Lightweight Distributed Suffix Array Construction*

Johannes Fischer† Florian Kurpicz†

Abstract

We present two new distributed suffix array construction algorithms. One of our algorithms requires only half as much memory as its competitor (PSAC) [Flick & Aluru, SC 2015], while achieving similar speed. In practice, we can compute on the same hardware suffix arrays for text twice as large as PSAC. The other algorithm still requires less memory than PSAC but is faster on some instances. As a by-product, we also engineered the first distributed string sorting algorithm. All of our algorithms are tested on text collections of up to 115 GB and running on 1280 cores.

1 Introduction

The suffix array [18,30] is one of the most well-researched full-text indices, with numerous applications in text indexing, text mining, computational biology, and many more areas. We focus on its construction, which boils down to *sorting* the suffixes in lexicographic order. There exists plenty of work with respect to suffix array construction in main memory, e.g., [2, 4, 10, 12, 14, 19, 22, 24–26, 29, 31–33, 35–41, 43, 44]. Puglisi et al. [42] give an overview of suffix array construction algorithms (SACAs), and categorize SACAs in three categories: (1) *Prefix doubling* algorithms start with the length-1 prefix of each suffix and determine their ranks, i.e., the number of smaller length-1 prefixes. Next, those ranks are used to determine the ranks of the length-2 prefixes. The length of the prefixes is doubled during each iteration until all ranks are unique. (2) *Recursive* SACAs reduce the size of the text until all suffixes (in the reduced text) are unique, and then recursively solve the problem for the larger texts. Last, (3) *induced copying* algorithms sort a small subset of suffixes and conclude the lexicographical order of all other suffixes using the sorted suffixes. Recently, another type of SACA has been introduced: (4) *grouping* [4]. Here, suffixes are assigned to groups that are then refined (and thus implicitly sorted). This process is similar to induced copying, but the groups have more properties that are used to obtain a linear time algorithm. Some SACAs combine recursion and induced copying principles to obtain linear running time, e.g., [40].

1.1 Related Work. Here, we focus mainly on practical work. All SACAs mentioned before work in *main memory*, where the *DivSufSort* [15,35], an induced copying algorithm, is the fastest algorithm is practice, despite having a supralinear running time. In theory, the first linear time algorithm was Kärkkäinen and Sanders' well-known recursive Skew/DC3 algorithm [22] with its generalization DCX [23]. Looking at other models of computation, Labeit et al. [28] parallelized the DivSufSort in *shared memory*. The DCX algorithm can also be parallelized in this model [22]. In *external memory* there exist SACAs based on prefix doubling [11, 13], recursive (DCX) [13] and induced copying [7, 20]. In *distributed memory*, the model that we are considering in this paper (see §2.1), Flick and Aluru [17] implemented a prefix doubling algorithm (PSAC), which is our main competitor. The DCX algorithm can also be parallelized in this model [27, 34].

We briefly consider distributed string sorting in §5. To our best knowledge, there are no other *distributed* string sorting implementations available. Bingmann et al. [6] give an overview about the state of art in *shared memory* string sorting, which we use for comparison (to get a first feeling for the competitiveness of our implementation).

1.2 Our Contributions. The two main results of this paper are our two lightweight distributed SACAs:

- Our first SACA is a distributed prefix doubling algorithm that is similar in speed compared to PSAC, but requires less memory (in practice), see §3.
- Then, we present the *first* distributed induced copying SACA that is also the *most lightweight* distributed SACA currently available, requiring 50 % less memory than PSAC (in theory and practice), see §4.

As an additional result, we present the *first* practical distributed string sorting implementation, which is part

*This work was supported by the German Research Foundation (DFG) SPP 1736 priority programme "Algorithms for Big Data".

†Technische Universität Dortmund, Department of Computer Science, johannes.fischer@cs.tu-dortmund.de, florian.kurpicz@tu-dortmund.de

of our distributed induced copying SACA, see §5. Furthermore, we conducted experiments on texts of size up to 115 GB, which are at least an order of magnitude greater than previously used texts, e.g., [17, 27].

2 Preliminaries

Let $\mathsf{T} = \mathsf{T}[0] \dots \mathsf{T}[n-1]$ be a *text* of size n consisting of characters from an ordered alphabet Σ of size $\sigma := |\Sigma|$. Integers from i to $j-1$ are represented by $[i, j)$. We use the notation $\mathsf{T}[i, j)$ for the *substring* $\mathsf{T}[i] \dots \mathsf{T}[j-1]$ and call $\mathsf{P}_i := \mathsf{T}[0, i)$ the i-th *prefix* of T. Analogously, $\mathsf{S}_i := \mathsf{T}[i, n)$ is the i-th *suffix* of T. The *suffix array* (SA) of T is the permutation of $[0, n)$ such that $\mathsf{S}_{\mathsf{SA}[i]} < \mathsf{S}_{\mathsf{SA}[i+1]}$ for all $0 \leq i < n-1$.

2.1 Setting and Model of Computation.

In our setting, algorithms run on p distinct *processing elements* (PEs) that are connected by a network, e.g., are distributed in a cluster on different physical hardware, such as CPUs or CPU-cores. Each PE has a unique *rank* in the range from 0 to $p-1$. We analyze our algorithms in the *bulk-synchronous parallel* (BSP) model [45]. Here, each algorithm is a sequence of *supersteps*, with each superstep being split into three phases: First, the PEs can perform any number of operations based on local data. We use w to denote the maximum time used by a PE. Second, the PEs can send data to other PEs (communication phase). Here, h is the maximum number of machine words communicated by each PE, and G is the running time required for the communication of one machine word. Last, all PEs wait until every PE has finished the first two phases. L is the time of this barrier synchronization. There is no synchronization between the first and second phase. PEs can start communicating as soon as they have finished working on the local data (but data received during the communication is not available for local operations before the next barrier synchronization). Then, the total running time of a BSP-algorithm is the time of all its supersteps, where the time of one superstep is $w + hG + L$.

The required space is not considered in this model. However, it is important to note that we need to keep the data that is communicated available until it has been received by its recipient. Hence, sending h computer words requires $2h$ computer words of space in total, as also described in [17].

2.2 Distributed Arrays.

We use *distributed arrays* to store the suffixes in the different (sub-)classes $C_{\alpha\beta}$ for $\alpha, \beta \in \Sigma$. Here, each PE holds a consecutive slice of the array, i.e., given a distributed array C of length ℓ, each PE holds $\Theta(\frac{\ell}{p})$ elements, such that on PE i the j-th local element is the $j + i\lfloor\frac{\ell}{p}\rfloor$-th global element. A distributed array supports two operations: *pushback* and *pushfront* put data in the rightmost or leftmost unused space, resp. The execution of one operation takes one superstep, independently of the amount of data stored. Since the operation can be called from multiple PEs during one superstep, the data are stored in an order depending on the rank of the PE that sends the data, i.e., when we insert data into a distributed array (using *pushfront*) originating from PE i and PE j with $i < j \in [0, p)$, then all data sent by PE i will have a smaller index in the distributed array than any data sent by PE j (in the same superstep). The operations are executed *delayed*, i.e., elements are not stored immediately, but buffered until *communicate()* is called. We only insert data using the operations described above or access already stored data. When we say in "reverse order", we access all elements stored in the distributed array from right to left. We indicate two concatenated arrays using $\otimes$. In this case, the whole arrays are concatenated, not just the local slices.

3 On Distributed Prefix Doubling

The prefix doubling technique has been introduced by Manber and Myers [30] when they first introduced the suffix array. All prefix doubling SACAs share a common core that we describe in the following. Given a text T of size n:

1. Associate each index with a rank, i.e., the rank of the character $\mathsf{T}[i]$ among all characters occurring in T, thus creating rank-tuples $\langle i, r\rangle$. Also, let $k = 0$.
2. Check if all ranks are unique: if so, sort the tuples by the second component. Now, the first component corresponds to the SA. Otherwise, continue.
3. Construct rank-triples $\langle\langle i, r\rangle, r'\rangle$, where $\langle i, r\rangle$ is the previously considered rank-tuple and r' is the rank of the tuple with index $i + 2^k$ (or 0 if $i + 2^k \geq n$).
4. Sort the rank-triples by $\langle r, r'\rangle$ and determine new ranks, i.e., compute new rank-tuples. Then, increase k by one and continue with Step 2.

The concrete implementation depends on the model of computation. Most prefix doubling algorithms differ in Steps 3 and 4. Here, in Step 3 we sort the rank tuples such that the required ranks are (if it exists) next to each other. To this end, we sort the tuples using $(i \mod 2^k, i \operatorname{div} 2^k)$ as keys, where i is the index of the sorted tuples. Now, the pair containing the other rank in Step 4 is adjacent (if it exists).

Since we do not depend on the distance of two tuples anymore, we can ignore rank-tuples that are not required any more. We make use of a technique

called *discarding*, introduced by Dementiev et al. [13]: we discard all suffixes that (a) have a unique rank and (b) are not required anymore to determine a unique rank for a suffix that does not have a unique rank. Case (b) happens if the suffix occurs after a different unique suffix in text order. We can determine all rank tuples that can be discarded in one simple scan. In addition, when we update the ranks, we must consider the lexicographically smaller but discarded suffixes. To this end, each rank always corresponds to the greatest possible rank the suffix could potentially obtain, and is only decreased during the algorithm. Therefore, smaller discarded suffixes do not interfere with the update of the ranks. This allows us to significantly reduce the cost of global sorting, i.e., sorting all rank-tuples on all PEs.

The difference to the already existing distributed prefix doubling SACA [17] is that they do not discard rank-tuples that are not required anymore. Instead, they manually compute new ranks *without* sorting, as soon as the number of non-unique items has sunken below a threshold. This allows them to save one global sorting step (which is expensive in practice) during each iteration of the algorithm.

A thorough analysis of a distributed prefix doubling algorithm is given by Flick and Aluru [17]. The analysis is the same for our distributed prefix doubling SACA, as there is no theoretical difference between the two approaches.

4 Distributed Induced Copying Suffix Array Construction

Our algorithm is a distributed variant of DivSufSort [15, 35]. Given a text T of size n, we want to compute the SA. We assume that T is distributed among all PEs such that each PE holds a consecutive slice T' of size $n' = \Theta(\frac{n}{p})$. Thus, $\mathsf{T}'[j] \coloneqq \mathsf{T}[i\lfloor\frac{n}{p}\rfloor + j]$ on all PE i for $i \in [0, p)$ (similar to the distributed array). Similarly, S'_j denotes the j-th suffix of T' with respect to the whole text, i.e., on PE i we have $\mathsf{S}'_j = \mathsf{T}[i\lfloor\frac{n}{p}\rfloor + j, n)$ for $i \in [0, p)$.

4.1 Classification of Suffixes.

We use the classification introduced by by Itoh and Tanaka [19] to distinguish between two *classes* of suffixes (originally called type A and type B suffixes) in combination with a notation established by Kärkkäinen et al. [20] for a similar classification [40]. Here, suffixes are represented by their starting positions in T.

$$C^- \coloneqq \{i \in [0, n) \colon \mathsf{S}_i > \mathsf{S}_{i+1}\} \text{ and}$$
$$C^+ \coloneqq \{i \in [0, n) \colon \mathsf{S}_i < \mathsf{S}_{i+1}\}.$$

We say "a suffix S_i is in C" if $i \in C$, where C can denote any class. The last suffix S_{n-1} is in C^-, as the empty string is lexicographically smaller than any suffix. Whenever the class of two consecutive suffixes differs, we are interested in the suffix *before* the change.

$$C^{-\triangleright} \coloneqq \{i \in C^- \colon i+1 \in C^+\} \cup \{n-1\} \text{ and}$$
$$C^{+\triangleright} \coloneqq \{i \in C^+ \colon i+1 \in C^-\}.$$

We call $C^{-\triangleright}$ and $C^{+\triangleright}$ *sub-classes*. Suffixes in $C^{+\triangleright}$ are often called $B^\star$-suffixes [15, 35]. The last suffix S_{n-1} belongs to $C^{-\triangleright}$ by definition. Also, the number of suffixes in both $C^{-\triangleright}$ and $C^{+\triangleright}$ is at most $\frac{n}{2}$. These sub-classes are later used to identify fine-grained intervals in the SA that we make heavy use of during our inducing step, see §4.5. To this end, we also need all suffixes that are followed by a suffix in the same class:

$$C^{-\odot} \coloneqq C^- \setminus C^{-\triangleright} \quad \text{and} \quad C^{+\odot} \coloneqq C^+ \setminus C^{+\triangleright}.$$

We further need to filter suffixes by their first (two) characters. For any (sub-)class of suffixes C described above, let $\alpha, \beta \in \Sigma$. Then:

$$C_\alpha \coloneqq \{i \in C \colon \mathsf{T}[i] = \alpha\} \text{ and}$$
$$C_{\alpha\beta} \coloneqq \{i \in C_\alpha \colon \mathsf{T}[i+1] = \beta\}.$$

This allows us to implicitly sort all suffixes lexicographically based on their type and first (two) characters:

LEMMA 4.1. *Let be $\alpha, \beta \in \Sigma$, then*

1. $\mathsf{S}_i < \mathsf{S}_j$ *if* $i \in C^-_\alpha$ *and* $j \in C^+_\alpha$, *or if* $i \in C^{+\triangleright}_{\alpha\beta}$ *and* $j \in C^{+\odot}_{\alpha\beta}$, *and*
2. $\mathsf{S}_i > \mathsf{S}_j$ *if* $i \in C^{-\triangleright}_{\alpha\beta}$ *and* $j \in C^{-\odot}_{\alpha\beta}$,

Proof. The first statement is proved in [15, Lemma 1]. The second statement works analogously.

Let $\overrightarrow{C}$ denote the starting positions of the suffixes in C in lexicographical order. Using Lemma 4.1 we get the following observation. (See Figure 1 for an example.)

OBSERVATION 1. *We can express the* SA *as follows:*
$\mathsf{SA} = \overrightarrow{C^{-\odot}_{00}}\,\overrightarrow{C^{-\triangleright}_{00}}\,\overrightarrow{C^{+\triangleright}_{00}}\,\overrightarrow{C^{+\odot}_{00}}\,\overrightarrow{C^{-\odot}_{01}} \dots \overrightarrow{C^{-\odot}_{\sigma-1\sigma-1}} \dots \overrightarrow{C^{+\odot}_{\sigma-1\sigma-1}}$

We also can identify the (sub-)class of a suffix easily by its first character and the class of its succeeding suffix (in text order):

OBSERVATION 2. *For all* $i \in [1, n)$

1. $i-1 \in C^{+\odot} \iff i \in C^+$ *and* $\mathsf{T}[i-1] \le \mathsf{T}[i]$,
2. $i-1 \in C^{-\triangleright} \iff$ *either* $i = n$ *or* $i \in C^+$ *and* $\mathsf{T}[i-1] > \mathsf{T}[i]$,
3. $i-1 \in C^{-\odot} \iff i \in C^-$ *and* $\mathsf{T}[i-1] \ge \mathsf{T}[i]$, *and*
4. $i-1 \in C^{+\triangleright} \iff i \in C^-$ *and* $\mathsf{T}[i-1] < \mathsf{T}[i]$.

i	0	1	2	3	4	5	6	7	8	9
$\mathsf{T}[i]$	a	b	b	c	a	b	a	b	c	a
class	+	+	+	−	+	−	+	+	−	−
sub-class	⊙	⊙	▷	▷	▷	▷	⊙	▷	⊙	▷

(a)

i	0	1	2	3	4	5	6	7	8	9
$\mathsf{SA}[i]$	9	4	0	6	5	1	7	2	8	3
$C_{\alpha\beta}$	$	ab			ba	bb	bc		ca	
class	−	+	+	+	−	+	+	+	−	−
sub-class	▷	▷	⊙	⊙	▷	⊙	▷	▷	⊙	▷

(b)

Figure 1: Classification of suffixes in text order (a) and suffix array order (b).

4.2 General Overview. Using the classification, we can compute the SA in three steps:

1. On each PE, we compute $C^{+\triangleright}$ and the sizes of $C_{\alpha\beta}$ for all $\alpha, \beta \in \Sigma$ and (sub-)classes C for T'. The results are communicated to get those sizes for T.
2. Next, we sort all suffixes in $C^{+\triangleright}$ lexicographically to compute $\overrightarrow{C^{+\triangleright}}$ using a distributed string sorting algorithm.
3. Last, we induce $\overrightarrow{C^{+\odot}}$ and $\overrightarrow{C^{-\triangleright}}$ using $\overrightarrow{C^{+\triangleright}}$ and then $\overrightarrow{C^{-\odot}}$ using $\overrightarrow{C^{-\triangleright}}$.

We need w bytes to store an index position in the SA, usually $w = 4$ for smaller texts and $w = 5$ for larger texts (up to 1 TB).[1] Considering the size of the text, we assume that we need one byte per character, i.e., $\sigma < 256$. Thus, our SA is w times as large as T.

4.3 Identifying Suffixes in $C^{+\triangleright}$. We first need to identify those suffixes that are in $C^{+\triangleright}$, which can be done by a right to left scan of the text. The last suffix is in C^- and thus we only need to look at two consecutive characters to identify the type of a suffix:

OBSERVATION 3. *Let $i \in [0, n-1)$. We know that $n-1 \in C^-$. If $\mathsf{T}[i] > \mathsf{T}[i+1]$, then $i \in C^-$ and if $\mathsf{T}[i] < \mathsf{T}[i+1]$, then $i \in C^+$. Last, if $\mathsf{T}[i] = \mathsf{T}[i+1]$, then $i \in C^- \Leftrightarrow i+1 \in C^-$.*

In our distributed setting, only for PE $p-1$ the class of the last suffix is known. Hence, we cannot simply scan T' right to left on any PE but the last. Instead, we identify the first suffix S_i that is definitely in C^-, i.e., the position where $\mathsf{T}'[i] > \mathsf{T}'[i+1]$ with $i < n'-1$. Starting at this suffix, we can use Observation 3 to classify all suffixes S'_j with $j < i$. Next, we identify the classes of all remaining suffixes. To this end, each PE sends $\mathsf{T}'[0]$ and the class of S'_0 to all other PEs. The class on PE i can be *unknown*, i.e., there has been no suffix that is definitely in C^-. In this case, we can conclude the type of all suffixes on PE i using the class and the first character of the first suffix on PE $i+1$. Since the class is known on PE $p-1$, we can resolve the class of all received suffixes and thus, we can classify all suffixes that have not been classified, yet. Within those suffixes, there is at most one suffix in $C^{+\triangleright}$.

In total, we scan the local text at most twice, send $\mathcal{O}(1)$ and receive $\mathcal{O}(p)$ computer words in one communication phase.[2] This leads to the following Lemma:

LEMMA 4.2. *Identifying all suffixes in $C^{+\triangleright}$ costs $\mathcal{O}\left(\frac{n}{p} + pG + L\right)$ time.*

While identifying the suffixes in $C^{+\triangleright}$ we also compute the number of suffixes in all other (sub-)classes without an overhead in running time. We use these sizes later during inducing (see §4.5) to determine the positions of an induced suffix in the SA using Observation 1.

Space. We need at most $\frac{wn}{2}$ bytes to store $C^{+\triangleright}$ in a distributed array. Since we need to communicate the suffixes, this requires twice the amount of space, resulting in wn bytes. Storing all those positions requires $2\sigma^2 w$ bytes space on each PE, $\sigma^2 w$ bytes for all suffixes in C^- and the same amount for the suffixes in C^+.

4.4 Sorting Suffixes in $C^{+\triangleright}$. We compute $\overrightarrow{C^{+\triangleright}}$ in two steps. First, we sort the substrings between two adjacent positions in $C^{+\triangleright}$ (in text order). Formally, let $next(i) = \min\{j > i : j \in C^{+\triangleright} \cup \{n\}\}$. This allows us to define the *$C^{+\triangleright}$-ending substrings* $\mathsf{T}_i^{+\triangleright} = \mathsf{T}[i, \min\{next(i)+2, n\})$. (The additional two characters are important to correctly sort the $C^{+\triangleright}$-ending substrings, see Figure 2 for an example.) We sort the $C^{+\triangleright}$-ending substrings using a distributed string sample sort.

[1] In theory, $\lceil \lg n \rceil$ bits would be sufficient. In practice, we use a multiple of one byte for faster access.

[2] In practice, the communication overhead is very small ($p \ll n$) and there are (again, only in practice) no unknown suffixes. Still, we could further reduce the communication in exchange for more supersteps using a prefix sum-like approach to resolve unknown classes. This results in costs of $\mathcal{O}\left(\frac{n}{p} + G + \lg pL\right)$.

Substring s_1	a	b	a	a	b
Substring s_2	a	b	a	b	

Figure 2: Two $C^{+\triangleright}$-ending substrings. The underlined characters correspond to positions in $C^{+\triangleright}$. If we consider only the substrings starting and ending at those positions, s_2 is lexicographically smaller than s_1, as it is a prefix of s_1. This can be avoided by considering one additional character.

A detailed description of the sorting is given in §5.[3]

Here, the number of $C^{+\triangleright}$-ending substrings m is at most $\frac{n}{2}$ and the *distinguish prefix size* D (i.e., the number of characters that must be compared to sort the $C^{+\triangleright}$-ending substrings lexicographically, see §5 for a formal definition) is in practice roughly the same at each PE (which is confirmed by our experiments, see Table 1). Hence, we can sort the $C^{+\triangleright}$-ending substrings in $\mathcal{O}\left(\frac{n}{p}\lg\sigma + \frac{n}{p}G + L\right)$ time, using our distributed string sorter described, which employs sample sort and sorts the strings locally using multi-key radix sort.

Having sorted the $C^{+\triangleright}$-ending substrings, we sort all suffixes in $C^{+\triangleright}$ by using the ranks of the $C^{+\triangleright}$-ending substrings. We use an approach similar to prefix doubling (see §3) with one difference: instead of using the original input text, we use the ranks of the $C^{+\triangleright}$-ending substrings (in text order) as input T' for the prefix doubling algorithm. During each iteration we double the size of the considered prefixes in T'. Thus, we implicitly double the number of considered consecutive $C^{+\triangleright}$-ending substrings in T. Then, we compute the new ranks using the old ones until all ranks are unique, as described in the prefix doubling SACA in §3. Since the prefix doubling algorithm relies on indices in the range from 0 to m, where m is the number if considered suffixes/substrings, we must transform the text positions of the $C^{+\triangleright}$-ending substrings accordingly. When all suffixes in $C^{+\triangleright}$ are sorted, we reverse the transformation by first sorting the rank-tuples in text order, and then identifying all suffixes in $C^{+\triangleright}$ during a single scan of the text.

During the doubling approach, we have keys of fixed length, i.e., the ranks. Hence, we can employ a distributed sample sort [9] to sort them and do not rely on a distributed string sorter as before. To this end, we

1. sort the data locally in $\mathcal{O}\left(\frac{m}{p}\lg\frac{m}{p}\right)$ time and choose $p-1$ local splitters from the local data, such that the p partitions that are implicitly given by the splitters have the same size (up to rounding). Then, we
2. gather all local splitters (on all PEs) to determine $p-1$ global splitters (in the same way as in Step 1) and partition the local data in total $\mathcal{O}\left(p\lg p + \frac{m}{p}\lg p + pG + L\right)$ time using the global splitters. Next, we
3. distribute these partitions in $\mathcal{O}\left(\frac{m}{p}G + L\right)$ time, such that for any two PEs i, j with $i < j$ all elements on PE i are smaller than all elements on PE j. Finally, we
4. merge the received partitions locally in $\mathcal{O}\left(\frac{m}{p}\lg p\right)$ time to finish the distributed sample sort.

Thus, we can sort m elements in $\mathcal{O}\left(\frac{m}{p}\left(\lg\frac{m}{p} + \lg p\right) + p\lg p + \left(\frac{m}{p} + p\right)G + L\right)$ time. In our scenario, $m = \mathcal{O}(n)$ and we need to use the distributed sample sort $\mathcal{O}(\lg n)$ times.

LEMMA 4.3. *Sorting all suffixes in $C^{+\triangleright}$ lexicographically, i.e., computing $\overrightarrow{C^{+\triangleright}}$ costs $\mathcal{O}\left(\frac{n}{p}(\lg n + \lg p) + \left(\frac{n}{p} + p\right)G + \lg nL\right)$ time.*

Space. During the sorting, we need $2w$ bytes to represent the starting position of each suffix and its rank. When we compute the new ranks, we must sort the starting positions based on two ranks. This leads to $3w$ bytes per considered suffix (at most $\frac{n}{2}$). In total, this requires $3wn$ bytes as we need space to receive data.

4.5 Inducing the Suffix Array. Now, we compute the SA by inducing all other suffixes using only $\overrightarrow{C^{+\triangleright}}$ and T, without any sorting. First, we induce $\overrightarrow{C^{+\odot}}$ and $\overrightarrow{C^{-\triangleright}}$ from $\overrightarrow{C^{+\triangleright}}$ and the already induced suffixes in $\overrightarrow{C^{+\odot}}$. Then, we add the last suffix to its correct position in $C^{-\triangleright}$. Last, we induce $\overrightarrow{C^{-\odot}}$ from $\overrightarrow{C^{-\triangleright}}$, see Algorithm 1.

We assume that the sorted suffixes in $\overrightarrow{C^{+\triangleright}_{\alpha\beta}}$ are stored in distributed arrays for $\alpha, \beta \in \Sigma$. In general, all C-objects in the algorithm are distributed arrays. The algorithm runs on all PEs, and each PE only considers its own slice of the distributed arrays when reading from it (but the concatenation in lines 3 and 12 still affects the whole distributed array). We first induce from right to left (first inducing phase), i.e., in decreasing lexicographical order (see loop starting at line 1). Here, we induce all suffixes in $\overrightarrow{C^{+\odot}}$ and $\overrightarrow{C^{-\triangleright}}$. Next, we add the last suffix (line 9) before starting the second inducing phase. Last, we induce the suffixes in increasing lexicographical order (see loop starting at line 10).

During this step we require access to the text (from arbitrary PEs), since we need to identify the bucket we

[3] In §5, we describe general purpose string sorting algorithms. Sorting $C^{+\triangleright}$-ending substrings is just a special case.

Algorithm 1: Inducing

```
1  for α = σ − 1 down to 0 do
2  |  for β = σ − 1 down to α do
3  |  |  for i ∈ C^{+▷}_{αβ} ⊗ C^{+⊙}_{αβ} in reverse order do
4  |  |  |  if i > 0 and T[i − 1] ≤ α then
5  |  |  |  |  C^{+⊙}_{T[i−1]α}.pushfront(i − 1)
6  |  |  |  else if i > 0 then
7  |  |  |  |  C^{−▷}_{T[i−1]α}.pushfront(i − 1)
8  |  |  communicate()
9  C^{−⊙}_{T[n−1]0}.pushback(n − 1)
10 for α = 0 to σ − 1 do
11 |  for β = 0 to α do
12 |  |  for i ∈ C^{−⊙}_{αβ} ⊗ C^{−▷}_{αβ} do
13 |  |  |  if i > 0 and T[i − 1] ≥ α then
14 |  |  |  |  C^{−⊙}_{T[i−1]α}.pushpack(i − 1)
15 |  |  communicate()
```

induce the suffix into. To this end we distribute the text, such that the substring $\mathsf{T}[i\lceil\frac{n}{p}\rceil, \min((i+1)\lceil\frac{n}{p}\rceil), n)$ is stored at PE i. Now, when we require the i-th character, we know that it is stored on PE i/ℓ and is the $i\%\ell$-th character in the local slice, where % denotes the *modulo* operator. Then, before we induce the next suffixes (pushfront or pushback operation in Algorithm 1), we retrieve the first character of *all* suffixes that are induced during this step in *one* communication phase, which allows us to induce into the correct distributed array.

In practice, we can also make use of the property that not all classes contain suffixes. The inner loops (see lines 2 and 11) are implemented such that they skip distributed arrays that cannot contain any (relevant) suffixes, which are characterized by the following lemma:

LEMMA 4.4. *Let be $\alpha, \beta \in \Sigma$, then*

1. $\alpha < \beta \Rightarrow C^{-}_{\alpha\beta} = \emptyset$,
2. $\alpha > \beta \Rightarrow C^{+}_{\alpha\beta} = \emptyset$, *and*
3. $\alpha = \beta \Rightarrow C^{-\triangleright}_{\alpha\beta} \cup C^{+\triangleright}_{\alpha\beta} = \emptyset$.

Proof. Due to the definition of $C^{-}_{\alpha\beta}$ and $C^{+}_{\alpha\beta}$ ($i \in C^{-} \Rightarrow \mathsf{T}[i] \geq \mathsf{T}[i+1]$ and $i \in C^{+} \Rightarrow \mathsf{T}[i] \leq \mathsf{T}[i+1]$), the first two statements are true. To prove the third statement we assume that $i \in C^{+\triangleright}$. Therefore, $\mathsf{S}_i < \mathsf{S}_{i+1}$ and $i+1 \in C^{-}$. This leads to $\mathsf{T}[i] = \mathsf{T}[i+1] \geq \cdots > \mathsf{T}[i+j]$ with $\mathsf{T}[i+j]$ being the first character strictly smaller than $\mathsf{T}[i+1]$. This contradicts our initial assumption. The proof of the last case ($\alpha = \beta \Rightarrow C^{-\triangleright}_{\alpha\beta} = \emptyset$) works analogously.

For each pair of characters there is a communication phase (lines 8 and 15). This would be sufficient if we did not insert into distributed arrays that we are currently traversing. Unfortunately, there is one case where this can happen. We describe how to handle this special cases in the next paragraph.

The Special Case. A *run* of length r denotes a substring $\mathsf{T}[i, i+r)$ with $\mathsf{T}[i] = \mathsf{T}[i+1] = \cdots = \mathsf{T}[i+r-1]$ for $i \in [0, n-r)$. The algorithm (as just described) cannot handle length-3 or longer runs, as this would require to induce in the same distributed array that we are currently traversing (lines 3 and 12). Since the arrays are updated just before the next character combination is considered, we never use the newly induced suffixes to further induce any suffix. Fortunately, handling those runs is easy in the distributed setting. If multiple runs of the same character occur, suffixes that are induced by the rightmost suffix in the run are interleaved. E.g., if we induce suffixes i and j in runs from SA-positions k and ℓ with $k < \ell$ during the first inducing step, we know that $i-1$ occurs left of $j-1$ in SA. This repeats until one or both runs end and can be generalized for an arbitrary number of runs, see Fig. 3 for an example. Hence, we can compute the part of the SA where the runs occur using only their length and the SA-position they have been induced from.

To this end, we first determine all runs that must be contained in the currently considered distributed array and compute their lengths. Next, we communicate this information among all PEs. Now, each PE can determine which entries must be stored in its locals slice of the distributed array, by simply unrolling the runs similar to the example given in Figure 3.

The Costs of Inducing. In total, we consider each entry of the SA exactly once. Since we use distributed arrays, we know that the number of SA-positions on all PEs is the same up to rounding. Also, the maximum number of computer words sent and received is (asymptotically) the same. While there can be communication phases where only one PE receives data, all PEs must receive the same amount of data at some point (as the content of the distributed array is stored equally among all PEs). The additional costs of the steps required by the special case are dominated by the costs described above. Since we need a communication phase for each pair of characters, we require $\mathcal{O}(\sigma^2)$ supersteps. All this requires wn bytes to store the induced suffixes and at most twice as much to send the positions, when we store them in a distributed array. This leads to the following lemma:

LEMMA 4.5. *We can induce all suffixes in $C^{-\triangleright}, C^{-\odot}$ and $C^{+\odot}$ in $\mathcal{O}\left(\frac{n}{p} + \frac{n}{p}G + \sigma^2 L\right)$ time.*

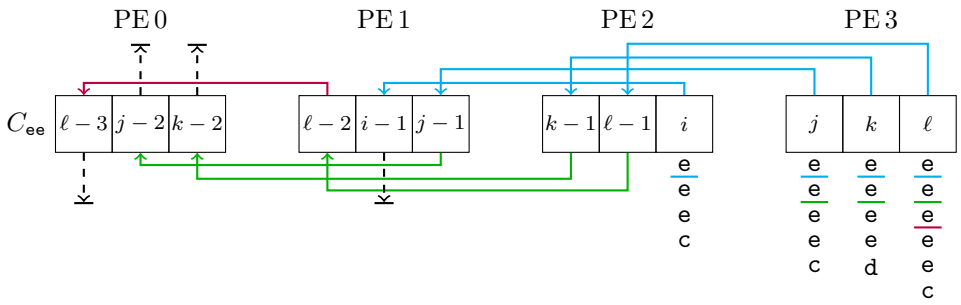

Figure 3: The distributed array C_{ee} of size 12 on four PEs. Initially, the text positions i, j, k and ℓ are contained in C_{ee} (the text starting at those positions up to the first mismatching character is given below these entries). In this example, we induce from right to left. The arrows indicate the interleaved occurrences of the induced suffixes that can be induced by just the length of the run. The colors of the arrows indicate the starting position of the suffix (see the horizontal bars in the text). Arrows ending in a bar indicate that the suffix is induced into a different distributed array.

Last, we need to transform the distributed arrays, such that all suffixes on PE i are lexicographically smaller than all suffixes on PE j if $0 \leq i < j < p$, i.e., compute the final SA. To this end, we compute the number of elements (in each distributed array) that we must sent to each PE, then during one large communication phase, we send them accordingly. The memory required during this phase is $2w\frac{n}{p}$ bytes per PE.

4.6 Space and Time Requirements. The most memory is required during the sorting of the suffixes in $C^{+\triangleright}$, where we need $3wn$ bytes of memory in addition to the text (n bytes) and $2\sigma^2 wp$ bytes for the size of the (sub-)classes. During the classification, we need $wn + 2\sigma^2 wp$ bytes and the text. Last, when inducing all other suffixes, we need wn bytes in addition to the text. This results in a maximum $3wn/p + 2\sigma^2 w$ bytes per PE, when we distribute all data equally among all PEs. Using Lemmas 4.2, 4.3, and 4.5 we get the following:

COROLLARY 4.1. *Using p PEs, we can compute the* SA *in* $\mathcal{O}\left(\frac{n}{p}(\lg n + \lg p) + \left(\frac{n}{p} + p\right)G + \left(\lg n + \sigma^2\right)L\right)$ *time, using* $3wn + 2\sigma^2 wp$ *bytes of space.*

The factor of σ^2 in the space and in the costs for the synchronization steps implies that this algorithm is only applicable to at most medium-sized alphabets.

5 Distributed String Sorting

In this section, we describe a distributed variant of *string sample sort* [6], which we used in §4.4 to sort the $C^{+\triangleright}$-ending substrings. Usually, *atomic* keys (keys that can be compared with a single comparison) are considered when sorting data. Longer strings, on the other hand, cannot be compared with a single comparison, but have to be compared character by character.

Let $S \coloneqq \{s_0, \ldots, s_{m-1}\}$ be a set of m strings and let D be the *distinguishing prefix size* of S, i.e., the number of characters that must be compared to sort S lexicographically. Formally, for a set $S \coloneqq \{s_0, \ldots, s_{m-1}\}$ of strings, $D \coloneqq 1 + \sum_{i=0}^{m-1} \max\{lcp(s_i, t) \colon t \in S \setminus s_i\}$, where $lcp(s,t) = \max\{i \colon s[0..i-1] = t[0..i-1]\}$. Further, let $M \coloneqq \sum_{i=0}^{m-1} |s_i|$ be the total length of all strings in S. The strings are distributed among all PEs such that each PE holds roughly the same number of characters (if possible). For simplicity, we assume that on each PE the local distinguishing prefix size, i.e., the number of characters that must be compared to sort all strings that are stored at the PE, is $D' = \Theta(\frac{D}{p})$ and that on each PE there are strings of total length of $m' = \Theta(\frac{M}{p})$ characters. Those assumptions are feasible in our scenario as we focus on the sorting of $C^{+\triangleright}$-ending substrings that all have a similar (short) length, see Table 1 for practical measurements confirming this simplifying assumption. Splitters are chosen in the same fashion they are chosen in our distributed sample sort for atomic keys, which is described in §4.4:

1. We first sort all strings locally on the PEs and determine the local splitters. These splitters are then shared among all PEs, and a common set of $p-1$ global splitters is chosen.

2. Using the global splitters, on each PE we determine p intervals (on the locally sorted strings) that have the global splitters as borders (the first and last interval has only one global splitter as upper and lower border, respectively).

3. We distribute the strings in these intervals among all PEs, such that the strings in the i-th interval on

any PE are sent to PE i.

4. Since all strings that have been sent to any PE are sorted, we simply merge the received intervals locally to obtain the final sorting.

The most time consuming task is the sorting in Step 1. Steps 2–4 work similar to the distributed sample sort described in §4.4. The only difference is that we need to consider strings instead of atomic keys.

Note that the sorting as described here differs from the canonical sample sort, as we first sort the locally on each PE before we determine the splitters. Since we sort the strings locally, we only have to merge them later. Hence, this approach could also be denoted as a sample and merge sort hybrid.

6 Experiments

We conduct multiple experiments to determine the running time and memory usage of three distributed SACAs and multiple (shared and distributed memory) string sorting algorithms. The experiments consist of two parts.

First (§6.1), we present the main result of this paper, and compare the following three distributed SACAs:

dPD our distributed prefix doubling SACA (see §3),

dDivSufSort our lightweight distributed SACA that is described in §4 and based on induced copying,

PSAC the previously fastest distributed prefix doubling SACA [17], and

DC3/DC7/DC13 Bingmann's implementations [5] of the distributed DCX algorithm [27], which use 32-bit integers and therefore are restricted to texts up to size 4 GB.

In addition, we include a shared memory version of DivSufSort [28] to the experiments. For the shared memory algorithms a PE is a thread. We execute the shared memory version of DivSufSort on 20 PEs.

Both our SACAs (dPD and dDivSufSort) are available from `https://github.com/kurpicz/dsss`. We are also aware of the following distributed SACAs that we did not include in our evaluation: *cloudSACA* [1,34] (cannot compute the SA for inputs of the size that we are considering due to their high memory requirements, see [17, §6.3] for details), and multiple distributed SACAs implemented using *Thrill* [8] (which is a framework for distributed big data batch computations that currently does not support high performance networks natively, which we use in our experiments).

Second (§6.2), we compare different sequential string sorting algorithms that we use in distributed string sorting (DSS) algorithms. As a sanity check, we compare our DSS algorithm with *shared memory* parallel string sorting (PSS) algorithms presented by Bingmann et al. [6], since we are not aware of any other distributed string sorting algorithm.

Name	n	σ	Mean	SD
CC	140 GB	242	6.00	0.92
DNA	164 GB	4	6.65	1.58
Prot	36.8 GB	26	6.12	0.66
Wiki	130 GB	213	6.39	1.49

Table 1: Information about the texts used in our experiments. *Mean* denotes the average length of all $C^{+\triangleright}$-ending substrings in the text, and *SD* denotes the standard deviation of those lengths.

For our experiments, we used real-world texts (see Table 1 for details). Whenever we use smaller texts throughout our experiments, we work on a prefix of the corresponding text.

CommonCrawl (CC) Text of websites contained in CommonCrawl's web archive, where we removed all meta data and markup commands (`http://commoncrawl.org/`).

DNA FASTQ files from the 1000 Genomes project, where we ignored all lines but the one containing the raw sequence letters (`http://www.internationalgenome.org/`).

Prot FASTA files (Protein-data) from the Universal Protein Resource (UniProt), where we only kept the sequence representation (`https://www.uniprot.org/downloads`).

Wiki Current version of each article on Wikipedia in English, Finnish, French, German, Italian, Polish, and Spanish in XML-format (`https://dumps.wikimedia.org`).

We conducted our experiments on a cluster. Here, we have access to 316 nodes each equipped with two Intel Xeon E5-2640v4 (10 cores each, 32 KB L1, 256K KB L2, 25.6 MB L3 cache) and 64 GB RAM. The nodes are connected via Interconnect Infiniband QDR. In our experiments, a PE corresponds to one MPI-thread running on one dedicated CPU-core. We compiled all algorithms with g++ version 7.3.0 using `-O3 -DNDEBUG -march=native` for optimization. Each reported time is the median time of five executions. The timing starts as soon as the local slices of the text are available in RAM.

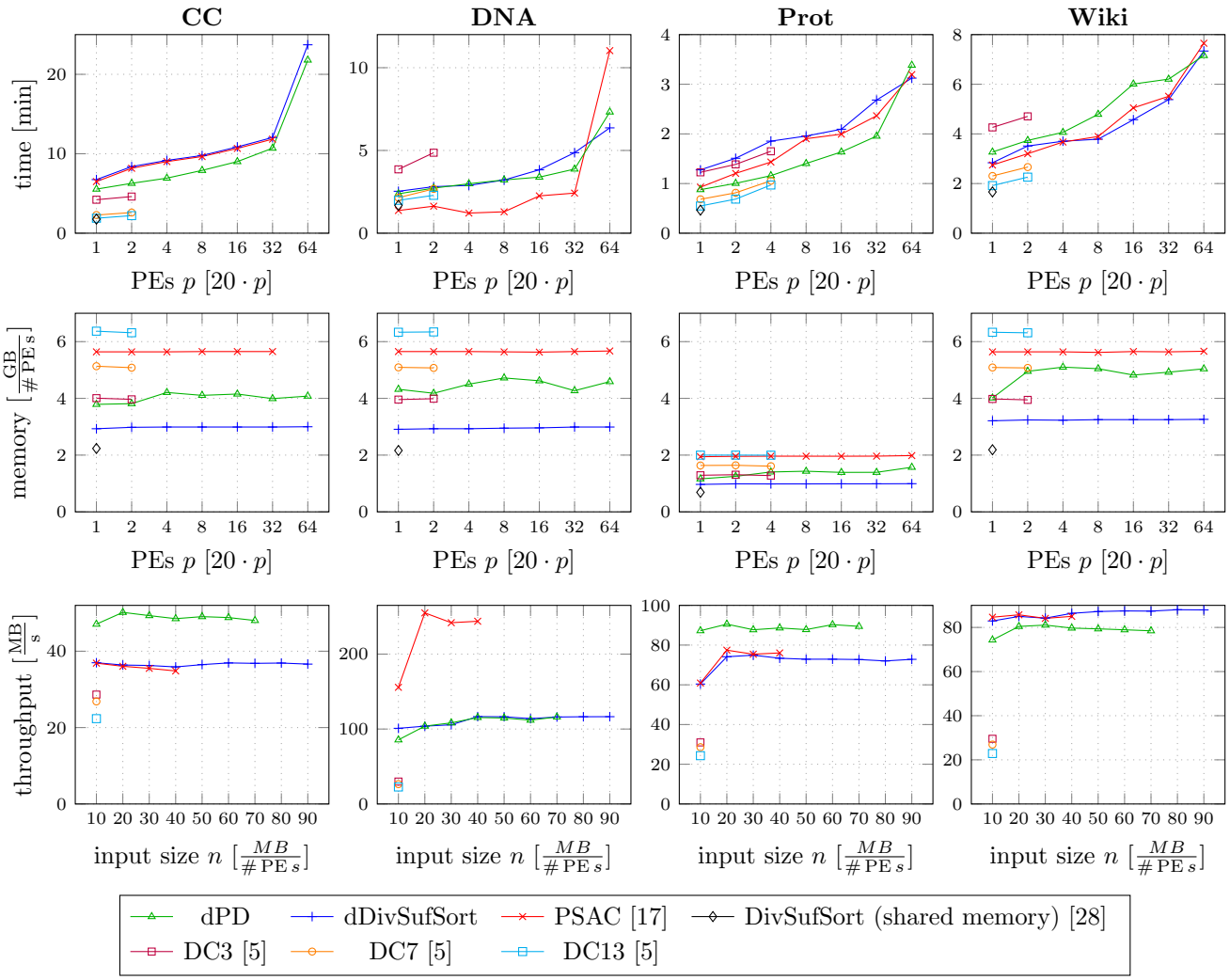

Figure 4: Running times of the SACAs in our weak scaling experiments using 90 MB (CC, DNA and Wiki) and 28 MB (Prot) input text per PE (first row), the corresponding memory peak per PE (second row), and the throughput during our breakdown test on 320 PEs.

6.1 Evaluating Distributed Suffix Array Construction. We visualized the results of our experiments in Figure 4. The first two rows correspond to our *weak scaling* experiment, where we run all SACAs with 90 MB (CC, DNA and Wiki) and 28 MB (Prot) of text as input per PE. Due to memory constraints dPD, PSAC, and the DCX algorithms cannot use all CPU-cores on each node, as the total amount of RAM does not suffice on a single node.

On all texts but DNA, one of our algorithms is the fastest, when the input text is larger than 4 GB. Our prefix doubling algorithm is faster than dDivSufSort on all texts but Wiki. PSAC is especially fast on DNA, here it is up to 2.5 times faster than our algorithms. On the other hand, PSAC crashes during the computation of the SA of the 115 GB prefix of CC. The DCX algorithms can run on up to 40 PEs (80 PEs for Prot), as in our experiment the input is larger than 4 GB on more PEs. DC7 and DC13 are the fastest distributed SACAs on all texts but DNA. The great increase in running time on more than 640 PEs is due to the cluster. Our nodes share switches with nodes used by other applications. These switches only have a 1 : 3 blocking ratio. Hence, we do not have a guaranteed bandwidth. On average, dDivSufSort is 25 % slower and dPD is 15 % slower than PSAC considering all texts. The median running time of dDivSufsort and dPD is 3.33 % slower and 5.3 % faster than PSAC, resp. Ignoring DNA, dDivSufSort is only 5.97 % slower and dPD is even 5.33 % faster than PSAC (on average). The shared memory version of DivSufSort

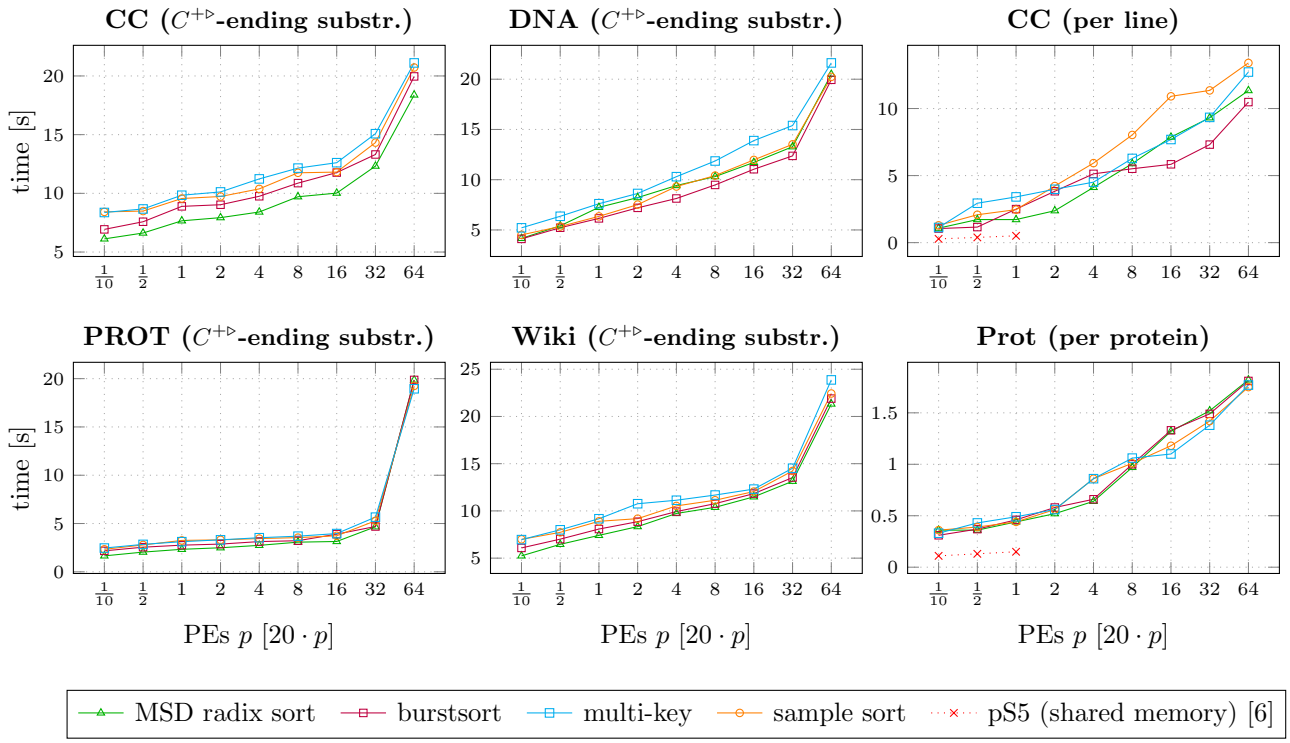

Figure 5: Running times of the DSS and PSS algorithms. We consider the $C^{+\triangleright}$-ending substrings of the input text (first two columns) and general substrings (last column). The total size of the input texts is 90 MB per PE for CC, DNA, Wiki and 28 MB per PE for Prot.

is slightly faster on all inputs but DNA, where PSAC is incredibly fast. This is the expected behavior when we compare a shared memory SACA with distributed memory SACAs.

The *memory peak* of all algorithms is as expected; it coincides with the theoretical results. dDivSufSort requires half as much memory as PSAC on all instances. This is because the dominating part (regarding the memory requirements) comes from the doubling steps (see §4.4 where we use prefix doubling on a text consisting of the ranks of the $C^{+\triangleright}$-ending substrings). The changing memory peak of dPD is due to the unknown amount of rank-tuples are stored at the PE where they are discarded. In practice, pPD requires less memory than PSAC, while in theory, both have the same memory requirements. To achieve this behavior, we use 40-bit integers for dPD and dDivSufSort. Unfortunately, PSAC does not support (and cannot be easily changed to support) those and relies on 64-bit integers. DC7 and DC13, the fastest distributed SACAs, require also the most memory, while using 32-bit integers. The shared memory version of DivSufSort requires less memory than any of the distributed memory algorithms. Again, this is the expected behavior, as shared memory algorithms do not need a buffer for communication.

Finally, we conducted a *breakdown test* (last row in Figure 4), where we run the SACAs on 320 PEs that are located on 16 nodes. We increase the input size until the SACA cannot compute the SA due to memory limitations. Here, we start with 10 MB per PE (3.2 GB in total), which is also the maximum input size for the DCX algorithms. All other algorithms can handle larger input sizes. PSAC works for up to 40 MB per PE, dPD works for up to 70 MB per PE, and dDivSufSort works for up to 90 MB per PE. We can see a small communication overhead when looking at the throughput for 10 MB inputs. Otherwise, the throughput of all algorithms matches the running times in our weak scaling experiment.

6.2 Evaluating Distributed String Sorting. As pDivSufSort needs to sort strings (§4.4), we are interested in the DSS that is best suited for sorting the $C^{+\triangleright}$-ending substrings of the input strings. Here, we briefly evaluate our new DSS algorithm from §5. There is a degree of freedom in Step 1 (local string sorting), which can use any sequential sorter. We tested all of the over 130 implementations from Bingmann et al. [6] and by

Kärkkäinen and Rantala [21], but only show the running times for the choices where our DSS is fastest on average (MSD radix sort, burstsort, multi-key quicksort, and string sample sort). Figure 5 labels the algorithms by these sequential string sorters.

The *MSD string sorting* (MSD) algorithms are among the fastest in all tests.This is due to the small size of the strings to be sorted. Second in running time (compared to different MSD radix sort variants) are *burstsort* and *sample sort* variants that are around 15 % and 18 % slower, resp. When considering larger alphabets (CC), burstsort is slower than sample sort and both are slower than MSD radix sort (10 % and 12 %). The caching advantage of burst sort (compared to MSD radix sort) is negligible as we use a more advanced version of MSD radix sort, which employs an oracle to reduce TLB and other cache misses. The behavior is as expected, the running time mainly depends on the sorter that we use to sort the data locally. All other steps (2–4) are the same, independently from the algorithm used to sort the strings locally.

We also tested the string sorting algorithms on more general strings. To get more realistic strings, we interpret each new line in CC and each protein in Prot as one string, see Figure 5 on the right. The strings are longer than the $C^{+\triangleright}$-ending substrings. On average, the substrings have length 39.15 and 54.14 with a standard deviation of 636.20 and 13.03 for CC and Prot. As a sanity check (and in the absence of a true competitor), we compared our distributed string sorter to the fastest PSS algorithm (pS5 [6]), which is around three times faster than our DSS algorithms. This comes at no surprise as we do not make use of shared memory parallelism but distributed memory parallelism, which allows us to use more than 20 PEs instead.

7 Conclusions

We presented the first distributed SACA based on induced copying and the first distributed string sorting implementations. Our SACA is competitive when it comes to running time but has an up to 50% smaller memory footprint than its competitor [17], allowing us to compute the SA for larger texts on the same hardware.

In the future, we want to also compute the size of the longest common prefixes (LCP) of all suffixes consecutive in the SA. Access to the SA and LCP-array allows us to compute further distributed full-text indices, e.g., [3,16]. Since we want to compute suffix arrays for even larger texts, a hybrid approach that combines external SACAs (e.g., [7, 20]) and our distributed SACA is a promising direction of research, as the size of the SA (and LCP-array) exceeds even the size of the distributed memory available in medium sized clusters.

References

[1] Ahmed Abdelhadi, AH Kandil, and Mohamed Abouelhoda. Cloud-based parallel suffix array construction based on mpi. In *Middle East Conference on Biomedical Engineering (MECBME)*, pages 334–337. IEEE, 2014.

[2] Donald A. Adjeroh and Fei Nan. Suffix sorting via Shannon-Fano-Elias codes. In *Data Compression Conference (DCC)*, page 502. IEEE, 2008.

[3] Diego Arroyuelo, Carolina Bonacic, Veronica Gil-Costa, Mauricio Marin, and Gonzalo Navarro. Distributed text search using suffix arrays. *Parallel Computing*, 40(9):471–495, 2014.

[4] Uwe Baier. Linear-time suffix sorting - A new approach for suffix array construction. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPIcs*, pages 23:1–23:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[5] Timo Bingmann. pdcx, `https://github.com/bingmann/pDCX`, 2018.

[6] Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *Algorithmica*, 77(1):235–286, 2017.

[7] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and LCP arrays in external memory. *ACM Journal of Experimental Algorithmics*, 21(1):2.3:1–2.3:27, 2016.

[8] Timo Bingmann, Simon Gog, and Florian Kurpicz. Scalable construction of text indexes. *CoRR (accepted for publication in IEEE BigData 2018)*, abs/1610.03007, 2016.

[9] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. *Commun. ACM*, 39(12es):273–297, 1996.

[10] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Annual Symposium Combinatorial Pattern Matching (CPM)*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003.

[11] Andreas Crauser and Paolo Ferragina. On constructing suffix arrays in external memory. In *European Symposium on Algorithms (ESA)*, volume 1643 of *LNCS*, pages 224–235. Springer, 1999.

[12] Felipe Alves da Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theor. Comput. Sci.*, 678:22–39, 2017.

[13] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12:3.4:1–3.4:24, 2008.

[14] Martin Farach. Optimal suffix tree construction with large alphabets. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE, 1997.

[15] Johannes Fischer and Florian Kurpicz. Dismantling divsufsort. In *Prague Stringology Conference (PSC)*, pages 62–76. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017.

[16] Johannes Fischer, Florian Kurpicz, and Peter Sanders. Engineering a distributed full-text index. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 120–134. SIAM, 2017.

[17] Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 16:1–16:10. ACM, 2015.

[18] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, chapter 3, pages 66–82. Prentice-Hall, 1992.

[19] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *International Symposium on String Processing (SPIRE)*, pages 81–88. IEEE, 1999.

[20] Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova. Engineering external memory induced suffix sorting. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 98–108. SIAM, 2017.

[21] Juha Kärkkäinen and Tommi Rantala. Engineering radix sort for strings. In *International Symposium on String Processing (SPIRE)*, volume 5280 of *LNCS*, pages 3–14. Springer, 2008.

[22] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[23] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

[24] Dong Kyue Kim, Junha Jo, and Heejin Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In *Workshop on Experimental and Efficient Algorithms (WEA)*, volume 3059 of *LNCS*, pages 301–314. Springer, 2004.

[25] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2-4):126–142, 2005.

[26] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

[27] Fabian Kulla and Peter Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.

[28] Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *J. Discrete Algorithms*, 43:2–17, 2017.

[29] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theor. Comput. Sci.*, 387(3):258–272, 2007.

[30] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[31] Michael A. Maniscalco and Simon J. Puglisi. An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithmics*, 12:1.2:1–1.2:23, 2007.

[32] Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 3111 of *LNCS*, pages 372–383. Springer, 2004.

[33] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[34] Ahmed A Metwally, Ahmed H Kandil, and Mohamed Abouelhoda. Distributed suffix array construction algorithms: Comparison of two algorithms. In *Cairo International Biomedical Engineering Conference (CIBEC)*, pages 27–30. IEEE, 2016.

[35] Yuta Mori. divsufsort, `https://github.com/y-256/libdivsufsort`, 2006.

[36] Yuta Mori. SAIS, `https://sites.google.com/site/yuta256/sais`, 2008.

[37] Joong Chae Na. Linear-time construction of compressed suffix arrays using o(n log n)-bit working space for large alphabets. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3537 of *LNCS*, pages 57–67. Springer, 2005.

[38] Ge Nong. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15, 2013.

[39] Ge Nong and Sen Zhang. Optimal lightweight construction of suffix arrays for constant alphabets. In *International Workshop on Algorithms and Data Structures (WADS)*, volume 4619 of *LNCS*, pages 613–624. Springer, 2007.

[40] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference (DCC)*, pages 193–202. IEEE, 2009.

[41] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

[42] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

[43] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Softw., Pract. Exper.*, 37(3):309–329, 2007.

[44] Julian Seward. On the performance of BWT sorting algorithms. In *Data Compression Conference, (DCC)*, pages 173–182. IEEE, 2000.

[45] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.