Improved MPC Algorithms for Edit Distance and Ulam Distance

Mahdi Boroujeni, Mohammad Ghodsi, and Saeed Seddighin

Abstract—Edit distance is one of the most fundamental problems in combinatorial optimization to measure the similarity between strings. Ulam distance is a special case of edit distance where no character is allowed to appear more than once in a string. Recent developments have been very fruitful for obtaining fast and parallel algorithms for both edit distance and Ulam distance. In this work, we present an almost optimal MPC (massively parallel computation) algorithm for Ulam distance and improve MPC algorithms for edit distance. Our algorithm for Ulam distance is almost optimal in the sense that (1) the approximation factor of our algorithm is $1+\epsilon$, (2) the round complexity of our algorithm is constant, (3) the total memory of our algorithm is almost linear ($\widetilde{O}_\epsilon(n)$), and (4) the overall running time of our algorithm is almost linear which is the best known for Ulam distance. We also improve the work of Hajiaghayi *et al.* for edit distance in terms of total memory. The best previously known MPC algorithm for edit distance requires $\widetilde{O}(n^{2x})$ machines when the memory of each machine is bounded by $\widetilde{O}(n^{1-x})$. In this work, we improve the number of machines to $\widetilde{O}(n^{(9/5)x})$ while keeping the memory limit intact. Moreover, the round complexity of our algorithm is constant and the total running time of our algorithm is truly subquadratic. However, our improvement comes at the expense of a constant factor in the approximation guarantee of the algorithm. This improvement is inspired by the recent techniques of Boroujeni *et al.* and Chakraborty *et al.* for obtaining truly subquadratic time algorithms for edit distance.

Index Terms—MapReduce, parallel algorithms, approximation algorithms, ulam distance, edit distance

1 Introduction

String similarity measures are among the most fundamental problems in computer science. The *edit distance* (a.k.a *Levenshtein distance*) is the most notable example of it. This problem has lots of applications in several fields such as computational biology, natural language processing, and information theory. In theoretical computer science, too, the problem has been very central and fundamental; the problem of computing the edit distance is a textbook example for dynamic programming.

In edit distance, we are given two strings s and $\overline{s}$ and we wish to transform s into $\overline{s}$ using the smallest number of edit operations. In each operation, we are allowed to (i) insert a character at a specific position, (ii) remove a character, or (iii) modify a character. For simplicity, we assume that all these edit operations incur equal costs. For two strings s and $\overline{s}$, $|s| = |\overline{s}| = n$, a classic dynamic program finds the edit distance between them in time $O(n^2)$. The idea is to define auxiliary variables $d_{i,j}$'s which denote the edit distance between the first i characters of s and the first j characters of $\overline{s}$. Next, we iteratively determine the values of the auxiliary variables based on the following formula.

$$d_{i,j} = \begin{cases} d_{i-1,j-1} & \text{if } s[i] = \overline{s}[j], \\ 1 + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}\} & \text{if } s[i] \neq \overline{s}[j]. \end{cases}$$

Although naïve, the above algorithm is almost the best we can do from a theoretical perspective. Since the late 60's, several studies were focused on improving the quadratic running time of the problem; however, thus far, the best-known algorithm runs in time $O(n^2/\log^2 n)$ [2]. The shortcoming of these studies is partly addressed by the work of Backurs and Indyk [3] (STOC'15) wherein the authors show a truly subquadratic time algorithm is impossible to achieve unless a widely believed conjecture (SETH[1]) fails.

Unfortunately, the quadratic dependency of the running time on the size of the input makes it impossible to use such algorithms for large inputs in practice. For example, a human genome consists of almost three billion base pairs that need to be incorporated in similarity measurements. Therefore, the need to compute/approximate these measures in better than quadratic time has led to several algorithmic breakthroughs. Near linear time solutions have been studied in a series of works [4], [5], [6], [7], [8], [9], [10] culminating in polylogarithmic approximation. Recently, a quantum algorithm is given for edit distance that approximates the solution within a constant factor in truly subquadratic time by exploiting triangle inequality [11]. Subsequent work discovers a novel classic replacement for the quantum techniques and obtains a truly

- *Mahdi Boroujeni is with the Department of Computer Engineering, Sharif University of Technology, Tehran 79417-76655, Iran. E-mail: safarnejad@ce.sharif.edu.*
- *Mohammad Ghodsi is with the Department of Computer Engineering, Sharif University of Technology, Tehran 79417-76655, Iran and also with the School of Computer Science, Institute for Research in Fundamental Sciences, Tehran 19538-33511, Iran. E-mail: ghodsi@sharif.edu.*
- *Saeed Seddighin is with Toyota Technological Institute at Chicago, Chicago, IL 60637 USA. E-mail: saeedreza.seddighin@ttic.edu.*

Manuscript received 22 Aug. 2020; revised 1 Apr. 2021; accepted 5 Apr. 2021. Date of publication 29 Apr. 2021; date of current version 24 May 2021.
(Corresponding author: Mahdi Boroujeni.)
Recommended for acceptance by J. Zola.
Digital Object Identifier no. 10.1109/TPDS.2021.3076534

1. The *strong exponential time hypothesis* states that no algorithm can solve the boolean satisfiability (CNF-SAT) problem in time $O(2^{n(1-\epsilon)})$.

TABLE 1
Our Results are Shown Along With Previous Massively Parallel Algorithms for Edit Distance

Problem	Reference	Approximation Factor	# Rounds	Memory of Each Machine	# Machines	Total Running Time
Our Results						
Ulam Distance	Theorem 4	$1+\epsilon$	2	$\widetilde{O}_\epsilon(n^{1-x})$	$\widetilde{O}_\epsilon(n^x)$	$\widetilde{O}_\epsilon(n)$
Edit Distance	Theorem 9	$3+\epsilon$	4	$\widetilde{O}_\epsilon(n^{1-x})$	$\widetilde{O}_\epsilon(n^{(9/5)x})$	$\widetilde{O}_\epsilon(n^{2-\min(\frac{1-x}{6},\frac{2x}{5})})$
Previous Work						
Edit Distance	[11]	$1+\epsilon$	$O(\log n)$	$\widetilde{O}_\epsilon(n^{8/9})$	$\widetilde{O}_\epsilon(n^{8/9})$	$\widetilde{O}_\epsilon(n^{2.6})$
Edit Distance	[20]	$1+\epsilon$	2	$\widetilde{O}_\epsilon(n^{1-x})$	$\widetilde{O}_\epsilon(n^{2x})$	$\widetilde{O}_\epsilon(n^2)$

Our algorithm for edit distance improves the previous algorithms in terms of total memory and total running time.

subquadratic time algorithm within a constant factor for classic computers [12]. Subsequent works improve the running time of the algorithm. Kouckỳ and Saks [13] and Brakensiek and Rubinstein [14] independently present near-linear time constant-factor approximation algorithms for edit distance where the input strings are far from each other. Finally, Andoni and Nosatzki [15] provide a similar near-linear time algorithm which does not impose any condition on input strings. Note that the approximation factors of these algorithms are exponentially or doubly exponentially large constants. The best algorithm within a factor of $3+\epsilon$ is presented by Goldenberg, Rubinstein, and Saha [16] with a running time of $\widetilde{O}(n^{1.6+o(1)})$.

Ulam distance is a special case of edit distance wherein the input strings s and $\overline{s}$ have no repetitive characters.[2] This additional restriction to the input makes the problem relatively easier to solve as Ulam distance admits an almost linear time solution. Notice that, verifying whether s and $\overline{s}$ are equal or not requires $\Omega(n)$ operations and thus there is no hope to solve or even approximate the solution in sublinear time. However, for the large solution regime, a constant approximate solution can be found in time $\widetilde{O}_\epsilon(\sqrt{n}+n/d)$ where d is the distance of the two input strings [17]. This was later improved to a $1+\epsilon$ approximation algorithm but for a more relaxed notion of distance wherein character substitution is not allowed [17], [18], [19]. The algorithm of Naumovitz *et al.* [17] obtains a $2+\epsilon$ approximate solution for the more conventional formulation of Ulam distance.

Another line of attack is to design efficient algorithms for string similarity measures is parallel computing [11], [20]. Motivated by modern fast, efficient, easy-to-use massively parallel distributed computing platforms such as MapReduce, Hadoop, and Spark [21], [22], [23], the massively parallel computation (MPC) model [24], [25], [26], [27] has been proposed and extensively studied to understand the power and limitations of these parallel computing platforms.

In contrast to the PRAM model where an $\Omega(\log n)$ factor in the round complexity is usually inevitable, MPC allows for sublogarithmic round complexity [24], [28], [29]. In the MPC model, each machine has unlimited access to its memory; however, two machines can only interact in between two rounds. Thus, a central parameter in this setting is the round complexity of the algorithm since network communication is the typical main bottleneck in practice. The ultimate goal is developing constant-round algorithms, which are highly desirable in practice.

The MPC Model. We assume throughout this paper that the input contains two strings of length n. In the MPC model [24], [25], [26], [27], the number of machines and the local memory size on each machine should be relatively smaller than the input size of the problem. Therefore, we fix an $0 < x < 1$ and consider the memory of each machine to be $\widetilde{O}_\epsilon(n^{1-x})$ and aim to minimize the number of machines needed to run the algorithm. In the MPC model, each algorithm runs in a number of rounds. In each round, every machine makes some computation on the data assigned to the machine. No communication between machines is allowed during a round. Between two rounds, machines are allowed to communicate so long as each machine receives no more communication than its memory. Any data that is the output of a machine must be computed locally from the data residing on the machine, and initially, the input data is distributed across the machines.

In this work, we present an almost optimal MPC algorithm for Ulam distance and improve MPC algorithms for edit distance. An overview of our results is shown in Table 1. Our algorithm for Ulam distance is almost optimal in the sense that (1) the approximation factor of our algorithm is $1+\epsilon$, (2) the round complexity of our algorithm is constant, (3) the total memory of our algorithm is almost linear ($\widetilde{O}_\epsilon(n)$), and (4) the overall running time of our algorithm is almost linear which is the best known for Ulam distance. Similar to edit distance and longest common subsequence (LCS) which are considered as dual problems, Ulam distance and longest increasing subsequence (LIS) are also seen as dual problems. LIS is equivalent to a special case of LCS where each string can contain each character at most once. In that sense, our result for Ulam distance complements the work of Im *et al.* [30], wherein a similar result is presented for LIS. It is worth mentioning that similar to Ulam distance, for which the running time improves for similar strings, LIS also admits very fast (polylogarithmic time) solutions when the two strings share a large subsequence [31]. However, these techniques do not improve the time or memory complexity of the solution in the MPC model since adaptive sampling is an inherent barrier for the MPC model.

The best previously known MPC algorithm for edit distance requires $\widetilde{O}_\epsilon(n^{2x})$ machines when the memory of each

2. W.l.o.g. in Ulam distance, s and $\overline{s}$ can be considered as two permutations of $[n]=\{1,\ldots,n\}$.

machine is bounded by $\widetilde{O}_\epsilon(n^{1-x})$ [20]. Thus, the total memory of their algorithm is $\widetilde{O}_\epsilon(n^{1+x})$. Indeed, the main question which is left unanswered is how best can one approximate edit distance in the MPC model with near-linear memory? In this work, we take a step forward toward the answer by improving the total memory of the algorithm to $\widetilde{O}_\epsilon(n^{1+(4/5)x})$. Moreover, the round complexity of our algorithm is constant and the total running time of our algorithm is truly subquadratic. However, our improvements come at the expense of a constant factor in the approximation guarantee of the algorithm. This improvement is inspired by the recent techniques of Boroujeni *et al.* [11] and Chakraborty*et al.* [12] for obtaining truly subquadratic time algorithms for edit distance. Moreover, using specific parameters and $\widetilde{O}_\epsilon(n^{5/17})$ machines, the total running time of our algorithm is $O(n^{1.883})$ and the parallel running time of our algorithm is $O(n^{1.353})$.

2 Preliminaries

In this work, we consider two problems, namely edit distance and Ulam distance. The input to both problems consists of two strings s and $\overline{s}$ both of length n. Let us begin by formally defining the edit distance of two strings s and $\overline{s}$.

> edit distance
> input: Two strings s and $\overline{s}$, both of size n.
> solution: The smallest number of operations that we need to perform on s to transform it into $\overline{s}$. We are allowed to (i) add a character at some position, (ii) remove a character, and (iii) change a character. All these operations come at a cost of 1.

For instance, for $s = \mathsf{elephant}$ and $\overline{s} = \mathsf{relevant}$, we can transform s into $\overline{s}$ by performing three operations

$$\mathsf{elephant} \xrightarrow{\text{insert 'r' at position0}} \mathsf{relephant} \xrightarrow{\text{replace 'p' with 'v' atposition4}} \mathsf{relevhant} \xrightarrow{\text{delete 'h' atposition5}} \mathsf{relevant}.$$

Therefore, $\mathsf{ed}(\text{"}\mathsf{elephant}\text{"}, \text{"}\mathsf{relevant}\text{"})$ is at most 3. With more inspection, it can be shown that here the edit distance is equal to 3. The other measure that we are interested in is the Ulam distance. Ulam distance is a special case of edit distance wherein each character appears at most once in each string. We denote the edit distance between s and $\overline{s}$ by $\mathsf{ed}(s, \overline{s})$, and their Ulam distance by $\mathsf{ulam}(s, \overline{s})$.

We consider the MPC setting in which we are provided with a certain number of machines each having a memory of at most $\widetilde{O}_\epsilon(n^{1-x})$. In each round, every machine is fed with a piece of information that fits within its memory and sequentially executes a stream of operations on the input. At the end of each round, every machine outputs some information whose length is bounded by $\widetilde{O}_\epsilon(n^{1-x})$. The outputs then are given as input to the machines at the beginning of the next round. We are interested in massively parallel algorithms that run in a constant number of rounds.

In our algorithms, we divide the strings into pieces of size $\mathcal{B} = n^{1-y}$ and refer to them as blocks. We denote each block i of s by $s[\ell_i, r_i]$ (thus, $r_i = \ell_i + \mathcal{B} - 1$). For the sake of analysis, we sometimes fix an optimal solution opt and denote by $\overline{s}[\alpha_i, \beta_i]$ the substring of $\overline{s}$ that corresponds to block $s[\ell_i, r_i]$ according to opt. Blocks of s and their corresponding blocks in $\overline{s}$ are shown in Fig. 1. We often denote the string positions by γ, κ, p, and q. For instance, $s[\gamma, \kappa]$ denotes the substring of s starting from position γ and ending at position κ. Also, to simplify the analysis, we make use of two notations $\widetilde{O}$ and $\widetilde{O}_\epsilon$ where both suppress $\mathsf{poly}(\log n)$ factors and the latter also hides $\mathsf{poly}(1/\epsilon)$ terms.

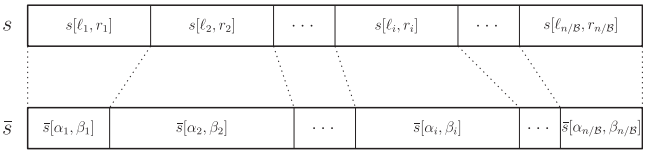

Fig. 1. The partitioning of s into n^y blocks of size $\mathcal{B} = n^{1-y}$ and the transformation of the blocks into their matches via opt is shown in this figure. Note that matched substrings span $\overline{s}$, $\ell_1 = \alpha_1 = 1$ and $r_{n/\mathcal{B}} = \beta_{n/\mathcal{B}} = n$.

3 Our Results

In this section, we briefly describe our massively parallel algorithms for Ulam distance and edit distance. The details of our algorithms are presented in Sections 4 and 5.

3.1 Ulam Distance

Here, we briefly present our massively parallel algorithm for Ulam distance using $\widetilde{O}_\epsilon(n^x)$ machines, each with a memory of $\widetilde{O}_\epsilon(n^{1-x})$. Recall that, the $\widetilde{O}_\epsilon$ notation suppresses $\mathsf{poly}(\log n)$ and $\mathsf{poly}(1/\epsilon)$ terms. The details of our algorithm are presented in Section 4. Our algorithm approximates Ulam distance within a factor of $1+\epsilon$ for an arbitrarily small constant $\epsilon > 0$ with high probability. The total running time[3] of our algorithm is $\widetilde{O}_\epsilon(n)$. Moreover, our algorithm runs in two MPC rounds.

In the sequential setting, a trivial $\Omega(n)$ lower bound holds for the running time of approximating Ulam distance within a $1+\epsilon$ factor. However, for the high distance regime, the running time can be improved to $\widetilde{O}_\epsilon(n/d + \sqrt{n})$ if character substitution is not allowed [17]. However, even when the distance is large, no $1+\epsilon$ approximation algorithm is known for Ulam distance if character substitution is allowed.

Recall that, Ulam distance is a special case of edit distance where the input strings s and $\overline{s}$ have no repetitive characters. Moreover, recall that for simplicity, we assume both input strings have length n. Recall that, we break s into blocks of size $\mathcal{B} = n^{1-y}$. In our algorithm for Ulam distance, we set $y = x$; hence, each block fits into the memory of one machine. We assume for simplicity that $\mathcal{B}$ is an integer.

To better understand our algorithm, we first illustrate how a block of s is mapped to a substring of $\overline{s}$. Note that, if for each block of s we know where it transforms into in opt, we can break the problem into n^x subproblems and solve Ulam distance for each of them individually. Since such information is not present, we form a set of candidate substrings on $\overline{s}$ for every block of s and compute the Ulam distances between each block of s and its corresponding candidate substrings. The construction in our algorithm ensures that each block

3. summing the running time of all machines

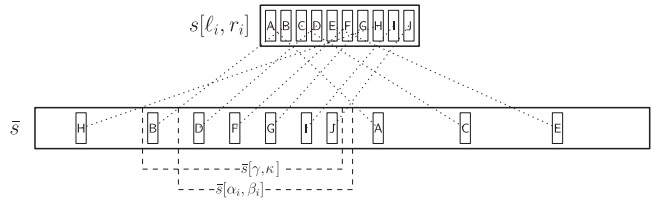

Fig. 2. If the Ulam distance between $s[\ell_i, r_i]$ and $\overline{s}[\alpha_i, \beta_i]$ is small, we solve local Ulam distance which gives us the best match of $s[\ell_i, r_i]$ in $\overline{s}$. Such a substring is shown as $\overline{s}[\gamma, \kappa]$ in this figure. We prove that $\overline{s}[\gamma, \kappa]$ and $\overline{s}[\alpha_i, \beta_i]$ intersect, which narrows down our search for $\overline{s}[\alpha_i, \beta_i]$. In this figure, common characters are connected via dotted lines.

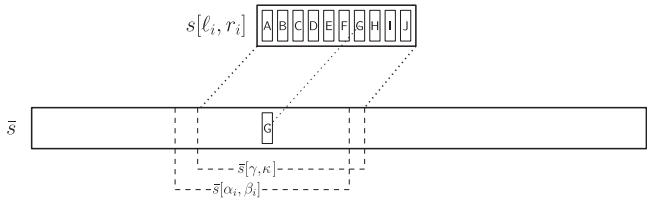

Fig. 3. If the Ulam distance between $s[\ell_i, r_i]$ and $\overline{s}[\alpha_i, \beta_i]$ is large but the number of unchanged characters is not small, we try to find a common character by randomized sampling. If character G is a common character, we extend it to form $\overline{s}[\gamma, \kappa]$ which narrows down our search for $\overline{s}[\alpha_i, \beta_i]$ to indices near γ and κ.

$s[\ell_i, r_i]$ has a corresponding candidate substring $\overline{s}[\gamma, \kappa]$ which is relatively close to $\overline{s}[\alpha_i, \beta_i]$. Recall that, $s[\ell_i, r_i]$ transforms into $\overline{s}[\alpha_i, \beta_i]$ in opt. The construction of the candidate substrings and computing the Ulam distance between each block and its candidate substrings are done in the first round of our algorithm. In the second round, we compute an approximately optimal transformation of s into $\overline{s}$ using a dynamic program on the information gathered in the first round.

In the following, we briefly formalize the ideas mentioned above. Without loss of generality, we assume that substrings $\overline{s}[\alpha_i, \beta_i]$ partition $\overline{s}$, i.e., each character of $\overline{s}$ belongs to exactly one $\overline{s}[\alpha_i, \beta_i]$. Recall that, $1 + \epsilon$ is the desired approximation factor of our algorithm where $\epsilon > 0$ is an arbitrarily small constant number. We fix another small number $\epsilon' = \epsilon/2$ for the sake of analysis. Let $u_i = \mathsf{ulam}(s[\ell_i, r_i], \overline{s}[\alpha_i, \beta_i])$ be the Ulam distance between the i'th block of s and its corresponding substring of $\overline{s}$ in opt. We call a substring $\overline{s}[\alpha'_i, \beta'_i]$ an approximately optimal candidate substring for a block $s[\ell_i, r_i]$ if both of the following conditions hold:

$$\alpha_i \leq \alpha'_i \leq \alpha_i + \epsilon' u_i \tag{1}$$

$$\beta_i - \epsilon' u_i \leq \beta'_i \leq \beta_i. \tag{2}$$

Note that using approximately optimal candidate substrings instead of the optimal ones imposes an additive error of at most $2\epsilon' u_i$ for each block which sum up to at most a total additive error of $2\epsilon' \cdot \mathsf{ulam}(s, \overline{s})$. For a small enough ϵ', this total additive error is in the range of the desired approximation factor. In our algorithm, we construct an approximately optimal candidate substring in the first phase with high probability unless u_i is too large and the number of common characters of $s[\ell_i, r_i]$ and $\overline{s}[\alpha_i, \beta_i]$ is too small (see Lemma 3). Note that, in such cases, one can remove all characters of $s[\ell_i, r_i]$ and add all characters of $\overline{s}[\alpha_i, \beta_i]$ instead, without losing more than a $1 + O(\epsilon')$ multiplicative factor. This claim is proven in Theorem 4.

In what follows, we explore some features of the approximately optimal candidate substrings. In our algorithm, if u_i is relatively small, we solve a local version of Ulam distance which finds a substring $\overline{s}[\gamma, \kappa]$ and guarantees that its endpoints are relatively close to the endpoints of $\overline{s}[\alpha_i, \beta_i]$. Next, we construct several candidate substrings with starting points near γ and ending points near κ. We prove that at least one of these candidate substrings satisfies both Conditions 1 and 2. This case is illustrated in Fig. 2

In cases that u_i is relatively large and c_i, the number of unchanged characters of transforming $s[\ell_i, r_i]$ into $\overline{s}[\alpha_i, \beta_i]$ in opt, is more than $\epsilon\mathcal{B}/4$, we sample each character of $s[\ell_i, r_i]$ with an independent probability of $\theta = (8/\epsilon'\mathcal{B})\log n$ and put them in a hitting set I. We show that at least one of the characters of I remain unchanged in the transformation of $s[\ell_i, r_i]$ into $\overline{s}[\alpha_i, \beta_i]$ in opt with high probability. For such an unchanged character, we make a substring $\overline{s}[\gamma, \kappa]$ based on the position of the unchanged characters and show that its endpoints are near the endpoints of $\overline{s}[\alpha_i, \beta_i]$. Note that since $\overline{s}$ contains distinct characters, it is easy to find where an unchanged character of s maps in $\overline{s}$. We prove this claim in Lemma 2. Afterward, we construct several candidate substrings with starting points near γ's and ending points near κ's. We prove in Lemma 3 that at least one of these candidate substrings satisfies both Conditions 1 and 2 with high probability. This case is shown in Fig. 3.

Recall that, the starting point of an approximately optimal candidate substring α'_i is allowed to be up to $\epsilon' u_i$ characters away from α_i. Hence, we define a gap $\mathcal{G}_i = \epsilon' u_i$ and only inspect the starting points and ending points with indices divisible by $\mathcal{G}_i$. We assume for simplicity that $\mathcal{G}_i$ is an integer. We show that the expected total number of candidate substrings for a block (for both cases) is at most

$$\Big[1 + \log_{1+\epsilon'} n \cdot (1 + \mathcal{B} \cdot (8/\epsilon'\mathcal{B})\log n)(1/\epsilon')\Big](1/\epsilon') = \widetilde{O}_\epsilon(1).$$

Therefore, we assign the computational task corresponding to each block of s, including the computation of the Ulam distance between the block and its corresponding candidate substrings, to one machine. Note that since $\overline{s}$ does not have any repetitive characters, the only information needed from $\overline{s}$ to be fed to the machine is the location of each characters of $s[\ell_i, r_i]$ in $\overline{s}$ if exists, which are $\widetilde{O}_\epsilon(n^{1-x})$ many locations. A more formal description of the first round of our algorithm is given in Algorithm 1. Finally, in the second round of our algorithm, we run a dynamic program to find a $1 + \epsilon$ approximation solution based on the candidate substrings we construct in the first round of the algorithm. This phase is shown in Algorithm 2.

Theorem 4 [restated]. For an arbitrarily small constant $\epsilon > 0$, there exists a massively parallel algorithm with $\widetilde{O}_\epsilon(n^x)$ machines for arbitrary $0 < x < 1/2$, each with a memory of $\widetilde{O}_\epsilon(n^{1-x})$ that approximates the Ulam distance of two strings of length n within a factor of $1 + \epsilon$ in two rounds with high probability. Moreover, the total computation of this algorithm is $\widetilde{O}_\epsilon(n)$.}

3.2 Edit Distance

Note that in the classic setting, edit distance can be solved within linear memory. However, existing massively parallel

algorithms for edit distance such as [11] and [20] use a superlinear amount of memory, the best of which uses $\widetilde{O}_\epsilon(n^{1-x})$ memory for each of its $\widetilde{O}_\epsilon(n^{2x})$ machines. Recall that, the $\widetilde{O}_\epsilon$ notation suppresses $\mathsf{poly}(\log n)$ and $\mathsf{poly}(1/\epsilon)$ terms. In this work, we improve the number of machines and thus the aggregated memory of our algorithm. Our algorithm runs on $\widetilde{O}_\epsilon(n^{(9/5)x})$ machines with memory $\widetilde{O}_\epsilon(n^{1-x})$. However, this comes at the expense of a constant factor loss in the approximation. More precisely, the approximation factor of our algorithm is $3+\epsilon$ and its round complexity is 4. Moreover, the total running time of our algorithm is $\widetilde{O}_\epsilon(n^{2-\min(\frac{1-x}{6},\frac{2x}{5})})$ which is truly subquadratic for any $0 < x < 1$.

Our algorithm uses two different approaches for small distances and large distances. The overall structure of our algorithm for small distances has similarities with the algorithm of [20] but improves it in terms of memory usage. For large distances, which is the hard case, we borrow some ideas from sequential algorithms of [11] and [12] and marry them with new techniques to run the algorithm in the parallel setting.

The very first step of our algorithm is to assume we have a given value n^δ and the task is to verify whether n^δ is relatively close to $\mathsf{ed}(s,\overline{s})$ or it is much smaller than the actual edit distance between s and $\overline{s}$. Previous work such as [11] has shown that such an assumption only adds a multiplicative $\widetilde{O}_\epsilon(1)$ term to the analysis of our algorithm. The overall idea is to try all values of $n^\delta = (1+\epsilon)^i$ for $0 \le i \le \log_{1+\epsilon} n$. In the sequential setting, we start by a small n^δ, and every time we fail to find a solution with that size, we try the next value for n^δ. The first time we find a solution, we are certain that it is a $1+\epsilon$ approximation of the optimal solution. In the parallel setting, we run our algorithm for all values of n^δ in parallel and only consider the smallest n^δ with a valid solution at the end. Therefore, this assumption does not affect the round complexity of our algorithm. Last but not least, our algorithm detects the case of $\mathsf{ed}(s,\overline{s}) = 0$ separately.

Let $1+\epsilon$ be the desired approximation factor of our algorithm for an arbitrarily small constant $\epsilon > 0$. For simplicity, we use $\epsilon' = \epsilon/22$ in the analysis of our algorithm. Our algorithm breaks the problem into n^y subproblems for each block of s. Recall that the memory of each machine is $\widetilde{O}_\epsilon(n^{1-x})$ hence, $\widetilde{O}_\epsilon(n^{y-x})$ blocks of size $\mathcal{B} = n^{1-y}$ fit into the memory of a single machine. Moreover, for a block $s[\ell_i, r_i]$ of s, we call a candidate substring $\overline{s}[\alpha'_i, \beta'_i]$ approximately optimal if both of these conditions hold.

$$\alpha_i \le \alpha'_i \le \alpha_i + \epsilon' n^{\delta-y}, \tag{3}$$

$$\beta_i - \epsilon' n^{\delta-y} - \epsilon' \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\alpha_i, \beta_i]) \le \beta'_i \le \beta_i. \tag{4}$$

We then use one of the following approaches to solve these subproblems based on whether n^δ is relatively small or relatively large.

3.2.1 Small Distances ($\mathbf{n^\delta \le n^{1-x/5}}$)

Note that if we know which part of $\overline{s}$ corresponds to each block of s in opt, the subproblems can be solved easily. Instead of this out of reach information, our algorithm finds $\widetilde{O}_\epsilon(n^y)$ candidate substrings of $\overline{s}$ for each block of s and ensures that at least one of the candidate substrings of each block is relatively close to its match in opt. The construction of the candidate substrings is similar to that of [20]. In this case, our algorithm consists of two rounds. In the first round, we compute the edit distance between each block and its candidate substrings. In the second round, we use the information obtained in the first round and run a dynamic program to find an approximately optimal transformation of s into $\overline{s}$. In the following, we briefly describe the two rounds of our algorithm in the case of small distances.

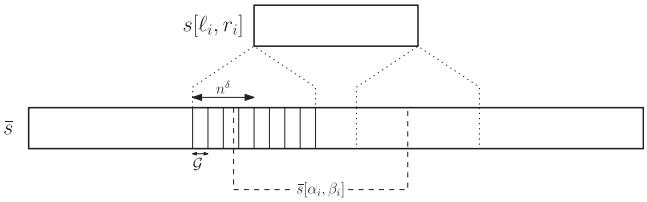

Fig. 4. In our algorithm for edit distance, we form the starting points of candidate substrings for a block $s[\ell_i, r_i]$ as follows. The starting points are in the range of $[\ell_i - n^\delta, \ell_i + n^\delta]$. Moreover, we consider starting points in this interval which are divisible by $\mathcal{G} = \epsilon' n^{\delta-y}$.

If for every block of s, we are able to find an approximately optimal candidate substring, the total additive error caused by using these substrings instead of optimal ones assuming $\mathsf{ed}(s,\overline{s}) \le n^\delta$ is at most

$$\sum_i \epsilon' n^{\delta-y} + \epsilon' n^{\delta-y} + \epsilon' \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\alpha_i, \beta_i]) \le 3\epsilon' n^\delta.$$

Therefore, using these substrings only incurs a multiplicative $1+3\epsilon'$ term to the approximation factor. We claim that if the size of $\overline{s}[\alpha_i, \beta_i]$ is not too small nor too large, our algorithm certainly finds an approximately optimal candidate substring for $s[\ell_i, r_i]$ in the first phase. We provide a formal proof of this claim in Lemma 5.

On the other hand, if $\overline{s}[\alpha_i, \beta_i]$ is too small or too large, we can remove $s[\ell_i, r_i]$ and insert $\overline{s}[\alpha_i, \beta_i]$ by imposing at most a $1+\epsilon'$ multiplicative factor to the approximation factor. We prove this claim in Lemma 6. The construction of candidate substrings is done as follows. Assuming n^δ is an upper bound on the solution size, we conclude that $|\ell_i - \alpha_i| \le n^\delta$. We then define a gap $\mathcal{G} = \lfloor \epsilon' n^{\delta-y} \rfloor$ and consider indices in the range of $[\ell_i - n^\delta, \ell_i + n^\delta]$ which are divisible by $\mathcal{G}$ as the starting points of our candidate substrings. It can be easily shown that one of these starting points meets condition 3. This structure is shown in Fig. 4. The total number of starting points for a block is therefore, $O(n^\delta)/\mathcal{G} = O((1/\epsilon')n^y)$. Moreover, for each starting point, we consider at most $O(\log_{1+\epsilon'} n) = \widetilde{O}_\epsilon(1)$ endpoints. The construction of endpoints is shown in Fig. 5. Each pair of a starting point and an ending point forms a candidate substring. Hence, the total number of candidate substrings for each block is $\widetilde{O}_\epsilon(n^y) \cdot \widetilde{O}_\epsilon(1) = \widetilde{O}_\epsilon(n^y)$. In the case of small distances, we use $y = x$.

Up to this point, our algorithm for the case of a small distance is similar to that of [20]. However, the algorithm of [20] assigns each pair of block/candidate substring to a single machine. On the contrary, since in this case, n^δ is small, starting points of candidate substrings of a block are not far from each other. Therefore, we give several candidate substrings of each block to a machine. This technique

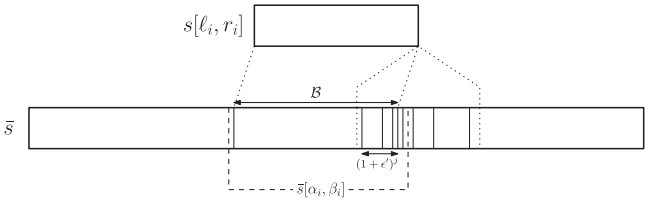

Fig. 5. We form the ending points of candidate substrings for a block $s[\ell_i, r_i]$ and a starting point γ around $\kappa = \gamma + \mathcal{B}$ as $\kappa \pm (1+\epsilon')^j$. Note that we only construct candidate with a length less than $(1/\epsilon')\mathcal{B}$. Moreover, we can safely neglect ending points beyond $\kappa + n^\delta$.

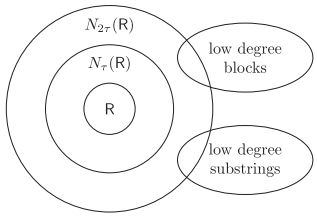

Fig. 6. The graph G_τ is shown in this figure. The set of representatives R is shown with the set of its neighbors with distance τ and 2τ. We show that high degree nodes are in $N_\tau(R)$ with high probability. Other low degree nodes may correspond to blocks or substrings.

reduces the number of machines and therefore the total memory of our algorithm. The total number of machines in our algorithm in this case is

$$n^x \cdot \frac{\widetilde{O}_\epsilon(n^\delta)}{n^{1-x}} = \widetilde{O}_\epsilon(n^{2x-(1-\delta)}).$$

Recall that in the first round of our algorithm, we compute the distances between each block and its candidate substrings. To find the edit distance between a block and one of its candidate substrings in our algorithm, we use a variant of the algorithm of [12] on each machine which uses linear memory, has an approximation factor of $3+\epsilon'$, and runs in time $O(n^{2-1/6})$. Therefore, the total running time of this phase is $\widetilde{O}_\epsilon(n^{2-(1-x)/6})$ and the parallel running time of this phase is $\widetilde{O}_\epsilon(n^{(1-\delta)+(2-1/6)(1-x)})$. In the second round of our algorithm, we find a $1+\epsilon'$ approximate solution based on the information obtained in the first round. Moreover, the total computation of the second round is done on a single machine with a running time of $\widetilde{O}_\epsilon(n^{2y})$.

3.2.2 Large Distances ($\mathbf{n^\delta > n^{1-x/5}}$)

In this case, we apply the triangle inequality technique of [11], randomized sampling technique of [11], and low degree extension technique of [12] to avoid explicitly computing the edit distance for all pairs of blocks and candidate substrings. Similar to the previous case, we partition s into n^y blocks of size $\mathcal{B} = n^{1-y}$. It is known that we can reduce the problem of computing the edit distance for all pairs of blocks and candidate substrings into verifying whether their distance is at most a given threshold τ or much larger than τ [11]. More precisely, similar to Lemma 5, we reduce the edit distance problem to finding the edit distance between smaller block/substring pairs. Afterward, the distance between a block and one of its substrings is found by discretizing the distance with several threshold τ ($\tau = 0$ and $\tau = (1+\epsilon')^j$ for $0 \le j \le \log_{\epsilon'} n$) and then checking whether the distance is at most τ or much larger than τ. Both steps impose small errors. In the following, we assume a threshold τ is fixed, and we aim to find all pairs of (block, candidate substring)'s with an edit distance of at most τ. In the final solution, we try all values of $\tau = 0$ and $\tau = (1+\epsilon')^j$ for $0 \le j \le \log_{\epsilon'} n$ in parallel. For a given τ, we define a graph G_τ by placing a node for every block and candidate substrings of all blocks and connect two nodes if their edit distance is at most τ. Note that G_τ is not explicitly constructed at first, and we try to find most of its edges with limited resources. We sometimes use the phrase the edit distance between two nodes which means the edit distance between their corresponding strings. Let $0 < \alpha < 1$ be a parameter we fix later. We call a node high degree (or dense) if it is connected to more than n^α nodes and call it low degree (or sparse) otherwise.

Our algorithm for the case of large distances consists of four rounds. In the first round, we find the neighbors of all high degree nodes with high probability. If most of the blocks are high degree, then our job is done. In the second and third rounds, we solve the problem for cases where a significant number of low degree blocks are present. In the fourth round, we combine the solutions of individual blocks into a general solution using a dynamic program. In the following, we briefly explore these ideas.

In the first round, we sample each node with an independent probability of $\widetilde{O}_\epsilon(1/n^\alpha)$ and call them representatives. We then compute the edit distances between each representative and all other nodes. We show that each high degree node of G_τ has at least one representative as its neighbor with high probability and therefore, we can approximately find all of its neighbors using the triangle inequality. More precisely, let z be a representative, $N_\tau(z)$ be the set of all nodes with a distance of at most τ to z, and $N_{2\tau}(z)$ be the set of all nodes with a distance of at most 2τ to z. We put an edge for each pair of $N_\tau(z) \times N_{2\tau}(z)$ in our graph. For a node v with a representative z as one of its neighbors, $v \in N_\tau(z)$ and $N_\tau(v) \subseteq N_{2\tau}(z)$ hold. Therefore, for a node v which has a representative neighbor, the edges between v and all of its neighbors are added. Moreover, if a node v is high degree, we show it has at least one representative as its neighbors with high probability. Furthermore, using triangle inequality, we show that each added edge has a distance of at most 3τ. In Fig. 6 the Venn diagram of the these sets are shown and in Lemma 7 a formal proof is presented.

Therefore, at the end of the first round, all neighbors of high degree nodes for all thresholds are found. However, some false positive neighbors may also be present where each of them has a distance of at most 3τ.

The second and third rounds consider low degree nodes. In the second round, we sample some low degree blocks and find their distance to their candidate substrings. In the third round, we extend some pairs of low degree blocks and candidate substring with distances less than τ to a number of adjacent blocks. In the following, we briefly describe the second and third rounds.

The neighbors of low degree nodes cannot be found similar to the high degree nodes. However, low degree nodes have other useful features:

- We partition s into several regions or larger blocks. If in the optimal solution opt, only a few low degree blocks correspond to a region, we can ignore low

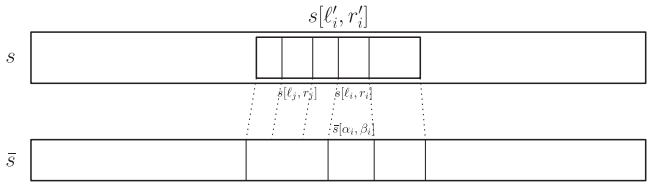

Fig. 7. If we know a block $s[\ell_i, r_i]$ and its corresponding substring $\overline{s}[\alpha_i, \beta_i]$ in opt, we can extend the substring. This extension gives us a candidate substring for each block $s[\ell_j, r_j]$ if it is contained in a larger block $s[\ell'_i, r'_i]$.

degree blocks of this region in our approximate solution since we are not going to lose much. Hence, we can only focus on regions with too many low degree blocks in opt.

- For such regions, we suppose we know a block $s[\ell_i, r_i]$ that transforms into $\overline{s}[\gamma, \kappa]$ in opt. Such an assumption is valid since there exist too many low degree blocks and we can hit at least one of them using random sampling. We can also assume we approximately know the edit distance between them by trying all values for τ. Moreover, since $s[\ell_i, r_i]$ is low degree, we can find $\overline{s}[\gamma, \kappa]$ by trying all candidate substrings of $s[\ell_i, r_i]$ and be sure that a small number of them can be (approximately) $\overline{s}[\gamma, \kappa]$. Otherwise, $s[\ell_i, r_i]$ is high degree.
- If we know a block $s[\ell_i, r_i]$ transforms into $\overline{s}[\gamma, \kappa]$ in opt, we can approximately guess that where the nearby blocks of $s[\ell_i, r_i]$ which are in the same region also transform into and create a pair of block/substring.

This technique is called the *extension of a low degree block* and it is shown in Fig. 7. More precisely, Suppose we know a block $s[\ell_i, r_i]$ transforms into $\overline{s}[\gamma, \kappa]$ in opt. We then consider larger blocks with size $n^{1-y'}$, assuming $y' < y$. Let $s[\ell'_i, r'_i]$ be a block of size $n^{1-y'}$ containing $s[\ell_i, r_i]$. We observe that $s[\ell'_i, r'_i]$ approximately transforms into $\overline{s}[\gamma - (\ell_i - \ell'_i), \kappa + (r'_i - r_i)]$ in opt. Moreover, we show that any block of size n^{1-y} such as $s[\ell_j, r_j]$ which is also contained in $s[\ell'_i, r'_i]$ approximately transforms into $\overline{s}[\gamma - (\ell_i - \ell_j), \kappa + (r_j - r_i)]$ in opt. To use such a block, we should have $\tau \approx \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\gamma, \kappa])$. To this end, we sample each low degree block with an independent probability of $\widetilde{O}_\epsilon(1/n^{(y-y')-(1-\delta)})$ and put them in a set L. We show that for each of the larger blocks which contain sufficiently many low degree normal sized blocks, L hits at least one of the normal sized blocks with the right τ with high probability. This claim is shown in Lemma 8. Note that the expected size of L is $n^y \cdot \widetilde{O}_\epsilon(1/n^{(y-y')-(1-\delta)}) = \widetilde{O}_\epsilon(n^{y'+(1-\delta)})$. For each block in L, we find its distance to all $\widetilde{O}_\epsilon(n^y)$ of its candidate substrings in the second round. The expected number of machines in the second round is equal to

$$|\mathsf{L}| \cdot \frac{O(n^\delta)}{n^{1-x}} = \widetilde{O}_\epsilon(n^{x+y'}).$$

The expected total running time of the second round is equal to

$$|\mathsf{L}| \cdot \widetilde{O}_\epsilon(n^y) \cdot \widetilde{O}_\epsilon(n^{2(1-y)}) = \widetilde{O}_\epsilon(n^{2-(y-y')}).$$

Moreover, the parallel running time of this phase is equal to

$$\widetilde{O}_\epsilon(n^{2-x-y}).$$

In the third round, we extend low degree blocks, for each of their candidate substrings with a distance of no more than τ. Note that low degree nodes have at most n^α such candidate substrings. For each of such pairs, we compute the edit distance of $n^{y-y'}$ pair of strings of size $\widetilde{O}_\epsilon(n^{1-y})$ in the third round. Therefore, the expected number of machines in the third round is at most

$$|L| \cdot n^\alpha \cdot n^{x-y'} = \widetilde{O}_\epsilon(n^{x+(1-\delta)+\alpha}).$$

The expected total running time of the third round is equal to

$$|L| \cdot n^\alpha \cdot n^{y-y'} \cdot \widetilde{O}_\epsilon(n^{2(1-y)}) = \widetilde{O}_\epsilon(n^{2-y+(1-\delta)+\alpha}).$$

Moreover, the parallel running time of this phase of our algorithm is equal to

$$\widetilde{O}_\epsilon(n^{2-x-y}).$$

In the fourth round of our algorithm, we use a DP algorithm (similar to the second round of small distances) to find a total solution. The final solution has an approximation factor of at most $3 + \epsilon$. In Section 5.3, we show by setting adequate parameters, the expected number of machines of our algorithm is $\widetilde{O}_\epsilon(n^{(9/5)x})$, each having a memory of size $O(n^{1-x})$. Moreover, the expected total time complexity of our algorithm is $\widetilde{O}_\epsilon(n^{2-2x/5})$.

By combining the two approaches for small distances and large distances, we achieve the desired massively parallel algorithm for edit distance.

Theorem 9 [restated]. Let $0 \leq x \leq 5/17$ and $\epsilon > 0$ be two arbitrary numbers. There exists a massively parallel algorithm that approximates the edit distance between two strings of length n within a factor of $3 + \epsilon$ in four rounds. The total computation of this algorithm is $\widetilde{O}_\epsilon(n^{2-\min(\frac{1-x}{6}, \frac{2x}{5})})$, and it uses $\widetilde{O}_\epsilon(n^{(9/5)x})$ machines each with a memory of $\widetilde{O}_\epsilon(n^{1-x})$. Moreover, the parallel running time of our algorithm is $\widetilde{O}_\epsilon(n^{2-\min(\frac{5+49x}{30}, \frac{11x}{5})})$.

4 Ulam Distance

The outline of our algorithm is presented in Section 3. In this section, we explain two phases of our algorithm in more details. Recall that in our algorithm, we divide string s into blocks of size $\mathcal{B} = n^{1-y}$ where $y = x$ in this Section. In the first phase, each machine receives a block of s and constructs a set of candidate substrings in $\overline{s}$ for the given block. Afterward, each machine computes the Ulam distance between their given block and the constructed candidate substrings. In the second phase of our algorithm, a single machine receives all of the information generated in the first phase and uses it to compute an approximately optimal solution via a dynamic program. In this section, we show that the approximation factor of our algorithm is $1 + \epsilon$ with high probability. In the analysis of our algorithm, we use a relatively smaller error threshold of $\epsilon' = \epsilon/2$.

4.1 Phase 1

In the first phase of our algorithm, we construct the candidate substrings for each block of s and compute the Ulam

distance between each block and its candidate substrings. The construction of the candidate substrings is done using two approaches. The first approach is suitable when $u_i = \mathsf{ulam}(s[l_i, r_i], \overline{s}[\alpha_i, \beta_i])$ is small and the second one is suitable when u_i is large.

If $u_i < \mathcal{B}/2$, we use the solution of the local version of Ulam distance (lulam) between $s[l_i, r_i]$ and the whole string of $\overline{s}$ to find an estimated location of α_i and β_i. The local Ulam distance between $s[l_i, r_i]$ and $\overline{s}$ is defined as the minimum Ulam distance between $s[l_i, r_i]$ and any substring of $\overline{s}$. A sequential algorithm for local Ulam distance is presented in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2021.3076534. Let $\overline{s}[\gamma, \kappa]$ be such a substring of $\overline{s}$ with the minimum Ulam distance to $s[l_i, r_i]$. We claim that $|\alpha_i - \gamma| \leq 2u_i$ and $|\beta_i - \kappa| \leq 2u_i$. Recall that we define a gap size $\mathcal{G}_i = \epsilon' u_i$ and consider substrings where their starting points and ending points indices are divisible by $\mathcal{G}_i$. The total additive error incurred by considering the gap for starting and ending points is at most $\sum_i 2\mathcal{G}_i \leq 2\epsilon' \cdot \mathsf{ulam}(s[l_i, r_i], \overline{s}[\alpha_i, \beta_i])$, which is negligible. Furthermore, since $|\alpha_i - \gamma| \leq 2u_i$ and $|\beta_i - \kappa| \leq 2u_i$, the total number of starting points and ending points we consider are at most $(4u_i + 1)/\mathcal{G}_i = O(1/\epsilon')$. This makes a total of $O(1/\epsilon'^2)$ candidate substrings for each block.

The next approach constructs candidate substrings for a block when $u_i \geq \mathcal{B}/2$. Let c_i be the number of unchanged characters in an optimal transformation of $s[l_i, r_i]$ into $\overline{s}[\alpha_i, \beta_i]$. If c_i is at most $\epsilon'\mathcal{B}/4$, we ignore the transformation, remove all characters of $s[l_i, r_i]$ and insert all characters of $\overline{s}[\alpha_i, \beta_i]$. This imposes an additive error of at most $2c_i \leq \epsilon'\mathcal{B}/2 \leq \epsilon' u_i$, which is also negligible. Hence, in the following, we assume that $c_i \geq \epsilon'\mathcal{B}/4$.

In this case, we use a randomized sampling method and choose each character of $s[l_i, r_i]$ with an independent probability of $\theta = (8/\epsilon'\mathcal{B})\log n$. This hitting set I has at least one of the c_i unchanged characters of the given block with a probability of at least

$$1 - (1-\theta)^{c_i} = 1 - (1 - ((8/\epsilon'\mathcal{B})\log n)^{\epsilon'\mathcal{B}/4} > 1 - e^{-2\log n} = 1 - 1/n^2.$$

Using the union bound, the overall probability of success for all machines is at least $1 - n^x/n^2 > 1 - 1/n$. The idea is to use the location of an unchanged character for locating α_i and β_i as follows. Let $s[p]$ be an unchanged character which is mapped to $\overline{s}[q]$. Moreover, let $\gamma = q - (p - \ell_i)$ and $\kappa = q + (r_i - p)$. We show that $|\alpha_i - \gamma| \leq u_i$ and $|\beta_i - \kappa| \leq u_i$. Similar to before, we only use the starting points and ending points whose indices are divisible by $\mathcal{G}_i$. The total additive error incurred by considering the gap for starting and ending points is at most $\sum_i 2\mathcal{G}_i = 2\epsilon' \cdot \mathsf{ulam}(s[l_i, r_i], \overline{s}[\alpha_i, \beta_i])$ likewise. Furthermore, the expected total number of starting points and ending points we investigate are each at most $O(\theta\mathcal{B}) \cdot (2u_i + 1)/\mathcal{G}_i = \widetilde{O}(1/\epsilon'^2)$. This makes an expected total of $O(1/\epsilon'^4)$ candidate substrings for each block.

Since we do not know u_i in advance, we try all values of $u_i = (1+\epsilon')^j$ and $u_i = 0$ as an estimated value. This imposes an extra $1/\epsilon'$ term in the time complexity. However, it does not affect the approximation factor of our algorithm. For a block $s[\ell_i, r_i]$, we feed the entire block and the locations of its characters in $\overline{s}$ to a machine. Since $\overline{s}$ has no repetitive characters, the input of each machine has size $O(\mathcal{B}) = O(n^{1-x})$. In each machine then, we try all $u_i = (1+\epsilon')^j$ for $0 \leq j < \log_{1+\epsilon'} n^{1-x}$. We also deal with $u_i = 0$ separately as a special case. For a fixed u_i, we identify a set of at most $\widetilde{O}(1/\epsilon'^4)$ candidate substrings as described above. Our algorithm then computes the Ulam distance between $s[\ell_i, r_i]$ and all of the candidate substrings and outputs a set of at most $\widetilde{O}(1/\epsilon'^5)$ tuples indicating the candidate substrings and their corresponding Ulam distances. Therefore, for each block, the running time of our algorithm is $\widetilde{O}(\mathcal{B}/\epsilon'^5)$, and the size of the output is $\widetilde{O}(1/\epsilon'^5)$. Hence, the total running time of the first phase is $n^x \cdot \widetilde{O}(\mathcal{B}/\epsilon'^5) = \widetilde{O}(n/\epsilon'^5)$ and the total output generated for the second phase is $\widetilde{O}(n^x/\epsilon'^5)$. A more detailed description of our algorithm is given in Algorithm 1.

Algorithm 1. Computing the Candidate Substrings for a Block $s[\ell_i, r_i]$

Data: $\ell_i, r_i, s[\ell_i, r_i], \overline{s}$.
Result: candidate substrings along with their Ulam distances from $s[\ell_i, r_i]$
1: $(\gamma, \kappa, d^*) \leftarrow \mathsf{lulam}(s[\ell_i, r_i], \overline{s})$;
2: **if** $d^* = 0$ **then**
3: **Output** $\langle [\ell_i, r_i], [\gamma, \kappa], 0\rangle$;
4: **for** $u_i = (1+\epsilon')^j$ *where* $j \in [0, \log_{1+\epsilon'} n]$ **do**
5: $\hat{u}_i = (1+\epsilon')u_i$;
6: $\mathcal{G}_i \leftarrow \max(\lfloor \epsilon' u_i \rfloor, 1)$;
7: **if** $u_i < \mathcal{B}/2$ **then**
8: **for** $sp = \max(1, \gamma - 2\hat{u}_i)$ *to* $\min(\gamma + 2\hat{u}_i, n)$ *in steps of* $sp \leftarrow sp + \mathcal{G}_i$ **do**
9: **for** $ep = \max(1, \kappa - 2\hat{u}_i)$ *to* $\min(\kappa + 2\hat{u}_i, n)$ *in steps of* $ep \leftarrow ep + \mathcal{G}_i$ **do**
10: **Output** $\langle [\ell_i, r_i], [sp, ep], \mathsf{ulam}(s[\ell_i, r_i], \overline{s}[sp, ep])\rangle$;
11: **else**
12: sample a hitting set I from $[\ell_i, r_i]$ with a probability of $(8/\epsilon'\mathcal{B})\log n$ for each element;
13: **for** $p \in \mathsf{I}$ **do**
14: $q \leftarrow$ the location of $s[p]$ in $\overline{s}$;
15: $\gamma \leftarrow q - (p - \ell_i)$;
16: $\kappa \leftarrow q + (r_i - p)$;
17: **for** $sp = \max(1, \gamma - \hat{u}_i)$ *to* $\min(\gamma + \hat{u}_i, n)$ *in steps of* $sp \leftarrow sp + \mathcal{G}_i$ **do**
18: **for** $ep = \max(1, \kappa - \hat{u}_i, sp)$ *to* $\min(\kappa + \hat{u}_i, n)$ *in steps of* $ep \leftarrow ep + \mathcal{G}_i$ **do**
19: **Output** $\langle [\ell_i, r_i], [sp, ep], \mathsf{ulam}(s[\ell_i, r_i], \overline{s}[sp, ep])\rangle$;

In the Lemmas 1 and 2, we show some properties for small and large u_i's, respectively. Then, in Lemma 3, we use the two previous lemmas to show an important property of Algorithm 1. The proofs of Lemmas 1, 2, and 3 are presented in Appendix B, available in the online supplemental material. In Section 4.2, we use this property to prove the approximation factor of our algorithm.

Lemma 1. *Let $s[\ell_i, r_i]$ be a block of s and $\overline{s}[\alpha_i, \beta_i]$ be its corresponding substring in an optimal transformation of s into $\overline{s}$. Moreover, let u_i be the Ulam distance between $s[\ell_i, r_i]$ and $\overline{s}[\alpha_i, \beta_i]$. If $u_i < \mathcal{B}/2$ and $\overline{s}[\gamma, \kappa]$ be the solution of local Ulam distance between $s[\ell_i, r_i]$ and $\overline{s}$, then two substrings $\overline{s}[\gamma, \kappa]$ and $\overline{s}[\alpha_i, \beta_i]$ intersect. In addition, $|\alpha_i - \gamma| \leq 2u_i$ and $|\beta_i - \kappa| \leq 2u_i$ hold.*

Lemma 2. *Let $s[\ell_i, r_i]$ be a block of s and $\overline{s}[\alpha_i, \beta_i]$ be its corresponding substring in an optimal transformation of s into $\overline{s}$. Moreover, let u_i be the Ulam distance between $s[\ell_i, r_i]$ and $\overline{s}[\alpha_i, \beta_i]$. If $u_i \geq \mathcal{B}/2$ and at least $\epsilon'\mathcal{B}/4$ unchanged character exist in the optimal transformation of $s[\ell_i, r_i]$ into $\overline{s}[\alpha_i, \beta_i]$ and $s[p] = \overline{s}[q]$ be one of these unchanged characters, then $|\alpha_i - \gamma| \leq u_i$ and $|\beta_i - \kappa| \leq u_i$ hold where $\gamma = q - (p - \ell_i)$ and $\kappa = q + (r_i - p)$.*

Lemma 3. *Let $s[\ell_i, r_i]$ be a block and $\overline{s}[\alpha_i, \beta_i]$ be its corresponding substring in an optimal transformation of s into $\overline{s}$. If the optimal transformation of $s[\ell_i, r_i]$ into $s[\alpha_i, \beta_i]$ has a size of less than $\mathcal{B}/2$ or has at least $\epsilon'\mathcal{B}/2$ unchanged characters, then Algorithm 1 outputs at least a candidate substring $\overline{s}[\alpha', \beta']$ for $s[\ell_i, r_i]$ with high probability such that*

1) $\alpha_i \leq \alpha'_i \leq \alpha_i + \epsilon' u_i$
2) $\beta_i - \epsilon' u_i \leq \beta'_i \leq \beta_i$

4.2 Phase 2

Recall that the output of the first phase of our algorithm is a set of $\widetilde{O}_\epsilon(n^x)$ tuples, $\langle [\ell_i, r_i], [\gamma, \kappa], d\rangle$, where $s[\ell_i, r_i]$ is the i'th block of s and $\overline{s}[\gamma, \kappa]$ is a corresponding candidate substring. Also, d is equal to the Ulam distance between $s[\ell_i, r_i]$ and $\overline{s}[\gamma, \kappa]$. In the second phase of our algorithm, we perform a dynamic program to compute a transformation from s into $\overline{s}$ based on the partial solutions gathered in the first phase. Notice that if we assume $x < 1/2$, the size of the output of the first phase, $\widetilde{O}_\epsilon(n^x)$, is relatively small and can be fed into the memory of a single machine. This machine is responsible for the entire computation of the second phase.

Algorithm 2. Computing an Approximate Solution for $\mathsf{ulam}(s, \overline{s})$ Based on the Tuples

Data: $T[1], T[2], \ldots$.
Result: an approximate value of Ulam distance between s and $\overline{s}$

$m \leftarrow$ the number of tuples;
$D \leftarrow$ an array of size m initially containing ∞ in all cells;
for $a \in [1, m], T[a] = \langle [\ell_i, r_i], [\gamma, \kappa], d\rangle$ **do**
  $D[a] \leftarrow \max\{\ell_i - 1, \gamma - 1\} + d$;
  for $b \in [1, a-1], T[b] = \langle [\ell'_i, r'_i], [\gamma', \kappa'], d'\rangle$ **do**
    if $r'_i < \ell_i$ *and* $\kappa' < \gamma$ **then**
      $D[a] \leftarrow \min\{D[a], D[b] + \max\{\ell_i - r'_i - 1, \gamma - \kappa' - 1\} + d\}$;
answer $\leftarrow \infty$;
for $a \in [1, m], T[a] = \langle [\ell_i, r_i], [\gamma, \kappa], d\rangle$ **do**
  answer $\leftarrow \min\{$answer$, D[a] + \max\{n - r_i, n - \kappa\}\}$;
return answer;

We assume the tuples are stored in array T and are sorted in increasing order of ℓ_i where ties are broken arbitrarily. In our dynamic program, we create an array D with the same length as T. Let the a'th tuple be equal to $\langle [\ell_i, r_i], [\gamma, \kappa], d\rangle$. We compute D such that each $D[a]$ approximates $\mathsf{ulam}(s[1, r_i], \overline{s}[1, \kappa])$. Let P_a be the set of indices of tuples $T[b] = \langle [\ell'_i, r'_i], [\gamma', \kappa'], e'\rangle$ such that $r'_i < \ell_i$ and $\kappa' < \gamma$. Then, the update rule of D is as follows.

$$D[a] := \min\Big\{\max\{\ell_i - 1, \gamma - 1\} + d, \min_{b \in P_a}\{D[b] + \max\{\ell_i - r'_i - 1, \gamma - \kappa' - 1\} + d\}\Big\}.$$

The term $d + \max\{\ell_i - 1, \gamma - 1\}$ correspond to the situation where characters of $s[1, \min\{\ell_i - 1, \gamma - 1\}]$ are substituted with $\overline{s}[1, \min\{\ell_i - 1, \gamma - 1\}]$, the remaining characters of $s[1, \ell_i - 1]$ are removed, and the remaining characters of $\overline{s}[1, \gamma - 1]$ are inserted. In the other term, $T[b]$ is the first tuple before $T[a]$ in the solution where $D[b]$ is the estimated value for $\mathsf{ulam}(s[1, r'_i], \overline{s}[1, \kappa'])$, the additional cost d corresponds to the cost of transforming $s[\ell_i, r_i]$ into $\overline{s}[\gamma, \kappa]$ and the cost $\max\{\ell_i - r'_i - 1, \gamma - \kappa' - 1\}$ corresponds to substituting/removing/adding the characters between the two tuples. This gives us an $\widetilde{O}_\epsilon(n^{2x})$ time algorithm with memory $\widetilde{O}_\epsilon(n^x)$ that runs on a single machine.

In Theorem 4, we show that the combination of Algorithms 1 and 2, approximates the Ulam distance of s and $\overline{s}$ within a factor of $1 + \epsilon$ with high probability.

Theorem 4. *For an arbitrarily small $\epsilon > 0$, there exists a massively parallel algorithm with $\widetilde{O}_\epsilon(n^x)$ machines for arbitrary $0 < x < 1/2$, each with a memory of $\widetilde{O}_\epsilon(n^{1-x})$ that approximates the Ulam distance of two strings of length n within a factor of $1 + \epsilon$ with high probability in two rounds. The expected total computation of this algorithm is $\widetilde{O}_\epsilon(n)$.*

The proof of Theorem 4 is presented in Appendix B, available in the online supplemental material.

5 Edit Distance

The outline of our algorithm for edit distance is presented in Section 3. In this section, we describe both cases of our algorithm in more details. Recall that our algorithm has two different approaches for relatively small and relatively large distances. Recall that we divide s into n^y blocks of size $\mathcal{B} = n^{1-y}$. We also construct a set of candidate substrings for each block. We define $\epsilon' = \epsilon/22$ for the simplicity of our analysis. We also assume that our solution is roughly equal to n^δ. Recall that we define the gap size $\mathcal{G} = \max\{\lfloor n^{\delta - x}\epsilon'\rfloor, 1\}$ and define the starting points of candidate substrings to be indices no more that n^δ apart from ℓ_i and are divisible by $\mathcal{G}$.

5.1 Small Distances ($\mathbf{n^\delta \leq n^{1-x/5}}$)

For small distances recall that we fix $y = x$. The main task in the first phase is to compute the edit distances between each block and its candidate substrings. In the second phase, we use the gathered information in the first phase to construct our final solution. The approximation factor of our algorithm, in this case, is $3 + \epsilon$ since we use a variant of the approximation algorithm of [12] to find the edit distance between any block and its candidate substrings instead of using the naïve DP algorithm.

5.1.1 Phase 1 for Small Distances

In the first phase of our algorithm for small distances, we construct candidate substrings for each block $s[\ell_i, r_i]$ and assign one block and several of its candidate substrings to each machine. The memory of each machine is $\widetilde{O}_\epsilon(n^{1-x})$; hence, $\widetilde{O}_\epsilon(n^\delta)/n^{1-x}$ machines are sufficient for each block since we partition candidate substrings by their starting points. Therefore, the total number of machines used in this

phase is $\widetilde{O}_\epsilon(n^{2x-(1-\delta)})$. In the following, we explain how we partition candidate substrings of a block.

For a fixed block and a fixed starting point, our algorithm considers several ending points. Note that, we only consider candidate substrings with a length of at most $(1/\epsilon')\mathcal{B}$. Therefore, the size of the feed of each machine is at most $\widetilde{O}_\epsilon(n^{1-x})$. More precisely, a machine is responsible to compute the edit distance between a block $s[\ell_i, r_i]$ and several starting points $\gamma_1, \gamma_2, \ldots, \gamma_\eta$, where $\eta = O(n^{1-x}/\mathcal{G})$. The feed given to this machine is $s[\ell_i, r_i]$ and $\overline{s}[\gamma_1, \gamma_\eta + (1/\epsilon')\mathcal{B}]$. Afterward, we assign a set of endpoints $\kappa_{j,1}, \kappa_{j,2}, \ldots, \kappa_{j,\eta'}$, where $\eta' = O(\log_{1+\epsilon'}((1/\epsilon')\mathcal{B}))$, for a starting point γ_j. We form $\kappa_{j,h}$'s as $\gamma + \mathcal{B} \pm (1+\epsilon')^a$ for $0 \le a \le \log_{1+\epsilon'}(\mathcal{B}/\epsilon')$ in addition to $\gamma + \mathcal{B}$. We then consider $\overline{s}[\gamma_j, \kappa_{j,h}]$'s as candidate substrings. The machine then computes the edit distance between the given block ($s[\ell_i, r_i]$) and the candidate substrings ($\overline{s}[\gamma_j, \kappa_{j,h}]$'s) and outputs a set of $\eta \cdot \eta' = \widetilde{O}_\epsilon(n^{1-\delta})$ tuples each consisting of a block, a candidate substring, and their edit distance. The running time of each machine is then $\eta \cdot \eta' \cdot \widetilde{O}_\epsilon(n^{(2-1/6)(1-x)}) = \widetilde{O}_\epsilon(n^{2-2x+(1-\delta)-(1-x)/6})$. Hence, the total running time of the first phase is $\widetilde{O}_\epsilon(n^{2x-(1-\delta)}) \cdot \widetilde{O}_\epsilon(n^{2-2x+(1-\delta)-(1-x)/6}) = \widetilde{O}_\epsilon(n^{2-(1-x)/6})$. The size of the ouput of a machine is $\eta \cdot \eta' = \widetilde{O}_\epsilon(n^{1-\delta})$; hence, the size of the total output produced in the first phase is $\widetilde{O}_\epsilon(n^{2x-(1-\delta)}) \cdot \widetilde{O}_\epsilon(n^{1-\delta}) = \widetilde{O}_\epsilon(n^{2x})$. The pseudo-code of the first phase is given in Algorithm 3.

Algorithm 3. Computing the Edit Distance Between a Block $s[\ell_i, r_i]$ and its Candidate Substrings With Starting Points $\gamma_1, \gamma_2, \ldots, \gamma_\eta$

Data: $\ell_i, r_i, \gamma_1, \ldots, \gamma_\eta, s[\ell_i, r_i], \overline{s}[\gamma_1, \gamma_\eta + \mathcal{B} \cdot 1/\epsilon']$.
Result: candidate substrings with specified starting points along with their edit distances from $s[\ell_i, r_i]$

for $j \in [1, \eta]$ **do**
  Output $\langle [\ell_i, r_i], [\gamma_j, \gamma_j + \mathcal{B}], \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\gamma_j, \gamma_j + \mathcal{B}]) \rangle$;
  for $a \in [0, \lfloor \log_{1+\epsilon'} \min\{\mathcal{B}/\epsilon', n^\delta\} \rfloor]$ **do**
    Output $\langle [\ell_i, r_i], [\gamma_j, \gamma_j + \mathcal{B} - \lfloor (1+\epsilon')^a \rfloor], \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\gamma_j, \gamma_j + \mathcal{B} - \lfloor (1+\epsilon')^a \rfloor]) \rangle$;
    Output $\langle [\ell_i, r_i], [\gamma_j, \gamma_j + \mathcal{B} + \lfloor (1+\epsilon')^a \rfloor], \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\gamma_j, \gamma_j + \mathcal{B} + \lfloor (1+\epsilon')^a \rfloor]) \rangle$;

In Lemma 5, we prove an important feature of Algorithm 3. In Section 5.1.2, we use this feature of Algorithm 3 to prove the correctness of our algorithm for small distances.

Lemma 5. *Let $s[\ell_i, r_i]$ be a block of s and $\overline{s}[\alpha_i, \beta_i]$ be its corresponding substring in an optimal transformation of s into $\overline{s}$. Moreover, assume $\alpha_i + \mathcal{G} + \epsilon'\mathcal{B} < \beta_i \le \alpha_i + \mathcal{B} \cdot 1/\epsilon'$ holds. Therefore, Algorithm 3 for at least one machine outputs a candidate substring $\overline{s}[\alpha', \beta']$ such that both of these conditions hold:*

- $\alpha_i \le \alpha'_i \le \alpha_i + \epsilon' n^{\delta - x}$,
- $\beta_i - \epsilon' n^{\delta-x} - \epsilon' \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\alpha_i, \beta_i]) \le \beta'_i \le \beta_i$.

The proof of Lemma 5 is presented in Appendix B, available in the online supplemental material.

5.1.2 Phase 2 for Small Distances

The second phase of our algorithm receives a set of $\widetilde{O}_\epsilon(n^{2x})$ tuples in the form of $\langle [\ell_i, r_i], [\gamma, \kappa], e \rangle$. In this notation, a tuple contains a block of s, one of its corresponding candidate substrings generated in the first phase, and their edit distance. Then, a dynamic program selects a subset of these tuples to form a total transformation of s into $\overline{s}$. We run the dynamic program in a single machine. This is possible due to the size of the output of Phase 1 and assuming $0 < x < 5/17$. We denote the tuples by $T[1], T[2], \ldots$ We assume that the tuples are sorted in increasing order of ℓ_i where ties are broken arbitrarily. In the dynamic program, for any tuple $a = \langle [\ell_i, r_i], [\gamma, \kappa], e \rangle$, we compute $D[a]$ as an approximation of $\mathsf{ed}(s[1, r_i], \overline{s}[1, \kappa])$ using a and tuples before a in T. The update rule of this dynamic program is

$$D[a] = \min\{e + (\ell_i - 1) + (\gamma - 1), \min_{\ell_i > r'_i, \gamma > \kappa', b < a, \langle [\ell'_i, r'_i], [\gamma', \kappa'], e' \rangle = T[b]} \{D[b] + e + (\ell_i - r'_i - 1) + (\gamma - \kappa' - 1)\}\}.$$

In this notation, b is the last tuple used before a. Cost $\ell_i - r'_i - 1$ corresponds to removing the characters between two tuples in s, cost $\gamma - \kappa' - 1$ corresponds to adding the characters between two tuples in $\overline{s}$, and cost e corresponds to using the transformation of the tuple a. The time complexity of this algorithm is $\widetilde{O}_\epsilon(n^{4x})$ and its memory is $\widetilde{O}_\epsilon(n^{2x})$.

Algorithm 4. Computing an Approximation of $\mathsf{ed}(s, \overline{s})$ Based on the Output Tuples of the First Round

Data: $T[1], T[2], \ldots$
Result: an approximate value of $\mathsf{ed}(s, \overline{s})$

$m \leftarrow$ the size of T;
$D \leftarrow$ an array of size m where all of its entries initialized to $+\infty$;
for $a \in [1, m], \langle [\ell_i, r_i], [\gamma, \kappa], e \rangle = T[a]$ **do**
  $D[a] \leftarrow (\ell_i - 1) + (\gamma - 1) + e$;
  for $b \in [1, a-1], \langle [\ell'_i, r'_i], [\gamma', \kappa'], e' \rangle = T[b]$ **do**
    if $\ell_i > r'_i$ *and* $\gamma > \kappa'$ **then**
      $D[a] \leftarrow \min\{D[a], D[b] + e + (\ell_i - r'_i - 1) + (\gamma - \kappa' - 1)\}$;
answer $\leftarrow \infty$;
for $a \in [1, m], \langle [\ell_i, r_i], [\gamma, \kappa], e \rangle = T[a]$ **do**
  answer $\leftarrow \min\{\mathsf{answer}, D[a] + (n - r_i) + (n - \kappa)\}$;
return answer;

In Lemma 6, we show that our overall algorithm consisting of Algorithms 3 and 4, approximates the edit distance of s and $\overline{s}$ within a factor of $3 + \epsilon$.

Lemma 6. *For an arbitrarily small $\epsilon > 0$, there exists a massively parallel algorithm that approximates the edit distance of two strings of length n if their distance is no more than n^δ within a factor of $3 + \epsilon$ in two rounds. The total computation of this algorithm is $\widetilde{O}_\epsilon(n^{2-(1-x)/6})$ and its parallel running time is $\widetilde{O}_\epsilon(n^{(1-\delta)+(2-1/6)(1-x)})$. Moreover, the algorithm uses $\widetilde{O}$ on$(n^{2x-(1-\delta)})$ machines each with a memory of $\widetilde{O}_\epsilon(n^{1-x})$.*

The proof of Lemma 6 is presented in Appendix B, available in the online supplemental material.

5.2 Large Distances ($\mathbf{n^\delta > n^{1-x/5}}$)

Recall that for large distances, we use four phases as follows. The goal of the algorithm in the first round is to find many edges of G_τ with the help of the triangle inequality. We run our algorithm for all thresholds $\tau = (1+\epsilon')^h$ for $0 \le h \le \log_{1+\epsilon'}(2n)$ and $\tau = 0$ in parallel. In the following, we assume a threshold τ is fixed. The algorithm, in this case, consists of four phases which are described as follows.

5.2.1 Phase 1: High Degree Case

In this phase, we use a random sampling method to select a set of representative nodes R. Recall that we chose each node of G_τ with an independent probability of $2\log n/n^\alpha$ where $0 < \alpha < 1$ is a parameter we fix later. We call a node *high degree* (dense) if its degree is at least n^α, and *low degree* (sparse), otherwise. Afterward, for each $z \in \mathsf{R}$, we find its edit distance to all other nodes of G_τ. We then use this information to generate several edges of G_τ as follows. We claim that for any edge of G_τ whose one of its nodes is a high degree node we generate it with high probability. Furthermore, some edges with a distance of at most 3τ may be generated in the output of the first phase.

Algorithm 5. Computing the Edit Distances Between Repsentative Nodes $z_1, z_2, \ldots$ and Nodes $v_1, v_2, \ldots$

Data: A subset of repsentative nodes $z_1, z_2, \ldots, z_\mu$, a set of nodes $v_1, v_2, \ldots, v_\mu$, and a number τ.
Result: edit distances between given repsentative and given nodes

for $z \in \{z_1, z_2, \ldots, z_\mu\}$ **do**
  for $v \in \{v_1, v_2, \ldots, v_\mu\}$ **do**
    compute the edit distance between z and v;
    if *v is a candidate substring node and* $\mathsf{ed}(z, v) \leq 2\tau$ **then**
      Output $\langle "cs", v, z, \tau\rangle$;
    if *z is a candidate substring node* **then**
      Output $\langle "cs", z, z, \tau\rangle$;
    if *v is a block node and* $\mathsf{ed}(z, v) \leq \tau$ **then**
      Output $\langle "b", v, z, \tau\rangle$;
    if *z is a block node* **then**
      Output $\langle "b", z, z, \tau\rangle$;

Each node of G_τ either corresponds to a block or a candidate substrings. The number of block nodes is n^y and the number of candidate substring nodes is computed as follows. The starting points of candidate substrings are divisible by $\mathcal{G}' = \max\{\lfloor n^{\delta-y}\epsilon' \rfloor, 1\}$. Therefore, the total number of starting point is at most $O(n/\mathcal{G}') = \widetilde{O}_\epsilon(n^{(1-\delta)+y})$ and the total number of nodes of G_τ is $\widetilde{O}_\epsilon(n^{(1-\delta)+y})$ Hence, the expected size of $|\mathsf{R}|$ is equal to $(2\log n/n^\alpha) \cdot \widetilde{O}_\epsilon(n^{(1-\delta)+y}) = \widetilde{O}_\epsilon(n^{(1-\delta)+y-\alpha})$. For each representative, we compute its edit distance to all other nodes. Hence, the expected number of pairs of nodes which we compute their edit distance is equal to

$$\widetilde{O}_\epsilon(n^{(1-\delta)+y-\alpha}) \cdot \widetilde{O}_\epsilon(n^{(1-\delta)+y}) = \widetilde{O}_\epsilon(n^{2(1-\delta)+2y-\alpha}).$$

By using the naïve DP algorithm for computing the edit distance between two nodes, the expected total running time of this phase is equal to

$$\widetilde{O}_\epsilon(n^{2(1-\delta)+2y-\alpha}) \cdot \widetilde{O}_\epsilon((n^{1-y})^2) = \widetilde{O}_\epsilon(n^{2+2(1-\delta)-\alpha}).$$

Since each machine has a memory of size $\widetilde{O}_\epsilon(n^{1-x})$ and the string size of each node is $\widetilde{O}_\epsilon(n^{1-y})$, the machine can keep n^{y-x} representatives and n^{y-x} extra nodes. Therefore, the expected number of required machines is

$$\widetilde{O}_\epsilon(n^{(1-\delta)+y-\alpha}) \cdot \frac{1}{n^{y-x}} \cdot \widetilde{O}_\epsilon(n^{(1-\delta)+y}) \cdot \frac{1}{n^{y-x}} = \widetilde{O}_\epsilon(n^{2x+2(1-\delta)-\alpha}).$$

In the following, we explain how we find the neighbors of high degree nodes using the edit distance between representatives and all nodes. For a node z, we call the set of nodes with an edit distance of at most τ to z (including z itself) as $N_\tau(z)$. Since we computed the distance of each representative to all other nodes, this information is available to us for arbitrary τ's. For each $z \in \mathsf{R}$, we connect all nodes of $N_\tau(z)$ to $N_{2\tau}(z)$. We claim that for each node with a degree of at least n^α, we found all of its neighbors with high probability. We may found some false positive neighbors; however, any additional generated edge has an edit distance of at most 3τ. The pseudocode of this phase is shown in Algorithm 5. We prove our claims in Lemma 7.

Lemma 7. *Let R be a random sampling of nodes of G_τ where each node is chosen with an independent probability of $p = 2\log n/n^\alpha$. Let v be a block node of G_τ with a degree of at least n^α. For each candidate substring node $u \in N_\tau(v)$, there exist a representative $z \in \mathsf{R}$ with high probability such that $v \in N_\tau(z)$ and $u \in N_{2\tau}(z)$. Moreover, if z is an arbitrary representative, v an arbitrary block node and u and candidate substring node, if $v \in N_\tau(z)$ and $u \in N_{2\tau}(z)$ then $\mathsf{ed}(v, u) \leq 3\tau$.*

The proof of Lemma 7 is presented in Appendix B, available in the online supplemental material.

5.2.2 Phases 2 and 3: Low Degree Nodes

In the second and third phases, we treat the low degree block nodes as follows. Since a node being high degree or low degree in G_τ highly depends on τ we define overall high degree and overall low degree nodes independent of τ as follows.

All blocks in G_τ for $\tau = n$ are high degree. As τ decreases, some of its edges get deleted and each nodes eventually becomes low degree (sparse). For a block $s[\ell_i, r_i]$, we call it *overall high degree* if it is high degree for $\tau = \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\alpha_i, \beta_i])$. We call other blocks as *overall low degree*. We then consider larger blocks of size $n^{1-y'}$. A larger block belong to one the two type:

i. Larger blocks that contain at most $\epsilon' n^{(y-y')-(1-\delta)}$ overall low degree blocks.
ii. Larger blocks that contain more than $\epsilon' n^{(y-y')-(1-\delta)}$ overall low degree blocks.

The overall low degree blocks of all larger blocks of type (i) can be ignored with a multiplicative error of no more than $1 + \epsilon'$. Therefore, from this point we only concentrate on larger blocks of type (ii). We claim that by sampling each low degree block with a probability of $p = 3(1/\epsilon'^2)\log^2 n/\ n^{(y-y')-(1-\delta)}$, we catch at least one overall low degree block for each larger block of type (ii) with high probability.

To compute the edit distance between the chosen block and all of their candidate substrings, we need an expected number of machines of at most

$$p \cdot n^y \cdot \frac{\widetilde{O}_\epsilon(n^\delta)}{n^{1-x}} = \widetilde{O}_\epsilon(n^{x+y'}).$$

Moreover, the expected total running time of computing these distances is at most

$$p \cdot n^{y} \cdot \widetilde{O}_{\epsilon}(n^{y}) \cdot \widetilde{O}_{\epsilon}(n^{2(1-y)}) = \widetilde{O}_{\epsilon}(n^{2-(y-y')+(1-\delta)}).$$

Suppose we find an overall low degree block $s[\ell_i, r_i]$ in G_τ where $\tau \leq \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\alpha_i, \beta_i]) \leq (1+\epsilon')\tau$. Also recall that, one of the candidate substrings such as $\overline{s}[\alpha'_i, \beta'_i]$ is guaranteed to be approximately optimal. If all of blocks in the same larger block as $s[\ell_i, r_i]$, such as $s[\ell_j, r_j]$ forced to be transformed it into $\overline{s}[\alpha'_i + (\ell_j - \ell_i), \beta'_i + (r_j - r_i)]$ the imposed multiplicative error is bounded by $2 + 3\epsilon'$. In the following, we use this technique to extend low degree block.

In the third round, every pair of low degree block and its neighbors is extended to $n^{y-y'}$ nearby blocks, which are in the same larger block. Note that since a low degree node has at most n^{α} neighbors, the total number of pairs is limited. More precisely, the expected number of generated pairs is at most

$$p \cdot n^{y} \cdot n^{\alpha} \cdot n^{y-y'} = \widetilde{O}_{\epsilon}(n^{y+\alpha+(1-\delta)}).$$

Since n^{y-x} pairs can be process in a single machine, the expected number of machines needed in this round is equal to

$$\widetilde{O}_{\epsilon}(n^{x+\alpha+(1-\delta)}).$$

Moreover, the expected total running time of the third round is

$$\widetilde{O}_{\epsilon}(n^{y+\alpha+(1-\delta)}) \cdot \widetilde{O}_{\epsilon}(n^{2(1-y)}) = \widetilde{O}_{\epsilon}(n^{2-y+\alpha+(1-\delta)}).$$

Algorithm 6. Extending high degree and low degree blocks

Data: The blocks $s[\ell_i, r_i]$ for, a subset of candidate substrings for these blocks, all tuples of type "b" ($T_b[1], T_b[2], \ldots$) for these blocks and all tuples of type "cs" ($T_{cs}[1], T_{cs}[2], \ldots$) for these candidate substrings.
Result: edit distances between high degree blocks and low degree extension for low degree blocks

1: **for** $\tau \in \{0, (1+\epsilon')^a$ *for* $0 \leq a \leq \log_{\epsilon'} n\}$ **do**
2: **for** *all given blocks* $s[\ell_i, r_i]$ **do**
3: **if** *there exists a tuple* $T_b[a]$ *of type "b" for the given block and* τ **then**
4: let z be the third value of $T_b[a]$;
5: **for** *every* $T_{cs}[b]$ *containing* z *and* τ **do**
6: let u be the third value of $T_{cs}[b]$;
7: **Output** $\langle$"high degree", $[\ell_i, r_i], u, \tau\rangle$;
8: **else**
9: **if** *a random variable with a common seed between machines is less than* $p = 3(1/\epsilon'^2)\log^2 n / n^{(y-y')-(1-\delta)}$ **then**
10: compute the edit distance $s[\ell_i, r_i]$ and all of related candidate substrings in the input;
11: **for** *any candidate interval* $\overline{s}[\gamma, \kappa]$ *where* $\mathsf{ed}(s[\ell_i, r_i], \overline{s}[\gamma, \kappa]) \leq \tau$ **do**
12: **for** *any block* $s[\ell_j, r_j]$ *which is the same larger block as* $s[\ell_i, r_i]$ **do**
13: **Output** $\langle$"extend", $[\ell_j, r_j], [\gamma + (\ell_j - \ell_i), \kappa + (r_j - r_i)], \tau\rangle$;

Algorithm 7. Computing the Edit Distances Between a Block and a Candidate Substring Which Came From a Low Degree Extension

Data: a block $s[\ell_i, r_i]$ and a candidate substring $\overline{s}[\gamma, \kappa]$.
Result: the edit distance between $s[\ell_i, r_i]$ and $\overline{s}[\gamma, \kappa]$

1: Use the naïve DP algorithm to compute $\mathsf{ed}(s[\ell_i, r_i], \overline{s}[\gamma, \kappa])$;
2: **Output** $\langle$"low degree ext", $[\ell_i, r_i], [\gamma, \kappa], \mathsf{ed}(s[\ell_i, r_i], \overline{s}[\gamma, \kappa])\rangle$;

5.2.3 Phase 4: Computing the Overall Transformation

The DP algorithm of this phase is similar to that of small distances. In this round, we receive at most $\widetilde{O}_{\epsilon}(n^{2y})$ tuples in the form of $\langle [\ell_i, r_i], [\gamma, \kappa], e\rangle$. In this notation, the tuple corresponds to a block, its corresponding candidate substring and their edit distance. Then, a dynamic program selects a subset of these tuples to form a total transformation of s into $\overline{s}$. We run the dynamic program in a single machine. We again denote the tuples by $T[1], T[2], \ldots$ and assume that the tuples are sorted in increasing order of ℓ_i where ties are broken arbitrarily. The time complexity of this algorithm if trivially implemented is $\widetilde{O}_{\epsilon}(n^{4y})$. However, by suitable data structure the time complexity is improved to $\widetilde{O}_{\epsilon}(n^{2y})$. Moreover, the memory complexity of this algorithm is $\widetilde{O}_{\epsilon}(n^{2y})$. The rest of the algorithm is the same as the second round of small distances. A minor difference is that if the candidate substring of two tuples intersects, we may choose both of them, but we add the cost of removing the common part.

In Lemma 8, we show that our overall algorithm consisting of Algorithms 5, 6, 7, and a dynamic program similar to Algorithm 4, approximates the edit distance of s and $\overline{s}$ within a factor of $3+\epsilon$. The proof of Lemma 8 is presented in Appendix B, available in the online supplemental material.

Lemma 8. *For an arbitrarily small $\epsilon > 0$, there exists a massively parallel algorithm that approximates the edit distance of two strings of length n if their distance is at most n^{δ} within a factor of $3+\epsilon$ in four rounds. This algorithm uses $\widetilde{O}_{\epsilon}\left(n^{\max\{2x+2(1-\delta)-\alpha, x+y', x+(1-\delta)+\alpha\}}\right)$ machines each with a memory of $\widetilde{O}_{\epsilon}(n^{1-x})$. The total computation of this algorithm is $\widetilde{O}_{\epsilon}\left(n^{2-x/4}\right)$.*

5.3 Overall Algorithm

By substituting $\delta = 1 - x/5$, $\alpha = (3/5)x$, $y = (6/5)x$, $y' = (4/5)x$, and $x \leq 5/17$, and using Lemmas 6 and 8 we can conclude the following theorem.

Theorem 9. *Let $0 \leq x \leq 5/17$ and $\epsilon > 0$ be two arbitrary numbers. There exists a massively parallel algorithm that approximates the edit distance between two strings of length n within a factor of $3+\epsilon$ in four rounds. The total computation of this algorithm is $\widetilde{O}_{\epsilon}(n^{2-\min(\frac{1-x}{6}, \frac{2x}{5})})$, and it uses $\widetilde{O}_{\epsilon}(n^{(9/5)x})$ machines each with a memory of $\widetilde{O}_{\epsilon}(n^{1-x})$. Moreover, the parallel running time of our algorithm is $\widetilde{O}_{\epsilon}(n^{2-\min(\frac{5+49x}{30}, \frac{11x}{5})})$.*

ACKNOWLEDGMENTS

A preliminary version of this article was presented at SPAA 2019 [1].

REFERENCES

[1] M. Boroujeni and S. Seddighin, "Improved MPC algorithms for edit distance and Ulam distance," in *Proc. 31st ACM Symp. Parallelism Algorithms Archit.*, 2019, pp. 31–40.

[2] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *J. Comput. Syst. Sci.*, vol. 20, no. 1, pp. 18–31, 1980.

[3] A. Backurs and P. Indyk, "Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)," in *Proc. 47th Annu. ACM Symp. Theory Comput.*, 2015, pp. 51–58.

[4] A. Andoni, R. Krauthgamer, and K. Onak, "Polylogarithmic approximation for edit distance and the asymmetric query complexity," in *Proc. IEEE 51st Annu. Symp. Found. Comput. Sci.*, 2010, pp. 377–386.

[5] Z. Bar-Yossef, T. Jayram, R. Krauthgamer, and R. Kumar, "Approximating edit distance efficiently," in *Proc. 45th Annu. IEEE Symp. Found. Comput. Sci.*, 2004, pp. 550–559.

[6] T. Batu, F. Ergun, and C. Sahinalp, "Oblivious string embeddings and edit distance approximations," in *Proc. 17th Annu. ACM-SIAM Symp. Discrete Algorithm*, 2006, pp. 792–801.

[7] B. Haeupler, A. Rubinstein, and A. Shahrasbi, "Near-linear time insertion-deletion codes and $(1+\varepsilon)$-approximating edit distance via indexing," in *Proc. 51st Annu. ACM SIGACT Symp. Theory Comput.*, 2019, pp. 697–708.

[8] P. Indyk, "Algorithmic applications of low-distortion geometric embeddings," in *Proc. 42nd IEEE Symp. Found. Comput. Sci.*, 2001, pp. 10–33.

[9] G. M. Landau, E. W. Myers, and J. P. Schmidt, "Incremental string comparison," *SIAM J. Comput.*, vol. 27, no. 2, pp. 557–582, 1998.

[10] A. Andoni and K. Onak, "Approximating edit distance in near-linear time," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, 2009, pp. 199–204.

[11] M. Boroujeni, S. Ehsani, M. Ghodsi, M. HajiAghayi, and S. Seddighin, "Approximating edit distance in truly subquadratic time: Quantum and MapReduce," in *Proc. 29th Annu. ACM-SIAM Symp. Discrete Algorithm*, 2018, pp. 1170–1189.

[12] D. Chakraborty, D. Das, E. Goldenberg, M. Koucky, and M. Saks, "Approximating edit distance within constant factor in truly subquadratic time," in *Proc. IEEE 59th Annu. Symp. Found. Comput. Sci.*, 2018, pp. 979–990.

[13] M. Koucký and M. Saks, "Constant factor approximations to edit distance on far input pairs in nearly linear time," in *Proc. 52nd Annu. ACM SIGACT Symp. Theory Comput.*, 2020, pp. 699–712.

[14] J. Brakensiek and A. Rubinstein, "Constant-factor approximation of near-linear edit distance in near-linear time," in *Proc. 52nd Annu. ACM SIGACT Symp. Theory Comput.*, 2020, pp. 685–698.

[15] A. Andoni and N. S. Nosatzki, "Edit distance in near-linear time: It's a constant factor," *FOCS*, pp. 990–1001, 2020.

[16] E. Goldenberg, A. Rubinstein, and B. Saha, "Does preprocessing help in fast sequence comparisons?," in *Proc. 52nd Annu. ACM SIGACT Symp. Theory Comput.*, 2020, pp. 657–670.

[17] T. Naumovitz, M. Saks, and C. Seshadhri, "Accurate and nearly optimal sublinear approximations to Ulam distance," in *Proc. 28th Annu. ACM-SIAM Symp. Discrete Algorithm*, 2017, pp. 2012–2031.

[18] D. Aldous and P. Diaconis, "Longest increasing subsequences: From patience sorting to the Baik-Deift-Johansson theorem," *Bull. Amer. Math. Soc.*, vol. 36, no. 4, pp. 413–432, 1999.

[19] D. E. Critchlow, *Ulam's Metric*. Hoboken, NJ, USA: Wiley, 2006.

[20] M. Hajiaghayi, S. Seddighin, and X. Sun, "Massively parallel approximation algorithms for edit distance and longest common subsequence," in *Proc. 30th Annu. ACM-SIAM Symp. Discrete Algorithm*, 2019, pp. 1654–1672.

[21] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Operating Syst. Des. Implementation*, 2004, pp. 10–10.

[22] T. White, *Hadoop: The Definitive Guide*, 1st ed. Newton, MA, USA: O'Reilly Media, Inc., 2009.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.

[24] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for MapReduce," in *Proc. 21st Annu. ACM-SIAM Symp. Discrete Algorithm*, 2010, pp. 938–948.

[25] A. Andoni, A. Nikolov, K. Onak, and G. Yaroslavtsev, "Parallel algorithms for geometric graph problems," in *Proc. 46th Annu. ACM Symp. Theory Comput.*, 2014, pp. 574–583.

[26] M. T. Goodrich, N. Sitchinava, and Q. Zhang, "Sorting, searching, and simulation in the MapReduce framework," in *Proc. Int. Symp. Algorithms Comput.*, 2011, pp. 374–383.

[27] P. Beame, P. Koutris, and D. Suciu, "Communication steps for parallel query processing," in *Proc. 32nd ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst.*, 2013, pp. 273–284.

[28] A. Czumaj, J. Lacki, A. Madry, S. Mitrovic, K. Onak, and P. Sankowski, "Round compression for parallel matching algorithms," in *Proc. 50th Annu. ACM SIGACT Symp. Theory Comput.*, 2018, pp. 471–484.

[29] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii, "Filtering: A method for solving graph problems in MapReduce," in *Proc. 23rd ACM Symp. Parallelism Algorithms Architectures*, 2011, pp. 85–94.

[30] S. Im, B. Moseley, and X. Sun, "Efficient massively parallel methods for dynamic programming," in *Proc. 49th Annu. ACM SIGACT Symp. Theory Comput.*, 2017, pp. 798–811.

[31] M. Saks and C. Seshadhri, "Estimating the longest increasing sequence in polylogarithmic time," in *Proc. IEEE 51st Annu. Symp. Found. Comput. Sci.*, 2010, pp. 458–467.

Mahdi Boroujeni received the BS and MS degrees in computer engineering from the Sharif University of Technology, Iran. He is currently working toward the PhD degree at the Sharif Univerity of Technology, Iran. His main research interest is efficient approximation algorithms for string similarity problems.

Mohammad Ghodsi received the BS degree in EE from the Sharif University of Technology, Iran, in 1975, the MS degree in EECS from the University of California, Berkeley, Berkeley, California, in 1978, and the PhD degree in computer science from Penn State University, State College, Pennsylvania, in 1989. He has been a faculty member of the Sharif University of Technology (SUT), Iran, since 1979. Presently, he is a full professor with the Computer Engineering Department, Sharif University of Technology, Iran. His main research interests include computational geometry and the design of efficient algorithms.

Saeed Seddighin received the BS degree from the Sharif University of Technology, Iran, and the PhD degree in computer science from the University of Maryland, College Park, Maryland. He is a research assistant professor at the Toyota Technological Institute at Chicago, Chicago, Illinois. He also spent seven months as a postdoc at Harvard University, Cambridge, Massachusetts. He is interested in approximation algorithms and algorithmic game theory.