Journal of Parallel and Distributed Computing 144 (2020) 68–79

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Sequential and parallel algorithms for all-pair k-mismatch maximal common substrings

Sriram P. Chockalingam [a,*], Sharma V. Thankachan [c], Srinivas Aluru [a,b,**]

[a] Institute for Data Engineering and Science, Georgia Institute of Technology, 756 W Peachtree St NW, 12th Floor, Atlanta, GA, 30308, USA
[b] Department of Computational Science and Engineering, Georgia Institute of Technology, 756 W Peachtree St NW, 13th Floor, Atlanta, GA, 30308, USA
[c] Department of Computer Science, University of Central Florida, 4000 Central Florida Blvd, Orlando, FL, 32816, USA

ARTICLE INFO

Article history:
Received 17 April 2018
Received in revised form 12 April 2020
Accepted 28 May 2020
Available online 4 June 2020

Keywords:
Approximate sequence matching
String algorithms
Suffix trees
Hamming distance
Parallel algorithms

ABSTRACT

Identifying long pairwise maximal common substrings among a large set of sequences is a frequently used construct in computational biology, with applications in DNA sequence clustering and assembly. Due to errors made by sequencers, algorithms that can accommodate a small number of differences are of particular interest. Formally, let $\mathcal{D}$ be a collection of n sequences of total length N, ϕ be a length threshold, and k be a mismatch threshold. The goal is to identify and report all k-mismatch maximal common substrings of length at least ϕ over all pairs of strings in $\mathcal{D}$. Heuristics based on seed-and-extend style filtering techniques are often employed in such applications. However, such methods cannot provide any provably efficient run time guarantees. To this end, we present a sequential algorithm with an expected run time of $O(N \log^k N + \text{occ})$, where occ is the output size. We then present a distributed memory parallel algorithm with an expected run time of $O\left(\left(\frac{N}{p}\log N + \text{occ}\right)\log^k N\right)$ using $O\left(\log^{k+1} N\right)$ expected rounds of global communications, under some realistic assumptions, where p is the number of processors. Finally, we demonstrate the performance and scalability of our algorithms using experiments on large high throughput sequencing data.

1. Introduction

Sequence matching algorithms are at the core of many applications in computational biology. Next Generation Sequencing (NGS) [15] instruments sequence hundreds of millions of short reads, that are typically randomly sampled from one or multiple genomes. Deciphering pairwise alignments between the reads is often the first step in many applications. For example, one may be interested in finding all pairs of reads that have a sufficiently long overlap, such as suffix/prefix overlap (for genomic or metagenomic assembly [20]), or substring overlap (for read compression [8], finding RNA sequences containing common exons [9,17], etc.). Much of modern-day high-throughput sequencing is carried out using Illumina sequencers, which have a small error rate ($<$ 1%–2%) and predominantly ($>$99%) substitution errors [16]. Thus, algorithms that tolerate a small number of mismatch errors can yield the same solution as the much more expensive alignment computations. Motivated by such applications, we formulate the following **all-pair k-mismatch maximal common substrings problem**:

Problem 1. Given a collection $\mathcal{D} = \{S_1, S_2, \ldots, S_n\}$ of n sequences with $\sum_i |S_i| = N$, a length threshold ϕ, and a mismatch threshold $k \geq 0$, report all k-mismatch maximal common substrings of length $\geq \phi$ between all pairs of sequences in $\mathcal{D}$.

A pair of two equal length substrings $S_i[x..(x+t-1)]$ and $S_j[y..(y+t-1)]$ is a k-mismatch common substring if the hamming distance between them is $\leq k$. Also, they are a k-mismatch maximal common substring if neither $S_i[(x-1)..(x+t-1)]$ and $S_j[(y-1)..(y+t-1)]$, nor $S_i[x..(x+t)]$ and $S_j[y..(y+t)]$ are a k-mismatch common substring pair.

In this paper, we present efficient solutions for this problem in both sequential as well as parallel settings. Our sequential algorithm runs in $O(N \log^k N + \text{occ})$ expected time, where occ is the output size. Our distributed memory parallel algorithm runs in $O(((N/p)\log N + \text{occ})\ \log^k N)$ expected time using $O(\log^{k+1} N)$ expected communication rounds, where p is the number of processors. Here we make a reasonable assumption that the number of occurrence of any τ-long substring across all sequences in $\mathcal{D}$ is $O(N/p)$. Under this assumption, our algorithm enforces an

* Corresponding author at: Institute for Data Engineering and Science, Georgia Institute of Technology, 756 W Peachtree St NW, 12th Floor, Atlanta, GA, 30308, USA.
** Corresponding author.
E-mail addresses: srirampc@gatech.edu (S.P. Chockalingam), sharma.thankachan@ucf.edu (S.V. Thankachan), aluru@cc.gatech.edu (S. Aluru).

https://doi.org/10.1016/j.jpdc.2020.05.018
0743-7315/

effective partitioning of a series of modified suffix trees to localize processing within each processor. We demonstrate the scalability and performance of our parallel algorithm using genomic datasets ranging in size from 18 million to over 270 million reads, on up to 1024 processor cores.

Related work. To solve such problems in practice, seed-and-extend style filtering approaches are often employed. The underlying principle is: if two sequences have a k-mismatch common substring of length $\geq \phi$, then they must have an exact common substring of length at least $\tau = \left\lceil \frac{\phi-k}{k+1} \right\rceil$. Therefore, using a fast hashing technique, all pairs of sequences that have a τ-length common substring are identified. Then, by exhaustively checking all such candidate pairs, the final output is generated. In the sequential setting, the filtering heuristics can be broadly classified under three categories: suffix filtering [12,23], spaced seeds filtering [3], and substring filtering [21]. In case of parallel heuristic methods, filtering based methods mainly focused on the corresponding applications. For example, [11] and [19] proposed suffix tree based parallel clustering of EST data. Clearly, a filtering-based algorithm cannot provide any run time guarantees and often times the candidate pairs generated can be overwhelmingly larger than the final output size. Recently, [22] published the first and only known sub-quadratic exact sequential algorithm for this problem that includes insertions and deletions along with mismatches. Work done on accelerating pairwise edit distance estimations among n sequences using sequence alignment can also be applied to this problem [18]. However, for a large number of short sequences (with low error rate, mostly mismatches), all pair sequence alignment is impractical because of its quadratic time complexity.

This paper is organized as follows. In Section 2, we introduce notations and data structures used in our algorithm. Due to the absence of a provably efficient sequential algorithm for this problem, we first design such an algorithm and present it in Section 3. In Section 4, we describe the parallel algorithm in detail and prove the claimed bounds on expected time and communication rounds. In Section 5, we discuss the implementation details of the parallel algorithm. Finally in Section 6, we discuss the results of our implementation on genomic and gene expression datasets.

2. Notation and preliminaries

Let Σ denote the alphabet of the sequences in $\mathcal{D}$. Throughout this paper, both $|\Sigma|$ and k are assumed to be constants. Let $\mathsf{T} = S_1\$_1S_2\$_2 \ldots S_n\$_n$ be the concatenation of all sequences in $\mathcal{D}$, separated by special characters $\$_1, \$_2, \ldots, \$_n$. Here each $\$_i \notin \Sigma$ is a unique special symbol and is lexicographically larger than all characters in Σ. Clearly, there exists a one to one mapping between the positions in T (except the $\$_i$ positions) and the positions in the sequences in $\mathcal{D}$. Let $\mathsf{seq}(x)$ denote the identifier of the sequence corresponding to the position x in T. We use $\mathsf{lcp}(\alpha, \beta)$ to denote the longest common prefix of strings α and β, and $\mathsf{lcp}_k(\alpha, \beta)$ to denote their longest common prefix while permitting at most k mismatches. The reverse of a string α is denoted by $\overleftarrow{\alpha}$. Therefore, $\overleftarrow{S_i}[x] = S_i[|S_i| - x + 1]$. We now briefly review some standard data structures that will be used in our algorithms.

2.1. Suffix tree, suffix array and LCP data structures

Denote the suffix of T starting at position x by $\mathsf{T}[x..]$, the prefix of T ending at x by $\mathsf{T}[..x]$, and the substring of T starting at position x and ending at position y by $\mathsf{T}[x..y]$. The generalized suffix tree of $\mathcal{D}$ (equivalently, the suffix tree of T), denoted by GST, is a lexicographic arrangement of all suffixes of T as a compact trie [14,24]. The GST consists of exactly $|\mathsf{T}|$ leaves, and at most $(|\mathsf{T}| - 1)$ internal nodes, all of which have at least two child nodes each. The edges are labeled with substrings of T. Let $path(u)$ refer to the concatenation of edge labels on the path from root to node u. If u is a leaf node, then $path(u)$ is a unique suffix of T (equivalently a unique suffix of a unique string in $\mathcal{D}$) and vice versa. For any node u in GST, *node-depth*(u) is the number of its ancestors and *string-depth*(u) is the length of its path.

The suffix array [13], SA, is such that $\mathsf{SA}[i]$ is the starting position of the suffix corresponding to the ith left most leaf in the suffix tree of T, i.e., the starting position of the ith lexicographically smallest suffix of T. The inverse suffix array ISA is such that $\mathsf{ISA}[j] = i$, if $\mathsf{SA}[i] = j$. The Longest Common Prefix array LCP is such that, for $1 \leq i < |\mathsf{T}|$

$$\mathsf{LCP}[i] = |\mathsf{lcp}(\mathsf{T}[\mathsf{SA}[i]..], \mathsf{T}[\mathsf{SA}[i+1]..])|$$

In other words, $\mathsf{LCP}[i]$ is the *string-depth* of the lowest common ancestor of the ith and $(i+1)$th leaves in the suffix tree. There exist optimal sequential algorithms for constructing all these data structures in $O(|\mathsf{T}|)$ space and time [10].

All operations on GST required for our purpose can be simulated using SA, ISA, LCP, and a range minimum query (RMQ) data structure [6] over the LCP array. A node u in GST can be uniquely represented by an interval $[\mathsf{sp}(u), \mathsf{ep}(u)]$, where $\mathsf{sp}(u)$ and $\mathsf{ep}(u)$ are respectively the leftmost and rightmost indexes of u's leaves in SA. *string-depth*(u) is the minimum value in $\mathsf{LCP}[\mathsf{sp}(u), \mathsf{ep}(u)-1]$ (can be computed in constant time using an RMQ). Similarly, the longest common prefix of any two suffixes can also be computed in $O(1)$ time. Finally, the k-mismatch longest common prefix of any two suffixes x and y can be computed in $O(k)$ time as follows: let $l = |\mathsf{lcp}(\mathsf{T}[x..], \mathsf{T}[y..])|$, then for any $k \geq 1$,

$$|\mathsf{lcp}_k(\mathsf{T}[x..], \mathsf{T}[y..])| = \begin{cases} l, \text{ If either of } \mathsf{T}[x+l], \mathsf{T}[y+l] \\ \quad \text{is a special character (i.e., } \$_i) \\ l + 1 + |\mathsf{lcp}_{k-1}(\mathsf{T}[(x+l+1)..], \\ \quad \mathsf{T}[(y+l+1)..])|, \text{otherwise.} \end{cases}$$

Based on these terminologies, we redefine Problem 1 as follows.

Problem 2. Given T, ϕ and k, report all tuples (x, y, t), such that

1. $t = \mathsf{lcp}_k(\mathsf{T}[x..], \mathsf{T}[y..]) \geq \phi$
2. $\mathsf{T}[x-1] \neq \mathsf{T}[y-1]$
3. $\mathsf{seq}(x) \neq \mathsf{seq}(y)$

The first and the second constraints ensure length threshold and left maximality (thereby right maximality) conditions, whereas the third constraint ensures that the suffixes belong to two different sequences.

3. Our sequential algorithm

3.1. The exact match case

As a warm up, we first show how to solve the exact match case ($k = 0$) in optimal $O(N + \mathsf{occ})$ worst case time. First create the GST, then identify all nodes u in GST such that *string-depth*$(u) \geq \phi$ and *string-depth*$(parent(u)) < \phi$. Such nodes are termed as marked nodes. Clearly, a pair of suffixes satisfies condition (1) specified in Problem 2 iff their corresponding leaves are under the same marked node. This allows us to process the suffixes under each marked node w independently as follows: let Suff_w denote the set of starting positions of the suffixes of T corresponding to the leaves in the subtree of w. That is, $\mathsf{Suff}_w = \{\mathsf{SA}[j] \mid \mathsf{sp}(w) \leq j \leq \mathsf{ep}(w)\}$. Then,

1. Partition Suff_w into (at most) $\Sigma + 1$ buckets, such that for each $\sigma \in \Sigma$, there is a unique bucket containing all suffixes with previous character σ. All suffixes that have a \$ symbol as the previous character are put together in a special bucket. Note that a pair (x, y) satisfies condition (2) specified in Problem 2 and hence a valid output pair, only if both x are y are not in the same bucket, or if both of them are in the special bucket.
2. Sort all suffixes w.r.t. the identifier of the corresponding sequence (i.e., $\mathsf{seq}(\cdot)$). Therefore, within each bucket, all suffixes from the same sequence appear contiguously.
3. For each x, report all answers of the from $(x, \cdot)$ as follows: scan every bucket, except the bucket in which x belongs to, unless x is in the special bucket. Then output (x, y) as an answer, where y is not an entry in the contiguous chunk of all suffixes from $\mathsf{seq}(x)$.

The construction of GST and Step (1) takes $O(N)$ time. Step (2) over all marked nodes can also be implemented in $O(N)$ time via integer sorting. By noting down the sizes of each chunk during Step (2), we can implement Step (3) in time proportional to the sizes of input and output. By combining all, the total time complexity is $O(N \cdot |\Sigma| + \mathsf{occ})$, i.e., $O(N + \mathsf{occ})$ for constant sized alphabet.

3.2. The k-mismatch case

Standard string data structures such as suffix trees and suffix arrays can directly support only exact matching. To enable finding approximate matches, we present the following novel approach: We transform the approximate matching problem over given strings into an equivalent exact matching problem over a set of carefully crafted inexact copies of the suffixes of the original strings. Such inexact copies of the suffixes will be termed as *modified suffixes*. Such a solution would be obvious if we could generate modified suffixes corresponding to all possible ways of accommodating k mismatches from the original suffixes. However, this strategy is combinatorially explosive. The key to our algorithm is the specification of a bounded set of modified suffixes that nevertheless are sufficient to compute all k-mismatch maximal common substrings. Details follow.

Definition 1. Let # be a special symbol not in Σ. A k-modified suffix is a suffix of T with k of its characters replaced by #.

Let Δ be a set of positions. Then, $\mathsf{T}^{\Delta}[x..]$ denotes the $|\Delta|$-modified suffix obtained by replacing $|\Delta|$ positions in the suffix $\mathsf{T}[x..]$ as specified by Δ. For example, let $\mathsf{T} = aaccgattcaa$, $\Delta = \{2, 4\}$, then $\mathsf{T}[5..] = gattcaa$ and $\mathsf{T}^{\Delta}[5..] = g\#t\#caa$.

Our algorithm consists of two main phases. In the first phase, we create a collection of sets of k-modified suffixes. In the second phase, we independently process each set and extract the answers. The first phase takes $O(NH^k)$ time, where as the second phase takes $O(NH^k + \mathsf{occ})$ time, where H is the height of GST. It is known that the expected value of H is $O(\log N)$ [4]. Therefore, by combining the time complexities of both phases with H replaced by $O(\log N)$, we obtain the expected run time as claimed. We now describe these phases in detail.

3.2.1. Details of phase-1

This phase is recursive (with levels of recursion starting from 0 up to k), such that at each level $h > 0$, we create a collection of sets of h-modified suffixes (denoted by $C_1^h, C_2^h, \ldots$) from the sets in the previous level. At level 0, there is only a single set C_0^1, the set of all suffixes of T. See Fig. 1 for an illustration. To generate the sets at level h, we take each set C_g^{h-1} at level $(h-1)$ and do the following:

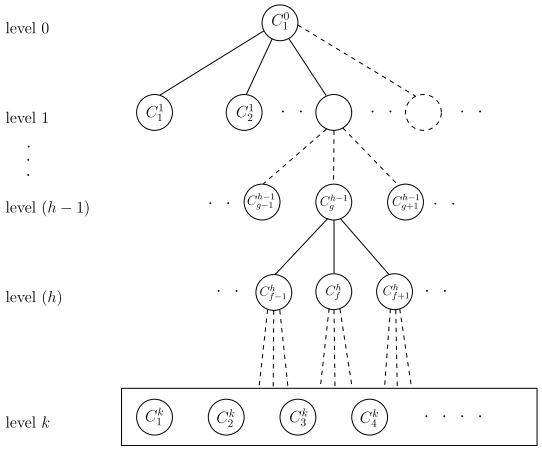

Fig. 1. The sets $\ldots, C_{f-1}^h, C_f^h, C_{f+1}^h \ldots$ of h-modified suffixes are generated from the set C_g^{h-1}.

- Create a compact trie of all strings in C_g^{h-1}
- For each internal node w in the trie, create a set consisting of the strings corresponding to the leaves in the subtree of w, but with their $(l+1)$th character replaced by #. Here l is *string-depth*(w). Those strings with their $(l+1)$-th character is a $\$_i$ symbol are not included.

From our construction procedure, the following properties can be easily verified.

Property 1. *All modified suffixes within the same set (at any level) have # symbols at the same positions and share a common prefix at least until the last occurrence of #.*

Property 2. *For any pair (x, y) of positions, there will be* **exactly one** *set at level h, $1 \le h \le k$, such that it contains h-modified suffixes of $\mathsf{T}[x..]$ and $\mathsf{T}[y..]$ with # symbols at the first h positions in which they differ. Therefore, the* lcp *of those h-modified suffixes is equal to the lcp_h of $\mathsf{T}[x..]$ and $\mathsf{T}[y..]$.*

We have the following result about the sizes of these sets.

Lemma 1. *No set is of size more than N and the sum of sizes of all sets at a particular level h is $\le N \times H^k$, where H is the height of* GST.

Proof. The first statement follows easily from our construction procedure and the second statement can be proved via induction. Let $\mathcal{S}_h$ be the sum of sizes of all sets at level h. Clearly, the base case, $\mathcal{S}_0 = |C_1^0| = N$, is true. The sum of sizes of sets at level h generated from C_g^{h-1} is at most $|C_g^{h-1}| \times$ the height of the compact trie over the strings in C_g^{h-1}. The height of the compact trie is $\le H$, because if we remove the common prefix of all strings in C_g^{h-1}, they are essentially suffixes of T. By putting these together, we have $\mathcal{S}_h \le \mathcal{S}_{h-1} \cdot H \le \mathcal{S}_{h-2} \cdot H^2 \le \cdots \le NH^k$. □

Space and Time Analysis: We now show that Phase-1 can be implemented in $O(N)$ space and $O(N \log^k N)$ time in expectation. Consider the step where we generate sets from C_g^{h-1}. The lexicographic ordering between any two $(h-1)$-modified suffixes in C_g^{h-1} can be determined in constant time. i.e., by simply checking the lexicographic ordering between those suffixes obtained by deleting their common prefix up to the last occurrence of #. Therefore, suffixes can be sorted using any comparison based sorting algorithm. After sorting, the lcp between two successive

strings can be computed in constant time. Using the sorted order and lcp values, we can construct the compact trie using standard techniques in the construction of suffix tree [5]. Finally, the new sets can be generated in time proportional to their size. In summary, the time complexity for a particular C_g^{h-1} is $O(|C_g^{h-1}|(\log N + H))$. Overall time complexity is

$$\sum_{h\leq k}\sum_{f}|C_f^k| + (H+\log n)\sum_{h<k}\sum_{f}|C_f^h| = O(N(\log N+H)^{k-1}H)$$

By replacing H by $O(\log N)$, we bound the expect run time of Phase-1 by $O(N\log^k N)$.

We generate the sets in pre-order of the corresponding node in the recursion tree. As soon as a set (at level k) is generated, we immediately pass it to Phase-2, extract the necessary information and discard it from the working space. Also, any set at level $h < k$ is also deleted after all k-level sets in its subtree are processed. This way, at any point of time in the execution of the algorithm, we need to maintain only k sets, corresponding to the sets in a root to leaf path in the recursion tree. Since the size of each set is at most N (Lemma 1), we can bound the working space also by $O(N)$, assuming $k = O(1)$.

3.2.2. Details of phase-2

In this phase, we seek to process each set C_f^k created by Phase-1 independently and generate the answers in time linear to the total size of all sets and output. i.e., $O(NH^k+\text{occ})$. We first present a simple $O((N+\text{occ})H^k)$ time approach. Following are the key steps.

1. Create a compact trie over all k-modified suffixes in C_f^k. Then identify the marked nodes as before. Recall that a marked node has a *string-depth* $\geq \phi$, where as the *string-depth* of its parent is $< \phi$.
2. Let Δ be the set of k positions corresponding to modifications in the k-modified suffixes in C_f^k. Clearly, if the leaves corresponding to two modified suffixes (say $\mathsf{T}^\Delta[x..]$ and $\mathsf{T}^\Delta[y..]$) are in the same subtree of a marked node, then their lcp_k is $\geq \phi$. If $\text{seq}(x) \neq \text{seq}(y)$ and $\mathsf{T}[x-1] \neq \mathsf{T}[y-1]$, then report (x, y) as an answer.

The trie can be created in time linear to the size of C_f^k. Note that the key step in the creation of a trie is the sorting of k-modified suffixes. To do it efficiently, we map each k-modified suffix to the lexicographic rank of the suffix obtained by removing all characters (from left) until its last # symbol. Using this as the key, the k-modified suffixes can be sorted via a linear time integer sorting. The second step of extracting answers can also be implemented using the exact same procedure described in Section 3.1. However, the problem with this approach is that, a pair (x, y) can get reported more than once, although only once per set. In the worst case, an answer can get reported H^k times. The resulting time complexity is therefore $O((N+\text{occ})H^k)$.

3.2.3. Improving the run time complexity via bucketing

To achieve the claimed $O(NH^k + \text{occ})$ run time, we need to ensure that each output (x, y) is reported exactly once. For this, we exploit Property 2 as follows: while processing a pair of two k-modified suffixes $\mathsf{T}^\Delta[x..]$ and $\mathsf{T}^\Delta[y..]$ under the subtree of some marked node, report (x, y) as an answer iff

1. $\text{lcp}(\mathsf{T}^\Delta[x..], \mathsf{T}^\Delta[y..]) = \text{lcp}_k(\mathsf{T}[x..], \mathsf{T}[y..])$.
2. $\text{seq}(x) \neq \text{seq}(y)$
3. $\mathsf{T}[x-1] \neq \mathsf{T}[y-1]$

From Property 2, for a pair (x, y), there will be only one pair of k-modified suffixes satisfying this condition (1). The following is unique to that pair: $\mathsf{T}[x+l-1] \neq \mathsf{T}[y+l-1]$ for all $l \in \Delta$. Therefore, the task can be executed efficiently by processing the set of k-modified suffixes in the subtree of each marked node w as follows:

1. Partition them into (at most) $\Sigma + 1$ buckets based on the previous character as in Section 3.1.
2. Partition the k-modified suffixes in each bucket into (at most) $|\Sigma|^k$ sub-buckets based on the sequence of k characters that were originally at the positions in Δ. Each sub-bucket is therefore associated with a unique string of length $(1+k)$: the (previous) character corresponding to the bucket in which it belongs to, followed by the sequence of k characters at the positions in Δ.
3. Within each sub-bucket, sort the k-modified suffixes based on the identifier of the sequence to which it belongs.
4. Finally, for each $\mathsf{T}^\Delta[x..]$, we visit each sub-bucket and find answers of the form $(x, \cdot)$ as follows: let $c_0, c_1, c_2, \ldots, c_k$ be the sequence of $(1+k)$ characters corresponding to a sub-bucket. If $c_0 \neq \mathsf{T}[x-1]$ or $\mathsf{T}[x-1]$ is some $\$_i$ symbol and $c_t \neq \mathsf{T}[x+t-1]$ for $t = 1, 2, \ldots k$, then for all entries $\mathsf{T}^\Delta[y..]$ in the sub-bucket with $\text{seq}(x) \neq \text{seq}(y)$, report (x, y) as an answer. Notice that the entries within a sub-bucket are sorted according to the sequence identifier. Therefore, all entries with $\text{seq}(x) = \text{seq}(y)$ comes together as a contiguous chunk, which can be easily skipped.

Analysis. The overall time for implementing the first three steps is $O(NH^k)$ and final step takes $O(NH^k|\Sigma|^k + \text{occ})$ time. Therefore, total time complexity is $O(NH^k + \text{occ})$, assuming k and $|\Sigma|$ are constants.

Theorem 1. *Problem 1 can be solved in $O(N)$ space and $O(N\log^k N + \text{occ})$ expected time, assuming k and $|\Sigma|$ are constants.*

Note: If the hamming distance between two reads is $< k$, then our algorithm will not output them as an answer. To capture such answers, we run the algorithm for all numbers of mismatches starting from 0 up to k. The run-time remains the same.

4. Our parallel algorithm

In this section, we show how to extend our ideas to obtain a provably efficient parallel algorithm. We assume that the input set of strings $\mathcal{D}$, or equivalently their concatenated string T, is distributed across the p processors such that each processor has the same number of total characters. Note that a maximal common substring with k-mismatches is essentially a concatenation of $(k+1)$ maximal exact matches separated by k mismatch positions in between. Among the various ways the k mismatches can be positioned in a substring of length $\geq \phi$, the shortest maximal exact match occurs when the k mismatch positions are uniformly distributed. This leads us to the following observation.

Observation 1. If two sequences $S_i, S_j \in \mathcal{D}$ have a *k-mismatch maximal common substring of length $\geq \phi$*, then within that region, there is a maximal exact match of length $\geq \tau = \left\lceil \frac{\phi-k}{k+1} \right\rceil$.

In other words, a k-mismatch maximal substring of length $\geq \phi$ contains at least one maximal exact match of length $\geq \tau$. Depending upon the position of that match among the $(k+1)$ matching segments, we categorize answers into different types: *An answer (x, y) is of type-h ($0 \leq h \leq k$) if the $(h+1)$th exact match segment is of length $\geq \tau$.* Fig. 2 illustrates the different types of output for $k = 2$.

We present our algorithm in terms of the generalized suffix tree (GST) of $\mathcal{D}$ (i.e., a suffix tree of T). However, in the actual implementation, the required operations are equivalently carried

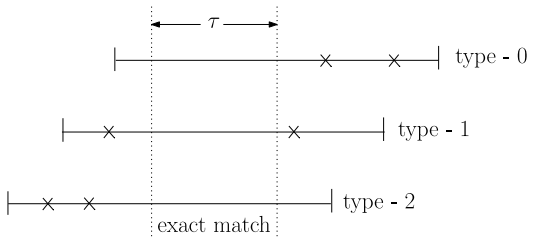

Fig. 2. Illustration of output types for $k = 2$.

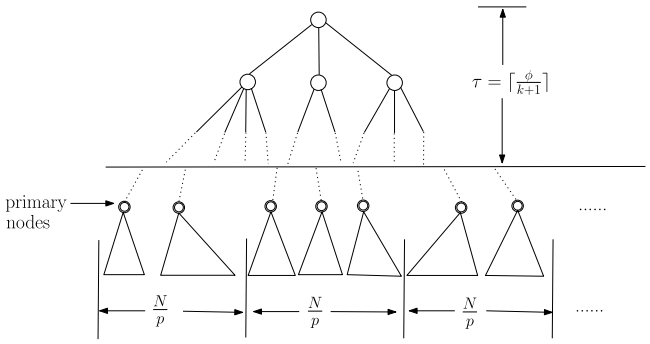

Fig. 3. Re-distribution of the primary nodes of GST.

out using SA, ISA, LCP and the RMQ data structures. While the GST view adds greatly to presentation clarity, its implementation using SA, ISA, and LCP leads to efficient implementation. Using the recent result by Flick and Aluru [7], we construct these data structures in $O(\mathsf{pSort}(N,p)\log N)$ time, where $\mathsf{pSort}(N,p)$ is the time for parallel sorting of N numbers using p processors. Flick and Aluru's [7] algorithm is based on a non trivial adaptation of the classic prefix doubling algorithm by Manber and Mayers [13]. After this step, each processor gets a contiguous chunk of size roughly N/p of each array, i.e., for $i = 1, 2, \ldots, p$, the ith processor (P_i) gets the chunk corresponding to the range $[1 + (i-1)N/p, iN/p]$. Additionally, we maintain all the above data structures for the reverse of T, denoted by $\overleftarrow{\mathsf{T}}$. The respective data structures for $\overleftarrow{\mathsf{T}}$ are denoted by SA′, ISA′, LCP′ and RMQ′.

Using a distributed GST, our algorithm first identifies all the set of suffixes having the initial exact match of $\geq \tau$ characters. It accomplishes this by selecting all the internal nodes u of GST such that *string-depth*$(u) \geq \tau$ and the *string-depth* of u's parent is $< \tau$ (Section 4.1). After re-distributing the chosen internal nodes across p processors, we process the nodes independently (Section 4.2) and generate the suffix pairs satisfying the three constraints (Sections 4.3 and 4.4). Since GST is stored in a distributed fashion, independently processing these internal nodes requires information about suffixes that are not stored locally. To retrieve the requisite information, we use rounds of all-to-all communication. The key reason for using this strategy is that independent processing of the suffix tree nodes localizes the computation and hence provides good performance by minimizing the communication overheads.

4.1. Load balancing via re-distribution

After construction of the data structures as mentioned above, the next step is *re-distribution*, where we partition the GST and distribute subtrees of the GST to the p processors. We term a GST node u as *primary* if *string-depth*$(u) \geq \tau$ and *string-depth*$(parent(u)) < \tau$. Alternatively, the first node of a root to leaf path that has a *string-depth* $\geq \tau$ is a primary node. See Fig. 3 for an illustration. We assign each *primary* node, along with all suffixes corresponding to the leaves in its subtree, to a single processor, in such a way that the total number of suffixes associated is approximately the same across all processors. We make the following reasonable assumption: τ is sufficiently long in practice, such that the frequency of occurrence of any *τ-long substring across all sequences in $\mathcal{D}$ is $\leq N/p$ (infact $\leq c \cdot N/p$ for any constant c works).* Under this assumption, we can easily perform re-distribution in such a way that the total number of suffixes associated with any processor is $\leq 2N/p$. Note that the redistribution amounts to merely adjusting the boundaries according to which the sorted arrays are partitioned across the processors.

Each primary node u is processed by the processor assigned to it. We seek to achieve the following: Let T[x..] and T[y..] be two suffixes under a primary node u, which is assigned to processor q, and l be the length of their lcp. Then, if T[x..$(x+l-1)$] and T[y..$(y+l-1)$] correspond to a maximal exact matching segment (which is clearly of length $\geq \tau$) of an answer (x', y'), pair (x', y') will be reported as an answer while processing the subtree rooted at u by q. We now present the details.

4.2. Processing of a primary node

We present an overview of our strategy using a small example. Let E be a set of three strings S_1, S_2 and S_3. Let S_1, S_2 and S_3 be the strings TAATACAGGTACATAACT, CAGGTACAGAACTAACGC and TAATTCAGGTACACTGAG respectively. Given the input set E, all 1-mismatch maximal common substring pairs of length at least 13 are desired. The expected output pairs for this example are $(S_1[6..18], S_2[1..13])$ and $(S_1[1..13], S_3[1..13])$. Clearly, the first pair is a type-0 output, while the second pair is a type-1 output. Based on Observation 1, the 1-mismatch maximal common substring pairs should have a maximal exact match of at least $\lceil (13-1)/(1+1) \rceil = 6$ characters. In the case of our example, this exact match is CAGGTACA and it occurs in suffixes $S_1[6..]$, $S_2[1..]$ and $S_3[6..]$. Therefore, there exists a primary node u in the generalized suffix tree of E, which includes the three suffixes and its *string-depth* is 7.

After selecting the primary node u, the first output pair $(S_1[6..18], S_2[1..13])$ can be generated from u by matching the modified suffixes $S_1^{\Delta_a}[6..]$ and $S_2^{\Delta_a}[1..]$, where $\Delta_a = \{8\}$. In case of the second output pair $(S_1[1..13], S_2[1..13])$, the modified position should be to the left of the initial exact match. This case can be handled by matching the modified suffixes $\overleftarrow{S_1[13..]}^{\Delta_b}$ and $\overleftarrow{S_3[13..]}^{\Delta_b}$, where $\Delta_b = \{8\}$. This example shows the differences in the way modified suffixes can be used to generate different types of output pairs.

We now illustrate the procedure to process a primary node u using Fig. 4. The procedure is recursive and is similar to that in Section 3.2.1, except that the set C_1^0 at the root denotes the set of all suffixes corresponding to the leaves in the subtree of u. Recall that any string within a set at level h is a suffix in C_0^1 with h of its characters replaced by #. All modified suffixes within a set share a common prefix of length at least up to the last position of modification. Also, by a straightforward generalization of Property 2, for any two suffixes T[x..], T[y..] $\in C_1^0$ and for any level $h \in [0, k]$, there will be a unique set containing two modified suffixes $\mathsf{T}^{\Delta}[x..]$ and $\mathsf{T}^{\Delta}[y..]$ such that Δ is the set of first h positions in which T[x..] and T[y..] differ, i.e., $\mathsf{lcp}(\mathsf{T}^{\Delta}[x..], \mathsf{T}^{\Delta}[y..]) = \mathsf{lcp}_h(\mathsf{T}[x..], \mathsf{T}[y..])$

From now onwards, we use the following terminology. The collection of sets at level h is termed an order-h universe of the initial set C_1^0, and its total size is $\leq H^k|C_1^0|$ (refer to Lemma 1).

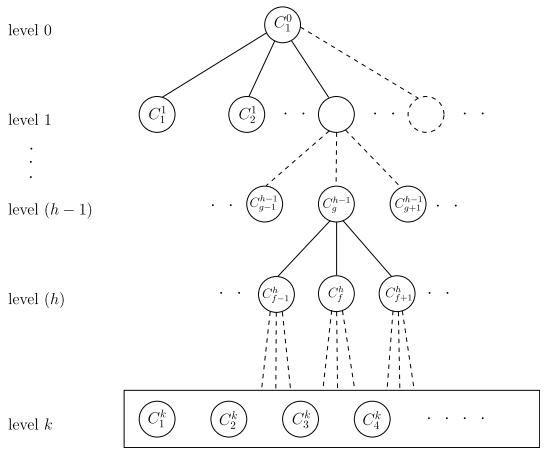

Fig. 4. Recursive generation of partitions.

Also, each set at level h is termed a partition of the order-h universe of C_0^1. In subsequent sections, we show how to process the partitions and generate answers. Specifically, type-0 answers are generated while processing the partitions at level k, whereas answers of any other type, say h, are generated while processing the partitions at level $(k-h)$. We start with the simpler case of type-0.

4.3. Generating type-0 answers

The partitions at level k are processed independently of each other. To process any partition C_f^k, first create a compact trie of all modified suffixes within C_f^k. Then, identify each node in the trie whose *string-depth* is $\geq \phi$, whereas the *string-depth* of its parent is $< \phi$. We call such nodes as *secondary nodes*. Then, for any two modified suffixes $\mathsf{T}^{\Delta}[x..]$ and $\mathsf{T}^{\Delta}[y..]$ corresponding to the leaves in the subtree of a secondary node v, $\mathsf{lcp}_k(\mathsf{T}[x..], \mathsf{T}[y..]) \geq \phi$. Therefore, to check if (x, y) is an answer, it is enough to check if $\mathsf{T}[x-1] \neq \mathsf{T}[y-1]$ and $\mathsf{seq}(x) \neq \mathsf{seq}(y)$. To do this efficiently, we apply the bucketing strategy described in Section 3.2.3 over the set of suffixes corresponding to each secondary node v.

4.4. Generating type-h answers for $h \geq 1$

We first present the key intuition behind our algorithm for generating type-h answers. Recall that in a type-h answer (x', y') with x and y as the starting positions of the $(h+1)$th exact match common segment, the lcp of $\mathsf{T}[x..]$ and $\mathsf{T}[y..]$ should be at least τ. In other-words, $\mathsf{T}[x..]$ and $\mathsf{T}[y..]$ must be suffixes corresponding to the leaves in the subtree of some primary node. Therefore, while processing a primary node u, we are looking for (x, y) pairs with the following constraints:

- $\mathsf{seq}(x) \neq \mathsf{seq}(y)$
- Let

  $t = \mathsf{lcp}_{k-h}(\mathsf{T}[x..], \mathsf{T}[y..])$,

  $\mathsf{T}[x-1] \neq \mathsf{T}[y-1]$ and

  $t' = \mathsf{lcp}_{h-1}(\overleftarrow{\mathsf{T}[..(x-2)]}, \overleftarrow{\mathsf{T}[..(y-2)]})$

  Then,

  $t + 1 + t' \geq \phi$

The type-h answer corresponding to the above constraints is $(x', y') = (x-t-1, y-t-1)$. In order to efficiently generate these pairs, we process each partition C_f^{k-h} as follows. We use Δ to denote the corresponding set of $(k-h)$ positions of modifications.

1. Create a compact trie of all modified suffixes in C_f^{k-h}. For each internal node w in the trie (we denote the *string-depth*(w) by t), we apply steps 2 and 3 described below.
2. Let $\mathcal{S}_w$ be the set of modified suffixes corresponding to the leaves in the subtree rooted at w. Create another set $\mathcal{S}'_w$ of reverse prefixes w.r.t. $\mathcal{S}_w$ as follows:

   $\mathcal{S}'_w = \{\overleftarrow{\mathsf{T}[..(x-2)]} \mid \mathsf{T}^{\Delta}[x..] \in \mathcal{S}_w\}$
3. Process each such $\mathcal{S}'_w$ as follows:
   (a) Create an order-$(h-1)$ universe of $\mathcal{S}'_w$ and process each partition *part* in it as described in steps 3b to 3d.
   (b) Create a compact trie of *part* and identify the secondary nodes within it. A node is secondary if its *string-depth* is $\geq \phi'$, where as the *string-depth* of its parent is $< \phi'$, where $\phi' = (\phi - 1 - t)$. Let Δ' be the positions of modifications of the modified suffixes in *part*.
   (c) Note that if $\overleftarrow{\mathsf{T}[..(x-2)]}^{\Delta'}$ and $\overleftarrow{\mathsf{T}[..(y-2)]}^{\Delta'}$ correspond to the leaves in the subtree of a secondary node, $\mathsf{T}[x-1] \neq \mathsf{T}[y-1]$ and $\mathsf{seq}(x) \neq \mathsf{seq}(y)$, then clearly $(x', y') = (x-1-t', y-1-t')$ is an answer, where t' is the lcp of $\overleftarrow{\mathsf{T}[..(x-2)]}^{\Delta'}$ and $\overleftarrow{\mathsf{T}[..(y-2)]}^{\Delta'}$. Using an exactly similar bucketing technique described in Section 4.3, we can process *part* in time proportional to $|part|$ and the number of answers generated from *part*.
   (d) Finally, we note the following key point: an answer (x', y') may get generated from multiple partitions. However, in exactly one partition, the positions specified in Δ and Δ' correspond to mismatches (from Property 2). Therefore, to avoid reporting the same answer (x', y'), we make sure that positions in Δ and Δ' are mismatch positions. We also check that $\mathsf{T}[x+t] \neq T[y+t]$ to ensure right maximality.

We now prove the correctness of the above procedure with respect to a fixed type-h answer (x', y') with x and y being the starting positions of the $(h+1)$th maximal exact common substring which is sufficiently long (i.e., length $\geq \tau$). As mentioned in Section 4.2, there exists a partition C_f^{k-h} where the positions of modifications (i.e., Δ) are the first $(k-h)$ mismatch positions between $\mathsf{T}[x..]$ and $\mathsf{T}[y..]$. While creating the compact trie of this particular partition C_f^{k-h}, there will be a node w from which the modified suffixes $\mathsf{T}^{\Delta}[x..]$ and $\mathsf{T}^{\Delta}[y..]$ diverge, i.e., $\mathsf{T}[x+t] \neq T[y+t]$, where $t = \mathsf{lcp}_{k-h}(\mathsf{T}[x..], \mathsf{T}[y..])$ and this ensures the right maximality of the match. With the condition $\mathsf{T}[x-1] \neq T[y-1]$, we are imposing the hth mismatch at these positions. Now, while creating the $(h-1)$th order universe out of $\mathcal{S}'_w$, there will again be a unique partition (say *part*) such that the modifications (denoted by Δ') are at the first $(h-1)$ positions in which $\overleftarrow{\mathsf{T}[..(x-2)]}$ and $\overleftarrow{\mathsf{T}[..(y-2)]}$ differ. Finally, we report the answer (x', y') uniquely while processing this specific partition.

Note that an answer can belong to multiple types. Therefore while processing and reporting a type h answer, we simply check if it belongs to another type $h' < h$. If so, we do not output it as a type-h answer.

4.5. Bounding the run time

After the initial construction of the GST and its re-distribution, the remainder of our algorithm localizes computations within each processor. This is the key reason for devising the complex

strategy outlined, which should provide good performance by minimizing the communication overhead. However, as a preparatory step to execute the primary nodes assigned to a processor, it needs to create the data structures SA, ISA, etc. for the suffixes in the underlying subtree. While these can be locally computed, the additional time for doing so can be saved by deducing them from the corresponding globally distributed arrays (SA, ISA, etc.) for the set of strings $\mathcal{D}$. We rely on rounds of all-to-all communication to achieve this. In this section, we prove that the expected computation time is $O((N/p + \text{occ}) \log^k N)$. In the next section, we bound the expected number of all-to-all communication rounds to $O(\log^{k+1} N)$ and the expected communication cost to $O((N/p) \log^{k+1} N)$. Adding together the computation and communication costs, we obtain the claimed run time of $O\left(\left(\frac{N}{p} \log N + \text{occ}\right) \log^k N\right)$ in expectation.

The key steps involved in our parallel algorithm are (i) recursive construction of sets of modified suffixes (partitions), and (ii) processing of some of these partitions to generate the outputs. The first part involves the construction of compact tries, whereas the second part involves bucketing, followed by sorting of modified suffixes w.r.t. $\text{seq}(\cdot)$ and scanning the buckets to generate the outputs. The construction of compact tries consists of two key steps: (i) arrange the modified suffixes in their lexicographic order, and (ii) compute the lcp between every pair of consecutive modified suffixes. The topology of the trie can be inferred in additional linear time [5].

Note that in our case, the given set of modified suffixes have a common prefix at least up to the last position (say l) of modification. Therefore, they can be sorted with respect to the rank (inverse suffix array value) of the corresponding suffix obtained after removing the first l characters. The pertinent inverse suffix array values can be obtained by querying on ISA (the required communication cost will be bounded in the next section). With such a reduction, the task of (modified) suffix sorting can be reduced to integer sorting, for which standard linear time algorithms can be used. Computing the lcp values also requires a linear number of queries on the distributed arrays. In summary, the compact tries needed throughout the execution of the algorithm can be constructed locally in time linear in the number of modified suffixes involved.

We now bound the total time for construction of the compact tries on any processor. Let u be a primary node of size m. Let H and H_r be the heights of the suffix trees of T and $\overleftarrow{\mathsf{T}}$, respectively. Then, the total of the number of modified suffixes and modified reverse prefixes generated while processing u is at most (refer to Lemma 1)

$$m(H^k + H^{k-1}H_r + H^{k-2}H_r^2 + .. + H_r^k) \leq mk \cdot \max\{H^k, H_r^k\}$$

Therefore, using our linear time trie construction procedure, the time for construction of all compact tries w.r.t. node u is $O(m \cdot max\{H^k, H_r^k\})$. Since the sum of subtree sizes of all primary nodes assigned to a processor is bounded by $O(N/p)$, the worst case run time on any processor is $O(N/p \cdot max\{H^k, H_r^k\})$.

The second part in processing a set involves bucketing followed by sorting and scanning. Here also, we use linear time integer sorting algorithms. The time for scanning and generating output pairs takes time linear to the size of the sets and the number of outputs generated. However, we may encounter the same answer while processing multiple sets (at most $k \cdot max\{H^k, H_r^k\}$ in number), although we report it only once. Therefore, we bound the total time by $O((N/p + \text{occ}) \cdot max\{H^k, H_r^k\})$. By replacing H and H_r by $O(\log N)$, the expected height of a suffix tree of a string of length N [4], we obtain the claimed run time.

4.6. Bounding the communication costs

Since the GST is distributed, the construction of the compact suffix tries required gathering relevant portions of the global data structures in a collective operation involving all the p processors. As mentioned in Section 4.5, the recursive construction of sets of modified suffixes and their compact tries require queries on the distributed array data structures. To answer such queries efficiently, we rely on all-to-all communication. Here we make the following standard assumptions on available memory and bandwidth: The number of elements that can be communicated in and out of a processor is at most $O(N/p)$ in a single round of collective communication. Given this assumption, we claim that the total number such rounds required to satisfy all ISA queries is $O(\log^{k+1} N)$ in the expected case.

First, we bound the communication costs w.r.t. the number of outgoing elements from a processor. The number of ISA queries required by a processor is equal to the number of modified suffixes (or modified reverse prefixes) it handles, which is $O((N/p) \times \log^k N)$ in expectation. As per our assumption, the number of elements that can be accommodated in a round is only $O(N/p)$. Therefore, the number of rounds of collective communication required is $O(\log^k N)$ in expectation. Bounding the communication costs w.r.t. the number of incoming elements to a processor is slightly tricky. We use the following key result [4]: *The expected length of the longest repeat within a string of length N is $O(\log N)$.* In other words, let $M = \max_{x,y \neq x} \text{lcp}(\mathsf{T}[x..], \mathsf{T}[y..])$, then the expected value of M is $O(\log N)$. Also, the lcp of any two k-modified suffixes is $\leq (k+1)M$. In order to bound the communication costs w.r.t. the number of incoming ISA queries, it is sufficient to prove the following.

Lemma 2. *For any fixed x, the number of $\text{ISA}[x]$ queries is $O(\log^{k+1} N)$ in expectation.*

Proof. Recall our recursive algorithm for creating partitions. While processing some node u (with *string-depth*(u) being t) in a trie, an $\text{ISA}[x]$ will be queried iff there exists a modified suffix (with starting position y) in the subtree of u, such that $y+t+1 = x$. The number of distinct such y's is bounded by the maximum possible value of t, which is $M(k+1)$. Additionally, the number of modified suffixes with a fixed starting position y is $\leq H^k$. By putting all together, the number of $\text{ISA}[x]$ queries is $\leq M(k+1)H^k$. By substituting $O(\log N)$ for H and M, we obtain the claimed bound. □

Using similar arguments, the number of $\text{ISA}'[x]$ queries for any fixed x can also be bounded by $O(\log^{k+1} N)$ in expectation. Therefore, the number of rounds of all-to-all communications needed is $O(\log^{k+1} N)$ in expectation.

Note that $O(N/p)$ elements are communicated between the p processors during each round of communication. Assuming that it takes a constant time to transfer an element from one processor to another, the total time taken by the communication rounds is bounded by $O\left(\frac{N}{p} \log^{k+1} N\right)$ in expectation.

4.7. Space complexity

GST is implemented as distributed SA, ISA, LCP, SA′, ISA′ and LCP′ arrays. All of these arrays require $O(N/p)$ space in each processor. Local *RMQ* and *RMQ*′ data structures take $O(N/p)$ space, while *PMN* and *PMN*′ arrays take $O(p)$ space per processor.

If all the tries constructed, described in Section 4.4, are processed simultaneously, then the space required to store all of the tries is bounded by runtime complexity, as derived in Section 4.5 without the occ term i.e., $O(N/p \cdot \log^k N)$. The term for occ is

not included because the output pairs need not be retained in main memory and can be written out to standard output or disk storage.

However, if we process the tries in multiple batches, with total size of $O(N/p)$ per batch in a processor, then the space complexity analysis follows the communication bounds discussed in Section 4.6. In this case, the space required is bounded by $O(N/p \cdot \log N)$ per processor per batch with a total of $O(\log^{k+1} N)$ batches. This also helps us to derive a bound on the number of batches required when the space available per processor is limited. If the space available per processor is bounded by $O(N/p)$, then the algorithm requires $O(\log^{k+2} N)$ batches of size $O(N/(\log Np))$ each.

5. Implementation details

We implemented our algorithm using C++ and MPI. We use block-wise distribution for the distributed SA and LCP arrays. We refer to the elements located within a processor as its 'local elements' or 'local array'.

5.1. Representation of GST by distributed data structures

We represent the generalized suffix tree (GST) of all the suffixes of the strings $S_1, \ldots, S_n$ using the following distributed data structures.

1. Distributed SA, ISA, and LCP w.r.t. T.
2. Distributed SA′, ISA′ and LCP′ w.r.t. $\overleftarrow{T}$
3. Local RMQ and RMQ′ data structures to answer range minimum queries for the local LCP and LCP′ arrays respectively, in every processor.
4. To enable distributed range minimum queries, we maintain in every processor an array of size p, *PMN*, where *PMN*[i] has the minimum value of processor i's local LCP. Similarly *PMN*′ is constructed for LCP′ array.

For SA and ISA arrays, we use 64-bit integers. The LCP array is implemented as a byte array because its entries in the worst-case are limited by the length of the reads, and high throughput sequencing reads have short lengths that can fit in a byte.

5.2. Representation of compact suffix tries

As described in Section 4.2, k-modified suffix sets are generated by a recursive construction of compact suffix tries. We represent a compact suffix trie by two arrays: (a) the sorted order of the suffixes that constitute the compact trie, and (b) LCP array, which contains the lengths of the longest common prefixes between every pair of consecutive suffixes.

5.3. Representation of internal nodes

An internal node u can be represented with the tuple $(\mathsf{sp}(u), \mathsf{ep}(u), \textit{string-depth}(u), \gamma(u))$ for both the GST and the compact suffix tries constructed at the lower levels of recursion. For an internal node u in the GST, $\gamma(u) = 0$. For an internal node in the suffix trie generated for a set C_f^h, $\gamma(u)$ is the length of the (h - 1)-modified suffix matched thus far.

To save space, we use the following representation: $(\mathsf{sp}(u), w(u), \textit{string-depth}(u), \gamma(u))$, where $w(u) = \mathsf{ep}(u) - \mathsf{sp}(u) + 1$. The advantage of this representation is that we can use a 32-bit integer for $w(u)$, even when T's size exceeds the limit of unsigned 32-bit integers. This is because for most datasets, the largest of the primary nodes includes only few tens of million suffixes (less than the size limit of a 32-bit integer). Also, both *string-depth*(u) and $\gamma(u)$ are bounded by the maximum length of the input sequence, and hence, we use a byte each for both *string-depth*(u) and $\gamma(u)$. In total, each internal node takes only 14 bytes in this representation.

5.4. Construction of distributed GST

After constructing SA and LCP using [7], we construct the ISA array corresponding to SA by an all-to-all communication of the pairs (SA[i], ISA[SA[i]]) for each SA entry. We then construct local RMQ table (an implementation of [6]) in each processor, corresponding to the local LCP array. We then construct *PMN* by gathering all processor LCP minimums. We also build a local RMQ table on the *PMN* array.

We repeat the above steps to construct SA′, ISA′, LCP′, RMQ′ and *PMN*′ w.r.t. $\overleftarrow{T}$. Not including the communication costs to construct SA and LCP, only a constant number of collective communication operations is required.

5.5. Selection of primary nodes

We select the regions of interest in SA by scanning the LCP array to select those regions where lcp values are $\geq \tau$. All primary node leaves should be from one of these regions. If a selected region straddles two processors, this region is shifted to the lower ranked processor so that any such region is completely contained within a processor. As noted earlier in Section 4.1, our assumption about the distribution of τ-length prefixes will limit the region size per processor to $O(N/p)$. Note that only SA and LCP arrays are shifted. ISA, RMQ, ISA′, and RMQ′ tables remain distributed as they were constructed.

In order to identify all the internal nodes from the selected regions, we use the all-nearest-smaller-values (ANSV) solution [2] with respect to the LCP array. Left-to-right and right-to-left ANSV solutions of the LCP array produce the left most ($\mathsf{sp}(u)$) and the right most ($\mathsf{ep}(u)$) indexes of the internal nodes respectively. After collecting the internal node tuples generated by the ANSV solution, we sort and remove the duplicate entries.

5.6. Batch processing of GST's internal nodes

After selecting the internal nodes of GST, we process these nodes in batches for two reasons. One, the memory available per processor is not large enough to accommodate all the suffix tries constructed. Two, the number of elements that can be communicated in a single all-to-all operation is limited in many MPI implementations, and hence the number of elements that can be queried and answered in distributed ISA and range minimum queries is limited. This limit depends upon the MPI implementation.

We process the internal nodes in batches of size B. We choose B to be more than the size of the largest primary node. We partition the nodes assigned to a processor into batches of size at least B and at most $2B$. Note that even if some processors complete all their batches earlier than others, they have to participate in the collective operations to answer the ISA and range minimum queries addressed to them.

In order to achieve better load balancing among batches, the batches are expected to have approximately uniform sizes across all the p processors. As described in Section 5.5, internal node tuples are identified by the ANSV solution and the duplicate entries are eliminated by sorting. Internal nodes are added to the batches in this final sorted order. There are two ways to sort these tuples — either by $(\mathsf{sp}(u), w(u))$ or by $(w(u), \mathsf{sp}(u))$. We found that sorting by $\mathsf{sp}(u)$ first produces more uniform batch sizes across processors compared to sorting by $w(u)$, and hence better run-time performance.

5.7. Construction of compact tries

Given an input set of suffixes C_f^h, construction of the compact trie is accomplished by building the corresponding SA and LCP arrays. ISA queries are used to rank the suffixes in the trie's SA, while range minimum query (RMQ) results are used to construct the trie's LCP array. We only describe these queries in relation to a C_f^h set generated during a rightward extension, i.e., modified suffix sets in generating type-0 outputs. In case of C_f^h being a part of order-h universe while processing $\mathcal{S}'_w$ (Section 4.4), we follow a similar procedure except that instead of ISA and RMQ, we use ISA′ and RMQ′.

5.7.1. SA of the compact trie

Note that C_f^h suffix sets are generated under the context of some internal node u. After generating C_f^h from an internal node u, the rank of these suffixes is queried from the distributed ISA. We preform one distributed ISA query for all suffixes of C_f^h at the current recursion level of the current batch as follows.

1. An all-to-all communication, where each processor sends all its C_f^h suffixes in the current batch as queries to the processors owning the ISA entries.
2. An all-to-all communication, where each processor returns the ISA values back to the processor from which the queries were received.

After receiving the ISA query results, we construct the trie's SA, and sort the SA entries based on their ISA rank. Since there is a constant number of all-to-all communications required for a round of suffix trie construction, communication complexity described in Section 4.6 remains valid.

5.7.2. LCP of the compact trie

After constructing the SA w.r.t. C_f^h's trie, we construct the LCP array by distributed range minimum queries as below.

1. Partition the entries of all the suffix arrays constructed in the previous step into p parts, each part corresponding to the processor owning the RMQ values of the SA entries.
2. An All-to-all communication to send the range minimum queries (i.e., SA indexes of interest).
3. An All-to-all communication to send/receive the range minimum query results. For a trie, RMQ results sent from a processor include the following:
   (a) The length of longest common prefixes between two consecutive suffixes of the trie's SA.
   (b) The minimum value between the left most (corresp. right most) SA entry in the queried processor and the left most (corresp. right most) suffix of the trie's SA entry in that processor.

   In distributed range minimum queries, the above results are sent for all the tries generated while processing the current batch.

Based on the RMQ results and the local *PMN* tables, the LCP values between two consecutive suffixes in the SA corresponding to the trie for C_f^h can be computed. The number of elements queried and answered is bounded in a similar manner to the ISA queries, and hence communication complexity remains the same.

5.8. Generation of maximal common substring pairs

After we construct the suffix tries using the procedure described in Section 5.7, we generate the valid pairs as given in Sections 4.3 and 4.4 using the k-modified suffix sets. However, in order to find which one of the $|\Sigma| + 1$ buckets a (modified) suffix belongs to, we query the distributed array T to identify the preceding character. Hence, maximal common substring pair generation adds an extra round of all-to-all communication to answer the queries against T.

Table 1
Datasets used for experiments.

	D1	D2	D3
Accession	SRR2891093	SRR2984882	SRR3169276
Type	RNA	DNA	RNA
Organism	H. sapiens	S. cerevisiae	H. sapiens
Source	HEK293T	WGS	Stem cells
Sequencer	NextSeq 500	HiSeq 2000	HiSeq 2500
No. of Reads	60,100,561	18,415,332	272,462,716
Read length	75	101	151
Total size	4.507 Gbp	1.860 Gbp	38.417 Gbp
Dataset Sizes after pre-processing			
	D1	D2	D3
No. of Reads	21,682,850	8,263,882	116,295,542
Data size	1.626 Gbp	0.835 Gbp	17.560 Gbp
Input size	3.154 Gbp	1.535 Gbp	34.318 Gbp

6. Experimental results

We ran our experiments on an Intel Xeon Infiniband Cluster. Each node has a 2.7 GHz 24-core Intel Xeon 6226 processor with 192 GB of main memory and is running RHEL7.6 operating system. The nodes of the cluster are interconnected with EDR (100 Gbps) InfiniBand. Experiments were conducted on up to 64 nodes using 16 cores per node, totaling 1,024 cores. We evaluated our algorithm on three different datasets, detailed in Table 1. Dataset D2 (NCBI SRA Accession Number SRR2984882) consists of 18.4 million reads, sampled from the genome of Yeast (S. cerevisiae). Datasets D1 (Accession Number SRR2891093) and D3 (Accession Number SRR3169276) are human RNA-Seq datasets, which are randomly sampled from expressed portions of the genome (RNA sequences produced by the genome). The whole genome dataset is sampled uniformly at random over the entire length of the genome. While the sampling is uniformly random over RNA sequences too, the frequency of each RNA sequence itself is proportional to the expression of the corresponding gene. Hence, these datasets constitute a highly non-uniform sampling of the underlying genomic space. The datasets also cover three different Illumina NGS sequencers — HiSeq 2000, HiSeq 2500 and the desktop sequencer NextSeq 500.

Reads may be redundant in the sense that a read may be fully contained in another. Clearly, such a contained read shares potentially the same overlaps with other reads as the read containing it. Hence, including the contained reads in the input is not useful. It also significantly increases the output size and runtime without adding any additional value. Hence, we developed a pre-processing algorithm to eliminate all redundant reads. Also, RNA-Seq reads are characterized by consecutive A's at the end, covering the poly-A tail of the mRNA molecule. Common substrings that contain these have no biological relevance, hence we trim the poly-A tail from the reads to avoid generating spurious output pairs.

Note that the input size is approximately twice that of the dataset size. This is because DNA is double stranded, with the two strands being reverse complements (under $A \leftrightarrow T$, $C \leftrightarrow G$ substitution) of each other. The input for each sequence consists of any one of its strands. To correctly infer a maximal common substring between two sequences given as complementary strands, the input must take both forms of the sequence into account, thus

Table 2
Runtime in seconds vs No. of cores for D1 with $\tau = 15$, 17 and $\tau = 19$.

$\tau = 15$				
No. of cores	$k = 1$		$k = 2$	
	Runtime (sec)	Relative speedup	Runtime (sec)	Relative speedup
64	947.81	1.00X	12005.6	1.00X
128	587.38	1.62X	7304.82	1.64X
256	280.21	3.38X	3428.42	3.50X
512	154.15	6.14X	1953.64	6.18X
1024	114.24	8.29X	1285.06	9.34X
	$\tau = 17$			
No. of cores	$k = 1$		$k = 2$	
	Runtime (sec)	Relative speedup	Runtime (sec)	Relative speedup
64	849.78	1.00X	10463.70	1.00X
128	532.09	1.59X	6828.73	1.53X
256	254.72	3.34X	2975.68	3.52X
512	141.88	5.98X	1699.56	6.16X
1024	102.05	8.32X	1097.10	9.53X
	$\tau = 19$			
No. of cores	$k = 1$		$k = 2$	
	Runtime (sec)	Relative speedup	Runtime (sec)	Relative speedup
64	792.48	1.00X	9229.78	1.00X
128	423.32	1.87X	4766.11	1.93X
256	236.61	3.35X	2639.18	3.49X
512	124.49	6.36X	1388.58	6.64X
1024	91.02	8.71X	991.36	9.31X

Table 3
Runtime in seconds vs. No. of cores for D2 with $\tau = 20$ and $\tau = 22$.

$\tau = 20$				
No. of cores	$k = 1$		$k = 2$	
	Runtime (sec)	Relative speedup	Runtime (sec)	Relative speedup
64	624.67	1.00X	11506.40	1.00X
128	359.86	1.73X	6449.24	1.78X
256	189.89	3.29X	3688.41	3.11X
512	108.76	5.74X	2082.66	5.52X
1024	87.72	7.12X	1580.13	7.28X
	$\tau = 22$			
No. of cores	$k = 1$		$k = 2$	
	Runtime (sec)	Relative speedup	Runtime (sec)	Relative speedup
64	593.01	1.00X	10305.10	1.00X
128	341.07	1.74X	5750.79	1.79X
256	186.95	3.17X	3328.68	3.09X
512	103.22	5.74X	1887.19	5.46X
1024	86.85	6.82X	1526.13	6.75X

doubling its size. However, after removing duplicates that were introduced by the addition of reverse complement sequences, the input size is slightly less than that of the dataset size.

The value of τ depends upon the application, error rate of the sequencing machine, and the read length. To demonstrate the scalability of our proposed algorithm, we ran experiments with multiple values of τ and k. For dataset D1, we ran experiments with $\tau = 15$, $\tau = 17$ and $\tau = 19$. For the longer read datasets D2 and D3, we used $\tau = 20, 22$ and $\tau = 30$ respectively. For all the chosen values of τ, we found that the largest subtree under an internal node at depth τ or greater is significantly smaller than N/p, validating the assumption used in deriving the run-time complexity. Another interesting observation is that between ≈15% and 35% of the leaves do not belong to any primary nodes (i.e., they are in the subtrees of internal nodes with *string-depth* less than τ). These are not considered for processing by the algorithm, hence eliminated.

The run-times for dataset D1 as a function of the number of processor cores are shown in Table 2. To illustrate the run-time behavior of the algorithm, the times are shown for $k = 1$ and $k = 2$, and for three different values of τ. The run-time grows exponentially with k, and this behavior is reflected in the significant increase in run-time for $k = 2$ over $k = 1$. However, given the improving error rates of sequencing machines, applications typically use very small values of k. As the value of τ increases, the number of k-mismatch maximal common substrings that satisfy the length criterion ($\phi \geq (k+1)\tau$) decreases, thus reducing the run-time.

The run-times for all parameter choices of k and τ, as the number of processor cores is varied from 64 to 1,024, demonstrate reasonably good, but not perfect, scaling. To better illustrate the scaling, the relative speedups when compared to the run-time on 64 cores are also shown. For $k = 1$, the relative speedup on 1,024 cores ranged from 8.29X to 8.71X. This slightly improves to 9.29–9.34X, for $k = 2$.

A similar set of experimental observations for dataset D2 with $\tau = 20$ and $\tau = 22$ are shown in Table 3. The reason for less than ideal speedup is due to the imbalance in the work assigned to the processor cores. While we make sure that every batch has approximately same size in our batch-wise processing, the distribution of internal node sizes is not uniform. We profiled the runs of datasets D1 (with $\tau = 15$) and D2 (with $\tau = 20$) using the `hpctoolkit` software [1]. `hpctoolkit` uses the CPU TIME timer on Linux to profile the code and sampling at a frequency of 200 samples per second per process. Using the profiler results, we compute, among the total CPU time, the total time spent by all processes waiting for the results of their distributed ISA and LCP queries (Table 4). This time can serve as a proxy for the imbalance in the work assigned to a processor. As the number of cores increases, we note that there is significant increase in the waiting time, while the total non-waiting time remains approximately the same except for $k = 1$ with 1024 cores. In this case, the additional time is due to the communication costs during the construction of GST.

Even though the input size for D1 is larger than D2, Fig. 5 shows that the ratio of time taken for $k = 2$ to the time taken for $k = 1$ is smaller for D1 compared to D2. For D1, $k = 2$ takes 10–12X longer than $k = 1$, while it is 17–20X longer for D2. This is due to difference in coverage of the underlying sampled space of the genome by the reads, in the respective datasets. D2 is a whole genome sequencing dataset for Yeast with a genome length of $\approx 12.1 \times 10^6$ base pairs. Coverage of the genome by the reads, defined as the ratio of the total length of the reads (N) to the length of the genome, is ≈68X.

While it is not customary to define coverage for RNA-Seq datasets as it varies for each expressed portion depending on the rate of expression, we can use the average of these individual coverages as an indicator of the depth of sampling. The total coding region within the human genome ($\approx 3.3 \times 10^9$) is <2%, which is sampled in RNA-seq datasets. This leads to an average coverage of ≈24X coverage for dataset D1. The lower the coverage, the fewer the number of overlaps between input sequences. Therefore, the height of GST is likely to be smaller, and hence, faster run-time. This particular difference in the datasets also has an effect on speedup, i.e., D1 exhibits better scaling than D2.

For $k = 3$, the run-times are too long to record observations on fewer cores, underscoring the importance of the parallel algorithm. Table 5 shows the run times on 1,024 cores for datasets D1 and D2. As we can see from the third column in Table 5, the ratio of run-times for $k = 3$ and $k = 2$ is roughly in line with the increase from $k = 2$ to $k = 1$. This reflects the exponential increase in run-time as a function of k.

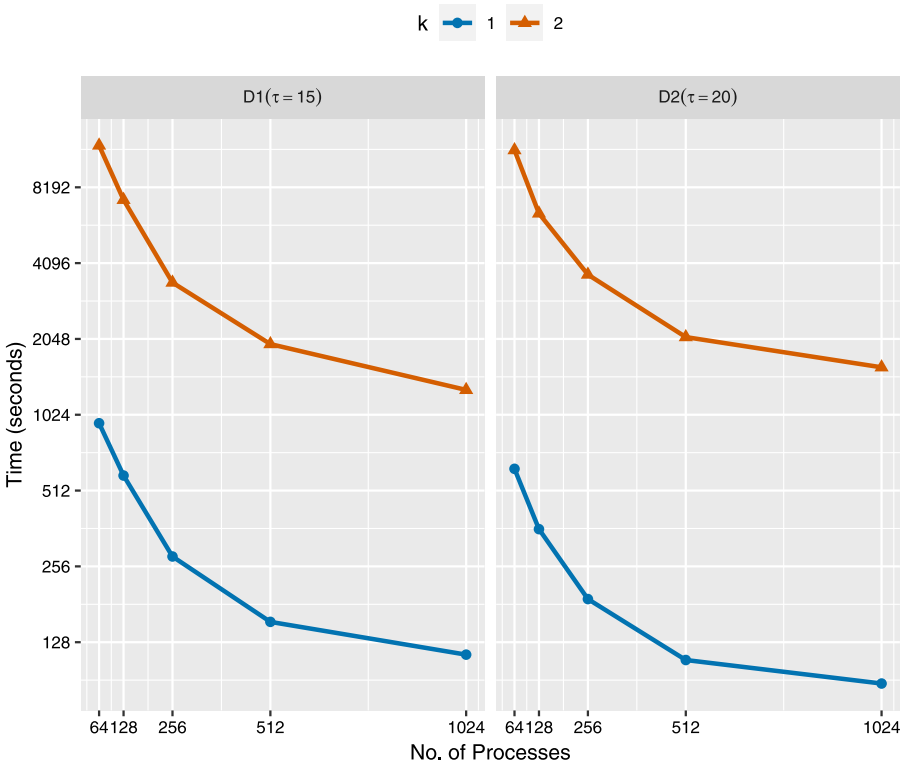

Fig. 5. Runtime vs No. of cores for datasets D1 and D2 with $\tau = 15$ and $\tau = 20$ respectively.

Table 4
Profiling for datasets D1 and D2 with $\tau = 15$ and $\tau = 20$. Total CPU time, query waiting time and the non-waiting time are presented in micro seconds.

D1 with $\tau = 15$						
No. of cores	$k = 1$			$k = 2$		
	Total CPU time (μs)	Query waiting time (μs)	Non-waiting time (μs)	Total CPU time (μs)	Query waiting time (μs)	Non-waiting time (μs)
64	5.9×10^{10}	6.9×10^{9}	5.2×10^{10}	7.5×10^{11}	2.0×10^{11}	5.5×10^{11}
128	7.0×10^{10}	2.0×10^{10}	4.9×10^{10}	8.3×10^{11}	2.8×10^{11}	5.4×10^{11}
256	7.7×10^{10}	2.7×10^{10}	5.0×10^{10}	9.5×10^{11}	4.0×10^{11}	5.5×10^{11}
512	8.6×10^{10}	3.1×10^{10}	5.4×10^{10}	1.0×10^{12}	4.9×10^{11}	5.5×10^{11}
1024	1.5×10^{11}	8.6×10^{10}	6.4×10^{10}	1.7×10^{12}	1.1×10^{12}	5.7×10^{11}
	D2 with $\tau = 20$					
No. of cores	$k = 1$			$k = 2$		
	Total CPU time (μs)	Query waiting time (μs)	Non-waiting time (μs)	Total CPU time (μs)	Query waiting time (μs)	Non-waiting time (μs)
64	4.3×10^{10}	1.0×10^{10}	3.2×10^{10}	8.0×10^{11}	3.4×10^{11}	4.6×10^{11}
128	4.6×10^{10}	1.4×10^{10}	3.1×10^{10}	9.8×10^{11}	5.2×10^{11}	4.6×10^{11}
256	5.5×10^{10}	2.1×10^{10}	3.4×10^{10}	1.1×10^{12}	7.1×10^{11}	4.6×10^{11}
512	6.7×10^{10}	2.8×10^{10}	3.8×10^{10}	1.4×10^{12}	9.2×10^{11}	4.7×10^{11}
1024	2.2×10^{11}	1.7×10^{11}	4.9×10^{10}	2.1×10^{12}	1.6×10^{12}	4.8×10^{11}

Table 5
Runtime for $D1$ and $D2$ with $k = 3$ and $p = 1024$.

Dataset	τ	Time (sec)	Runtime for $k = 3$ / Runtime for $k = 2$
D1	15	18535.0	14.42
D1	17	14898.2	13.57
D1	19	12207.5	12.37
D2	20	38241.7	24.20
D2	22	33502.4	21.29

In order to push the limits of our algorithm, we ran dataset D3 containing over 270 million reads with longer read length of 151, whose total size is an order of magnitude larger than D1. For $k = 1$, we were able to process such a large dataset in under 1.5 h (50001.29s) on 1,024 cores.

7. Conclusion

Approximate sequence matching algorithms are of significant interest in computational biology as replacement for quadratic alignment-based algorithms, particularly as high-throughput sequencers are producing large-scale datasets. In this paper, we presented a parallel algorithm for finding k-mismatch, all-pair maximal common substrings between a large set of strings. While the only sub-quadratic sequential algorithm to solve this problem is the one recently proposed by [22], there is no parallel algorithm

to solve this problem to date. Our work achieves an expected parallel run-time complexity of $O\left(\left(\frac{N}{p}\log N + \text{occ}\right)\log^k N\right)$, where occ is the number of such reported maximal common substrings. We note that for $k = 0$ this reflects the best possible run-time for computing exact all-pair maximal common substrings, while the run-time degrades slowly by a factor of $\log n$ for each additional error tolerated.

We also present our algorithm for the practical distributed memory model of parallel computation, and demonstrate its performance on real, large-scale datasets. While the scaling results are constrained by the size of the parallel computer available to us, we see no difficulty for the algorithm to scale beyond the 1,024 cores it is demonstrated on. The algorithm is useful in identifying overlaps between Illumina sequencer reads which typically contain a small rate of substitution errors. As these sequencers account for over 90% of DNA and RNA sequencing worldwide, the algorithm could have significant impact. We are currently exploring the use of this algorithm for applications in genome mapping and assembly.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This research is supported in part by the U.S. National Science Foundation under IIS-1416259, CCF-1704552 and CCF-1703489.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, N.R. Tallent, HPCToolkit: Tools for performance analysis of optimized parallel programs, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 685–701.
[2] O. Berkman, B. Schieber, U. Vishkin, Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values, J. Algorithms 14 (3) (1993) 344–370.
[3] S. Burkhardt, J. Kärkkäinen, Better filtering with gapped q-grams, Fund. inform. 56 (1–2) (2003) 51–70.
[4] L. Devroye, W. Szpankowski, B. Rais, A note on the height of suffix trees, SIAM J. Comput. 21 (1) (1992) 48–53.
[5] M. Farach-Colton, P. Ferragina, S. Muthukrishnan, On the sorting-complexity of suffix tree construction, J. ACM 47 (6) (2000) 987–1011.
[6] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, Springer, 2007, pp. 459–470.
[7] P. Flick, S. Aluru, Parallel distributed memory construction of suffix and longest common prefix arrays, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 16.
[8] M.H.-Y. Fritz, R. Leinonen, G. Cochrane, E. Birney, Efficient storage of high throughput DNA sequencing data using reference-based compression, Genome Res. 21 (5) (2011) 734–740.
[9] M.G. Grabherr, B.J. Haas, M. Yassour, J.Z. Levin, D.A. Thompson, I. Amit, X. Adiconis, L. Fan, R. Raychowdhury, Q. Zeng, et al., Full-length transcriptome assembly from RNA-Seq data without a reference genome, Nature Biotechnol. 29 (7) (2011) 644–652.
[10] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[11] A. Kalyanaraman, S. Aluru, S. Kothari, V. Brendel, Efficient clustering of large EST data sets on parallel computers, Nucleic Acids Res. 31 (11) (2003) 2963–2974.
[12] G. Kucherov, D. Tsur, Improved filters for the approximate suffix-prefix overlap problem, in: International Symposium on String Processing and Information Retrieval, Springer, 2014, pp. 139–148.
[13] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (5) (1993) 935–948.
[14] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.
[15] M.L. Metzker, Sequencing technologies – the next generation, Nat. Rev. Genet. 11 (1) (2010) 31–46.
[16] K. Nakamura, T. Oshima, T. Morimoto, S. Ikeda, H. Yoshikawa, Y. Shiwa, S. Ishikawa, M.C. Linak, A. Hirai, H. Takahashi, et al., Sequence-specific error profile of Illumina sequencers, Nucleic Acids Res. 39 (13) (2011) e90.
[17] O. Sakarya, H. Breu, M. Radovich, Y. Chen, Y.N. Wang, C. Barbacioru, S. Utiramerur, P.P. Whitley, J.P. Brockman, P. Vatta, et al., RNA-Seq mapping and detection of gene fusions with a suffix array algorithm, PLoS Comput. Biol. 8 (4) (2012) e1002464.
[18] A. Sarje, S. Aluru, All-pairs computations on many-core graphics processors, Parallel Comput. 39 (2) (2013) 79–93.
[19] T.E. Scheetz, N. Trivedi, K.T. Pedretti, T.A. Braun, T.L. Casavant, Gene transcript clustering: a comparison of parallel approaches, Future Gener. Comput. Syst. 21 (5) (2005) 731–735.
[20] J.T. Simpson, R. Durbin, Efficient de novo assembly of large genomes using compressed data structures, Genome Res. 22 (3) (2012) 549–556.
[21] J.T. Simpson, R. Durbin, Efficient de novo assembly of large genomes using compressed data structures, Genome Res. 22 (3) (2012) 549–556.
[22] S.V. Thankachan, C. Aluru, S.P. Chockalingam, S. Aluru, Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis, in: International Conference on Research in Computational Molecular Biology, Springer, 2018, pp. 211–224.
[23] N. Välimäki, S. Ladra, V. Mäkinen, Approximate all-pairs suffix/prefix overlaps, Inform. and Comput. 213 (2012) 49–58.
[24] P. Weiner, Linear pattern matching algorithms, in: Switching and Automata Theory, 1973, pp. 1–11.

Sriram P. Chockalingam is a research scientist in the Institute for Data Engineering and Science at the Georgia Institute of Technology, Atlanta, GA. He received his Ph.D. degree in Computer Science and Engineering from Indian Institute of Technology Bombay, India. His research interests include parallel algorithms, approximate sequence matching and systems biology.

Sharma V. Thankachan is an Assistant Professor in the Department of Computer Science at University of Central Florida, Orlando. He received his Ph.D. degree in Computer Science from Louisiana State University in 2014. Prior to that, received his B. Tech. degree in Electrical and Electronics Engineering from National Institute of Technology Calicut, India in 2006. His research interests include parallel algorithms, computational biology and succinct data structures.

Srinivas Aluru is the Executive Director of the Georgia Tech Interdisciplinary Research Institute (IRI) in Data Engineering and Science (IDEaS) and a professor in the School of Computational Science and Engineering within the College of Computing. He co-leads the NSF South Big Data Regional Innovation Hub which nurtures big data partnerships between organizations in the 16 Southern States and Washington D.C., and the NSF Transdisciplinary Research Institute for Advancing Data Science. Aluru conducts research in high performance computing, data science, bioinformatics and systems biology, combinatorial scientific computing, and applied algorithms. He pioneered the development of parallel methods in computational biology, and contributed to the assembly and analysis of complex plant genomes. His contributions in scientific computing lie in parallel Fast Multipole Method, domain decomposition methods, spatial data structures, and applications in computational electromagnetics and materials informatics. Aluru serves on the editorial boards of the *IEEE Transactions on Big Data*, *ACM/IEEE Transactions on Computational Biology and Bioinformatics*, the *Journal of Parallel and Distributed Computing*, and the *International Journal of Data Mining and Bioinformatics*. He is a Fellow of the American Association for the Advancement of Science (AAAS) and the Institute for Electrical and Electronic Engineers (IEEE).