Lyndon Words Accelerate Suffix Sorting

Nico Bertram
Department of Computer Science, Technische Universität Dortmund, Germany

Jonas Ellert
Department of Computer Science, Technische Universität Dortmund, Germany

Johannes Fischer
Department of Computer Science, Technische Universität Dortmund, Germany

Abstract

Suffix sorting is arguably the most fundamental building block in string algorithmics, like regular sorting in the broader field of algorithms. It is thus not surprising that the literature is full of algorithms for suffix sorting, in particular focusing on their practicality. However, the advances on practical suffix sorting stalled with the emergence of the DivSufSort algorithm more than 10 years ago, which, up to date, has remained the fastest suffix sorter. This article shows how properties of Lyndon words can be exploited algorithmically to accelerate suffix sorting again. Our new algorithm is 6–19% faster than DivSufSort on real-world texts, and up to three times as fast on artificial repetitive texts. It can also be parallelized, where similar speedups can be observed. Thus, we make the first advances in practical suffix sorting after more than a decade of standstill.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Suffix array, suffix sorting, Lyndon words, string algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.15

Supplementary Material *Software*: `https://github.com/jonas-ellert/gsaca-double-sort`
Software: `https://github.com/jonas-ellert/gsaca-lyndon`

Funding This work has been supported by the German Research Association (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A6.

1 Introduction & Related Work

Since its introduction [13] in 1990, the suffix array – storing the order of the lexicographically sorted suffixes – has become one of the most important data structures in the field of string processing. Its applications include text indexing, text compression, and in particular the construction of the Burrows–Wheeler transformation.

From a theoretical point of view, the story is almost over: the suffix array can be computed in asymptotically optimal $\mathcal{O}(n)$ time and using only $\mathcal{O}(1)$ additional words of working space [9][1]. However, the fast *practical* construction of the suffix array remains an active topic of research. The efficiency of existing suffix sorters varies immensely [2], and the worst-case time and space bounds do not accurately predict the real world performance. In fact, the practically fastest and also highly memory efficient algorithm DivSufSort is not amongst the linear time algorithms [6], and has remained on the top of the scoreboard ever since its introduction more than a decade ago.

In 2016, Baier introduced the algorithm GSACA [4], which is the first to construct the suffix array in linear time without using recursion. It utilizes properties of so-called *Lyndon words*, which are strings that are lexicographically smaller than all of their proper

[1] for integer alphabet $[1, \sigma]$ with $\sigma \leq n$

suffixes (for example, `artist` is a Lyndon word; `concert` is not a Lyndon word because it is lexicographically larger than its suffix `cert`). Conceptually, the algorithm consists of two phases. Franek et al. showed that the first phase computes (a partially sorted version of) the Lyndon array (a definition follows later), which is then used in the second phase to induce the suffix array [8]. Despite its interesting theoretical properties, GSACA cannot compete with the best suffix sorters in practice.

Our Contributions. We make the first advances on practical suffix sorting since the introduction of DivSufSort more than a decade ago. Our starting point is the GSACA algorithm, but we show how special properties of Lyndon words allow us to use fast integer sorting algorithms for its two phases. As a result, we obtain an efficient algorithm that is also easy to parallelize. Our sequential implementation is around 6–19% faster than DivSufSort on real-world inputs, and up to three times as fast on artificial repetitive inputs. However, it comes at the cost of a larger memory footprint (even though we still use much less space than the original implementation of GSACA). Our parallelization scales well up to at least 16 cores, and on large inputs it is faster than Labeit's parallel implementation of DivSufSort [12], the currently fastest shared memory suffix sorter.

The rest of the paper is organized as follows: In Section 2 we introduce the definitions and notation that we use throughout the paper. We explain our new version of GSACA in Section 3, and give implementation details and a description of our parallelization in Section 4. We conclude the paper with an experimental evaluation in Section 5.

2 Preliminaries

We write $\lg x$ for $\log_2 x$. For $i, j \in \mathbb{N}$, we use the closed, half-open, and open interval notations $[i, j] = [i, j+1) = (i-1, j] = (i-1, j+1)$ to represent the set $\{x \mid x \in \mathbb{N} \wedge i \leq x \leq j\}$. Our analysis is performed in the word RAM model [10], where we can perform fundamental operations (logical shifts, basic arithmetic operations etc.) on words of size w bits in constant time. For the input size n of our problems we assume $\lceil \lg n \rceil \leq w$.

A *string* (also called *text*) over the *alphabet* Σ is a finite sequence of *symbols* from the finite and totally ordered set Σ. We say that a string S has length n and write $|S| = n$, if S is a sequence of exactly n symbols. The string of length 0 is called empty string and denoted by ϵ. The i-th symbol of a string S is denoted by $S[i]$, while the *substring* from the i-th to the j-th symbol is denoted by $S[i..j]$. For $i > j$ we define $S[i..j] = \epsilon$. For convenience, we use the interval notations $S[i..j+1) = S(i-1..j] = S(i-1..j+1) = S[i..j]$. The i-th *suffix* of S is defined as $S_i = S[i..n]$, while the substring $S[1..i]$ is called *prefix* of S. A prefix or suffix of S is called *proper*, if and only if its length is at least 1 and at most $n-1$. Let S and T be two strings over Σ of lengths n and m, respectively. The concatenation of S and T is denoted by ST. The length of the *longest common prefix (LCP)* between S and T is defined as $\textsc{lcp}(S, T) = \max\{\ell \mid \ell \in [0, \min(n, m)] \wedge S[1..\ell] = T[1..\ell]\}$. The *longest common extension (LCE)* of indices i and j is the length of the LCP between S_i and S_j, i.e. $\textsc{lce}(i, j) = \textsc{lcp}(S_i, S_j)$. The total order on Σ induces a total order on the set Σ^* of strings over Σ. Let S and T be strings over Σ, and let $\ell = \textsc{lcp}(S, T)$. We say that S is lexicographically smaller than T and write $S \prec T$, if and only if either $\ell = n < m$ (i.e. S is a prefix of T) or $\ell < \min(n, m) \wedge S[\ell+1] < T[\ell+1]$. We write $S \preceq T$ to denote $S \prec T \vee S = T$.

We can simplify the description of our algorithm with a special symbol $\$ \notin \Sigma$ that is smaller than all symbols from Σ. We say that S is *null-terminated*, if $S[n] = \$ \wedge \forall i \in [1, n) : S[i] \neq \$$.

Lexicographical Ordering of Suffixes. The *suffix array* lexicographically orders the suffixes of a string. To save space we only store the starting index of each suffix.

▶ **Definition 1** (Suffix Array). *Given a string S of length n, its suffix array* A *is the unique permutation of* $[1, n]$ *that satisfies* $S_{\mathsf{A}[1]} \prec S_{\mathsf{A}[2]} \prec \ldots \prec S_{\mathsf{A}[n]}$*. The inverse suffix array* A^{-1} *is the inverse permutation of* A*, i.e.* $\forall i \in [1, n] : \mathsf{A}^{-1}[\mathsf{A}[i]] = i$*.*

Baier's algorithm computes the suffix array by exploiting properties of *Lyndon words*. A Lyndon word is a string that is lexicographically smaller than all of its proper suffixes, i.e. S is a Lyndon word if and only if $\forall i \in [2..n] : S \prec S_i$ [5]. The Lyndon array of S identifies the longest Lyndon word starting at each text position.

▶ **Definition 2** (Lyndon Array). *Given a string S of length n, its Lyndon array λ is defined by* $\forall i \in [1, n] : \lambda[i] = \max\{\ell \mid \ell \in [1, n - i + 1] \wedge S[i..i + \ell) \text{ is a Lyndon word}\}$*. We write* $w_\lambda(i) = S[i..i + \lambda[i])$ *to denote the longest Lyndon word that starts at index i.*

An important property of the Lyndon array is that it inherently encodes some information about the lexicographical ordering of the suffixes.

▶ **Lemma 3** ([7, Lemma 15]). *Let S be a string of length n with Lyndon array λ, and let $i \in [1, n]$. It holds $\lambda[i] = \ell$ if and only if*

$$(i + \ell \le n + 1) \ \wedge \ (i + \ell \le n \implies S_i \succ S_{i+\ell}) \ \wedge \ (\forall j \in (i, i + \ell) : S_i \prec S_j).$$

We conclude the preliminaries by showing two relations between Lyndon words and the lexicographical order of suffixes:

▶ **Lemma 4.** *Let S be a string of length n, let λ be its Lyndon array, and let $i, j \in [1, n]$ be arbitrary indices. If $w_\lambda(i) \prec w_\lambda(j)$, then $S_i \prec S_j$.*

Proof. If $w_\lambda(i) \prec w_\lambda(j)$, then $w_\lambda(j)$ is not a prefix of $w_\lambda(i)$. If also $w_\lambda(i)$ is not a prefix of $w_\lambda(j)$, then the first mismatch between $w_\lambda(i)$ and $w_\lambda(j)$ determines the lexicographical order of S_i and S_j. Thus we only have to consider the case where $w_\lambda(j) = w_\lambda(i)\alpha$ for a non-empty string α. Let $S[i..i + \lambda[j]) = w_\lambda(i)\beta$, then it must hold $\beta \prec \alpha$ and thus also $S_i \prec S_j$. Otherwise, [5, Prop. 1.5] would imply that $w_\lambda(i)\beta$ is a Lyndon word. ◀

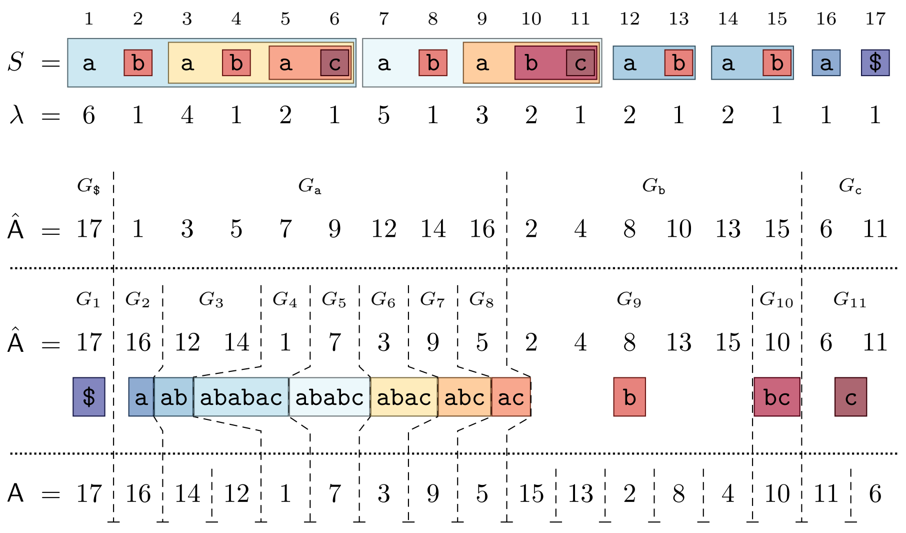
▶ **Lemma 5.** *Let $S_i = \alpha S_j$ (with $j = i + |\alpha|$) be a suffix of a string, where α is a Lyndon word. It holds $S_i \succ S_j \iff w_\lambda(i) = \alpha$. If $S_i \prec S_j$, then $\alpha w_\lambda(j)$ is a Lyndon word.*

Proof. Lemma 3 directly implies $S_i \succ S_j \iff w_\lambda(i) = \alpha$. Assume $S_i \prec S_j$, then Lemma 4 implies $w_\lambda(j) \succeq \alpha$. If $w_\lambda(j) = \alpha$, then due to Lemma 3 it holds $S_j \succ S_{j+|\alpha|}$, which leads to the contradiction $S_i = \alpha S_j \succ \alpha S_{j+|\alpha|} = S_j$. Thus it holds $w_\lambda(j) \succ \alpha$, and [5, Prop. 1.3] implies that $\alpha w_\lambda(j)$ is a Lyndon word. ◀

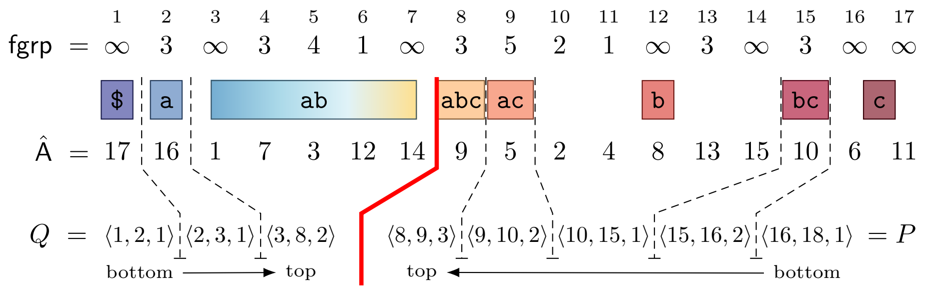
3 Sequential Algorithm

We start by giving a high level description of Baier's algorithm. For clarity, we write $\hat{\mathsf{A}}$ to denote the *not yet computed* suffix array, i.e. an array that serves as preliminary storage during the execution of the algorithm, and ultimately becomes the actual suffix array A. We use the terms suffix and index interchangeably. The algorithm consists of three main steps. For each step, we provide an example in Figure 1a.

Initialization: We sort and group the suffixes by their first symbol. The suffixes of each group are stored in increasing index order in a consecutive interval of $\hat{\mathsf{A}}$, and the order of the intervals is determined by the rank of the starting symbols. In our example, the order of the alphabet is `$` < `a` < `b` < `c`. Thus, the leftmost group contains the suffixes that start with `$`, followed by the group of suffixes that start with `a`, and so forth.



(a) Baier's algorithm. Initially, we group the indices by symbol (above the first dotted line). In the first phase, we group the indices by longest Lyndon words (below the first dotted line). In the second phase, we lexicographically sort the suffixes within each group (below the second dotted line).



(b) Data structures during Phase 1. The stack Q contains groups that we may still have to refine (left of the red line). The stack P contains only final groups (right of the red line).

Figure 1 Baier's algorithm and data structures used during Phase 1. The colored boxes represent the group contexts, which are also exactly the longest Lyndon words in **(a)**. (Best viewed in color.)

Phase 1: We refine the groups such that two suffixes S_i and S_j belong to the same group if and only if they share the longest Lyndon word $w_\lambda(i) = w_\lambda(j)$. Again, the indices of each group are stored in increasing order in a consecutive interval of $\hat{\mathsf{A}}$. The order of the groups is determined by the lexicographical order of the Lyndon words. In our example it holds $w_\lambda(3) = \texttt{abac} \prec \texttt{abc} = w_\lambda(9)$, and thus the group containing index 3 is stored to the left of the group containing index 9. From Lemma 4 follows that the grouping after Phase 1 is compatible with the suffix array.

Phase 2: We lexicographically sort the suffixes within each group to obtain the suffix array.

Baier uses a special form of induced copying (see e.g. [11, 14]) for Phase 1 and 2, which is elegant but results in a noncompetitive practical performance (see [4, Table 2] and [3, Chapter 6]). In the remainder of this section we explain how to instead use integer sorting for these phases, which allows a more efficient implementation of the algorithm. First, we give a formal definition of the grouping structure.

▶ **Definition 6** (Suffix Grouping). *A* suffix grouping *consists of an array* $\hat{\mathsf{A}}$ *and a list* $G_1, \ldots, G_m$ *of* groups. *A group with* context $\alpha \in \Sigma^+$ *is a triple* $\langle \ell, r, |\alpha| \rangle$ *with* $\ell, r \in [1, n]$ *and* $\ell < r$*, where the following 3 properties hold.*

1. *The interval* $\hat{\mathsf{A}}[\ell..r)$ *contains exactly the elements of* $\mathsf{A}[\ell..r)$ *in increasing (by text position) order. We write* $i \in \langle \ell, r, |\alpha| \rangle$ *to denote* $\exists x \in [\ell, r) : \hat{\mathsf{A}}[x] = i$.
2. *All the suffixes share prefix* α*, i.e.* $\forall i \in \langle \ell, r, |\alpha| \rangle : S_i = \alpha S_{i+|\alpha|}$.
3. *The context* α *is a Lyndon word.*

The groups $G_1, \ldots, G_m$ *form a partition of* $\hat{\mathsf{A}}$ *as follows. Let* $G_i = \langle \ell_i, r_i, |\alpha_i| \rangle$ *be the* i*-th group, then it holds* $\ell_1 = 1$ *and* $r_m = n + 1$. *For* $i \in [2, n]$ *it holds* $\ell_i = r_{i-1}$. *We write* $G_i \prec G_j \iff i < j$ *to denote that any suffix in* G_i *is lexicographically smaller than any suffix in* G_j*, which also means that the context of* G_i *is lexicographically not larger than the context of* G_j. *A group* $\langle \ell, r, |\alpha| \rangle$ *is called* final *if* $\forall i \in \langle \ell, r, |\alpha| \rangle : w_\lambda(i) = \alpha$.

As mentioned earlier, the initial suffix grouping partitions the suffixes by their first symbol, which can be implemented as follows. We stably sort the array $\hat{\mathsf{A}} = [1, 2, \ldots, n]$ in increasing order, using key $S[i]$ for entry i. After that, we determine the group borders with a simple scan over the sorted suffixes. We store the groups on a stack Q, where the bottommost element is the leftmost group, and the topmost element is the rightmost group.

3.1 Phase 1 with Integer Sorting

The goal of Phase 1 is to sort the group contexts lexicographically. To this end, we refine the groups by splitting them into subgroups with possibly longer contexts. The general idea is as follows. For any index i in a group $G_k = \langle \ell, r, |\alpha| \rangle$, let $j = i + |\alpha|$ be the position right after the context. If $S_i \succ S_j$ then $w_\lambda(i) = \alpha$ (Lemma 5), and we place i into a *final* subgroup with unchanged context α. If however $S_i \prec S_j$, then $\alpha w_\lambda(j)$ is a Lyndon word (Lemma 5), and we place i into a subgroup with context $\alpha w_\lambda(j)$. We repeatedly refine the subgroups in the same way, until all groups are final. At the point in time at which we refine the group $G_k = \langle \ell, r, |\alpha| \rangle$, the data structures used by our Phase 1 algorithm are the following (see Figure 1b for an example):

- A stack Q contains groups $G_1, \ldots, G_k$ that form a partition of $\hat{\mathsf{A}}[1..r)$. The groups are stored in increasing lexicographical order (the bottommost group G_1 is lex. smallest, the topmost group G_k is lex. largest).
- A stack P contains final groups $F_1, \ldots, F_h$ that form a partition of $\hat{\mathsf{A}}[r..n]$. The groups are stored in decreasing lexicographical order (the bottommost group F_1 is lex. largest, the topmost group F_h is lex. smallest).
- Together with the array $\hat{\mathsf{A}}$, the groups $G_1, \ldots, G_k, F_h, \ldots, F_1$ are a suffix grouping according to Definition 6.
- A length-n array fgrp maps suffixes to their final groups. If $\exists x \in [1, h] : i \in F_x$, then $\mathsf{fgrp}[i] = x$. Otherwise, $\mathsf{fgrp}[i] = \infty$. Note that fgrp inherently encodes information about the lexicographical order of suffixes due to $\forall i, j \in [1, n] : \mathsf{fgrp}[i] < \mathsf{fgrp}[j] \implies S_i \succ S_j$.

Now we describe Phase 1 in detail. The description is accompanied by pseudocode in Algorithm 1. The algorithm takes the array $\hat{\mathsf{A}}$ and the stack Q from the initialization as input. The stack P is initially empty, and all entries of fgrp are set to ∞ (lines 1–2). During the execution of the algorithm, we may mark the groups on the stack Q as *ready* (indicating that the group can easily be refined) or *final* (indicating that no further refinement is necessary). Initially, all groups are unmarked. The refinement is performed in a simple loop. While the stack Q is not empty, we pop the topmost group $G_k = \langle \ell, r, |\alpha| \rangle$ and process it (lines 3–4). Depending on the marking of the group, there are three possible cases:

Algorithm 1 Phase 1 with integer sorting.

Input: Initial suffix grouping represented by array Â and stack Q (all groups unmarked).
Output: Final suffix grouping represented by array Â and stack P.

```
1:  $P \leftarrow$ empty stack
2:  for $i \in [1, n]$ do $\mathsf{fgrp}[i] \leftarrow \infty$
3:  while $Q$ is not empty do
4:      $\langle \ell, r, |\alpha| \rangle \leftarrow Q.pop()$
5:      if $\langle \ell, r, |\alpha| \rangle$ is marked final $\vee\ \ell = 1$ then
6:          $P.push(\langle \ell, r, |\alpha| \rangle)$
7:          for $i \in [\ell, r)$ do
8:              $\mathsf{fgrp}[\hat{A}[i]] \leftarrow |P|$
9:      else if $\langle \ell, r, |\alpha| \rangle$ is marked ready then
10:         Stably sort $\hat{A}[\ell..r)$ in decreasing order,
            using key $\mathsf{fgrp}[\hat{A}[i] + |\alpha|]$ for entry $\hat{A}[i]$.
11:         $\ell' \leftarrow \ell$
12:         for $i \in (\ell, r)$ in increasing order do
13:             if $\mathsf{fgrp}[\hat{A}[i-1] + |\alpha|] \neq \mathsf{fgrp}[\hat{A}[i] + |\alpha|]$ then
14:                 Let $\beta$ be the context of $F_{\mathsf{fgrp}[\hat{A}[i-1]+|\alpha|]}$.
15:                 $Q.push(\langle \ell', i, |\alpha\beta| \rangle)$
16:                 $\ell' \leftarrow i$
17:         Let $\beta$ be the context of $F_{\mathsf{fgrp}[\hat{A}[r-1]+|\alpha|]}$.
18:         $Q.push(\langle \ell', r, |\alpha\beta| \rangle)$
19:     else
20:         for $i \in [\ell, r)$ in decreasing order do
21:             if $i < r - 1 \wedge \hat{A}[i+1] = \hat{A}[i] + |\alpha|$ then
22:                 if $sg(\hat{A}[i+1]) = \infty$ then $sg(\hat{A}[i]) \leftarrow \infty$
23:                 else $sg(\hat{A}[i]) \leftarrow sg(\hat{A}[i+1]) + 1$
24:             else
25:                 if $\mathsf{fgrp}[\hat{A}[i] + |\alpha|] = \infty$ then $sg(\hat{A}[i]) \leftarrow \infty$
26:                 else $sg(\hat{A}[i]) \leftarrow 1$
27:         Stably sort $\hat{A}[\ell..r)$ in decreasing order,
            using key $sg(\hat{A}[i])$ for entry $\hat{A}[i]$.
28:         $\ell' \leftarrow \ell$
29:         while $sg(\hat{A}[\ell']) = \infty$ do $\ell' \leftarrow \ell' + 1$
30:         if $\ell' > \ell$ then $Q.push(\langle \ell, \ell', |\alpha| \rangle)$ marked final
31:         for $i \in (\ell', r)$ in increasing order do
32:             if $sg(\hat{A}[i-1]) \neq sg(\hat{A}[i])$ then
33:                 $Q.push(\langle \ell', i, |\alpha| \rangle)$ marked ready
34:                 $\ell' \leftarrow i$
35:         $Q.push(\langle \ell', r, |\alpha| \rangle)$ marked ready
```

final group

ready group

unmarked group

The group is marked final, or it is the group containing only the lexicographically smallest suffix $S_n = \$$ (which gets processed last). During an earlier processing step, we have already ensured that $\forall i \in G_k : w_\lambda(i) = \alpha$. We simply push the group onto the final stack P, which now contains $|P| = h + 1$ groups. We update the array fgrp accordingly by assigning $\mathsf{fgrp}[i] \leftarrow h + 1$ for each index $i \in G_k$ (lines 5–8).

The group is marked ready. During an earlier processing step, we have already ensured that for each index $i \in G_k$, the index $i + |\alpha|$ is contained in a group that is lexicographically larger than G_k. Since all the lexicographically larger groups are final, it holds $\mathsf{fgrp}[i + |\alpha|] \neq \infty$. Particularly, for any two indices $i, j \in G_k$ it holds $\mathsf{fgrp}[i + |\alpha|] > \mathsf{fgrp}[j + |\alpha|] \implies S_i \prec S_j$ and $\mathsf{fgrp}[i + |\alpha|] < \mathsf{fgrp}[j + |\alpha|] \implies S_i \succ S_j$. We sort the interval $\hat{\mathsf{A}}[\ell..r)$ in *decreasing* order, using key $\mathsf{fgrp}[i + |\alpha|]$ for index i (line 10). Indices that share the same key x form a subgroup H_x (we again determine the group borders by scanning; lines 12–13). Let β be the context of the final group F_x, then $\alpha\beta$ becomes the context of subgroup H_x (lines 14–15 and 17–18). We push the subgroups onto the stack Q in lexicographically increasing order. All subgroups are unmarked.

The group is unmarked. We have already seen that ready and final groups are relatively easy to process. In this processing step, we split an unmarked group into at most one final subgroup H_∞, and possibly multiple ready subgroups $H_1, H_2, \ldots H_m$ (where m is unknown in advance). The lexicographical order of subgroups is $H_\infty \prec H_m \prec \ldots \prec H_1$. All subgroups have unchanged context α; the context extension only takes place when processing the ready subgroups. The subgroup H_∞ will contain exactly the indices $i \in G_k$ with $w_\lambda(i) = \alpha$. Lemma 4 implies that these suffixes are lexicographically smallest amongst the suffixes in G_k. Consider any index $i \in G_k$, and let x be the smallest positive integer such that $i' = i + x \cdot |\alpha| \notin G_k$. It is easy to see that $S_i = \alpha^x S_{i'}$.

- If i' is in one of the lexicographically smaller groups $G_1, \ldots, G_{k-1}$, then $S_i \succ S_{i'}$ and simple properties of the lexicographical order imply $S_i \succ S_{i+|\alpha|} \succ S_{i+2|\alpha|} \succ \ldots \succ S_{i+x|\alpha|}$. It follows from Lemma 3 that $w_\lambda(i) = \alpha$, and we place i into subgroup H_∞. Note that if $x > 1$ and $i \in H_\infty$, then $i + |\alpha| \in H_\infty$.
- If i' is in one of the lexicographically larger groups $F_1, \ldots, F_h$, then $S_i \prec S_{i'}$ and simple properties of the lexicographical order imply $S_i \prec S_{i+|\alpha|} \prec S_{i+2|\alpha|} \prec \ldots \prec S_{i+x|\alpha|}$. It follows from Lemma 3 that $\lambda[i] > |\alpha|$, and we place i into subgroup H_x. Note that if $x > 1$ and $i \in H_x$, then $i + |\alpha| \in H_{x-1}$. Thus H_x can be marked ready.

  Now we show that in fact $H_m \prec H_{m-1} \prec \ldots \prec H_1$. Assume that we place two indices i and j into subgroups H_x and H_y respectively. We have to show that $(x > y \implies S_i \prec S_j)$ and $(x < y \implies S_i \succ S_j)$. If $x > y$, then $S_i = \alpha^y S_{i+y\cdot|\alpha|}$ and $S_j = \alpha^y S_{j+y\cdot|\alpha|}$. Because of $x > y$ it holds $i + y \cdot |\alpha| \in G_k$, while $j + y \cdot |\alpha|$ is in one of the lexicographically larger groups $F_1, \ldots, F_h$. Thus it holds $S_{i+y\cdot|\alpha|} \prec S_{j+y\cdot|\alpha|}$ and therefore also $S_i \prec S_j$. The proof of $x < y \implies S_i \succ S_j$ works analogously.

As seen above, if both i and $i + |\alpha|$ are in G_k, then it holds $i + |\alpha| \in H_\infty \implies i \in H_\infty$ and $i + |\alpha| \in H_{x-1} \implies i \in H_x$. In such cases, we can easily compute i's subgroup from $(i + |\alpha|)$'s subgroup. We only need an efficient way to check whether $i + |\alpha| \in G_k$ actually holds. Conveniently, $i + |\alpha| \in G_k$ if and only if i and $i + |\alpha|$ are *neighboring* entries in $\hat{\mathsf{A}}[\ell..r)$. This is due to the fact that there cannot be a suffix $S_j = \alpha S_{j+|\alpha|}$ with $j \in (i, i + |\alpha|)$ (otherwise α would have a proper prefix that is also a proper suffix, which contradicts the definition of Lyndon words).

Lines 19–35 of Algorithm 1 describe our strategy for unmarked groups in technical detail. First, we compute a key $sg(\hat{\mathsf{A}}[i])$ for each $i \in [\ell, r)$ in decreasing order, indicating that we place index $\hat{\mathsf{A}}[i]$ into subgroup $H_{sg(\hat{\mathsf{A}}[i])}$. If for some index $\hat{\mathsf{A}}[i]$ it holds $\hat{\mathsf{A}}[i] + |\alpha| \in G_k$, i.e. if $\hat{\mathsf{A}}[i]$ and $\hat{\mathsf{A}}[i] + |\alpha|$ are neighbors in $\hat{\mathsf{A}}[\ell..r)$, then we compute $\hat{\mathsf{A}}[i]$'s subgroup from $(\hat{\mathsf{A}}[i] + |\alpha|)$'s subgroup as described above (lines 21–23). Otherwise, we inspect $\mathsf{fgrp}[\hat{\mathsf{A}}[i]+|\alpha|]$ to decide whether we place $\hat{\mathsf{A}}[i]$ into subgroup H_∞ or subgroup H_1 (lines 24–26). Finally, we rearrange $\hat{\mathsf{A}}[\ell..r)$ according to the new subgroups (line 27). We push the subgroups onto stack Q in increasing lexicographical order (once again computing the group borders with a simple scan). If H_∞ exists, then we mark it final (lines 29–30). All other groups are marked ready (lines 31–35).

3.2 Phase 2 with Integer Sorting

In the second phase, we lexicographically sort each final group, and simultaneously compute the inverse suffix array (see Algorithm 2). We proceed similarly to the first phase, using the stack P and the array $\hat{\mathsf{A}}$ as input. First, we pop the topmost group of P (which is the lexicographically smallest group containing only the special suffix $S_n = \$$), and assign $\mathsf{A}^{-1}[1] = n$ (line 1). Then we sort the remaining groups in a simple loop. While P is not empty, we pop the topmost group $F_h = \langle \ell, r, |\alpha| \rangle$ of the stack (lines 2–3) and lexicographically sort it. At this point in time, we have already sorted the suffixes of all groups that are lexicographically smaller than F_h because P contains the groups in lexicographical order (the topmost group is the lexicographically smallest unsorted group). Therefore, when processing F_h we have $\forall i \in [1,\ell) : \mathsf{A}^{-1}[\mathsf{A}[i]] = i$.

Since the group is final, it holds $S_{\hat{\mathsf{A}}[i]} \succ S_{\hat{\mathsf{A}}[i]+|\alpha|}$ for each $i \in [\ell, r)$. Thus $\hat{\mathsf{A}}[i] + |\alpha|$ either was in one of the lexicographically smaller groups that we have already sorted, or it holds $\hat{\mathsf{A}}[i] + |\alpha| \in F_h$. Ideally, we would simply sort $\hat{\mathsf{A}}[\ell..r)$ using key $\mathsf{A}^{-1}[\hat{\mathsf{A}}[i] + |\alpha|]$ for entry $\hat{\mathsf{A}}[i]$. However, if $\hat{\mathsf{A}}[i] + |\alpha| \in F_h$, then we have not computed $\mathsf{A}^{-1}[\hat{\mathsf{A}}[i] + |\alpha|]$ yet. We solve this problem by first rearranging F_h into subgroups $H_1 \prec H_2 \prec \ldots \prec H_m$, where subgroup H_x contains the suffixes that have prefix α^x, but not prefix α^{x+1} (the correctness of the lexicographical order of these subgroups can be shown similarly to the order of subgroups for unmarked groups in Phase 1). We assign the indices to the subgroups in decreasing order (line 4). If $\hat{\mathsf{A}}[i] + |\alpha| \notin F_h$ (as before, this is the case if and only if $\hat{\mathsf{A}}[i]$ and $\hat{\mathsf{A}}[i] + |\alpha|$ are not neighbors in $\hat{\mathsf{A}}[\ell..r)$), then we place $\hat{\mathsf{A}}[i]$ into subgroup H_1 (line 6). Otherwise, we have already placed $\hat{\mathsf{A}}[i] + |\alpha|$ into some subgroup H_{x-1}, and we place i into subgroup H_x (line 5). After we have rearranged the indices according to the new grouping (line 7), we finally sort each subgroup using key $\mathsf{A}^{-1}[\hat{\mathsf{A}}[i] + |\alpha|]$ for entry $\hat{\mathsf{A}}[i]$ (lines 8–15). Whenever we sort a group, we also update the inverse suffix array. Since no two indices $\hat{\mathsf{A}}[i]$ and $\hat{\mathsf{A}}[i] + |\alpha|$ are in the same subgroup, and due to the lexicographical order of subgroups, the required keys are always available once they are needed.

4 Implementation Details

Our C++17 implementation of the algorithm is publicly available on GitHub[2]. In general, it closely follows the description from Section 3. In this section, we discuss the choice of integer sorters as well as other practical optimizations, including the parallelization of the algorithm.

Sequential Implementation. The main computational effort of the algorithm lies in integer sorting, as well as in the computation of the keys prior to sorting. Finding the group borders after sorting only requires simple sequential scans that are cache efficient and very fast in

[2] `https://github.com/jonas-ellert/gsaca-double-sort`

Algorithm 2 Phase 2 with integer sorting.

Input: Final suffix grouping represented by array $\hat{\mathsf{A}}$ and stack P.
Output: Suffix array $\hat{\mathsf{A}}$ and inverse suffix array A^{-1}.

```
1: $P.pop()$;  $\mathsf{A}^{-1}[1] \leftarrow n$;
2: while $P$ is not empty do
3:     $\langle \ell, r, |\alpha| \rangle \leftarrow P.pop()$
4:     for $i \in [\ell, r)$ in decreasing order do
5:         if $i < r-1 \wedge \hat{\mathsf{A}}[i+1] = \hat{\mathsf{A}}[i] + |\alpha|$ then  $sg(\hat{\mathsf{A}}[i]) \leftarrow sg(\hat{\mathsf{A}}[i+1]) + 1$
6:         else  $sg(\hat{\mathsf{A}}[i]) \leftarrow 1$
7:     Stably sort $\hat{\mathsf{A}}[\ell..r)$ in increasing order, using key $sg(\hat{\mathsf{A}}[i])$ for entry $\hat{\mathsf{A}}[i]$.
8:     $\ell' \leftarrow \ell$
9:     for $i \in (\ell, r)$ in increasing order do
10:        if $sg(\hat{\mathsf{A}}[i-1]) \neq sg(\hat{\mathsf{A}}[i])$ then
11:            Sort $\hat{\mathsf{A}}[\ell'..i)$ in increasing order, using key $\mathsf{A}^{-1}[\hat{\mathsf{A}}[i] + |\alpha|]$ for entry $\hat{\mathsf{A}}[i]$.
12:            for $j \in [\ell'..i)$ do $\mathsf{A}^{-1}[\hat{\mathsf{A}}[i]] \leftarrow i$
13:            $\ell' \leftarrow i$
14:    Sort $\hat{\mathsf{A}}[\ell'..r)$ in increasing order, using key $\mathsf{A}^{-1}[\hat{\mathsf{A}}[i] + |\alpha|]$ for entry $\hat{\mathsf{A}}[i]$.
15:    for $j \in [\ell'..r)$ do $\mathsf{A}^{-1}[\hat{\mathsf{A}}[i]] \leftarrow i$
```

practice. We use two different sorting algorithms. Whenever the keys are from a small domain, we use a simple counting sort. This applies to the initialization (in practice we assume the byte alphabet $[0, 255]$), the processing of unmarked groups in the first phase, and the computation of subgroups in the second phase (Algorithm 1, line 27 and Algorithm 2, line 7). In both of the latter cases, we simultaneously compute the keys and count their frequencies. When refining ready groups in the first phase (Algorithm 1, line 10), and when performing the final sorting in the second phase (Algorithm 2, lines 11 and 14), we use MSD radix sort. We focus on speed and thus do not use an in-place variant of the sorter. Therefore, apart from the space needed for the suffix array, one auxiliary array (for fgrp and A^{-1}), and the stacks, we need additional space linear in the size of the largest group that we encounter.

We provide three versions DS1, DS3, and DSH of our sequential implementation that differ only in the initialization (where DS stands for *double sort* because we reduced both phases to integer sorting). The first version DS1 corresponds to the description in Section 3, i.e. we sort the suffixes by their first symbol. The second version DS3 directly sorts the suffixes by their first three symbols, resulting in smaller groups after the initialization (potentially resulting in a smaller memory footprint). Finally, the version DSH directly sorts the suffixes using key $w_\lambda(i)$ for suffix S_i. However, if $\lambda[i] > 8$ then we use key $w_\lambda(i)[1..8]$. Thus we immediately place all suffixes with $\lambda[i] < 8$ into their correct final groups, skipping many refinement steps and therefore accelerating Phase 1. The initialization of DSH consists of three steps.

Extract: We use a modification of Duval's algorithm [5] to compute for each $i \in [1, n]$ the key $x(i) = w_\lambda(i)$, if $\lambda[i] \leq 8$, or $x(i) = w_\lambda(i)[1..8]$ otherwise. We then use a hash table[3] to check whether this is the first time we have seen key $x(i)$. If it is, then we add the

[3] We use an implementation of Robin Hood hashing by Martin Ankerl, see `https://github.com/martinus/robin-hood-hashing`

tuple $\langle x(i), i\rangle$ to the table. Otherwise, let j be the index where we first discovered key $x(i)$. We store $\hat{\mathsf{A}}[i] = j$, indicating that we will place i into the same group as j.

Sort: We use comparison sorting to sort the tuples $\langle x(i), i\rangle$ by their first component.

Rearrange: We group the indices according to the sorted keys, and push the groups onto the stack Q. If a group has context α with $|\alpha| < 8$ then we mark it final.

Parallel Implementation. For our parallel algorithms we implemented the same ideas as described in Section 3 using OpenMP for parallelization. We parallelized the counting sort in the initialization as well as the processing of each suffix group in Phases 1 and 2. Because processing a single suffix group consists mainly of integer sorting and sequential scans we can parallelize them well in practice. We replace the sequential scans with parallel prefix sums. For integer sorting we use `ips4o` [1]. In case we have to sort a range $\hat{\mathsf{A}}[\ell..r)$ of elements stably we sort $\hat{\mathsf{A}}[\ell..r)$ by their keys, breaking ties by sorting elements with equal keys by their entry $\hat{\mathsf{A}}[i]$. Since for small suffix groups the overhead to process the group in parallel is too high compared to the sequential implementation we use a threshold. If the size of a suffix group is at most 1024, we switch to the sequential implementation (we performed a preliminary evaluation to determine a value that performs well in practice).

Unfortunately, we cannot achieve a perfect speedup. We simply cannot process multiple suffix groups in parallel because we heavily rely on processing them in decreasing lexicographical order. For that reason the running time of our parallelization is still linear with respect to the number of processed suffix groups (which is n in the worst case).

Similarly to our sequential implementation, we provide two different parallel versions PDS1 and PDS2. In PDS1 we sort the suffixes in the initialization by their first symbol using a parallel counting sort. In PDS2 we sort the suffixes by their first two symbols using `ips4o`.

5 Experimental Evaluation

We evaluated our algorithms on a number of real and artificial texts taken from the Pizza & Chili text corpus[4]. In Table 1 we give an overview of the used texts. We divide the texts into three different categories. Real texts (`PC-Real`) include `english`, `dna`, `sources`, `proteins` and `dblp.xml`. They are good examples of texts that occur in real-world applications. Real repetitive texts (`PC-Rep-Real`) include `cere`, `einstein.en.txt`, `kernel` and `para`. These texts might still occur in real-world applications, but they are rather repetitive and thus highly compressible. And lastly, artificial repetitive texts (`PC-Rep-Art`) include `fib41`, `rs.13` and `tm29`. These texts were created artificially with the goal of repetitiveness in mind. All the aforementioned texts are sufficiently short to compute the 32-bit suffix array. For our weak scaling experiments we use a collection of larger texts (`Large`) that are summarized at the bottom of Table 1. From each text we use a prefix of up to 16 GiB for our experiments, which means that 32 bits are not sufficient to address the suffixes. Thus, we compute the 64-bit suffix array instead.

5.1 Experimental Setup and Results

We conducted our experiments on a Linux machine with an AMD EPYC 7452 processor (32 cores, 2.35 GHz, L1 32K, L2 512K, L3 16M) and 1 TB of RAM. The code was compiled using GCC 7.5.0. We repeated all of our experiments five times and use the median as the

[4] `http://pizzachili.dcc.uchile.cl/`

Table 1 Texts from the Pizza & Chili text corpus and large texts. Apart from the text size n and the alphabet size σ, we provide the number of suffix groups (SGs) and the average size of the suffix groups that we process in Phase 1.

Category	Text	n (MiB)	σ	Count of SGs	Avg. SG Size
PC-Real	english	1,024	237	102,640,785	40.72
	dna	386	16	44,436,473	34.49
	sources	202	230	27,961,111	28.92
	proteins	1,024	27	144,323,711	28.62
	dblp.xml	283	97	16,091,809	71.9
PC-Rep-Real	cere	440	5	10,585,469	174.06
	einstein.en.txt	446	139	409,384	4,568.34
	kernel	247	160	4,027,324	256.15
	para	410	5	12,241,149	139.98
PC-Rep-Art	fib41	256	2	98	10,935,276.92
	rs.13	207	2	197	4,400,958.45
	tm29	256	2	194	5,534,751.23
Large	cc.txt	32,768	243	1,528,252,068	88.53
	dna.txt	32,768	4	2,955,252,973	44.56
	prot.txt	32,768	26	4,308,219,247	30.47

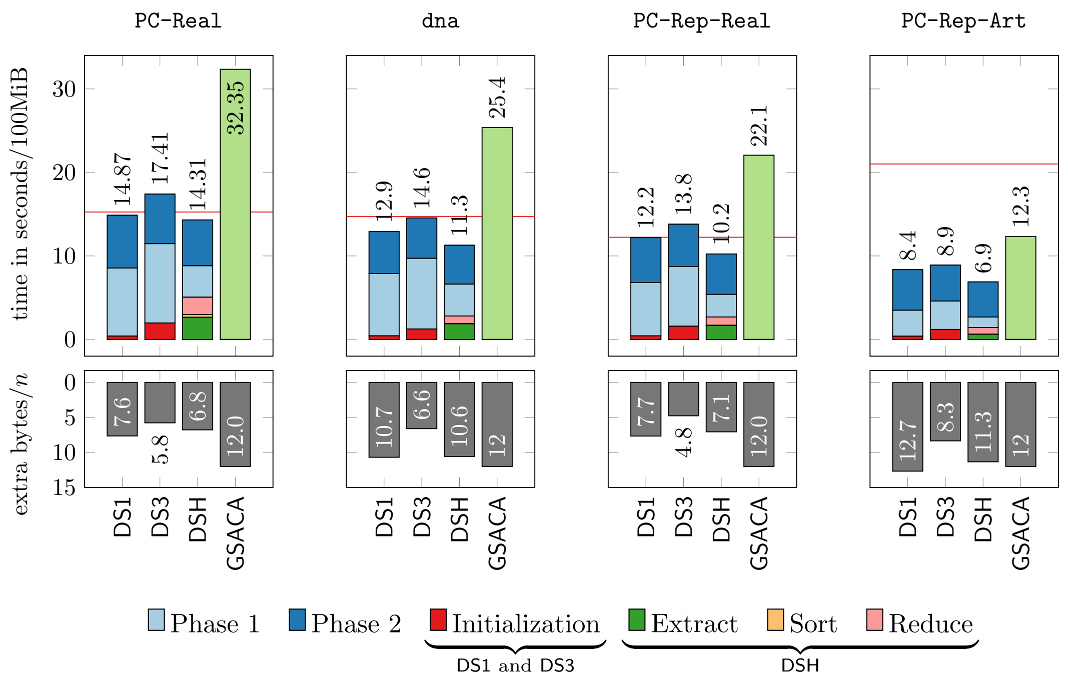


Figure 2 Running time and additional memory usage of the sequential algorithms averaged for each text category, and additionally for the text dna. The red line marks the running time of DivSufSort. We do not show the memory usage of DivSufSort because it requires only a very small constant amount of additional working space. (Best viewed in color.)

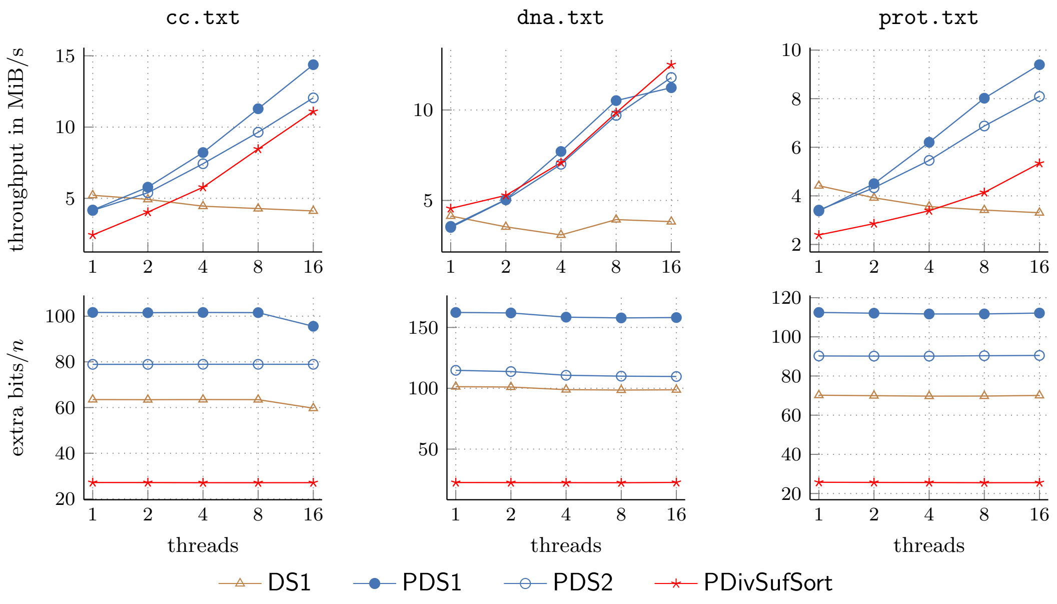


Figure 3 Throughput and additional memory usage for each parallel algorithm on large texts. The text size scales with the number of used threads (1GiB per thread). (Best viewed in color.)

final result. For the sequential experiments we compare our algorithms DS1, DS3, and DSH with Baier's original implementation of GSACA [4] and the currently fastest suffix sorter DivSufSort [6]. For the weak scaling experiments we compare our parallel algorithms PDS1 and PDS2 with Labeit's parallel implementation PDivSufSort of DivSufSort [12], and with our sequential DS1 as a baseline to see how well our parallelization scales.

Sequential Results. For each of the categories `PC-Real`, `PC-Rep-Real` and `PC-Rep-Art`, we averaged the running time and additional memory usage of the texts in each category. Before computing the average, we normalized the running time for each algorithm to show how long the algorithms take for 100 MiB of the input text. The additional memory is normalized as well to show the additional memory usage for each byte in the input text. The results for each category can be seen in Figure 2. Additionally, we provide a separate plot for `dna`, which is one of the most relevant text types in practice.

All of our sequential algorithms are significantly faster than Baier's original implementation. Our fastest sequential algorithm DSH is around twice as fast as Baier's algorithm, and is in each category even faster than DivSufSort (over 6% faster on `PC-Real`, over 19% faster on `PC-Rep-Real`, and over three times as fast on `PC-Rep-Art`). On `PC-Real` and `PC-Rep-Real`, the additional memory usage of our algorithms is much lower than Baier's algorithm. Of our new algorithms, DS3 has the lowest memory usage, but the running time is not as good as DS1 and DSH. None of our algorithms comes close to the memory usage of DivSufSort, which only uses a very small amount of constant additional memory.

Weak Scaling Results. Figure 3 shows the results of our weak scaling experiments. We calculate for each text the throughput (text size divided by running time) in MiB/s and the additional memory in bits/n for up to 16 threads. The input size grows proportionally to the number of used threads, such that the input size is p GiB when using p threads. We do not include results for more than 16 threads because otherwise the memory usage exceeds the maximum amount of 500 GB memory that we can address on a single NUMA node, and in some cases even the total memory of 1 TB (which is a limitation of our algorithms).

On `cc.txt` and `prot.txt`, our parallel variants are significantly faster than PDivSufSort for all input sizes. On `dna.txt`, the performance of all algorithms is similar. The memory usage of PDS1 is up to five times as large as PDivSufSort and the memory usage of PDS2 is up to four times as large as PDivSufSort on all input texts. The memory usage of DS1 is on all input texts lower than PDS1. This is due to the fact that the sequential algorithm uses 40-bit types for the additional arrays fgrp and A^{-1}. The parallel algorithms however use 64-bit types because the running time gets slower when using 40-bit types in parallel.

References

1 M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. In-place parallel super scalar samplesort (IPS^{4}o). In *25th European Symposium on Algorithms (ESA)*, pages 9:1–9:14, 2017. URL: `https://arxiv.org/abs/1705.02257`.

2 Johannes Bahne, Nico Bertram, Marvin Böcker, Jonas Bode, Johannes Fischer, Hermann Foot, Florian Grieskamp, Florian Kurpicz, Marvin Löbel, Oliver Magiera, Rosa Pink, David Piper, and Christopher Poeplau. SACABench: Benchmarking suffix array construction. In *Proceedings of the 26th International Symposium on String Processing and Information Retrieval (SPIRE 2019)*, pages 407–416, Segovia, Spain, 2019. `doi:10.1007/978-3-030-32686-9_29`.

3 Uwe Baier. Linear-time suffix sorting - A new approach for suffix array construction. Master's thesis, Ulm University, 2015.

4 Uwe Baier. Linear-time suffix sorting - A new approach for suffix array construction. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, Tel Aviv, Israel, 2016. `doi:10.4230/LIPIcs.CPM.2016.23`.

5 Jean Pierre Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4(4):363–381, 1983. `doi:10.1016/0196-6774(83)90017-2`.

6 Johannes Fischer and Florian Kurpicz. Dismantling DivSufSort. In *Proceedings of the 21st Prague Stringology Conference (PSC 2019)*, pages 62–76, Prague, Czech Republic, August 2017. URL: `http://www.stringology.org/event/2017/p07.html`.

7 Frantisek Franek, A. S. M. Sohidull Islam, Mohammad Sohel Rahman, and William F. Smyth. Algorithms to compute the Lyndon array. In *Proceedings of the 20th Prague Stringology Conference (PSC 2016)*, pages 172–184, Prague, Czech Republic, August 2016. URL: `http://www.stringology.org/event/2016/p15.html`.

8 Frantisek Franek, Asma Paracha, and William F. Smyth. The linear equivalence of the suffix array and the partially sorted Lyndon array. In *Proceedings of the 21st Prague Stringology Conference (PSC 2017)*, pages 77–84, Prague, Czech Republic, August 2017. URL: `http://www.stringology.org/event/2017/p08.html`.

9 Keisuke Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In *Proceedings of the 23rd Prague Stringology Conference (PSC 2019)*, pages 111–125, Prague, Czech Republic, 2019. URL: `http://www.stringology.org/event/2019/p11.html`.

10 Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998)*, pages 366–398, Paris, France, 1998. `doi:10.1007/BFb0028575`.

11 Florian Kurpicz. *Parallel Text Index Construction*. PhD thesis, Technical University of Dortmund, 2020.

12 Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and FM-index construction. *Journal of Discrete Algorithms*, 43:2–17, 2017. `doi:10.1016/j.jda.2017.04.001`.

13 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the 1st Annual Symposium on Discrete Algorithms (SODA 1990)*, pages 319–327, San Francisco, California, USA, 1990. URL: `https://dl.acm.org/doi/10.5555/320176.320218`.

14 Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007. `doi:10.1145/1242471.1242472`.