On the Benefit of Merging Suffix Array Intervals for Parallel Pattern Matching

Johannes Fischer[1], Dominik Köppl[1], and Florian Kurpicz[1]

1 **Dept. of Computer Science, Technische Universität Dortmund, Germany**
johannes.fischer@cs.tu-dortmund.de,
{dominik.koeppl,florian.kurpicz}@tu-dortmund.de

Abstract

We present parallel algorithms for exact and approximate pattern matching with suffix arrays, using a CREW-PRAM with p processors. Given a static text of length n, we first show how to compute the suffix array interval of a given pattern of length m in $\mathcal{O}\left(\frac{m}{p} + \lg\lg p \lg\lg n\right)$ time for $p \leq m$. For approximate pattern matching with k differences or mismatches, we show how to compute all occurrences of a given pattern in $\mathcal{O}\left(\frac{m^k}{p}\sigma^k \max(k, \lg\lg n) + occ\right)$ time, where σ is the size of the alphabet and $p \leq \sigma^k m^k$. The workhorse of our algorithms is a data structure for merging suffix array intervals quickly: Given the suffix array intervals for two patterns P and P', we present a data structure for computing the interval of PP' in $\mathcal{O}(\lg\lg n)$ sequential time, or in $\mathcal{O}\left(\lg_p \lg n\right)$ parallel time. All our data structures are of size $\mathcal{O}(n)$ bits (in addition to the suffix array).

1998 ACM Subject Classification I.1.2 Algorithms

Keywords and phrases parallel algorithms, pattern matching, approximate string matching

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

We consider parallelizing indexed pattern matching queries in static texts, using (compressed) suffix arrays [10, 11] and (compressed) suffix trees [12, 14] as underlying indexes. We work with the *concurrent read exclusive write* (CREW) *parallel random access machine* (PRAM) with p processors, as this model most accurately reflects the design of existing multi-core CPUs. Our starting point is that a (possibly very long) pattern can be split up into several subpatterns that can be matched in parallel. In a suffix array, this will result in p subintervals, each corresponding to one of the subpatterns. These subintervals will then be combined (using a merge tree approach) to finally yield the interval for the entire pattern. From this interval, all occurrences of the pattern in the text could then be easily listed.

We also consider parallel indexed pattern matching with k errors, again using the same indexes as in the exact case. Here, we follow the approach of Huynh et al. [6], whose basic idea is to first make all possible modifications of the pattern within distance k, and then match those modifications in the suffix array. To avoid repeated computations of subintervals, a preprocessing is performed for every prefix and suffix of the pattern. We show how to parallelize both steps (preprocessing and the actual matching), resulting in a fast parallel matching algorithm. We stress that in the case of approximate pattern matching, parallel pattern matching algorithms are of even more practical importance than in the exact case, as this is an inherently time-consuming task in the sequential case, even for short patterns.

1.1 Our Results

In the abstract, we stated the results for uncompressed suffix arrays [10] as the underlying index, the latter using $n \lg n$ bits of space for a text of length n. However, there exists a wealth of compressed versions of suffix arrays [11], which are smaller (using $|\mathrm{CSA}|$ bits), but often have nonconstant access time t_{SA}. (See also Fig. 1 for a couple of known trade-offs.) Here, we state our results more generally, using the parameters $|\mathrm{CSA}|$ and t_{SA}.

Our first result (Thm. 8) is an index of size $|\mathrm{CSA}| + \mathcal{O}(n)$ bits, which, with $p \leq m$ processors, allows us to compute the suffix array interval of a pattern of length m in $\mathcal{O}\left(t_{\mathrm{SA}} \frac{m}{p} + t_{\mathrm{SA}} \lg\lg p \cdot \lg\lg n\right)$ time and $\mathcal{O}(t_{\mathrm{SA}} m + t_{\mathrm{SA}} \cdot p \cdot \lg\lg p \cdot \lg\lg n)$ work. Our second result (Thm. 12) is an index of the same size $|\mathrm{CSA}| + \mathcal{O}(n)$ bits that can find all occ occurrences of a pattern in $\mathcal{O}\left(t_{\mathrm{SA}} \frac{m^k \sigma^k}{p} \max(k, \lg\lg n) + t_{\mathrm{SA}} \mathrm{occ}\right)$ time, for $p \leq m^k \sigma^k$. Both results rely on the ability to merge two suffix array intervals quickly, a task for which we give a data structure of size $\mathcal{O}(n)$ bits on top of CSA that allows us to do the merging in $\mathcal{O}(t_{\mathrm{SA}} \lg\lg n)$ sequential (Lemma 4) or in $\mathcal{O}\left(t_{\mathrm{SA}} \lg_p \lg n\right)$ parallel time (Lemma 7).

1.2 Related Work

We are only aware of one article addressing the parallelization of single queries [7]. Their main result is to augment a suffix tree with a data structure of size $\mathcal{O}(n \lg p)$ words that answers pattern matching queries using $\mathcal{O}\left(\frac{m}{p} \lg p\right)$ time and $\mathcal{O}(m \lg p)$ work, which is worse than ours in all three dimensions. Parallelizing approximate pattern matching has not been done earlier, to the best of our knowledge. Another natural approach for exploiting parallelism would be distributing the patterns to be matched onto the different processors and answer them in parallel; this is more of a load balancing problem and cannot be compared with our approach. Parallel construction of text indices is another road of research [4, 8], and could easily be combined with our approach. Finally, in the early 1990's, some work has been done on parallelizing online pattern matching algorithms [2, 3].

2 Preliminaries

Let $T = t_1 \dots t_n$ be a *text* of length n consisting of characters contained in an integer *alphabet* Σ of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$. $T[i..j]$ represents the *substring* $t_i \dots t_j$ for $1 \leq i \leq j \leq n$. We call $T[i..n]$ the i-th *suffix* of T and $T[1..i]$ the i-th *prefix* of T. We denote the length of the *longest common prefix* of two strings S and T by $\mathsf{lcp}(S, T)$. The *suffix array* (SA) of a text T of length n is a permutation of $\{1, \dots, n\}$ such that $T[\mathrm{SA}[i]..n]$ is lexicographically smaller than $T[\mathrm{SA}[i+1]..n]$ for all $i = 1, \dots, n-1$. We denote the inverse of SA with SA^{-1}.

For a pattern $\alpha \in \Sigma^*$, let $\mathcal{I}(\alpha)$ be the interval with $T[SA[i]..SA[i] + |\alpha| - 1] = \alpha \iff i \in \mathcal{I}(\alpha)$. If we consider two intervals $\mathcal{I}(\alpha)$ and $\mathcal{I}(\beta)$ and the corresponding merged interval $\mathcal{I}(\alpha\beta)$, we call $\mathcal{I}(\alpha)$ the *left side* interval, $\mathcal{I}(\beta)$ the *right side* interval and $\alpha\beta$ the *considered common prefix*. Let $\Psi^k[i] = \mathrm{SA}^{-1}[\mathrm{SA}[i] + k]$ be the position of the suffix $T[\mathrm{SA}[i] + k..n]$ in the SA. Given an interval $\mathcal{I}(\alpha)$ in the SA, we define $\Psi^k(\mathcal{I}(\alpha)) = \{\Psi^k[i] \colon i \in \mathcal{I}(\alpha)\}$ for all $k \leq |\alpha|$.

An *interval* $\mathcal{I} = [b..e]$ is the set of consecutive integers from b to e, for $b \leq e$. For an interval $\mathcal{I}$, we use the notations $\mathsf{b}(\mathcal{I})$ and $\mathsf{e}(\mathcal{I})$ to denote the beginning and end of $\mathcal{I}$; i.e., $\mathcal{I} = [\mathsf{b}(\mathcal{I})..\mathsf{e}(\mathcal{I})]$. We write $|\mathcal{I}|$ to denote the length of $\mathcal{I}$; i.e., $|\mathcal{I}| = \mathsf{e}(\mathcal{I}) - \mathsf{b}(\mathcal{I}) + 1$.

|CSA|	t_{SA}	t_Ψ	reference
$2n \lg n$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	[10]
$nH_k + o(nH_k) + \mathcal{O}\big(n + \sigma^{k+1} \lg n + n \lg n/s\big)$	$\mathcal{O}(\lg s)$	$\mathcal{O}(1)$	[1]

Figure 1 Different representations of the compressed suffix array using |CSA| bits with the time bounds t_{SA} and t_Ψ for accessing a value of SA and Ψ, respectively. $s = \omega(\lg_\sigma n)$.

2.1 Suffix Tree

The *suffix tree* (ST) of a text T is the tree obtained by compacting the trie of all suffixes of T; it has n leaves and at most n internal nodes, where n is the length of T. Each edge is labeled with a string; the leaf corresponding to the i-th suffix is labeled with i. Reading the labels of the leaves from left to right gives the SA.

Since we target small space bounds our focus is on a compressed representation of the ST. The main ingredient of the so-called compressed suffix tree is a *compressed suffix array*. Dependent on its implementation, the compressed suffix array takes $\mathcal{O}(|\mathrm{CSA}|)$ bits of space, and gives access to SA and Ψ with t_{SA} and t_Ψ time, respectively – see Figure 1 for common representations. With additional $\mathcal{O}(n)$ bits [14], it can answer queries on the LCP-array that stores the values $\mathsf{lcp}(\mathrm{SA}[i], \mathrm{SA}[i+1])$ for each $1 \le i \le n-1$. The last ingredient of the compressed suffix tree is a navigation data structure with $\mathcal{O}(n)$ bits representing the tree topology.

For our purpose, we need the following queries on the suffix tree: $\mathsf{lca}(u, v)$ returns the lowest common ancestor of two nodes u and v, $\mathsf{label}(e)$ returns the label of an edge e, $\mathsf{label}(\ell)$ returns the label of a leaf ℓ, $\mathsf{pathlabel}(v)$ returns the labels on the edges of the path from the root to v. These queries can be answered by a constant number of accesses on the text, the SA of the text, its inverse and the LCP-array.

2.2 Integer Dictionaries

An *integer dictionary* is a set consisting of tuples of the form (k, v), where $k \in U := [1..|U|]$ is an integer from a universe U with $|U| = n^{\mathcal{O}(1)}$; we call k a *key* and v a *value*. There is at most one tuple $(k, \cdot)$ for each key k, so a tuple is determined by its key. A common task is to find a tuple in a dictionary by a given key. Besides, we are interested in finding the *successor* (*predecessor*) of a key k, i.e., the largest (smallest) key k' in the dictionary with $k' < k$ ($k' > k$). We define the operations $\mathsf{key}((k, v)) = k$ and $\mathsf{val}((k, v)) = v$.

A well-known integer dictionary representation is the *y-fast trie* [17]. It can perform lookups, predecessor and successor queries in $\mathcal{O}(\lg \lg n)$ expected time, and uses $\mathcal{O}(n \lg n)$ bits of space for storing n elements. It consists of an x-fast trie whose leaves store binary search trees. In more detail, the x-fast trie stores $\mathcal{O}(n/\lg n)$ entries in $\mathcal{O}(\lg n)$ hash tables, and each leaf stores $\mathcal{O}(\lg n)$ entries in its binary search tree. Although the y-fast trie is a dynamic data structure, we only need a static version. Therefore, we use perfect hashing [5] as our hashing method, resulting in $\mathcal{O}(\lg \lg n)$ time w.h.p. in worst case for all queries, while keeping the same space bounds and linear deterministic construction time. Alternatively, we can construct the hash tables in $\mathcal{O}(n \lg \lg n)$ deterministic time [13, Theorem 1], resulting in $\mathcal{O}(\lg \lg n)$ deterministic worst case time for all queries. Further, we can exchange the binary trees with sorted arrays, which will be useful later when we parallelize the queries.

3 Suffix Array Interval Merging

To perform the merging of two suffix intervals in $\mathcal{O}(t_{\mathrm{SA}} \lg \lg n)$ time, we adapt the idea from [9, Lemma 19]. In their method, the aim is to output all occurrences resulting from the merging of two suffix array intervals in $\mathcal{O}(t_{\mathrm{SA}}(\lg \lg n + \mathrm{occ}))$ time. Here, we show how to modify their approach such that only the resulting interval is returned, leading to $\mathcal{O}(t_{\mathrm{SA}} \lg \lg n)$ time. Although our method is similar to [9], we give the full proof for completeness.

Their idea is to sample the Ψ- and lcp-values of each $(\lg^2 n)$-th suffix array position. The sampling is stored in y-fast tries such that we can break a search in a sorted array down to a y-fast trie query, or to a binary search on a range of size $\mathcal{O}(\lg^2 n)$ — both can be performed in $\mathcal{O}(\lg \lg n)$ time. To lower the space consumption, the sampling is done only for certain nodes determined by the heavy path decomposition of the suffix tree, whose definition follows.

3.1 Heavy Path Decomposition

The *heavy path decomposition* of a rooted tree assigns a *level* to each node of the tree. The level of the root is 1. A node inherits the level of its parent if its subtree is largest among the subtrees of all its siblings; we call such a node *heavy*. Otherwise, it has the level of its parent incremented by one; we then call the node *light*. Further, we define the root to be light. A connected maximal subgraph consisting of edges that connect nodes with the same level is called *heavy path*. A heavy path starts with a light node, called *head*, and ends at a leaf that is heavy.

3.2 Precomputed Data Structures

We first present a simple data structure for the child-operation $\mathsf{child}(u, c)$, i.e., find the child v of u such that the label of the edge between u and v starts with character c. We use Δ as the sampling rate throughout this section.

▶ **Lemma 1.** *A suffix tree can be augmented with a data structure of size $\mathcal{O}(n \lg n/\Delta)$ bits answering $\mathsf{child}(v, c)$ in $\mathcal{O}(t_{\mathrm{SA}} \lg \Delta)$ time.*

Proof. We sample the children of each internal node u and store the sampled children in a y-fast trie with the first character of the edge label between u and the respective child as key. Given a node u that has m children, we sample every Δ-th child of u so that u's y-fast trie contains m/Δ elements. Since a suffix tree has less than $2n$ nodes, storing a y-fast trie for each internal node takes $\mathcal{O}(n \lg n/\Delta)$ bits overall.

We search $\mathsf{child}(u, c)$ in the following way: Since the children of a node u are sorted by the first character of the edge connecting u with its respective child, the y-fast trie of u can retrieve the first child v whose edge label $\mathsf{label}(u, v)$ is lexicographically at least as large as c for a given character c. If c is a prefix of $\mathsf{label}(u, v)$, then we are done. Otherwise, say that v is the i-th child of u, we can find $\mathsf{child}(u, c)$ by a binary search on the range between the $(i - \Delta)$-th child and the i-th child in $\mathcal{O}(t_{\mathrm{SA}} \lg \Delta)$ time. ◀

We also need a simple $\mathcal{O}(n)$-bits data structure to find the heavy leaf of a given heavy path in constant time [9, Lemma 15].

Next, we define three types of integer dictionaries that we are going to index in a y-fast trie to allow fast lookups. The sets sample leaves contained in a subtree rooted at some

head with the sampling interval Δ. For every light node v, we define

$$\Gamma(v) := \left\{ (\Psi^{|v|}[i], i) : i \equiv 1 \pmod{\Delta} \wedge i \in \mathcal{I}(v) \right\}.$$

Given a heavy leaf ℓ and its head v, we define the two integer dictionaries

$$H_{\mathrm{L}}(\ell) := \{(\mathsf{lcp}(\mathsf{label}(\ell), i), i) : i \equiv 1 \pmod{\Delta} \wedge i \in \mathcal{I}(v) \wedge i \leq \mathsf{label}(\ell)\}$$

and

$$H_{\mathrm{R}}(\ell) := \{(\mathsf{lcp}(\mathsf{label}(\ell), i), i) : i \equiv 1 \pmod{\Delta} \wedge i \in \mathcal{I}(v) \wedge i > \mathsf{label}(\ell)\}.$$

We store $\Gamma(v)$ in a y-fast trie for each light node v, $H_{\mathrm{L}}(\ell)$ and $H_{\mathrm{R}}(\ell)$ in a y-fast trie for each heavy leaf ℓ. Given an interval $\mathcal{J}$, we can find

- an $i \in \Gamma(v)$ with $\mathsf{b}(\mathcal{J}) \leq \mathsf{key}(i) = \Psi^{|v|}[\mathsf{val}(i)] \leq \mathsf{e}(\mathcal{J})$,
- an $i_l \in H_{\mathrm{L}}(u)$ with $\mathsf{b}(\mathcal{J}) \leq \mathsf{key}(i_l) = \mathsf{lcp}(\mathsf{val}(i_l), \mathsf{label}(\ell)) \leq \mathsf{e}(\mathcal{J})$, and
- an $i_r \in H_{\mathrm{R}}(u)$ with $\mathsf{b}(\mathcal{J}) \leq \mathsf{key}(i_r) = \mathsf{lcp}(\mathsf{val}(i_r), \mathsf{label}(\ell)) \leq \mathsf{e}(\mathcal{J})$,

in $\mathcal{O}(\lg \lg n)$ time.

▶ **Lemma 2.** *We need $\mathcal{O}\big(n(\lg^2 n)/\Delta\big)$ bits of space to store the y-fast tries for all $\Gamma(\cdot)$, $H_{\mathrm{L}}(\cdot)$, and $H_{\mathrm{R}}(\cdot)$.*

Proof. Since the subtrees of the light nodes sharing the same level are disjoint, summing over the size of $\Gamma(v)$ for all light nodes v with the same level yields at most n/Δ elements. Since the heavy path decomposition has at most $\mathcal{O}(\lg n)$ different levels, there are at most $\mathcal{O}(\lg n)$ light nodes, and for each light node v we store $\Gamma(v)$ with $|\Gamma(v)| \leq n/\Delta$ in a y-fast trie.

We analyze the size of $H_{\mathrm{L}}(\cdot)$ by identifying a leaf with its label. The sampling of $H_{\mathrm{L}}(\cdot)$ considers only n/Δ leaf labels. A leaf ℓ has at most $\mathcal{O}(\lg n)$ light nodes as ancestors. So there are at most $\mathcal{O}(\lg n)$ heavy leaves ℓ_H having the label of ℓ in their set $H_{\mathrm{L}}(\ell_H)$. Hence, summing over the size of $H_{\mathrm{L}}(\ell_H)$ for all heavy leaves ℓ_H yields $\mathcal{O}(n \lg n/\Delta)$. The same considerations lead to the same size bounds for $H_{\mathrm{R}}(\cdot)$. ◀

▶ **Lemma 3.** *Given the compressed suffix tree of T and the above constructed y-fast tries, we can merge two SA-intervals in $\mathcal{O}(t_{\mathrm{SA}} \lg \Delta)$ time.*

Proof. Let $\mathcal{I}(\alpha)$ and $\mathcal{I}(\beta)$ be two SA-intervals for the pattern $P := \alpha\beta$. Our task is to search the interval $\mathcal{I}(P) \subseteq \mathcal{I}(\alpha)$ with $\Psi^{|\alpha|}[i] \in \mathcal{I}(\beta)$ for all $i \in \mathcal{I}(P)$. Since $i \mapsto \Psi[i]$ is monotonically increasing for $i \in \mathcal{I}(\alpha)$, the merge could be solved with two binary searches in $\mathcal{I}(\alpha)$. To obtain the $\mathcal{O}(\lg \Delta)$ time bound we will either use the y-fast tries, or perform a binary search on $\mathcal{O}(\Delta)$-large intervals.

Let us take the node v whose suffix interval is $\mathcal{I}(\alpha)$, i.e., the lowest common ancestor of the leaves with the labels $\mathsf{b}(\mathcal{I}(\alpha))$ and $\mathsf{e}(\mathcal{I}(\alpha))$. We consider two cases:

Node v is heavy. Let H be the heavy path to which v belongs, ℓ its heavy leaf, and u its head.

If $\Gamma(\ell)$ is empty, there are less than Δ leaves in the subtree rooted at u. Since $\mathcal{I}(P) \subset \mathcal{I}(u)$, we can find $\mathcal{I}(P)$ by binary search.

Otherwise ($\Gamma(\ell) \neq \emptyset$), let $q := \mathsf{lcp}(\Psi^{|\alpha|}[\mathsf{label}(\ell)], \mathsf{b}(\mathcal{I}(\beta)))$. The value q is the length of the longest common prefix of P and the path label of ℓ, subtracted by $|\alpha|$. By definition of q, there is a node r on H whose path label coincides with $\alpha\beta[1..q]$. In particular, this is the node on the path H whose path label is the longest prefix of P. Since $\mathcal{I}(P) \subset \mathcal{I}(r)$,

our task is to find r in $\mathcal{O}(t_{\mathrm{SA}} \lg \Delta)$ time. To this end, we locate a leaf whose LCA with ℓ is r.

The interval boundaries can be found by a coarse search on the y-fast tries of $H_{\mathrm{L}}(\ell)$ and $H_{\mathrm{R}}(\ell)$, and a subsequent refinement step using binary search. Let $k := \mathsf{label}(\ell)$. Since $i \mapsto \mathsf{lcp}(i,k)$ is monotonically increasing for $i < k$, and monotonically decreasing for $i > k$, we can perform the binary search for a value on the key-sorted integer dictionaries $\{(\mathsf{lcp}(i,k), i) : i < k\}$ and $\{(\mathsf{lcp}(i,k), i) : i > k\}$. We compute the tuple $j_l \in H_{\mathrm{L}}(\ell) \cup \{(|\mathsf{pathlabel}(k)|, k)\}$ with

$$\mathsf{lcp}(\mathsf{val}(j_l) - \Delta, k) \leq |\alpha| + q \leq \mathsf{key}(j_l) = \mathsf{lcp}(\mathsf{val}(j_l), k)$$

and the tuple $j_r \in H_{\mathrm{R}}(\ell) \cup \{(|\mathsf{pathlabel}(k)|, k)\}$ with

$$\mathsf{key}(j_r) = \mathsf{lcp}(\mathsf{val}(j_r), k) \leq |\alpha| + q \leq \mathsf{lcp}(\mathsf{val}(j_r) + \Delta, k).$$

Since $\mathsf{lcp}(\mathsf{val}(j_l) - \Delta, k) \leq |\alpha| + q \leq \mathsf{lcp}(\mathsf{val}(j_r) + \Delta, k)$, we can find one of the positions $i_l \in [\mathsf{val}(j_l) - \Delta .. \mathsf{val}(j_l)]$ and $i_r \in [\mathsf{val}(j_r) .. \mathsf{val}(j_r) + \Delta]$ by binary search such that $\mathsf{lcp}(i_l, k) = \mathsf{lcp}(i_r, k) = |\alpha| + q$. The binary search takes $\mathcal{O}(t_{\mathrm{SA}} \lg \Delta)$ time. On finding i_l or i_r, we can retrieve r, i.e., the lowest common ancestor of ℓ and the leaf with label i_l or i_r. If the pattern is a prefix of the label path of r, then $\mathcal{I}(P) = \mathcal{I}(r)$, and we are done. Otherwise, we choose the child w of r whose edge label S starts with $\beta[q+1]$; w can be retrieved in $\mathcal{O}(t_{\mathrm{SA}} \lg \Delta)$ time by Lemma 1. The child w must be a light node, for otherwise we get a contradiction to the definition of r. We set $v \leftarrow w$, $\alpha \leftarrow P[1..|\alpha| + q + |S|]$, $\beta \leftarrow P[|\alpha| + 1..|P|]$, and jump to the next case:

Node v is light. If $\Gamma(v)$ is empty, then $|\mathcal{I}(v)| < \Delta$. Therefore, we can find the interval boundaries of $\mathcal{I}(P)$ in $\mathcal{I}(v)$ with a binary search in $\mathcal{O}(t_\Psi \lg \Delta)$ time. Otherwise, we use the y-fast trie storing $\Gamma(v)$ to find the tuple $j_l \in \Gamma(v)$ with the smallest key satisfying $\mathsf{b}(\mathcal{I}(\beta)) \leq \mathsf{key}(j_l) = \Psi^{|\alpha|}[\mathsf{val}(j_l)]$ and the tuple $j_r \in \Gamma(v)$ with the largest key satisfying $\mathsf{key}(j_r) = \Psi^{|\alpha|}[\mathsf{val}(j_r)] \leq \mathsf{e}(\mathcal{I}(\beta))$. If both exist, we can find the positions $\mathsf{b}(\mathcal{I}(P)) \in [\mathsf{val}(j_l) - \Delta .. \mathsf{val}(j_l)]$ and $\mathsf{e}(\mathcal{I}(P)) \in [\mathsf{val}(j_r) .. \mathsf{val}(j_r) + \Delta]$ by two binary searches. If there is no tuple $i \in \Gamma(v)$ with $\mathsf{b}(\mathcal{I}(\beta)) \leq \mathsf{key}(i) \leq \mathsf{e}(\mathcal{I}(\beta))$, we search with the y-fast trie of $\Gamma(v)$ the tuple $k_l \in \Gamma(v)$ with

$$\mathsf{key}(k_l) = \Psi^{|\alpha|}[\mathsf{val}(k_l)] \leq \mathsf{b}(\mathcal{I}(\beta)) \leq \Psi^{|\alpha|}[\mathsf{val}(k_l) + \Delta]$$

and the tuple $k_r \in \Gamma(v)$ with

$$\Psi^{|\alpha|}[\mathsf{val}(k_r) - \Delta] \leq \mathsf{e}(\mathcal{I}(\beta)) \leq \Psi^{|\alpha|}[\mathsf{val}(k_r)] = \mathsf{key}(k_r).$$

Both values exist, and $\mathsf{val}(k_r) - \mathsf{val}(k_l) \leq \Delta$. So we find the interval $\mathcal{I}(P)$ by applying two binary searches to the range $\mathsf{val}(k_l) .. \mathsf{val}(k_r)$.

◀

Setting $\Delta := \lg^c n$ for $c \geq 2$ yields:

▶ **Lemma 4.** *Given the compressed suffix tree of T, there is a data structure of size $\mathcal{O}(n)$ bits that allows us to merge two SA-intervals in $\mathcal{O}(t_{\mathrm{SA}} \lg \lg n)$ time.*

4 Parallel Exact Pattern Matching

We parallelize the merging of suffix array intervals that we presented in Section 3 and show that parallel queries in the suffix tree using consecutive subpatterns and linear space can be solved in parallel on a CREW-PRAM. For this, we use parallel binary search:

▶ **Lemma 5** ([15, Theorem 2.1]). *Given a sorted array of size n, a binary search requires $\mathcal{O}(\lg_p n)$ time when operating on a CREW-PRAM with p processors.*

We conclude that we can parallelize the query on y-fast tries in the same way:

▶ **Lemma 6.** *A y-fast trie over an integer dictionary can do lookups, predecessor and successor queries in $\mathcal{O}(\lg_p \lg n)$ time using p processors.*

Proof. We can find an element in an x-fast trie in $\mathcal{O}(\lg_p \lg n)$ time using parallel binary search (Lemma 5) on the $\mathcal{O}(\lg n)$ hash tables. The sorted arrays stored at the leaves can similarly be searched in $\mathcal{O}(\lg_p \lg n)$ time, again using Lemma 5. ◀

Let us focus on the merging of two suffix array intervals treated in Section 3. The dominant term of its running time is due to the query time of the y-fast tries and the binary searches. As we can parallelize both, a parallelization of the merging algorithm improves the time bounds significantly:

▶ **Lemma 7.** *Given p processors and two intervals $\mathcal{I}(\alpha)$ and $\mathcal{I}(\beta)$, the merged interval $\mathcal{I}(\alpha\beta)$ can be computed in $\mathcal{O}(t_{\mathrm{SA}} \lg_p \lg n)$ time and $\mathcal{O}(t_{\mathrm{SA}} \cdot p \cdot \lg_p \lg n)$ work.*

Proof. We can merge two suffix intervals in $\mathcal{O}(t_{\mathrm{SA}} \lg \lg n)$ time using Lemma 4. Recalling the proof of Lemma 3, we took the node v whose suffix array interval is $\mathcal{I}(\alpha)$. There, in both cases (v is either heavy or light), the time is dominated by searching in y-fast tries, and/or by binary searching in $\mathcal{O}(\Delta)$ sampled Ψ- or lcp-values. Both can be parallelized by applying Lemmas 5 and 6, respectively. During the parallel merging, we use all p processors $\mathcal{O}(\lg_p \lg n)$ times. This yield to $\mathcal{O}(t_{\mathrm{SA}} \cdot p \cdot \lg_p \lg n)$ work. ◀

Being able to merge two suffix array intervals, now show how to compute the suffix array interval of a pattern P in parallel. To this end, we decompose the pattern in subpatterns $\alpha_1, \ldots, \alpha_p$ such that $P = \alpha_1 \alpha_2 \ldots \alpha_p$, and then compute the suffix array intervals for the subpatterns. Then we merge those intervals in parallel.

▶ **Theorem 8.** *Given a text of size n and a pattern of size m. With $p \leq m$ processors, we can compute the suffix array interval of the pattern in $\mathcal{O}\left(t_{\mathrm{SA}} \frac{m}{p} + t_{\mathrm{SA}} \lg \lg p \cdot \lg \lg n\right)$ time and $\mathcal{O}(t_{\mathrm{SA}} \cdot m + t_{\mathrm{SA}} \cdot p \cdot \lg \lg p \cdot \lg \lg n)$ work. The index uses $|\mathrm{CSA}| + \mathcal{O}(n)$ bits of space.*

Proof. Let $P = \alpha_1^0 \alpha_2^0 \ldots \alpha_p^0$ be a query of length m such that $|\alpha_i^0| = \frac{m}{p}$ for $i = 1, \ldots, p$. The computation of all intervals $\mathcal{I}(\alpha_i^0)$ requires $\mathcal{O}\left(t_{\mathrm{SA}} \frac{m}{p}\right)$ time. Thus, in the first merge step we have two processors to compute each of the intervals $\mathcal{I}(\alpha_i^1) := \mathcal{I}(\alpha_{2i-1}^0 \alpha_{2i}^0)$ for $i = 1, \ldots, \frac{p}{2}$. Since each merge step halves the number of intervals, in the k-th merge step ($1 \leq k \leq \lg p$), we have 2^k processors to compute each of the intervals $\mathcal{I}(\alpha_i^k) := \mathcal{I}(\alpha_{2i-1}^{k-1} \alpha_{2i}^{k-1})$ for $i = 1, \ldots, \frac{p}{2^k}$. As we require $\mathcal{O}(\lg p)$ merge steps and can use Lemma 7 with 2^k processors in the k-th merge step, the interval $\mathcal{I}(P)$ can be computed in $\mathcal{O}\left(t_{\mathrm{SA}} \sum_{k=1}^{\lg p} \lg_{2^k} \lg n\right) = \mathcal{O}\left(t_{\mathrm{SA}} \lg \lg n \sum_{k=1}^{\lg p} \frac{1}{\lg 2^k}\right) = \mathcal{O}(t_{\mathrm{SA}} \lg \lg p \cdot \lg \lg n)$ time, given the intervals $\mathcal{I}(\alpha_i^0)$ of the subpatterns. In total, the query P can be solved in $\mathcal{O}\left(t_{\mathrm{SA}} \frac{m}{p} + t_{\mathrm{SA}} \lg \lg p \cdot \lg \lg n\right)$ time.

During the computation of the suffix array intervals of all subpatterns we use all p processors, which results in $\mathcal{O}(m)$ work. The same holds for each merging step, as we use all processors to parallelize the binary search. We have $\mathcal{O}(\lg p)$ merge steps. During the i-th merge step, we merge $\frac{p}{2^i}$ suffix array intervals with 2^i processors each. Using Lemma 7 the total work is $\mathcal{O}\left(t_{\mathrm{SA}} \cdot m + t_{\mathrm{SA}} \cdot \sum_{i=1}^{\lg p} \frac{p}{2^i} \cdot 2^i \cdot \lg_{2^i} \lg n\right) = \mathcal{O}(t_{\mathrm{SA}} \cdot m + t_{\mathrm{SA}} \cdot p \cdot \lg \lg p \cdot \lg \lg n)$. ◀

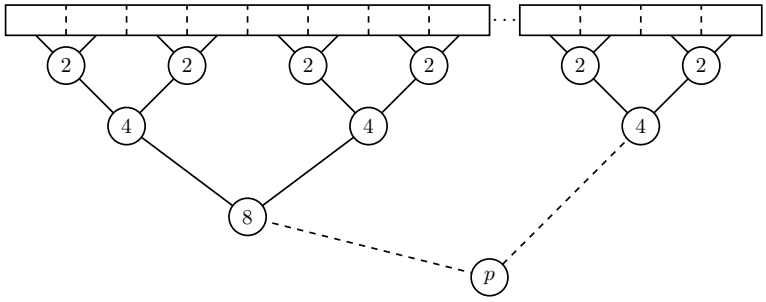

Figure 2 Merge tree resulting from the merging of p suffix array intervals, i.e., the suffix array intervals of the subpatterns. The number in each node denotes the number of processors available for the merging of the two considered suffix array intervals.

5 Parallel Approximate Pattern Matching

In this section, we consider two different distances for the approximate string matching problem. The first distance we consider is the *Levenshtein distance*, where the distance between two patterns P and P' is the minimal number of the operations *insert*, *change* and *remove* required to change P' into P. The second one is the *Hamming distance*, where the distance of two pattern P and P' of equal length is the number of mismatching positions, i.e., $|\{i\colon P[i] \neq P'[i]\}|$. We consider two problems related to these distances.

k-difference problem Given a text T of length n and a pattern P of length m, we want to report all occurrences $i \in \{1,\dots,n\}$ such that $T[i..i+j]$ and P have a Levenshtein distance of at most k for at least one $j \in \{0,\dots,n-i\}$.

k-mismatch problem Given a text T of length n and a pattern P of length m, we want to report all occurrences $i \in \{1,\dots,n\}$ such that $T[i..i+m]$ and P have a Hamming distance of at most k.

We apply the results from Section 3 to parallelize the approximate string matching algorithm by Huynh et al. [6]. To do so, we first present an approach to compute the suffix array intervals of all prefixes and suffixes of the pattern in parallel, see Figure 3.

▶ **Lemma 9.** *Given a text of length n, we can compute the suffix intervals of all prefixes and suffixes of a pattern of length m parallel in $\mathcal{O}\left(t_{\mathrm{SA}}\frac{m}{p}\lg p \lg\lg n\right)$ time with $p \leq m$ processors.*

Proof. Let P be a pattern of length m and α_i subpatterns of P such that $|\alpha_i| = \frac{m}{p}$ for $i = 0,\dots,p-1$ and $P = \alpha_0\alpha_1\dots\alpha_{p-1}$. Thus, the j-th prefix of a subpattern α_i is $P[1 + i\frac{m}{p}..i\frac{m}{p}+j]$ for all $i = 0,\dots,p-1$ and $j = 1,\dots,\frac{m}{p}$. First, we compute the suffix array intervals for all those prefixes, i.e., $\mathcal{I}\left(P[1+i\frac{m}{p}..i\frac{m}{p}+j]\right)$ for all $i = 1,\dots,p$ and $j = 1,\dots,\frac{m}{p}$ in parallel, which requires $\mathcal{O}(m/p)$ time, as no merging is necessary during this step.

In the second step, we merge the suffix array intervals in parallel. Since we want the suffix array intervals of all prefixes of the pattern, we start merging the intervals $\mathcal{I}\left(P[1+2i\frac{m}{p}..(2i+1)\frac{m}{p}]\right)$ as the left interval with each of the intervals $\mathcal{I}\left(P[1+(2i+1)\frac{m}{p}..(2i+1)\frac{m}{p}+j\right)$ as right side interval for all $i = 0,\dots,\frac{p}{2}-1$ and $j = 1,\dots,\frac{m}{p}$. This results in the intervals $\mathcal{I}\left(P[1+2i\frac{m}{p}..(2i+1)\frac{m}{p}+j\right)$ for $i = 1,\dots,\frac{p}{2}$ and $j = 1,\dots,\frac{m}{p}$. During each merge step, we halve the number of left side intervals that we have to consider during the next merge step but double the number of right side intervals that

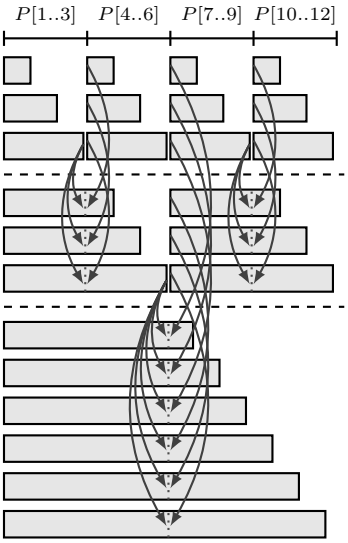

Figure 3 Schematic picture of the merging process to compute the suffix array intervals of all prefixes of a pattern P of length $m = 12$ using $p = 4$ processors. The gray blocks above the first line represent the suffix array intervals of all prefixes of the subpatterns. The blocks between the dashed lines represent the suffix array intervals after the first merge step. The intervals that are merged are shown by arrows. The blocks below the second dashed line are the suffix array intervals computed in the second merge step. Again, the merged suffix array intervals are shown by arrows.

operation	c	P'	intervals to merge
substitution	$c \in \Sigma \setminus \{P[i]\}$	$P[1..i-1]cP[i+1..m]$	$\mathcal{I}(\mathsf{child}(v,c))$ and $\mathcal{I}(P[i+1..m])$
deletion	–	$P[1..i-1]P[i+1..m]$	$\mathcal{I}(v)$ and $\mathcal{I}(P[i+1..m])$
insertion	$c \in \Sigma$	$P[1..i-1]cP[i..m]$	$\mathcal{I}(\mathsf{child}(v,c))$ and $\mathcal{I}(P[i..m])$

Figure 4 Let P' be the resulting string of introducing an error in the pattern $P[1..m]$ at position i. Further, let v be the suffix tree node with $\mathcal{I}(v) = \mathcal{I}(P[1..i-1])$. We can compute the two suffix intervals considered for merging in $\mathcal{O}(t_{\mathrm{SA}} \lg \lg n)$ time, and perform the merging in the same time.

are merged, i.e., in the k-th merge step, we merge the intervals $\mathcal{I}\Big(P[1+2^k i\frac{m}{p}..(2^k+1)i\frac{m}{p}]\Big)$ with each of the intervals $\mathcal{I}\Big(P[1+(2^k+1)i\frac{m}{p}..(2^k+1)i\frac{m}{p}+j]\Big)$ for $i = 0,\ldots,\frac{p}{2^k}-1$ and $j = 1,\ldots,2^k\frac{m}{p}$. Thus, in each merge step, we merge $\mathcal{O}(m)$ intervals. In the end, we obtain the suffix array intervals of the prefixes of P, i.e., the intervals $\mathcal{I}(P[1..j])$ for $j = 1,\ldots,m$. Since we start with p left side intervals, and each merge step halves the number of left side intervals, we end up with $\lg p$ merge steps.

The computation of the suffix array intervals of all suffixes of P works analogously. Using Lemma 4, we can compute all required suffix array intervals in $\mathcal{O}\Big(t_{\mathrm{SA}}\frac{m}{p}\lg p \lg\lg n\Big)$ time. ◀

▶ **Theorem 10.** *With* $|\mathrm{CSA}|+\mathcal{O}(n)$ *bits of space, the* 1*-difference and* 1*-mismatch problems can be solved in parallel in* $\mathcal{O}\Big(t_{\mathrm{SA}}\frac{m\sigma}{p}\lg\lg n + occ\Big)$ *for* $p \le m\sigma$.

Proof. We precompute the suffix array intervals $\mathcal{I}(P[i..m])$ and $\mathcal{I}(P[1..i])$ for all $1 \le i \le m$ in parallel by Lemma 9. The exact matches are found in the interval $\mathcal{I}(P[1..m])$. To compute the matches with one error, we iterate over all positions $P[1..m]$, and introduce

an error at one position. An error can be introduced by an insertion, a deletion, or an substitution. Let us fix one modification occurring at position i, and call the modified string P'. Our task is to find $\mathcal{I}(P')$. To this end, we exploit some already computed results, i.e., we have $\mathcal{I}(P'[1..i-1]) = \mathcal{I}(P[1..i-1])$ and $\mathcal{I}(P'[i+1..m]) = \mathcal{I}(P[i+1..m])$ or $\mathcal{I}(P'[i+1..m+1]) = \mathcal{I}(P[i..m])$ (in case of an insertion) – see Figure 4. If P' resulted from an insertion or substitution, the interval $\mathcal{I}(P'[1..i-1])$ can be enhanced to $\mathcal{I}(P'[1..i])$ by $\mathsf{child}(v, P'[i])$ in $\mathcal{O}(t_{\mathrm{SA}} \lg \lg n)$ time due to Lemma 1, where v is the node with $\mathcal{I}(v) = \mathcal{I}(P'[1..i-1])$. Finally, we can compute $\mathcal{I}(P')$ by merging two intervals in $\mathcal{O}(t_{\mathrm{SA}} \lg \lg n)$ time with Lemma 4.

Introducing an error in P at m different positions with σ different characters is embarrassingly parallel. So we get a time bound of $\mathcal{O}\left(t_{\mathrm{SA}} \frac{m\sigma}{p} \lg \lg n + \mathrm{occ}\right)$ when executing the algorithm with p processors in parallel. ◀

Up to now, we have assumed that the time for the output is in $\mathcal{O}(\mathrm{occ})$. Unfortunately, this is not always the case, as an occurrence of a pattern with k errors may be reported multiple times. For example, the pattern `aba` could be reported twice at the first position of the text `aaa` as the second position of the pattern could either be deleted or replaced. In such a case, we want to report only one occurrence. Therefore, we need to make sure that each occurrence of a pattern is reported just once, regardless how many different combinations of operations can be used to change the pattern to the corresponding substring. This problem has been discussed and solved in [6].

▶ **Lemma 11** ([6, Discussion related to Theorem 2]). *Given a pattern P, we can check whether an occurrence of the pattern P' with at most k errors is minimal regarding its distance and its edit operations to P in $\mathcal{O}(k)$ time whenever we append a character or want to report an occurrence.*

Using Lemmata 9 and 11, we can solve the 1-difference and 1-mismatch problems in parallel as described above. The same is true for the k-difference and k-mismatch problems.

▶ **Theorem 12.** *Using $|\mathrm{CSA}| + \mathcal{O}(n)$ bits of space, the k-difference and k-mismatch problems can be solved in parallel in $\mathcal{O}\left(t_{\mathrm{SA}} \frac{m^k\sigma^k}{p} \max(k, \lg \lg n) + occ\right)$ for $p \leq m^k\sigma^k$ processors.*

Proof. The idea of the algorithm is similar to the algorithm of Theorem 10. First, we compute all suffix array intervals of all the suffixes and prefixes of the pattern, i.e., $\mathcal{I}(P[i..m])$ and $\mathcal{I}(P[1..i])$ for all $i = 1, \ldots, m$. We want to introduce at most k errors in parallel. Again, we parallelize over the positions of the introduced errors. Similar to the idea of Theorem 10, we merge different suffix array intervals. But in this case, we cannot parallelize over one position, instead we have to parallelize considering up to k positions where we can include an error. We still can merge two intervals in $\mathcal{O}(t_{\mathrm{SA}} \lg \lg n)$ time using Lemma 4.

The number of patterns P' that have a distance of at most k from P is bounded by $\mathcal{O}(\sigma^k m^k)$ [16, Theorem 6]. Thus, we get an executing time of $\mathcal{O}\left(t_{\mathrm{SA}} \frac{m^k\sigma^k}{p} \max(k, \lg \lg n) + occ\right)$ using $p \leq \sigma^k m^k$ processors in parallel. The $\mathcal{O}(\max(k, \lg \lg n))$-term results from the check of whether the occurrence is computed with minimal distance to the pattern P which has to be done every time we update the considered pattern and requires $\mathcal{O}(k)$ time using Lemma 11. ◀

References

1 D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms (TALG)*, 10(4):article 23, 2014.

2 Dany Breslauer and Zvi Galil. An optimal O(log log n) time parallel string matching algorithm. *SIAM J. Comput.*, 19(6):1051–1058, 1990.

3 Dany Breslauer and Zvi Galil. A lower bound for parallel string matching. *SIAM J. Comput.*, 21(5):856–862, 1992.

4 Martin Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *Proc. ICALP*, volume 1099 of *LNCS*, pages 550–561. Springer, 1996.

5 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

6 Trinh ND Huynh, Wing-Kai Hon, Tak-Wah Lam, and Wing-Kin Sung. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science*, 352(1):240–249, 2006.

7 Matevz Jekovec and Andrej Brodnik. Parallel query in the suffix tree. *CoRR*, abs/1509.06167, 2015.

8 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In *Proc. CPM*, volume 9133 of *LNCS*, pages 329–342. Springer, 2015.

9 Tak-Wah Lam, Wing-Kin Sung, and Swee-Seong Wong. Improved Approximate String Matching Using Compressed Suffix Data Structures. *Algorithmica*, 51(3):298–314, 2007.

10 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

11 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):Article No. 2, 2007.

12 Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 322–333. Springer, 2010.

13 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Proc. ICALP (1)*, volume 5125 of *LNCS*, pages 84–95. Springer, 2008.

14 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst*, 41(4):589–607, 2007.

15 Marc Snir. On parallel searching. *SIAM J. Comput.*, 14(3):688–708, 1985.

16 Esko Ukkonen. Approximate string-matching over suffix trees. In *Proc. CPM*, volume 684 of *LNCS*, pages 228–242. Springer, 1993.

17 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Inform. Process. Lett.*, 17(2):81–84, 1983.