Computing and Informatics, Vol. 39, 2020, 757–779, doi: 10.31577/cai_2020_4_757

View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Computing and Informatics (E-Journal - Institute of Informatics,...

DISTRIBUTED ALGORITHM FOR PARALLEL EDIT DISTANCE COMPUTATION

Muhammad Umair Sadiq, Muhammad Murtaza Yousaf

Punjab University College of Information Technology (PUCIT)
University of the Punjab
Lahore, Pakistan
e-mail: {umair.sadiq, murtaza}@pucit.edu.pk

Abstract. The edit distance is the measure that quantifies the difference between two strings. It is an important concept because it has its usage in many domains such as natural language processing, spell checking, genome matching, and pattern recognition. Edit distance is also known as Levenshtein distance. Sequentially, the edit distance is computed by using dynamic programming based strategy that may not provide results in reasonable time when input strings are large. In this work, a distributed algorithm is presented for parallel edit distance computation. The proposed algorithm is both time and space efficient. It is evaluated on a hybrid setup of distributed and shared memory systems. Results suggest that the proposed algorithm achieves significant performance gain over the existing parallel approach.

Keywords: Edit distance, dynamic programming, parallel computing, distributed memory system, MPI, OpenMP, speedup

1 INTRODUCTION

Measuring the similarity between two strings helps to solve problems in many domains such as spell checking, spam filtering, nucleotide sequence matching, virus signature matching in computer security, natural language processing (NLP), speech recognition, and pattern recognition [1, 2, 3, 4, 5]. String similarity/matching comes in two forms: approximate string matching and exact string matching [6]. In the exact string matching, all the appearances of the pattern are required to be found in the given string. In the approximate string matching, the difference between the given pattern and the string is measured. Levenshtein distance is the measure that

tells the difference between two strings. It counts the number of edit operations (insert, replace, and delete) that are required to transform one string to another [7]. In literature, it is often referred to as edit distance [8, 9], but some other definitions of edit distance exist as well [10]. In this study, we would consider the Levenshtein's definition of the edit distance.

Sequentially, edit distance/Levenshtein distance can be computed by using dynamic programming based strategy but if large strings such as deoxyribonucleic acid (DNA) sequences are compared, then it may not give result in a reasonable amount of time. Therefore, a parallel solution is required to compute result in acceptable time if the data size is large. This work is about design and evaluation of a distributed algorithm for parallel edit distance computation between two strings. The proposed algorithm is evaluated on a cluster by using Message Passing Interface (MPI). MPI is designed to work with different parallel architectures and it serves as a standard [11]. It defines certain point to point and the collective communication protocols to program distributed parallel systems. The rest of the paper is organized as follows: Section 2 discusses some preliminary concepts. Section 3 presents the background of the edit distance. Section 4 covers the related work. Section 5 presents a distributed algorithm for parallel edit distance computation. Section 6 presents experimental evaluation of the proposed algorithm. Finally, Section 7 concludes the paper and discusses the artery of future work.

2 PRELIMINARIES

This section introduces some basic terminologies and key concepts that will be used in the rest of the paper.

2.1 Cost of Communication

While designing a distributed algorithm, computation, as well as communication time between different nodes, is also essential. For analysis of communication time, consider this model of communication. The time required to communicate a message between two nodes of a distributed memory system is equal to $t_s + t_w\eta$ where t_s is startup time to prepare the message for transmission, t_w is per word transfer time, and η is number of words [12].

2.2 Exclusive Scan Operation

Exclusive scan operation is an important primitive in parallel computing. It operates on an ordered set $[x_0, x_1, \dots, x_{n-1}]$ of n elements. It uses a binary associative operator $\oplus$. It returns the result of form: $[-, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \dots, x_0 \oplus x_1 \oplus x_2 \oplus \dots \oplus x_{n-1}]$. In the output each jth element is cumulative result of all input elements from 0th to j^{th} element (excluding j^{th} element itself). If all n elements are divided among p processors then this operation requires $(t_s + t_w\eta)\log n$ time on distributed memory system [12].

2.3 Speedup

In parallel computing, speedup is the measure of the increase of performance of parallel algorithm compared to sequential algorithm. It is the ratio of sequential execution time to parallel execution time.

2.4 Parallel Efficiency

Parallel efficiency is the measure of effectiveness of the resource utilization. It is the ratio of speedup to the number of compute nodes.

3 SEQUENTIAL COMPUTATION OF THE EDIT DISTANCE

Sequentially, edit distance is computed by using dynamic programming based strategy in which a table Lev of size $(m+1) \times (n+1)$ is built where m is size of first string and n is size of second string [7]. Given two strings A and B of size m and n, respectively, edit distance table can be computed with Algorithm 1.

Each cell (i, j) in the edit distance table represents the value of the edit distance between the first i characters of string A and the first j characters of string B. Cell (m, n) in the table represents the value of the edit distance between both strings. Figure 1 shows a sample edit distance table for strings "ACER" and "CARE".

Edit distance between "ACER" and "CARE" is 3 because to make this conversion following operations are required: delete 'A', match 'C', replace 'E' with 'A', match 'R', and delete 'E'. There is no cost for matching a character. Figure 1 also illustrates the dependence of the calculation of a single cell on other cells of the edit distance table.

Sequential time complexity of the algorithm is $O(mn)$ which is obvious from the size of the edit distance table. Space complexity of the algorithm is $O(n)$ [7].

4 RELATED WORK

To reduce the computation time of the edit distance many efforts have been made and this section covers various such studies.

Masek and Paterson [8] presented a little restricted but fast sequential edit distance algorithm for abstract unit-cost RAM machine. It requires the strings to be of equal sizes. Mathies [9] presented a fast parallel algorithm for the edit distance computation. It requires mn processors on abstract parallel random-access machine. Apostlico et al. [13] presented a parallel edit distance algorithm for the abstract parallel random-access machine. A space efficient algorithm for the edit distance is presented in [14].

A bit parallel algorithm for the problem of approximate string matching [15] is presented in [16]. This algorithm is serial and depends upon the word size of machine. It allows to process only w cells at a time where w is the largest word

Algorithm 1: Sequential computation of the edit distance [7].

```
Input: Strings: A[0...m−1] and B[0...n−1]
1  m ← A.length
2  n ← B.length
3  Let Lev[0...n], LevP[0...n] be new arrays
4  for j = 0 to n do
5  |   Lev[j] ← j
6  end
7  LevP ← Lev
8  for i = 1 to m do
9  |   for j = 0 to n do
10 |   |   if j = 0 then
11 |   |   |   Lev[j] ← i
12 |   |   else if A[i−1] = B[j−1] then
13 |   |   |   Lev[j] ← LevP[j−1]
14 |   |   else
15 |   |   |   Lev[j] ← min(LevP[j], LevP[j−1], Lev[j−1]) + 1
16 |   |   end
17 |   end
18 |   if i ≠ n then
19 |   |   Swap(Lev, LevP)
20 |   end
21 end
Output: Lev[n]
```

size on a given machine. This algorithm is later modified to compute the edit distance and is presented in [10]. In [17], Myers bit parallel algorithm [16] is also implemented on GPUs using collaborative parallelization for large bitwise operations and concurrent pattern matching. An implementation of Myers algorithm [16] is also presented in [18].

An obvious way to compute the edit distance is to use diagonal parallel approach. It calculates the edit distance table diagonal-wise by simultaneously computing all entries in a diagonal. Its main disadvantage is unbalanced workload among processors because sizes of the diagonals vary in each step [19]. In [20], an efficient parallel algorithm for longest common sub-sequence problem is presented for shared memory multi-core systems and GPUs. Sadiq et al. [21] presented a parallel algorithm for the edit distance problem. Authors resolved the dependences in the dynamic programming table and manage to calculate the edit distance table row-wise where every row is computed in parallel. An additional preprocessing step is added which helps in resolving the dependences in the dynamic programming table. Their proposed algorithm is evaluated on GPUs and multi-core systems. It achieved good

			C	A	R	E
	i, j	0	1	2	3	4
	0	0	1	2	3	4
A	1	1	1	1	2	3
C	2	2	1	$Lev[i,j] = 2$	2	3
E	3	3	2	2	3	2
R	4	4	3	3	2	3

Figure 1. Edit distance table between strings: "ACER" and "CARE" and dependences of a cell (i, j) in the edit distance table

performance gains. Similar strategies to [21] for the problem of approximate string matching have been proposed in [22, 23].

Another problem that is related to the edit distance is the sequence alignment problem. In that, similar regions of two sequences are aligned together by inserting gaps in the sequences. Major algorithms of the sequence alignment problem also follow the dynamic programming. Furthermore, their solutions have similar dependences in the dynamic programming table as in the case of edit distance. Comprehensive studies have been made for this problem. Aluru et al. [24] presented a distributed algorithm for various algorithms of biological sequence comparison. It introduces the way to calculate the alignment table row-wise or column-wise by using parallel scan operation [12, 25]. This algorithm is implemented by using MPI. Results are presented for two type of sequences: one having complete match case and other having complete mismatch case. This algorithm achieved good speedup on both setups. Scalability of their algorithm is evaluated by a varying number of processors. Authors indicated that their method can be used to parallelize other algorithms that needed to calculate such a score table. An exact parallel space and time optimal algorithm for the sequence alignment is proposed in [26]. It also uses the parallel scan operation to exploit parallelism. Furthermore, it is important to note that edit distance can also be computed using the parallel scan operation [25].

In [27] two streaming algorithms for biological sequence alignment are presented for GPUs. These streaming algorithms are also based on diagonal parallel approach. A parallel algorithm for local alignment is presented in [28]. It uses a master and slave model. This algorithm is also space efficient. It is evaluated by using MPI and

cluster of eight and sixty nodes. The authors evaluated their algorithm by using random pair of sequences in the range of 1 KBP (Kilo Base Pairs) and 1 600 KBP.

A parallel and space efficient algorithm for sequence alignment called *z-align* is presented in [29]. The authors executed their algorithm in four phases:

1. distribution of input data including sequences,
2. calculation of the similarity matrix,
3. gathering of the best score and their coordinates at the master processor,
4. obtaining actual alignment(s) in limited space using a master-slave model and self-scheduling policy.

This algorithm is evaluated on a cluster of sixteen processors. The authors compared sequences of size between 1 KB and 3 MB. A parallel algorithm for multiple sequence alignment is proposed in [30]. It is evaluated using a cluster of systems connected through network. Parallelism is achieved by partitioning the dynamic programming matrix among host systems. The authors also evaluated the scalability of the algorithm.

To summarize, the following are major principles on which parallel computation of the edit distance has been done. Some earlier solutions are based on a parallel random access machine model that is really fast in terms of time complexity [8, 9, 13], but due to the quadratic space complexity and resource requirements such models are not implemented practically. Bit parallel approach [10, 16, 17] is quite common, in which a couple of cells are represented as a single word and bitwise operations are used to perform simultaneous computations. Another approach that is quite common is diagonal based approach [19] in which edit distance table is solved anti-diagonal-wise, while computing each anti-diagonal in parallel. There is not dependence among the cells of anti-diagonal according to the algorithm presented as Algorithm 1. It can be further seen in the Figure 1. The number of cells in each anti-diagonal is different, what lowers the parallelism in certain stages of the algorithm. Furthermore, it is also not a load balanced approach. Another way of solving the edit distance table is to use the parallel scan approach, which resolves the dependences in the table and computes it row-wise. It increases the number of steps for computation of each row, but it is the load balanced approach. Another method of resolving the dependences in the dynamic programming table involves a preprocessing step [21, 22, 23]. After changing the dependences, updated algorithm computes entries of each row of the table in parallel.

Overall, extensive studies have been made for parallel computation of the edit distance and related problems. Most of the studies primarily focuse on GPU-based solution. In string comparison, problem size can be very large, therefore the scalable solution is required for these problems. A solution for distributed memory systems is always a good choice for scaling because there is no limit to the number of processing nodes. To the best of our knowledge, distributed solutions exist [24, 26, 28] in literature but those are not specific to the edit distance. Therefore, this study focuses on designing a distributed algorithm for parallel edit distance calculation.

5 A DISTRIBUTED ALGORITHM FOR PARALLEL EDIT DISTANCE COMPUTATION

To compute multiple entries of the edit distance table simultaneously, dependences for computing each cell should be investigated. Computations in the first row and the first column of the edit distance table are independent of any cell (Algorithm 1). For any cell (i, j) in i^{th} row, where $i > 0$ and $j > 0$, there are two possibilities: either cell (i, j) can be a *match case* where the character of string A matches with the corresponding character of string B. Otherwise, it will be a *non-match case*.

According to Algorithm 1, to compute any cell (i, j) which is a *non-match case*, three values should be known in advance: the value of left cell $(i, j-1)$, the value of upper cell $(i-1, j)$, and the value of diagonal cell $(i-1, j-1)$ (Figure 1). To compute any other cell in the edit distance table which is not a *non-match case*, only the values from $(i-1)^{\text{st}}$ row are required.

Yousaf at el. [31] presented a parallel algorithm for the edit distance computation that resolves the dependences in the edit distance table. This algorithm computes all cells in the ith row of the edit distance table simultaneously based on the $(i-1)^{\text{st}}$ row only. Yousaf at el. [31] proved that the value of a *non-match case* can also be computed from $(i-1)^{\text{st}}$ row with the following equation:

$$Lev[j] = \min(LevP[j] + 1, LevP[j-1] + 1, LevP[mp-1] + k).$$

Here mp is the position of the last *match case* in the ith row and k is the distance of cell (i, j) from last *match case*. According to this new equation, dependences of the cell (i, j) of the edit distance table are changed (as illustrated in Figure 2). Based on these new dependences each cell (i, j) of an ith row can be computed based on $(i-1)^{\text{st}}$ row.

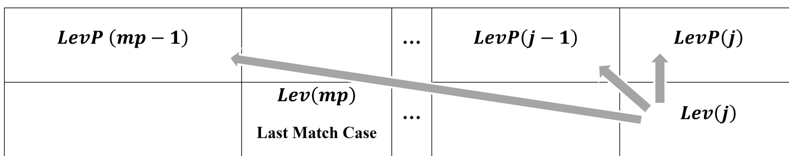


Figure 2. Dependences of cell (i, j) according to parallel edit distance algorithm presented in [31]

Given character set Σ having $|\Sigma|$ number of unique characters, last *match case* for each cell in an ith row can be found by computing a Last Match Case Table (LMT) of size $(|\Sigma| - 1) \times n$. It contains the last match positions of the unique characters of character set Σ against string B. It can be computed by using Algorithm 2.

Table 1 shows a sample LMT for character set $\{A, C, G, T\}$ and String = "ABACUS".

Algorithm 2: Computing LMT

```
    Input: Character set Σ and String B
 1  |Σ| ← Σ.length
 2  n ← B.length
 3  Let LMT[0...|Σ|−1, 0...n] be a new table
 4  for i = 0 to |Σ|−1 do
 5  |   for j = 0 to n do
 6  |   |   if j = 0 then
 7  |   |   |   LMT[i][j] ← 0
 8  |   |   else if match case then
 9  |   |   |   LMT[i][j] ← j
10  |   |   else
11  |   |   |   LMT[i][j] ← LMT[i][j−1]
12  |   |   end
13  |   end
14  end
```

	0	1 A	2 B	3 A	4 C	5 U	6 S
A	0	1	1	3	3	3	3
C	0	0	0	0	4	4	4
G	0	0	0	0	0	0	0
T	0	0	0	0	0	0	0

Table 1. LMT for $\Sigma = \{A, C, G, T\}$ and String = "ABACUS"

The value of the last match position can be incorporated with the following equation:

$$Lev[j] = \min(LevP[j] + 1, LevP[j-1] + 1, LevP[LMT[c][j] - 1] + (j - LMT[c][j])).$$

Here $LMT[c][j]$ is the position of the last *match case* in an ith row. This algorithm computes the same edit distance score as sequential algorithm. Its proof of correctness is established in [31] and [21]. Furthermore, the size of each row is the same in the edit distance table, therefore this approach allows balanced work division among the processing elements.

Now, let us introduce a distributed way of computing edit distance based on [31].

5.1 Distributed Algorithm

The proposed method is divided into three parts:

1. Distribution of the strings,

2. Distributed computation of the LMT,
3. Distributed computation of edit distance table using LMT.

Assume that there are p processors having ID in the range of 0 to $p-1$. For simplicity, also assume that n (size of string B) is divisible by p. However, it can be generalized for an arbitrary number of processors. Now, we will discuss all three steps one by one.

5.1.1 Distribution of the Strings

Initially, strings are distributed among each computing machine as plain text files. Only a respective part of the string is moved to main memory on which processing is required. Edit distance table can be computed row by row by reformulating the dependences (as shown in [21, 31]). Each row can be divided among multiple processors. So, every processor will compute $O\left(\frac{n}{p}\right)$ part of the row. To compute its part of the row each processor needs $O\left(\frac{n}{p}\right)$ fraction of the string B, therefore each processor p_r gets second string from $B[r(\frac{n}{p})]$ to $B[(r+1)\frac{n}{p}]$ where r is ID of the processor. This process is illustrated in Figure 3 which shows the distribution of a row and string B among p processors.

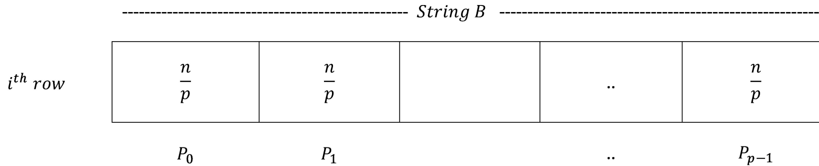


Figure 3. Distribution of a row of the edit distance table and string B among p processors

In the edit distance table, computing a row requires only one character from string A. So, string A can be obtained in chunks by each processor. Every time processing on one chunk is completed, next chunk is obtained.

5.1.2 Distributed Computation of the *LMT*

The computation of the LMT is divided in equal parts of $\frac{n|\Sigma|}{p}$ among the p processors (as illustrated in Figure 4).

LMT is computed row by row by modifying Algorithm 2. Each processor can compute its part of ith row of the LMT in two steps:

1. In the chunk, all values after first *match case* can be computed in a similar manner to Algorithm 2 and all the values before first *match case* are undefined initially.

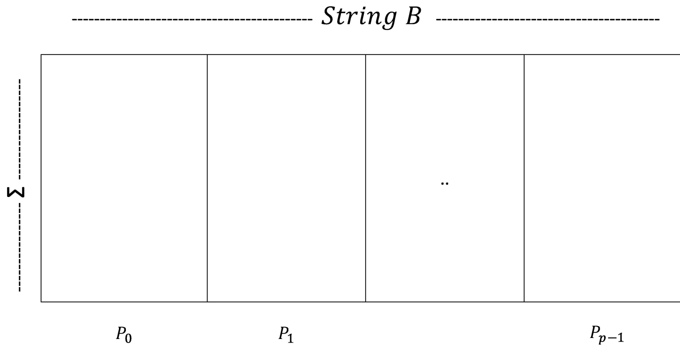


Figure 4. Distribution of the LMT among p processors

2. In any row of the LMT, values always increase or remain the same. So, all the values before first *match case* can be found by an exclusive scan operation (with maximum as binary associative operator) on extreme right value in the chunk of each processor. This value can be used in the place of undefined values.

Hence with these steps, each processor has all required values of ith row of the LMT. Now consider an example of computation of the LMT where a row of the table is computed for a character 'A' and string "ATACG". The whole row is divided among four processors in $\frac{n}{p}$ chunks where the size of each chunk is two. Table 2 illustrates both the steps to compute a row of the LMT.

(A)	0 $P_0(0)$	1 $P_0(1)$ A	2 $P_1(0)$ T	3 $P_1(1)$ A	4 $P_2(0)$ C	5 $P_2(1)$ G
Values computed after step 1	0	1	–	3	–	–
Values received after step 2	–		1		3	

Table 2. Steps to compute LMT (for a character 'A' and string "ATACG")

Similarly, all rows of the LMT can be computed. At one time, all the processors will be computing their chunk of one specific row. After computation of a row, all processors are synchronized. Then next row is computed. The procedure to compute the LMT for an arbitrary process p_r is presented as Algorithm 3. Algorithm 3 takes string B and character set as input and computes the LMT.

5.1.3 Distributed Computation of the Edit Distance Table

The computation of the edit distance table is also divided in equal parts of $\frac{mn}{p}$ among p processors (as illustrated in Figure 5).

Each processor will compute $\frac{mn}{p}$ part of the edit distance table. Edit distance table is computed row by row, therefore to store current and previous row of the edit

---

Algorithm 3: Computation of the LMT at processor p_r where r is a unique identifier for the process between $[0 \ldots p-1]$

Input: Character set Σ and String B

1 $|\Sigma| \leftarrow \Sigma.length$
2 $n \leftarrow B.length$
3 $B_r \leftarrow B[r\frac{n}{p} \ldots r(\frac{n}{p}+1)]$
4 Let $LMT[0 \ldots (|\Sigma|-1), 0 \ldots \frac{n}{p}]$ be a new table
5 **for** $i = 0$ *to* $|\Sigma| - 1$ **do**
6     $jc \leftarrow$ initial index of a row in p_r's chunk
7   **for** $j = 0$ *to* $\frac{n}{p}$ **do**
8   **if** $jc = 0$ **then**
9             $LMT[i][j] \leftarrow 0$
10   **else if** $B_r[j] \neq \Sigma[i]$ *and* $j = 0$ **then**
11             $LMT[i][j] \leftarrow undef$
12   **else if** $B_r[j] == \Sigma[i]$ **then**
13             $LMT[i][j] \leftarrow jc$
14   **else**
15             $LMT[i][j] \leftarrow LMT[i][j-1]$
16   **end**
17         $jc++$
18   **end**
19   Values before first *match case* $\leftarrow$ Exclusive Scan (Max) on $LMT[|\Sigma|-1][\frac{n}{p}-1]$
20 **end**

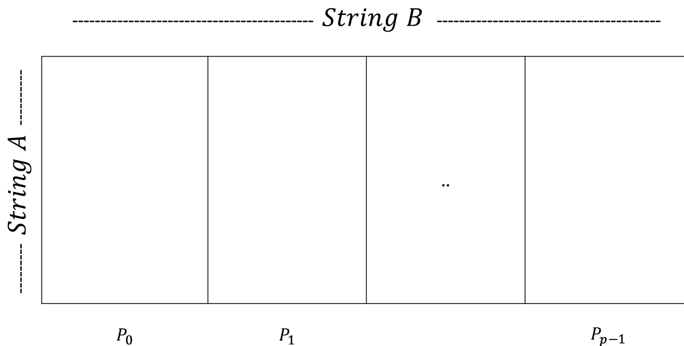


Figure 5. Distribution of the edit distance table among p processors

distance table, each process will allocate $O\left(\frac{n}{p}\right)$ space for both rows. At one time, all processors will be computing their part of one specific row. After computation of the row, all processors are synchronized. Then the next row is computed, and the same process follows. Computation of an ith row of the edit distance table is done in similar manner to Algorithm 1 but the following two cases must be delt with.

1. To compute the extreme left value in chunk of each processor, the value of its diagonal cell is required (Figure 2), which is not available locally. It can be found from the preceding processor.

2. It is possible that for some initial cells in the chunk of a processor, the value of the last *match case* lies in the chunk of the preceding processor/s (Figure 2). Therefore, that value must be obtained before computation of the ith row.

These both cases can be managed by two communication steps. The first case can be handled as follows: Before computation of an ith row, every p^{th} processor communicates its extreme right value of the $(i-1)^{\text{st}}$ row with $(p+1)^{\text{st}}$ processor. To handle the second case, an exclusive scan operation (with maximum as binary associative operator) is performed with the value at very last *match case* in each processor's chunk (as there can be more than one *match cases* in each processor's chunk).

Now, each processor has every value to compute its part in i^{th} row of the edit distance table. Furthermore, all the values in the chunk of each processor can also be computed in parallel as their computation is dependent only on $(i-1)^{\text{st}}$ row [31]. In multi-core system, we can divide this computation further among multiple threads. Similarly, all the rows can be computed. The procedure to compute the edit distance table is presented as Algorithm 4 which shows working of an arbitrary process p_r. This algorithm takes strings A and B as input and computes the value of the edit distance.

5.2 Analysis of the Algorithm

5.2.1 Computational Complexity

LMT and the edit distance table are computed row by row and each row is equally divided among processors. Therefore, each processor computes $O(\frac{n|\Sigma|}{p})$ part of the *LMT* and $O(\frac{mn}{p})$ part of the edit distance table. So, total computational complexity is $O(\frac{mn}{p})$ because $\frac{n|\Sigma|}{p} < \frac{mn}{p}$. Edit distance table is further divided among the available threads in case of hybrid implementation. If number of available threads is t, then computation time is reduced to $O(\frac{mn}{pt})$ because chunk of row for each process is further divided among the cores of a multi-core system. Hence, the computation of a row would take $O(\frac{n}{pt})$ time.

Algorithm 4: Computation of edit distance table at processor p_r

Input: Strings: A and B, and LMT

1 $m \leftarrow A.length$
2 $n \leftarrow B.length$
3 $B_r \leftarrow B[r\frac{n}{p} \ldots r(\frac{n}{p}+1)]$
4 $Ac \leftarrow A[0 \ldots \frac{n}{p}]$
5 Let $Lev[0 \ldots \frac{n}{p}]$, $LevP[0 \ldots \frac{n}{p}]$ be new arrays
6 $j_initial \leftarrow$ initial index of a row in p_r's chunk
7 $jc \leftarrow j_initial$
8 **for** $j = 0$ *to* $\frac{n}{p} - 1$ **do**
9     $Lev[j] \leftarrow jc$
10     $jc{+}{+}$
11 **end**
12 $LevP \leftarrow Lev$
13 **for** $i = 1$ *to* m **do**
14   Get next chunk of A if required
15     $ch \leftarrow$ next character in A
16     $jc \leftarrow j_initial$
17   **for** $j = 0$ *to* $\frac{n}{p} - 1$ **do**
18   **if** $jc = 0$ **then**
19             $Lev[j] \leftarrow i$
20   **else if** $Ac[i] = B_r[j]$ **then**
21   **if** $j == 0$ **then**
22                 $Lev[j] \leftarrow pre_end_value$
23   **else**
24                 $Lev[j] \leftarrow LevP[j-1]$
25   **end**
26   **else**
27             $c \leftarrow$ row index of character $Ac[i]$ in LMI
28             $lmp \leftarrow LMI[c][j]$
29             $lmv \leftarrow$ Value at last *match case* according to lmp
30             $Lev[j] \leftarrow \min(LevP[j]+1, LevP[j-1]+1, (jc - lmp) + lmv)$
31   **end**
32         $jc{+}{+}$
33   **end**
34   **if** $i \neq n$ **then**
35   **if** $r \neq p-1$ **then**
36             $end_value \leftarrow Lev[\frac{n}{p} - 1]$
37   Send end_value to processor p_{r+1}
38   **end**
39   **if** $r \neq 0$ **then**
40   Receive pre_end_value from processor p_{r-1}
41   **end**
42         $lmv \leftarrow$ Exclusive Scan (Max) on very last *match case* in p_r's chunk
43   Swap(Lev, $LevP$)
44   **end**
45 **end**

Output: $Lev[\frac{n}{p} - 1]$ if $(r == p-1)$

5.2.2 Communication Time

Total communication required for the LMT is $|\Sigma|(t_s+t_w\eta)\log n$ because to compute one row of the LMT, one exclusive scan operation is required. In the edit distance table, one exclusive scan operation per row is required. Furthermore, for each row one extreme right value in the chunk of the processor should also be communicated. Its communication time is $m(t_s+t_w\eta)(\log n+1)$. So, the total communication time for this algorithm is $|\Sigma|(t_s+t_w\eta)\log n + m(t_s+t_w\eta)(\log n+1)$.

5.2.3 Space Complexity

Space requirement for string B is $O\left(\frac{n}{p}\right)$. LMT requires $O\left(\frac{n|\Sigma|}{p}\right)$ space. Edit distance table requires $O\left(\frac{n}{p}\right)$ space. String A can be obtained by processors in arbitrary sized multiple chunks, but if the chunk size taken is less than $O\left(\frac{n}{p}\right)$ then space complexity would be optimal. Hence, the total space complexity would be $O\left(\frac{n}{p}\right)+O\left(\frac{n|\Sigma|}{p}\right)$. Here $|\Sigma|$ is constant. So, the overall space complexity is $O\left(\frac{n}{p}\right)$.

5.2.4 Comparison with the Existing Algorithms

Table 3 shows the time and space complexity of algorithms that are closely related to the edit distance. Although some of them are not proposed for the distributed memory environment, this comparison suggests that our proposed algorithm is equally efficient as most of the state-of-the-art algorithms in terms of time and space complexity.

Algorithm	Time Complexity	Space Complexity
Huang [14]	$\frac{(m+n)^2}{p}$	$\frac{m+n}{p}$
Myers [16], Chacón et al. [17]	$\frac{mn}{w}$	mn
Šošić and Šikić [18]	$\frac{mn}{w}$	$m+n$
Sadiq et al. [21]	$\frac{mn}{p}$	$m+n$
Aluru et al. [24]	$\frac{mn}{p}$	$m+\frac{n}{p}$
Rajko and Aluru [26]	$\frac{mn}{p}$	$\frac{m+n}{p}$

Table 3. Space and time complexity of the existing algorithms related to the edit distance

Here m and n are lengths of string A and string B, respectively, w is the maximum word size that a machine can process, and p is the number of processing units. Faster algorithms [9, 13] are proposed for the parallel random access machine (PRAM) model but they are never implemented practically. Their space complexity is also quadratic.

6 EXPERIMENTS AND RESULTS

We have used five-nodes cluster with a minimum specification of one node: Intel Core-i5-3570K 3.40 GHz CPU having 4 physical cores, 4 logical processors, and 8 GB of main memory. All nodes in the cluster are interconnected to centralized hub by using fast ethernet cables. Furthermore, we used MPI (mpich version 3.2) for implementation of the algorithms. Four processes are launched at each node of the cluster for the pure MPI implementation (twenty processes in total for all nodes in the cluster). We have also used OpenMP for parallelism on one node. In the OpenMP implementation, computation inside the chunk of a row of each process is divided among multiple threads. Those threads are mapped into multiple cores. We have used OpenMP pragmas for static division of the work among the multiple threads. In the hybrid experiments using MPI + OpenMP, total five processes are launched (one for the each node of the cluster), where each process launches four threads to completely utilize all the cores of one node in the cluster.

We compared the proposed algorithm with an existing parallel scan approach [24] that involves a higher number of steps than the proposed algorithm. Experiments are performed for two types of datasets:

- Random strings
- Real DNA strings obtained from the National Center for Biotechnology Information website (NCBI) [32]

6.1 Experiments with Random Strings

In the first set of experiments, we have used strings in the range of 100 000 to 1 000 000. Strings are generated randomly from twenty-six letters in the Latin alphabets. In each experiment, strings of equal size are compared.

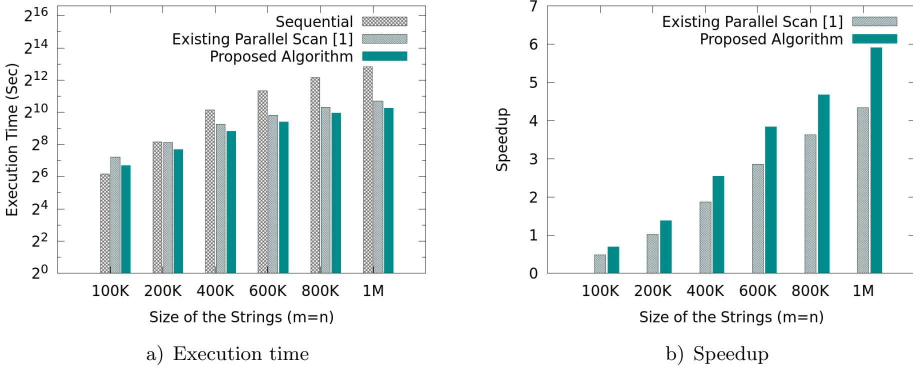
6.1.1 Distributed Memory Setup (MPI)

Results for the MPI by using randomly generated strings show that the proposed algorithm achieved speedup up to 5.90× and the existing parallel scan approach [24] achieved speedup up to 4.33×. It is evident from the results that with increase in the problem size, the performance gain of the proposed algorithm is larger than the existing parallel scan approach. Figure 6 shows the scaled execution time and speedup for different sizes of problem with randomly generated strings.

6.1.2 Hybrid Setup (MPI + OpenMP)

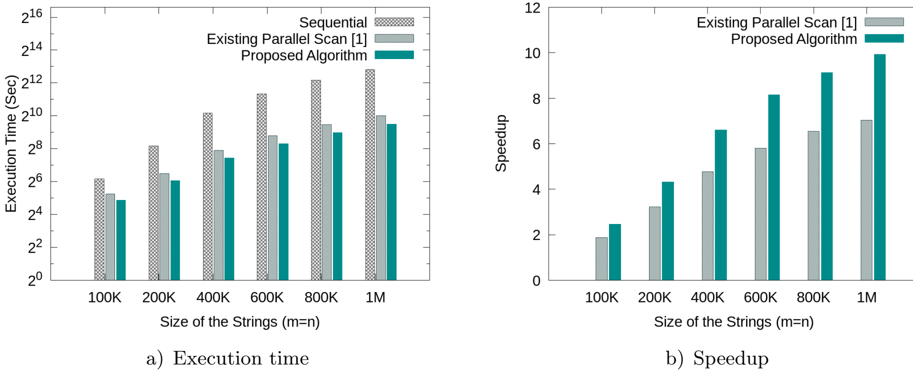
Results show that the proposed algorithm achieved speedup up to 9.93× and the existing parallel scan approach [24] achieved speedup up to 7.04×. Figure 7 shows the results for hybrid setup of MPI and OpenMP.

Results show that by using hybrid solution maximum attained speedup of the proposed algorithm is improved from 5.90× to 9.93×. In the existing parallel scan



a) Execution time b) Speedup

Figure 6. Results and comparisons for MPI with randomly generated strings



a) Execution time b) Speedup

Figure 7. Results and comparisons for hybrid (MPI + OpenMP) setup with randomly generated strings

approach, maximum attained speedup is improved from 4.33× to 7.04×. Hence, in the MPI-only implementation the performance is slower because explicit inter-node communication is required among the processes running on one node of the cluster.

6.2 Experiments with DNA Strings

In the next set of experiments real DNA strings are also compared. DNA strings are taken from NCBI website [32] which maintains a database of many DNA strings. Information of the DNA strings which are compared is presented in Table 4. This table also shows the value of edit distance between each two DNA strings.

Size of the LMT is reduced in DNA strings because the number of characters in character set of the DNA strings is four ($\{A, C, G, T\}$) which is smaller than the

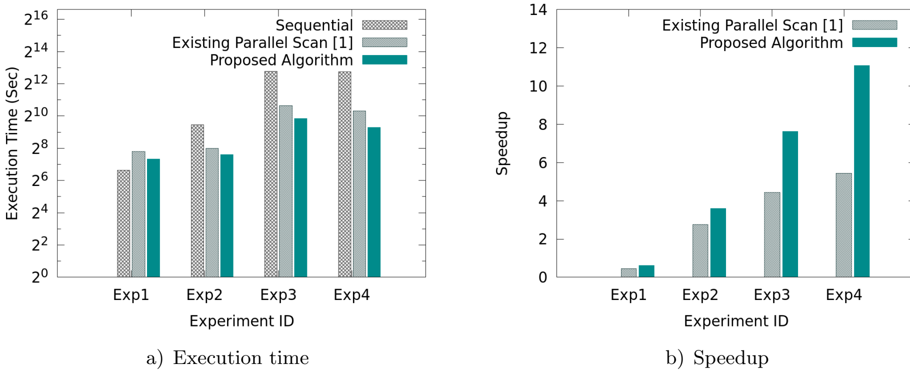
Experiment ID	String A		String B		Edit Distance
	Name	Size	Name	Size	
Exp1	gbgss201	156 931	gbpln104	79 314	91 590
Exp2	gbgss201	156 931	gbhtg11	606 452	450 982
Exp3	gbhtg11	606 452	gbgss116	1 517 819	969 770
Exp4	gbuna1	308 453	gbinv32	3 424 429	3 116 517

Table 4. The list of experiments for DNA strings comparison

character set of randomly generated strings of latin letters. Hence, in this case, less time is required for the processing of the LMT.

6.2.1 Distributed Memory Setup (MPI)

Results on the MPI show that the proposed algorithm achieved speedup up to 11.05× and the existing parallel scan approach [24] achieved speedup up to 5.44×.



a) Execution time

b) Speedup

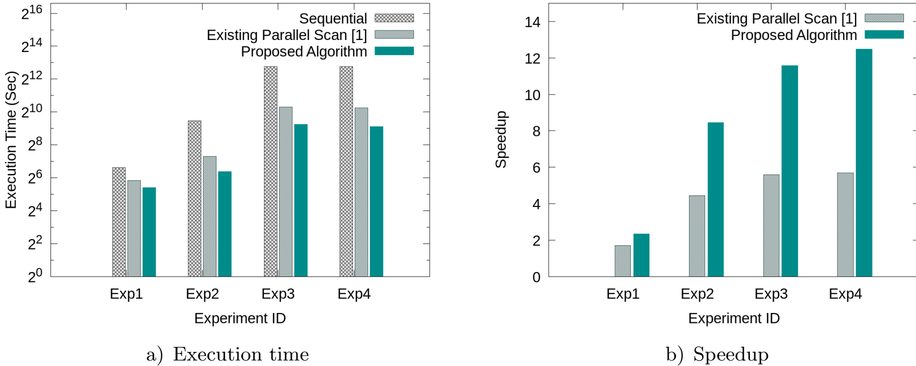
Figure 8. Results and comparisons for MPI-only setup with DNA strings

Figure 8 shows scaled execution time and speedup for the experiments mentioned in Table 4 for MPI.

6.2.2 Hybrid Setup (MPI + OpenMP)

Results on hybrid setup of MPI and OpenMP show that the proposed algorithm achieved speedup up to 12.49× and the existing parallel scan approach [24] achieved speedup up to 5.68×. Speedup is improved compared to the MPI-only results (where maximum obtained speed up is 11.05× and 5.44× for the proposed algorithm and the existing parallel scan approach, respectively). In the MPI-only implementation, explicit inter-node and intra-node communication is required, while in the hybrid implementation, explicit inter-node communication is avoided. Performance gain for the proposed algorithm (by using OpenMP with MPI) is larger compared to the

existing parallel scan approach where proposed method gained added speedup of up to 1.43× and the existing parallel scan method boosted up to 0.24 times. Hence, parallel scan method requires more communication than the proposed algorithm. Figure 9 shows the experimental results on hybrid setup of MPI and OpenMP by using DNA strings.



a) Execution time

b) Speedup

Figure 9. Results and comparisons for hybrid (MPI + OpenMP) setup with DNA strings

Since the computation as well as communication time plays its role in the overall running time of the algorithm, both are equally important. Furthermore, in the proposed method an additional table (LMT) is also computed, therefore it is essential to analyze its running time as well. In the Table 5 and Table 6, detailed communication and computation time is presented (for the proposed algorithm) for the LMT and edit distance table. Results show that time required for the processing of the LMT is negligible compared to the edit distance table. Overall communication time is less than computation time and total running time increases with increase in problem size. Furthermore, it can be observed that communication time for MPI-only experiments is significantly larger than for the hybrid experiments due to added explicit inter-node communication among the processes in case of MPI-only implementation.

An interesting case can be seen in the results of Exp3 and Exp4. For the proposed algorithm, running time of Exp4 is smaller than Exp3 despite the size of the edit distance table being approximately the same in both cases (9.20×10^{11} and 1.05×10^{12} respectively for Exp3 and Exp4). This is due to the fact that in the approximately same sized problems, overall running time would be smaller for the tables having small number of large sized rows (where the size of string A determines the number of rows and the size of string B determines the size of a row). Exp4 has larger sized rows than Exp3, therefore in the one row more parallelism is achieved. It also has a small number of rows which reduces communication and synchronization overhead required after the computation of a row. This observation indicates that running time is small if the computation required is larger than communication,

i.e., the size of a row is large and the number of the rows is small. This indicates that communication and synchronization are more expensive operations than the computation.

Exp. ID	LMT		Edit Distance Table		Total		
	Comp.	Comm.	Comp.	Comm.	Comp.	Comm.	All
Exp1	0.006	0.015	67.437	93.214	67.444	93.229	160.673
Exp2	0.003	0.021	92.653	103.947	92.656	103.967	196.623
Exp3	0.004	0.023	493.335	426.950	493.339	426.973	920.311
Exp4	0.005	0.021	411.978	212.800	411.983	212.822	624.804

Table 5. Detailed communication and computation time (in seconds) for the proposed algorithm using MPI-only implementation

Exp. ID	LMT		Edit Distance Table		Total		
	Comp.	Comm.	Comp.	Comm.	Comp.	Comm.	All
Exp1	0.002	0.002	20.054	22.389	20.056	22.391	42.447
Exp2	0.003	0.001	67.764	15.678	67.767	15.678	83.446
Exp3	0.007	0.002	534.848	71.853	534.856	71.855	606.711
Exp4	0.015	0.001	505.576	47.455	505.591	47.456	553.047

Table 6. Detailed communication and computation time (in seconds) for the proposed algorithm using hybrid setup of MPI and OpenMP

6.3 Summary of the Results

These results suggest that the proposed algorithm significantly outperforms the existing parallel scan approach [24]. It is up to 12.49× faster than a sequential algorithm. It also utilizes given resources effectively compared to the existing parallel scan approach. For the small problem sizes, the sequential algorithm performs better compared to the parallel solutions because of the communication and synchronization overheads of parallel solutions. In the proposed algorithm, time required for the processing of the LMT is negligible compared to the edit distance table. Size of the LMT is smaller for the DNA strings compared to randomly generated strings (as there are only four characters in character set of DNA strings), therefore processing time of the LMT is smaller for DNA strings. Furthermore, in the proposed approach, the overall communication time is smaller compared to the computation time which is always desired because the communication is expensive compared to the computation.

Another important factor that should be considered is parallel efficiency of both algorithms. It is the ratio of speedup and number of compute nodes. Total compute nodes are twenty. Maximum speedup attained for the proposed algorithm is 12.49× and 7.04× for parallel scan approach. Hence, maximum parallel efficiency of the proposed algorithm is 0.62 and it is 0.35 for parallel scan approach.

The reason for this low parallel efficiency is that the explicit synchronization and communication is required at the end of computation of each row. Due to these overheads the parallel efficiency is decreased. In case of the parallel scan approach, parallel efficiency is even lower because it involves more steps which add more parallel overheads. These overheads are unavoidable because of the nature of the algorithm. Moreover, in general, these trade-offs exist when designing the parallel algorithm.

7 CONCLUSION AND FUTURE WORK

In this paper, we introduced a distributed algorithm for the parallel edit distance computation. The proposed algorithm ensures a balanced workload among the processors. To the best of our knowledge, this is the first time when a distributed algorithm of the edit distance is presented. Earlier studies [14, 24] have a mention about distributed computation of edit distance, but its practical implementation is not proposed.

We have presented results for random and real DNA strings. Evaluation on hybrid setup of OpenMP and MPI shows that the proposed algorithm achieved 12.49× speedup compared to the sequential version of the algorithm. Furthermore, it also outperforms the parallel scan approach [24] significantly. There are many other problems that are related to edit distance. Damerau–Levenshtein distance, approximate string matching, longest common sub-sequence, and sequence alignments are an example of such problems. It is good challenge to design their solutions for distributed memory systems. Furthermore, in future we intend to design a distributed memory solution that could exploit parallelism by using GPU on its each node.

REFERENCES

[1] Droppo, J.—Acero, A.: Context Dependent Phonetic String Edit Distance for Automatic Speech Recognition. 2010 IEEE International Conference on Acoustics, Speech, and Signal Processing, Dallas, Texas, USA, 2010, pp. 4358–4361, doi: 10.1109/ICASSP.2010.5495652.

[2] Li, H.—Homer, N.: A Survey of Sequence Alignment Algorithms for Next-Generation Sequencing. Briefings in Bioinformatics, Vol. 11, 2010, No. 5, pp. 473–483, doi: 10.1093/bib/bbq015.

[3] Varol, C.—Abdulhadi, H. M. T.: Comparision of String Matching Algorithms on Spam Email Detection. 2018 International Congress on Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT), IEEE, Ankara, Turkey, 2018, pp. 6–11, doi: 10.1109/IBIGDELFT.2018.8625317.

[4] Ying, Z.—Robertazzi, T. G.: Signature Searching in a Networked Collection of Files. IEEE Transactions on Parallel and Distributed Systems, Vol. 25, 2014, No. 5, pp. 1339–1348, doi: 10.1109/TPDS.2013.258.

[5] Yu, M.—Li, G.—Deng, D.—Feng, J.: String Similarity Search and Join: A Survey. Frontiers of Computer Science, Vol. 10, 2016, No. 3, pp. 399–417, doi: 10.1007/s11704-015-5900-5.

[6] Navarro, G.: A Guided Tour to Approximate String Matching. ACM Computing Surveys (CSUR), Vol. 33, 2001, No. 1, pp. 31–88, doi: 10.1145/375360.375365.

[7] Wagner, R. A.—Fischer, M. J.: The String-to-String Correction Problem. Journal of the ACM (JACM), Vol. 21, 1974, No. 1, pp. 168–173, doi: 10.1145/321796.321811.

[8] Masek, W. J.—Paterson, M. S.: A Faster Algorithm Computing String Edit Distances. Journal of Computer and System Sciences, Vol. 20, 1980, No. 1, pp. 18–31, doi: 10.1016/0022-0000(80)90002-1.

[9] Mathies, T. R.: A Fast Parallel Algorithm to Determine Edit Distance. Carnegie Mellon University. Journal contribution, 1988, doi: 10.1184/R1/6587387.v1.

[10] Hyyrö, H.: A Bit-Vector Algorithm for Computing Levenshtein and Damerau Edit Distances. Nordic Journal of Computing, Vol. 10, 2003, No. 1, pp. 29–39.

[11] Gropp, W.—Hoefler, T.—Thakur, R.—Lusk, E.: Using Advanced MPI: Modern Features of the Message-Passing Interface. MIT Press, 2014.

[12] Grama, A.—Kumar, V.—Gupta, A.—Karypis, G.: Introduction to Parallel Computing. Addison Wesley, 2003.

[13] Apostolico, A.—Atallah, M. J.—Larmore, L. L.—McFaddin, S.: Efficient Parallel Algorithms for String Editing and Related Problems. SIAM Journal on Computing, Vol. 19, 1990, No. 5, pp. 968–988, doi: 10.1137/0219066.

[14] Huang, X.: A Space-Efficient Parallel Sequence Comparison Algorithm for a Message-Passing Multiprocessor. International Journal of Parallel Programming, Vol. 18, 1989, No. 3, pp. 223–239, doi: 10.1007/BF01407900.

[15] Sellers, P. H.: The Theory and Computation of Evolutionary Distances: Pattern Recognition. Journal of Algorithms, Vol. 1, 1980, No. 4, pp. 359–373, doi: 10.1016/0196-6774(80)90016-4.

[16] Myers, G.: A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. Journal of the ACM (JACM), Vol. 46, 1999, No. 3, pp. 395–415, doi: 10.1145/316542.316550.

[17] Chacón, A.—Marco-Sola, S.—Espinosa, A.—Ribeca, P.—Moure, J. C.: Thread-Cooperative, Bit-Parallel Computation of Levenshtein Distance on GPU. Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14), ACM, Munich, Germany, 2014, pp. 103–112, doi: 10.1145/2597652.2597677.

[18] Šošić, M.—Šikić, M.: Edlib: A C/C++ Library for Fast, Exact Sequence Alignment Using Edit Distance. Bioinformatics, Vol. 33, 2017, No. 9, pp. 1394–1395, doi: 10.1093/bioinformatics/btw753.

[19] Balhaf, K.—Shehab, M. A.—Al-Sarayrah, W. T.—Al-Ayyoub, M.—Al-Saleh, M.—Jararweh, Y.: Using GPUs to Speed-Up Levenshtein Edit Distance Computation. 7th International Conference on Information and Communication Systems (ICICS), IEEE, Irbid, Jordan, 2016, pp. 80–84, doi: 10.1109/IACS.2016.7476090.

[20] Yang, J.—Xu, Y.—Shang, Y.: An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs. Proceedings of the World Congress on Engineering (WCE 2010), London, U.K., 2010, Vol. 1, pp. 499–504.

[21] Sadiq, M. U.—Yousaf, M. M.—Aslam, L.—Aleem, M.—Sarwar, S.—Jaffry, S. W.: NvPD: Novel Parallel Edit Distance Algorithm, Correctness, and Performance Evaluation. Cluster Computing, Vol. 23, 2020, pp. 879–894, doi: 10.1007/s10586-019-02962-w.

[22] Ho, T.—Oh, S.—Kim, H.: A Parallel Approximate String Matching Under Levenshtein Distance on Graphics Processing Units Using Warp-Shuffle Operations. PloS ONE, Vol. 12, 2017, No. 10, Art. No. e0186251, doi: 10.1371/journal.pone.0186251.

[23] Guo, L.—Du, S.—Ren, M.—Liu, Y.—Li, J.—He, J.—Tian, N.—Li, K.: Parallel Algorithm for Approximate String Matching with K Differences. 2013 IEEE Eighth International Conference on Networking, Architecture and Storage, Xi'an, China, 2013, pp. 257–261, doi: 10.1109/NAS.2013.40.

[24] Aluru, S.—Futamura, N.—Mehrotra, K.: Parallel Biological Sequence Comparison Using Prefix Computations. Journal of Parallel and Distributed Computing, Vol. 63, 2003, No. 3, pp. 264–272, doi: 10.1016/S0743-7315(03)00010-8.

[25] Sanders, P.—Träff, J. L.: Parallel Prefix (Scan) Algorithms for MPI. In: Mohr, B., Träff, J. L., Worringen, J., Dongarra, J. (Eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4192, 2006, pp. 49–57, doi: 10.1007/11846802_15.

[26] Rajko, S.—Aluru, S.: Space and Time Optimal Parallel Sequence Alignments. IEEE Transactions on Parallel and Distributed Systems, Vol. 15, 2004, No. 12, pp. 1070–1081, doi: 10.1109/TPDS.2004.86.

[27] Liu, W.—Schmidt, B.—Voss, G.—Muller-Wittig, W.: Streaming Algorithms for Biological Sequence Alignment on GPUs. IEEE Transactions on Parallel and Distributed Systems, Vol. 18, 2007, No. 9, pp. 1270–1281, doi: 10.1109/TPDS.2007.1069.

[28] Boukerche, A.—de Melo, A. C. M. A.—de Oliveira Sandes, E. F.—Ayala-Rincon, M.: An Exact Parallel Algorithm to Compare Very Long Biological Sequences in Clusters of Workstations. Cluster Computing, Vol. 10, 2007, No. 2, pp. 187–202, doi: 10.1007/s10586-007-0020-0.

[29] Batista, R. B.—Boukerche, A.—de Melo, A. C. M. A.: A Parallel Strategy for Biological Sequence Alignment in Restricted Memory Space. Journal of Parallel and Distributed Computing, Vol. 68, 2008, No. 4, pp. 548–561, doi: 10.1016/j.jpdc.2007.08.007.

[30] Lopes, H. S.—Lima, C. R. E.—Moritz, G. L.: A Parallel Algorithm for Large-Scale Multiple Sequence Alignment. Computing and Informatics, Vol. 29, 2012, No. 6+, pp. 1233–1250.

[31] Yousaf, M. M.—Sadiq,M. U.—Aslam, L.—Ul Qounain, W.—Sarwar, S.: A Novel Parallel Algorithm for Edit Distance Computation. Mehran University Research Journal of Engineering and Technology, Vol. 37, 2018, No. 1, pp. 223–232, doi: 10.22581/muet1982.1801.20.

[32] National Center for Biotechnology Information (NCBI), `https://www.ncbi.nlm.nih.gov/`.

Muhammad Umair Sadiq received his B.Sc. and M.Sc. degrees in computer science from PUCIT, University of the Punjab, Lahore, Pakistan in 2016 and 2018, respectively. His research interests include parallel and distributed computing, multi-core computing, performance analysis, and GPGPU computing.

Muhammad Murtaza Yousaf is Professor at PUCIT, University of the Punjab, Lahore, Pakistan. He obtained his Ph.D. from University of Innsbruck, Austria in 2008. He worked on networks for grid computing during his Ph.D. His current areas of research include transport layer of networks, parallel and distributed computing, cloud computing, data science, and interdisciplinary research.