $del \leftarrow score_mat[i-1, j] + gap$
22: $ins \leftarrow score_mat[i, j-1] + gap$
23: $score_mat[i, j] \leftarrow max(sub_val, del, ins)$
24: **end for**
25: **end for**

3. Parallelization Models

Parallel computing requires a multicore architecture to perform multiple tasks simultaneously. In the case of a large complex task, parallelization can be achieved by breaking the task into multiple subtasks [31]. Based on the use of the memory architecture during implementation, multicore architecture can be divided into two groups [32], [33], [34]: shared memory system and distributed memory system as shown in Figure 9.

- Shared memory system: Shared memory architecture (also known as the symmetric multiprocessing or SMP model) is a type of computer architecture where a single physical memory space is shared among multiple processors or CPUs. Each CPU has direct access to all memory locations, allowing it to exchange information

Algorithm 2 Longest Common Subsequence (A, B)

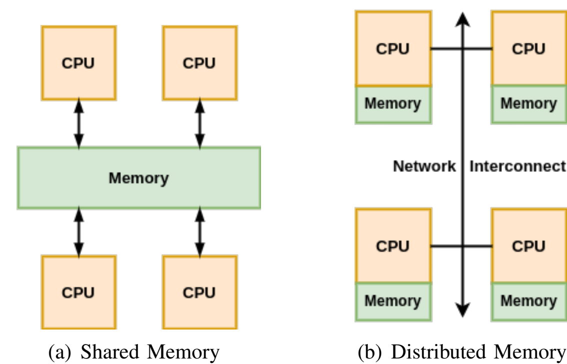
1: $m \leftarrow$ size of sequence A
2: $n \leftarrow$ size of sequence B
3: $score_mat \leftarrow$ A 2D-array of size $(m+1, n+1)$
//Initialize the 1st row of score matrix
4: **for** $row \leftarrow 0$ to $m+1$ **do**
5: $score_mat[row, 0] = 0$
6: **end for**
//Initialize the 1st column of score matrix
7: **for** $col \leftarrow 0$ to $n+1$ **do**
8: $score_mat[0, col] = 0$
9: **end for**
//Compute scores
10: **for** $dia \leftarrow 2$ to $m+n$ **do**
11: $start_ind \leftarrow max(1, dia-n)$
12: $end_ind \leftarrow min(m, dia-1)$
13: **for** $i \leftarrow start_ind$ to end_ind **do**
14: $j \leftarrow dia - i + 1$
15: **if** $A[i] = B[j]$ **then**
16: $score_mat[i, j] \leftarrow score_mat[i-1, j-1]+1$
17: **else**
18: $score_mat[i, j] \leftarrow max(score_mat[i-1, j], score_mat[i, j-1])$
19: **end if**
20: **end for**
21: **end for**

with other CPUs by writing and reading from the same locations.

The advantage of the shared memory model is that it simplifies programming, as processors can easily communicate with each other by accessing the same memory location. This makes it easy to write parallel programs, as it is not necessary to manage communication between processors explicitly. However, shared memory architectures can suffer from performance bottlenecks due to contention for access to the shared memory, which can result in reduced performance. In addition, shared memory architectures are limited by the physical size of the memory, which can limit the scalability of the system. In this architecture, programming models like OpenMP are used to exploit parallelism.

- Distributed memory system: In contrast to the shared memory system, distributed memory system (also known as message-passing architecture) is a type of computer architecture where each processor has its own memory space, and processors communicate by passing messages to each other. In other words, processors do not share memory, and each processor communicates with other processors via a communication network, passing messages and data between them as needed.

Each CPU in a distributed memory model has its



(a) Shared Memory (b) Distributed Memory

Figure 9. Parallel computer architecture based on the memory implementation [34]

own local memory, and communication between CPUs is handled directly, allowing the model to expand to a high number of CPUs or processors. This makes it well-suited for large-scale parallel processing applications. However, distributed memory architectures can be more difficult to program than shared memory architectures, as the programmer must explicitly manage the communication between processors. In addition, communication overhead can be a bottleneck for performance, specifically if the communication network is slow or congested. In distributed memory systems, programming models like MPI are commonly used to manage message passing and parallelism.

A. *OpenMP*

OpenMP is an Application Programming Interface (API) library built for shared memory systems that utilize thread-level parallelization. In OpenMP-based parallelization, all threads share memory and data [35]. At the execution time, an OpenMP program uses one thread (known as a master thread) for all the sequential sections and multiple threads (known as slave threads) for the parallel sections of the program.

In the parallel OpenMP implementation of the LCS and NW methods, the master thread (thread number 0) executes the sequential sections until a parallel section is arrived, represented by OpenMP directive *#pragma omp parallel* as shown in Figure 10. In the parallel section, the master thread creates multiple slave threads based on the number of cores available in the system [37]. In parallel processing, diagonal-wise score computation is performed, and these computation tasks are distributed between master and slave threads [36].

Figure 11 shows the distribution of diagonal elements between threads for score computation. Figure 11(a) shows that the master thread is used for the score computation of only element of the first diagonal. The score computations of the second diagonal's elements are shown in Figure 11(b). the master thread creates one slave thread to compute

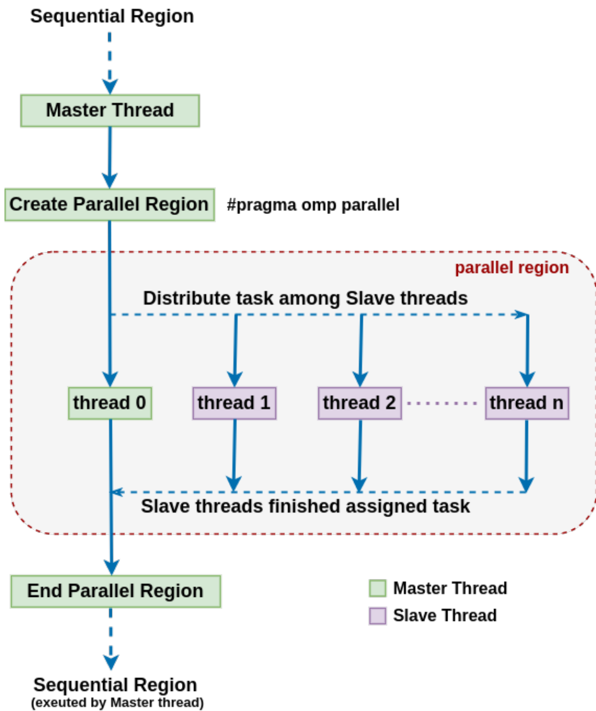


Figure 10. Threads creation by master-slave for the execution of parallel region in OpenMP based parallelization [36]

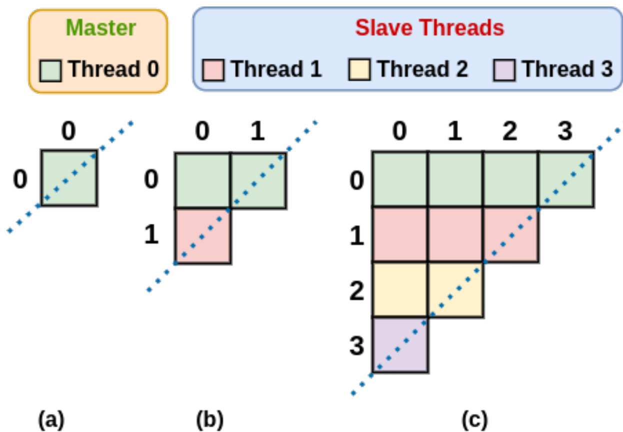


Figure 11. Distribution of a diagonal's elements between master and slave threads in OpenMP for calculating scores in parallel (a) for 1st diagonal, (b) for 2nd diagonal, and (c) for 4th diagonal

the score of both elements of the second diagonal in parallel. Similarly, Figure 11(c) shows that in the case of the fourth diagonal, the master thread creates three slave threads to compute the score of all four elements in parallel.

B. Message Passing Interface

MPI is one of the best parallel computing paradigms designed for distributed memory systems. In MPI, processes communicate data between them using explicit messages [38]. The MPI API library contains a rich number of functions that allow processes to do point-to-point or collective communication. MPI distributes data between the processor

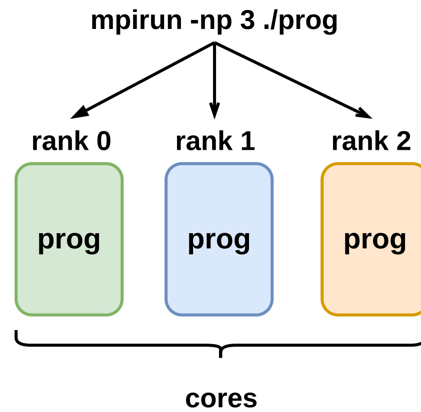


Figure 12. Creation of multiple instances of a program and their distribution between different cores using MPI

using these functions to do parallel processing for tasks that requires high performance. MPI programming has a substantial advantage in terms of program speedup because each process handles a distinct component of the same job concurrently and independently. The *mpirun* command is used to exploit the MPI-based parallelism [39]. The mpirun starts the multiple instances of the same task to run in parallel on available processors, as shown in Figure 12. When an MPI library function is called in the program, the

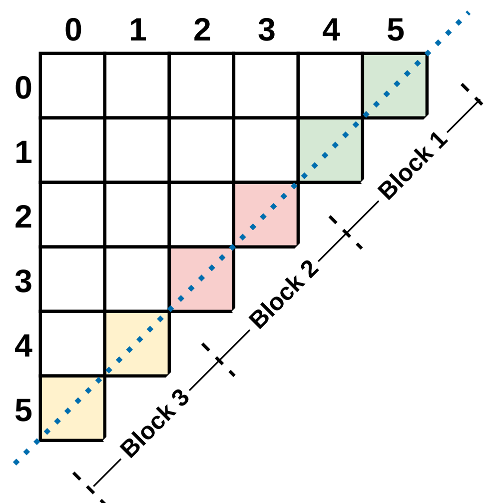


Figure 13. Example of distribution of elements between process in MPI

processes communicate through these functions and start the point-to-point or collective data transfer based on the type of communication between other instances of the program running in parallel.

In the distributed memory system, the MPI parallel

paradigm divides the elements of a diagonal based on the number of processes available. Then each of these processes computes the score of those elements in parallel. The number of elements in a block is calculated as follows:

$$block_length = \frac{total_elements}{size} \quad (4)$$

where:
- *block_length* is the number of elements in a block,
- *total_elements* is the total number of elements in the diagonal, and
- *size* is the total number of processes.

After that, each process calculates the starting index and ending index of the assigned elements using Eq. 5 and Eq. 6.

$$start_index = round(block_length * rank) \quad (5)$$

$$end_index = round(block_length * (rank + 1)) - 1 \quad (6)$$

where:
- *start_index* is an index of the first element of the block,
- *end_index* is an index of the last element of the block, and
- *rank* is the unique number assigned to each process in MPI

The *start_index* and *end_index* are used by the respective processes to iterate through the elements that are assigned to them. Figure 13 shows the distribution of diagonal elements between three processes. Here diagonal has a total of six elements, and each process is assigned a block of length two, i.e., each block contains two elements of the diagonal.

4. Experimental Setup and Result Analysis

This section describes the experimental setup and test inputs used in the experiments, followed by the experimental result discussion. Figure 1 shows that DNA consists

TABLE I. Details of hardware/software specifications used for experiments

Hardware/Software	Specification
Processor	Intel® Xeon® E5-2699 v3
Clock speed	2.30GHz
RAM	64 GB
Operating System	Ubuntu 20.04.2 (64-bit)
C Compiler	GCC v9.4.0
OpenMP	v4.5
MPI	v4.0.3

of four characters, i.e., A, C, G, and T. A generator program is created to randomly generated the pair of DNA sequences of lengths 10k, 15k, 30k, 45k, 60k, 75k, 90k, and 100k for testing the implementations. The sequential and parallel (OpenMP and MPI) versions of both global alignment algorithms are implemented in C language. The computation time spent generating the score matrix is measured for each implementation (serial, OpenMP, and MPI) of both algorithms. For the correctness of the parallel implementations, their maximum scores are compared with the serial implementation.

The run times are measured in seconds and are the average value of five execution of the application for each input for all three implementations. All experiments were carried out on a computer system with Intel® Xeon® processor with 62GB RAM. Table I shows the details of the hardware and software used for the experiments.

A. Result Analysis

We have analyzed execution time with varying numbers of threads and processes for OpenMP and MPI, respectively. In an OpenMP implementation, the number of created threads is equal to the number of active cores. In contrast, in an MPI implementation, the number of processes is equal to the number of active cores.

Table II and Table III show the averaged execution time analysis of the three implementations (sequential, OpenMP, and MPI) of both Needleman-Wunsch and Longest common subsequences methods versus the size of the DNA sequences respectively. The execution time includes both the initialization steps and score-filling steps of the score matrix.

From Table II, it is clear that, in contrast to the parallel implementation, the execution time of the Needleman-Wunsch technique grows exponentially with the amount of the input DNA sequences when the method is sequentially implemented. The results show that there is a gradual increase in execution time in both OpenMP and MPI implementations. We can also observe that for larger DNA sequences, the execution time taken by 28 threads is faster than the execution time for 32 threads. This downgrade in performance is due to the communication and waiting time overhead between a large number of threads for accessing the data from shared memory locations in OpenMP loops. In MPI-based implementation, The execution time is very fast compared to sequential implementation and increases rapidly as the level of parallelization, i.e., the number of processes, increases. After a certain number of processes, this difference in the execution time does not increase very much with the increase in the number of processes. The results show that there is very less difference between the execution time for 28 processes and 32 processes.

From Table III, we can see that sequential and parallel implementation of LCS methods yields results approximately similar to the NW method. Here also in OpenMP implementation, execution time decreases with the increase of the thread, but like the NW method, after a certain number of threads, execution time starts to increase. The MPI implementation for the LCS method executes very fast compared to sequential and OpenMP implementation. From the experimental results, we also performed a speedup analysis between the sequential and parallel implementations of both NW and LCS methods for DNA sequences of size

TABLE II. Comparison in execution time of sequential and parallelized implementations of Needleman-Wunsch method

Execution Type	Threads/ Processes	Sequence Length							
		10K	15K	30K	45K	60K	75K	90K	100K
		Execution Time (in seconds)							
Sequential	1	2.637	6.091	21.655	48.546	86.488	134.373	193.116	256.194
Parallel Shared Memory (OpenMP)	4	2.009	4.996	20.643	44.810	84.454	131.904	190.765	238.980
	8	0.854	2.411	10.327	24.342	44.355	70.299	100.731	127.430
	12	0.507	1.350	6.551	16.544	29.990	47.528	69.241	89.629
	16	0.431	0.878	5.072	12.475	23.288	35.626	52.064	66.887
	20	0.373	0.900	4.225	10.394	18.637	28.454	40.519	58.732
	24	0.340	0.729	3.597	9.242	17.447	26.662	45.637	60.212
	28	0.310	0.666	3.079	8.174	18.587	29.703	38.454	66.635
	32	0.303	0.715	3.044	8.960	19.177	32.613	50.009	83.923
Parallel Distributed Memory (MPI)	4	0.799	1.847	6.595	16.277	27.983	43.031	60.979	74.949
	8	0.478	1.048	3.919	8.774	15.338	23.018	32.379	41.688
	12	0.331	0.702	2.761	5.97	11.313	16.646	23.302	28.138
	16	0.28	0.592	2.06	4.56	8.395	12.401	17.165	22.252
	20	0.244	0.453	2.013	3.669	6.598	9.954	13.772	17.55
	24	0.188	0.41	1.534	3.172	5.726	10.664	14.799	16.186
	28	0.215	0.358	1.281	2.874	7.7	14.687	16.134	16.232
	32	0.188	0.393	1.203	3.494	8.676	15.892	14.372	17.819

TABLE III. Comparison in execution time of sequential and parallelized implementations of Longest Common Subsequence method

Execution Type	Threads/ Processes	Sequence Length							
		10K	15K	30K	45K	60K	75K	90K	100K
		Execution Time (in seconds)							
Sequential	1	2.020	4.397	16.361	34.652	61.497	96.181	138.456	187.250
Parallel Shared Memory (OpenMP)	4	1.469	3.589	15.183	33.453	59.934	94.786	137.893	172.837
	8	0.530	1.587	7.795	18.224	32.322	52.165	75.892	94.718
	12	0.347	0.948	4.805	12.821	22.160	36.796	52.864	71.029
	16	0.339	0.709	3.650	9.877	17.844	28.341	40.600	49.249
	20	0.270	0.708	3.114	8.023	14.002	23.565	35.209	43.313
	24	0.288	0.610	2.749	6.962	13.564	20.841	32.065	39.135
	28	0.262	0.539	2.360	6.906	11.788	21.292	30.862	51.201
	32	0.324	0.737	2.186	7.384	12.005	22.573	35.605	39.256
Parallel Distributed Memory (MPI)	4	0.575	1.262	4.861	11.110	18.755	29.823	41.441	53.469
	8	0.324	0.706	2.585	5.852	10.261	15.790	22.622	36.060
	12	0.240	0.508	1.896	4.167	6.923	10.627	16.015	20.070
	16	0.198	0.418	1.363	3.212	5.193	8.464	13.056	15.225
	20	0.169	0.329	1.161	2.527	4.766	6.451	9.908	12.195
	24	0.155	0.314	1.062	2.301	3.524	5.669	7.870	11.792
	28	0.147	0.297	0.925	2.029	3.346	6.536	8.919	11.809
	32	0.129	0.260	0.840	1.767	3.164	6.227	9.515	14.182

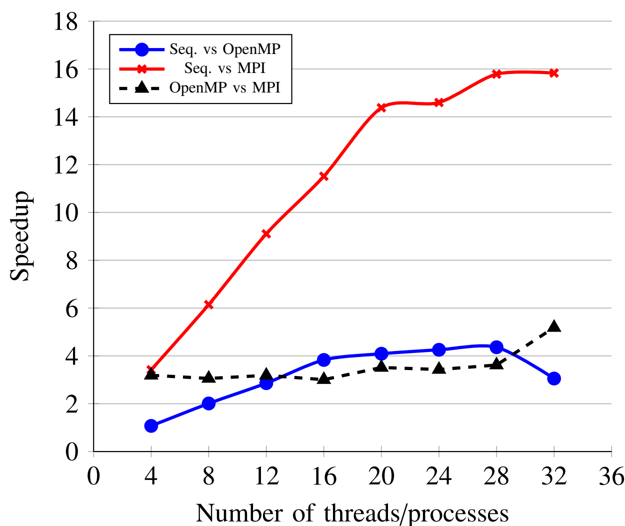
100K. The Eq. 7 has been used to determine the speed difference between implementations.

$$Speed\ Up = \frac{Sequential\ Execution\ Time}{Parallel\ Execution\ Time} \quad (7)$$

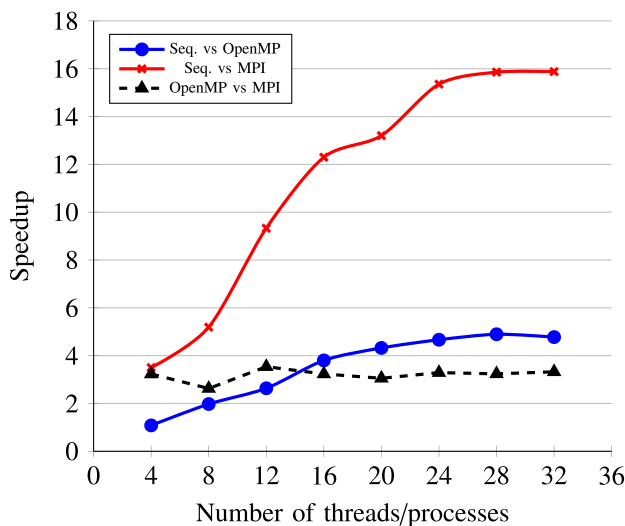
Table IV shows the difference in speed between the sequential execution times and parallel execution times of both the NW and LCS methods for DNA sequences of size 100K. Data from Table IV demonstrates that for large DNA sequences, the MPI-based parallel implementation of NW and LCS methods executes more quickly than the OpenMP-based implementation. Compared to OpenMP, MPI implementation offers a significant speedup. Figure 14(a) and 14(b) illustrate the corresponding graphical representation of Table IV. The results showed that for large DNA sequences, the OpenMP-based parallel implementation of both methods is approximately 4.36 times faster than the sequential implementation. Also, the MPI-based parallel implementation is three times faster than sequential implementation when the number of processes is low. This speedup is increased up to 15 times as the number of processes increases. To analyze the performance difference between OpenMP and MPI implementations, we have also compared the speed of these two parallelization paradigms.

TABLE IV. Speedup the comparison of sequential and parallelized implementations

No. of Threads/ Processes	NW			LCS		
	OpenMP	MPI	OpenMP vs MPI	OpenMP	MPI	OpenMP vs MPI
4	1.07	3.42	3.19	1.08	3.50	3.23
8	2.01	6.15	3.06	1.98	5.19	2.63
12	2.86	9.10	3.19	2.64	9.33	3.54
16	3.83	11.51	3.01	3.80	12.30	3.23
20	4.09	14.38	3.52	4.32	13.20	3.05
24	4.25	14.60	3.43	4.67	15.35	3.29
28	4.36	15.78	3.62	4.89	15.86	3.24
32	4.01	15.83	3.95	4.54	15.88	3.32



(a) Speedup comparison for NW



(b) Speedup comparison for LCS

Figure 14. Speedup analysis of sequential, OpenMP, and MPI implementations for (a) Needleman-Wunsch(NW) method and (b) Longest Common Subsequence(LCS) method

The results show that for both the NW and LCS methods, MPI implementations run approximately three times faster than OpenMP implementations.

5. Conclusions and Future Work

In this paper, we have performed a detailed performance analysis of parallelized implementations of Needleman-Wunsch and longest common subsequences algorithms, two widely used methods for global sequential alignment of DNA sequences. The serial and parallel implementation of both methods were executed on a high-end multicore system. In the parallel implementation, we used both OpenMP and MPI parallelization paradigms to measure the performance of shared memory architecture and distributed memory architecture with respect to the execution time and speedup for varying input sizes.

The results of the experiment demonstrated that the OpenMP-based implementation of methods executes approximately three times faster than the sequential implementation. For MPI-based implementation, the program runs approximately 14 times faster compared to the sequential one when the number of cores is high. The MPI-based parallel implements also outperformed the OpenMP-based parallel implementation and achieved a speedup of 3.20. We planned to implement these methods on GPUs and compare the OpenMP and MPI implementation with Compute Unified Device Architecture (CUDA) based implementations in our future work. Also, an OpenMP and MPI-based hybrid implementation can be developed to achieve a faster execution time.

References

[1] C.-A. Leimeister, T. Dencker, and B. Morgenstern, "Accurate multiple alignment of distantly related genome sequences using filtered spaced word matches as anchor points," *Bioinformatics*, vol. 35, pp. 211 – 218, 2019.

[2] 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.

[3] J. Fang, X. Zhu, C. Wang, and L. Shangguan, "Applications of dna technologies in agriculture," *Current Genomics*, vol. 17, no. 4, pp. 379–386, 2016.

[4] A. Abboud, V. V. Williams, and O. Weimann, "Consequences of faster alignment of sequences," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2014, pp. 39–51.

[5] V. Polyanovsky, M. Roytberg, and V. Tumanyan, "Comparative analysis of the quality of a global algorithm and a local algorithm for alignment of two sequences," *Algorithms for molecular biology : AMB*, vol. 6, p. 25, 10 2011.

[6] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[7] S. B. Needleman and C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[8] X. Huang and K.-M. Chao, “A generalized global alignment algorithm,” *Bioinformatics*, vol. 19, no. 2, pp. 228–233, 01 2003.

[9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool,” *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022283605803602

[10] M. Kuang, Y. Zhang, T.-W. Lam, and H.-F. Ting, “Mlprobs: A data-centric pipeline for better multiple sequence alignment,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 20, no. 1, pp. 524–533, 2023.

[11] S. Sahoo, S. P. Routray, D. S. K. Nayak, and T. Swarnkar, “An enhanced web-based tools for multiple sequence alignment: A comparative approach,” in *2022 3rd International Conference on Computing, Analytics and Networks (ICAN)*, 2022, pp. 1–5.

[12] F. Zhang, S. Angizi, J. Sun, W. Zhang, and D. Fan, “Aligner-d: Leveraging in-dram computing to accelerate dna short read alignment,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 13, no. 1, pp. 332–343, 2023.

[13] R. Langarita, A. Armejach, J. Setoain, P. Ibáñez-Marín, J. Alastruey-Benedé, and M. Moretó, “Compressed sparse fm-index: Fast sequence alignment using large k-steps,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 19, no. 1, pp. 355–368, 2022.

[14] S. Park, H. Kim, T. Ahmad, N. Ahmed, Z. Al-Ars, H. P. Hofstee, Y. Kim, and J. Lee, “Saloba: Maximizing data locality and workload balance for fast sequence alignment on gpus,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 728–738.

[15] Z. Wang, Y. Mou, K. Han, P. Wang, and Y. Shi, “Bio-informatics sequence alignment technology based on estimate algorithm,” in *2022 IEEE 2nd International Conference on Electronic Technology, Communication and Information (ICETCI)*, 2022, pp. 322–325.

[16] C. E. R. Alves, E. N. Cáceres, and F. Dehne, “Parallel dynamic programming for solving the string editing problem on a cgm/bsp,” in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA ’02. New York, NY, USA: Association for Computing Machinery, 2002, p. 275–281.

[17] S. T. Thant Sin, “The parallel processing approach to the dynamic programming algorithm of knapsack problem,” in *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, 2021, pp. 2252–2256.

[18] A. H. Wright, “Approximate string matching using withinword parallelism,” *Software: Practice and Experience*, vol. 24, no. 4, pp. 337–362, 1994.

[19] 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.

[20] 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.

[21] A. Goyal, H. J. Kwon, K. Lee, R. Garg, S. Y. Yun, Y. H. Kim, S. Lee, and M. S. Lee, “Ultra-fast next generation human genome sequencing data processing using dragentm bio-it processor for precision medicine,” *Open Journal of Genetics*, vol. 7, no. 1, pp. 9–19, 2017.

[22] R. E. Bellman, *Dynamic Programming*. New York: Dover Publications Inc, 2013.

[23] 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.

[24] 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.

[25] A. Rashed, H. Amer, M. El-Seddek, and H. E.-D. Moustafa, “Sequence alignment using machine learning-based needleman–wunsch algorithm,” *IEEE Access*, vol. PP, pp. 1–1, 07 2021.

[26] L. Cai, X. Huang, C. Liu, F. Rosamond, and A. Song, “Parameterized complexity and biopolymer sequence comparison,” *Comput. J.*, vol. 51, 05 2008.

[27] 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.

[28] 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.

[29] Z. Ke and V. Ng, “Automated essay scoring: A survey of the state of the art.” in *IJCAI*, vol. 19, 2019, pp. 6300–6308.

[30] 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.

[31] Y. Jararweh, M. Al-Ayyoub, M. Fakirah, L. Alawneh, and B. B. Gupta, “Improving the performance of the needleman-wunsch algorithm using parallelization and vectorization techniques,” *Multimedia Tools and Applications*, vol. 78, pp. 1–17, 02 2019.

[32] D. D. Shrimankar and S. R. Sathe, “Performance analysis of openmp and mpi for nw algorithm on multicore architecture,” *International Journal of Advanced Studies in Computers, Science and Engineering*, vol. 3, no. 6, pp. 23–34, 2014, copyright - Copyright International Journal of Advanced Studies in Computers, Science and Engineering 2014.

[33] 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.

[34] B. Barney, “Introduction to parallel computing tutorial,” https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial.

[35] https://www.openmp.org/.

[36] B. Barney, "LLNL HPC Tutorials: OpenMP," https://hpc-tutorials.llnl.gov/openmp/.

[37] S. R. Sathe and D. D. Shrimankar, "Parallelization of dna sequence alignment using openmp," in *Proceedings of the 2011 International Conference on Communication, Computing and amp; Security*, ser. ICCCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 200–203.

[38] https://www.open-mpi.org/.

[39] B. Barney, "LLNL HPC Tutorials: Message Passing Interface(MPI)," https://hpc-tutorials.llnl.gov/mpi/.

Dr. Mushtaq Ahmed is presently working as an Associate Professor in the Department of Computer Science and Engineering at Malaviya National Institute of Technology Jaipur, Rajasthan, India. He has more than 23 years of teaching experience. His research domain includes Network on Chip, Embedded Systems, Parallel Computing, Cloud Computing, Wireless Sensor Networks, Fault Tolerant Systems, and High speed Overlay Networking. He has published more than fifty papers in various International Journals and conferences. He is a Fellow IE(India), Life Member of ACM, Life member of ISTE, Member IEEE, and Member IEANG.

Akash Yadav is a research scholar in the Department of Computer Science and Engineering at Malaviya National Institute of Technology Jaipur, Rajasthan, India. He earned a Master's Degree in Information Security from Motilal Nehru National Institute of Technology Allahabad, UP, India. His research focuses on compilers, parallel programming, algorithms, and theory of computations.

Alina Khan is under-graduate student pursuing MBBS from Karaganda Medical University, Republic of Kazakhstan.