Fast and Lightweight Distributed Suffix Array Construction

Manuel Haag
Karlsruhe Institute of Technology, Germany

Florian Kurpicz
Karlsruhe Institute of Technology, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Matthias Schimek
Karlsruhe Institute of Technology, Germany

Abstract

The suffix array contains the lexicographical order of all suffixes of a text. It is one of the most well-studied text indices with applications in bioinformatics, compression, and pattern matching. The main bottleneck of distributed-memory suffix array construction algorithms is their memory requirements. Even careful implementations require 30×–60× the input size as working memory. We present a scalable and lightweight distributed-memory adaptation of the difference cover (DCX) suffix array construction algorithm. Our approach relies on novel bucketing and random chunk redistribution techniques which reduce our memory requirement to 20×–26× the input size for medium-sized inputs and to 14×–15× for large-sized inputs. Regarding running time, we achieve speedups of up to 5× over current state-of-the-art distributed suffix array construction algorithms.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Computing, Suffix Array Construction

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.47

Related Version *Previous Version*: `https://arxiv.org/abs/2412.10160`

Supplementary Material
Software: `https://github.com/mschimek/distributed_suffix_sorting/`
archived at `swh:1:dir:d268977edf8b2c9a7427b598a7f3e87493fb2566`
Software: `https://github.com/mschimek/distributed_suffix_sorting_evaluation`
archived at `swh:1:dir:991feb9b718b7cc1b22f14047efbd281a9fc0e3a`

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).
The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`).
The authors gratefully acknowledge the computing time provided on the high-performance computer HoreKa by the National High-Performance Computing Center at KIT (NHR@KIT). This center is jointly supported by the Federal Ministry of Education and Research and the Ministry of Science, Research and the Arts of Baden-Württemberg, as part of the National High-Performance Computing (NHR) joint funding program (`https://www.nhr-verein.de/en/our-partners`). HoreKa is partly funded by the German Research Foundation (DFG).

1 Introduction

The suffix array [30, 22] is one of the most studied text indices. Given a text T of length n, its suffix array SA contains a permutation of the integers $1, \dots, n$ such that the corresponding suffixes appear in lexicographical order. More concretely, SA[i] is the starting position of the i-th smallest suffix of T, or equivalently, we have $T[\mathrm{SA}[i]..n] \leq T[\mathrm{SA}[j]..n]$ for any $1 \leq i \leq j \leq n$. To compute the suffix array, we have to (implicitly) sort all suffixes of the text. Therefore, the task of constructing the suffix array is sometimes referred to as suffix sorting. Suffix arrays can be constructed in linear time requiring only constant working space in addition to the space for the suffix array [23, 29].

Suffix arrays have numerous applications in pattern matching and text compression [34]. They are a very powerful full-text index and are used as a space-efficient replacement [1] of the suffix tree, which is considered to be one of the most powerful full-text indices. Furthermore, suffix arrays can be used to compute the Burrows-Wheeler transform [12], which is the backbone of many compressed full-text indices [17, 21].

One problem when considering texts as input is that the amount of textual data to be processed is ever-increasing with no sign of slowing down. For example, the English Wikipedia contains around 60 million pages and grows by around 2.5 million pages each year.[1] A snapshot of all public source code repositories on GitHub requires more than 21 TB to store.[2] Furthermore, the capability to sequence genomic data is increasing exponentially, due to technical advances [37]. All these examples show the importance of scalable algorithms for the analysis of textual information many of which use the suffix array as a building block.

In this paper, we consider distributed-memory suffix sorting. Here, we can utilize many processing elements (PEs) that are connected via a network, e.g., high-performance clusters or cloud computing. In this setting, the main obstacle when computing suffix arrays is the immense amount of working memory required by the current state-of-the-art algorithms. Even carefully engineered implementations require around 30×–60× the input size as working space [19, 20]. Additionally, there is a significant space-time trade-off. The memory-efficient algorithms tend to be slower. We thus ask the question:

Is there a scaling, fast, and *memory-efficient suffix array construction algorithm in distributed memory?*

Summary of our Contributions. We answer this question positively by providing a scalable, fast, *and* space-efficient distributed-memory suffix array construction algorithm (SACA), using a bucketing approach in conjunction with a randomized (chunk-based) redistribution scheme for load balancing with provable performance guarantees. Our algorithm is the fastest (on all inputs but one) distributed-memory suffix array construction algorithm on up to 12 288 cores and requires only 20×–26× the input size as working space. As a side result of independent interest, we improve the load balancing for a distributed prefix-doubling algorithm, enhancing its performance significantly. This might be of interest for future work, as prefix doubling algorithms can more easily be extended to also compute the longest common prefix (LCP) array, another important data structure for text processing.

Paper Outline. First, in Section 2, we introduce some basic concepts required for suffix array construction and distributed-memory algorithms. Next, in Section 3, we discuss previous work on suffix array construction. In Section 4, we present the main result of this paper.

[1] `https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia`, last accessed 2024-12-11.
[2] `https://archiveprogram.github.com/arctic-vault/`, last accessed 2024-12-11.

We start with a description of the distributed-memory variant of the DCX [26] suffix array construction algorithm in Section 4.2. In Section 4.2.1, we demonstrate how a previously developed technique for space-efficient string sorting [31, 28] can be applied to the DCX suffix sorting to obtain a more lightweight algorithm. Subsequently, in Section 4.2.2, we introduce a *randomized chunking scheme* to provide provable load-balancing guarantees for our space-efficient (suffix) sorting approach, followed by a brief analysis of the overall algorithm in Section 4.2.3. Finally, an experimental evaluation of our highly-engineered implementation using MPI is presented in Section 5, followed by a conclusion in Section 6.

2 Preliminaries

We assume a distributed-memory machine model consisting of p processing elements (PEs) allowing single-ported point-to-point communication. The cost of exchanging a message of h *machine words* between any two PEs is $\alpha + \beta h$, where α accounts for the message start-up overhead and β quantifies the time to exchange one machine word. Let h be the maximum number of words a PE sends or receives, then collective operations like *broadcast, prefix sum, (all-)reduce, and (all-)gather* can be implemented in time $\mathcal{O}(\alpha \log p + \beta h)$ [35]. An alltoall exchange using direct messaging is possible in $\mathcal{O}(\alpha p + \beta h)$ [35].

The input to our algorithms is a text T consisting of $n-1$ characters over an alphabet Σ. By $T[i]$, we denote the i-th character of T for $0 \leq i < n-1$. We assume $T[n-1]$ to be a sentinel character $\$ \notin \Sigma$ with $\$ < z$ for all $z \in \Sigma$. The i-th suffix of T, $s_i = T[i,n)$, is the substring starting at the i-th character of T. Due to the sentinel element all suffixes are prefix-free. The *suffix array* SA contains the lexicographical ordering of all suffixes of T. More precisely, SA is an array of length n with $\mathrm{SA}[i]$ containing the index of the i-th smallest suffix of T. A length-l- (or simply l)-prefix of a suffix with starting position i is the substring $T[i,l)$.

In our distributed setting, we assume that each PE i holds a subarray T_i of T as input such that T is the concatenation of all local input arrays T_i, i.e., $T = T_0 \circ \ldots \circ T_{p-1}$. Furthermore, we assume the input to be well-balanced, i.e., $|T_i| = \Theta(n/p)$. For our DCX algorithm, we assume a suitable padding of up to X sentinel characters at the end of the text.

3 Related Work

There has been extensive research on the construction of suffix arrays in the sequential, external-memory, shared-memory parallel, and (to a somewhat lesser extent) in the distributed-memory setting. All suffix array construction algorithms are based on three general algorithmic techniques: *prefix doubling*, *induced copying*, and *recursion* or combinations thereof. In the following, we give a brief overview of these techniques. For a more comprehensive overview, we refer to the most recent surveys [7, 9].

Prefix-Doubling. In algorithms based on prefix doubling, the suffixes are iteratively sorted by their length-h prefix starting with $h = 1$. Now, all suffixes that share a common h-prefix are said to be in the same h-group and have an h-rank corresponding to the number of suffixes in lexicographically smaller h-groups. By sorting all suffixes based on their h-group, we can compute the corresponding suffix array SA_h. Note that this suffix array does not necessarily have to be unique, as the order of suffixes within an h-group is not unique. If for some h, all h-groups contain only a single suffix, we have $\mathrm{SA}_h = \mathrm{SA}$. Therefore, the idea is to increase h until all h-ranks are unique. To this end, during each iteration, the length of the considered prefixes is doubled. Fortunately, we do not have to compare the prefixes explicitly. Instead, during iteration $i > 0$, for a suffix s_j starting at index j, the rank of its length-h prefix can be inferred from the pair of ranks $(\mathrm{rank}_{h/2}[j], \mathrm{rank}_{h/2}[j + h/2])$ computed in the previous iterations for the suffixes s_j and $s_{j+h/2}$. This yields the correct result as comparing two suffixes s_i and s_j by their length-h prefix can be achieved by (a) comparing their first $h/2$ characters and – if they tie – (b) comparing the following $h/2$ characters, which in turn is equivalent to lexicographically comparing the ranks of the corresponding $h/2$-prefixes. Using the overlap of suffixes in a text, prefix-doubling boils down to at most $\mathcal{O}(\log n)$ rounds in which n pairs of integers have to be sorted. Thus, this approach has an overall complexity in $\mathcal{O}(n \log n)$ in the sequential setting, when using integer sorting algorithms. The first suffix array construction algorithm [30] is based on prefix-doubling. In the sequential setting, this approach has not received much attention, due to its non-linear running time. However, in distributed memory, the fastest currently known suffix array construction algorithm is based on prefix doubling [20].

Induced-Copying. Induced-copying algorithms sort a (small) subset of suffixes and then *induce* the order of all other suffixes using the subset of sorted suffixes. First, all suffixes are classified using one of two [25, 33] classification schemes. Next, all suffixes necessary for inducing the order of the remaining suffixes are sorted directly. Then, the properties of the classification allow us to induce the order of the remaining suffixes based on their class, starting characters, and preceding or succeeding directly sorted suffix. The inducing part of these algorithms usually consists of just two scans of the text, where for each position only one or two characters have to be compared. Combined with a recursive approach, induced copying algorithms can compute the suffix array in linear time requiring only constant working space in addition to the space for the suffix array [23, 29]. This combination is also very successful, as it is used by the fastest sequential suffix array construction algorithms [4, 18, 24, 32]. Interestingly, there is only one linear-time suffix array construction algorithm based on induced copying that does not also rely on recursion [5]. In distributed memory, induced copying algorithms are space-efficient [19].

Recursive Algorithms. The third and final technique is to use recursion to solve subproblems of ever decreasing sizes. Here, the general idea is to partition the input into multiple different (potentially overlapping) substrings. A subset of these substrings can then be sorted using an integer sorting algorithm (in linear time). If all substrings are unique, we can compute a suffix array together with the remaining suffixes not yet sorted. Otherwise, we recurse on the non-unique ranks of the substrings as new input. We then use the suffix array from the recursive problem to compute the unique ranks from the original subset of substrings. The first linear-time suffix array construction algorithm is purely based on recursion [26]. This algorithm is also the foundation of the distributed-memory suffix array construction algorithm presented in this paper. It already has been considered in distributed memory [6, 10]. However, all implementations are direct adaptations of the sequential algorithm to distributed memory.

4 A Space-Efficient Variant of Distributed DCX

In this section, we describe the general idea of the sequential DC3 algorithm [26]. Then, we give a canonical transformation of the sequential DC3 algorithm to a distributed-memory algorithm. Here, we also consider the more general form – the DCX algorithm. Finally, we discuss how to optimize this canonical transformation into a scaling, fast, and memory-efficient distributed suffix array construction algorithm.

4.1 The Sequential DCX Algorithm

The *skew* or **D**ifference **C**over 3 algorithm (DC3) – and its generalization DCX – is a recursive suffix array construction algorithm that exhibits linear running time (in the sequential setting). As our main contribution is a fast and lightweight distributed variant of the algorithm, we now briefly discuss its key ideas, beginning with the eponymous concept of a *difference cover*.

Let X be a positive integer. A subset $D_X \subseteq \{0, \ldots, X-1\}$ is a difference cover modulo X if for every $r \in \{0, \ldots, X-1\}$ there are $i, j \in D_X$ such that $i - j \bmod X = r$. Put differently, D_X *covers* the set $\{0, \ldots, X-1\}$ in the sense that we have $\{0, \ldots, X-1\} = \{i - j \bmod X \mid i, j \in D_X\}$.

A key property of a difference cover modulo X, on which the DCX algorithm relies, is that for all $i, j \in \mathbb{N}$, there is a $0 \leq l < X$ such that $(i + l) \bmod X \in D_X$ and $(j + l) \bmod X \in D_X$. $X = 3$ is the smallest X for which a non-trivial difference cover exists, with $D_3 = \{1, 2\}$.

DCX Algorithm. We now discuss the main steps of the DCX algorithm. It starts by classifying the suffixes s_j of a given text T into different sets, based on their starting position j. Suffixes with index $(j \bmod X) \in D_X$ constitute the *(difference cover) sample* S_{D_X}.

For now, assume that we know a relative ordering of the suffixes in S_{D_X} within the text. Since D_X is a difference cover, for *any* two suffixes s_i and s_j, there is an $l < X$ such that s_{i+l} and s_{j+l} are in S_{D_X} for which we already know a relative ordering. Hence, for comparing s_i and s_j, it is sufficient to compare the l-prefixes of s_i and s_j, using the ranks of s_{i+l} and s_{j+l} to break ties. That is, we compare tuples $(T[i, i+l), \mathrm{rank}[i+l])$ and $(T[j, j+l), \mathrm{rank}[j+l])$. For $X = 3$, this rank-*inducing* can be achieved using linear-time integer sorting.

It remains to discuss how the relative ordering of the sample suffixes can be obtained. We start by sorting the X-prefixes of the sample suffixes. If they are unique, we are already done, as we can take their rank for the ordering. Otherwise, we replace the sample suffixes j with the rank of their X-prefix and reorder them by $(j \bmod X, j \operatorname{div} X)$. This results in an auxiliary text T' that contains the renamed difference cover sample suffixes in original text order within each equivalence class in D_X. We can now recursively apply the algorithm to this text T', yielding a suffix array SA'. From SA' we can retrieve a relative ordering of the sample suffixes with regard to the original text T.

For DC3, the number of sample suffixes is $\leq 2/3n$. As all other operations can be achieved with work linear in the size of the input, the overall complexity of the algorithm is in $\mathcal{O}(n)$.

4.2 The Distributed DCX Algorithm

Our distributed suffix array construction is a simple and practical distributed variant of the DCX algorithm for $X \geq 3$. Algorithm 1 shows a high-level pseudocode for the algorithm.

We now discuss the algorithm in some more detail. The input to the algorithm on PE i is the local chunk T_i of the input text T.

1. **Sort the Difference Cover Sample.** In the first phase of the algorithm, we select, on each PE i, the difference cover (DC) sample suffixes starting at (global) positions j with $(j \bmod X) \in D_X$. As in the sequential setting, we want to compute unique ranks of these suffixes first. For that, we first globally sort the X-prefixes of all DC sample suffixes. If all of them are unique, this already constitutes the relative ordering of the sample suffixes within the final suffix array. This rank information can then be used to compare any two suffixes s_i and s_j (see Section 4.1) and we continue with step three. Otherwise, we have to recurse on the sample suffixes as described in the following step two of the algorithm.

Algorithm 1 High-level overview of a simple distributed variant of the DCX algorithm.

```
   Input: Text $T_i$ on PE $i$.
   Output: Local part of distributed suffix array of $T$.
 1 $o_i = \texttt{PrefixSum}(|T_i|)$                                   // global text index offset
 2 $C_i = \langle 0 \le j < |T_i| \mid (j + o_i \bmod X) \in D_X \rangle$          // DC sample positions
 3 $S_i = \langle (\underbrace{T_i[j, j+X)}_{X\text{-prefix}}, \underbrace{j + o_i}_{\text{global idx}}) \mid j + o_i \in C_i \rangle$   // (X-prefix,idx) of DC samples
 4 globally sort $S_i$ by first entry
 5 if all first entries of $S$ are unique then
 6     $R_i = \langle (\texttt{rank}(t, S), j) \mid t = (\text{prefix}, j) \in S_i \rangle$   // unique ranks of DC samples
 7 else
 8     $P_i = \langle (\texttt{rank}(t, S), j) \mid t = (\text{prefix}, j) \in S_i \rangle$   // replace X-prefix with rank
 9     globally sort $P_i$ by $(j \bmod X, j \operatorname{div} X)$
10     $SA'_i \leftarrow$ recursively call DCX on $T'_i = \langle r \mid (r, j) \in P_i \rangle$
11     $R_i = \langle (j, \underbrace{\texttt{mapback}(SA[j])}_{\text{restore idx of DC samples in } T}) \mid 0 \le j < |SA'_i| \rangle$
12 globally sort $R_i$ by second component          // bring $R_i$ in text order
   // construct (X-prefix, <ranks>) tuples for all suffixes
13 $S_i = \langle (\underbrace{T_i[j, \dots, j+X)}_{X\text{-prefix}}, \underbrace{R_i[\texttt{next}(j, 1)], \dots, R_i[\texttt{next}(j, v)]}_{\text{ranks of DC samples following } j}, \underbrace{j + o_i}_{\text{global idx}})) \mid 0 \le j < |T_i| \rangle$
14 globally sort $S_i$ by appropriate comparison function (see [26])
15 output last entry of $S_i$ as suffix array $SA_i$
```

2. **Compute Unique Ranks Recursively.** If the ranks are not already unique, we locally create an array P_i by replacing each entry $(X-\text{prefix}, j)$ of S_i with $(\text{rank}[j], j)$. The distributed array $S = \bigcup S_i$ contains the DC sample suffixes sorted by their X-prefix. The rank$[j]$ of the j-th sample suffix is the global rank of its X-prefix within S. Equal prefixes obtain the same rank. Since S is sorted, the ranks can be computed by a prefix sum. Afterwards, we globally sort P_i by $(j \bmod X, j \operatorname{div} X)$. This rearranges the newly renamed sample suffixes in their original order by respecting the equivalence class of their starting index within D_X. We then recursively call the DCX algorithm on the text T'_i where T'_i contains the new names of the sample suffixes from P_i dropping the index. From the suffix array SA' of T', we can determine the rank of each sample suffix j using the `mapback` function which maps the index of a DC sample in T' back to its original position in T. Due to the construction of T', the unique ranks of the DC sample suffixes within T' also yield a relative ordering of the DC sample suffixes in T [26].
3. **Sort All Suffixes.** Now, we construct for each suffix s_j in T a tuple containing: (1) the X-prefixes of the suffix, (2) the previously computed ranks of the following $|D_X|$ DC samples after s_j in text order, and (3) its global index j within T. Globally sorting theses tuples using the previously discussed comparison function for suffixes s_i, s_j yields the suffix array of the original text T.

Existing distributed DC$\{3, 7, 13\}$ implementations [27, 6] broadly follow this straightforward adaptation of the algorithm to the distributed setting. However, this approach is not space-efficient. Materializing the X-prefixes of the suffixes for the final sorting step results in a memory blow-up proportional to X compared to the actual input. Consequently, sorting suffixes on distributed machines using DCX with large X is not feasible due to the limited main memory, even though DCX with $X > 3$ shows better performance on many real-world inputs [19]. We propose a technique to overcome this problem in the following Section 4.2.1.

4.2.1 Bucketing

In the sequential or shared-memory parallel setting, X-prefixes of suffixes can be sorted space-efficiently as each such element e can be represented as a pointer to the starting position of the suffix within the input text. This space-efficient sorting, however, is no longer possible in distributed memory. If we want to globally sort a distributed array of suffix-prefixes, we have to fully materialize and exchange them – resulting in a memory blow-up of at least a factor X. A simple idea to prevent this blow-up is to use a partitioning strategy which divides the elements of the distributed array into multiple buckets using splitter elements and processes only one bucket at a time.

We now describe a generalization of a bucketing technique for space-efficient sorting which has been previously proposed for scalable distributed string sorting [31, 28].

Whenever a distributed array of elements with a space-efficient representation has to be globally sorted, we first determine $q+1$ global splitter elements $s_0, s_1, \dots s_q$ with $s_0 = -\infty$ and $s_q = \infty$. We then locally partition the array into q buckets, such that element e with $s_k < e \leq s_{k+1}$ is placed in bucket k. We then execute q global sorting steps. In each step k, we materialize and communicate the elements from bucket k using a distributed sorting algorithm. Assuming that the splitters are chosen such that the global number of elements in each bucket is n/q and the elements within each bucket are equally distributed among the PEs (see Section 4.2.2 how this can be ensured), we only have to materialize $n/(pq)$ elements per bucket and PE instead of n/p elements per PE when using only one sorting phase.

By choosing q proportional to the memory blow-up caused by materializing an element, we can keep the overall memory consumption of this sorting approach in $\mathcal{O}(n/p)$.

4.2.2 Load-Balanced Bucketing via Random Chunk Redistribution

The global number of elements per bucket can be balanced by judiciously choosing the splitter elements, e.g., by regular or random sampling, or even using multi-sequence selection [36, 2]. However, the number of elements per PE within a bucket can vary greatly depending on the input. Assume an input which is already globally sorted with $q < p$ buckets. In this setting, all n/p elements located on the first PE have to be materialized when processing the first bucket. This results in memory blow-up and poor load-balancing across the PEs. Increasing the number of buckets q can only address the memory consumption issue but does not help with load-balancing.

A standard technique to resolve this kind of problem is a random redistribution of the elements to be sorted. However, this is not directly possible for elements which are stored in a space-efficient manner as in our case. We propose to solve this problem by randomly redistributing not single prefixes of suffixes but whole chunks of the input text before sorting. To be more precise, we partition T into consecutive chunks C_j of size c, i.e., $T = C_0 \circ \dots \circ C_{n/c-1}$. For the sake of simplicity, we assume chunks to be aligned with PE boundaries. Each chunk is then sent to a random PE.

▶ **Theorem 1** (Random Chunk Redistribution). *When redistributing chunks of size c uniformly at random across p PEs, with $q < p$ buckets each containing n/q elements, the expected number of elements from a single bucket received by a PE is $n/(pq)$.*

Furthermore, the probability that any PE receives $2n/(pq)$ or more elements from the same bucket is at most $1/p^\gamma$ for $n \geq 8c(\gamma+2)pq\ln(p)/3$ and $\gamma > 0$.

Proof. Let Y_i^k denote the number of elements belonging to bucket k which are assigned to PE i. In the following, we will determine the expected value of Y_i^k and show that $\mathbb{P}[Y_i^k \geq 2\mathbb{E}[Y_i^k]]$ is small. This will then be used to derive the above-stated bounds.

Let c_j^k be the number of elements belonging to bucket k in chunk j. For the sake of simplicity, we assume all buckets to be of equal size, thus, $\sum_{j=0}^{n/c-1} c_j^k = n/q$. We define

$$X_{j,i}^k = \begin{cases} c_j^k & \text{if chunk } j \text{ is assigned to PE } i \\ 0 & \text{otherwise,} \end{cases}$$

for chunk j with $0 \le j < n/c$, PE i with $0 \le i < p$, and bucket k with $0 \le k < q$. Thus, the random variable $X_{j,i}^k$ indicates the number of elements from bucket k received by PE i if chunk j is assigned to this PE. Hence, we can express Y_i^k as the sum over all $X_{j,i}^k$, i.e., $Y_i^k = \sum_{j=0}^{n/c-1} X_{j,i}^k$. As all chunks are assigned uniformly at random and there are p PEs, we furthermore have $\mathbb{E}[X_{j,i}^k] = c_j^k/p$. By the linearity of expectation, we can derive the expected value of Y_i^k as

$$\mathbb{E}[Y_i^k] = \mathbb{E}\left[\sum_{j=0}^{n/c-1} X_{j,i}^k\right] = \sum_{j=0}^{n/c-1} \mathbb{E}[X_{j,i}^k] = \sum_{j=0}^{n/c-1} \frac{c_j^k}{p} = \frac{n}{pq}.$$

For each bucket k, we now bound the probability $\mathbb{P}[Y_i^k \ge 2n/(pq)]$ that PE i receives two times its expected number of elements or more. We have

$$\mathbb{P}\left[Y_i^k \ge \frac{2n}{pq}\right] = \mathbb{P}\left[\sum_{j=0}^{n/c-1} X_{j,i}^k \ge \frac{2n}{pq}\right] = \mathbb{P}\left[\sum_{j=0}^{n/c-1} X_{j,i}^k - \mathbb{E}[X_{j,i}^k] \ge \frac{n}{pq}\right].$$

As the value of $X_{i,j}^k$ is bounded by the chunk size c, the Bernstein inequality [11, Theorem 2.10, Corollary 2.11] yields the following bound

$$\mathbb{P}\left[\sum_{j=0}^{n/c-1} X_{j,i}^k - \mathbb{E}[X_{j,i}^k] \ge \frac{n}{pq}\right] \le \exp\left(-\frac{\left(\frac{n}{pq}\right)^2}{2\left(\sum_{j=0}^{n/c-1} \mathbb{E}[(X_{j,i}^k)^2] + \frac{cn}{3pq}\right)}\right). \tag{1}$$

Since we find $\mathbb{E}[(X_{j,i}^k)^2] = (c_j^k)^2/p$, it follows that

$$\sum_{j=0}^{n/c-1} \mathbb{E}[(X_{j,i}^k)^2] = \sum_{j=0}^{n/c-1} (c_j^k)^2/p \le \frac{1}{p} \sum_{j=0}^{n/(qc)-1} c^2 = \frac{cn}{pq},$$

as the sum of the squares of a set of elements $0 \le a_i \le c$ with $\sum_i a_i = b$ and b divisible by c is maximized if they are distributed as unevenly as possible, i.e., $a_i = c$ for b/c elements and 0 for all others. We can use this estimation for an upper bound on the right-hand side of (1)

$$\exp\left(-\frac{\left(\frac{n}{pq}\right)^2}{2\left(\sum_{j=0}^{n/c-1} \mathbb{E}[(X_{j,i}^k)^2] + \frac{cn}{3pq}\right)}\right) \le \exp\left(-\frac{\left(\frac{n}{pq}\right)^2}{2\left(\frac{cn}{pq} + \frac{cn}{3pq}\right)}\right) = \exp\left(-\frac{3n}{8pqc}\right). \tag{2}$$

Combining these estimations, we obtain the bound

$$\mathbb{P}\left[Y_i^k \ge \frac{2n}{pq}\right] \le \exp\left(-\frac{3n}{8pqc}\right) \le \exp\left(-(\gamma+2)\ln p\right) = \frac{1}{p^{\gamma+2}}$$

for $n \ge 8pqc\ln(p)(\gamma+2)/3$. Using the union-bound argument yields the following estimation

$$\mathbb{P}\left[\bigcup Y_i^k \ge 2\frac{n}{pq}\right] \le \sum_{i=0}^{p-1}\sum_{k=0}^{q-1} \mathbb{P}[Y_i^k \ge 2\frac{n}{pq}] \le \sum_{i=0}^{p-1}\sum_{k=0}^{q-1} \frac{1}{p^{\gamma+2}} \le \frac{1}{p^\gamma}.$$

Hence, we obtain $\frac{1}{p^\gamma}$ as an upper bound on the probability that any PE receives more than two times the expected number of elements $n/(pq)$ for any bucket when assuming $q \le p$. ◀

Theorem 1 shows that combining a random chunk redistribution with the bucketing approach yields a space-efficient solution to the sorting problems occurring within our distributed variant of the DCX algorithm.

Within the DCX algorithm – depending on the actual step – we do not only redistribute chunks of the text but also corresponding rank entries and some bookkeeping information like the global index of a chunk. Furthermore, we require each chunk to have an *overlap* of X characters to ensure that an X-prefix for each element within a chunk can be constructed without communication. By choosing $c \geq X$, one can assure that the total size of a chunk remains in $\mathcal{O}(c)$. The redistribution can be achieved in time $\mathcal{O}(\alpha p + \beta n/p)$ as each PE sends and receives at most $\mathcal{O}(n/p)$ elements assuming the conditions of Theorem 1 are met.

4.2.3 Overall Analysis

We now briefly outline the running time of our algorithm using bucketing in conjunction with random chunk redistribution. All steps outside the recursion of distributed DCX are either global sorting steps, alltoall exchanges or local work linear in the size of the text per PE (see Algorithm 1). Additionally, we perform a constant number of allreduce and prefix sum operations, whose running time is dominated by the alltoall exchanges.

First, we consider the sorting steps. Assume that we want to sort a distributed array of m elements of type e with $\mathcal{O}(m/p)$ elements per PE, each of size $\mathcal{O}(e_s)$ while comparing any two elements takes time $\mathcal{O}(e_c)$. To facilitate the analysis, we assume a distributed (k-level) sorting routine with a running time

$$\mathrm{T}_{\mathrm{sort}}(m,p,k) = \mathcal{O}\left(\underbrace{e_c \frac{m}{p}\log m}_{\text{local work}} + \underbrace{\alpha k \sqrt[k]{p}}_{\text{start-up latency}} + \underbrace{\beta k e_s \frac{m}{p}}_{\text{communication costs}}\right)$$

which also balances the output, i.e., after sorting, each PE holds $\Theta(m/p)$ elements.[3] A sorting step with $q < p$ buckets and random chunk redistribution can then be realized in time

$$\begin{aligned}\widetilde{\mathrm{T}}_{\mathrm{sort}}(m,p,q,k) &= q\mathrm{T}_{\mathrm{sort}}(\mathcal{O}(m/q),p,k) + \mathcal{O}\left(\alpha p + \beta\frac{m}{p}\right)\\ &= \mathcal{O}\left(\underbrace{e_c \frac{m}{p}\log m}_{\text{local work}} + \underbrace{\alpha(p + qk\sqrt[k]{p})}_{\text{start-up latency}} + \underbrace{\beta k e_s \frac{m}{p}}_{\text{communication costs}}\right)\end{aligned}$$

with probability $\geq 1 - 1/p^{\gamma}$ for any fixed $\gamma > 0$ and sufficiently large m, increasing only the latency term due to q sorting steps and an additional alltoall exchange. Note that the time required for the computation of the bucket splitters and subsequent classification is dominated by the running time mentioned above when using regular sampling with $\Theta(q)$ samples per PE and the distributed (k-level) sorting routine.

In this setting, we see that all sorting steps (and the alltoall exchanges) in DCX are dominated by the final sorting of all suffixes, where we have $e_s = \mathcal{O}(X \log \sigma + \sqrt{X} \log n)$[4] and $e_c = \mathcal{O}(X)$. Due to the geometric decrease of the input size within the recursion, we

[3] Note that for example the running time of adaptive multi-level sample sort (AMS) [2, Theorem 3] using k levels of indirection comes close to this. AMS also balances the output.

[4] Within the recursive calls, we have $e_s = \mathcal{O}(X \log n)$ as the alphabet size is now $\mathcal{O}(n)$. However, as the problem size also decreases by a factor $\Theta(\sqrt{X})$ [13, 26], this increase in alphabet size can be accounted for by the $\mathcal{O}(\sqrt{X} \log n)$ term in e_s of the first level.

reach an overall problem size of $\mathcal{O}(n/p)$ after $\mathcal{O}(\log p)$ iterations. Gathering all remaining data on a single PE and applying the sequential DCX algorithm yields the running time stated in Theorem 2.

▶ **Theorem 2.** *Using a distributed sorting routine with the above-stated properties, our lightweight distributed DCX algorithm has a running time*

$$DCX(n,p) = \mathcal{O}\left(\underbrace{X\frac{n}{p}\log n}_{\textit{local work}} + \underbrace{\alpha\log(p)(p + qk\sqrt[k]{p})}_{\textit{start-up latency}} + \underbrace{\beta X\frac{n}{p}\left(\log(\sigma) + \frac{\log n}{\sqrt{X}}\right)}_{\textit{communication costs}}\right)$$

with probability $\geq 1 - 1/p^{\gamma}$ *for any fixed* $\gamma > 0$ *and sufficiently large* n.

4.2.4 Further Optimizations

In addition to the techniques described above, we also utilize discarding and packing, two techniques commonly used in distributed- and external-memory SACAs.

Discarding. After sorting the X-prefixes of the sample suffixes, we have to recursively apply the DCX algorithm (or any other suffix sorting algorithm) to a smaller subproblem if there are duplicate ranks. However, in order to obtain overall unique ranks for the sample suffixes, we do not have to recurse on all of them but can discard suffixes whose ranks are already unique after initial sorting. This *discarding* technique has been used in the external-memory setting [15] but has not been explored for distributed memory yet.

Packing. Packing is an optimization for small-sized (integer) alphabets proposed for distributed memory prefix-doubling by Flick et al. [20]. Assume $b = \lceil\log\sigma\rceil < B$, where B is the size of one machine word. Instead of using one machine word per character, we can pack up to B/b characters into one word and exchange/sort them at once.

5 Experimental Evaluation

For our extensive evaluation, we use up to 256 compute nodes (12 288 cores) of SuperMUC-NG where each node is equipped with one Intel Skylake Xeon Platinum 8174 processor with 48 cores and 96GB of main memory.[5] The internal interconnect is a fast OmniPath network with 100 Gbit/s.

We compare the following algorithms. See Table 1 for an overview.

ℓDCX. Our implementation of the lightweight distributed DCX algorithm described above. In the experiments, we use $X = 39$. In preliminary experiments, we found this value to provide the best trade-off between reduction ratio and running time per level. We use the bucketing technique for sorting the DC samples and also for the final sorting of all suffixes. For sorting the DC samples, we use 16 buckets in the first two levels of recursion. For the final merging, we use 64 buckets in the first two levels and 16 in a third level. In later levels of recursion, the remaining number of suffixes is small enough so that all elements can

[5] We also conducted preliminary experiments on the HoreKa supercomputer. As the results obtained there are in line with our findings from SuperMUC-NG presented here, we omit them.

be sorted at once. Bucket splitters are determined by centrally sorting a random sample of 2×10^4 elements and then drawing the splitters equidistantly from the sorted sample elements. Additionally, we employ our randomized redistribution of chunks with 10^4 chunks per PE. We use AMS-sort by Axtmann et al. [2, 3] as our distributed sorting routine. Due to the large number of single sorting steps, we require a low-latency sorting algorithm (at least for large PE configurations). Therefore, we use AMS with two levels as a default. As the discarding optimization has certain overheads, we only apply it when the reduction ratio is more than 0.7. Furthermore, we use 40-bit integers for storing rank information to reduce the memory footprint.

PSAC [20]. We explore two different original configurations. PSAC is the standard (more memory-efficient) configuration performing prefix-doubling as described in Section 3. Once the ratio of suffixes with non-unique h-prefix falls below $n/10$, it switches to another algorithm which tries to avoid global sorting steps as much as possible but has a somewhat higher memory consumption. This approach sorts each non-unique range of suffixes with identical h-prefix independently by splitting the PEs into different groups and can be regarded as a kind of discarding. The configuration PSAC$^+$ runs this second algorithm immediately. Additionally, we implemented a 40-bit integer version of their algorithm as well as a version which uses AMS for distributed sorting. Due to the small performance benefit of using AMS, we do not include these variants in the plots.

dPD [19]. This algorithm also relies on distributed prefix doubling with discarding, i.e., it does not continue sorting suffixes whose h-prefix is already unique. In contrast to PSAC$^+$, dPD redistributes the remaining non-unique suffixes across all PEs and keeps using a global sorting routine. dPD uses 40-bit integers.

The original sorting algorithm within dPD gathers suffixes with identical h-prefix on the same PE. While this facilitates the rank computation for the next prefix-doubling round, it introduces severe load-balancing problems in case of many identical h-prefixes. We provide an improved version of this algorithm, where identical h-ranks may span multiple consecutive PEs. In combination with AMS as a sorting subroutine, this yields a much more scalable algorithm with a substantially improved load-balancing. We refer to this new variant as dPD* (dPD^{*,1} and dPD^{*,2} indicates the usage of one- or two-level AMS sort).

B-DC$_X$ [6]. A straightforward adaptation of the sequential DCX algorithm to the distributed-memory setting for $X = \{3, 7, 13\}$. The implementation is limited to inputs of size 4 GB due to the internal usage of 32-bit integers.

Other Algorithms. There also exists a lightweight distributed suffix sorting algorithm based on induced copying [19]. Unfortunately, we were not able to run this algorithm successfully on our system. It also suffers from the same load-balancing issues as dPD. While this algorithm exhibits a relatively low memory blow-up factor of around 30 on up to 1 280 cores, the running-time of the algorithm is reported to be always slower than or on par with the default configuration of PSAC or the non-load-balanced dPD.

There also exists prefix-doubling and DC3/DC7 implementations [10] in the distributed external-memory BigData framework *Thrill* [8]. Although the framework supports MPI, their focus is on non-MPI data center clusters. We were not able to execute their algorithms on more than one compute node because Thrill relies on external local disk storage, which

Table 1 Overview of algorithms used in this experimental evaluation. All algorithms, except for libsais are distributed-memory algorithms.

Name	Brief Description	Reference
ℓDCX	Our main contribution described in Section 4	[here]
PSAC	Prefix doubling w/o discarding, last 10 % are sorted with PSAC$^+$	[20]
PSAC$^+$	Prefix doubling, each h-group is sorted independently (discarding-like)	[20]
PSAC$_{40}$	Same as PSAC but using 40-bit integers	[here]
PSAC$^+_{40}$	Same as PSAC$^+$ but using 40-bit integers	[here]
dPD	Prefix doubling w/ discarding, 40-bit integers, and PE-local ranks	[19]
dPD^{*,1}	Same as dPD but w/o PE-local ranks and using AMS with one level	[here]
dPD^{*,2}	Same as dPD but w/o PE-local ranks and using AMS with two levels	[here]
B-DC$_X$	Basic MPI implementation of DCX for $X = \{3, 7, 13\}$	[6]
libsais	Fastest sequential and shared-memory suffix sorting algorithm	[24]

is very limited on our cluster. On a single compute node (48 cores), their algorithms were more than a factor of two slower than ℓDCX on all data sets but DNA-Rep. Here, their DC7 implementation is only a factor of 1.8 slower than our algorithm.

Recently and independently, Ferguson also started working on a variant of a distributed-memory difference cover algorithm using a similar bucketing approach, implemented in the Chapel parallel programming language [16]. Their algorithm does not use random chunk redistribution and other optimizations present in our implementation. Preliminary comparisons executed by Ferguson on a low-latency Cray supercomputer demonstrated that our implementation was approximately twice as fast on a DNA dataset. In preliminary experiments on our system – which has higher communication latency – Ferguson's implementation exhibited less favorable scaling behavior. Additionally, it requires around 20×–80× the input size as working memory. Therefore, it is not included in our evaluation.

In addition to the above-mentioned distributed-memory algorithms, we also compared our algorithm to the currently fastest and highly-engineered sequential and shared-memory parallel suffix sorting implementation libsais [24].

General Settings and Data Sets. All algorithms are implemented in C++ (C for libsais) and compiled with `GCC 12.2.0` with optimization flags `-O3` and `-march=native`. We use IntelMPI 2021.11 for interprocess communication. Additionally, our DCX implementation uses the zero-overhead MPI bindings KaMPIng [40]. Reported times are the average of at least three runs. We generally observed only very low variations in running time.

We use inputs from the five data sets given in Table 2 with different characteristics regarding the alphabet size σ and the longest common prefix (LCP) distribution.

5.1 Weak-Scaling Experiments

Figure 1 shows the running times of weak-scaling experiments with 20 MB of text data per PE (960 MB per compute node). In Figure 2, we present the corresponding memory blow-up, i.e., the maximum peak memory aggregated over each compute node divided by the total input size on a compute node. We read the maximum *resident set size (rss)* of each MPI process to obtain these values.

Table 2 Data sets used in our evaluation. Statistics consider the first 50 GB of each data set.

Name	Brief Description	σ	avg. LCP	max LCP
CC [14]	Websites crawled by Common Crawl Project	243	$1{,}04 \times 10^4$	$1{,}84 \times 10^6$
WIKI [41]	XML data sets of Wikipedia	213	396.07	$1{,}58 \times 10^6$
PROT [39]	Protein data from *Universal Protein Resource*	26	179.31	$3{,}43 \times 10^4$
DNA [38]	DNA Data from *1000 Genomes Project*	4	25.15	$3{,}57 \times 10^3$
DNA-REP	Concatenate first MB of DNA 245 760×	4	$2{,}50 \times 10^{10}$	$5{,}00 \times 10^{10}$

We see that ℓDCX is the fastest (distributed) algorithm on CC on all evaluated numbers of PEs. Compared to the previous algorithms, we achieve speedups of up to 4.3, while being at least 1.6× more memory-efficient. The improved prefix-doubling algorithm dPD* performs better, especially when using AMS with two levels (dPD^{*,2}). However, ℓDCX is still about a factor of 1.6 faster. A similar trend can be seen for WIKI, although less pronounced, with speedups of 3.2 and 1.2 on 256 nodes, respectively. This is due to the lower LCP values of WIKI compared to the CC data set. The prefix-doubling algorithms require fewer iterations and therefore perform better. This becomes even more evident for DNA and PROT, as these data sets have even lower LCP values. Additionally, the smaller alphabet size of these texts makes the packing optimization more effective. These factors explain the lower running time of all algorithms on these data sets. On DNA, our improved dPD* is fastest on 256 compute nodes and outperforms ℓDCX by a factor of 1.2.

Due to the repetitive nature of DNA-REP, the data set has very high average LCP values and therefore is a particularly hard instance for prefix-doubling algorithms. Consequently, these algorithms were not able to terminate within the time limit for all PE configurations (or even crashed on some configurations). Our ℓDCX algorithm is, due to its algorithmic properties, largely insensitive to this input characteristic and shows a very good scaling behavior even on this hard input instance with speedups of up to 5.5 over its competitors.

The DC$\{3, 7, 13\}$ implementation by Bingmann (B-DC$_X$) does not scale well.[6] However, at least the DC$\{7, 13\}$ variants have competitive running times for CC and WIKI on up to two compute nodes. Nonetheless, their memory consumption reveals a key problem of a straightforward adaptation of DCX to distributed memory. Although using 32-bit integers internally, DC3 exhibits a blow-up of $\approx 50\times$ while DC7 and DC13 have a blow-up of $\geq 60\times$ and $\approx 80\times$, respectively.

The running times for the shared-memory parallel variant of libsais (using all 48 cores of one compute node) are given for the input of all p cores. We see that libsais is faster than ℓDCX when using only one compute node. This is not surprising as libsais leverages native OpenMP parallelization while our implementation of DCX uses interprocess-communication via MPI even when running on only one compute node. But from only two compute nodes on (4 compute nodes for DNA-REP) ℓDCX clearly surpasses libsais effectively utilizing the additional cores. We cannot run libsais on inputs larger than $8 \times 48 \times 20$ MB due to the limited main memory of 96 GB per compute node.

Overall, our lightweight distributed DCX algorithm shows a very good scaling behavior and is either fastest or competitive with our improved version of prefix-doubling with discarding, which is fastest on the remaining inputs (again at least for large PE configurations).

[6] While the scaling behavior of the implementation could certainly be improved, we chose not to do so as our engineered space-efficient ℓDCX algorithm already supersedes it in terms of implementation.

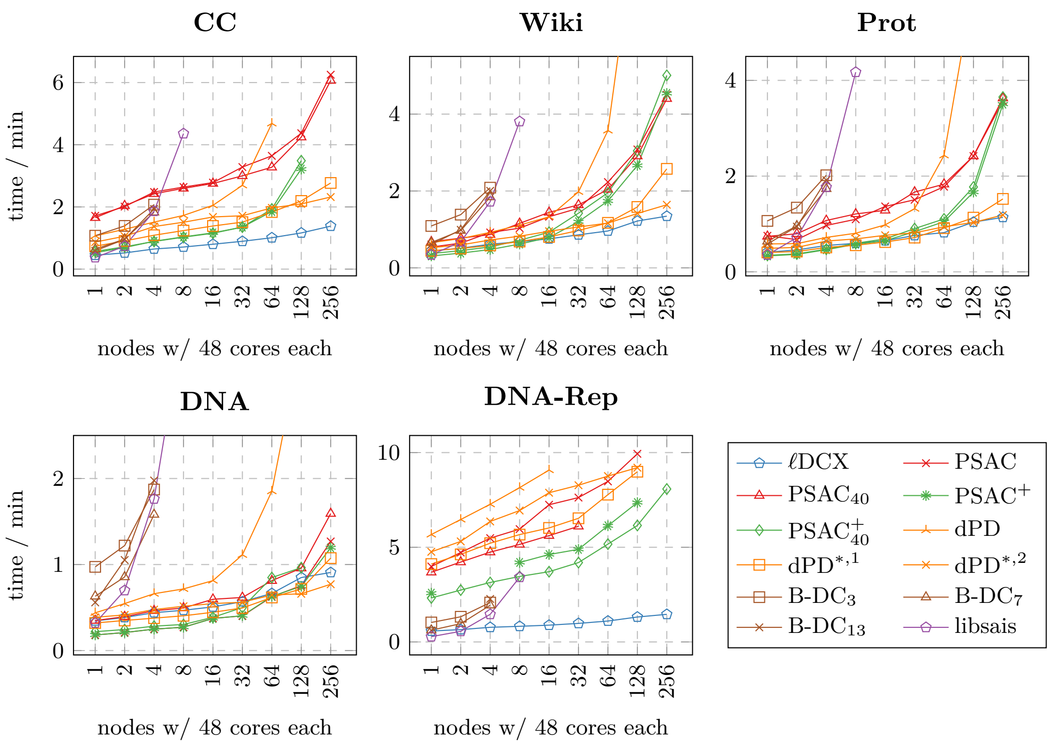

Figure 1 Running time in minutes of the suffix array construction algorithms in our weak-scaling experiments with 20 MB per PE. Missing data points are due to time limits or crashes.

Regarding the memory consumption (Figure 2), we see that ℓDCX is clearly the most memory-efficient one of all evaluated distributed algorithms due to the bucketing with randomized chunk redistribution approach. We see a slight increase in memory consumption for larger PE configurations which we assume to be caused by external factors such as the internal buffer allocation strategy of the MPI implementation and the release policy of memory allocators as we could not detect a significant increase in bucket imbalances.

5.2 Breakdown Experiments

Figure 3 shows the result of a breakdown test, where we run the algorithms on 16 compute nodes (768 cores) and increase the input size per PE until the algorithms can no longer finish due to memory limitations. We start the experiment with 20 MB (15.36 GB in total). The standard PSAC variants using 64-bit integers can handle up to 30 MB (at least for DNA). The 40-bit variants as well as the different variants of dPD can handle up to 40 MB of text per PE. Our ℓDCX implementation is, as expected, far more memory-efficient and can handle up to 130 MB of input per PE (6.24 GB per compute node). With inputs size larger than 100 MB per PE, the maximum memory consumption of ℓDCX is only between 14× to 15× the input size. We attribute this to fixed-size memory overheads of the MPI runtime becoming less important with increasing input size. These values come close to the theoretically expected memory consumption of our implementation.

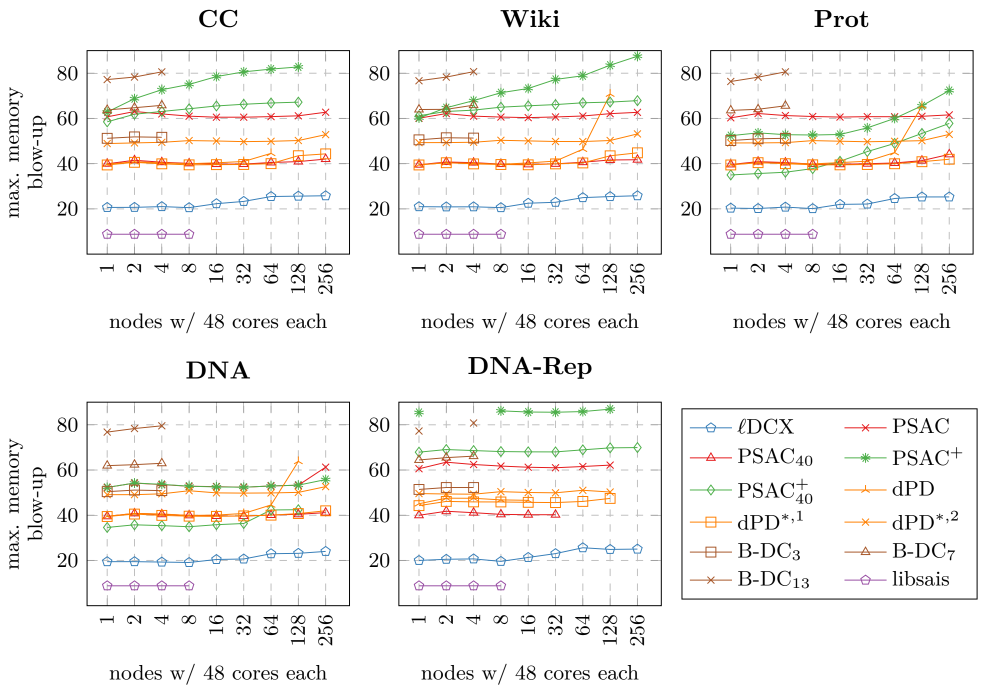

Figure 2 Memory blow-up of the suffix array construction algorithms in our weak-scaling experiments with 20 MB per PE.

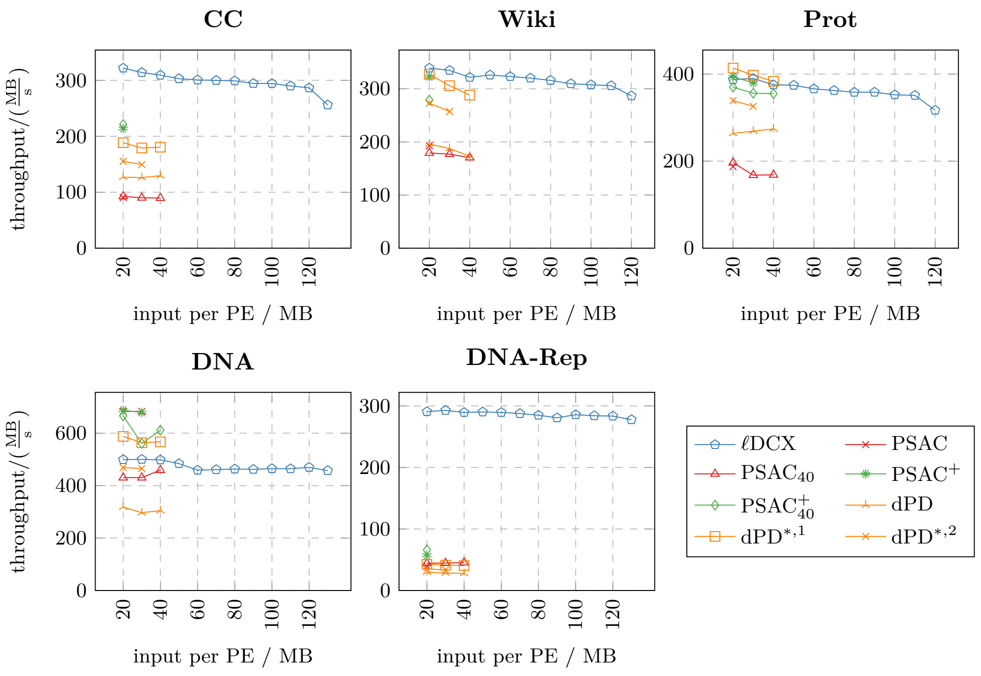

Figure 3 Breakdown test on 16 compute nodes (768 cores) with increasing input size.

6 Conclusion and Future Work

In this work, we present a fast and lightweight distributed variant of the DCX suffix sorting algorithm. We rely on a bucketing scheme for space-efficient distributed sorting to reduce the algorithm's memory consumption and combine it with a load-balancing approach based on random chunk redistribution. In our extensive experimental evaluation, we show that our algorithm is up to 5× faster than previous distributed suffix sorting algorithms while being substantially more memory-efficient. For future work, we plan to extend our bucketing and random chunk redistribution approach to other models of computing such as the distributed external-memory model and distributed multi-GPU architectures.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

2 Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. In *SPAA*, pages 13–23. ACM, 2015. doi:10.1145/2755573.2755595.

3 Michael Axtmann and Peter Sanders. *Robust Massively Parallel Sorting*, pages 83–97. SIAM, 2017. doi:10.1137/1.9781611974768.7.

4 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 *SPIRE*, volume 11811 of *Lecture Notes in Computer Science*, pages 407–416. Springer, 2019. doi:10.1007/978-3-030-32686-9_29.

5 Uwe Baier. Linear-time suffix sorting – A new approach for suffix array construction. In *CPM*, volume 54 of *LIPIcs*, pages 23:1–23:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.CPM.2016.23.

6 Timo Bingmann. pdcx. https://github.com/bingmann/pDCX, 2018.

7 Timo Bingmann. *Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2018. URL: https://publikationen.bibliothek.kit.edu/1000085031.

8 Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with c++. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 172–183. IEEE, 2016. doi:10.1109/BIGDATA.2016.7840603.

9 Timo Bingmann, Patrick Dinklage, Johannes Fischer, Florian Kurpicz, Enno Ohlebusch, and Peter Sanders. Scalable text index construction. In *Algorithms for Big Data*, volume 13201 of *Lecture Notes in Computer Science*, pages 252–284. Springer, 2022. doi:10.1007/978-3-031-21534-6_14.

10 Timo Bingmann, Simon Gog, and Florian Kurpicz. Scalable construction of text indexes with thrill. In *IEEE BigData*, pages 634–643. IEEE, 2018. doi:10.1109/BIGDATA.2018.8622171.

11 Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration Inequalities – A Nonasymptotic Theory of Independence*. Oxford University Press, 2013. doi:10.1093/ACPROF:OSO/9780199535255.001.0001.

12 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center, 1994.

13 Charles J Colbourn and Alan CH Ling. Quorums from difference covers. *Information Processing Letters*, 75(1-2):9–12, 2000. doi:10.1016/S0020-0190(00)00080-6.

14 Common Crawl. Common Crawl WET Files from CC-MAIN-2019-09. https://commoncrawl.org, 2019. Downloaded WET files from segments: https://data.commoncrawl.org/crawl-data/CC-MAIN-2019-09/segments/1550247479101.30/wet/CC-MAIN-20190215183319-20190215205319-#ID.warc.wet, where #ID ranges from 00000 to 000639, and https://data.commoncrawl.org/crawl-data/CC-MAIN-2019-09/segments/1550247479159.2/wet/CC-MAIN-20190215204316-20190215230316-#ID.warc.wet, where #ID ranges from 00000 to 00199. Only textual content was retained; HTML tags and Common Crawl metadata were removed.

15 Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. *ACM J. Exp. Algorithmics*, 12:3.4:1–3.4:24, 2008. doi:10.1145/1227161.1402296.

16 Michael Ferguson. ssort_chpl: Chapel-based suffix sorting module, 2025. Accessed: 2025-04-21. URL: https://github.com/femto-dev/femto/tree/main/src/ssort_chpl.

17 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.

18 Johannes Fischer and Florian Kurpicz. Dismantling divsufsort. In *Stringology*, pages 62–76. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017. URL: http://www.stringology.org/event/2017/p07.html.

19 Johannes Fischer and Florian Kurpicz. Lightweight distributed suffix array construction. In *ALENEX*, pages 27–38. SIAM, 2019. doi:10.1137/1.9781611975499.3.

20 Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In *SC*, pages 16:1–16:10. ACM, 2015. doi:10.1145/2807591.2807609.

21 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

22 Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. In *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice-Hall, 1992.

23 Keisuke Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In *Stringology*, pages 111–125. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019. URL: http://www.stringology.org/event/2019/p11.html.

24 Ilya Grebnov. libsais: A fast linear time suffix array and burrows-wheeler transform construction library. https://github.com/IlyaGrebnov/libsais, 2025. Accessed: 2025-04-19.

25 Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *SPIRE/CRIWG*, pages 81–88. IEEE, 1999. doi:10.1109/SPIRE.1999.796581.

26 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

27 Fabian Kulla and Peter Sanders. Scalable parallel suffix array construction. *Parallel Comput.*, 33(9):605–612, 2007. doi:10.1016/J.PARCO.2007.06.004.

28 Florian Kurpicz, Pascal Mehnert, Peter Sanders, and Matthias Schimek. Scalable distributed string sorting. In *ESA*, volume 308 of *LIPIcs*, pages 83:1–83:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ESA.2024.83.

29 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *DCC*, page 422. IEEE, 2018. doi:10.1109/DCC.2018.00075.

30 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *SODA*, pages 319–327. SIAM, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

31 Pascal Mehnert. Scalable distributed string sorting algorithms. Master's thesis, Karlsruhe Institute of Technology, Germany, 2024.

32 Yuta Mori. libdivsufsort. https://github.com/y-256/libdivsufsort, 2015.

33 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. doi:10.1109/TC.2010.188.

34 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

35 Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox.* Springer, 2019. doi:10.1007/978-3-030-25209-0.

36 Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *J. Parallel Distributed Comput.*, 14(4):361–372, 1992. doi:10.1016/0743-7315(92)90075-X.

37 Zachary D Stephens., Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13(7):1–11, July 2015. doi:10.1371/journal.pbio.1002195.

38 The 1000 Genomes Project. FASTQ Files from the 1000 Genomes Project. ftp://ftp.sra.ebi.ac.uk/vol1/fastq/DRR000/, 2025. Downloaded FASTQ files with numbers DRR000001 to DRR000439 (excluding DRR000394). Only the DNA sequence lines were retained; characters other than A, C, G, and T were removed.

39 The UniProt. FASTA Files from the Universal Protein Resource (UniProt). https://ftp.uniprot.org/pub/databases/uniprot/current_release/uniparc/fasta/active/. From https://ftp.uniprot.org/pub/databases/uniprot/current_release/uniparc/fasta/active/uniparc_active_p#ID.fasta.gz, where #ID ranges from 1 to 200. Only sequence representations were retained.

40 Tim Niklas Uhl, Matthias Schimek, Lukas Hübner, Demian Hespe, Florian Kurpicz, Daniel Seemaier, Christoph Stelz, and Peter Sanders. KaMPIng: Flexible and (near) zero-overhead C++ bindings for MPI. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024. doi:10.1109/SC41406.2024.00050.

41 Wikimedia Foundation. Wikipedia XML Dumps (March 2019) for de, en, es, fr. https://dumps.wikimedia.org/, 2019. Downloaded files available at https://dumps.wikimedia.org/#IDwiki/20190320/#IDwiki-20190320-pages-meta-current.xml.bz2, where #ID is *de*, *en*, *es*, and *fr*.