information

Article

Toward Efficient Similarity Search under Edit Distance on Hybrid Architectures

Madiha Khalid [1,2,*], Muhammad Murtaza Yousaf [2] and Muhammad Umair Sadiq [1,3]

1 Department of Computer Science, University of the Punjab, Lahore 54000, Pakistan
2 Department of Software Engineering, University of the Punjab, Lahore 54000, Pakistan
3 Department of Computer Science, Government College University, Lahore 54000, Pakistan
* Correspondence: madiha.khalid@pucit.edu.pk

Abstract: Edit distance is the most widely used method to quantify similarity between two strings. We investigate the problem of similarity search under edit distance. Given a collection of sequences, the goal of similarity search under edit distance is to find sequences in the collection that are similar to a given query sequence where the similarity score is computed using edit distance. The canonical method of computing edit distance between two strings uses a dynamic programming-based approach that runs in quadratic time and space, which may not provide results in a reasonable amount of time for large sequences. It advocates for parallel algorithms to reduce the time taken by edit distance computation. To this end, we present scalable parallel algorithms to support efficient similarity search under edit distance. The efficiency and scalability of the proposed algorithms is demonstrated through an extensive set of experiments on real datasets. Moreover, to address the problem of uneven workload across different processing units, which is mainly caused due to the significant variance in the size of the sequences, different data distribution schemes are discussed and empirically analyzed. Experimental results have shown that the speedup achieved by the hybrid approach over inter-task and intra-task parallelism is 18 and 13, respectively.

Keywords: similarity search; sequence comparison; parallel edit distance; CUDA; inter-task parallelism; intra-task parallelism; hybrid parallelism

Citation: Khalid, M.; Yousaf, M.M.; Sadiq, M.U. Toward Efficient Similarity Search under Edit Distance on Hybrid Architectures. *Information* **2022**, *13*, 452. https://doi.org/10.3390/info13100452

Academic Editor: Chuan-Ming Liu

Received: 18 August 2022
Accepted: 20 September 2022
Published: 26 September 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In the modern high-performance computing landscape, a variety of approaches to parallel computing enables maximum performance gains for high-performance computing applications. High-performance computing applications can be found in nearly every field, ranging from core computer science to urban planning and health care. Particularly, as the technologies such as next-generation sequencing, artificial intelligence, and Internet of Things evolve, the size and amount of datasets are growing exponentially. One of the most fundamental activities common to many applications is the discovery of similar objects in the available data, which is commonly called a similarity search. A similarity search, as a primitive operation, has a broad spectrum of applications in many domains such as search engines, spam filters, data cleaning, plagiarism detection, data integration, biological sequence analysis, error checking, and pattern recognition. Particularly, it has gained the increasing attention of researchers after the emergence of the current information explosion in many fields of life sciences. The problem of string similarity search involves quantifying the similarity between strings and subsequently using this quantification to find all strings similar to a given query string. For example, in some applications, especially for mobiles, when users make typos, a string similarity search helps find suggestions of similar words from the dictionary.

To quantify the similarity between two strings, three types of similarity functions are used: token-based similarity functions, character-based similarity functions, and hybrid

functions. Among these functions, accessing similarity at the character level is the most widely used function [1]. Character-based similarity (commonly known as sequence-based measurement or edit distance) takes two character strings and quantifies the similarity between them by counting the single character elementary edit operations required to transform one string to the other. The most common method that uses this approach is Edit Distance or Levenshtein distance [2]. Another kindred approach is based on finding the lengths of the longest common subsequences of the two sequences [3]. Other common algorithms that are based on a similar approach are Damerau–Levenshtein [4], Jaro [5], JaroWinkler [6], Smith–Waterman [7], Needleman–Wunsch [8], Hirschberg's [9], and N-gram [10].

All these methods are built upon formal recursive definitions; therefore, the straightforward recursive method of measuring similarity between two strings using any of the mentioned algorithms takes exponential time. This exponential slowdown is impractical for strings larger than tens of characters. To improve the performance of recursive solutions, generally, a dynamic programming-based memoization technique is used that prevents the redundant computations of subproblems. It solves each subproblem exactly once and memorizes the solutions in a table, which can then be used to create an optimal solution for the larger problem. The general form of a dynamic programming-based solution to character-based similarity approaches can be represented in a general form, as shown in Figure 1. In this form, the algorithm processes a dynamic programming table T of size $n \times m$ according to a recurrence function f, where n and m are lengths of input strings A and B, respectively. The cell $T[i, j]$ is computed based on the output of the recurrence function $f(\ )$ that operates over inputs T, n, m, i, and j. The typical implementation of dynamic programming algorithms uses a loop-based implementation that iteratively populates the Dynamic Programming (DP) table. This approach also benefits from prefetching optimizations and has good spatial locality. However, these solutions require quadratic running time that scales poorly as the data size grows.

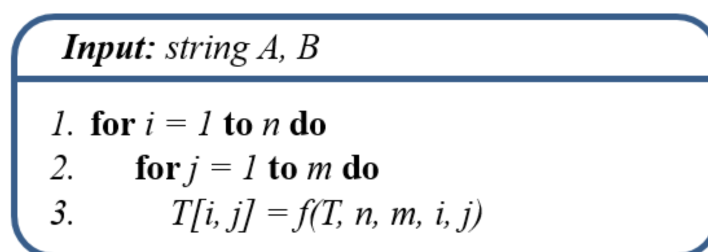


Figure 1. The general form of dynamic programming-based string similarity measures. source [11].

Levenshtein/Edit distance is one of the most popular and widely adopted algorithms used to compute optimal similarity scores [12]. The computation of edit distance between two strings is a quadratic time operation. Quite enough emphasis has been given to developing fast, scalable, and memory efficient techniques for addressing an edit distance-based similarity search [13–25]. This is a challenging problem because computing edit distance itself is a compute-intensive problem and for large collections of data, and calculating the edit distance of the query sequence with each sequence in the set makes it a more computationally intensive problem. Furthermore, the rate of growth of data in terms of the number of sequences and size of each sequence in the dataset slows down the process, which makes it inapt to apply on very large datasets. Most modern strategies either use some preprocessing schemes to index the data in the dataset [20–25], which allows for faster query evaluation, or use parallel computing to enhance the overall performance of the edit distance algorithm [13–19,26–28].

In recent years, the use of high-performance computing (HPC) to enhance the efficiency of computationally intensive problems is becoming popular. It has been demonstrated that the use of HPC technologies can significantly reduce the running time while ensuring accuracy [29]. In this paper, we investigate the problem of edit similarity search under

different parallel computing models. We introduce three scalable parallel computing approaches along with their analysis to support an efficient similarity search based on edit distance. These approaches include inter-task parallelism, intra-task parallelism, and a combination of both approaches. These approaches are proposed for a cluster of homogenous computing nodes. The proposed computing approaches are based on master–worker architecture where the master node acts as a dispatcher that distributes the workload among the worker nodes in a load-balanced manner and later aggregates the results. The worker nodes compute the similarity scores on the allocated data. The experimental evaluation proves that the hybrid approach which is based on a Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA—a parallel computing platform for general-purpose computing on GPUs) outperforms the other two approaches. Our hybrid approach achieves better runtime due to its intra-task parallelism and ability to efficiently utilize on-chip shared memory that significantly reduces the communication time. The hybrid approach achieves a speedup of 18 and 13 over inter-task and intra-task parallelism approaches, respectively. The experiments reveal that the hybrid approach is scalable and shows increased performance as the computational resources are increased. Moreover, it has been observed during the evaluation phase that the typical data distribution, i.e., a random but equal number of sequences assigned to each machine, results in an uneven amount of work across the cluster nodes due to the significant variance in the sizes of the sequences. To balance the workload across all the machines in the cluster, we used a simple yet effective technique that performs cyclic distribution after sorting the input sequences. Experimental results have shown that the speedup achieved by the hybrid approach using sorted sequences and cyclic distribution over random distribution is 1.59.

The rest of the paper is organized as follows: we review the related work to similarity search and previous parallel algorithms for edit distance computation in Section 2. In Section 3, we provide relevant background and essential concepts necessary to understand the proposed approaches. Section 4 introduces proposed parallel approaches and discusses their time and space requirements. Section 5 presents the experimental evaluation and performance comparisons. Finally, we conclude this work in Section 6.

2. Related Work

The problem of similarity search under edit distance has been extensively investigated in the literature for more than two decades. In the literature, many of the studies either use some preprocessing schemes to index the data in the dataset [20–23,30–38], which allows for faster query evaluation, or use parallel computing to enhance the overall performance of the edit distance algorithm. A similarity search generally corresponds to threshold-based similarity or top-k query [39]. Several existing studies for similarity search use either n-gram based indexes [10,27], trie-based indexes [20–23] or B+-trees [24,25].

The main idea behind the n-gram based inverted index is that if two strings are similar, they must share a specific number of tokens or n-grams, where n-grams are substrings of a string. To determine whether a string has a certain number of n-grams in common with the query, an inverted index is created in which the entries are n-grams and each entry keeps an inverted list that stores the strings containing the n-grams. This inverted index is used to find all strings that share a specific number of n-grams with the query string. This approach is efficient for threshold-based similarity with small threshold values; however, it suffers with larger threshold values. Additionally, due to the decomposition of strings into overlapping n-grams, it imposes a high space overhead. In most cases, the space overhead is more than five times the size of the original dataset. In order to reduce the size of inverted lists, Li et al. [30,35] proposed a method that uses variable-length n-grams. They worked on the concept that the fixed-length grams may not be efficient because some grams are frequent while others are infrequent. To address this issue, they selected high-quality grams to avoid generating very frequent grams. Kim et al. [31] and Behm et al. [32] propose several algorithms for compressing inverted lists while maintaining query performance. A couple of studies [33,34] proposed a prefix-based approach that is optimized for a specific

threshold. The authors in [36] focused on designing disk-based indexes for string similarity search by extending n-gram-based inverted indexes. Zhang et al. [38] proposed an efficient algorithm for a similarity search under an edit distance that uses a q-gram based index that partitions the string into a hierarchy of substrings.

The authors in [20–22] presented specialized algorithms based on tries. Strings are organized in a trie during an offline indexing phase. In the searching phase of query processing, they keep a record of the prefixes of input strings that are within the edit distance of the query string. The trie's active nodes or valid nodes are the ends of these prefixes. Deng et al. [23] extended the trie structure to support K-Nearest Neighbor (KNN) queries based on similarity search. However, these trie-based methods are limited to main memory. Although the performance of trie-based approaches is better than q-gram-based approaches [21], the main disadvantage of trie-based approaches is that the efficiency of these methods is dependant on the count of active nodes, which is generally very large: for example in [20,21], they are in the order of 10^5 and they grow exponentially with the alphabet size. This results in a slow query response even if the entire query only matches a few prefixes. To alleviate this problem, Qin et al. [37] proposed a method that maintains a small set of active nodes.

Another approach to similarity search employs B+-trees to index strings in order to respond to threshold-based similarity and KNN queries. Lu et al. [24] proposed a method that uses B+-tree to address edit distance-based similarity search. The algorithm first makes partitions of strings as per a set of reference strings, then in all partitions, strings are indexed using a B+-tree according to the distance of these strings with their respective string and finally, this B+-tree is used to answer string similarity-based queries. Another approach [25] proposed Bed-tree: a B+-tree-based index structure for addressing similarity search queries using edit distance and normalized edit distance. The authors identify the properties of a mapping from string space to integer space and propose three distinct transformations to capture various aspects of string information that allow efficient pruning during searching on the tree. However, the performance of this algorithm is limited to long strings. Several other studies in the literature are focused on using hashing, trees, signature-based and partition-based algorithms [17,40–45] to support edit distance-based similarity search.

As similarity search is a computationally intensive problem and the demands of applications include the processing of very large datasets that makes it even more time-consuming, therefore, many research studies are focused on achieving maximum performance gains using parallel computing. Jiang et al. [16] proposed a parallel partition-based framework for string similarity search under edit distance. The authors evaluate the algorithm with varying numbers of threads $(2-4)$ on a multicore machine with 16 processors of 2 GHz each. In [18], a parallel algorithm for approximate string matching with k-differences is proposed on GPUs. In the proposed algorithm, all threads within the same warp share data using a warp shuffle operation. The performance of the algorithm is further optimized by using GPU memory structures. Researchers in [13,14] also adopted warp shuffle operations to minimize the communication overhead among threads. Zhou et al. [17] presented GENIE, a framework of generic inverted index that attempts to reduce parallel programming complexity on GPUs. GENIE supports data types and similarity measures including edit distance and similarity measures that can be modeled in the match-count model.

In order to improve the performance of similarity search, some techniques employ the combined use of CPU and GPUs. Groth et al. [19] presented a parallel edit-distance based method for approximate similarity search using adaptive radix trees. The authors proposed several variants of the algorithm for CPUs and a GPU-based implementation to improve query throughput and accelerate the performance of the application. A research [26] modifies the traditional dynamic programming algorithm of edit distance to eradicate control flow divergence and reduce memory requirement. The algorithm divides the problem into independent quadrants and uses shared memory and GPU registers available to efficiently store data between different algorithm phases. Matsumoto and Yiu [46] proposed a CPU–GPU-based algorithm with compressed partial heap sort for similarity

search. A recent study [28] uses vector and line quantization methods for large-scale similarity search. A hierarchical index structure is proposed that is generated by vector and line quantization methods to improve accuracy and efficiency with the roughly equivalent amount of memory usage. Based on this index structure, a novel system is introduced called VLQ-ADC. The authors have evaluated VLQ-ADC on two billion-scale benchmark datasets SIFT1B and DEEP1B. A heterogeneous CPU-GPU computing system for measuring the similarity of RNA/DNA sequences was presented in [15]. The proposed system used a co-run computation model for maximum resource utilization where the workloads were assigned to and computed on both CPU and GPU devices at the same time. Workloads are distributed to CPU and GPU devices based on their computing capacity by employing a pre-computation mechanism. In computational biology and bioinformatics, the fundamental problem of database search can be recast as a similarity search problem where the similarity of a query sequence (a new DNA nucleotide or protein amino acid) is determined with a sequence database (known set of DNA or protein sequences). Several research efforts [47] are geared toward using multi-core CPUs, GPUs, or a combination of both to speed up the efficiency of the database search in biological databases.

Many attempts have been made to accelerate the performance of traditional edit distance computation. Edit distance-based similarity search uses the dynamic programming-based algorithm to compute the edit distance table. Each element of the edit distance table is dependent on the preceding elements in the same row or column. Therefore, designing a parallel algorithm for edit distance is not straightforward. In this regard, a common parallel approach is diagonal parallelization [13,14,47–50]. The idea behind diagonal parallelization is to process the edit distance table in diagonal stripes, which means all the entries within the same diagonal can be computed simultaneously. The edit distance table contains $m + n + 1$ diagonals where m and n are the lengths of strings to be compared and the count of cells in diagonals rises from 1 to $min(m, n)$ and then falls back to 1. Therefore, the resource utilization in this approach is limited by $O(min(m, n))$. In addition, the whole table is to be kept during the entire execution time, which makes its space complexity quadratic. Moreover, the varying number of cells in each diagonal would result in an unbalanced workload distribution. Lastly, because memory accesses for this approach do not coalesce, thus, it does not take advantage of spatial locality. Some research studies [51–55] use the bit parallelism approach, which is based on the number of bits that can be processed simultaneously. The number of bits that can be processed at a time typically depends on the word size of the underlying machine.

In contrast to the diagonal parallelization approach, the authors in [56–58] proposed an algorithm with a parallel scheme in which all elements in the same row of the edit distance table can be calculated in parallel by resolving dependencies. With this approach, we can process the maximum number of threads possible in parallel, which is up to the length of the input string. It is worth mentioning that the length of the input string is significantly greater than the query string. Thus, this approach makes maximum resource utilization, particularly, best suited for GPUs where there are many processors available. In this paper, we introduced several approaches to parallel edit distance-based similarity search using the row parallelization method introduced in [57,58].

3. Preliminaries

In this section, we provide relevant background and essential concepts necessary to understand the proposed parallel algorithms and their analysis presented in this paper.

3.1. Similarity Search under Edit Distance

Given a sequence $A = \langle a_1, a_2, ..., a_\ell \rangle$ of length ℓ, and a collection of sequences $S = \{s_1, s_2, ..., s_n\}$ having n sequences, where characters $a_i, s_{k,j} \in \Sigma$, a finite set of characters for $1 \leq i \leq \ell, 1 \leq j \leq \ell, 1 \leq k \leq n$. The similarity search is used to find the similarity of query sequence A with all the sequences in S such that every individual sequence s_k in S is compared with A to find the edit distance EDT between them.

Typically, a similarity search is defined using a user-specified threshold T, which determines whether the sequences are similar or not. The sequences are considered similar if the similarity score between them is less than the threshold value T.

3.2. Levenshtein Distance

The Levenshtein distance, also known as edit distance, is a string similarity measure that is defined as the minimum number of single-character edit operations required to transform one string into another. Here, an edit operation can be defined by either deleting, inserting, or replacing a single character. Formally, Lavenshtein distance is defined as: given two arbitrary strings, $A = \langle a_1, a_2, a_3, ..., a_n \rangle$ and $B = \langle b_1, b_2, b_3, ..., b_m \rangle$, where characters $a_i, b_j \in \Sigma$, a finite set of characters, for $1 \leq i \leq n$, $1 \leq j \leq m$. The Levenshtein distance between A and B denoted as $EDT_{A,B}$ is the minimum number of edit operations to make A and B identical, where $0 \leq EDT_{A,B} \leq max(|a|, |b|)$. The $EDT_{A,B}$ should satisfy the following three properties:

1. $EDT_{A,B} = 0 \quad iff \quad A = B$
2. $EDT_{A,B} > 0 \quad iff \quad A \neq B$
3. $EDT_{A,B} = EDT_{B,A}$

The smaller value of Levenshtein distance indicates more similarity between the two strings. If the Levenshtein distance between two string sequences is 0, then both the sequences will be considered identical, because no edit operations are needed to convert one sequence into another. Similarly, if A = *"march"* and B = *"cart"*, then the edit distance is 3 because two substitutions (replacement of *'m'* with *'c'* and *'c'* with *'t'*) and one deletion (deletion of *'h'*) are required to transform A into B. The higher the value of the edit distance, the more different the sequences are.

The most common way to compute the Levenshtein/edit distance between two sequences is a dynamic programming-based solution that uses the recurrence given in Equation (1). For two sequences A and B of lengths n and m, respectively, a dynamic programming table EDT of size $(n+1) \times (m+1)$ is computed that keeps track of the edit distance and edit operations between them. The last cell of the EDT table, i.e., $cell(n, m)$ represents the quantified dissimilarity between the two sequences. Each cell in the EDT table is computed using the recurrence presented in Equation (1).

$$EDT_{i,j} = \begin{cases} max(i,j) & if\ min(i,j) = 0 \\ EDT_{i-1,j-1} & if\ A_i = B_j \\ min \begin{cases} EDT_{i-1,j-1} + 1 \\ EDT_{i,j-1} + 1 \\ EDT_{i-1,j} + 1 \end{cases} & \text{otherwise} \end{cases} \quad (1)$$

Here, $0 \leq i \leq n$, $0 \leq j \leq m$, A_i is the ith character of sequence A, B_j is the jth character of sequence B, and $EDT_{i,j}$ is the edit distance between the initial i characters of A and initial j characters of B. The final value of the edit distance is at $EDT_{n,m}$. Figure 2 shows the edit distance table EDT between *"SPEED"* and *"SPACER"*. The edit distance is 3, which is stored at $EDT(6,5)$ and the edit operations that yield this edit distance can be found by tracing back the path from $EDT(6,5)$ to $EDT(0,0)$.

	“ ”	**S**	**P**	**E**	**E**	**D**
“ ”	0	←1	←2	←3	←4	←5
S	↑1	↖0	←1	←2	←3	←4
P	↑2	↑1	↖0	←1	←2	←3
A	↑3	↑2	↑1	↖1	↖2	↖3
C	↑4	↑3	↑2	↖2	↖2	↖3
E	↑5	↑4	↑3	↖2	↖2	↖3
R	↑6	↑5	↑4	↑3	↑3	↖3

Figure 2. An example of Levenshtein/edit distance table for input sequences *"SPACER"* and *"SPEED"*. The path that yields the edit distance is highlighted to trace back the required operations. Here, ← denotes "insertion" operation, ↑ denotes "deletion" operation, and ↖ denotes "substitution" operation if $A_i \neq B_j$.

The time complexity of the sequential algorithm is $O(n, m)$, where n and m are the lengths of input sequences. The space complexity is also $O(n, m)$ if the entire edit distance table is to be stored for a trace-back to find the required operations. However, if only the edit distance score is required, then only two rows (the current row and its preceding row) need to be allocated, which can be recycled for the entire computation. Therefore, the space complexity in this case is $O(min(n, m))$.

3.3. Computational Dependencies

Now that we have defined the fundamental recurrence relations in the problem, we can extend our discussion to analyze the dependencies in the solution. In the *EDT* table, the base case values (i.e., the values in the first row and first column) are trivial to compute; they are the lengths of substrings. The rest of the table is computed as per two observations:

1. If it is a match case $(A_i = B_j)$: The edit distance is the distance between two substrings that are one character shorter than the current substring, i.e., $EDT_{i,j} = EDT_{i-1,j-1}$.
2. If it is a non-match case $(A_i \neq B_j)$: The edit distance is one greater than the smallest edit distance of any of the three possible substring situations, i.e., $EDT_{i,j} = 1 + min(EDT_{i,j-1}, EDT_{i-1,j}, EDT_{i-1,j-1})$.

This clearly means that the computation of the value in each cell is dependent upon the three values in the same table that are in the preceding row and column. This dependency is illustrated in Figure 3.

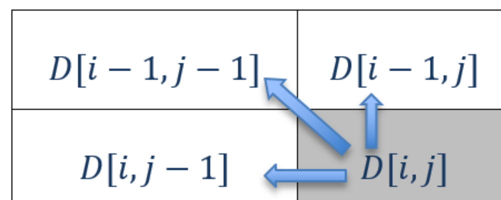


Figure 3. Computational dependency of a cell in edit distance table.

3.4. Redefining Computational Dependencies

According to Figure 3, in order to compute any cell in the edit distance table, three values must be needed: i.e., the value of the preceding cell, the value of the upper cell in the same column, and the value of the diagonal cell. Due to these dependencies, the calculation of each cell in a row depends on the cell in the same row but in the preceding

column, and two cells from the previous row. The same is true in the case of column-wise computation. Therefore, neither a row nor a column can be computed in parallel.

Yousaf et al. [58] established that the computational dependencies in the edit distance table can be redefined such that the computation of each cell can only be dependent on its preceding row. This means $EDT[i,j]$ is not dependent on $EDT[i,j-1]$. The authors proved that given the information about the last match case, all three values that are required to be known for the computation of a cell can come from its preceding row. Thereby, this made it possible to simultaneously compute a complete row of the edit distance table. According to Yousaf et al. [58], the new dependencies are shown in Figure 4.

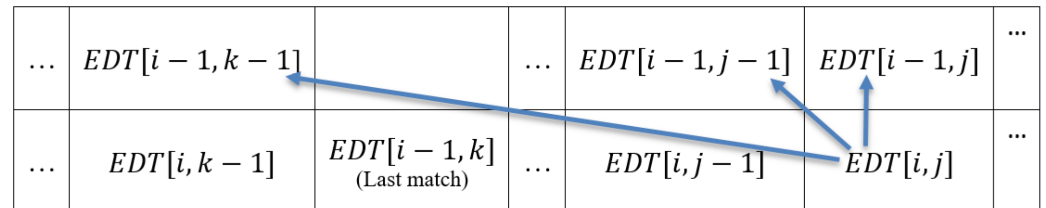


Figure 4. Redefining dependencies according to [58].

3.5. Communication Cost

The cost of communication becomes significant while designing a parallel algorithm. In a distributed computing environment, inter-processor communication happens when the communicating/coordinating tasks are located on different processors. The communication cost of an algorithm is the time taken by its tasks to send and receive messages. The communication time T_{comm} required to transmit a message from one node to another can be defined by two parameters: the message startup time t_s, which is the time required to prepare the message for transmission, and the per word transfer time t_ω, which is typically determined by the bandwidth of the communication channel that connects the two nodes. Thus, the cost of communication required to transfer η words is $T_{comm} = t_s + t_\omega \eta$.

3.6. Speedup

Speedup is defined as the measure of relative performance of two algorithms that process the same problem. The notion of speedup is more specifically established for parallel algorithms. In its classical form, it compares the performance of a sequential algorithm with its parallel algorithm.

4. Parallel Approaches for Similarity Search under Edit Distance

In this section, we present three approaches for parallel computation of similarity search using the edit distance algorithm, named inter-task parallelism, intra-task parallelism, and a hybrid approach that combines both approaches. In this paper, a task is defined as the computation of similarity score between two sequences. We define inter-task parallelism a parallel computing approach where n tasks are simultaneously running on available computing nodes. Intra-task parallel is defined as the parallel computing approach where each task, i.e., sequence comparison with the query string, is distributed over cluster nodes to be performed in parallel. In contrast, the hybrid approach is the combination of both models that distribute n sequences over the cluster nodes, and each task assigned to one computing node will run in parallel by several threads on the same machine. In the following subsections, these three approaches will be discussed in detail along with their complexity analysis.

4.1. Inter-Task Parallelism

In this approach, the n number of sequences in set S needs to be distributed over a cluster of p processing nodes. This approach is based on master–worker architecture where the master node acts as a dispatcher that distributes the input sequences among the worker

nodes and later aggregates the results. The worker nodes compute the similarity scores on the allocated sequences.

To compare the query sequence A with every sequence in the set S, S will be distributed among available processors. Since every sequence comparison is an independent set of operations, therefore, each sequence comparison can be performed in parallel. Assume that there are p processing nodes with identities ranging from 0 to $p-1$. To perform sequence comparisons in parallel, the master node will distribute set S among p worker nodes such that every node will receive an n/p distinct set of sequences from S. Every worker node will start computing similarity between the query string and one of the sequences allocated to it. Figure 5 illustrates this process in the form of a flowchart.

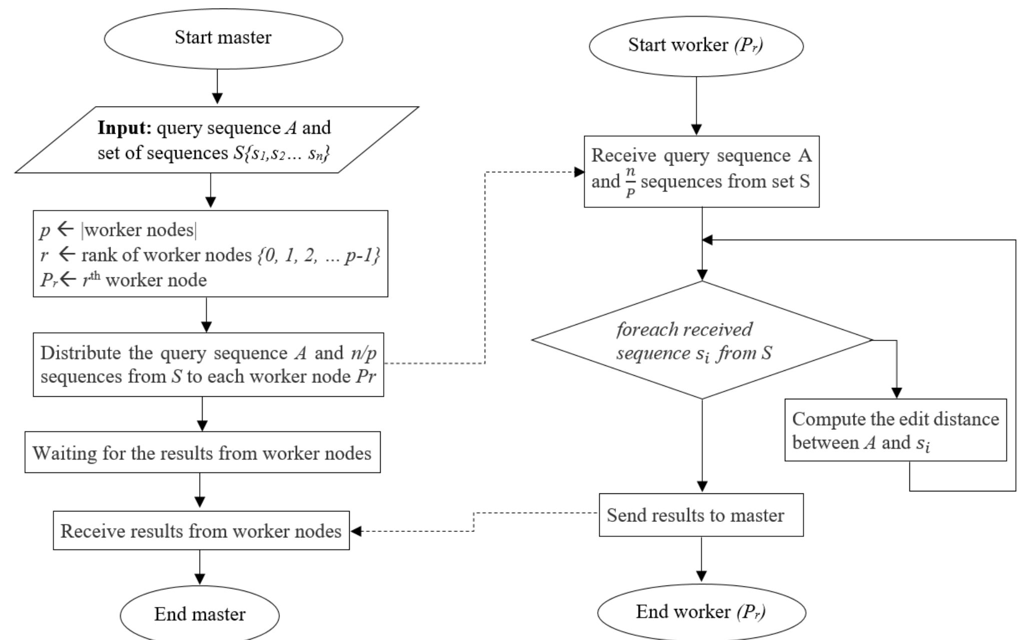


Figure 5. Flowchart of inter-task parallelism approach. The dotted lines show the communication between master and worker nodes.

Complexity Analysis: The computation of edit distance between two sequences, say A and B, using a traditional dynamic programming-based solution would require the computation of a dynamic programming table of size $(\ell+1) \times (m+1)$, where ℓ and m are the lengths of A and B, respectively. The time and space complexity of computing the edit distance between two sequences is $O(\ell m)$. Since one sequential comparison takes $O(\ell m)$ time and every processor performs n/p simultaneous comparisons, that makes the computation cost $O(\ell m) \times n/p$.

4.2. Intra-Task Parallelism Approach

In this approach, each task, i.e., each sequence comparison with the query string, is distributed over cluster nodes to be performed in parallel. While performing a one-to-one comparison of two sequences A and s1, both the sequences will be distributed among worker nodes. To perform a one-to-one comparison in parallel, the dependencies can be resolved as suggested by Sadiq et al. [56]. The authors proposed an algorithm that redefines the dependencies of the edit distance table such that all the cells in a row can be computed simultaneously. As discussed in Section 3.4, the computation of any cell in the EDT table can only be dependent on the values in its preceding row, thereby resolving intra-row dependencies. This enables the simultaneous computation of all the cells within a row. Since the size of each row in the edit distance table is the same, therefore, this approach makes a fair and balanced distribution of work among the available processing nodes. However, to make this method work, the algorithm precomputes a Last Match index Table

(LMT) that records the position of the last match of the unique characters in character set Σ when compared with the query sequence. This table can also be computed in parallel.

To compare two sequences using the method inspired by [56], both the sequences must be distributed over p processing nodes. Assume that there are p processing nodes with identities ranging from 0 to $p-1$. Since the computation of every row is dependent on its preceding row and the columns within a row can be computed in parallel, therefore, each row can be distributed among multiple processors such that every processor will obtain ℓ/p part of the row (as illustrated in Figure 6). To perform computation on its respective part of the row, every processor will receive the chunk of one sequence, i.e., sequence s1, starting from ℓ/p and ending at $(r+1)(\ell/p)$, where r is the identity of the processor. As soon as one sequence comparison is completed by all the processors, the next sequence from set S will be distributed among all the processors. This process will continue until all the comparisons will be completed.

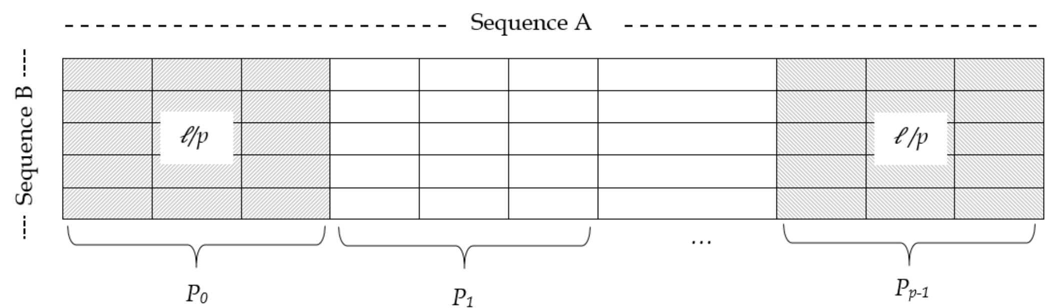


Figure 6. Mapping of a sequence having length ℓ to p processing nodes.

For a parallel comparison, each worker node will receive query sequence A and a part of sequence $s1$. First, each processor will compute the match index table for the part of the sequence that is assigned to that particular processor, and then, the EDT table is computed. At any time, all processors will be computing a part of one row. After the computation of every row, all the processors will become synchronized. Similarly, all the processors will compute the EDT table for their part (see Figure 7).

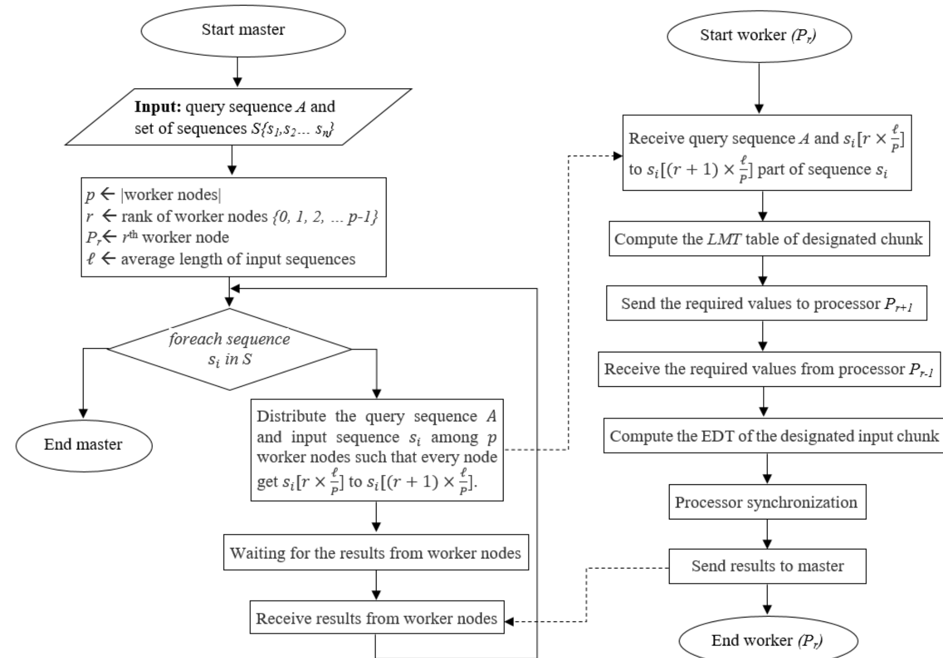


Figure 7. Flowchart of intra-task parallelism approach. The dotted lines show the communication between master and worker nodes.

Complexity Analysis: One parallel sequence comparison requires the computation of a match index table as well as an EDT table. For the computation of the match index table, every processor will be computing $(\ell|\Sigma|)/p$ values, whereas EDT table computation would require $(\ell m)/p$ simultaneous computations by each processor. Therefore, the total time to perform a parallel one-to-one comparison is $(\ell|\Sigma|)/p + (\ell m)/p$. Since these computations will be performed for every sequence in the set S and we have n number of sequences in S, the total time complexity will be $\{(\ell|\Sigma|)/p + \ell m/p\} \times n$.

As regards the space complexity, each row of the EDT table is dependent on its preceding row; therefore, each processor will be needing the space to store two rows at a time: the current row and the preceding row. Hence, each processor will allocate $2 \times \ell/p$ space for two rows. The match index table requires $\ell|\Sigma|/p$ space. The query sequence can be obtained by processors in arbitrary sized multiple chunks, but if the chunk size taken is less than ℓ/p, then the space complexity would be optimal. Hence, the total space complexity would be $(\ell/p) + (\ell|\Sigma|/p)$. Here, $|\Sigma|$ is constant. So, the overall space complexity is $O(\ell/p)$.

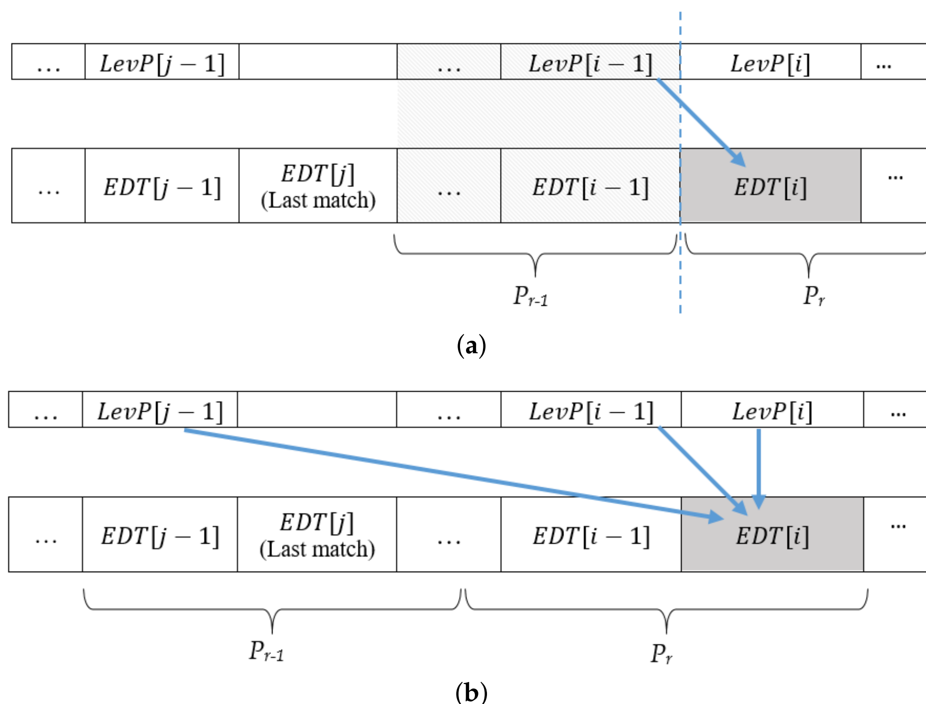
Since the computation of each row is distributed among several processors, it requires inter-process communication. The computation is of the form that the calculation of each cell in the EDT table would require a value from its preceding row, and that value can come from the part of the sequences which is assigned to another processor. This can happen in two cases:

1. To compute the first value (leftmost value), i.e., $EDT[i]$ of the edit distance table, each processor P_r needs the diagonal value of the previous row, say $LevP[i-1]$, which is not available locally and can be found from the preceding processor P_{r-1}.
2. There is a possibility that for some initial cells of a processor P_r, the value of the last match case resides in the part of data that is assigned to one of the preceding processors.

Figure 8 depicts both cases where the computation requires values from preceding processors. Both cases can be handled by communicating two values from the preceding processor. For the first case, where the diagonal value of the first cell of EDT is not available locally, every processor P_r communicates its rightmost value of the previous row to its following processor P_{r+1}. For the second case, where the last match case resides in the preceding processor, an exclusive scan operation (with the maximum as the binary associative operator) is performed with the value at the last match case of the processor (since there can be multiple match cases in each processor). Now, every processor has all the values to calculate its share in the ith row of the EDT table. Since for the computation of the EDT table, one exclusive scan operation per row is required, and for each row, one value (rightmost) should be communicated to the next processor, therefore, its communication time is $(\ell/p) \times (\log \ell + 1) \times n$. In contrast, communication involved in the computation of the LMT table is $n \times (\log \ell)|\Sigma|$ because to compute one row of the LMT, one exclusive scan operation is required. The total communication time for this approach is $(t_s + t_\omega \eta) \times \{(\log \ell)|\Sigma| + (\ell/p) \times (\log \ell + 1)\} \times n$.

4.3. Hybrid Approach

In this approach, we use a combination of the above two approaches using CPUs and GPUs to speed up the overall performance. The data distribution model is similar to inter-task parallelism. Every processor will receive n/p sequences from the set S. However, each sequence comparison will be performed in parallel using multiple GPUs. The comparison will be performed using the algorithm discussed in intra-task parallelism with the exception that several threads will be launched to perform a one-to-one comparison in parallel. Figure 9 illustrates the flow of the hybrid approach. This approach also takes advantage of shared memory to improve the overall performance of the algorithm, and the inter-processor communication overhead can be greatly reduced by using a combination of constant and shared memory.



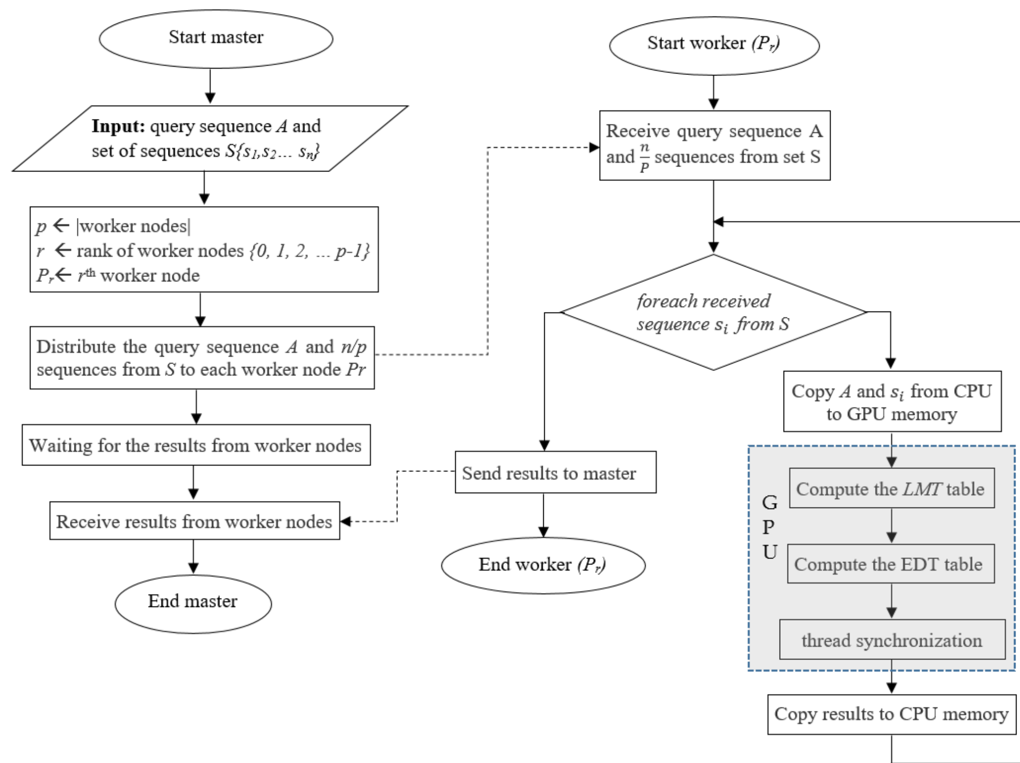


Figure 8. Two cases where the computation of a cell requires values from preceding processors. The computation of $EDT[i]$ is dependent upon the values $LevP[i], LevP[i-1]$ and $LevP[j-1]$. (**a**) Case 1: where the leftmost value $EDT[i]$ of processor P_r requires diagonal value $LevP[i-1]$ that comes from processor P_{r-1}. (**b**) Case 2: where last match value of processor P_r is found in its preceding processor. The last match case lies in that part of data that is assigned to the processor P_{r-1}.

Figure 9. Flowchart of the hybrid approach. The shaded area shows the part of computation performed in parallel by GPU threads. The dotted lines show the communication between master and worker nodes.

Complexity Analysis: This approach can perform n/p sequence comparisons simultaneously where each computation can be further parallelized by distributing it among t threads. Because each row of EDT can be computed in ℓ/p time and there are m of them, the time complexity of computing EDT becomes $O(\ell m)/t$. Furthermore, by taking the transpose of EDT, it is always possible to take $min(\ell, m)$ as rows. For the computation of the LMT, every processor will be computing $(\ell|\Sigma|)/t$ values. Since these computations will be performed for every sequence in the set S and we have n number of sequences in S that are divided to p processors, the total time complexity will be $\{(\ell|\Sigma|)/t + \ell m/t\} \times n/p$.

LMT requires $\ell|\Sigma|$ space. The edit distance table requires $O(\ell m)$ when all edit operations are required and $O(\ell)$ when only the score of edit distance is required, because only two adjacent rows must be stored when only the score of edit distance is required. Furthermore, if the size of the column is significantly greater than the size of the row, EDT can be computed column-wise using similar steps. It is worth noting that the computation of EDT is significantly larger when compared to the computation of LMT. The computation of LMT involves the unique characters in the query sequence. This computation can be improved by using a parallel prefix operation with max as the binary associative operator. Using parallel prefix operation can significantly improve the running time if the input sequence is very large. The computation itself does not require any communication because each comparison is parallelized by using GPUs threads and shared memory.

5. Experiments and Evaluation

In this section, we evaluate the proposed parallel approaches for similarity search.

Setup: For experimental evaluation, we implemented the algorithms using CUDA and MPI. For evaluation purposes, we used a cluster of five processing nodes where the minimum specification of a node is Intel Core-i5-3570K 3.40 GHz CPU having four physical cores, four logical processors, and 8 GB of main memory. Each node is equipped with an NVIDIA GeForce GTX 660 GPU with 960 compute cores, five streaming processors, and 2 GB of main memory. All nodes in the cluster are interconnected to a centralized hub by using Ethernet cables.

Dataset: We use two publicly available real datasets to perform the experiments: a dataset of genome sequences for which we extracted a hundred DNA sequences from NCBI [59] website in the range of 16 KB to 9 MB, and a dataset of GPS trajectories named GeoLife [60] by Microsoft. The genomics dataset has the sequence sizes in the range of 16 KB to 9 MB. To perform a similarity search, we used sequences gbpln103 having 16,871 base pairs (bp) and gbgss201 having 156,931 base pairs (bp) as query sequences. We extracted first twenty trajectories of the users from the GeoLife GPS dataset with sizes in the range of 510 KB to 18 MB. The trajectories are encoded as character sequences by taking GeoHash of the coordinates information of users. For that purpose, python-geohash [61] library is used. We used sequence 001 (1,728,404 characters long) and sequence 006 (505,199 characters long).

5.1. Evaluating Execution Time

A set of experiments was performed to evaluate the performance of all three approaches. Since the cluster in our testbed contains twenty processors, therefore, we launched an equivalent number of processes on five systems. For data distribution, we use the straightforward workload distribution, i.e., distribution of an equal amount of random n/p sequences to each processor. For hybrid parallelism, we used GPUs on each machine to perform the job. We launched one process per machine, where each process uses multi-threading on the GPUs to perform its job. The results are presented in Figures 10 and 11.

We plot the running time for varying lengths of query sequences for both datasets. Figure 10a–c show the running time on the GeoLife dataset with a long query sequence 001 (i.e., 1,728,404 characters), and Figure 10d–f show the running time with a short query sequence (i.e., 505,199 characters). Similarly, Figure 11a–c show the running time on the genome dataset with a short query sequence (i.e., 16,871 bp) and Figure 11d–f illustrate

the running time with a long query sequence (i.e., 156,931 bp) on the same dataset. It is worth noting that the comparison of longer query sequences took more time. The results clearly showed that the hybrid approach that combines inter-task and intra-task parallelism requires significantly less execution time than the other two approaches that are solely based on inter-task or intra-task parallelism.

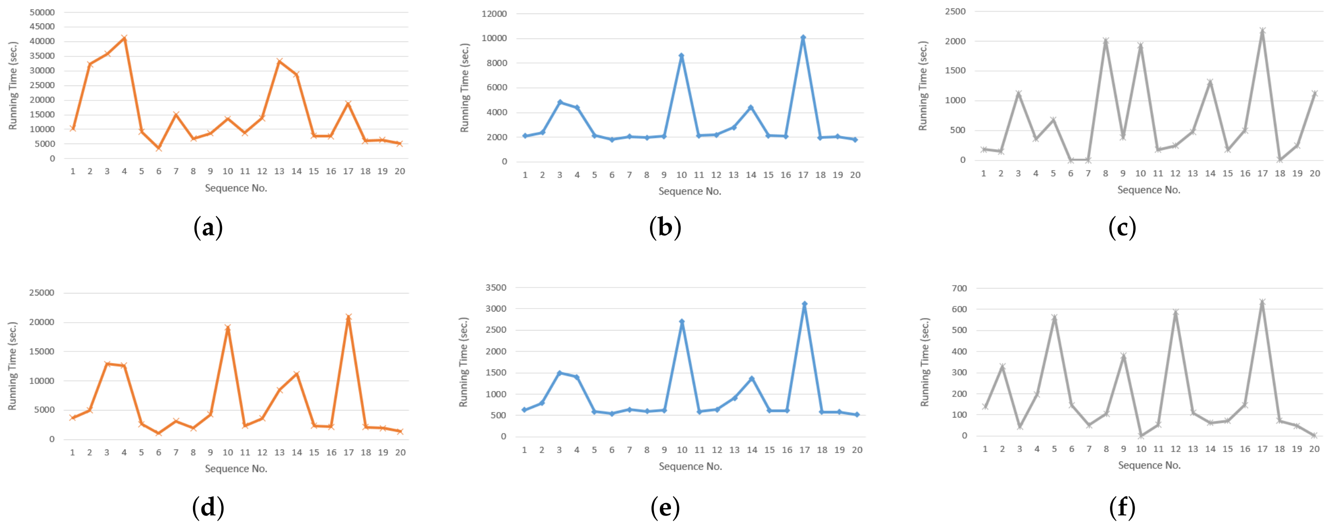


Figure 10. Execution time of proposed parallel approaches with GeoLife dataset and varying lengths of query strings. (**a**) Inter-Task parallelism with query sequence 001. (**b**) Intra-Task parallelism with query sequence 001. (**c**) Hybrid parallelism with query sequence 001. (**d**) Inter-Task parallelism with query sequence 006. (**e**) Intra-Task parallelism with query sequence 006. (**f**) Hybrid parallelism with query sequence 006.

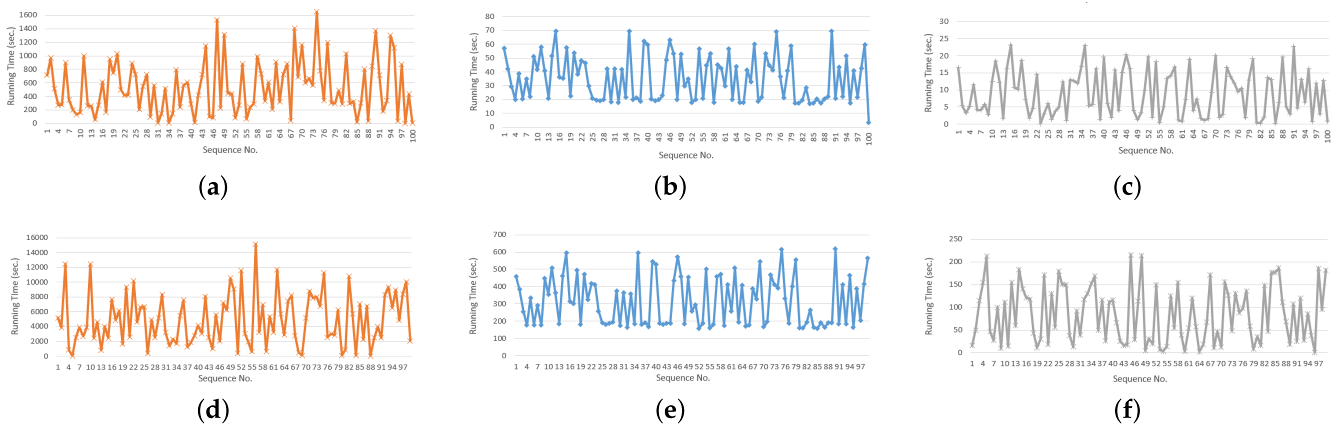


Figure 11. Execution time of proposed parallel approaches with Genome dataset and varying lengths of query strings. (**a**) Inter-Task parallelism with query sequence gbpln103. (**b**) Intra-Task parallelism with query sequence gbpln103. (**c**) Hybrid parallelism with query sequence gbpln103. (**d**) Inter-Task parallelism with query sequence gbgss201. (**e**) Intra-Task parallelism with query sequence gbgss201. (**f**) Hybrid parallelism with query sequence gbgss201.

For the intra-task parallelism, explicit communication is required among the processes. To analyze the effect of communication cost over total execution time, we computed the communication and computation time separately for that version (see Table 1). In other versions (the inter-task parallelism approach and hybrid approach), explicit communication is not needed. Table 1 clearly shows that more than 50% of the execution time is spent

in communication. Yet, this approach gives better results in comparison with inter-task parallelism despite spending more than half of the time in communication.

Table 1. Communication and computation time for intra-task parallelism approach.

Dataset	Query Sequence	Communication Time	Computation Time
Genome	gbgss201	17,539	13,082
	gbpln103	1771	1723
GeoLife	001	34,689	29,341
	006	10,729	8809

Figure 12 compares the performance of all three approaches (inter-task parallelism, intra-task parallelism, and hybrid approach) in a random data distribution setting. It is evident from the experimental results that the hybrid approach outperforms the other two approaches. The hybrid approach achieves better run time due to its intra-task parallelism and ability to efficiently utilize on-chip shared memory that significantly reduces the communication time. The hybrid approach achieves a speedup of 18 and 13 over inter-task and intra-task parallelism, respectively.

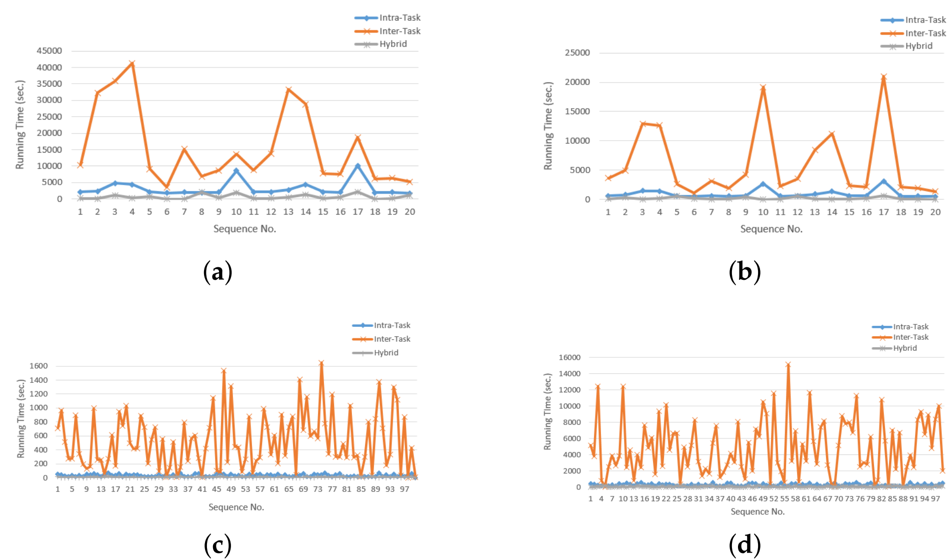


Figure 12. Comparison of execution time of proposed parallel approaches. (**a**) Comparison of execution time of all three parallel approaches on GeoLife dataset with query sequence 001. (**b**) Execution time of all three parallel approaches on GeoLife dataset with query sequence 006. (**c**) Execution time of all three parallel approaches on Genome dataset with query sequence gbpln103. (**d**) Execution time of all three parallel approaches on Genome dataset with query sequence gbgss201.

5.2. Evaluating Load Balancing

The results presented in Figure 10 are produced by using a random but equal amount of sequence distribution to each machine. If we observe the execution time on individual machines (Table 2), it can be seen that each machine obtains an uneven amount of work and their execution time varies significantly. Thus, the processors that received the larger-sized sequences finished later than those that received smaller-sized sequences. The total execution time is equivalent to the process that takes the largest amount of time because every process executes in parallel. The analysis of standard deviation (Table 2) reflects the dispersion in the workload: that is, 3637 and 673 in the case of inter-task parallelism and hybrid parallelism, respectively. The main cause of this uneven workload at different processing units is the significant variance in the size of the sequences.

Table 2. Execution time of individual machines for one-to-many comparison using inter-task parallelism and hybrid parallelism.

Machine Name	Execution Time (s)	
	Inter-Task Parallelism	**Hybrid Parallelism**
WS111	46,455.8	1704.76
WS112	35,823.9	2519
WS113	39,810.7	619.281
WS114	42,213.1	2081.14
WS115	43,971.6	2329.82
Total Execution time	46,456	2519
Standard Deviation	3637	673

To balance the workload across all the machines in the cluster, we used a simple yet effective technique that performs distribution after sorting the input sequences. To distribute the n/p input sequences among p processors, the following two-step procedure will be followed:

1. Sort all the sequences in ascending order according to their sizes.
2. From the sorted list of sequences, distribute n/p sequences to processors in a circular manner except that in every alternate step, the processor ordering will be reversed.

To understand the distribution mechanism, let us take a small example where there are 16 sequences in the set S and 4 processors P_0, P_1, P_2, and P_3. First, the sequences in S will be sorted according to their sizes. For simplicity, let us assume that the sequences are numbered from 1 to 16 according to their sizes, i.e., 1 is the smallest sized sequence and 16 is the longest sequence. Every processor will receive n/p sequences for the processing, which means in this case, every processor will receive 4 sequences. The sequences will be distributed to processors in a circular manner that means P_0 will receive sequence 1, P_1 will receive sequence 2, P_2 will receive sequence 3, P_3 will receive sequence 4, for the next 4 sequences the distribution order will be reversed i.e., P_3 will receive sequence 5, P_2 will receive sequence 6, and so on. This process is illustrated in Figure 13. Through this distribution method, every processor will receive a roughly equal share of the workload.

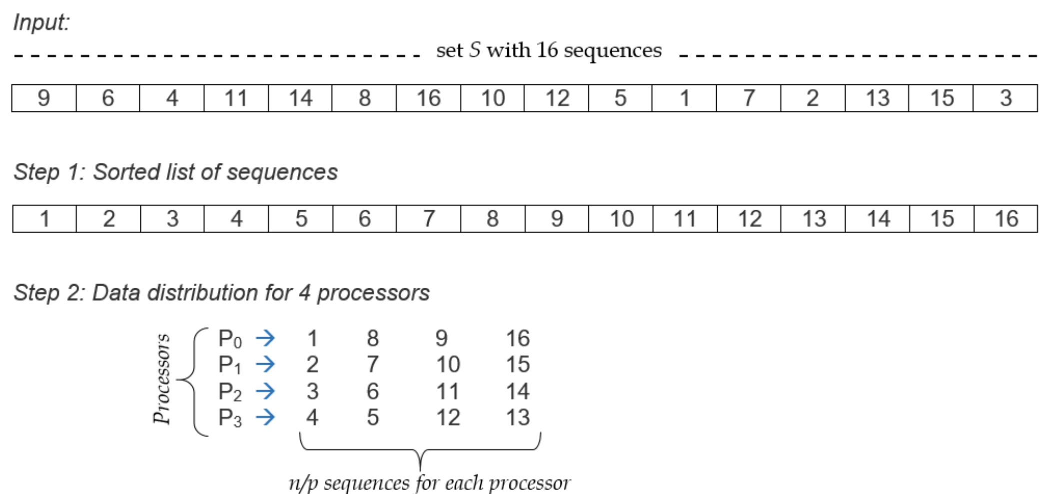


Figure 13. An example of data distribution strategy for 4 processors and 16 input sequences.

Since the experiments have shown that the hybrid approach outperforms the other two approaches in a random data distribution setting, we performed a set of experiments using the hybrid approach with two different data distribution schemes: (1) Random distribution of sequences with block division and (2) Sequence distribution after sorting them by their sizes and cyclic division among the processes.

Figure 14 plots the results of the hybrid approach using both the data distribution schemes. It can be seen that the results obtained by sorted and cyclic division among the processes give the peak performance. Moreover, the standard deviation of the execution time is also reduced to the lowest as the workload is balanced among the processes. The sorted and cyclic division achieved a speedup of 1.28 over random sequence distribution with block division on the Genome dataset and 1.59 over the GeoLife dataset.

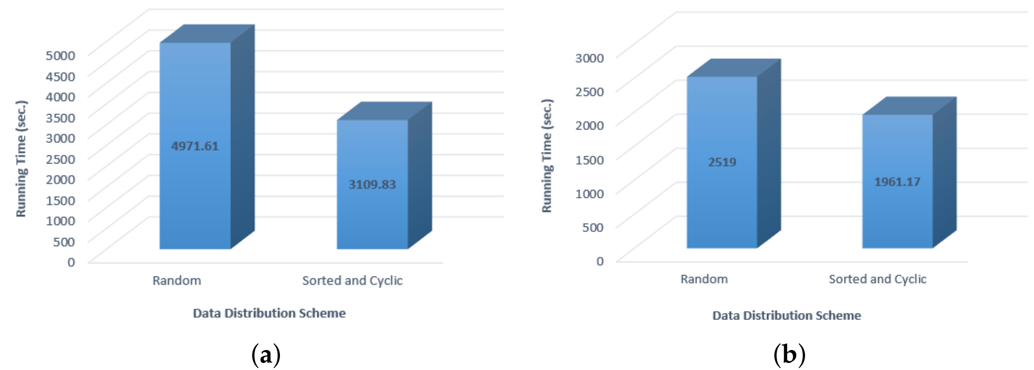


Figure 14. Comparison of load balancing techniques using (**a**) GeoLife dataset (**b**) Genome dataset.

5.3. Evaluating Scalability

We validated the scalability by varying the number of computing resources. We performed the similarity search using the hybrid approach and sorted and cyclic data distribution on a single machine. We then gradually increased the number of machines for the same query and dataset. When we increased the number of machines to two, the execution time becomes decreased by 57%. Similarly, when the number of machines jumps from two to three, the execution time was decreased by 61%. We increased the number of machines to five; Figure 15 shows the results of scalability. The general trend is that the algorithm significantly increases its performance as the computational resources become increased.

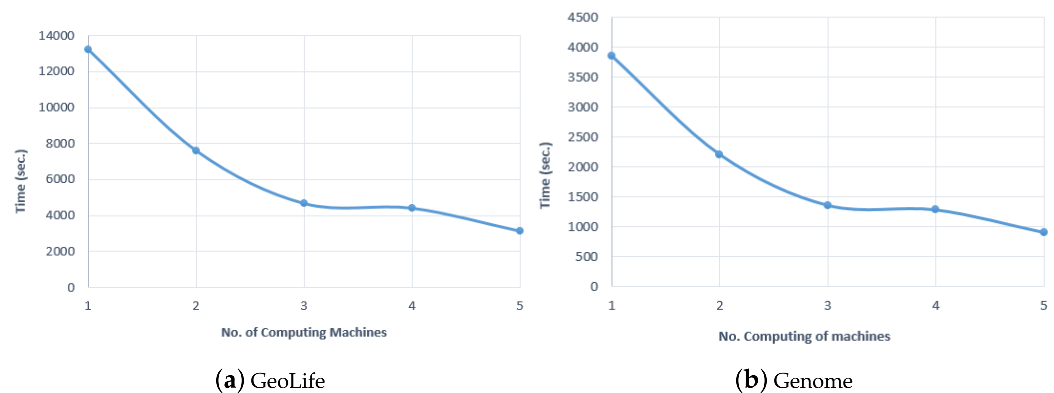


Figure 15. Evaluating scalability using (**a**) GeoLife dataset (**b**) Genome dataset.

6. Conclusions

In this paper, we investigate the problem of similarity search under edit distance. The edit distance computation is a quadratic time operation, which in its canonical implementation renders a chain of computational dependencies that makes the parallelization of the algorithm difficult. We argue that the parallel similarity search under edit distance is well suited to be executed effectively in both shared-memory and distributed-memory environments. It performs even better in hybrid environments with a setting of multicore CPUs and GPUs. We introduce three parallel algorithms: namely, inter-task parallelism, intra-task parallelism, and a combination of both approaches. We conducted an extensive

set of experiments on real datasets to prove the performance of our algorithms. The experimental results have revealed that the hybrid parallelism approach which is based on distributed as well as shared memory architecture outperforms the other two approaches. Our hybrid approach consistently performs better due to its intra-task parallelism and ability to efficiently utilize on-chip shared memory that significantly reduces the communication time. The hybrid approach achieves a speedup of 18 and 13 over inter-task and intra-task parallelism approaches, respectively. The hybrid approach is scalable and can be further speeded up on parallel hardware such as multi-core CPUs and GPGPUs. Moreover, a simple yet effective data distribution scheme has been introduced to balance the workload across all the machines in the cluster. The experimental results have demonstrated that the new data distribution scheme achieved superior performance over its counterpart.

Author Contributions: Conceptualization, M.K. and M.M.Y.; methodology, M.K.; software, M.K.; validation, M.K. and M.U.S.; writing; M.K.; supervision, M.M.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The link to Genome data set is https://www.ncbi.nlm.nih.gov/ and GeoLife GPS Trajectories is https://www.microsoft.com/en-us/download/details.aspx?id=52367&from=https%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fdownloads%2Fb16d359d-d164-469e-9fd4-daa38f2b2e13%2F.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Prasetya, D.D.; Wibawa, A.P.; Hirashima, T. The performance of text similarity algorithms. *Int. J. Adv. Intell. Inform.* **2018**, *4*, 63–69. [CrossRef]
2. Levenshtein, V. Binary codes capable of correcting spurious insertions and deletion of ones. *Probl. Inf. Transm.* **1965**, *1*, 8–17.
3. Wagner, R.A.; Fischer, M.J. The string-to-string correction problem. *J. ACM (JACM)* **1974**, *21*, 168–173. [CrossRef]
4. Damerau, F.J. A technique for computer detection and correction of spelling errors. *Commun. ACM* **1964**, *7*, 171–176. [CrossRef]
5. Jaro, M.A. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *J. Am. Stat. Assoc.* **1989**, *84*, 414–420. [CrossRef]
6. Winkler, W.E. *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*; Bureau of the Censu: Washington, DC, USA, 1990.
7. Smith, T.F.; Waterman, M.S. Identification of common molecular subsequences. *J. Mol. Biol.* **1981**, *147*, 195–197. [CrossRef]
8. Needleman, S.B.; Wunsch, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* **1970**, *48*, 443–453. [CrossRef]
9. Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences. *Commun. ACM* **1975**, *18*, 341–343. [CrossRef]
10. Kondrak, G. N-gram similarity and distance. In Proceedings of the International Symposium on String Processing and Information Retrieval, Buenos Aires, Argentina, 2–4 November 2005; Springer: Berlin/Heidelberg, Germany, 2005; pp. 115–126. [CrossRef]
11. Khalid, M. Bulk Data Processing of Parallel String Similarity Measures. Ph.D. Dissertation, University of the Punjab, Lahore, Punjab, Pakistan, 2021.
12. Minghe, Y.; Guoliang, L.D.D.; Feng, J. String similarity search and join: A survey. *Front. Comput. Sci.* **2016**, *10*, 399–417. [CrossRef]
13. Nunes, L.S.; Bordim, J.L.; Nakano, K.; Ito, Y. A fast approximate string matching algorithm on GPU. In Proceedings of the 2015 Third international symposium on computing and networking (CANDAR), Sapporo, Japan, 8–11 December 2015; pp. 188–192.
14. Nunes, L.S.; Bordim, J.L.; Nakano, K.; Ito, Y. A memory-access-efficient implementation of the approximate string matching algorithm on GPU. In Proceedings of the 2016 Fourth International Symposium on Computing and Networking (CANDAR), Hiroshima, Japan, 22–25 November 2016; pp. 483–489.
15. Chen, X.; Wang, C.; Tang, S.; Yu, C.; Zou, Q. CMSA: A heterogeneous CPU/GPU computing system for multiple similar RNA/DNA sequence alignment. *BMC Bioinform.* **2017**, *18*, 315. [CrossRef]
16. Jiang, Y.; Deng, D.; Wang, J.; Li, G.; Feng, J. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In Proceedings of the Joint EDBT/ICDT 2013 Workshops, Genoa, Italy, 18–22 March 2013; pp. 341–348. [CrossRef]
17. Zhou, J.; Guo, Q.; Jagadish, H.; Krcal, L.; Liu, S.; Luan, W.; Tung, A.K.; Yang, Y.; Zheng, Y. A generic inverted index framework for similarity search on the gpu. In Proceedings of the 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 16–19 April 2018; pp. 893–904. [CrossRef]

18. Ho, T.; Oh, S.R.; Kim, H. A parallel approximate string matching under Levenshtein distance on graphics processing units using warp-shuffle operations. *PLoS ONE* **2017**, *12*, e0186251. [CrossRef]
19. Groth, T.; Groppe, S.; Koppehel, M.; Pionteck, T. Parallelizing Approximate Search on Adaptive Radix Trees. In Proceedings of the SEBD, Villasimius, Sardinia, Italy, 21–24 June 2020.
20. Ji, S.; Li, G.; Li, C.; Feng, J. Efficient interactive fuzzy keyword search. In Proceedings of the 18th International Conference on World Wide Web, Madrid, Spain, 20–24 April 2009; pp. 371–380.
21. Chaudhuri, S.; Kaushik, R. Extending autocompletion to tolerate errors. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, Providence , RI, USA, 29 June–2 July 2009; pp. 707–718.
22. Li, G.; Ji, S.; Li, C.; Feng, J. Efficient fuzzy full-text type-ahead search. *VLDB J.* **2011**, *20*, 617–640. [CrossRef]
23. Deng, D.; Li, G.; Feng, J.; Li, W.S. Top-k string similarity search with edit-distance constraints. In Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE), Brisbane, QLD, Australia, 8–12 April 2013; pp. 925–936.
24. Lu, W.; Du, X.; Hadjieleftheriou, M.; Ooi, B.C. Efficiently Supporting Edit Distance Based String Similarity Search Using B^+-Trees. *IEEE Trans. Knowl. Data Eng.* **2014**, *26*, 2983–2996. [CrossRef]
25. Zhang, Z.; Hadjieleftheriou, M.; Ooi, B.C.; Srivastava, D. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, Indianapolis, IN, USA, 6–10 June 2010; pp. 915–926.
26. Farivar, R.; Kharbanda, H.; Venkataraman, S.; Campbell, R.H. An algorithm for fast edit distance computation on GPUs. In Proceedings of the 2012 Innovative Parallel Computing (InPar), San Jose, CA, USA, 13–14 May 2012; pp. 1–9. [CrossRef]
27. Wang, X.; Ding, X.; Tung, A.K.; Zhang, Z. Efficient and effective knn sequence search with approximate n-grams. *Proc. VLDB Endow.* **2013**, *7*, 1–12. [CrossRef]
28. Chen, W.; Chen, J.; Zou, F.; Li, Y.F.; Lu, P.; Wang, Q.; Zhao, W. Vector and line quantization for billion-scale similarity search on GPUs. *Future Gener. Comput. Syst.* **2019**, *99*, 295–307. [CrossRef]
29. Johnson, J.; Douze, M.; Jégou, H. Billion-scale similarity search with gpus. *IEEE Trans. Big Data* **2019**, *7*, 535–547. [CrossRef]
30. Li, C.; Wang, B.; Yang, X. VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. In Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), Vienna, Austria, 23–27 September 2007; Volume 7, pp. 303–314.
31. Kim, M.S.; Whang, K.Y.; Lee, J.G.; Lee, M.J. n-gram/2L: A space and time efficient two-level n-gram inverted index structure. In Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), Trondheim, Norway, 30 August–2 September 2005; pp. 325–336.
32. Behm, A.; Ji, S.; Li, C.; Lu, J. Space-constrained gram-based indexing for efficient approximate string search. In Proceedings of the 2009 IEEE 25th International Conference on Data Engineering, Shanghai, China, 29 March–2 April 2009; pp. 604–615.
33. Qin, J.; Wang, W.; Lu, Y.; Xiao, C.; Lin, X. Efficient exact edit similarity query processing with the asymmetric signature scheme. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, Athens, Greece, 12–16 June 2011; pp. 1033–1044.
34. Wang, J.; Li, G.; Feng, J. Can we beat the prefix filtering? An adaptive framework for similarity join and search. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, 20–24 May 2012; pp. 85–96.
35. Yang, X.; Wang, B.; Li, C. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, 9–12 June 2008; pp. 353–364.
36. Behm, A.; Li, C.; Carey, M.J. Answering approximate string queries on large data sets using external memory. In Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, Hannover, Germany, 11–16 April 2011; pp. 888–899.
37. Qin, J.; Xiao, C.; Hu, S.; Zhang, J.; Wang, W.; Ishikawa, Y.; Tsuda, K.; Sadakane, K. Efficient query autocompletion with edit distance-based error tolerance. *VLDB J.* **2020**, *29*, 919–943. [CrossRef]
38. Zhang, H.; Zhang, Q. Minsearch: An efficient algorithm for similarity search under edit distance. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Virtual Event, 6–10 July 2020; pp. 566–576. [CrossRef]
39. Yang, Z.; Yu, J.; Kitsuregawa, M. Fast algorithms for top-k approximate string matching. In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, Atlanta, GA, USA, 11–15 July 2010.
40. Mishra, S.; Gandhi, T.; Arora, A.; Bhattacharya, A. Efficient edit distance based string similarity search using deletion neighborhoods. In Proceedings of the Joint EDBT/ICDT 2013 Workshops, Genoa, Italy, 18–22 March 2013; pp. 375–383. [CrossRef]
41. Wang, J.; Li, G.; Deng, D.; Zhang, Y.; Feng, J. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea, 13–17 April 2015; pp. 519–530.
42. McCauley, S. Approximate similarity search under edit distance using locality-sensitive hashing. *arXiv* **2019**, arXiv:1907.01600.
43. Yu, M.; Wang, J.; Li, G.; Zhang, Y.; Deng, D.; Feng, J. A unified framework for string similarity search with edit-distance constraint. *VLDB J.* **2017**, *26*, 249–274. [CrossRef]
44. Pranathi, P.; Karthikeyan, C.; Charishma, D. String similarity search using edit distance and soundex algorithm. *Int. J. Eng. Adv. Technol. (IJEAT)* **2019**, *8*, 2249–8958.

45. Deng, D.; Li, G.; Feng, J. A pivotal prefix based filtering algorithm for string similarity search. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, Snowbird, UT, USA, 22–27 June 2014; pp. 673–684.
46. Matsumoto, T.; Yiu, M.L. Accelerating exact similarity search on cpu-gpu systems. In Proceedings of the 2015 IEEE International Conference on Data Mining, Atlantic City, NJ, USA, 14–17 November 2015; pp. 320–329. [CrossRef]
47. Shehab, M.A.; Ghadawi, A.A.; Alawneh, L.; Al-Ayyoub, M.; Jararweh, Y. A hybrid CPU-GPU implementation to accelerate multiple pairwise protein sequence alignment. In Proceedings of the 2017 8th International Conference on Information and Communication Systems (ICICS), Irbid, Jordan, 4–6 April 2017; pp. 12–17.
48. Edmiston, E.W.; Core, N.G.; Saltz, J.H.; Smith, R.M. Parallel processing of biological sequence comparison algorithms. *Int. J. Parallel Program.* **1988**, *17*, 259–275. [CrossRef]
49. Zhong, C.; Chen, G.L. Parallel algorithms for approximate string matching on PRAM and LARPBS. *J. Softw.* **2004**, *15*, 159–169.
50. Man, D.; Nakano, K.; Ito, Y. The approximate string matching on the hierarchical memory machine, with performance evaluation. In Proceedings of the 2013 IEEE 7th International Symposium on Embedded Multicore Socs, Tokyo, Japan, 26–28 September 2013; pp. 79–84.
51. Zhang, J.; Lan, H.; Chan, Y.; Shang, Y.; Schmidt, B.; Liu, W. BGSA: A bit-parallel global sequence alignment toolkit for multi-core and many-core architectures. *Bioinformatics* **2019**, *35*, 2306–2308. [CrossRef] [PubMed]
52. Myers, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM (JACM)* **1999**, *46*, 395–415. [CrossRef]
53. Hyyrö, H. A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nord. J. Comput.* **2003**, *10*, 29–39.
54. Xu, K.; Cui, W.; Hu, Y.; Guo, L. Bit-parallel multiple approximate string matching based on GPU. *Procedia Comput. Sci.* **2013**, *17*, 523–529. [CrossRef]
55. Lin, C.H.; Wang, G.H.; Huang, C.C. Hierarchical parallelism of bit-parallel algorithm for approximate string matching on GPUs. In Proceedings of the 2014 IEEE Symposium on Computer Applications and Communications, Weihai, China, 26–27 July 2014; pp. 76–81.
56. Sadiq, M.U.; Yousaf, M.M. Distributed Algorithm for Parallel Edit Distance Computation. *Comput. Inform.* **2020**, *39*, 757–779. [CrossRef]
57. 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. *Clust. Comput.* **2020**, *23*, 879–894. [CrossRef]
58. Yousaf, M.M.; Sadiq, M.A.; Aslam, L.; Ul Qounain, W.; Sarwar, S. A novel parallel algorithm for edit distance computation. *Mehran Univ. Res. J. Eng. Technol.* **2018**, *37*, 223–232. [CrossRef]
59. The National Center for Biotechnology Information. 1988. Available online: https://www.ncbi.nlm.nih.gov/ (accessed on 20 March 2022).
60. Zheng, Y.; Zhang, L.; Xie, X.; Ma, W.Y. Mining Interesting Locations and Travel Sequences from GPS Trajectories. In Proceedings of the Proceedings of the 18th International Conference on World Wide Web, New York, NY, USA, 20–24 April 2009; pp. 791–800. [CrossRef]
61. Python Geo-hash Library. 2011. Available online: https://pypi.org/project/python-geohash/ (accessed on 22 March 2022).