A Novel Parallel Algorithm for Edit Distance Computation

MUHAMMAD MURTAZA YOUSAF*, MUHAMMAD UMAIR SADIQ*, LAEEQ ASLAM*, WAQAR UL QOUNAIN*, AND SHAHZAD SARWAR*

RECEIVED ON 14.06.2017 ACCEPTED ON 21.08.2017

ABSTRACT

The edit distance between two sequences is the minimum number of weighted transformation-operations that are required to transform one string into the other. The weighted transformation-operations are insert, remove, and substitute. Dynamic programming solution to find edit distance exists but it becomes computationally intensive when the lengths of strings become very large. This work presents a novel parallel algorithm to solve edit distance problem of string matching. The algorithm is based on resolving dependencies in the dynamic programming solution of the problem and it is able to compute each row of edit distance table in parallel. In this way, it becomes possible to compute the complete table in min(m,n) iterations for strings of size m and n whereas state-of-the-art parallel algorithm solves the problem in max(m,n) iterations. The proposed algorithm also increases the amount of parallelism in each of its iteration. The algorithm is also capable of exploiting spatial locality while its implementation. Additionally, the algorithm works in a load balanced way that further improves its performance. The algorithm is implemented for multicore systems having shared memory. Implementation of the algorithm in OpenMP shows linear speedup and better execution time as compared to state-of-the-art parallel approach. Efficiency of the algorithm is also proven better in comparison to its competitor.

Key Words: Edit Distance, Levenshtein Distance, OpenMP, Speedup.

1. INTRODUCTION

Comparison of two strings helps in solving problems from many domains including bioinformatics (DNA analysis) [1], text-processing (spell-checkers, plagiarism detection, and error correction), signal processing, information retrieval, speech recognition, and web mining. String matching or string comparison comes into different forms: finding if a string is substring of another string, identifying the longest common subsequence, and checking how similar or dissimilar two strings are [2]. All these forms of string matching have their own applications in different areas.

This work will be focusing on the problem of checking how similar two strings are, in other words, how closely two strings resemble. In this regard, a well-defined measure exists, called Levenshtein distance. In simple words, Levenshtein distance is the number of transformation-operations (deletion, insertion, or substitution) required to transform one string to another. Sometimes, Levenshtein distance is also referred as edit distance between two strings. Edit distance find its applications in natural language processing where spell correction is most common use of it and in computational biology it is used

Corresponding Author (E-Mail:murtaza@pucit.edu.pk)
* Punjab University College of Information Technology, University of the Punjab, Lahore.

for matching and aligning DNA sequences. It is also used for machine translation, information extraction and speech recognition.

Dynamic programming solutions exist to find edit distance but it becomes computationally intensive when the lengths of strings become very large. Hence, a parallel algorithm can always help in finding the solution in reasonable time. This study presents a novel parallel algorithm to compute edit distance. The theoretical design is thoroughly evaluated and compared with state-of-the-art parallel approach. Further, the algorithm is implemented in OpenMP for multicore systems that showed improved results.

Rest of the paper is organized as follows: Section 2 explains the Levenshtein distance, Section 3 discusses parallel approaches to compute Levenshtein distance, hence cover the related work, Section 4 presents the justifications for our novel parallel approach to compute Levenshtein distance and theoretically compares our approach with state-of-the-art, and Section 5 discusses the implementation and experimental results. Finally, Section 6 concludes the work highlighting future directions.

2. LEVENSHTEIN DISTANCE (EDIT DISTANCE)

This section defines theLevenshtein distance or simply edit distance and explain by using a simple example, its mathematical formulation, and its dynamic programming solution.

2.1 Definition

Given two strings/sequences $A = \langle a_1, a_2,\ldots,a_m\rangle$ and $B = \langle b_1, b_2,\ldots,b_n\rangle$of size m and n respectively, over a finite $X = \langle X_1, X_2,\ldots,X_k\rangle$, the edit distance between A and B, represented by $ED_{AB}$is the minimum number of weighted transformation-operations that are required to transform A into B. This work assumes that the weighted transformation-operations are insert, remove, and substitute and weight of each operation is 1.

If A =⟨Thursday⟩ and B = ⟨Tuesday⟩then $ED_{AB} = 0$ because no transformation-operation is required because both the strings are identical.

If A =⟨Thursday⟩ and B = ⟨Tuesday⟩then $ED_{AB} = 2$ because one remove (remove 'h') and one substitution (replace 'r' with 'e') is required to transform A into B.

2.2 Mathematical Formulation

Given two strings/sequences $A = \langle a_1,a_2,\ldots,a_m\rangle$ and $B = \langle b_1,b_2,\ldots,b_n\rangle$ of size m and n respectively, over a finite alphabet $X=\langle x_1,x_2,\ldots x_k\rangle$, the edit distance between A and B, represented by ED_{AB} is defined by the recurrence in Equation (1).$ED_{AB}(i,j)$ is the distance between the first i characters of string A and the first j characters of string B.

$$ED_{AB}(i,j) = \begin{cases} j & \text{If } i = 0 \\ i & \text{If } j = 0 \\ ED_{AB}(i-1,j-1) & \text{If } A(i-1) = B(j-1) \\ \min\begin{cases} ED_{AB}(i,j-1)+1 \\ ED_{AB}(i-1,j)+1 \\ ED_{AB}(i-1,j)+1 \end{cases} & \text{Otherwise} \end{cases} \quad (1)$$

Where $(1 \le I \le m \text{ and } 1 \le j \le n)$

2.3 Dynamic Programming Solution

Given two strings of length m and n, a distance table D of size (m+1, n+1) is built in which D[i,j] is the distance between the first i characters of first string and the first j characters of second string. D[m, n] would be edit distance between both strings. This table can be filled in row-major order, i.e. row-by-row from top to bottom, and left to right within each row or in column major order i.e. column by column from left to right and top to bottom within each column. Sequential calculation of the table takes O(mn) time where m and n are lengths of strings. A

sample distance table for strings A = ⟨OMONA⟩ and B = ⟨MOON⟩ is presented in Fig. 1. Where $ED_{AB} = 3$ because two substitutions (replace 'O' with 'M' and replace 'M' with 'O') and one remove (remove 'A') is required to transform A into B.

2.4 Parallel Algorithm Formulation

This section discusses that how the distance table can be built in parallel. In order to do that, it is important to understand how the entries of the table are populated. As mentioned in the previous section, in case of sequential computation, the table can be filled in row-major or column-major order. But, to compute more than one entry simultaneously, some dependency analysis is required.

2.5 Computational Dependency Analysis

It can be observed from Equation (1) that each value of the distance table is computed based on three other values of the distance table which are in previous row and column as depicted in Fig. 2. According to this dependency neither a row nor a column can be calculated in parallel because computation of every entry in a row is dependent on the previous entry in same row and same is true in case of a column.

3. RELATED WORK

To solve edit distance problem in parallel major solutions are based on bit parallel [3] and diagonal parallelism approach. Bit parallel algorithms depend upon machine word size but as machine word size increases their performance decreases hence these are not applicable to general processors [4].

		O	M	O	N	A
	0	1	2	3	4	5
M	1	1	1	2	3	4
O	2	1	2	1	2	3
O	3	2	2	2	2	3
N	4	3	3	3	2	3

FIG. 1. A SAMPLE EDIT DISTANCE TABLE

Parallel algorithms to compute edit distance that are based on diagonal approach, compute the distance table diagonally i.e. one diagonal at a time because from dependence analysis it can be observed that each diagonal is dependent only on previous diagonal as shown in Fig. 3. Recently, most of the parallel algorithms to compute edit distance are based on diagonal approach. With this approach, if there are two strings of size m and n, then at most min (m, n) cells of distance table can be computed in parallel as it would be size of largest diagonal. Further, when m and n are almost same, this largest amount of parallelism will be attained only few number of times. With varying amount of parallelism at each step, it is also hard to maintain load balancing in diagonal based approaches [5-7].

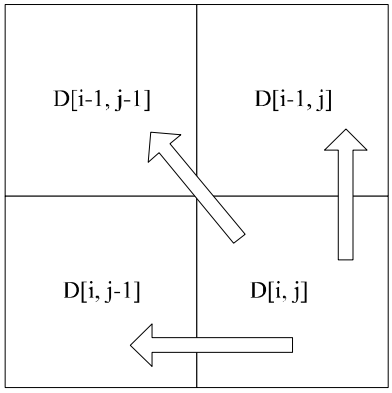


FIG. 2. COMPUTATIONAL DEPENDENCY OF D[I, J] IN EDIT DISTANCE TABLE

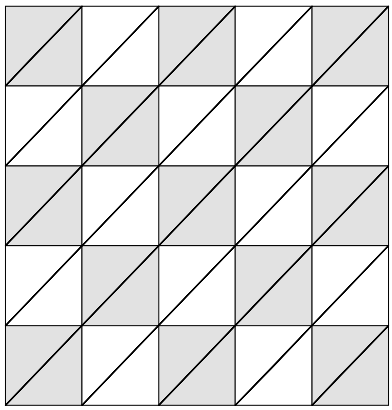


FIG. 3. DIAGONAL APPROACH

Parallel algorithm to solve edit distance problem used in [8] also uses diagonal based approach but it is specific to FPGA. Niewiarowski et. al. [9] used .NET Framework 4.0 technology with a specific implementation of threads using the System.Threading.Tasknamespace library and it requires specific number of threads to be executed in parallel. For different amount of threads it is not cost effective.

4. MAJOR CONTRIBUTION – NOVEL PARALLEL ALGORITHM

This section presents the novel parallel algorithm to compute edit distance.

Definition-1: if i^{th} character of first string matches with j^{th} character of second string then D[i,j] is called a **match case**.

Definition-2: if i^{th} character of first string does not match with j^{th} character of second string then D[i,j] is called a **non-match case**.

Considering a cell D[i,j-1] of edit distance table with the edit distance 'n', there are following observations:

Observation-0: With the assumption that the weight of each transformation-operation (insert, remove, and substitute) is 1, it is obvious based on recurrence of Equation (1) that the edit distance of two adjacent cells in a row or in a column will not differ by more than 1. Hence, D[i-1, j-1] may have 'n-1', 'n', or 'n+1'.

Observation-1: If the edit distance in the i^{th} row is increasing and the edit distance in previous row is also increasing. Possible cases, depicted in Fig. 4.

Observation-2: If the edit distance in the i^{th} row is increasing and the edit distance in previous row remains same. Possible cases, depicted in Fig. 5.

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i-1	n-1	n	i-1	n	n-1	i-1	n+1	n+2
i	n	n+1	i	n	n+1	i	n	n
Case 'a'	Not possible because D[i,j] cannot have a value greater than 'n', hence it violates the recurrence of Equation (1).							
Case 'b'	Only possible if D[i,j] is a non-match case, hence according to recurrence of Equation (1), D[i,j] would take min(n, n, n+1)+1.							
Case 'c'	Possible in following two situations:							
	If D[i,j] is a match case, hence according to recurrence of Equation (1), D[i,j] takes the value of top-left cell.							
	If D[i,j] is a non-match case, hence according to recurrence of Equation (1), D[i,j] would take min(n, n+1, n+2)+1.							

FIG. 4. CASES FOR OBSERVATION-1

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i - 1	n - 1	n - 1	i - 1	n	n	i - 1	n + 1	n + 1
i	n	n + 1	i	n	n + 1	i	n	n + 1
Case 'a'	Not possible because D[i, j] cannot have a value greater than 'n'.							
Case 'b'	Only possible if D[i, j] is a non-match case.							
Case 'c'	Possible in following two situations:							
	If D[i, j] is a non-match case.							
	If D[i, j] is a non-match case.							

FIG. 5. CASES FOR OBSERVATION-2

Observation-3: If the edit distance in the i^{th} row is increasing and the edit distance in previous row is decreasing. Possible cases, depicted in Fig. 6.

Observation-4: If the edit distance in the i^{th} row remains same and the edit distance in previous row is increasing. Possible cases, depicted in Fig. 7.

Observation-4(a): If the case 'a' of observation 4 continues for the next column, then it would definitely be a match case at D[i, j+1]. This self-explanatory situation is depicted in Fig. 8.

Observation-5: If the edit distance in the i^{th} row remains same and the edit distance in previous row also remains same. Possible cases, depicted in Fig. 9.

Observation-6: If the edit distance in the i^{th} row remains same and the edit distance in previous row is decreasing. Possible cases, depicted in Fig. 10.

Observation-7: If the edit distance in the i^{th} row is decreasing and the edit distance in previous row is increasing. Possible cases, depicted in Fig. 11.

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i - 1	n - 1	n - 2	i - 1	n	n - 1	i - 1	n + 1	n
i	n	n + 1	i	n	n + 1	i	n	n + 1
Case 'a'	Not possible because D[i,j] cannot have a value greater than 'n-1'.							
Case 'b'	Not possible because D[i,j] cannot have a value greater than 'n'.							
Case 'c'	Possible in following two situations:							
	If D[i,j] is a match case.							
	If D[i,j] is a non-match case.							

FIG. 6. CASES FOR OBSERVATION-3

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i - 1	n - 1	n	i - 1	n	n + 1	i - 1	n + 1	n + 2
i	n	n	i	n	n	i	n	n
Case 'a'	Possible if D[i,j] is a non-match case.							
Case 'b'	Possible if D[i,j] is a match case.							
Case 'c'	Not possible because D[i,j] cannot have a value less than 'n+1'.							

FIG. 7. CASES FOR OBSERVATION-4

	j - 1	j	j + 1
i - 1	n - 1	n	n + 1
i	n	n	N

FIG. 8. SITUATION FOR OBSERVATION-4(A)

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i - 1	n - 1	n - 1	i - 1	n	n	i - 1	n + 1	n + 1
i	n	n	i	n	n	i	n	N
Case 'a'	Possible if D[i,j] is a non-match case.							
Case 'b'	Possible if D[i,j] is a match case.							
Case 'c'	Not possible because D[i,j] cannot have a value less than 'n+1'.							

FIG. 9. CASES FOR OBSERVATION-5

Observation-8: If the edit distance in the i^{th} row is decreasing and the edit distance in previous row remains same. Possible cases, depicted in Fig. 12.

Observation-9: If the edit distance in the i^{th} row is decreasing and the edit distance in previous row is also decreasing. Possible cases, depicted in Fig. 13.

Theorem-1: For a non-match case D[i, j+k] (k> 0) with last match case D[i,j]

D[i, j+k] = min(D[i-1, j-1]+k, D[i-1, (j+k)-1]+1, D[i-1, j+k]+1)

Proof: As D[i, j] is a match case, D[i,j]= D[i-1,j-1].

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i - 1	n - 1	n - 2	i - 1	n	n - 1	i - 1	n + 1	n
i	n	n	i	n	n	i	n	N
Case 'a'	Not possible because D[i,j] cannot have a value greater than 'n-1'.							
Case 'b'	Possible if D[i,j] is a match case.							
Case 'c'	Possible if ifD[i,j] is a non-match case.							

FIG. 10. CASES FOR OBSERVATION-6

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i - 1	n - 1	n	i - 1	n	n + 1	i - 1	n + 1	n + 2
i	n	n - 1	i	n	n - 1	i	n	n - 1
Case 'a'	Possible if D[i,j] is a match case.							
Case 'b'	Not possible because D[i,j] cannot have a value less than 'n'.							
Case 'c'	Not possible because D[i,j] cannot have a value less than 'n+1'.							

FIG. 11. CASES FOR OBSERVATION-7

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i - 1	n - 1	n - 1	i - 1	n	n	i - 1	n + 1	n + 1
i	n	n - 1	i	n	n - 1	i	n	n - 1
Case 'a'	Possible if D[i,j] is a match case.							
Case 'b'	Not possible because D[i,j] cannot have a value less than 'n'.							
Case 'c'	Not possible because D[i,j] cannot have a value less than 'n+1'.							

FIG. 12. CASES FOR OBSERVATION-8

	Case (a)			Case (b)			Case (c)	
	j -1	j		j-1	j		j-1	j
i - 1	n - 1	n - 2	i - 1	n	n - 1	i - 1	n + 1	n
i	n	n - 1	i	n	n - 1	i	n	n - 1
Case 'a'	Possible if D[i,j] is a match case.							
Case 'b'	Not possible because D[i,j] cannot have a value less than 'n-1'.							
Case 'c'	Not possible because D[i,j] cannot have a value less than 'n+1'.							

FIG. 13. CASES FOR OBSERVATION-9

A non-match case D[i, j+k] (k > 0), with last match case D[i,j], may happen under Observations-1 (Case b and c), Observation-2 (Case b and c), Observation-3 (Case c), Observation-4 (Case a), Observation-5 (Case a), and/or Observation-6 (Case c).

Fact-1: Under Observations-1 (Case b and c) the edit distance in i^{th} row and in $(i-1)^{th}$ row are increasing from the last match case. In this case, D[i, j+k] would have increased 'k' times from D[i,j] or D[i-1, j-1]. Hence, for D[i, j+k], D[i-1, j-1]+k would serve as it will be minimum out of D[i-1, j-1]+k, D[i-1, (j+k)-1]+1, and D[i-1, j+k]+1.

Fact-2: Under Observation-2 (Case b and c), and Observation-3 (Case c) the edit distance in i^{th} row is increasing and in $(i-1)^{th}$ row is decreasing or stays same from the last match case. In this case, edit distance in $(i-1)^{th}$ row will be less than edit distance in i^{th} row. Hence, for D[i, j+k], D[i-1, (j+k)-1]+1 or D[i-1, j+k]+1 would serve.

Fact-3: Under Observations-4(a), it cannot continue.

Fact-4: Under Observation-5 (Case a) and Observation-6 (Case c), the edit distance in i^{th} row remains same and in $(i-1)^{th}$ row is decreasing or stays same from the last match case. In this case, edit distance in $(i-1)^{th}$ row will be less than edit distance in ith row. Hence, for D[i, j+k], D[i-1, (j+k)-1]+1 or D[i-1, j+k]+1 would serve.

Based on above facts, for any valid permutation of Observations-1 (Case b and c), Observation-2 (Case b and c), Observation-3 (Case c), Observation-4 (Case a), Observation-5 (Case a), and/or Observation-6 (Case c), D[i, j+k] = min(D[i-1, j-1]+k, D[i-1, (j+k)-1]+1, D[i-1, j+k]+1).

According to Theorem-1, the dependency for the computation of edit distance has been shifted to previous row only as depicted in Fig. 14. It makes it possible to compute a complete row of D in parallel having the information of last match case available.

For a single character in firs string, it is possible to calculate a row containing last matching index against all the characters in second string. Similarly, it is also possible to calculate such row for all unique characters in first string. Let us call it MI (match Index) table. Given 'u' unique characters in first string and 'n' characters in second string, $MI_{u,n}$ table can be built using the recurrence in Equation (2).

$$MI_{i,j} = \begin{cases} 0 & \text{If } j = 0 \\ j & \text{If match case} \\ MI_{i,j-1} & \text{Otherwise} \end{cases} \quad (2)$$

Here, it is important to note that the cost of computing MI table is significantly low as compared to the overall computation required to build an edit distance table.The MI table is computed only for unique characters in the smaller string and considering a string composed of conventional Latin letters, the number of unique characters can be at most twenty-six. Considering any other alphabet, this number would always remain a constant so the computation of MI table would be mostly a constant time operation. All the rows of MI table can also be computed in parallel because the computation of

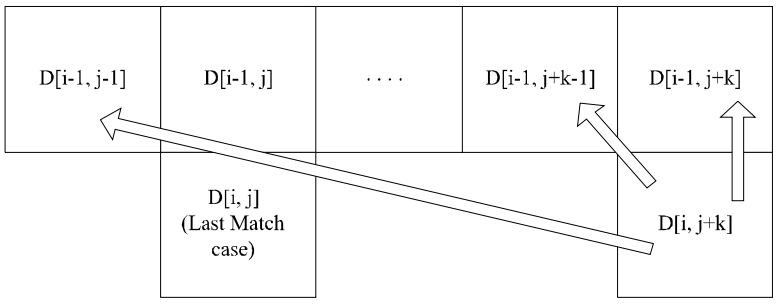


FIG. 14. SHIFT OF DEPENDENCY ON PREVIOUS ROW

its each row is independent. This fact further reduces the computation time of MI table and makes it a low cost operation.

Parallel Algorithm: Given two strings of length m and n, a distance table D of size (m+1, n+1) is built in following steps:

(1) Build $MI_{u,n}$ by computing each of its row in parallel.

(2) Build D row by row. Compute an individual row in parallel according to Theorem 1

Now it will be possible to simultaneously compute each cell in a row of table D.

Analysis: Given two strings of length 'm' and 'n', our new parallel algorithm is capable of computing a complete row of D in parallel so the computation of all the cells in a row can be distributed among processing nodes. In this way, the computation of D can be completed in min(m,n) iterations because it is always possible to take max(m,n) as the rows of the table. Whereas, state-of-the-art parallel algorithm that is based on diagonal approach has max(m,n) iterations. This phenomenon will be dominant when lengths of both the strings mismatch greatly, hence our algorithm will significantly perform better than diagonal based algorithm.

Another facet of the algorithm is its load balanced approach. Each iteration of the algorithm has same amount of computation so all the independent tasks in a single iteration can be distributed uniformly among the processing nodes. On the other hand, diagonal based parallel algorithm lacks this feature.

Our algorithm will also have an additive advantage in implementation. As the algorithm processes row by row, it can always exploit spatial locality in underlying memory system. The diagonal based algorithm requires to access cells from different rows and columns in its each iteration and that will always increase cache misses.

5. IMPLEMENTATION AND RESULTS

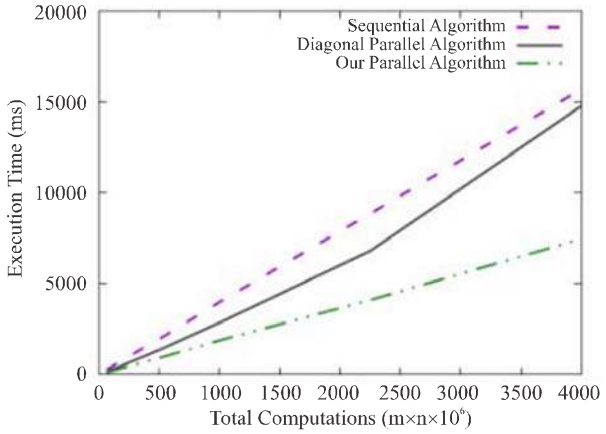
The algorithm is implemented for shared memory environment using OpenMP in conjunction with C++. OpenMP has emerged as a shared-memory standard and it is programming language tailored for a shared-memory multiprocessing so it is a natural fit compared to other API's.

The implementation is run on Intel Core-i3-2370M 2.40 GHZ having 2 cores and 4 logical processors and results are compared with sequential algorithm and diagonal parallel algorithm. The algorithm is utilizing CPU more than 90% so with increased computing power i.e. number of processors, this algorithm will perform even better. Strings are generated randomly of equal sizes and results are averages of fifteen experiments.

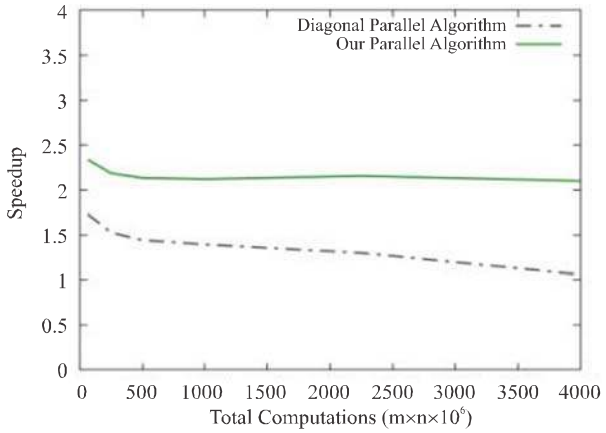
Results of execution time, speedup, and efficiency are presented in Fig. 15(a-c). Considering strings of size m and n, the problem size is defined in terms of m+n. For the first scenario, m and n are equal. Execution time is calculated in milliseconds. Speedup is the measure of increase of performance of parallel algorithm compared to sequential algorithm. Efficiency is a measure of the fraction of time for which a processing element is usefully employed. It is defined as the ratio of speedup to the number of processing elements. From the results, it is evident that our algorithm outperforms the state-of-the-art parallel approach to solve the edit distance problem. Particularly, it has achieved almost linear speedup that is result of load balanced feature of our parallel algorithm.

The experiments were also performed for another setting when the length of both the strings is not equal and resulting in a rectangular edit distance table. In this setting, the experiments were performed for different proportion of m and n assuming $\alpha m = n$. The á was varied from 2 to 9000. For increasing value of α, the performance of diagonal based approach becomes closer to the

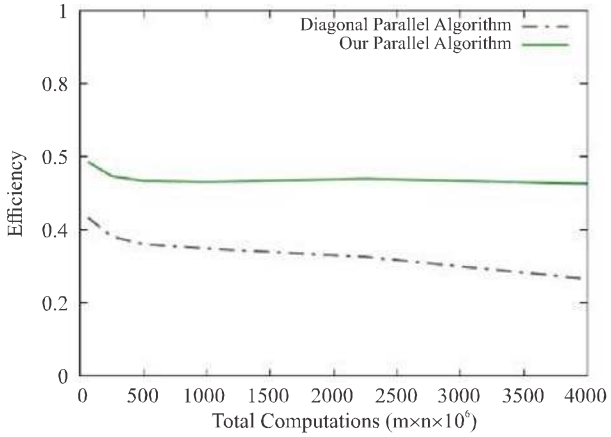
performance of sequential algorithm whereas, our algorithm shows consistent behavior for varying values of á. The results presented in Fig. 16(a-c) are for $\alpha = 5000$.



(a) EXECUTION TIME



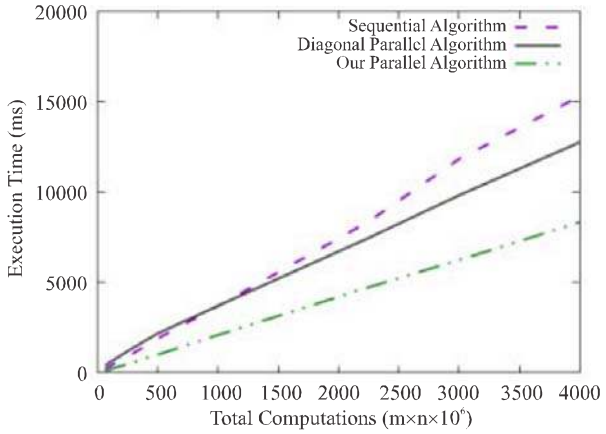
(b) SPEEDUP



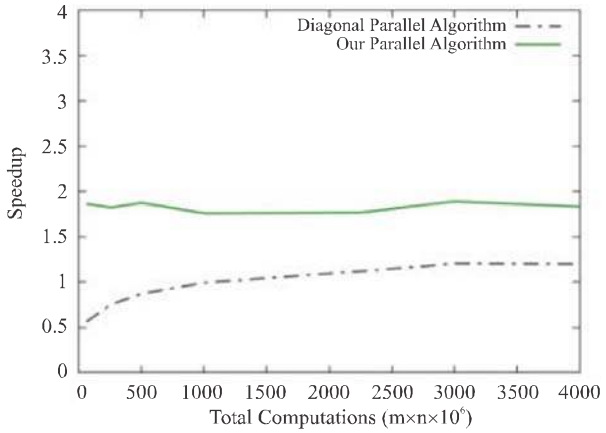
(c) EFFICIENCY

FIG. 15. RESULTS AND COMPARISON WHEN M AND N ARE EQUAL

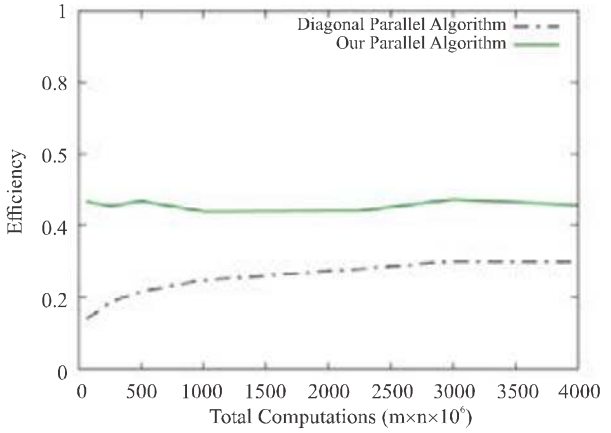
It is clear that our algorithm outperforms diagonal based approach in terms of execution time, speedup, and efficiency.



(a) EXECUTION TIME



(b) SPEEDUP



(c) EFFICIENCY

FIG. 16. RESULTS AND COMPARISON FOR A = 5000

6. CONCLUSION

Our novel parallel approach to compute edit distance is a load balanced approach that effectively utilizes the underlying computing resources. It exploits cache management of the architecture by working only in rows. It provides significant additive advantages when problem size is huge and particularly when the length of one string is quite large as compared to other. The results of implementation in OpenMP are promising and show improved performance as compared to state-of-the-art diagonal based approach. Further, we plan to implement and test our algorithm for varying parallel computing platforms. We also intend to use our algorithm to solve some real life problems that are based on edit distance.

ACKNOWLEDGEMENT

The authors are thankful to the Punjab University College of Information Technology, University of Punjab, Lahore, Pakistan, for providing necessary infrastructure to conduct this research.

REFERENCES

[1] Lan, H., Chan, H.,Xu, Y., Schmidt, K., Peng, B., and Liu, W., "Parallel Algorithms for Large-Scale Biological Sequence Alignment on Xeon-Phi Based Clusters", BMC Bioinformatics, Volume 17, No. 9, pp. 11-23, 2016.

[2] Qu, J., Zhang, G., Fang, Z., and Liu, J., "A Parallel Algorithm of String Matching Based on Message Passing Interface for Multicore Processors", International Journal of Hybrid Information Technology, Volume 9, No. 3, pp. 31-38, 2016.

[3] Mitani, Y., Ino, F., and Hagihara, K., "Parallelizing Exact and Approximate String Matching via Inclusive Scan on a GPU", IEEE Transactions on Parallel and Distributed Systems, Volume 28, No. 7, pp. 1989-2002, 2017.

[4] Yang, C., and Zhang, K., "Parallel Approaches to Edit Distance and Approximate String Matching", Carnegie Mellon University, 2014.

[5] Dhraief, A., Issaoui, R., and Belghith A., "Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability", 1st International Conference on Advanced Communications and Computation, 2011.

[6] Yang, J., Xu, Y., and Shang, Y., "An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs", Proceedings of World Congress on Engineering, London, UK, 2010.

[7] Kloetzli, J., Strege, B., Decker, J., and Olano, M., "Parallel Longest Common Subsequence using Graphics Hardware", Eurographics Symposium on Parallel Graphics and Visualization, 2008.

[8] Churchill, D., Gillard, P., Hamilton, M., and Wareham, T., "Prototyping Parallel Sequence Edit Distance Algorithms in FPGA Hardware", Proceedings of 14thAnnual New Found Land Electrical and Computer Engineering Conference, 2004.

[9] Niewiarowski, A., and Stanuszek, M., "Parallelization of the Levenshtein Distance Algorithm", Technical Transactions, Volume 3-NP, pp. 109-122, 2014.