RACE: A Scalable and Elastic Parallel System for Discovering Repeats in Very Long Sequences

Essam Mansour
King Abdullah University of Science and Technology
essam.mansour@kaust.edu.sa

Ahmed El-Roby
School of Computer Science, University of Waterloo
aelroby@uwaterloo.ca

Panos Kalnis
King Abdullah University of Science and Technology
panos.kalnis@kaust.edu.sa

Aron Ahmadia
Department of Statistics, Columbia University
aron@ahmadia.net

Ashraf Aboulnaga
Qatar Computing Research Institute (QCRI)
aaboulnaga@qf.org.qa

ABSTRACT

A wide range of applications, including bioinformatics, time series, and log analysis, depend on the identification of repetitions in very long sequences. The problem of finding maximal pairs subsumes most important types of repetition-finding tasks. Existing solutions require both the input sequence and its index (typically an order of magnitude larger than the input) to fit in memory. Moreover, they are serial algorithms with long execution time. Therefore, they are limited to small datasets, despite the fact that modern applications demand orders of magnitude longer sequences.

In this paper we present RACE, a parallel system for finding maximal pairs in very long sequences. RACE supports parallel execution on stand-alone multicore systems, in addition to scaling to thousands of nodes on clusters or supercomputers. RACE does not require the input or the index to fit in memory; therefore, it supports very long sequences with limited memory. Moreover, it uses a novel array representation that allows for cache-efficient implementation. RACE is particularly suitable for the cloud (e.g., Amazon EC2) because, based on availability, it can scale elastically to more or fewer machines during its execution. Since scaling out introduces overheads, mainly due to load imbalance, we propose a cost model to estimate the expected speedup, based on statistics gathered through sampling. The model allows the user to select the appropriate combination of cloud resources based on the provider's prices and the required deadline. We conducted extensive experimental evaluation with large real datasets and large computing infrastructures. In contrast to existing methods, RACE can handle the entire human genome on a typical desktop computer with 16GB RAM. Moreover, for a problem that takes 10 hours of serial execution, RACE finishes in 28 seconds using 2,048 nodes on an IBM BlueGene/P supercomputer.

Proceedings of the VLDB Endowment, Vol. 6, No. 10

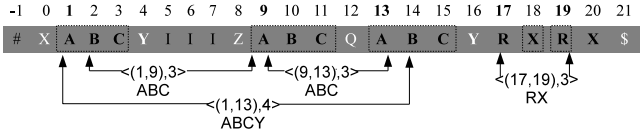


Figure 1: Input string S. # and $ are the start and end symbols, respectively. $\langle(1,9),3\rangle$ is a maximal pair of length 3 corresponding to maximal repeat ABC at positions 1 and 9. $\langle(1,13),3\rangle$ is *not* a maximal pair because repeat ABC at positions 1 and 13 can be extended to ABCY, which corresponds to $\langle(1,13),4\rangle$.

1. INTRODUCTION

Given string S, a *maximal repeat* is a substring s that appears at least twice in S and is maximal (i.e., it cannot be extended to the left nor right). In the example of Figure 1, ABC is a maximal repeat because it appears at positions 1 and 9, and the corresponding symbols at the left (i.e., X, Z) and right (i.e., Y, Q) are different. A maximal repeat can appear multiple times in S. Each pair of appearances is a *maximal pair*. For example, $\langle(1,9),3\rangle$ is a maximal pair for maximal repeat ABC of length 3 at positions 1 and 9.

In practice S can be any sequence, such as historical market data, where each symbol represents a stock trade; or web logs, where symbols represent actions (e.g., clicking a URL). The set of maximal pairs is a superset of other interesting repetitive structures, such as the supermaximal and tandem repeats [9]. Hence, computing maximal pairs is crucial in a variety of applications, such as time series [23], identification of common functionality in business processes [5], text summarization [15], compression [18], and bioinformatics [24].

The most widely adopted algorithm for finding maximal pairs was proposed by Gusfield [9] and is based on suffix trees [19]. A suffix tree is a trie that indexes all suffixes of S. The algorithm traverses the entire tree top-down and then returns recursively from the leaves to the root, generating maximal pairs along the way. The time complexity is $\mathcal{O}(|\Sigma||S| + |z|)$, where Σ is the alphabet and z is the set of all maximal pairs in S. The value of $|z|$ can be quadratic to the input size, and much larger than $|\Sigma|$ and $|S|$. Hence, $|z|$ is the dominant factor. Later approaches [1, 14, 16, 22] utilize different index structures and some of them reduce

complexity to $\mathcal{O}(|z|)$.

Existing approaches have two limitations: (*i*) They require both the sequence and the index structure to be in memory, hence they are limited to small sequences. Consider sequence alignment, an important tool in bioinformatics, which can be solved efficiently by concatenating DNA sequences of different species and computing the maximal pairs [10]. The human genome alone contains 2.6G symbols; together with its suffix tree, it needs 35GB to 68GB of RAM, depending on the implementation. If multiple sequences are concatenated, we need 100's GB to TB of RAM. Similar problems exist in other domains such as analysis of stock market data, or web logs. (*ii*) Existing approaches support only serial execution on a single machine. Even if enough memory is available in one machine, the execution time can be prohibitively long due to the large result size.

In this paper we propose RACE, a parallel system for computing maximal pairs in very long sequences using limited memory. RACE decomposes the problem into independent tasks that can number in the tens of thousands, and that require minimal coordination. Therefore, it achieves good load balance with limited communication cost, and scales in the cloud (e.g., Amazon EC2) or other massively parallel infrastructures. For example we have deployed our system on 2,048 nodes of an IBM BlueGene/P supercomputer. RACE implements disk-based structures and does not need the input sequence S or its suffix tree to fit in memory. Therefore, it can support very long sequences on machines with limited memory. For example, we managed to process multi-gigabyte long sequences on a typical desktop computer with only 16GB RAM.

RACE is suitable for the cloud because it supports *elasticity*. A master node maintains a pool of pending tasks that can be assigned dynamically to more or fewer machines, if needed. However, for a specific problem size and computing infrastructure, using resources beyond a threshold does not pay off (i.e., the rate of speedup decreases), mainly because the system cannot achieve load balance. Motivated by this, we developed a model to estimate the expected speedup, based on statistics gathered from a sample of the workload. If the prices of the cloud provider are given, our model can be used to estimate how many machines should be rented to meet the user's deadline and budget. Alternatively, our model can decide whether it pays off to rent a few large (in terms of memory and CPU) and expensive machines, instead of more, smaller and cheaper ones.

In contrast to existing methods that navigate the suffix tree top-down, RACE follows a bottom-up approach that has two advantages: (*i*) Supports efficiently complex filtering criteria such as periodic multi-ranges (e.g., compute the maximal pairs of stock market data only on Mondays during the past decade), without needing to re-index the input sequence. (*ii*) Allows RACE to represent the tree as a set of arrays, instead of nodes with pointers to children. With the array representation, tree traversal is implemented iteratively as a series of joins between the current level and the level above. By not using pointers, our approach minimizes random memory accesses and is more *cache-efficient*, which is important for performance on modern multicore systems [21]; our experiments show 2 to 5 times improvement.

Our contributions are:

- We are the first to develop a parallel method, called RACE, for computing maximal pairs. Our method scales from stand-alone multicore machines to massively parallel supercomputers. In particular, RACE can scale elastically on the cloud.
- We implement a cache-efficient representation of the index, suitable for modern architectures; and we support disk-based suffix trees, allowing RACE to scale to very long sequences using limited memory.
- We propose a cost model to estimate the speedup and the expected financial cost on various combinations of cloud computing infrastructure.
- We evaluated our system with large real datasets from different application domains. RACE can process the entire human genome on a single machine with 16GB RAM. Also, it can solve a problem whose serial execution takes more than 10 hours, in 28 seconds using 2,048 nodes on an IBM BlueGene/P supercomputer.

The rest of this paper is organized as follows: Section 2 discusses the related work. Section 3 introduces essential background. RACE and its cache-efficient implementation are introduced in Section 4, whereas Section 5 discusses elasticity and the cost model for the cloud. Section 6 presents our evaluation and Section 7 concludes the paper.

2. RELATED WORK

This section presents a comparative analysis of the most important maximal pairs algorithms; Table 1 summarizes the comparison. Based on the application domain, the set of maximal pairs can be restricted by different criteria, such as *repeat length*, *bounded gap*, and *range*. Repeat length filters out maximal repeats that are below a user-specified length. Bounded gap restricts the distance between the two occurrences of the pair to a minimum width. Range confines the search for maximal pairs in a specific region(s) of the string. The existing algorithms do not support efficient range filtering as they are based on depth-first traversal of a suffix tree, but the leaves of the tree are not ordered according to the input string. The table also indicates whether the method is disk-based, which is necessary for long inputs; and whether the method is parallel, which is important as maximal pairs extraction is computationally expensive.

Gusfield's algorithm [9] is based on suffix trees; details are given in the next section. Traversing the suffix tree requires random accesses to both the input string and the tree, because logically connected nodes are not physically close in memory. Therefore, locality is poor and the number of cache misses is large. The complexity of this algorithm depends on the alphabet size $|\Sigma|$. It needs $\mathcal{O}(|\Sigma||S| + |z|)$ time to find the maximal pairs of a string of length $|S|$, where z is the set of maximal pairs. Gusfield's algorithm does not support filtering and is not parallelized. Brodal et al. [6] extended Gusfield's algorithm to finding maximal pairs under the bounded gap constraint. Their method computes the maximal pairs in $\mathcal{O}(|\Sigma||S| \log |S| + |z|)$. Another method, called REPuter [14], implemented Gusfield's algorithm using a space-efficient suffix tree structure. In some cases REPuter may improve locality. Nevertheless, it still requires random accesses to the string and the tree during traversal.

A different implementation of Gusfield's algorithm uses enhanced suffix arrays (*eSAs*) [1]. eSAs reduce the space requirements compared to the suffix tree. Also, they are processed in sequential order, which leads to better locality

Table 1: Comparison of the most important algorithms for finding maximal pairs.

	Data structure	Extra index	Complexity	Length	Gap	Range	Parallel	Disk-based
Gusfield's [9]	tree		$\mathcal{O}(\lvert\Sigma\rvert\lvert S\rvert + \lvert z\rvert)$					
Brodal's [6]	tree		$\mathcal{O}(\lvert\Sigma\rvert\lvert S\rvert \log \lvert S\rvert + \lvert z\rvert)$		✓			
REPuter [14]	tree		$\mathcal{O}(\lvert\Sigma\rvert\lvert S\rvert + \lvert z\rvert)$					
eSAs [1]	arrays		$\mathcal{O}(\lvert S\rvert + \lvert z\rvert)$					
Stehr's [22]	arrays	✓	$\mathcal{O}(\lvert z\rvert)$		✓			
R3 [16]	arrays	✓	$\mathcal{O}(\lvert z\rvert)$	✓				
Vmatch [13]	arrays		$\mathcal{O}(\lvert S\rvert + \lvert z\rvert)$	✓	✓			
RACE [our's]	arrays		$\mathcal{O}(\lvert S\rvert + \lvert z\rvert)$	✓	✓	✓	✓	✓

than the previous algorithms. However, there is still random access of the input string during processing. The method did not implement any filtering. Another approach, called Vmatch [13] also utilized eSAs but added filtering based on repeat length and bounded gap. Note that, when using eSAs, the time complexity does not depend on the alphabet size. If extra indices are also constructed, as in the case of Stehr's algorithm [22] and R3 [16], complexity is reduced to $\mathcal{O}(|z|)$. Stehr's algorithm supports bounded gap filtering, whereas R3 facilitates filtering based on repeat length.

The main drawback of existing approaches based on both suffix trees and eSAs is the restriction to small sequences since they are memory resident. Even if there was enough memory (i.e., in the order of hundreds of GBs to TBs of RAM), none of the above mentioned methods is suitable for large-scale parallelization, which is necessary in order to process large sequences. In contrast, our approach (RACE) utilizes a disk-based suffix tree and, to the best of our knowledge, is the first parallel method for finding maximal pairs. Therefore, it scales to very long sequences. Moreover, RACE supports filtering efficiently based on length, gap and range.

Recently, RepMaestro [3], Beller et al. [4] and Kulekci et al. [12] proposed methods for finding various types of maximal *repeats* using disk-based indices (note that none of these methods have been parallelized). Both Beller's and Kulekci's algorithms have time complexity $\mathcal{O}(|S| \log |S|)$. In contrast, finding maximal repeats is a less complex problem than finding maximal pairs, because there can be at most $|S|$ maximal repeats in a sequence S, but there can be up to $|S|^2$ maximal pairs [9]. Therefore, in the worst case, any maximal pairs algorithm will be bounded by $\mathcal{O}(|S|^2)$. Note that the set of maximal pairs is a subset of the Cartesian product of appearances of the maximal repeats, since there are combinations of appearances of maximal repeats that do not qualify as maximal pairs. For example, in Figure 1, maximal repeat `ABC` appears at positions 1, 9 and 13, but $\langle(1, 13), 3\rangle$ is *not* a maximal pair. Computing efficiently the qualifying combinations is not straight-forward, which is the exact reason for developing the specialized maximal pairs computation methods described above.

3. BACKGROUND

Let S denote an input sequence of length $|S|$ and alphabet Σ. We assume S starts with `#` and ends with `$`, where $\{\texttt{\#}, \texttt{\$}\} \notin \Sigma$. A maximal *pair* is denoted as a tuple of three integers $\langle(p_1, p_2), len\rangle$, where p_1 and p_2 are two occurrences of a maximal *repeat*, whose length is *len*; as shown in Figure 1.

3.1 Suffix Trees and Disk-Based Indexing

A *suffix tree* $\mathcal{T}$ is a trie that indexes all suffixes in S. $\mathcal{T}$ contains $|S|+1$ leaf nodes ranging from v_0 to $v_{|S|}$. The leaves are not at the same level nor sorted. The concatenation of the edge labels on the path from the root to a node is called *path label*. A node, whose path label is ω, is denoted as $\overline{\omega}$. Figure 2 depicts two subtrees for S that index all suffixes starting with `ABC` and `BC`, respectively. A suffix S_i is the substring $S[i]S[i+1]\cdots\$$, where $0 \leq i \leq |S|$. For each suffix S_i there is a leaf node v_i whose path label is S_i. For example, in Figure 2, leaf node v_{14} represents suffix $S_{14} =$ `BCYRXRX$`.

Each internal node of the suffix tree has at least two children. Assuming that the path label of an internal node is ω, the path label of each child extends ω by at least one character, such as $\overline{\omega c}$ and $\overline{\omega g}$, where $c \neq g$. As we will explain in the next section, this is an important property, because it guarantees that each internal node is *right diverse*.

Early suffix tree construction algorithms assumed that both the input string and the resulting suffix tree could fit in memory. Such methods are impractical because the suffix tree is more than an order of magnitude larger than the input string, meaning that even a medium sized input (i.e., a few GBs) would require hundreds of GBs of RAM. For this reason current suffix tree construction methods [8, 11, 20] are disk-based and allow both the input string and the resulting tree to be much larger than the available memory.

In this paper, we utilize ERa [17] to construct suffix trees. ERa generates a set of variable length prefixes that decompose the suffix tree into subtrees, each of which fits in memory. For example, in Figure 2, `ABC` and `BC` are the prefixes; the corresponding subtrees are the ones rooted at nodes u_1 and u_3, respectively. The advantage of such decomposition is that each subtree can be processed independently allowing for large scale parallelism, assuming that the shorter maximal pair is at least as long as the longest prefix[1].

3.2 Finding Maximal Pairs Using Suffix Trees

Gusfield's algorithm [9] is the core algorithm for calculating maximal pairs using suffix trees. The algorithm traverses the tree top-down to find repeats that are right- and left-diverse. A repeat is right-diverse if it cannot be extended to the right. For example, in Figure 2, `ABCY` is right-diverse because the symbols at positions 5 and 17 are `I` and `R`, respectively, so `ABCY` cannot be extended to the right. Left-diverse repeats are defined in a similar way.

By default, all internal nodes of a suffix tree are right-diverse (see Section 3.1). Therefore, Gusfield's algorithm needs to find left-diverse repeats. To do so, the algorithm first finds the left character for every leaf $\overline{\nu}_i$. Recall that $\overline{\nu}_i$ corresponds to suffix S_i starting at position i in the string. Let $c \in \Sigma$ be a character and define $f(\cdot)$ as follows:

$$f(\overline{\nu}_i, c) = \begin{cases} \{i\} & \text{if } c = S[i-1] \\ \varnothing & \text{otherwise} \end{cases} \quad (1)$$

[1]In most cases the condition is satisfied, because short repeats are very frequent, so they are useless in practice.

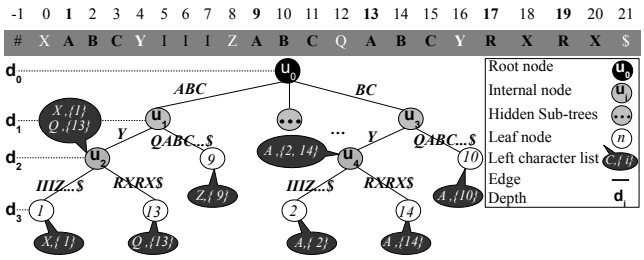


Figure 2: The sub-trees of prefixes ABC and BC in S.

Intuitively, $f(\cdot)$ returns i if the left character of $\overline{\nu}_i$ is c. In the example of Figure 2, $\overline{\nu}_{13}$ corresponds to S_{13} = ABC...$ and the character at the left is $S[12] = Q; therefore $f(\overline{\nu}_{13}, Q) = 13$. Similarly, $f(\overline{\nu}_2, A) = 2$ and $f(\overline{\nu}_{14}, A) = 14$.

The output of $f(\cdot)$ is used by the internal nodes to group together suffixes with the same left character. Let $\overline{\omega}$ be an internal node and $\overline{\omega}_1, \ldots, \overline{\omega}_k$ be its children. From the construction of the tree, a node can have at most as many children as the alphabet size, therefore $1 < k \leq |\Sigma|$. Define $f(\cdot)$ for internal nodes as follows:

$$f(\overline{\omega}, c) = \bigcup_{j=1}^{k} f(\overline{\omega}_j, c); 1 < k \leq |\Sigma| \quad (2)$$

We have two forms of $f(\cdot)$: Equation 1, which is used for leaf nodes, and Equation 2, which is applied on internal nodes and groups the results of the children by left character. For example, internal node $\overline{u_4}$ has two children: $\overline{\nu}_2$ and $\overline{\nu}_{14}$. As mentioned before, $f(\overline{\nu}_2, A) = 2$ and $f(\overline{\nu}_{14}, A) = 14$. By grouping these together, we get: $f(\overline{u_4}, A) = \{2, 14\}$. Let us now define formally the notion of left-diversity:

DEFINITION 1 (LEFT-DIVERSITY). *An internal node $\overline{\omega}$ of a suffix tree is left-diverse, if the subtree rooted at $\overline{\omega}$ contains at least two leaves with different left characters.*

For example, internal node $\overline{u_2}$ is left-diverse because the left characters of its children (i.e., $\overline{\nu}_1$ and $\overline{\nu}_{13}$) are X and Q, respectively. On the other hand, $\overline{u_4}$ is not left-diverse, because the left character of both of its children is A.

LEMMA 1. *An internal node $\overline{\omega}$ is left-diverse, if any of its children is left-diverse.*

The lemma follows directly from Definition 1. For example, internal node $\overline{u_1}$ is left-diverse, because its child $\overline{u_2}$ is left-diverse. Let us focus again on internal node $\overline{\omega}$ and its children $\overline{\omega}_1, \ldots, \overline{\omega}_k$, and assume we have calculated $f(\cdot)$ for each of the children. In order to generate maximal pairs, we need repeats that are right-diverse, meaning that they come from two different children $\overline{\omega}_i, \overline{\omega}_j$, where $i \neq j$. The repeats must also be left-diverse, meaning that $c_n \neq c_m$, where c_n, c_m are characters. Let $x \in f(\overline{\omega}_i, c_n)$ and $y \in f(\overline{\omega}_j, c_m)$. The set $MP(\overline{\omega})$ of maximal pairs for node $\overline{\omega}$ is the Cartesian product of all x and y values that satisfy the above conditions. Formally:

$$MP(\overline{\omega}) = \begin{cases} \{\ \langle (x, y), |\omega| \rangle \ \ x \in f(\overline{\omega}_i, c_n), \ \ y \in f(\overline{\omega}_j, c_m); \\ \forall i, j : 1 \leq i \leq k-1; 2 \leq j \leq k; i < j; \\ \forall c_n, c_m \in \{\{\Sigma\}, \#\}\ \} \text{ if } c_m \neq c_n, \\ \varnothing \qquad \text{otherwise} \end{cases} \quad (3)$$

In Figure 2, node $\overline{u_2}$ has two children $\overline{\nu}_1$ and $\overline{\nu}_{13}$ that generate $f(\overline{\nu}_1, X) = 1$ and $f(\overline{\nu}_{13}, Q) = 13$, respectively. Since the lists come from two different children (i.e., right-diverse) and the left characters differ (i.e., left-diverse), we get one maximal pair: $MP(\overline{u_2}) = \{\langle (1, 13), 4 \rangle\}$ that corresponds to maximal repeat ABCY, whose length is 4. Similarly, for node $\overline{u_1}$ we join the lists from $\overline{u_2}$ with that from $\overline{\nu}_9$. The result contains two maximal pairs: $MP(\overline{u_1}) = \{\langle (1, 9), 3 \rangle, \langle (9, 13), 3 \rangle\}$, both corresponding to maximal repeat ABC. Gusfield's algorithm can now be summarized as follows: the algorithm descends the tree top-down. At the leaf level it calculates Equation 1. Then it ascends the tree recursively back to the root. At each internal node, it aggregates the lists using Equation 2. It also generates maximal pairs using Equation 3.

3.3 RAST: A Naïve Disk-Based Approach

Existing maximal pair approaches are memory resident methods (see Table 1). None of them is parallel. As we explained above, this limits existing methods to very small inputs. To scale to multi-GB long sequences, we developed an extension of Gustfield's algorithm called *RAST*. RAST uses ERa [17] (see Section 3.1) to construct a disk-based suffix tree and uses the resulting subtree decomposition as a set of tasks that can be executed in parallel. Recall that, by construction, each subtree fits in memory. Therefore, for each subtree RAST applies Gustfield's algorithm.

Compared to existing systems, RAST supports much larger sequences. However, RAST has three main drawbacks: (*i*) Tree nodes are scattered throughout the memory, causing many cache misses that slow down the entire process. (*ii*) By construction, the leaves of the suffix tree are not ordered by the position of each suffix in the string, rendering filtering by range (an important operation in many applications) inefficient. (*iii*) Task decomposition based on the resulting subtrees may be too coarse and may not utilize efficiently the available CPUs for parallel processing.

4. IMPROVING CACHE EFFICIENCY

Next we present our approach, called RACE[2], which solves the aforementioned problems of RAST. In contrast to existing methods, RACE navigates the tree bottom-up. Section 4.1 presents the CO-SUFFIX data structure that facilitates the bottom-up traversal and the efficient implementation of range queries. Section 4.2 discusses the COL data structure that improves cache-efficient computation of Equations 1, 2, and 3. Section 4.3 presents the serial version of our algorithm. Then, Section 5 focuses on the parallel version and explains how RACE efficiently utilizes the available CPUs and supports elasticity in cloud environments.

4.1 The CO-Suffix Model

Our main objective is to recover the left character lists without following pointers between scattered nodes to avoid poor memory locality. A bottom-up navigation is more suitable for maximal pairs computation, where we access directly leaf nodes and their corresponding left character lists. The suffix tree can be represented as a list of nodes, where we have direct access to the leaf nodes. Each node contains the tuple (d, **nid**, pid, lch, len). d denotes the depth of the node. **nid** identifies the node, whose parent ID is pid. len

[2]RACE: *R*epeat *A*nalytics in *C*loud *E*nvironments.

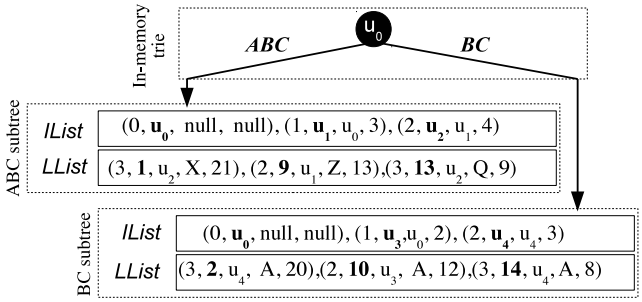


Figure 3: The CO-Suffix representation for the suffix tree shown in Figure 2. ABC and BC are prefixes sharing different set of suffixes.

is the length of the node path label. The leaf node has the suffix starting position as **nid**, and a left character *lch*.

The suffix tree of S contains $\mathcal{O}(|S|)$ internal nodes and exactly $|S|$ leaf nodes [17]. Internal nodes do not have left character, i.e *lch* is null for internal nodes. Moreover, after accessing the leaf nodes, bottom-up navigation goes through a set of internal nodes only. Therefore, CO-Suffix models the suffix tree as two lists (*IList* and *LList*), where $IList = \{(d, \textbf{nid}, pid, len)\}$ and $LList = \{(d, \textbf{nid}, pid, lch, len)\}$. *LList* is sorted by **nid**. Equation 1 is evaluated by accessing directly 50% of the suffix tree (leaf nodes only). The other 50% of the suffix tree (internal nodes) is required to evaluate Equations 2 and 3. In the CO-Suffix sub-tree, each node in *LList* represents a suffix tree leaf node combined with its left character. Therefore, RACE does not need to access S. That eliminates the random access to the input sequences.

Figure 3 illustrates the sub-trees using CO-Suffix and the left character lists associated with leaf nodes shown in Figure 2. The leaf node at depth 3, whose starting position is 1 and attached with left character list $\{\texttt{X}, \{1\}\}$, is represented in Figure 3 as $\langle 3, \textbf{1}, u_2, \texttt{X}, 21\rangle$, where u_2 is the parent id, $lch = \texttt{X}$ and 21 is the path label length. The internal node u_2 at depth 2 is represented in Figure 3 as $\langle 2, u_2, u_1, 4\rangle$, where u_1 is the parent id and 4 is its path label length.

Sorting numerically the suffix starting positions facilitates the recovery of leaf nodes lying inside a specific query range. Therefore, CO-Suffix guarantees a 100% pruning accuracy, where it accesses only the suffixes lying in the query range. We use a binary search algorithm to identify the leaf nodes belonging to the query range. *IList* and *LList* are to be stored in contiguous memory without any pointers. Therefore, using CO-Suffix avoids the serialization overhead required to exchange suffix sub-trees among compute nodes.

We utilized ERa [17] to index the input sequence. As a preprocessing, the constructed disk-based suffix sub-trees are mapped into CO-Suffix. The mapper traverses each sub-tree once to generate the corresponding *IList* and *LList*. The mapping is very fast w.r.t the suffix tree construction.

4.2 COL: A Cache Oriented List

Equation 2 collects descendant leaves under a specific internal node and groups them by the left characters. Each group represents a left character list. The group members are conceptually adjacent; allowing for spatial locality. Moreover, the descendant leaves collected at the deepest internal node u_i is a subset of descendant leaves to be collected at its parent node u_j. This gives an opportunity for exploiting temporal locality, where the sets maintained for the children of node u_j are grouped together without physical data movement. The properties of spatial and temporal locality are discussed in Sub-section 4.4. We benefit from these properties in developing a cache oriented list (COL).

The COL list is a contiguous data structure, which maintains the sets of descendant leaves (left character lists) in contiguous memory. The maintenance of left character lists is done level by level. We refer to all the descendant leaves at the current level as *active rows* and they are candidates to be part of the maximal pairs set at this level. We model COL as a 8-tuple list (*st*, *d*, *na*, *ca*, *len*, *pa*, *sp*, *lch*), where:

ca	**c**urrent **a**ncestor	Maximal pairs are computed for each *ca*.
d	**d**epth	Tree level (root is at 0).
sp	**s**tart **p**os	Corresponds to leaf *nid* (Figure 3).
lch	**l**eft **ch**aracter	*lch* should be different to report this row in maximal pair(s), see Equation 3.
pa	**p**revious **a**ncestor	*pa* must be different since *sp*s of same *pa* have the same right character (see Section 3).
len	repeat **len**gth	maintained by moving up from depth i to $i-1$.
na	**n**ext **a**ncestor	used to move up by joining *na* with *IList* .
st	**st**atus of the tuple	status is *n*: leaf will be considered at **n**ext levels; or *c*: leaf is one of the descendants at the **c**urrent level and it represents an *active row*.

Algorithm 1: Finding Maximal Pairs for a Subtree

```
Input: subtree, min_len, and gap
Output: MP a set of maximal pairs ⟨(pos_i, pos_j), len⟩
1  MP = {}
2  COL is the 8-tuples list
3  counter = 0 // number of active rows
4  removed = 0 // number of removed rows
5  HTable hash table of the internal nodes
6  INITIALIZE(subtree.LList,COL)
7  if (SIZEOF(COL) > 1) then
8      HTable = BUILDHTABLE(subtree.IList)
9  while (SIZEOF(COL) > 1 and CURRENT_LEVEL(COL) ≠ root) do
10     counter = MOVEUP(HTable, COL)
11     removed += FILTERBYLENGTH(min_len, COL, counter)
12     MP = MP + GETMAXIMALPAIRS(gap, COL, counter)
13 return MP
```

4.3 RACE Serial Execution

The RACE algorithm computes maximal pairs for each sub-tree (represented as *IList* and *LList*). For each sub-tree, RACE is mainly: (a) fetching the related leaves from *LList* into a COL list, (b) building a hash table for *IList*, and (c) looping from the deepest level of the sub-tree to the root to (*i*) recover the repeat length *len* of the internal nodes at this level, (*ii*) filter the tuples, whose length is less than the minimum length, and (*iii*) join the COL list to report the maximal pairs at this level, Equation 3. Algorithm 1 is the pseudo code for processing a sub-tree.

We demonstrate Algorithm 1 using query Q targeting the entire S. Tables 2, 3 and 4 illustrates snapshots of the COL

list while processing subtree $\overline{\texttt{ABC}}$ from Figure 3. In line 5 of Algorithm 1, a hash table for the internal nodes ($IList$) is maintained to look up the parent of each row while moving up. We look up the parent from $IList$ using na. COL is initialized by fetching all leaves belonging to the query in the sub-tree. As shown in Table 2, the 3 leaf nodes of sub-tree $\overline{\texttt{ABC}}$ shown in Figure 3 are fetched. The COL rows are sorted descendingly by d. At a certain depth, the leaves (rows) of related siblings have the same ca value, such as leaves 1 and 13 at depth 3 shown in Table 2. Since the initialization fetched leaf nodes, (i) the attributes referring to next (na) and current (ca) ancestors are assigned the same value, and (ii) the attribute previous ancestor (pa) is $null$. In Table 2, there are only two rows belonging to the deepest level ($d = 3$).

st	d	na	ca	len	pa	sp	lch
n	3	u_2	u_2	$null$	$null$	1	X
n	3	u_2	u_2	$null$	$null$	13	Q
n	2	u_1	u_1	$null$	$null$	9	Z

Table 2: Trace 1 of query Q.

The suffixes (branches) of the sub-tree are recovered in a bottom-up fashion. Iteratively, COL is joined with $IList$ to move from level i to $i-1$ until the sub-tree root is reached. In Table 3, RACE recovers the repeat length of the active rows at $depth = 3$ by joining COL shown in Table 2 with $IList$ shown in Figure 3; There are only two rows moved up to $depth$ 2, i.e there are two active rows with $st = c$.

st	d	na	ca	len	pa	sp	lch
c	2	u_1	u_2	4	$null$	1	X
c	2	u_1	u_2	4	$null$	13	Q
n	2	u_1	u_1	$null$	$null$	9	Z

Table 3: Trace 2 of query Q.

The active rows are joined to report $\langle(1, 13), 4\rangle$ as a pair with a maximal repeat of length 4. The joining conditions are (i) pa and lch are different and (ii) ca is the same. That means leaves coming from different siblings have different left characters and share the same parent, as discussed in Equation 3. RACE iterates until all tuples in COL reach the root or COL becomes empty, line 9. To move to level $d = 1$, the rows at depth 2 shown in Table 3 are joined with $IList$ of sub-tree ABC shown in Figure 3. The COL list after the join is shown in Table 4, where all rows has the root as na, and $activerows = 3$. Then, the tuples $\langle(1, 9), 3\rangle$ and $\langle(9, 13), 3\rangle$ are reported as maximal pairs.

st	d	na	ca	len	pa	sp	lch
c	1	u_0	u_1	3	u_1	1	X
c	1	u_0	u_1	3	u_1	13	Q
c	1	u_0	u_1	3	$null$	9	Z

Table 4: Trace 3 of query Q.

The tuples whose len is less than the given minimum repeat length must be removed from COL. In line 11 of Algorithm 1, `removed` is incremented to refer to the total number of filtered rows. The function GETMAXIMALPAIRS cartesian products the COL active rows at line 12. In Table 4, GETMAXIMALPAIRS reports $\langle(1, 9), 3\rangle$ and $\langle(9, 13), 3\rangle$ as maximal pairs, but $\langle(1, 13), 3\rangle$ is not a maximal pair since the corresponding tuples have the same pa, see Equation 3.

For queries targeting a specific range $\langle r_1, r_2\rangle$, the function INITIALIZE at line 6 is to have a range parameter and fetch only leaf nodes, where $r_1 \leq \text{sp} \leq r_2$. Moreover, the function GETMAXIMALPAIRS is to exclude any pair $\langle(p_1, p_2), len\rangle$, where $(p_1 + len)$ or $(p_2 + len) > r_2$. For example, assume a query targets the range $\langle 1, 13\rangle$. Although all leaves of sub-tree $\overline{\texttt{ABC}}$ will be fetched, the pair $\langle(1, 13), 4\rangle$ will be excluded as $(13 + 4) > 13$. For sub-tree $\overline{\texttt{BC}}$, the leaf node, whose $sp = 14$, will be excluded.

4.4 Spatial and Temporal Locality

For spatial locality, in COL descendant leaves corresponding to a specific left character list are contiguous in memory. Intuitively, the cartesian product of these leaves has very high locality. As shown in Tables 2, 3, and 4, the active rows are next to each others. For example, cartesian product of tuples in Table 4 is done by moving over contiguous memory. In COL, the conceptually adjacent elements are also physically adjacent in memory.

For temporal locality, COL is sorted first by st then d. Hence, the *active rows* of the current level are on top and followed by rows to be considered in the upper levels. By moving up, the rows with $d = i$ are automatically considered as descendants at level $i - 1$ without data movement, as shown in Tables 3 and 4. For example, the 3^{rd} row in Table 3 is added to the descendant set (active rows) at $d = 1$ in Table 4 by only updating the value of its st.

The COL list is a maximal pairs *compact format* for all leaves joined together at a certain level, see for example the COL list at $depth = 1$ shown in Table 4. The attributes, which are not needed to regenerate the pairs, will be omitted. The compact format w.r.t the set of maximal pairs demands significantly less space and is computed in less time. It is useful to re-use the compact format with different queries.

5. UTILIZING CLOUD RESOURCES

This section presents a fine-grained decomposition and scheduling mechanism that allows RACE to run on large-scale infrastructures, such as supercomputers or computing clouds. The goals when using such an infrastructure are achieving balanced workload and high speedup efficiency. We also discuss our elasticity cost model. Existing maximal pairs methods cannot be trivially parallelized on cloud infrastructures for the following reasons:

High I/O cost. Existing methods assume that the suffix trees are memory resident, and they access these trees in a random-access pattern. If the trees are stored on disk, these methods incur high I/O cost in order to read the required data from the disk or through the network. The size of the suffix tree is significantly larger than the sequence size (potentially an order of magnitude larger or more). Therefore, memory resident methods require huge memory budget that is not available on a single compute node in a supercomputer or cloud, and very expensive to provide in a large server. Since such memory is typically not be available, these methods incur substantial I/O cost.

Dependency delay. Existing methods process the suffix tree level by level from leaves to root. The descendants of a node at level i become descendants of its parent at level $i - 1$. This causes considerable delay as a task at level $i - 1$ waits for the corresponding tasks at level i to finish.

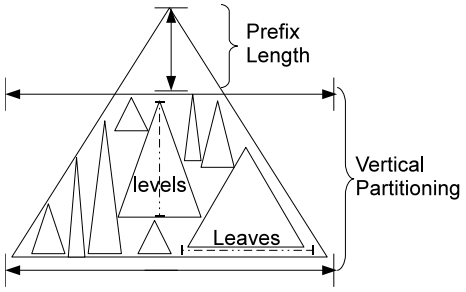


Figure 4: Suffix tree decomposition leads to sub-trees (tasks) of different height (levels), and width (number of leaves). The longer the prefix length, the more sub-trees of arbitrary sizes.

Lack of decomposition support. Existing methods deal with the suffix tree as one unit without decomposition into sub-trees. Therefore, it is not easy to parallelize the computation among multiple compute nodes, since the suffix tree would need to be copied or compute nodes would need to communicate with other nodes to exchange lists of descendant nodes.

5.1 Workload and Task Decomposition

RACE decomposes the maximal pair computation into independent tasks of arbitrary sizes that are processed simultaneously in parallel. Each task is represented by a sub-tree assigned to one processor. RACE utilizes a disk-based construction method, called ERa [17], that builds the suffix sub-trees using a set of variable length prefixes. As discussed in Section 3.1, this set of prefixes is specified such that each prefix corresponding to a sub-tree fits into the given memory budget. The bigger the memory budget, the smaller the number of sub-trees, and the faster the construction time.

RACE uses *fine-grained* decomposition to generate a large number of sub-trees (tasks) of small sizes, which is better for load balancing. Intuitively, fine-grained decomposition enables more tasks to be performed in parallel and consequently reduces the time required to compute maximal pairs. One way to ensure that RACE generates sufficiently many tasks is to use longer prefixes when RACE maps the sub-trees into the CO-Suffix model. The mapping time is not significantly affected by extending the prefix length, since the main factor is the total size of the suffix tree to be traversed. One drawback of this approach is that users will not be able to find maximal pairs whose length is smaller than the prefix used. Practically, for very long sequences, users will not be interested in small repeats, so this is not a severe limitation.

For example, the suffix tree of the Human Genome can be divided into $3,522$ or $12,952$ sub-trees with maximum prefix length ($PLen$) 13 or 20 characters, respectively. A protein sequence, whose length is equal to the Human Genome, can be divided into $18,927$ sub-trees with maximum prefix length of 8 characters; the larger the alphabet the more sub-trees. These subtrees can be used to process queries with minimum length that is greater than or equal to $PLen$. The fine-grained decomposition is limited by an inherent bound, which is the length of the smallest repeat of interest.

Having decomposed the suffix tree into sub-trees, the main challenge in order to achieve efficient utilization of parallel computing resources, is to balance the workload among the different processors. The fine-grained decomposition generates sub-trees of different sizes, as shown in Figure 4. Thus, it is important to balance the load among the compute nodes by assigning a different number of subtrees to each compute node, such that the total processing time of each compute node is approximately the same. Finding an optimal assignment of subtrees to nodes is an NP-hard problem, but RACE uses efficient parallel execution strategies to provide a good solution, as we describe next.

5.2 Strategies for Parallel Execution

Our parallel execution strategies aim at (i) reducing the time spent in interacting between different compute nodes and (ii) minimizing the time that compute nodes spend idle, waiting for other nodes to process more tasks. Each sub-tree represents an independent task whose data can fit into the main memory of a single compute node. Therefore, the maximal pair computation per sub-tree does not require any communication among nodes. The main challenge for RACE is to minimize idle time by utilizing efficient scheduling. We propose static and dynamic scheduling algorithms that balance the workload on all compute nodes.

5.2.1 Static Scheduling

Static scheduling specifies, prior to computation, a set of sub-trees for each compute node to process. We developed an inexpensive heuristic algorithm for this assignment, called Set_{robin}, which tries to achieve load balance and reduce idle time. Static scheduling leads to totally independent tasks with no interaction between the compute nodes, as shown Figure 5. The Set_{robin} algorithm identifies the sets of sub-trees assigned to each node in a round-robin manner, so that each set gets non-adjacent sub-trees. Fine-grained decomposition is the main foundation of Set_{robin}, where we have substantially more sub-trees than the P nodes. Since Set_{robin} distributes the sub-trees among the P compute nodes in a round-robin manner, it provides the following: (i) the algorithm is guaranteed to create the required number of sets, and (ii) the algorithm generates sets of different sub-tree sizes, and even if different sub-trees require different processing time, the overall workload should balance out.

The scalability of RACE relies on balancing the workload. Increasing the number of compute nodes reduces the average number of sub-trees assigned to each node. Thus, the probability of load imbalance increases. This happens regardless of the algorithm used for static scheduling, and is addressed by dynamic scheduling, which we describe in the next section. Two additional drawbacks of static scheduling are: (i) The number of leaves (or size of subtrees) is not the major factor affecting the running time. It is rather the number of maximal pairs produced, which cannot be predicted beforehand. The distribution of the maximal repeats is not uniform, so dividing tasks according to number of leaves or subtrees does not necessarily guarantee a balanced workload. (ii) Static scheduling is not flexible in terms of nodes joining/leaving the network. Such nodes incur the overhead of rescheduling and sending data to/from them.

5.2.2 Dynamic Scheduling

Dynamic scheduling distributes the sub-trees among compute nodes during the maximal pairs computation to achieve

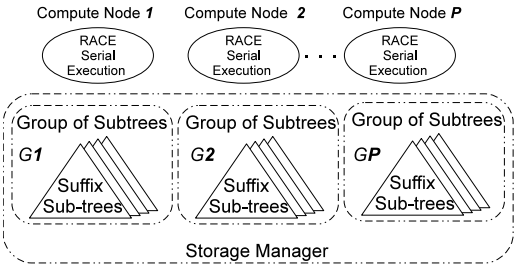


Figure 5: Static scheduling partitions the sub-trees into P sets, with a different number of sub-trees in each set, where P is the number of compute nodes.

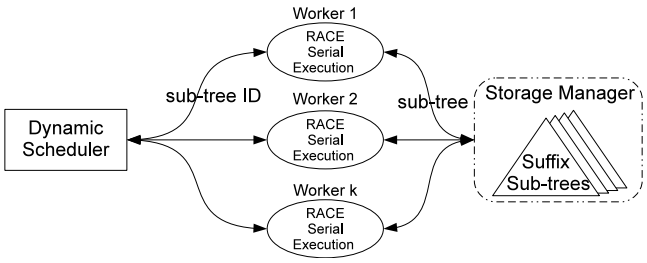


Figure 6: Centralized dynamic scheduler identifies a sub-tree ID for an idle worker node, which fetches the sub-tree from shared storage.

a better load balancing. We developed a centralized dynamic scheduling method, which classifies the compute nodes into a *master node* (the dynamic scheduler) that manages the pool of sub-trees, and *worker nodes* that rely on the master to obtain tasks, as shown in Figure 6.

A centralized dynamic scheduler may become a serialization bottleneck, where workers can become idle while waiting to get a task from the master. In order to avoid this bottleneck, we need to keep the master as lightweight as possible. Therefore, in our dynamic scheduler, a worker only gets the sub-tree ID from the master, not the sub-tree itself. The worker fetches the sub-trees from shared storage, such as a SAN-attached disk or a set of compute nodes acting as storage. The maximal pairs computation requires a significant amount of time compared to the time to fetch the sub-trees to the local memory of the worker nodes, so a serialization bottleneck is unlikely.

The time for the maximal pair computation is determined primarily by the number of maximal pairs z, plus the sequence size $|S|$, which is usually small compared to z. In dynamic scheduling, each worker processes a set of sub-trees based on the actual task running time. Therefore, dynamic scheduling achieves a better load balance compared to the static scheduling, especially when we scale to a large number of nodes. The fine-grained decomposition is important for dynamic scheduling in order to increase (i) the maximum number of tasks that can be processed simultaneously, which we refer to as the *maximum degree of concurrency*, and (ii) the average number of tasks per worker. The higher this average, the more balanced the workload.

Our goal is to run RACE on large-scale infrastructures such as supercomputers and computing clouds. In these environments, the availability of resources changes dynamically. Our dynamic scheduling method supports on-demand elastic scale-up or -down to any number of available compute nodes. As long as the maximum degree of concurrency allows more tasks to run in parallel, the master can assigns tasks to the new worker nodes without difficulty. Also, a worker in scaling-down might leave the system as soon as it finishes the assigned sub-tree without any overhead, since there is no need for data communication. We also provide a cost model that enables us to make decisions about when to scale up or down. We discuss this cost model next.

5.3 The RACE Elasticity Model

RACE with the dynamic scheduler can run on any number of compute nodes, and it can elastically scale up and down as necessary. This raises an interesting question of whether to scale up or down, and by how much. This question is of particular interest if there is a monetary cost associated with using the computing infrastructure. For example, cloud service providers such as Amazon EC2 allow users to rent resources on a time-unit basis (e.g., per hour). The more resources a user rents for RACE, the faster the computation can proceed. However, more resources come at a higher monetary cost. This leads to the question of how fast should a computation be, and hence what is the least amount of resources that needs to be rented for this computation. A user can specify a time deadline for their maximum pairs query. It is then *cost-effective* for the user to identify the minimum required resources to meet this deadline. For example, if the deadline is 30 minutes, it is a waste of budget to rent 1024 machines to accomplish a task within 15 minutes. In contrast, 512 machines that accomplish the task within 30 minutes with less cost are cost effective. We provide a cost model for RACE to estimate the cost-effective resources that are required to complete the task within a specific time constraint.

Furthermore, cloud service providers adopt different revenue management strategies, such as bid price control and dynamic pricing, to reduce unused capacity and increase resource utilization [2]. Users could be attracted to rent for a specific time period more resources if the price goes down or the application workload increases. RACE furnishes the users with information assisting in making these decisions.

We estimate *cost-effective resources* (CER) based on *resource efficiency* (RE), which specifies the time that resources consume to complete a task. RE is given by the following equation:

$$RE(Q,P) = \frac{AT(Q)}{SE(Q,P)\cdot P} \tag{4}$$

This equation estimates the resource efficiency of P compute nodes for a specific query Q. This estimation is based on the *approximate time* (AT) that is required to execute Q on a single node, and the *speedup efficiency* (SE) on P nodes. Speedup efficiency is defined as $(\frac{\tau_1}{P*\tau_P})$, where τ_1 is the time of serial execution, τ_P is the time achieved using P compute nodes. It is a measure of resource efficiency, or the average utilization of the P nodes. For example, if RACE complete a task in 10 hours using one machine: (i) with 100% speedup efficiency on 5 machines RACE would consume 2 hours, and (ii) with 50% speedup efficiency on 100 machines RACE would consume 0.2 hours. A 100% speedup efficiency corresponds to linear speedup, which is not always

possible for several reasons: (i) contention for shared resources, which causes a significant amount of performance interference; (ii) communication overhead, which might increase when using more nodes; (iii) load imbalance, which leads to more idle nodes. Since it is difficult to totally avoid these reasons, it is not usual to achieve linear speedup [7].

The RACE speedup efficiency is affected by imbalance in the workload, as discussed in previous sections. The average number of tasks per compute node significantly affects workload balance; the more tasks, the less imbalance. The average number of tasks is equal to $(\frac{Q.NT}{P})$, where $Q.NT$ is the total number of tasks (sub-trees) of a query Q. We estimate SE as follows:

$$SE(Q,P) = \min\left(\ln\left(\left(\frac{Q.NT}{P}\right)^{\alpha}\right) + \frac{\beta}{\ln(P)}, 1\right) \quad (5)$$

This equation is based on our empirical observation that a natural logarithmic correlation between SE and both V and P gives a good fit. Intuitively, P should not exceed $Q.NT$. The variables α and β are obtained as follows: We run a set of experiments on a particular computing infrastructure with various queries. On the collected data, we perform non-linear regression. We transform the data using a logarithmic function and perform least-square fitting. For example, using this approach, we experimentally found that for the IBM Blue Gene/P supercomputer the best values for α and β are 0.1634 and 2.5, respectively. Theoretically, this process should be repeated for different infrastructures. In practice, we got similar results for our Linux cluster, too. Therefore, for simplicity, in our experimental evaluation we use the same parameter values irrespectively of the infrastructure. Experiment-driven modeling of performance is an interesting research area in its own right. We have found that the model structure and parameter fitting approach described above result in accurate models in our setting, and we leave further refinement of this approach to future work.

The next term we need to estimate for Equation 4 is AT. We estimate the time of processing a query Q serially against a sequence S using a sample R extracted from S. We studied the performance of RACE in terms of time and sequence size. The results show that RACE has a linear relationship between the sequence size and processing time, which is captured by the following equation:

$$AT(Q) = \gamma \cdot \tau_R \frac{|Q.S|}{|Q.R|} \quad (6)$$

where τ_R is the time to process Q against R. γ is a constant that is estimated experimentally, using linear regression. Theoretically, γ depends on the dataset. In practice, we estimated the parameter only for one dataset (i.e., DNA) and applied it to all of our datasets without significant error. The larger the sample, and the closer the ratio of maximal pairs between S and R, the better the estimated time. Another possibility is that users may be aware of the actual time based on previous executions. In this case, the actual time is used for AT instead of the estimate. Having estimated the two components required for estimating RE in Equation 4, we now turn our attention to using RE to estimate CER, the cost-effective resources to meet a given deadline T. This is used to decide the amount of resources needed beyond a given P, and it is done by finding the minimum $k \geq 0$ such that $RE(Q,\ P\ +\ k) \leq T$. The CER returned is $(P + k)$.

Table 5: ERA parallel indexing for the Human Genome using limited memory budgets.

System	P	CPU (GHz)	RAM/node (GB)	Time (min)
Multicore	8	2.6	2	19
Multicore	4	3.33	3	29
Cluster	16	3.33	4	8.3
BlueGene/P	149	0.83	1	41

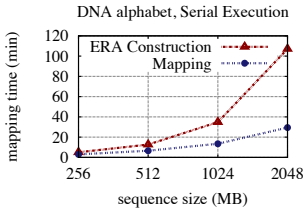


Figure 7: The execution time of mapping a suffix tree to CO-Suffix sub-trees.

6. EVALUATION

RACE serial algorithm is implemented in C. RACE runs in parallel using two versions one based on MPI[3] and another one based on Hadoop. We used three large real datasets, specified as follows: (i) DNA[4] (Human Genome, 2.6GBps), from an alphabet of 4 symbols; (ii) Protein[5] dataset containing 3GBps from an alphabet of 20 symbols; and (iii) English[6] text (26 symbols) of size 3GB chars. For the third dataset we extracted only the English alphabet symbols existing in an XML file representing Wikipedia in 2011. We conducted comprehensive experiments on different architectures: (i) a multicore 32bit Linux machine; (ii) a Linux cluster of 20 machines; and (iii) IBM BlueGene/P.

6.1 Indexing Sequences with Limited Memory

We start by demonstrating the effectiveness of RACE with limited memory. Existing maximal pair methods, including state-of-the-art tools such as *Vmatch* [13], require a large amount of memory to fit the entire input sequence S, its suffix tree index whose size is about $26|\mathtt{S}|$, plus the data structures used for the computation. For example *Vmatch* was not able to index the Human Genome on a workstation with 24GB RAM. On the other hand, RACE works with limited memory, where both the input sequence and its suffix tree do not fit into memory. We construct disk-based suffix sub-trees using ERa [17]. Then, the sub-trees are mapped into CO-Suffix. Table 5 presents the time to index the Human Genome with different memory budgets, which sometimes is less than the sequence size (2.6GB).

The mapping time increases linearly with the sequence size, as shown in Figure 7. The concluding remarks of these experiments are the ability of RACE to process very large sequences while both the sequence and its suffix tree are out of core. This allows RACE to work with a limited memory budget, which is the case in supercomputers (i.e 4GB RAM per compute node.) and cloud clusters (i.e 68GB RAM is

[3] http://www.mcs.anl.gov/research/projects/mpi
[4] http://www.ensembl.org/info/data/ftp/index.html
[5] http://www.uniprot.org/uniprot/?query=&format=*
[6] http://en.wikipedia.org/wiki/Wikipedia:Database_download

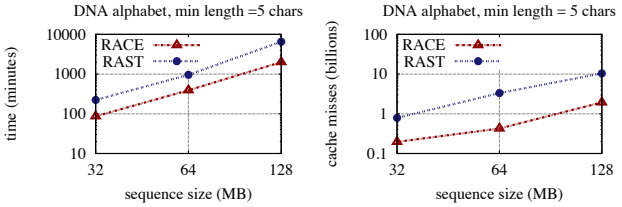
Table 6: Query Q on Human Genome with min length= 100 chars. Strong scalability test using dynamic and static scheduling implemented using MPI (DY and SR) and Hadoop static scheduling (HSR). The speedup efficiency of DY is strongly correlated with the average number of tasks per core (C). Thus, RACE elasticity model is not sensitive to system architectures, and queries workload (sequence and alphabet sizes).

		Run time in min			Speedup Efficiency (SE) %		
C	tasks/C	DY	SR	HSR	DY	SR	HSR
1	3520	120	120	120	100	100	100
32	110	3.92	4.07	5.56	95.66	92.14	67.45
64	55	1.94	2.14	3	96.65	87.62	62.5
128	27.5	1.07	1.26	2.23	87.62	74.4	42
256	13.75	0.66	0.83	2	71.02	56.5	23.44

(A) Cluster

		Run time in min		Speedup Efficiency (SE) %	
C	tasks/C	DY	SR	DY	SR
1	12955	581	581	100	100
128	101.2	4.67	5.19	97	87
256	50.6	2.37	2.77	96	82
512	25.3	1.22	1.72	93	65
1024	12.65	0.77	1.08	73	52
2048	6.3	0.47	0.59	60	48

(B) Blue Gene/P



(a) sequence scalability (b) cache misses

Figure 8: RACE vs RAST. RACE is faster because it suffers from less cache misses.

the largest capacity for an EC2 instance). Moreover, we do these preprocessing in parallel.

6.2 Cache Efficiency of RACE and RAST

We used C in implementing both RACE, our optimized method, and RAST, the naive method that presented in Section 3. The serial execution performance of both RACE and RAST is evaluated in terms of time and cache misses, as shown in Figure 8(a). This experiment was executed on a Linux machine with a quad-core AMD CPUs at 3.2GHz and 8GB RAM. In this experiment, both RACE and RAST were tested against DNA sequences of different sizes ranging from 256MB to 2048MB.

RACE is faster than RAST since RACE is a cache friendly algorithm because it utilizes the COL and CO-Suffix models (Section 4). Figure 8(b) shows the number of cache misses for RACE and RAST, measured using the Linux `perf` tool. RACE incurs fewer cache misses so it outperforms RAST by a ratio that increases correspondingly to the increase in the sequence size.

6.3 Utilization of Large Infrastructures

We developed a parallel versions of RACE using MPI and Hadoop. Our job scheduling strategies are classified into dynamic and static. We utilized a strong scalability test that computes the maximal pairs with the minimum length of 100 characters in the entire Human Genome using a varying number of compute nodes (C). The test compares the different scheduling strategies in a computer cluster using MPI and Hadoop version, and in an IBM Blue Gene/P supercomputer using the MPI version only.

6.3.1 RACE Speedup Efficiency Based on MPI

Utilizing MPI allows RACE to run on different architectures, namely multicore systems, computer clusters, and supercomputers. DY and SR refer to RACE dynamic and static strategies implemented using MPI. We used a Linux cluster consisting of 20 machines, each with 24 AMD Opteron cores at 2.10GHz and 90GB RAM. The results of the strong scalability test are summarized in Table 6(A). The primary computational resource of IBM Blue Gene/P consists of 16,384 quad-core PowerPC processors @850MHz, with 4GB RAM per core (64TB distributed RAM). The results of the strong scalability test on this system is summarized in Table 6(B).

For query Q, RACE serial execution consumed 2 hours in the cluster and 9.6 hours on the supercomputer. RACE parallel execution consumed for the same query about 40 seconds using 256 processor of the cluster and 28 seconds using 2048 processors of the supercomputer. RACE decomposes a maximal pair query into fine-grained independent tasks, where each task is associated with only one sub-tree. The independent tasks limits the overall communication cost. Hence, the main factor to achieve high speedup efficiency is to balance the workload among the compute nodes. Our fine-grained tasks are of unknown and arbitrary sizes. DY achieves a better balance workload than SR since sub-trees are distributed at run time based on their actual workload. Thus, DY shows a better efficiency. The higher average number of tasks per node, the less imbalance is. We observed similar trends on the other datasets.

We decomposed the Human Genome suffix tree into 3520 and 12955 sub-trees, for the cluster and supercomputer respectively. Therefore a similar average number of tasks per node is achieved in the cluster and supercomputer, as shown in Table 6. The speedup efficiency of RACE is similar in the both system architectures, where the average number is similar, as shown in Table 6. This shows that our elasticity model is not sensitive to system architectures nor queries workload (sequence or alphabet sizes).

The speedup efficiency of RACE increases proportionally with the increase of the sequence size, as shown in Figure 9. The number of sub-trees increases proportionally to the increase of the sequence (Human Genome of size 2.6 GB against 512 MB from same dataset). Increasing the number of sub-trees achieves a higher degree of concurrency, where more tasks could be processed simultaneously. Therefore, RACE guarantees very high speedup efficiencies on thousands of compute nodes in cases of long sequences.

6.3.2 RACE Speedup Efficiency Based on Hadoop

Hadoop has a robust job scheduler due to its high degree of fault tolerance. Moreover, a large community, ranging from big or startup companies to research labs, back and use Hadoop. Therefore, RACE also adopted Hadoop to

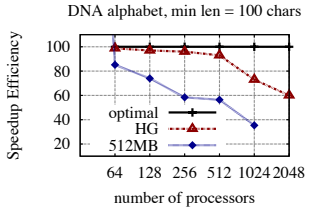
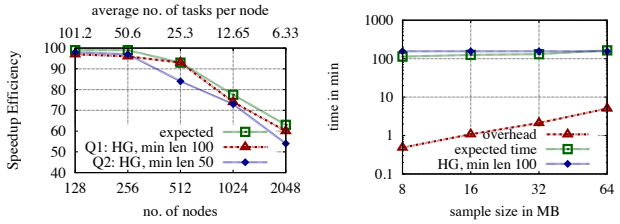


Figure 9: RACE speedup efficiency increases correspondingly by enlarging the input sequence.



(a) speedup efficiency (b) serial execution time

Figure 10: (a) Estimating the speedup efficiency on Blue Gene/P for different queries over Human Genome (HG), and (b) Estimating the serial execution time for Human Genome using samples of different sizes on the Linux workstation.

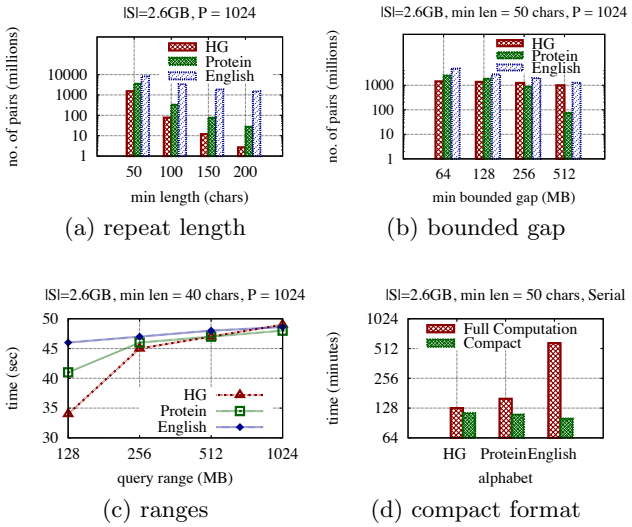
implement its static scheduling. `HSR` refers to RACE static strategy implemented using Hadoop.

RACE is implemented using C. Therefore, we utilized Hadoop Pipes to enable tasktrackers to communicate C/C++ map or reduce processes. The map function for each mapper processes a set of tasks (sub-trees) predefined using a round robin mechanism. Then, the mapper emits the number of pairs generated from these sub-trees. We utilized only one reducer for aggregating the total number of pairs. `HSR`, compared to `SR`, suffers from different kind of overheads, such as running over JVM, pipes and mapper initializations. Moreover, due to fault tolerance support, `HSR` sometimes reschedules few mapping processes. `DY` and `SR` do not support fault tolerance and runes without pipes.

6.4 Cost-Effectiveness and Elasticity Analysis

The monetary cost is proportional to the amount of resources utilized to accomplish a task within a given deadline (these resources could be rented from a cloud provider like Amazon EC2). We determine the cost effective resources by estimating the speedup efficiency and task time. This section presents experimental results that analyze the accuracy of our cost models used to estimate these quantities.

In RACE, the average number of tasks per node is an important factor, which changes inversely proportional with the number of nodes. The accuracy of the estimated speedup may be affected by other factors, such as the task workload and interference of accessing shared resources. The Human Genome index is divided into 12955 sub-trees. Figure 10(a) compares the estimated speedup of querying Human Genome to the actual time of two different queries of minimum length 100 and 50. On average, the estimation error in the expected speedup is less than 10%.



(a) repeat length (b) bounded gap

(c) ranges (d) compact format

Figure 11: Analyzing HG, Protein and English sequences of the same size, 2.6GBps, using 1024 node of IBM Blue Gene/P Each query takes a maximum of one minute. The compact format is generated using serial execution on the Linux workstation.

Approximating the query time is crucial for estimating resource efficiency (Equation 4). We estimate the serial execution time of query `Q` using Equation 6. A random sample of the sequence is i) indexed using ERa, ii) mapped into CO-Suffix, then iii) `Q` is issued against it. We refer to the total time of the three operations as `overhead`. Figure 10(b) shows the `overhead` using only one compute node. We got a very good estimated time using a sample of 32MB, which consumed about 1% overhead compared to the actual serial execution time of processing `Q` against Human Genome.

6.5 A Comprehensive Maximal Pairs System

This section demonstrates the RACE comprehensive support for finding maximal pairs. The RACE system (*i*) works with cloud infrastructures; (*ii*) is able to filter the pairs using repeat length, bounded gap and/or range; (*iii*) supports different alphabets ; and (*iv*) provides a compact format for the maximal pairs. This format consumes much less time and space compared to the actual generation of maximal pairs. The compact format is easy to iterate on while processing another string query, such as sequence alignment.

Figure 11 shows maximal pairs analysis on the three different datasets. There is a higher probability to have different left and right characters with larger alphabets. Therefore, the number of maximal pairs increases proportionally to the increase in the alphabet size, as shown in Figure 11(a). Bounded gap filtering affects the number of maximal pairs based on the distribution of the maximal repeats in the sequence, as shown in Figure 11(b). Figure 11(c) shows queries of different range size.

The sequence size is the dominating factor for generating the compact format. We evaluated the generation of the compact format and maximal pairs using a minimum length of 50 characters. The dominating factor for generating the compact format and maximal pairs is the sequence size and

number of pairs, respectively, as shown in Figure 11(d). The compact format classifies the maximal pairs based on the repeat length (Section 4.2). Thus, the generated compact format is to used for queries with min length 50 or longer.

7. CONCLUSION AND OUTLOOK

In this paper we focused on the discovery of maximal repeats in sequences, an important operator for many applications ranging from bioinformatics and text mining to compression and analysis of stock market trends. Our solution, RACE, provides cache-efficient computation of the maximal pairs with limited memory. RACE utilizes a fine-grained decomposition of the problem and a dynamic scheduling mechanism to achieve balanced workload and high speedup efficiency. These characteristics render our solution very versatile. It can process very long sequences on a common standalone machine with a few GB of memory, or it can scale to thousands of nodes (in our experiments we demonstrated scalability up to 2048 nodes on a supercomputer) and complete in seconds jobs that would otherwise require hours or days. RACE is also suitable for cloud environments since it can scale out on the fly, if more resources are needed. We also developed an elasticity model that assists the user in selecting the appropriate combination of cloud resources in terms of price and speed.

In the future we plan to generalize our approach to a framework for cloud-oriented methods, where the method is capable of (*i*) decomposing its task on-demand; and (*ii*) estimating the cost-effective resources. We also plan to focus on applications demanding large-scale maximal pairs computation, such as sequence alignment and assembly.

8. REFERENCES

[1] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. *Enhanced Suffix Arrays and Application, Handbook of Computational Molecular Biology (Chapter 7)*. Information Science Series. 2006.

[2] A. Anandasivam. *Consumer Preferences and Bid-Price Control for Cloud Services*. PhD thesis, the faculty of Economics of the Karlsruhe Institute of Technology (KIT), 2010.

[3] N. Askitis and R. Sinha. RepMaestro: scalable repeat detection on disk-based genome sequences. *Bioinformatics*, 26(19):2368–2374, 2010.

[4] T. Beller, K. Berger, and E. Ohlebusch. Space-efficient computation of maximal and supermaximal repeats in genome sequences. In *Proc. of the Int. Conf. on String Processing and Information Retrieval (SPIRE)*, pages 99–110, 2012.

[5] R. P. J. C. Bose and W. M. P. van der Aalst. Abstractions in process mining: A taxonomy of patterns. In *Proc. of the Int. Conf. on Business Process Management (BPM)*, pages 159–175, 2009.

[6] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In *Proc. of the Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 134–149. 1999.

[7] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, 1989.

[8] A. Ghoting and K. Makarychev. Serial and parallel methods for I/O efficient suffix tree construction. pages 827–840, 2009.

[9] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[10] W. Haque, A. Aravind, and B. Reddy. Pairwise sequence alignment algorithms: a survey. In *Proc. of the Conf. on Information Science, Technology and Applications (ISTA)*, pages 96–103, 2009.

[11] E. Hunt, M. P. Atkinson, and R. W. Irving. Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, 11:256–271, 2002.

[12] M. O. Kulekci, J. S. Vitter, and B. Xu. Efficient maximal repeat finding using the burrows-wheeler transform and wavelet tree. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 9(2):421–429, 2012.

[13] S. Kurtz. *The Vmatch large scale sequence analysis software*. Center for Bioinformatics, University of Hamburg, 2011.

[14] S. Kurtz and C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999.

[15] Y. Ledeneva, A. Gelbukh, and R. A. García-Hernández. Terms derived from frequent sequences for extractive text summarization. In *Proc. of the Int. Conf. on Computational linguistics and intelligent text processing*, pages 593–604, 2008.

[16] M. Linhard. Data structure for representation of maximal repeats in strings. Master's thesis, Comenius University, 2007.

[17] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. Era: Efficient serial and parallel suffix tree construction for very long strings. In *Proc. of the VLDB Endowment (PVLDB)*, volume 5, 2011.

[18] S. Maruyama, H. Miyagawa, and H. Sakamoto. Improving time and space complexity for compressed pattern matching. In *the Int. Symposium on Algorithms and Computation (ISAAC)*, pages 484–493, 2006.

[19] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23:262–272, 1976.

[20] B. Phoophakdee and M. J. Zaki. Genome-scale disk-based suffix tree indexing. pages 833–844, 2007.

[21] V. Saxena, Y. Sabharwal, and P. Bhatotia. Performance evaluation and optimization of random memory access on multicores with high productivity. In *High Performance Computing (HiPC), 2010 Int. Conf. on*, pages 1 –10, 2010.

[22] H. Stehr. Constrained matching of large biological sequences. Master's thesis, University Hamburg, 2005.

[23] A. Udechukwu, K. Barker, and R. Alhajj. Discovering all frequent trends in time series. In *Proc. of the winter Int. symposium on Information and communication technologies (WISICT)*, pages 1–6, 2004.

[24] N. E. Whiteford, N. J. Haslam, G. Weber, A. Prügel-Bennet, and C. Neylon. Visualizing the repeat structure of genomic sequences. *Complex Systems Publications, Inc., Champaign, IL*, 17(4):381–398, 2008.