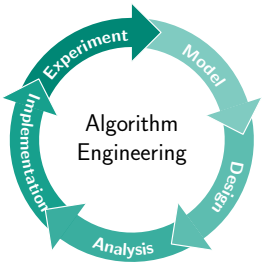


Distributed Suffix Sorting

Master's Thesis of

Manuel Haag

at the Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

Reviewer:	Prof. Dr. Peter Sanders
Advisor:	M.Sc. Matthias Schimek
Second advisor:	Dr. Florian Kurpicz

20. September 2024 – 20. March 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . . . .

(Manuel Haag)

Abstract

Suffix arrays [62] are one of the most fundamental text indices that form the basis for fast substring search in DNA or other text corpora, text compression and many other string algorithms. Their construction, also referred to as *suffix sorting*, consists in determining the lexicographic order of all suffixes of a text. With the rapid growing availability of genomic sequence and increasing amount of textual data in today's information age, scalable constructions of suffix arrays become of tremendous importance. To this end, we implement and evaluate a recursive distributed memory suffix array algorithm, called *Difference Cover modulo X* (DCX), that leverages the combinatorial structure of *difference covers* to efficiently sort the suffixes of the text. Our implementation adapts the sequential DCX algorithm [46] and is a generalization to larger difference covers of the distributed DC3 implementation by Kulla and Sanders [52]. Furthermore, we propose a new load-balancing method for compressed input representations of overlapping strings. In our experimental evaluation on up to 6144 cores, we show that distributed DCX is scalable, fast and still space-efficient, requiring 20 bytes per input and only 14.2 bytes for larger inputs (92.16 GB). On 3 out of 4 real world instances, we outperform the current state-of-the-art achieving speedups of up to 3.2×, while being only 1.15× slower on a DNA data set for 6144 cores. Although requiring more time on DNA data, our algorithm offers a good time-space trade-off, enabling it to handle much larger text inputs. Moreover, we are able to process texts 3× as large as our competitor using the same amount of RAM and compute resources.

Zusammenfassung

Suffixarrays [62] sind einer der grundlegendsten Textindizes, die die Grundlage für die schnelle Teilzeichensuche in DNA oder anderen Textkorpora, die Textkompression und viele andere String-Algorithmen bilden. Ihre Konstruktion, die auch als *Suffixsortierung* bezeichnet wird, besteht in der Bestimmung der lexikographischen Reihenfolge aller Suffixe eines Textes. Mit der schnell wachsenden Verfügbarkeit von genomischen Sequenzen und der zunehmenden Menge an Textdaten im heutigen Informationszeitalter sind skalierbare Konstruktionen von Suffixarrays von enormer Bedeutung. Zu diesem Zweck implementieren und evaluieren wir einen rekursiven Suffixarray-Algorithmus mit verteiltem Speicher, *Difference Cover modulo X* (DCX), der die kombinatorische Struktur des *Difference Cover* nutzt, um die Suffixe des Textes effizient zu sortieren. Unsere Implementierung passt den sequentiellen DCX-Algorithmus [46] an und ist eine Verallgemeinerung der verteilten DC3-Implementierung von Kulla und Sanders [52] auf größere Difference Cover. Darüber hinaus schlagen wir eine neue Lastverteilungsmethode für komprimierte Eingabedarstellungen von überlappenden Zeichenketten vor. In unserer experimentellen Evaluierung auf bis zu 6144 Kernen zeigen wir, dass verteiltes DCX skalierbar, schnell und dennoch speichereffizient ist. Es benötigt 20 Byte pro Eingabe und nur 14,2 Byte für größere Eingaben (92,16 GB). Bei 3 von 4 realen Instanzen übertreffen wir den aktuellen Stand der Technik und erreichen Beschleunigungen von bis zu 3,2×, während wir bei einem DNA-Datensatz für 6144 Kerne nur 1,15× langsamer sind. Obwohl unser Algorithmus bei DNA-Daten mehr Zeit benötigt, bietet er ein guten Kompromiss zwischen Zeit und Platz, das es ihm ermöglicht, viel größere Texteingaben zu verarbeiten. Darüber hinaus sind wir in der Lage, Texte zu verarbeiten, die 3× so groß sind wie die unseres Konkurrenten, wobei wir die gleiche Menge an RAM und Rechenressourcen verwenden.

Acknowledgments

I would like to thank my supervisors, Matthias Schimek and Florian Kurpicz, for guiding me through this project with their expertise, helpful advice and encouraging support.

I 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 at Leibniz Supercomputing Centre (`www.lrz.de`).

Contents

Abstract i

Zusammenfassung ii

1. Introduction 1
- 1.1. Contribution . . . 2
- 1.2. Outline . . . 2

2. Preliminaries 4
- 2.1. Definitions and Notations . . . 4
- 2.2. Model of Computation . . . 4

3. Related Work 7
- 3.1. Sequential Suffix Sorting . . . 7
- 3.2. Distributed Suffix Sorting . . . 9
  - 3.2.1. Difference Cover Modulo X . . . 9
  - 3.2.2. PSAC Using Global Sorting . . . 9
  - 3.2.3. PSAC With Avoiding Global Sorting . . . 12
  - 3.2.4. Distributed Prefix Doubling with Discarding . . . 13
  - 3.2.5. Distributed DivSufSort . . . 13

4. Building Blocks 18
- 4.1. Distributed Atomic Sorting . . . 18
- 4.2. Distributed String Sorting . . . 20
  - 4.2.1. Sequential String Sorting . . . 20
  - 4.2.2. LCP-Merging . . . 21
  - 4.2.3. LCP-Compression . . . 22
  - 4.2.4. Distinguishing Prefix Approximation . . . 23

5. Suffix Sorting using Difference Covers 25
- 5.1. Example Execution of DC3 Algorithm . . . 25
- 5.2. Difference Cover Samples . . . 27
- 5.3. The General DCX Algorithm . . . 28
- 5.4. The Distributed DCX Algorithm . . . 30
- 5.5. Discarding . . . 37
- 5.6. Space-Efficient Sorting Using Bucketing . . . 39
- 5.7. Chunking . . . 43
- 5.8. Further Optimizations . . . 44
  - 5.8.1. Incorporating String Sorting . . . 45

5.8.2. Packing . . . 46
5.9. Implementation Details . . . 46
5.9.1. Static String Containers . . . 46
5.9.2. Smaller Data Types for Ranks . . . 47
5.9.3. Bucketing . . . 47
5.10. Suffix Array Checking Algorithm . . . 48

6. Experimental Evaluation **50**
6.1. Experimental Setup . . . 50
6.2. Evaluating Distributed DCX . . . 52
6.2.1. String Sorting Variants . . . 53
6.2.2. Comparison-based-Sorting vs. String Sorting . . . 55
6.2.3. Discarding . . . 56
6.2.4. Chunk Sizes . . . 57
6.2.5. Bucket Sizes . . . 58
6.3. Comparison with State-of-the-Art . . . 59
6.3.1. Weak-Scaling Experiments . . . 60
6.3.2. Breakdown Test . . . 61

7. Conclusion **70**

A. Appendix **80**
A.1. Theoretical Guarantees of Random Chunk Redistribution . . . 80

List of Figures

2.1. Suffix array of the text T = `banana$` with LCP-array. The distinguishing prefixes are colored in red. 5

3.1. Timeline of *sequential* suffix array construction with algorithms that share techniques are marked with an arrow. Figure based on [12, 54, 81]. The three techniques are shown as columns and algorithms that combine multiple techniques are crossing the borders. Suffix array construction algorithms with linear running time are highlighted in dark gray. If an implementation is publicly available, the algorithm is also marked in brown. 8
3.2. Example execution of PSAC1 from original paper [32]. 10

4.1. LCP-aware Loser Tree with eight input streams. 23

6.1. Running times and blowup of DCX for different period lengths X with different string sort optimizations, 15.36 GB input and 768 PEs. 63
6.2. Running times and blowup of DCX for different period lengths X with different sorter configurations, 15.36 GB input and 768 PEs. 64
6.3. Barplot shows time spent on the on the first level (first bar), the second level (second bar) and time spent on the remaining levels (third bar) for different period lengths X with 15.36 GB input and 768 PEs. 65
6.4. Influence of discarding with different period lengths X on 15.36 GB input and 768 PEs. 66
6.5. Influence of block sizes on 15.36 GB input and 768 PEs. 67
6.6. Running times and blowup of the suffix array algorithms in our weak scaling experiments with 20MB per PE. 68
6.7. Throughput and blowup during our breakdown test on 768 PEs. 69

List of Tables

4.1. Complexity of parallel sorting algorithms [7]. Implicit $O(\cdot)$. $k = \sqrt[r]{p}$, where r is the number of levels used in AMS. 19

5.1. Table of difference covers D_X for $X = 2, 3 \ldots, 12$ and their corresponding value of the reduction ratio λ. 28
5.2. Example Phase 1 of distributed DCX. 31
5.3. Example Phase 2 of distributed DCX. 31
5.4. Example Phase 3 of distributed DCX. 32
5.5. Example Phase 4 of distributed DCX. 34
5.6. Example of discarding. Unique ranks are colored in red and unique ranks that are required to determine not-unique ranks are marked with an arrow. 38
5.7. Example of discarding in Phase 3. Unique ranks are colored in red and unique ranks that are required to determine not-unique ranks are marked with an arrow. Unused tie-break ranks r' are colored in gray. 39
5.8. Example bucketing of 3-prefixes with 3 PEs and 3 buckets. 41
5.9. Example random redistribution with none-overlapping chunks. Send/receive buffers are sorted by sending/receiving PE and then by their original order. 44

6.1. LCP statistics and alphabet size of our texts. LCPs were computed on the first 50 GB of the data. 51
6.2. Quantiles of LCPs of our texts. LCPs were computed on first 50 GB of the data. 52
6.3. Reduction of communication volume in Phase 4 on level for DC133. The Char columns shows the reduction of send characters and Total the overall reduction. The last column is the average length of the approximated distinguishing prefix. 54
6.4. Number of packed 64-bit words and chars in PACKED. 56
6.5. Time spent on sorting in Phase 4 on level 0 and 1 for DC39. 56
6.6. Text size on level 1, 2, 3 of DC39-DISCARD. Empty cells indicate that the algorithm already finished. For DC39-NO-DISCARD, the text-sizes are $(7/39)^i$ for $i \in \{1, 2, 3\}$: 17.95%, 3.22%, 0.58% independent of the input text. 57
6.7. Time, memory and imbalance for chunking experiment with DC39. The best values in each row are marked in bold. 58
6.8. Our best configuration of DCX. 59

List of Algorithms

1. Inducing Step DivSufSort 16
2. Hypercube Algorithm Design Pattern 20
3. Distributed DCX. 35
4. Distributed DCX Phase 3 with Discarding. 40
5. Bucketing Technique 42
6. Distributed Suffix Array Checking. 49

1. Introduction

The *suffix array* [62] is one of the most popular data structures for text indexing and string algorithms. It stores the permutation of all suffix positions in sorted lexicographic order. Although, the total length of all suffixes is quadratic in the text size, optimal construction in linear time requiring only constant working space in addition to the suffix array are possible [38, 60]. Suffix array construction algorithms typically exploit the fact that the suffixes overlap, differentiating them from the related problem of sorting variable-sized strings.

There are numerous applications of suffix arrays in efficient string processing. Enhanced with the *longest common prefix array* they serve as a space-efficient replacement of *suffix-trees* [2], one of the most powerful full-text indices. Using suffix arrays, efficient localization of patterns in unstructured text is possible in time proportional to the *pattern* length as opposed to the *text* length [3]. In text compression, suffix arrays allow to compute the *Burrows-Wheeler transform* [22], which is the backbone of many compressed full-text indices [29, 35]. Further, they are widely applied to solve a variety of real-world applications in computational biology [61, 78] such as DNA-sequencing [37, 92].

The ever increasing amount of textual data in today's information age creates a need for scalable text processing algorithms. Many of which use the suffix array as building block. Due to technical advances, the availability of sequence genomic data is growing rapidly [89]. In 2020, all public source code repositories on GitHub were archived and requires more than 21 TB to store[1]. A recent dataset of December 2024[2] created by CommonCrawl, a free open repository of web crawl data, contains about 2.64 billion web pages of 394 TiB uncompressed content.

Suffix array construction algorithms have been well studied in the *sequential* setting [9, 12, 30, 42, 46, 81], in *shared-memory* [56, 57, 79], in *external-memory* [25, 43, 46] and to a somewhat smaller degree in *GPU* [20, 26] and in *distributed* [31, 32, 52] settings. Sequential and shared-memory approaches are limited by the CPU power and RAM size of a single machine. Similarly for algorithms running on the GPU. External memory algorithms often have long running times due to mostly sequential computations and limited I/O bandwidth. Distributed memory algorithms overcome these limitations in scalability by utilizing multiple compute nodes connected over a network. Here, the bottleneck lies in the inevitable *communication-overhead* to coordinate the compute nodes. Communication-overhead slows down distributed algorithm for smaller input sizes in comparison with sequential or shared-memory algorithms. Since the RAM of each compute node is limited as well, *memory-efficiency* plays an important role in designing scalable

[1] `https://archiveprogram.github.com/arctic-vault/`, last accessed 2025-01-26.

[2] `https://commoncrawl.org/blog/december-2024-crawl-archive-now-available`, last accessed 2025-01-26.

algorithms. Halving the memory peak of a distributed algorithm means that inputs of double the size can be processed utilizing the same amount of resources. Current state-of-the-art implementations of distributed suffix array algorithms require around 30×-60× the input size as working space [31, 32].

Difference Cover modulo X (DCX) is a recursive linear time algorithm based on *difference cover samples*, a sample of suffixes chosen according to a special combinatorial structure called *difference covers*. It recursively computes the suffix array of the samples and uses their ranks to efficiently sort all suffixes. The X in the algorithms name refers to the size of the congruence class ring underlying the difference cover. Part of the algorithm requires an efficient string sorting routine for fixed-length substrings of size X. In the distributed setting, prefixes of length X or simply *X-prefixes* of all suffixes have to be *materialized*, i.e. converted to an uncompressed format, to be able to exchange them between the processing elements (PEs). Using differences covers with larger X, increases the amount of work for the sorting and merging routines, but decreases the text size in the recursive calls.

1.1. Contribution

In this thesis, we develop a distributed suffix array construction algorithm. It builds on the recursive linear time algorithm DCX [46] and generalizes the distributed DC3 implementation of Kulla and Sanders [52] to difference covers with larger X.

Furthermore, we implemented a variety of optimizations to make DCX faster and more space-efficient in distributed memory. We adapt the *discarding* mechanism from an external prefix doubling algorithm [25] to discard suffix that are not required anymore in subsequent recursions. To reduce the memory peak caused by materializing X-prefixes, we implement the methods proposed for space-efficient sorting [55, 68], which we call *bucketing*. In this technique, the strings are partitioned into buckets, which are materialized and sorted one at a time. We combine bucketing with a new randomized *chunking* scheme for load-balancing overlapping strings in compressed format. Further, we evaluate comparison-based and string sorting algorithms for sorting X-prefixes in the distributed setting.

Finally, we perform an extensive evaluation of our algorithm and compare our best configuration of DCX with the current state-of-the-art distributed suffix array algorithm PSAC [32]. In our experiments, we show that our algorithm exhibits good scaling behavior up to 6144 PEs (128 compute nodes). It outperforms our competitor on 3 out of 4 real world inputs for more than 768 PEs (16 compute nodes) with speedups up between 2.2× and 3.2× (for 6144 PEs), while being competitive on a DNA dataset (1.5× slower for 768 PEs, 1.15× slower for 6144 PEs). Moreover, we are able to process inputs 3× as large as the other algorithms.

1.2. Outline

This thesis is structured as follows. In Chapter 2, we introduce relevant definitions and notations used throughout this thesis and discuss related work on suffix sorting in Chapter 3. Chapter 4 lays the necessary foundation on comparison-based and string

sorting that form core building blocks of our DCX implementation. Our main work is presented in Chapter 5, explaining DCX in-depth and algorithmic improvements thereof. Then, we present our experimental results in Chapter 6. Finally, we conclude this thesis and point out possible future work in Chapter 7.

2. Preliminaries

In this chapter, we introduce basic definitions and notations in Section 2.1. Section 2.2 specifies the parallel distributed memory model and collective communication operations we are using.

2.1. Definitions and Notations

The input to our algorithms is a text T consisting of n characters on a finite alphabet Σ of size σ. To refer to the i-th character, we use $T[i]$ for $0 \leq i < n$. By $T[i, j]$ we denote the text in the closed interval of the i-th and j-th character for $i \leq j$. Analogously, $T[i, j)$ is the text in the half-open interval of i and j. We call $T[0, k)$ the *k-prefix* of T for $0 \leq j < n$. The i-th suffix of T is denoted by $S_i = T[i, n-1]$. We assume that $T[n-1]$ is a sentinel character $\$ \notin \Sigma$ with $\$ < z$ for all $z \in \Sigma$. By this assumption, the suffixes of the text are prefix free, meaning that no suffix is a prefix of another suffix.

The *suffix array* SA stores the lexicographic ordering of all suffixes of T. In particular, $\text{SA}[i]$ is the index of the i-th smallest suffix of T. Given a sequence of elements U on a total order, we define the *rank* of an element $x \in U$ as $rank(x) = |\{y \mid y < x, y \in U\}|$. In a sorted sequence of unique elements, $rank(x)$ is the position of x in the sorted sequence. The *inverse suffix array* ISA of T contains the ranks of each suffix in a lexicographic ordering, i.e. $\text{ISA}[\text{SA}[i]] = i$. We denote a set of strings by $\mathcal{S}$ and the distinguishing prefix size of $\mathcal{S}$ by $D(\mathcal{S})$ or just D, i.e. the total number of characters that have to be inspected in order to establish the lexicographic ordering of $\mathcal{S}$. Given two strings s_1 and s_2, let $\text{LCP}(s_1, s_2)$ denote their *longest common prefix*. For a sorted set of strings $\mathcal{S}$, we define the LCP-array as $\mathcal{H}(\mathcal{S}) = [\perp, h_1, h_2, \ldots h_{|\mathcal{S}|-1}]$, where $h_i = \text{LCP}(s_{i-1}, s_i)$ for $1 \leq i \leq |\mathcal{S}|$, and the sum of LCPs by $\mathcal{L}(\mathcal{S}) = \sum_{i=1}^{|\mathcal{S}|-1} h_i$. We denote the concatenation of two strings s_1 and s_2 by $s_1 \odot s_2$. Figure 2.1 shows the suffix array, the LCP-array and the distinguishing prefixes of the text $T = \texttt{banana\$}$

In our distributed setting, we number the processing elements (PEs) from 0 to $p-1$. Each PE receives a local subarray T_i of the input text T, such that the concatenation of T_i is equal to T and the input is well-balanced, i.e. $T_i \in \Theta(n/p)$. Our suffix array algorithm is recursive in nature. For brevity, we refer to the recursion level simply as *level*. The initial recursive call starts at level 0.

2.2. Model of Computation

A common abstraction of communication in distributed memory algorithms is the *single-ported message passing model*. In this model, a distributed memory machine consists of p

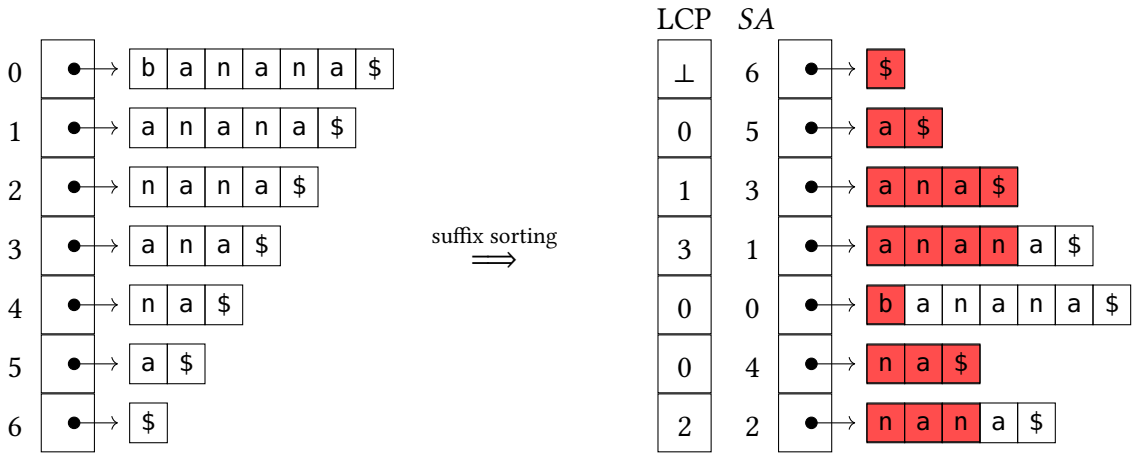


Figure 2.1.: Suffix array of the text T = `banana$` with LCP-array. The distinguishing prefixes are colored in red.

random access machines, called *processing elements* (PEs), connected by a communication network. Each PE executes the same program, has a unique rank $i \in [0, p)$ and knows the total number of PEs p. The PEs communicate by exchanging *point-to-point* messages. However, each PE can only send and receive at most one message at the same time. The cost of communication is assumed to follow a *linear model* [33], where sending a message of m machine words takes $\alpha + \beta m$ time. Here, α defines the *message startup overhead* and β the *time to communicate* one machine word with $\alpha \gg \beta$. For simplicity, we assume that a machine word corresponds to one data element.

Collective operations are high-level communication primitives between all PEs based on low-level point-to-point message exchanges. They are important building blocks for distributed algorithms and are well studied from a theoretical and practical perspective. In the following, we give a brief overview of the most common collective operations.

Broadcast. A single PE – called the *root* – replicates the same message m to all PEs. This primitive is often combined with other collective operations to share the state of a variable or the final result of a computation with all PEs.

Gather. The *gather* operations allows to collect messages m_i from each PE i on the root. After completion, the root received a vector of messages sorted by the PE number i. Gather enables the root to perform local computations using values from all PEs. Following it up with broadcast, it shares a set of messages m_i with all PEs, which is called *allgather*. The inverse operation of gather is *scatter*. Given a set of messages m_i on the root, scatter sends message m_i to the i-th PE.

Reduction. A *reduction* computes the result of $\bigoplus_{i=0}^{p-1} m_i$ for an associative operator $\oplus$ and message m_i from PE i. Usually, the elements are either integers or vectors of integers and operators are addition, multiplication, minimum, maximum, logical AND or logical OR. A

gather operation is equivalent to a reduction with the concatenation operator. Combining a reduction with a broadcast is called *allreduce*.

Prefix-Sum. Given messages m_i for each PE i and an associative operator $\oplus$, the *prefix-sum* or *scan* calculates the result $\bigoplus_{i=0}^{j} m_i$ on PE j. When leaving out the element m_j in the sum, the operation is called *exclusive* prefix-sum.

Alltoall. The *alltoall* operation is used for simultaneous data exchange between all PEs. Each PE i sends message $m_{i,j}$ to PE j. It can be thought of as the transposition of a matrix, where the i-th row consists of the messages to be sent from PE i and the j-th column of the messages that PE j should receive.

Complexity of Collective Operations. Broadcast, gather, reduction and prefix-sum can be implemented in optimal communication time $O(\alpha \log p + \beta l)$, where l is the number of messages involved in the operation. A lower bound for latency in broadcast is $\Omega(\alpha \log p)$, since the number of PEs that receive m can at most double with every round of communication. Similar arguments can be made for the other operations. The lower bound on communication $\Omega(\beta l)$ follows from the restriction, that only a single message can be send and received by a PE at the same time. Two-tree algorithms implement these collective operations combining two pipelined binary trees to better use the available bandwidth while achieving optimal communication time [85].

To implement the alltoall operation with p equal sized messages of length l, the *1-factor algorithm* [84, p. 414] can be used. It requires $O(\alpha p + \beta p l)$ time and performs $p - 1$ rounds of direct message exchanges. In case of small message sizes l, an algorithm based on hypercube communication may perform better [84, p. 415]. By communicating data in only $\log p$ rounds, it has a lower latency, but uses a higher communication volume and has complexity of $O(\alpha \log p + \beta p l \log p)$. In particular in suffix sorting, we oftentimes require *irregular* alltoall exchanges, where a message $m_{i,j}$ can have arbitrary length $|m_{i,j}|$. Performing two successive uniform alltoall exchanges, the irregular alltoall operation can be realized using the *two-phase algorithm* [84, p. 417]. Each message $m_{i,j}$ is split into p equally sized messages $m_{i,j}^k$ for $k \in [0, p)$, which are send indirectly via PE k and reconstructed on PE j. Overall, the two-phase algorithm requires $O(\alpha p + \beta h)$ time, where $h = \max_k \{\sum_i n_{i,j}, \sum_j n_{i,j}\}$ is the so-called *bottleneck communication volume*.

3. Related Work

There exists plenty of work on suffix array construction algorithms in the sequential setting [9, 12, 30, 42, 46, 81], shared-memory [56, 57, 79], external-memory [25, 43, 46] and to a somewhat lesser extend in *GPU* [20, 26] and in distributed memory settings [31, 32, 52] . However, algorithms in the extended settings usually have a sequential counterpart. For the scope of this thesis, we will focus on the most important sequential (Section 3.1) and distributed algorithms (Section 3.2). A more comprehensive overview can be found in most recent surveys [12, 14, 81]. See Figure 3.1 for a timeline of sequential suffix array algorithms.

3.1. Sequential Suffix Sorting

Despite the large number of algorithms, they can be categorized into *three* basic sorting principles: *prefix doubling*, *induced copying* and *recursion*.

Prefix Doubling. The original suffix array construction algorithm by Manber and Myers [62] is based on prefix doubling [47]. All prefix doubling algorithms share a common core [31] that we describe in the following . Let T be a text of size n.

1. Set $k = 0$ and for each suffix S_i create a rank tuple $\langle i, r \rangle$, where r is the rank in the sorting of suffixes by their first character $T[i]$.

2. If the ranks are unique, sort the rank tuples by r. Now, the first component corresponds to the SA of T. Otherwise, continue.

3. Construct rank triples $\langle i, r, r' \rangle$ based on the rank tuple $\langle i, r \rangle$ and the rank r' of the tuple with index $i + 2^k$ (or 0 if $i + 2^k \geq n$).

4. Determine new rank tuples by sorting the rank triples by $\langle r, r' \rangle$. Increase k by one and continue with Step 2.

Prefix doubling algorithms successively determine the ranks of $T[i, i + 2^k)$ by reusing the rank information from the previous iteration, instead of applying string sorting directly. Once all ranks are unique, the SA can be extracted from the indices i in the rank tuples. Each sorting step takes $O(n)$ with integer sorting and at most $O(\log n)$ times prefixes have to be doubled, resulting in $O(n \log n)$ complexity. The fastest currently know suffix array construction algorithm in distributed memory [32] is based on prefix doubling. In Section 3.2, we give a detailed description of the algorithm.

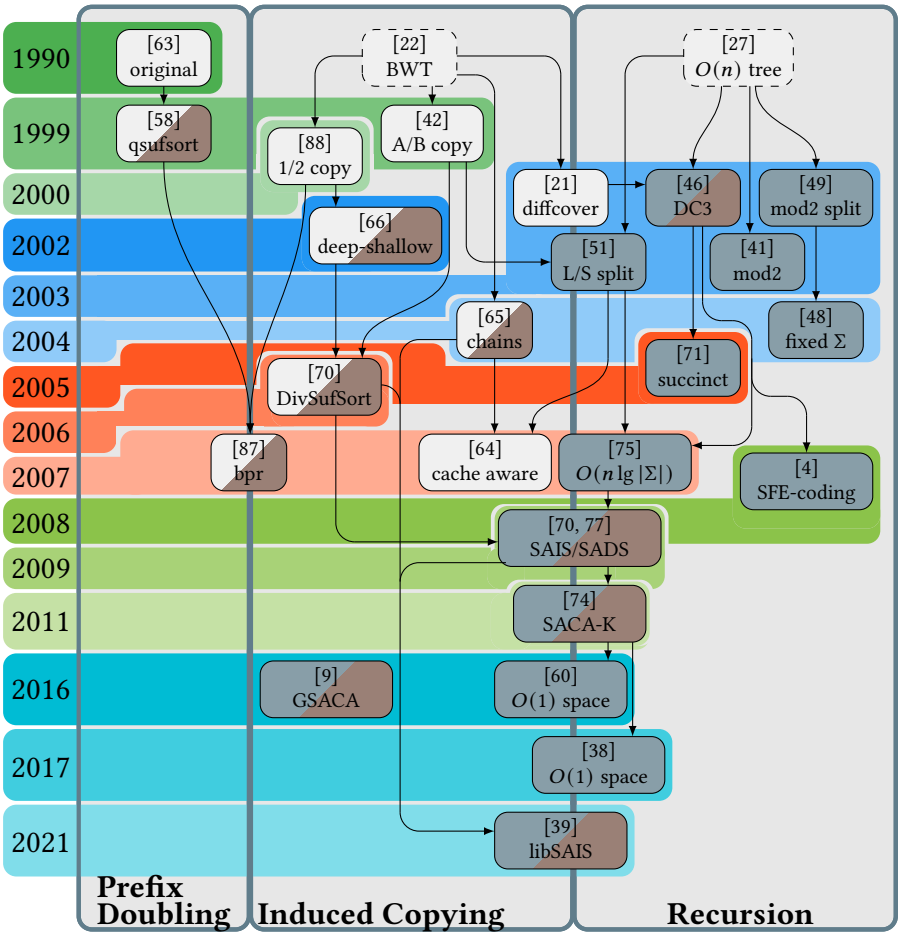


Figure 3.1.: Timeline of *sequential* suffix array construction with algorithms that share techniques are marked with an arrow. Figure based on [12, 54, 81]. The three techniques are shown as columns and algorithms that combine multiple techniques are crossing the borders. Suffix array construction algorithms with linear running time are highlighted in dark gray. If an implementation is publicly available, the algorithm is also marked in brown.

Induced-Copying. Induced-copying algorithms operate by sorting a small subset of suffixes and then use this subset to *induce* the order of the remaining suffixes. All suffixes are classified by using one of two classification schemes [42, 77], with those requiring manual sorting placed into a special class. Then, the order of all non-special suffixes can be induced based on their class, their first character, preceding and succeeding special class suffixes. The induction process typically involves only two passes over the text, with each pass requiring the comparison of just one or two characters at each position. Induced-copying algorithms are often combined with a recursive approach, achieving linear time for the construction of the suffix array and requiring only constant working space in addition to the space for the suffix array [38, 60]. The currently fastest sequential and shared-memory implementation of a suffix array algorithm LIBSAIS [44, 76, 90, 93] is based on the inducing principle.

Recursive Algorithms. The last principle is the well know divide-and-conquer method, in which subproblems of decreasing sizes are solved using recursion. Two strings x and y are formed from the input text T, such that if SA_x is computed, SA_y can be constructed efficiently and finally, SA can be computed from both suffix arrays in linear time. The string x consists of substrings of T that can be sorted with integer sorting (linear time) to determine their ranks. If not all ranks are unique, the algorithm continues recursively on x until all ranks are unique. SA_y can be inferred using the rank information from SA_x and SA is usually constructed using some merging routine. Kärkkäinen et al. [46] proposed the first linear time suffix array construction algorithm using purely recursion. This algorithm forms the basis for our distributed memory version presented in this thesis. It already has been considered in distributed memory [11, 16, 52], but there is still more room for engineering as we will discuss in Section 5.

3.2. Distributed Suffix Sorting

In this section, we explain related work on the DCX algorithm in distributed memory (Section 3.2.1) and current state-of-the-art distributed suffix array algorithms, with which we will compare our algorithm in the experimental evaluation (Section 6). The first algorithm PSAC is a prefix doubling algorithm consisting of two variants (Section 3.2.2 and Section 3.2.3). Then, we present another prefix doubling algorithm that uses a discarding scheme we also include in DCX (Section 3.2.4). The last algorithm is a distributed version of the sequential suffix array algorithm DivSufSort based on the inducing principle (Section 3.2.5).

3.2.1. Difference Cover Modulo X

There is already some work on the Difference Cover Modulo X (DCX) algorithm in the distributed setting. Kulla and Sanders showed the scalability of the DC3 in distributed memory [52]. Bingmann implemented a distributed version of DC3, DC7 and DC13[1], which however is restricted to inputs up to 4 GB due to the use of 32 bit integers to address memory. Metwally et al. implemented DC3 for the AWS and Azure cloud[2] [69] and compared an optimized version of DC3 to another distributed algorithm Futamura-Aluru-Kurtz (FAK) [34]. Their experiments indicate that FAK performs better in practice, although DC3 has better theoretical guarantees. To the best of our knowledge, the only distributed DCX implementation that considers larger values of X (up to 133), is part of the recent indexing and search system FEMTO [28].

3.2.2. PSAC Using Global Sorting

PSAC is currenlty the fastest suffix array construction algorithm in distributed memory [32]. As such, it is the main competitor in our experimental evaluation of our algorithm. PSAC is able to optionally compute the LCP-array alongside the suffix array at the cost

[1] `https://github.com/bingmann/pDCX/`

[2] `https://github.com/aametwally/cloudSACA`

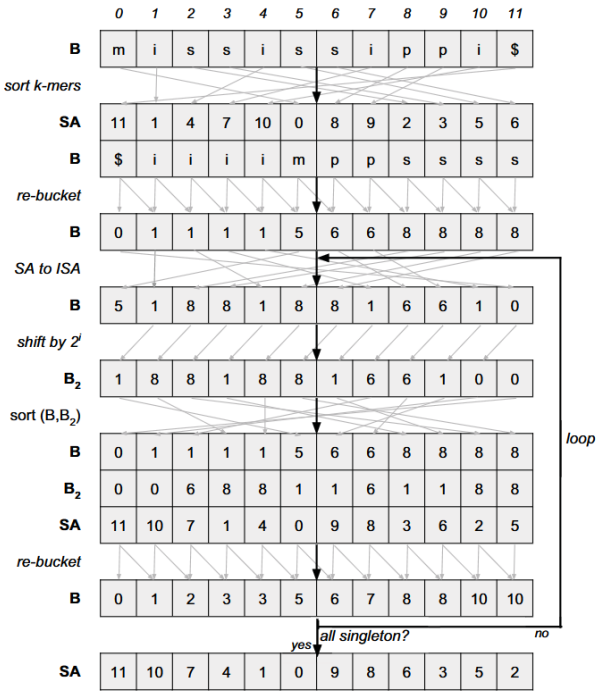


Figure 3.2.: Example execution of PSAC1 from original paper [32].

of higher running time, but we are only interested in the suffix array construction part. Internally, PSAC uses two versions of prefix doubling, which we call PSAC1 and PSAC2. In this part, we give a detailed description of the first and in the following Section 3.2.3 of the second version.

PSAC1 follows the prefix doubling sorting principle, successively sorting the suffixes by $T[i, i+h)$, which the authors call *h-prefix*, doubling h in each step. Suffixes sorted by their h-prefix are in *h-order* and elements with the same rank are said to be in the same *h-group*. The authors use three arrays SA_h, B and B_2 to organize their data. SA_h keeps track of the global indices of suffixes in h-order. If all suffix ranks are unique, SA_h contains the suffix array of the input text. In an h-order, the array B is aligned with SA_h and stores the ranks of the h-groups by $B[i] = i$, for the first position of a group and $B[j] = i$, for all elements of that same group. B_2 contains the shifted ranks required for constructing the rank triples in Step 3, i.e. $B_2[i] = B[i+h]$ (or 0, if $i + h \geq n$).

During the algorithm, all data including input, output and working data is distributed equally among all PEs. Therefore, each PE has $\lfloor \frac{n}{p} \rfloor$ or $\lceil \frac{n}{p} \rceil$ elements. For simplicity, we assume that p divides n. The authors use their own parallel implementation of sample sort with regular sampling for all sorting routines. Figure 3.2 shows an example execution of PSAC1.

Termination. As in Step 2 of prefix doubling, the algorithm terminates once all suffix ranks in B are unique. Sorting $\langle SA_h[i], B[i] \rangle$ by the second component yields the SA in the first component.

Tuple Sorting. Analogously to Step 1 and Step 3 in prefix doubling, rank tuples $\langle i, B[i]\rangle$ are sorted by $B[i]$ and rank triples $\langle i, B[i], B_2[i]\rangle$ are sorted by $\langle B[i], B_2[i]\rangle$. The separate arrays SA_h, B and B_2 (or SA_h and B) are zipped into a vector of triples (or tuples) before sorting and afterwards are unzipped again into separate vectors.

Initialization. In Step 1 of prefix doubling, instead of initializing the ranks with the 1-order, the algorithm computes the k-order by sorting, skipping several prefix doubling iterations. The elements to be sorted are called *k-mers* and are the k-prefixes of all suffixes. They choose k depending on the alphabet size σ and the number of bits in a machine word l by

$$k = \left\lfloor \frac{l}{\log_2(\sigma)} \right\rfloor$$

to be able to pack the k-prefix into a single machine word. The ranks of the k-mers are computed in the *rebucketing* step.

Rebucketing. After determining the h-order of the suffixes by sorting (Step 1 and Step 4 of prefix doubling), the array B has to be filled with the corresponding ranks. In a first pass, the start of an h-group is set to its own index i whenever $B[i-1] \neq B[i]$ or $B_2[i-1] \neq B_2[i]$, otherwise to 0. In the initialization step, only B is considered in the comparison. Now, the first elements of each h-group have the correct rank. Then, a prefix-scan with the max operation is performed to set the remaining ranks.

SA to ISA. In the next step, the array B is brought back into text-order to align the ranks of each suffix with its position. By the data distribution, it is easy to determine the target processor of the element with index i by $\lfloor i\frac{p}{n} \rfloor$. Elements are bucketed according to their target processor and send with the alltoall collective. Then, the text-order is achieved by locally reassigning the elements according to their SA_h index.

Shifting. The array B_2 is obtained by shifting B by h positions. Each PE has at most two PEs, from which it has to send data, and two from, which it has to receive data. This is a straight forward communication pattern and is realized by two point-to-point communications.

Complexity. Since the algorithm performs $O(\log n)$ rounds and uses only scans and sorting routines in each step, the overall complexity is given by $O(T_{sort}(n, p) \log n)$, where $T_{sort}(n, p)$ is the time complexity of the underlying distributed sorting algorithm on n elements using p PEs.

Memory Consumption. The highest main memory is required when the triple vector consisting of entries of SA_h, B and B_2 is sorted. To store the triple vector, 3 words are required for each position. Additionally, for alltoall MPI communication, receive buffers of the same size have to be allocated. Therefore, in total the algorithm requires 6 words for each of the $\frac{n}{p}$ entries on a PE. Assuming a words size of 8 byte, the algorithm requires at least 48 bytes additional per input character.

3.2.3. PSAC With Avoiding Global Sorting

On many real world inputs, a large fraction of h-groups become unique (singleton) after only few iterations. A significant overhead can be avoided by only sorting the non-singleton h-groups instead of globally sorting all together. To improve on PSAC1, the authors propose an other prefix doubling version PSAC2. Since PSAC2 has a higher memory usage due to two additional array, they still use PSAC1 for the initial k-mer sort and the first few iterations. Once the number of non-singleton groups is smaller than εn, for some tuning parameter ε, they switch to PSAC2, which we describe in the following.

To organize the data, the authors use five arrays, SA_h, B, ISA_h W and M distributed in the same block layout as before. SA_h and B have the same meaning as in PSAC1. $ISA_h[i]$ contains the current h-group rank of the suffix S_i. The array W keeps track of the start indices of each non-singleton group and M holds memory for tuples that are used to exchange ranks of the h-order. PSAC2 receives SA_h, B, ISA_h for the current prefix length h as input from PSAC1.

Determine non-singleton h-groups. First, the algorithm determines all positions of non-singleton h-groups using the local property of B. Either an element is not the representative of its h-group, $B[i] \neq i$, or the element is a representative with at least one more element in the same group, $B[i+1] = i$. This check can be performed in a single local scan. Each PE j with $j+1 < p$ receives the first element of PE $j+1$ via point-to-point communication to be able to check the second condition for the last element as well.

Exchange Shifted Ranks. In order to apply prefix doubling to a non-singleton h-group with global indices G, we require the rank of the corresponding suffix shifted by h. An element $i \in G$ corresponds to the suffix with index $SA_h[i]$, thus the index of the shifted suffix is at position $SA_h[i] + h$. The rank information is contained in $ISA_h[SA_h[i] + h]$. This location might not be available locally and communication is necessary. Instead of sending separate messages, the algorithm exchanges the requested ranks using two alltoall calls. For the first call, requests in the form of $\langle SA_h[i] + h, i\rangle$ for each non-singleton position $i \in W$ are stored in the array M. These requests are bucketed by their target PE $\left\lfloor \frac{p}{n}(SA_h[i] + h) \right\rfloor$ and communicated using an alltoall call. Now, each PE overwrites the first values in the received tuples $\langle i, j\rangle$ with $\langle ISA_h[i], j\rangle$, which corresponds to $B_2[i]$ in PSAC1. Finally, the second alltoall call sends the data back to its origin.

Bucket Sorting. After receiving the rank information in the array M, each non-singleton h-group has to be sorted by $ISA_h[i]$. However, an h-group might span more than one processor. To handle overlapping h-groups, first, all h-groups, which are local on a PE are sorted. Then, each PE with overlapping h-groups participates in a parallel sort with one or both of its neighbors, inducing the $2h$-order of the suffix array. Here, the authors split the MPI communicator into a sub-communicator for each parallel sorting routine. The ranks of the $2h$-order are determined as before with the rebucketing routine. During rebucketing, a new array W_{new} replaces W, containing those indices of W, which remain non-singleton.

Update ISA. Finally, the ISA_h must be updated with the newly determined $2h$-group ranks. Similarly to the exchange of shifted ranks, the array M is reused to send tuples $\langle SA_h[i], B[i]\rangle$ for each $i \in W$ to the target PE $\left\lfloor \frac{p}{n} SA_h[i] \right\rfloor$ with another alltoall communication. The receiving PE now updates the local ISA_h by $ISA_h[SA_h[i]] = B[i]$.

3.2.4. Distributed Prefix Doubling with Discarding

Fischer and Kurpicz [31] proposed another distributed prefix doubling algorithm adapting a discarding mechanism introduced by Dementiev et al. [25]. There are two main difference to PSAC1. First, to exchange the shifted ranks (Step 3 of prefix doubling), they sort the rank tuples such that required ranks are next to each other (if they exits). Sorting the global indices i by (i **mod** h, i **div** h) yields the desired property. Secondly, this allows to ignore rank tuples that are not required anymore, since required ranks are not spatially separated. A suffix can be discarded if it is unique and is not required to determine ranks of non-unique suffixes. The second condition occurs if the left neighbor of the suffix is unique as well. In other words, for each sequence of adjacent unique ranks, we only have to keep the first element of that sequence. Discarding can be implemented with a single scan and point-to-point communication between adjacent PEs to handle missing elements at the edges. To avoid interference of smaller discarded suffix with the update of ranks, the authors always set a suffix to the highest possible rank it could potentially obtain. This discarding strategy significantly reduces the overhead of globally sorting all rank tuples on all PEs. Since there is no theoretical difference to PSAC1, prefix doubling with discarding has the same complexity of $O(T_{sort}(n,p)\log n)$.

3.2.5. Distributed DivSufSort

In the same paper [31], Fischer and Kurpicz also present a distributed variant of DivSufSort [30, 70]. We will state lemmas without proof and refer to the paper for more details. First, we introduce some notation used by the authors. Given a text T of size n, let T' denote the consecutive slice of size $n' = \Theta(\frac{n}{p})$ local on a PE. S'_j denotes the j-th suffix in T' with respect to the whole text.

Classification of Suffixes. The authors use a classification scheme with two *classes* originally introduced by Itoh and Tanaka [42].

$$C^- = \{i \in [0,n) \mid S_i > S_{i+1}\}$$
$$C^+ = \{i \in [0,n) \mid S_i < S_{i+1}\}$$

A suffix S_i is represented by its starting position i in the text T. We say "a suffix S_i is in C" if $i \in C$ for some class C. Consecutive suffixes that differ in their class form *sub-classes* and are later used to identify fine-grained intervals in the SA. We are interested in the suffix *before* the change.

$$C^{-\triangleright} = \{i \in C^- \mid i+1 \in C^+\} \cup \{n-1\}$$
$$C^{+\triangleright} = \{i \in C^+ \mid i+1 \in C^-\}$$

Notice that both sub-classes have at most $\frac{n}{2}$ elements. Suffixes that are followed by the same class are also required and are denoted by:

$$C^{-\otimes} = C^- \setminus C^{-\triangleright} \quad \text{and} \quad C^{+\otimes} = C^+ \setminus C^{+\triangleright}.$$

Further, it will be necessary to filter a (sub-)class C by the first one or two characters. Let $\alpha, \beta \in \Sigma$, then:

$$C_\alpha = \{i \in C \mid T[i] = \alpha\}$$
$$C_{\alpha\beta} = \{i \in C_\alpha \mid T[i+1] = \beta\}$$

Let $\overrightarrow{C}$ denote the starting positions of a class C in lexicographic order. The key element of the algorithm is that the suffix array can be expressed using the introduced sub-classes.

Lemma 1 (Observation 1. in [31])**.** We can express the SA as follows:

$$SA = \overrightarrow{C^{-\otimes}_{00}}\,\overrightarrow{C^{-\triangleright}_{00}}\,\overrightarrow{C^{+\triangleright}_{00}}\,\overrightarrow{C^{+\otimes}_{00}}\,\overrightarrow{C^{-\otimes}_{01}} \ldots \overrightarrow{C^{-\otimes}_{\sigma-1\sigma-1}}\,\overrightarrow{C^{-\triangleright}_{\sigma-1\sigma-1}}\,\overrightarrow{C^{+\triangleright}_{\sigma-1\sigma-1}}\,\overrightarrow{C^{+\otimes}_{\sigma-1\sigma-1}}$$

General Overview. Using this classification, the suffix array can be computed in three steps.

1. Compute $C^{+\triangleright}$ and the sizes of $C_{\alpha\beta}$ for all $\alpha, \beta \in \Sigma$ and (sub-)classes C locally for T'. Aggregate the results over all PEs to get the sizes for T.
2. Sort $C^{+\triangleright}$ lexicographically using a distributed string sorting algorithm to compute $\overrightarrow{C^{+\triangleright}}$.
3. Induce all other suffixes using only $\overrightarrow{C^{+\triangleright}}$ and T, without any sorting.

Lemma 2 (Observation 2. in [31])**.** For all $i \in [1, n)$

a) $i-1 \in C^{+\otimes} \Leftrightarrow i \in C^+$ *and* $T[i-1] \leq T[i]$,

b) $i-1 \in C^{+\triangleright} \Leftrightarrow$ *either* $i = n$ *or* $i \in C^+$ *and* $T[i-1] > T[i]$,

c) $i-1 \in C^{-\otimes} \Leftrightarrow i \in C^-$ *and* $T[i-1] \geq T[i]$,

d) $i-1 \in C^{+\triangleright} \Leftrightarrow i \in C^-$ *and* $T[i-1] < T[i]$.

Lemma 3 (Observation 3. in [31])**.** Let $i \in [0, n-1)$. We know that $n-1 \in C^-$. If $T[i] > T[i+1]$, then $i \in C^-$ and if $T[i] < T[i+1]$, then $i \in C^+$. Last, if $T[i] = T[i+1]$ then $i \in C^- \Leftrightarrow i+1 \in C^-$.

Step 1: Identifying Suffixes in $C^{-\triangleright}$. To determine the (sub-)class of a suffix S_i in Step 1, it is sufficient to examine the class C^+ or C^- of S_{i-1} and the characters $T[i-1]$ and $T[i]$ (see Lemma 2). Using Lemma 3, the class membership of C^+ and C^- for each suffix can be determined.

In the sequential setting, this step can be implemented with a simple right to left scan. However, in the distributed setting we cannot simply scan T' from right to left, since the class of the last suffix in T' in only know for PE $p-1$. To resolve this problem, the authors determine the classes in two right to left scans and one broadcast operation per PE. In the first scan, the first suffix is determined that satisfies $T[i] > T[i+1]$ for $i < n'-1$. We can conclude that this suffix is in C^- without the knowledge of the class of its right neighbor. Using Lemma 3, all suffixes S'_j with $j < i$ can be classified. Now, each PE communicates the class of S'_0 to all other PEs, if the class is known, otherwise it communicates *unknown*. The latter case occurs if no suffix in T' can be determined definitely, i.e. all characters in T' are the same. To determine the classes of the remaining suffix, the class of the right neighboring suffix S'_0 can be used, if it is known. If not, the first PE j with $j > i$, on which the class of the first suffix is know, is selected. With the class information of j and the first character of the first suffix, the remaining types can be concluded using the last part of Lemma 2. Notice that in this case for all k with $i < k < j$, the local texts T' consists of a single character. The authors claim that such cases do not occur in practice. Alongside the two scans, $C^{+\triangleright}$ and the sizes of the other sub-classes can be determined without an overhead in running time.

Step 2: Sorting Suffixes in $C^{+\triangleright}$. For sorting the suffixes in $C^{+\triangleright}$, we require the substrings between two adjacent elements of $C^{+\triangleright}$ in text-order. Given $i \in C^{+\triangleright}$ the next adjacent element is $next(i) = \min\{j > i \mid j \in C^{+\triangleright} \cup \{n\}\}$. A *$C^{+\triangleright}$-ending substring* is defined as $T_i^{+\triangleright} = T[i, \min\{next(i)+2, n\})$. The two additional characters are necessary to correctly sort $C^{+\triangleright}$. Now, all $C^{+\triangleright}$-ending substring are sorted using distributed string sorting. In their implementation, the authors employ sample sort and use multikey radix sort to locally sort the strings. Computing $\overrightarrow{C^{+\triangleright}}$ can be viewed as another suffix array construction problem on the ranks of the $C^{+\triangleright}$-ending substring. For this purpose, the prefix doubling algorithm described earlier in Section 3.2.4 is used. The algorithm relies on indices in the range from 0 to m, where m is text length. Thus, before prefix doubling they transform the ranks to the correct range. Finally, to obtain $\overrightarrow{C^{+\triangleright}}$ they sort the rank tuples $\langle i, r\rangle$ by r, where $i \in C^{+\triangleright}$ and r is the corresponding entry in the computed suffix array. The first component now contains $\overrightarrow{C^{+\triangleright}}$.

Step 3: Inducing the Suffix Array. Algorithm 1 summarizes the inducing step. All PEs work on their own slice of the sub-classes. However, the concatenation $\odot$ in Line 3 and Line 14 apply to the global array. The global text T has to be accessed by all PEs. Therefore, in Line 8 and Line 17 the first characters necessary in the next iteration are communicated between the PEs. To facilitate localizing the required characters, the author distribute the text T such that $T[i\lceil\frac{n}{p}\rceil, \min\{(i+1)\lceil\frac{n}{p}\rceil, n\})$ is located on PE. Now, the i-th character can be easily localized using division and modulo computations.

Algorithm 1: Inducing Step DivSufSort

```
1  for α = σ − 1 down to 0 do
2      for β = σ − 1 down to α do                          // α > β ⇒ C^+_αβ = ∅
3          for i ∈ C^{+▷}_αβ ⊙ C^{+⊗}_αβ in reverse order do
4              if i > 0 and T[i − 1] ≤ α then               // Lemma 2 a)
5                  C^{+⊗}_{T[i−1]α}.pushfront(i − 1)
6              else if i > 0 then                           // Lemma 2 b)
7                  C^{+▷}_{T[i−1]α}.pushfront(i − 1)
8          communicate()                                    // T[i − 1] for next step
9
10 C^{−⊗}_{T[n−1]0}.pushback(n − 1)                         // handle last suffix
11
12 for α = 0 to σ − 1 do
13     for β = 0 to α do                                    // α < β ⇒ C^−_αβ = ∅
14         for i ∈ C^{−⊗}_αβ ⊙ C^{−▷}_αβ in reverse order do
15             if i > 0 and T[i − 1] ≥ α then               // Lemma 2 c)
16                 C^{−⊗}_{T[i−1]α}.pushback(i − 1)
17         communicate()                                    // T[i − 1] for next step
18
```

First, $\overrightarrow{C^{+\otimes}}$ and $\overrightarrow{C^{-\triangleright}}$ are induced in a right to left scan in reverse lexicographic order (see loop starting at Line 1). Not all sub-classes have to be traversed, since they are empty by definition. Each filtered sub-class is filled by repeated application of Lemma 2 a) and Lemma 2 b). The last suffix is added before continuing with the second inducing step (Line 10). Similarly to the first inducing step, the sub-class $C^{-\otimes}$ can be induced in a left to right scan in lexicographic order (see loop starting at Line 12) with application of Lemma 2 c).

There is a special case that has to be handled separately. A *run* of length r is a sequence of r characters in T that are all the same. The algorithm as described above, cannot induce the class of 3-runs or longer runs. This would require to induce in the same array that is currently traversed (Line 3 and Line 14). To resolve this, the runs of the same characters are unrolled from right to left until one run ends. Using the length of each run and the SA positions, these special cases can be resolved.

Memory Consumption. Let w be the size of a word in bytes. The authors use $w = 5$ and can process text up to 1 TB. The maximum main memory is required during the sorting of the suffixes in $C^{+\triangleright}$. When new ranks are computed in the sorting, the suffixes starting position are sorted together with two ranks, requiring 3 words per considered suffix. There are at most $n/2$ suffix of class $C^{+\triangleright}$. Taking into account that we also require space for the receive buffers, the total memory required is $3wn$ bytes. Further, $2\sigma^2 wp$ bytes are required

to store the sizes of the sub-classes. With equal distribution of the data, this results in a maximum of $3wn/p + 2\sigma^2 w$ bytes per PE in addition to the input text.

4. Building Blocks

Distributed sorting routines form core building blocks in our DCX implementation. We have to repeatedly sort sequence of numbers, sequence of strings of length X and a mixture of both types of sequence. Conventional sorting assumes *atomic* elements, which means elements can be compared and swapped in constant time. However, this is not the case if we consider the lexicographic sorting of strings. Here, it is important to exploit the structure of the keys and avoid repeated comparison. Specialized *string sorting* algorithms can be used for that purpose. Since X is rather small (≤ 133) and fixed during the algorithm, *comparison-based* or *atomic sorting* can be applied as well to sort the X-length strings. We want to evaluate and compare both types sorting. In the following, we give an introduction to distributed atomic sorting in Section 4.1 and distributed string sorting in Section 4.2.1.

4.1. Distributed Atomic Sorting

There is a wide variety of distributed atomic sorting algorithms. For a more comprehensive overview of distributed sorters, we refer to [5]. The various algorithms offer a trade-off between latency and communication volume. Low latency makes them more efficient for small sized inputs, while low communication volume is more beneficial for large inputs. We briefly introduce some distributed sorting algorithms that we use as building blocks in our algorithm. Table 4.1 shows the latency and communication volume complexities of the presented algorithms. (All)-gather-merge, simply collects and sorts all data one a single PE.

Sample Sort. Sample Sort [18] consists of three phases.

1. A sorted set of $p - 1$ splitter elements is selected that partitions the keys into p buckets.
2. Each PE sends the i-th bucket to PE i.
3. The elements are sorted within each bucket. Alternatively, one can also use a k-way merging procedure.

To select the splitter elements from the n input elements, a sample of $ps \leq n$ elements is chosen uniformly at random. The parameter s is the so-called *oversampling ratio*. This sample is sorted, and the elements with the ranks $s, 2s, \ldots (p-1)s$ are chosen as splitters. The sorting of the samples is another distributed sorting problem, which can be handled by using a different distributed sorter or by gathering the samples on a single PE and sorting them locally. Using the latter option, the algorithm is efficient for a minimum size of $n \in \Omega(p^2/\log p)$, i.e. the isoefficiency function is $\Omega(p^2/\log p)$ [53].

Table 4.1.: Complexity of parallel sorting algorithms [7]. Implicit $O(\cdot)$. $k = \sqrt[r]{p}$, where r is the number of levels used in AMS.

Algorithm	Latency [α]	Comm. Vol. [β]
(All)-gather-merge	$\log p$	n
Sample Sort	$\geq p$	$\geq n/p$
AMS	$k \log_k p$	$\frac{n}{p} \log_k p$
Rquick	$\log^2 p$	$\frac{n}{p} \log p$

Adaptive Multi-Level Sample Sort (AMS-sort) [6, 7]. Sample sort suffers from a latency bottleneck, since every PE receives at least $p - 1$ splitters. Instead, AMS divides PEs into groups of size p', only exchanges elements between groups, and proceeds recursively in k levels. Using only $r = p/p'$ splitters, the authors improve the isoeffiency function to $\Omega(p^{1+1/k}/\log p)$. The parameter k is a trade-off between asymptotic scalability and communication overhead from data exchange operations. This approach is based on previous work of Gerbessiotis and Valiant [36], who developed a multi-level variant of sample sort in the *bulk-synchronous parallel model*. The authors of AMS improve the original idea in various ways. They developed a fast work inefficient sorting algorithm to sort the samples, advanced data delivery algorithms and a scalable adaptation of the idea of *overpartitioning* [59]. Using overpartitioning for achieving imbalance of ε, reduces sample size required for a good load-balance from $O(1/\varepsilon^2)$ to $O(1/\varepsilon)$. In this method, $kp - 1$ splitters are selected from kps samples. The resulting kp buckets are then assigned to the PEs in a load-balanced way. Given an upper bound L on the number of elements per PE, the array of buckets sizes is scanned and they skip to the next PE-group when the total load would exceed L. An optimal value of L can be determined using a binary search.

Rquick [7]. Rquick is a parallel quicksort implementation that uses the hypercube design pattern (see Algorithm 2). Such algorithms use communication in a conceptional hypercube of dimension d such that $p = 2^d$ to exchange messages between PEs. While iterating through the j-th dimensions of the hypercube, the communication partner of PE i is determined by $i \oplus 2^j$. The topology allows an efficient implementation of basic communication primitives such as *all-gather* or *all-reduce* and routing data for random start or destination nodes [24] requires only $O(\alpha \log p)$ startup overhead overall. Initially, to avoid skewed data distribution, Rquick randomly redistributes the data and locally sorts the elements. In the j-th hypercube iteration, a splitter element s is calculated using a communication efficient median approximation. Each PE locally partitions the data according to s into $S_<$ and $S_{>=}$. The communication partner with the 0-bit at the position j receives and merges the elements smaller than s and his partner the elements larger or equal than s. In addition, the authors use a low-overhead tie-breaking scheme to make the algorithm robust against repeated keys. Rquick closes the gap between very small and very large inputs, outperforming competitors on small inputs for 2^3 to 2^{14} elements per core. This can be useful, for example, when sorting a small sample of elements to determine splitter elements in sample sort.

Algorithm 2: Hypercube Algorithm Design Pattern

```
1 local computation on PE i
2 for 0 ≤ j < p do
3     send message m to PE i ⊕ 2^j
4     receive message m' from PE i ⊕ 2^j
5     perform local computation using m and m'
```

4.2. Distributed String Sorting

In our suffix sorting algorithm, (short) strings of length X have to be sorted as a subroutine. We implemented our own distributed string sorting routine, which builds on sequential string sorting (Section 4.2.1), efficient k-way merging of strings using LCP-aware Loser Trees (Section 4.2.2) as well as LCP-compression (Section 4.2.3) and distinguishing prefix approximation (Section 4.2.4), techniques to reduce the communication volume of the exchanged strings.

4.2.1. Sequential String Sorting

We give a brief introduction to the sequential string sorting algorithms we incorporated into our implementations. Bingmann [12] provides an overview of the most important sequential sorting algorithms, including Multikey Quicksort, MSD Radix Sort, Burstsort, LCP-Mergesort and Insertion Sort. We will focus on the first two, since they performed the best in Bingmann's experimental evaluation. Both algorithms can optionally compute the LCP arrays of the strings during the sorting.

Multikey Quicksort. Bentley and Sedgewick [10] adapted Quick Sort for string data. Let $\mathcal{S}$ be the set of input strings with a common prefix of h. The algorithm uses the character $x = s[h]$ of a pivot string $s \in \mathcal{S}$ to split $\mathcal{S}$ into $\mathcal{S}_<$, $\mathcal{S}_=$ and $\mathcal{S}_>$ based on comparison with $s'[h]$ for $s' \in \mathcal{S}$. Each partition is sorted recursively, with the exception of $\mathcal{S}_=$, if $x = \$$ is the terminating character. The common prefix of the strings in $\mathcal{S}_=$ is increased by one, avoiding comparing the characters found to be equal with x. In the base case, Insertion Sort is used for constant size inputs. Multikey Quicksort has an expected execution time of $\mathcal{O}(D + n \log n)$, where D is the distinguishing prefix of $\mathcal{S}$ and n the number of input strings.

MSD Radix Sort. Again, consider a set of input strings $\mathcal{S}$ with a common prefix h. Most Significant Digit (MSD) Radix Sort produces σ subproblems by partitioning $\mathcal{S}$ based on $s[h]$ for $s \in \mathcal{S}$ into $\mathcal{S}_x$ for $x \in \Sigma$, which are sorted recursively with a common prefix of $h+1$. Paige and Tarjan [80] presented the first $\mathcal{O}(D + \sigma)$ radix sort algorithm. There is much research on practical implementations of radix sort. McIlroy, Bostic, and McIlroy [67] were the first to engineer variants of radix sort and to propose concrete practical considerations. Ng and Kakehi [72] extended radix sort with a *caching* variant. To avoid cache misses, they propose to fetch z characters at once from a string, instead of a single character. The cached

chars are stored in buffer aligned with the string pointer array. Kärkkäinen and Rantala [45] presented an extensive experimental study of radix sorter variants and developed the fastest practical radix string sorters. They incorporated caching optimizations, adaptive 16-bit radix sorts, and optimizations of inner loops taking advantage modern processors' super-scalar accelerations and memory latency hiding. Various high quality string sorting algorithms are implemented by Rantala and are available in a public repository [82].

4.2.2. LCP-Merging

In distributed sample sort, each PE receives p buckets of sorted strings that need to be merged. This p-way merging procedure can be solved using *Loser Trees* [50, 83]. If additionally the LCP-values of the strings are know, Loser Trees can be enhanced with LCP-values to save character comparisons in the merging process. The so-called *LCP-aware Loser Tree* [15] adapt a 2-way LCP-aware merging procedure proposed by [73] to k-way merging of strings with LCP-values. In the following, we describe the LCP-Compare function used to defined 2-way LCP-aware merging, k-way merging with Loser Trees and their extensions to LCP-aware Loser Trees.

Binary LCP-Compare [73]. Consider the comparison of two strings s_1 and s_2. Additionally, we know their LCP-values with another string p where $p \leq s_1$ and $p \leq s_2$. We denote their LCP-values by $h_1 = \text{LCP}(p, s_1)$, $h_2 = \text{LCP}(p, s_2)$ and the output LCP of the comparison by $h' = \text{LCP}(s_1, s_2)$. There are three cases to distinguish:

1. $h_a = h_b$: Then, s_1 and s_2 share a common prefix of length $h = h_a$ and we compare the two strings character-by-character starting at the positions $(h + 1)$, saving h character comparisons. We set h' to h plus the number of extra comparisons.

2. $h_a < h_b$: Thus, s_1 and s_2 differ at position $l = h + 1$. From $p \leq s_1$ and $p[l] = s_2[l] < s_1[l]$ follows $s_2 < s_1$ without comparing additional characters. We set h' to h_a.

3. $h_a > h_b$: The same argument as in case 2 can be applied symmetrically and we get $s_1 < s_2$ and set h' to h_b.

Using binary LCP-Compare, we can define a LCP-aware 2-way merging routine. The above strings s_1 and s_2 take the role of the next candidates of the two streams and p the role of the last element that was written to the output stream. In the beginning, p is the empty string. Say s_1 is the smaller element and s_1' the next string in its corresponding stream. The LCP-values for the next comparison are given by $h_1' = \text{LCP}(s_1, s_1')$, which is know from the LCP-array, and $h_2' = h' = \text{LCP}(s_1, s_2)$.

k-way Merging with Loser Trees [50, 83]. In a Loser Tree, the next elements of the k input streams are considered as k players participating in a tournament organized in a binary tree. The winner of a game is determined by binary comparison. Each leaf in the binary tree corresponds to one of the k players and each inner node represents a game between the two children nodes. The winner advances to the parent node and the loser is stored in

the inner node. After $\log_2 k$ rounds and a total of $k-1$ games, the overall winner is the smallest element among the k elements.

In the k-way merging routine, first an initial round on all nodes is played bottom up, the winner can be written to the output stream and the next element from the corresponding input stream takes its place. Now, only the $\log_2 k$ games along the root-to-leaf path, in which the winner participated, have to be replayed. This procedure is repeated until all input streams are empty. Empty streams are replaced by sentinels larger than all the other elements, to avoid corner case. We can assume k to be a power of two, by filling up empty streams as needed. Thus, we can assume the binary tree to be perfect and represent it implicitly in an array. Determining the parent corresponds to division by two: $\lfloor i/2 \rfloor$. The root is stored at $i = 1$ (using 1-indexed arrays). Overall, to merge all n elements, $k-1$ comparison are needed for the initial round and $(n-1)\log_2 k$ for replaying the games of the remaining elements.

k-way LCP-Merging [15]. We now explain how to extent k-way merging with LCP-values to save character comparisons. Binary comparisons are performed with the LCP-Compare function. Therefore, the input streams now additionally contain the corresponding LCP-arrays to the sorted sequence of strings. Games played between leaf nodes use the LCP-values directly from the LCP-array. For inner nodes, where s_1 was the loser string in a game with s_2, we store the output LCP $h = \text{LCP}(s_1, s_2)$ alongside the loser string. The winner s_2 advances up the tree and uses the LCP-value h in the next round. When replaying games on a root-to-leaf path, the requirements for LCP-Compare are fulfilled. Since the winner string w was also the winner on each game played on the path, each loser string on that path s stores $\text{LCP}(w, s_i)$ and w takes the role of p as in the 2-way LCP-aware merging procedure. Figure 4.1 shows a LCP-aware Loser Tree with eight input streams. The following theorem bounds the maximum number of required comparisons.

Theorem 1 (Complexity of LCP-k-way-merging [15])**.** *Let $\mathcal{S}_0$ be the merged output sequence of the k input sequences $\mathcal{S}_1, \mathcal{S}_2, \ldots \mathcal{S}_k$. A LCP-aware Loser Tree needs at most $\Delta L + |\mathcal{S}_0| \log k + k$ character comparisons, where $\Delta L = \mathcal{L}(\mathcal{S}_0) - \sum_{k=1}^{K} \mathcal{L}(\mathcal{S}_k)$ is the sum of increments to LCP array entries.*

4.2.3. LCP-Compression

LCP-compression [17] is a technique to reduce the communication volume when exchanging a sorted set of strings $\mathcal{S}$. The idea is to send each common prefix of $\mathcal{S}$ only once and reconstruct the original string using the LCP-values. Let $\mathcal{H}(\mathcal{S}) = [\perp, h_1, h_2, \ldots h_{|\mathcal{S}|-1}]$ be the LCP-array of $\mathcal{S}$. The character $s_i[h_i]$ for $1 \leq i \leq |\mathcal{S}|$ is the first character, in which s_i and s_{i-1} differ. Thus, we only have to send the characters $s'_i = s_i[h_i, |s_i|)$. We set $s'_0 = s_0$, since the first string has no preceding string. The compressed set of strings of $\mathcal{S}$ is

$$\mathcal{S}' = [s'_0, s'_1, s'_2, \ldots, s'_{|\mathcal{S}|-1}].$$

We can reconstruct $\mathcal{S}$ from $\mathcal{S}'$ and $\mathcal{H}(\mathcal{S})$ in a left to right scan. The first string s_0 is the same in both strings. Every subsequent string s_i is obtained by copying the first h_i

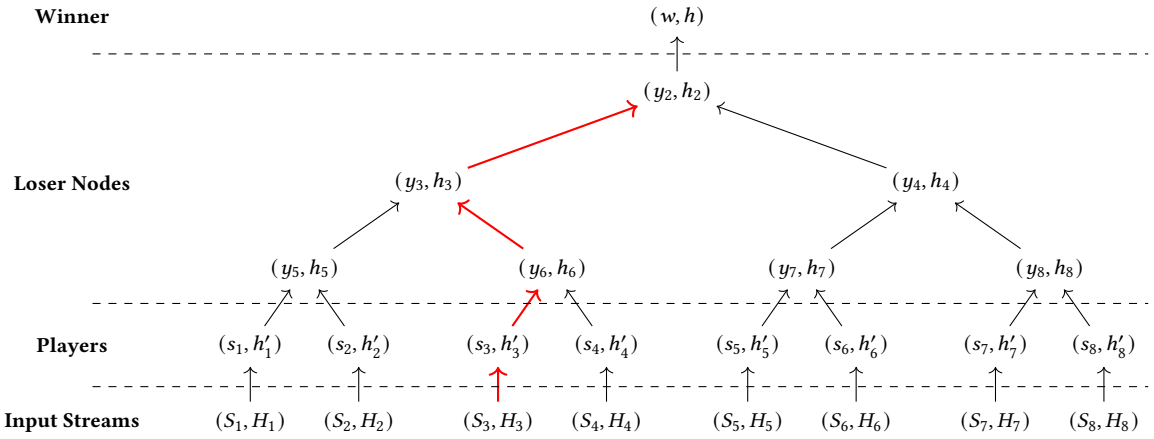


Figure 4.1.: LCP-aware Loser Tree with eight input streams.

characters of the preceding string and concatenating it with the compressed string s'_i, i.e. $s_i = s_{i-1}[0, h_i) \odot s'_i$.

Compression and decompression perform just one iteration over the data and copies each character at most once. Thus, LCP-compression takes linear time complexity. The compression sends $\sum_{i=1}^{|\mathcal{S}|-1} h_i$ less characters than a direct exchange, but requires to send the LCP-array $\mathcal{H}(\mathcal{S})$ alongside the compressed strings $\mathcal{S}'$. Additionally, there is trade-off between time spent on the compression/decompression algorithm and the time saved by the reduced communication volume.

4.2.4. Distinguishing Prefix Approximation

When sorting a set of strings $\mathcal{S}$, it is sufficient to examine the *distinguishing prefixes* of each string to determine their sorted order. Only these prefixes of the strings have to be communicated between the PEs. Determining the distinguishing prefix is equivalent to checking whether there are any duplicates of it. Apart from communicating the entire prefix, there is no known deterministic solution. However, we can use randomization. We compute fingerprints by hashing the considered prefixes. Unique prefixes are identified by unique fingerprints. False positives may lead us to miss distinct prefixes, which have the same fingerprint by pure chance. In this case, we would send more characters than necessary, but we preserve correctness.

Approximating distinguishing prefixes is desirable. It reduces the time spent on computing the distinguishing prefix, while still decreasing the communication volume. An efficient approximation in distributed memory based on prefix doubling is described in [86]. During the algorithm, a candidate set C_i of all strings $\mathcal{S}_i$ on PE i is maintained whose distinguishing prefix has not yet been determined. Initially, C_i contains all strings in $\mathcal{S}_i$ and the current prefix length l is set to a small starting value. The algorithm works in rounds.

1. **Local Processing.** Compute the fingerprints of the l-prefixes in the candidate set C_i, locally sort the hash values and remove duplicates.

2. **Exchange**. Send the hash values h in the range $(k-1)(m/p) \leq h < k(m/p)$ to PE k, where m is the maximum value of the hash function.

3. **Duplicate Detection.** Locally determine duplicates of the received hash values by sorting or merging the incoming fingerprints. Send a bit array to each PE, where the j-th bit indicates whether the j-th received fingerprint of the sending PE is unique.

4. **Local Postprocessing.** Store l as the length of the distinguishing prefix for unique candidates and remove them from C_i. Set l to $2 \cdot l$ and continue with Step 1 until the candidate set is empty.

5. Suffix Sorting using Difference Covers

Difference Cover modulo X (DCX) was the first linear time suffix array construction algorithm [46]. The key idea is to recursively compute the suffix array of sample suffixes with a useful mathematical structure. Using the ranks of the sample suffixes, a comparison function can be defined that allows to sort all suffixes efficiently.

This chapter is structured as follows. First, we explain the sequential DC3 algorithm with a detailed example execution (Section 5.1), we introduce *difference covers* (Section 5.2), a combinatorial structure underlying DC3 that allows to define a generalized variant called DCX (Section 5.3). Next, we present our main algorithm, the distributed version of DCX (Section 5.4). In the following sections, we describe *discarding* (Section 5.5), *bucketing* (Section 5.6) and *chunking* (Section 5.7), algorithmic techniques that aim to improve distributed DCX in running time and memory consumption. Then, we explain further optimizations regarding string sorting and packing (Section 5.8). Lastly, we give details about string containers, data types and practical considerations when implementing bucketing (Section 5.9).

5.1. Example Execution of DC3 Algorithm

Our description of DC3 and DCX closely follows Kärkkäinen et al. [46]. We begin with a detailed example execution of DC3 on the following text. The text is padded with three sentinel characters to account for samples that extend beyond the text boundary.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	b	c	a	b	b	c	c	a	b	$	$	$

Step 0: Construct Samples. We denote the set of periodic positions for $k = 0, 1, 2$ by

$$P_k = \{i \in [0, n] \mid i \bmod 3 = k\}.$$

Let $C = P_1 \cup P_2$ be the set of *sample positions* and S_C the *sample suffixes* with starting positions C.

Example.

$$P_1 = \{1, 4, 7, 10\},\ P_2 = \{2, 5, 8, 11\},\ C = \{1, 2, 4, 5, 7, 8, 10, 11\}.$$

Step 1: Sort Sample Suffixes. In this step, we want to determine the ranks of the sample suffixes S_C. We require a helper string, which characters consist of triples $[x\ y\ z]$ for $k = 1, 2$

$$R_k = \bigodot_{i \in P_k} [T[i]\ T[i+1]\ T[i+2]],$$

where $\bigodot$ is the concatenation operator. Let $R = R_1 \odot R_2$. Notice that the set of suffixes of R corresponds to the set of sample suffixes S_C. By sorting the suffixes of R, we also get the order of the sample suffixes.

Example.

$$R = \texttt{[bbc][abb][cca][b\$\$][bca][bbc][cab][\$\$\$]}$$

To be able to work with an integer alphabet, we do not directly sort the suffixes of R. Instead, we radix sort the triples of R and replace each with its rank to obtain R'.

Example.

triple	$\$\$\$$	abb	b$\$$	bbc	bbc	bca	cab	cca
rank	0	1	2	3	3	4	5	6

$$R' = (3, 1, 6, 2, 4, 3, 5, 0).$$

If all ranks of R' are unique, we have already determined the sample suffix ranks. This occurs, if the sample suffixes are distinguishable by their first three characters. Otherwise, we call DC3 recursively on R', to obtain the suffix array of $SA_{R'}$. Inverting the permutation $SA_{R'}$, we get the ranks of each sample suffix, i.e. the inverse suffix array $ISA_{R'}$.

$$SA_{R'} = (7, 1, 3, 0, 5, 4, 6, 2),\ ISA_{R'} = (3, 1, 7, 2, 5, 4, 6, 0).$$

Now, we know the order of all sample suffixes and still have to determine the ranks for all suffixes. Let $rank(S_i)$ denote the rank of a sample suffix in S_C. We set $rank(S_i) = 0$ for padding positions and leave $rank(S_i)$ undefined for non-sample positions.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	b	c	a	b	b	c	c	a	b	$	$	$
$ranks(S_i)$	$\perp$	3	4	$\perp$	1	3	$\perp$	6	5	$\perp$	2	0	0	0

Step 2: Sort Non-Sample Suffixes. To sort the non-sample suffixes S_{B_0}, we can use the following observation:

$$S_i \leq S_j \Leftrightarrow (T[i], rank(S_{i+1}) \leq (T[j], rank(S_{j+1}), \quad i, j \in B_0.$$

When we compare the suffixes S_i and S_j character-by-character we can stop and use $rank(\cdot)$ once both positions correspond to sample positions. This occurs after the first step, since non-samples positions are in P_0, and P_1 are sample-positions. Again, radix sort is used to sort the tuples. The choice of samples can be generalized to use so-called *difference cover samples* as we will later see in Section 5.3.

Example. $S_9 < S_0 \Leftrightarrow (\mathsf{a}, S_{10}) < (\mathsf{a}, S_1) \Leftrightarrow (\mathsf{a}, 2) < (\mathsf{a}, 3)$.

Step 3: Merge Sample and Non-Sample Suffixes. Using standard 2-way merging on the two sorted sets, we obtain the suffix array of T. With a similar argument as in Step 2, we can define a comparison function for a suffix $S_i \in S_C$ and $S_j \in S_{P_0}$ by distinguishing two cases:

$$i \in P_1 : S_i \leq S_j \Leftrightarrow (T[i], rank(S_{i+1})) \leq (T[j], rank(S_{j+1}))$$
$$i \in P_2 : S_i \leq S_j \Leftrightarrow (T[i], T[i+1], rank(S_{i+2})) \leq (T[j], T[j+1], rank(S_{j+2})).$$

In the first case, $i+1 \in P_2$ and $j+1 \in P_1$, thus, the ranks are well defined. In the second case, $i+1 \in P_0$ and $j+1 \in P_1$, consequently we have to compare an additional character. After two steps we have $i+2 \in P_1$ and $j+2 \in P_2$ and we can use the rank information of the sample suffixes.

Example. $S_0 < S_4 \Leftrightarrow (\mathsf{a}, S_1) \leq (\mathsf{a}, S_5) \Leftrightarrow (\mathsf{a}, 3) \leq (\mathsf{a}, 5)$ and $S_8 < S_3 \Leftrightarrow (\mathsf{c}, \mathsf{a}, S_{10}) \leq (\mathsf{c}, \mathsf{a}, S_5) \Leftrightarrow (\mathsf{c}, \mathsf{a}, 2) \leq (\mathsf{c}, \mathsf{a}, 3)$.

Complexity. On each level the algorithm takes linear time, as radix sort is used for sorting routines. Overall, we obtain linear complexity by applying the Master Theorem to the recurrence $T(n) = T(2n/3) + O(n)$.

5.2. Difference Cover Samples

The sample used in DC3 is a special case of a *difference cover*. Here, we introduce some basic definitions and properties of difference covers. In the next section, we describe the generalized DCX algorithm that works with any difference cover.

Definition 1. A set $D_X \subseteq [0, X)$ is a *difference cover* modulo X if

$$\{(i - j) \bmod X \mid i, j \in D_X\} = [0, X).$$

We call X the *period length* and denote the *reduction ratio* of D_X by $\lambda = |D_X|/X$. In the generalized version of DCX, λ will be the reduction ratio of the recursive subproblem. Table 5.1 shows a list difference covers up to $X = 133$ and their corresponding value of λ.

Clearly, $\sqrt{X}$ is a lower bound on the size of any difference cover D_X. Otherwise, there are to few of pairs of integers to produce $[0, X)$. Furthermore, for any X, a difference cover D_X of size at most $\sqrt{1.5X} + 6$ can be computed in $O(\sqrt{X})$ time [23].

Example. $D_7 = \{1, 2, 4\}$ is a minimal difference cover for $X = 7$.

$$0 \equiv 1 - 1, \quad 1 \equiv 2 - 1, \quad 2 \equiv 4 - 2, \quad 3 \equiv 4 - 1, \quad 4 \equiv 1 - 4, \quad 5 \equiv 2 - 4, \quad 6 \equiv 1 - 2.$$

A smaller difference cover cannot exists, since $\sqrt{7} \approx 2.645\ldots$ is the lower bound.

Definition 2. Any index $i \in [0, n]$ can be written uniquely as $i = X \cdot m + d$ for $m \leq n$ and $d \in [0, X)$. We call m the *period* and d the *phase* of i.

Example. Let $X = 7$ and $i = 18 = 7 \cdot 2 + 4$. Then, the period of i is 2 and the phase is 4.

$\lvert D_X \rvert$	Ring	D_X	λ
2	$\mathbb{Z}_3$	$\{1, 2\}$	0.667
3	$\mathbb{Z}_7$	$\{1, 2, 4\}$	0.429
4	$\mathbb{Z}_{13}$	$\{1, 2, 4, 10\}$	0.308
5	$\mathbb{Z}_{21}$	$\{1, 2, 7, 9, 19\}$	0.238
6	$\mathbb{Z}_{31}$	$\{1, 2, 4, 9, 13, 19\}$	0.194
7	$\mathbb{Z}_{39}$	$\{1, 2, 17, 21, 23, 28, 31\}$	0.179
8	$\mathbb{Z}_{57}$	$\{1, 2, 10, 12, 15, 36, 40, 52\}$	0.140
9	$\mathbb{Z}_{73}$	$\{1, 2, 4, 8, 16, 32, 37, 55, 64\}$	0.123
10	$\mathbb{Z}_{91}$	$\{1, 2, 8, 17, 28, 57, 61, 69, 71, 74\}$	0.110
11	$\mathbb{Z}_{95}$	$\{1, 2, 6, 9, 19, 21, 30, 32, 46, 62, 68\}$	0.116
12	$\mathbb{Z}_{133}$	$\{1, 2, 33, 43, 45, 49, 52, 60, 73, 78, 98, 112\}$	0.090

Table 5.1.: Table of difference covers D_X for $X = 2, 3 \ldots, 12$ and their corresponding value of the reduction ratio λ.

Definition 3. Given a difference cover D_X, a *difference cover sample* is a X-periodic sample of D_X

$$C = \{i \in [0, n] \mid i \textbf{ mod } X \in D_X\}.$$

Example. Let $n = 18$, $X = 7$ and $D_7 = \{1, 2, 4\}$, then

$$C = \{1, 2, 4, 8, 9, 11, 18\}$$

An important property of difference covers that we later require to define the comparison function for sorting all suffixes is given by the following Lemma.

Lemma 4 (Lemma 1 in [46])**.** If D_X is a difference cover modulo X, and i and j are integers, there exists $l \in [0, X)$ such that $(i + l)$ **mod** X and $(j + l)$ **mod** X are in D_X.

Proof. Since D_X is a difference cover, by definition, there exists $i', j' \in D_X$, such that $(i - j) \equiv (i' - j')$ **mod** X. We define $l = (i' - i)$ **mod** X. Then

$$\begin{aligned} i + l &\equiv i + (i' - i) \equiv i' \in D_X \ \textbf{ mod } X \\ j + l &\equiv i' - (i - j) \equiv i' - (i' - j') \equiv j' \in D_X \ \textbf{ mod } X \end{aligned}$$

□

Example. Let $D_7 = \{1, 2, 4\}$, $X = 7$ and $i = 4$, $j = 5$. Then 4+4 $\equiv 1 \in D_7$ and 5+4 $\equiv 2 \in D_7$. Hence $l = 4$ in this case.

5.3. The General DCX Algorithm

DC3 uses difference cover samples based on $D_3 = \{1, 2\}$ over the ring $\mathbb{Z}_3$ to sort all suffixes. The generalized DCX algorithm works similar in principle and can use any difference cover D_X.

Step 0: Construct Samples. For $k \in [0, X)$, we denote the set of periodic positions by

$$P_k = \{i \in [0, n] \mid i \bmod X = k\}.$$

Let $\overline{D}_X = [0, X) \setminus D_X$ be the elements not contained in the difference cover D_X. Sample positions are now defined as $C = \bigcup_{k \in D_X} P_k$ and non-sample positions as $\overline{C} = [0, n] \setminus C$.

Step 1: Sort Sample Suffixes. To determine the ranks of the sample suffixes S_C, we define the helper strings with characters consisting of X-tuples for $k \in D_X$

$$R_k = \bigodot_{i \in P_k} [T[i]\ T[i+1]\ \ldots, T[i+X-1]],$$

and let $R = \bigodot_{i \in D_X} R_i$. Recall, that the suffixes of R correspond to the sample suffixes S_C. Thus, sorting the suffixes of R we get the order of the sample suffixes. Similar to DC3, we sort the X-tuples, replace them by their ranks and recursively continue with R' if not all ranks are unique. Note that the underlying alphabet Σ changes to $[0, M]$ in a recursive step where M is the largest rank in R'. We define $rank(S_i)$ as before.

Step 2: Sort Non-Sample Suffixes. We sort each S_{P_k} for $k \in \overline{D}_X$ separately. A similar observation to the one in Step 2 of DC3 (Section 5.1) can be made.

$$S_i \leq S_j \Leftrightarrow (T[i, i+l), rank(S_{i+l}) \leq (T[j, j+l), rank(S_{j+l}).$$

When comparing two suffixes S_i, S_j with $i, j \in P_k$, we compare character-by-character until the current position is a sample suffix. By Lemma 4, this takes $l \in [0, X)$ steps. We could apply a X-way merging routine with a $\Theta(X)$-comparison function, directly generalizing DC3. This would result in $O(nX \log X)$ complexity. However, there is better way, which only requires $O(nX)$ time.

Step 3: Sort by First X Characters. We separate the sample suffixes S_C into S_{P_k} for $k \in D_X$, keeping each set sorted. All the sets S_{P_k} for $k \in [0, X)$ are now in different sorted sequences. In the next step, we concatenated these sequence and stably sort them by their first X characters. Let S^α denote the set of suffixes starting with $\alpha \in \Sigma^X$ and $S^\alpha_{P_k} = S^\alpha \cap S_{P_k}$. After the sorting, the sequence is grouped into sets S^α, which in turn are grouped into subgroups S_{P_k}, $k \in [0, X)$. For $X = 3$, the situation could look like this:

$S^{\mathsf{aaa}}_{P_0}$	$S^{\mathsf{aaa}}_{P_1}$	$S^{\mathsf{aaa}}_{P_2}$	$S^{\mathsf{aab}}_{P_0}$	$S^{\mathsf{aab}}_{P_1}$	$S^{\mathsf{aab}}_{P_2}$	$S^{\mathsf{aac}}_{P_0}$	$\cdots$

Step 4: Merge Sample and Non-Sample Suffixes. Notice, that by the sorting we only have to reorder suffixes within each group S^α. We merge the subgroups in each S^α, $\alpha \in \Sigma^X$ using a comparison-based X-way merging routine. This completes the sorting. To compare the suffix S_i with S_j, we compare the ranks $rank(S_{i+l}),\ rank(S_{j+l})$, where $l \in [0, X)$ such that $(i+l) \bmod X$ and $(j+l) \bmod X$ are both in D_X (Lemma 4).

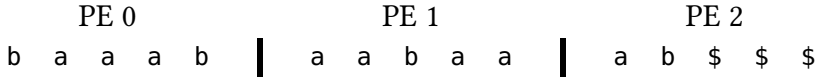
Complexity. Similar to DC3, we obtain a recurrence $T(n, X) = T(\lambda n) + \mathcal{O}(nX)$ where $\lambda = |D_X|/X$. The work on a level is dominated by radix sorting the X-tuple. Applying the Master Theorem results in $\mathcal{O}(nX)$ complexity.

The Choice of the Period Length X. DCX allows flexibility in the choice of the difference cover D_X and X. Larger values of X achieve a greater reduction of the subproblem, but increase the work of the sorting routines. In practice, the best choice of X depends on characteristic of the input and the concrete implementation.

5.4. The Distributed DCX Algorithm

Distributed DCX is a straightforward generalization of the distributed DC3 algorithm by Kulla and Sanders [52]. It works similarly to the sequential DCX algorithm. However, in the last step, instead of merging multiple sorted sequences, it globally sorts all samples and non-samples together. Distributed k-way merging requires us to partition the sorted sequence among the PEs, such that after local merging the global sequence is sorted. This is similar to how in distributed sorting a single sequence is partitioned. Thus, the simpler globally sorting is the preferred choice in the distributed setting.

Algorithm 3 shows a high-level pseudocode for the algorithm. As a running example we compute the SA of the following text using distributed DC3 with $D_3 = \{1, 2\}$ and three PEs. We indicate the distribution of the data between the PEs with extra white spaces and bars.



input text T with padding

Phase 1: Sorting of the Difference Cover Sample (Line 2 - Line 3). In the first phase of the algorithm, the difference cover samples S are sorted. Sorting the samples requires access to parts that are not local to a PE. Thus, we explicitly *materialize* a sample with global index i together with the next X characters $T[i, i + X)$. Each PE j materializes the difference cover samples of their local input text T_j. The last samples on PE $j < p - 1$ might overlap with the local text on PE $j + 1$. To handle this, PE $j + 1$ sends its first $X - 1$ characters to PE j. The samples of the last PE $j = p - 1$ can reach beyond the input text. Therefore, at the beginning of a level, the last PE appends X copies of the character \$ to the input text. These padding characters will not be part of the final suffix array. In the example, we indicate characters from neighboring PEs in gray.

	PE 0							PE 1							PE 2				
i	0	1	2	3	4			5	6	7	8	9			10	11	12	13	14
T	b	a	a	a	b	a	a	a	a	b	a	a	a	b	a	b	$	$	$

input text T with shifts and padding

i	1	2	4	5	7	8	10	11
S	aaa	aab	baa	aab	baa	aaa	ab$	b$$

materialize difference cover samples S

i	1	8	2	5	10	11	4	7
S	aaa	aaa	aab	aab	ab$	b$$	baa	baa

difference cover samples S in lexicographic order

Table 5.2.: Example Phase 1 of distributed DCX.

Phase 2: Naming the Sample Ranks (Line 4). Now, the samples are replaced with their rank. The rank entries in the array P can be expressed in terms of the following sum:

$$\Delta_i = S[i] \neq S[i+1] \quad \text{for } 0 \leq i < |S| - 1$$
$$P[i] = \sum_{j=0}^{i-1} \Delta_j \quad \text{for } 0 \leq i < |S|.$$

Each PE locally determines Δ_i by comparing the X-prefixes of consecutive samples. This requires another communication with the next PE (if $j < p - 1$) to compare the last entry. Afterwards, we compute the local ranks using a local scan on Δ_i. Let e_j be the local sum of Δ_i on PE j. To correctly obtain the global ranks, we shift each rank by o_j, the result of the global scan operation on e_j. In the example, shifted samples are marked in gray and e_j is marked in bold.

	PE 0				PE 1				PE 2		
S	aaa	aaa	aab	aab	aab	ab$	b$$	baa	baa	baa	
Δ_i	0	1	0		1	1	1		0		
local ranks	0	0	1	**1**	0	1	2	**3**	0	0	**0**
o_j				0				1			4
P	0	0	1		1	2	3		4	4	

computing the sample ranks in P

Table 5.3.: Example Phase 2 of distributed DCX.

Phase 3: Recursively Compute Unique Ranks (Line 5 - Line 10). If the last rank in P is $|S|-1$, all ranks are unique. This is communicated via a broadcast operation from the last PE. Otherwise, we determine the unique sample ranks by recursion. We sort the ranks by their global index i using $(i \textbf{ mod } X, i \textbf{ div } X)$ as the comparison function. This partitions the samples into blocks $A_0, A_1, \ldots, A_{|D_X|-1}$ where the k-th block contains the ranks of the k-th difference cover element $(i \textbf{ mod } X)$ sorted by their global index $(i \textbf{ div } X)$. Recursively calling DCX on the reordered sample ranks yields the suffix array SA' of the sample suffixes. Line 7 pairs up the new ranks $(0, 1, \ldots |S|-1)$ with the original global indices of the sample suffixes. Finally, we bring the sample ranks back into text-order (Line 9 - Line 10).

The index transformation is realized by the MapBack function (Line 21 - Line 27). Any global sample index $i = X \cdot m + d$ is uniquely determined by its period m and its phase d. In the loop, we determine into which block A_k the index i belongs and thus the corresponding phase $d = DC[k]$. The period m is the local index i in A_k.

	PE 0			PE 1			PE 2	
i	1	4	7	10	2	5	8	11
P	0	4	4	2	1	1	0	3

P sorted by $(i \textbf{ mod } X, i \textbf{ div } X)$

SA'	6	0	5	4	3	7	2	1
phase(SA')	2	1	2	2	1	2	1	1
period(SA')	2	0	1	0	3	3	2	1
MapBack(SA')	8	1	5	2	10	11	7	4
P	0	1	2	3	4	5	6	7

map back SA', $|A_0| = |A_1| = 4$

i	1	2	4	5	7	8	10	11
P	1	3	7	2	6	0	4	5

P with unique sample ranks in text-order

Table 5.4.: Example Phase 3 of distributed DCX.

Phase 4: Globally Sort All Suffixes (Line 11 - Line 14). In the last phase, we globally sort all suffixes using the sample ranks we determined in Phase 3. The comparison function is stated in Line 31 - Line 36. Because of Lemma 4, it is sufficient to compare $l \in [0, X)$ characters and then access the corresponding ranks at positions $r_1, r_2 \in [0, |D_X|)$ to realize the comparison. The constants l, r_1 and r_2 only depend on the underlying difference cover and the remainder of the global indices $i \textbf{ mod } X$ and $j \textbf{ mod } X$. Therefore, we can precompute this information in a lookup-table.

Similar to Phase 1, we make the information required for comparing two suffixes locally available to each PE by materializing substrings of the text and ranks. The text is shifted in the same way as in Phase 1. For shifting the ranks, PE $i < p - 1$ receives the first $|D_X| - 1$ locally stored ranks of PE $i + 1$. The last PE uses $|D_X| - 1$ dummy ranks $\perp$ instead. Dummy ranks are never accessed since the X-prefixes at the end of the text contain a unique number of \$ characters. To compare one suffix with global index i, we store the next $X - 1$ characters $T[i, i + X - 1)$ and the next $|D_X|$ ranks $R[\mathrm{F}(i), \mathrm{F}(i) + |D_X|)$ for a transformed index $\mathrm{F}(i)$. The last character of the X-prefix is not needed, since $l < X$.

The transformation $\mathrm{F}(i)$ works as follows. We know that every block of $|D_X|$ consecutive ranks in R corresponds to the difference cover samples of every block of X consecutive characters in T. To map from the X-blocks to the $|D_X|$-blocks, we determine the period by i **div** X and compute the position of the first rank in this period by $|D_X| \cdot (i \textbf{ div } X)$. Now, we have to offset this position by a constant $d \in [0, |D_X|)$ to jump to the rank of the next difference cover sample to the right of i. The constant d only depends on the underlying difference cover and the phase of i and can be precomputed in a lookup-table as well.

Finally, we extract the global indices from the sorted sequence M, which compose the suffix array SA of the input text.

	PE 0				PE 1				PE 2			
j	0	1	2		3	4	5		6	7	8	9
R	1	3	7	2	2	6	0	4	4	5	$\perp$	$\perp$

shifted sample ranks R with dummy rank

i	0	1	2	3	4	5	6	7	8	9	10	11
$2 \cdot (i \textbf{ div } 3)$	0	0	0	2	2	2	4	4	4	6	6	6
$LT_1[i \textbf{ mod } 3]$	0	0	1	0	0	1	0	0	1	0	0	1
$F(i)$	0	0	1	2	2	3	4	4	5	6	6	7
T	b	a	a	a	b	a	a	b	a	a	a	b
$T[i, i + 2)$	ba	aa	aa	ab	ba	aa	ab	ba	aa	aa	ab	b\$
$R[F(i), F(i) + 1]$	(1, 3)	(1, 3)	(3, 7)	(7, 2)	(7, 2)	(2, 6)	(6, 0)	(6, 0)	(0, 4)	(4, 5)	(4, 5)	($\perp$, $\perp$)

materializing characters and ranks of M

i	8	1	9	5	2	10	6	3	11	7	0	4
$T[i, i + 2]$	aa	aa	aa	aa	aa	ab	ab	ab	b\$	ba	ba	ba
$R[F(i), F(i) + 1]$	(0, 4)	(1, 3)	(4, 5)	(2, 6)	(3, 7)	(4, 5)	(6, 0)	(7, 2)	($\perp$, $\perp$)	(6, 0)	(1, 3)	(7, 2)

sorting all suffixes in M

Computing the Lookup-Tables. Lookup-table LT_1 stores the offset used when mapping a suffix in a X-block to the corresponding sample rank in the $|D_X|$-blocks of the same period. Let D be a bit array of length X with $D[i] = 1$ if $i \in |D_X|$ and 0 otherwise. LT_1 is

	0	1	2
0	(1, 0, 0)	(1, 0, 1)	(2, 1, 1)
1	(1, 1, 0)	(0, 0, 0)	(0, 0, 0)
2	(2, 1, 1)	(0, 0, 0)	(0, 0, 0)

lookup-table LT_2 with constants (l, r_1, r_2)

i	j	l	r_1	r_2	$T[i, i+l)$	$T[j, j+l)$	$R[F(i), F(i)+1][r_1]$	$R[F(i), F(i)+1][r_2]$
6	3	1	0	0	a	a	6	7
2	6	2	1	1	aa	ab	7	0
8	1	0	0	0	-	-	0	1

example comparison function

Table 5.5.: Example Phase 4 of distributed DCX.

the exclusive prefix sum of D, i.e. $LT_1[i] = LT_1[i-1] + D[i-1]$ for $i > 0$ and $LT_1[0] = 0$. This holds, since we have to skip over a rank entry, once we pass a sample suffix in the X-block. The second lookup-table LT_2 contains the constant l, such that $i + l \in D_X$ and $j + l \in D_X$ for $i, j \in [0, X)$ and the rank positions r_1 and r_2 in the $|D_X|$ materialized ranks of suffix S_i and S_j. The constant l can be computed as follows. Set $l = 0$ and increment l until $D[(i + l) \bmod X] = D[(j + l) \bmod X] = 1$. The value of r_1 and r_2 are the exclusive prefix sum of the entries we traverse in D, by the same reasoning as before.

Data Distribution. We assume that at the beginning the input text T is distributed equally among the PEs. Each PE j holds a consecutive slice T_j of size $\Theta(n/p)$. The distribution of the data may change depending on the distributed sorter used. There is no guarantee for a minimal number of elements on a PE. In fact, a PE might have no elements after sorting. To perform the shifts of ranks and characters between adjacent PEs, we require at least $X - 1$ local elements. Thus, whenever the smallest data size of all PEs is smaller than $4\times$ the average size, we balance the data using an alltoall communication. When the total size of the recursive string is smaller than $2pX$, we gather all characters on a single PE and use a sequential suffix sorting algorithm to avoid corner cases. We could also balance with respect to the largest number of elements on a PE after sorting. However, many distributed sorters, like for example AMS, guarantee a configurable imbalance, which is why did not include this step in our implementation.

Memory Consumption. Memory-efficiency is vital in the design of scalable algorithms to be able to handle large inputs. Here, we give a detailed analysis of the total memory consumption of distributed DCX (Algorithm 3). Let w_c, w_r, w_X denote the number of bytes required to store an input character, a rank/index and an entry in lookup-tables. In a recursive call, w_c depends on the largest rank given in Line 4. We may assume that w_c, w_r and w_X are constant and on deeper levels $w_c = w_r$. The difference cover has a size of $O(|D_X|) = O(\sqrt{X})$ (see Section 5.2). Let $\lambda = |D_X|/X$ be the reduction ratio of the text size

Algorithm 3: Distributed DCX.

```
1  Function DCX(T):
2      S = ⟨(T[i, i+X), i) | i ∈ [0, n), i mod X ∈ D_X⟩      // generate DC-samples
3      sort S by the first component                          // lexicographic-order
4      P = Name(S)                                            // compute sample ranks
5      if names in P are not unique then
6          sort P by (i mod X, i div X)
7          SA' = DCX(⟨c | (c, i) ∈ P⟩)                        // recursively compute SA'
8          P = ⟨(i, MapBack(SA'[i]) | 0 ≤ i < |SA'|⟩          // new sample ranks
9      sort P by the second component                         // text-order
10     R = ⟨r | (i, r) ∈ P⟩
11     M = ⟨(T[i, i+X−1), R[F(i), F(i)+|D_X|), i) | i ∈ [0, n)⟩
12     sort M by Cmp                                          // sort all suffixes
13     SA = ⟨i | (T', R', i) ∈ M⟩                             // extract SA
14     return SA
15 Function Name(⟨(a_1, b_1), ..., (a_k, b_k)⟩):
16     x = 1
17     for i = 1 to k do
18         output tuple (x, b_i)
19         if a_{i−1} ≠ a_i then                              // compare X-prefix
20             x++
21 Function MapBack(i):
       // A_0, A_1, ..., A_{|D_X|−1} blocks of samples in P
22     B = ⟨0, |A_0|, |A_0|+|A_1|, ..., Σ_{k=0}^{|D_x|−1} A_k⟩
23     for 0 ≤ k < |D_X| do
24         if i < B[k+1] then                                 // determine block
25             d = DC[k]                                      // phase of i
26             m = i − B[k]                                   // period of i
27             return X · m + d
28 Function F(i):
29     d = LT_1[i mod X]
30     return (|D_X| · (i div X)) + d                         // position in R
31 Function Cmp((T_1, R_1, i), (T_2, R_2, j)):
32     l, r_1, r_2 = LT_2[i mod X, j mod X]
33     for 0 ≤ k < l do
34         if T_1[k] ≠ T_2[k] then                            // compare X−1-prefix
35             return T_1[k] < T_2[k]
36     return R_1[r_1] < R_2[r_2]                             // compare sample ranks
```

per level and $||A||$ the memory consumption of the object A. For now, we fix a level $r \geq 0$. Counting the size of the vectors is straightforward. The vectors S, P, SA' and R have one entry per difference cover sample, while M and SA have an entry for each input character.

- $||S|| = \lambda^{r+1} n \cdot (X w_c + w_r)$
- $||P|| = \lambda^{r+1} n \cdot 2w_r$
- $||SA'|| = \lambda^{r+1} n \cdot w_r$
- $||R|| = \lambda^{r+1} n \cdot w_r$
- $||M|| = \lambda^r n \cdot ((X-1) \cdot w_c + (|D_X|+1) \cdot w_r)$
- $||SA|| = \lambda^r n \cdot w_r$
- $||LT_1|| = X \cdot w_X$
- $||LT_2|| = X^2 \cdot 3w_X$

Now, we highlight local memory peaks in the pseudocode. We deallocate vectors as soon as they are not required anymore. The memory of lookup-tables is negligible, thus we ignore them in the following. In sorting routines, we require an additional receive buffer for communication of the same size as the container to be sorted.

- Line 3: $||S||$ + receive buffer
- Line 4: $||S|| + ||P||$
- Line 6: $||P||$ + receive buffer
- Line 7: $||SA'|| + ||P||$
- Line 10: $||P||$ + receive buffer
- Line 12: $||M||$ + receive buffer
- Line 13: $||M|| + ||SA||$

The maximum memory peak occurs in Line 12 when sorting all suffixes

$$2\lambda^r n \cdot ((X-1) \cdot w_c + (|D_X|+1) \cdot w_r) \in O(nX).$$

In addition to the memory consumption stated above, we have to store the inputs to the recursive calls. Therefore, on level r we use

$$\sum_{i=1}^{r} w_r \lambda^r n = w_r n \left(\frac{1-\lambda^{r+1}}{1-\lambda} - 1 \right) \leq w_r n \frac{1}{1-\lambda} \in O(n)$$

additional bytes. Here, we used $\frac{1}{1-\lambda} \leq \frac{1}{1-(2/3)} = 3$. The overall memory complexity is given by

$$O(nX).$$

In practice, we use 1-byte input characters, 5-byte integers for indices/ranks (see Section 5.9.2) and difference covers up to $X = 133$. To reduce the memory footprint of distributed DCX, we adapt a space-efficient sorting technique [55, 68] (see Section 5.6) and use random redistribution of chunks of the data (see Section 5.7).

Complexity. In distributed DCX, we only use linear scans, sorting routines, alltoall data exchanges and communication between adjacent PEs. Thus, the time complexity on a fixed level is dominated by the sorting of all suffixes (Line 12). The following recurrence describes the overall time complexity:

$$T(n, p, X) = T(\lambda n, p, X) + O(T_{sort}(n, p, X)),$$

where $\lambda = |D_X|/X$ is the reduction ratio of the text size. We will assume that the size of the local data n' is always distributed equally among the PEs, i.e. $n' \in \Theta(n/p)$. For simplicity, we use the sorting complexity of sample sort for $T_{sort}(n,p)$, which is given by:

$$T_{sort}(n,p) = O\left(\frac{n}{p}\log\frac{n}{p} + \alpha p + \beta\frac{n}{p}\right).$$

The three parts come from sorting the local data of size $O(n/p)$ and the complexity of an irregular alltoall communication $O(\alpha p + \beta h)$ with $h \in O(n/p)$. We may assume that a comparison of an element of M costs $O(X)$ and that the total size of the data to be exchanged is in $O(nX)$ (see memory analysis). Therefore, the sorting complexity of M becomes:

$$T_{sort}(n,p,X) = O\left(\frac{nX}{p}\log\frac{n}{p} + \alpha p + \beta\frac{nX}{p}\right).$$

Let $r \leq \lceil \log_{1/\lambda} n \rfloor$ denote the number of levels. We unroll the recurrence and plug-in the formula for $T_{sort}(n,p,X)$ for a sufficiently large constant $C > 0$:

$$\begin{aligned}
T(n,p,X) &\leq C\sum_{i=0}^{r-1} T_{sort}(\lambda^i n, p, X) \\
&= C\sum_{i=0}^{r-1}\left(\frac{\lambda^i nX}{p}\log_2\frac{\lambda^i n}{p} + \alpha p + \beta\frac{\lambda^i nX}{p}\right) \\
&\leq C\left[\frac{1}{1-\lambda}\frac{nX}{p}\log_2\frac{n}{p} + \alpha\lceil\log_{1/\lambda} n\rceil p + \beta\frac{1}{1-\lambda}\frac{nX}{p}\right] \\
&\in O\left(\frac{nX}{p}\log\frac{n}{p} + \alpha\frac{\log n}{\log X}p + \beta\frac{nX}{p}\right).
\end{aligned}$$

In the last step, we used that $\frac{1}{1-\lambda} \leq \frac{1}{1-(2/3)} = 3$ and $1/\lambda = X/|D_X| \in O(\sqrt{X})$.

5.5. Discarding

Discarding is a technique originally proposed by Dementiev et al. [25] to improve the I/O efficiency of an external memory prefix doubling suffix array algorithm. Fischer and Kurpicz [31] successfully included this technique in their distributed prefix doubling implementation. It is based on the observation that after a prefix doubling iteration some suffixes with unique rank can be *discarded* from the next iteration. A rank r is not required in the next iteration if (a) r is unique and (b) r is not needed to determine a non-unique rank. Condition (b) occurs if the previous rank r' of r in text-order is unique as well. When comparing suffixes character-by-character, the comparison result will at the latest be determined on the previous character r' and we never have to examine r. Globally, this property implies that, from any group of consecutive unique ranks, we only have to keep the first rank.

Example. Consider the example in Table 5.6. We want to determine the ranks of the string 1 2 3 1 4 5 6 7. Inverting the permutation of unique ranks yields the suffix array. Since the suffix 1 2 3 1 4 5 6 7 and 1 4 5 6 7 are the only suffixes that are not uniquely determined by their first character, we only have to decide on the relative order of those two suffixes. The characters 3, 5, 6 and 7 are unique and are not the first element of a group of unique characters. Thus, the reduced text is 1 2 1 4. Looking at the rank array of both text, we see that the relative order of unique ranks has not changed. The non-unique ranks can be determined only from the ranks of the reduced text. The suffix 1 2 1 4 is smaller than the suffix 1 4, thus the first suffix has a lower rank (rank 1) than the second suffix (rank 2).

		↓			↓			
text	1	2	3	1	4	5	6	7
reduced text	1	2	1	4				
ranks reduced text	1	3	2	4				
ranks text	1	3	4	2	5	6	7	8

Table 5.6.: Example of discarding. Unique ranks are colored in red and unique ranks that are required to determine not-unique ranks are marked with an arrow.

Discarding in Phase 3 of DCX. We can modify Phase 3 of DCX (Line 5 - Line 10) to include discarding. Recall, that the goal of Phase 3 is to recursively compute the unique ranks of the difference cover samples. The general idea is to compute the suffix array SA_d of the reduced text and to use the ranks of the inverse suffix array ISA_d as a tie-breaker when determining the relative order of the non-unique ranks.

Algorithm 4 outlines the procedure. We sort the ranks P by (i **mod** X, i **div** X) as before (Line 1). In one scan over the ranks P_r, we construct a boolean mask B that is `true`, if the i-th rank cannot be discarded, `false` otherwise (Line 2 - Line 5). We then construct the reduced string T_d and recursively compute the corresponding suffix array SA_d of it (Line 6 - Line 7). Since we require the ranks R_d of the reduced string T_d, we invert the suffix array SA_d by pairing it with $0, 1, \ldots, |SA_d| - 1$ and sorting it by the second component (Line 8 - Line 9). Now, we construct a vector of triples RRI, where the first component is the rank r from the naming procedure P, the second the rank of the reduced string r', if the current position was not discarded, and in the last component the global index i of the difference cover sample (Line 11 - Line 17). We sort RRI by the first two components. To determine the relative order of ranks that were not unique before, r' serves as a tie-breaker. After the sorting, the last component of RRI contains the suffix array of the difference cover samples. We zip the suffix array indices in RRI with new ranks $0, 1, \ldots |P| - 1$ and rearrange them into text-order such that the first component contains the unique ranks of the samples (Line 18 - Line 19). Table 5.7 shows a larger example of discarding in Phase 3.

Discarding has a slight overhead compared to the normal Phase 3, but is more efficient if many ranks can be discarded. In our implementation, we first determine the size of the reduced string and decide based on a threshold whether to use the discarding procedure.

	↓				↓				↓			
P	4	3	6	7	5	0	1	7	9	8	2	1
i	1	4	7	10	13	16	2	5	8	11	14	17

P sorted by $(i \bmod X, i \operatorname{div} X)$

B	1	0	0	1	1	0	1	1	1	0	0	1
T_d	4			7	5		1	7	9			1
SA_d	6			3	0		2	1	4			5
R_d	2			4	3		1	5	6			0

computing the ranks R_d of the reduced text T_r

RRI												
r	0	1	1	2	3	4	5	6	7	7	8	9
r'	⊥	0	1	⊥	⊥	2	3	⊥	4	5	⊥	6
i	16	17	2	14	4	1	13	7	10	5	11	8
P	0	1	2	3	4	5	6	7	8	9	10	11

compute unique sample ranks in P by sorting RRI by (r, r')

i	1	2	4	5	7	10	11	13	14	16	17
P	5	2	4	9	7	8	10	6	3	0	1

P with unique sample ranks in text-order

Table 5.7.: Example of discarding in Phase 3. Unique ranks are colored in red and unique ranks that are required to determine not-unique ranks are marked with an arrow. Unused tie-break ranks r' are colored in gray.

5.6. Space-Efficient Sorting Using Bucketing

A severe disadvantage of distributed DCX is its high memory consumption of $O(nX)$. In the sequential setting, the X-prefixes can be represented space-efficiently by a pointer to the starting position of the suffix. However, in distributed memory, it is necessary to materialize multiple characters and ranks for each suffix. A simple idea to reduce the memory overhead, is to partition the suffixes into buckets using global splitter elements, and process one bucket at a time. Using q buckets, we can reduce the memory overhead of materialization by a factor of q. This space-efficient sorting technique, we call *bucketing*, was proposed in previous work on distributed string sorting [55, 68].

In the following, we describe this general technique, which we use as a building block in our distributed variant of DCX. The space-efficient representation in our setting is a sequence of elements Q. Each element is represented by its starting position in the array

Algorithm 4: Distributed DCX Phase 3 with Discarding.

sort P by $(i \bmod X, i \operatorname{div} X)$
$P_r = \langle c \mid (c, i) \in P \rangle$ // extract sample-ranks
$B = \langle P_r[0] \text{ is not unique} \rangle$ // bitmask of positions to keep
for $1 \leq k < |P_r|$ **do**
  $B[i] = \text{not } (P_r[i] \text{ and } P_r[i-1] \text{ unique})$
$T_d = \langle P_r[i] \mid 0 \leq i < |P_r| \in P \text{ if } B[i] \rangle$ // reduced text
$SA_d = DCX(T_d)$ // recursively compute SA_d
$P_d = \langle (i, SA_d[i] \mid 0 \leq i < |SA_d| \rangle$
sort P_d by the second component // ranks in text-order
$R_d = \langle r \mid (r, i) \in P_d \rangle$ // extract tie-break ranks
$k = 0$
$RRI = \langle \rangle$ // (sample rank, tie-break rank, global index)
for $0 \leq j < |P|$ **do**
  $(r, i) = P[j]$ // sample ranks and global index
  $r' = R_d[k++] \text{ if } B[j] \text{ else } \perp$ // tie-break-rank, if exists
  $RRI[j] = (r, r', i)$
sort RRI by the first two components
$P = \langle (j, i) \mid 0 \leq j < |RRI|, (r, r', i) = RRI[j] \rangle$
sort P by the second component // sample ranks in text-order

and a length attribute. For example, in Phase 1 of DCX, the elements to be sorted are the X-prefixes of the difference cover samples. Algorithm 5 outlines the procedure.

Bucketing. In the first step, we determine global splitter elements $-\infty = s_0 < s_1 < \cdots < s_{q-1}$. One way to do this is by sampling m elements uniformly at random, globally sort them, select every $\lfloor m/q \rfloor$-th element as a splitter and communicate the splitters to all PEs. This induces a partition of the elements Q into q buckets $Q_k = \{x \in Q \mid s_k < x \leq s_{k+1}\}$ for $k \in [0, q)$. Ideally, the bucket sizes are of similar size $|Q_k| \approx n/q$ and the elements within each bucket are distributed equally among the PEs. In our experiments, we can empirically confirm the first assumption. However, the second does not necessarily hold. We counteract this with a new load-balancing technique called *chunking* (see Section 5.7).

Now, we execute q global sorting steps. In each step, we materialize the current bucket Q_k, globally sort the elements, store the order of elements (in our case the starting positions in the array) into B_k. Finally, we append the elements of B_k into a single vector and record its bucket size into S. Later, we require the bucket sizes S to identify the regions B_k in B. After q sorting steps, B is not yet globally sorted and has to be rearranged between the PEs. Locally on a PE $j \in [0, p)$, the vector B consists of up to q regions, $B_0^j, B_1^j \ldots B_{q-1}^j$. Let $B'_k = B_k^0 \odot B_k^1 \odot \cdots \odot B_k^{p-1}$ for $k \in [0, q)$ be the sorted bucket of the k-th sorting step. The sorted sequence is given by $B'_0 \odot B'_1 \odot \cdots \odot B'_{q-1}$. Table 5.8 shows an example of bucketing of 3-prefixes with 3 buckets and 3 PEs.

	PE 0							PE 1							PE 2				
i	0	1	2	3	4			5	6	7	8	9			10	11	12	13	14
T	b	a	a	a	b	a	a	a	a	b	a	a	a	b	a	b	$	$	$

input text T with padding and shifts

Q_1	(1, aaa)	(2, aab)	(5, aab)	(8, aaa)	(9, aab)			
Q_2	(3, aba)		(6, aba)				(10, ab$)	
Q_3	(0, baa)	(4, baa)	(7, baa)				(11, b$$)	

partition of 3-prefixes into buckets Q_1, Q_2, Q_3 induced by splitters s_1 = ab$, s_2 = b$$

B_1	1	8	2	5		9		
B_2	10		3			6		
B_3	11	0	4			7		

order B_1, B_2, B_3 of 3-prefixes in Q_1, Q_2, Q_3 after sorting

1	8	2	5	9	10	3	6	11	0	4	7

order of 3-prefixes after rearranging

Table 5.8.: Example bucketing of 3-prefixes with 3 PEs and 3 buckets.

Rearranging the Buckets. Equally rearranging the buckets can be done with two alltoall communications and some local reordering afterwards. We use the local array B directly as the send buffer and compute the send counts of PE $j \in [0, p)$ in a left to right scan over the buckets $B_0^j, B_1^j, \ldots B_{q-1}^j$. Let $g_k = \sum_{j<p} |B_k^j|$ the global size of bucket $k \in [0, q)$ and let $t_j \in \{\lfloor n/p \rfloor, \lceil n/p \rceil\}$ be the target size of the receive buffer on PE j. Initially, we set the current target PE r to 0. Then, we iterate over the buckets B_k^j from left to right. Bucket B_k^j on PE j is processed as follows.

1. Set the remaining bucket size z to $|B_k^j|$.
2. Compute the global index i of the first entry of B_k^j via prefix sums
$$i = g_0 + g_1 + \cdots + g_{k-1} + |B_k^0| + |B_k^2| + \ldots |B_k^{j-1}|.$$
3. If $r < p$ and $z > 0$, increment the send counts of PE r by
$$x = \min(y, z),$$
$$\text{where } y = \max(t'_r - i, 0) \text{ and } t'_r = t_0 + t_1 + \cdots + t_r.$$
Update $z = z - x$, $i = i + x$.
4. Increment r if $z > 0$ and continue with Step 3. until $r = p$ or $z = 0$.

Algorithm 5: Bucketing Technique

```
1  Function Bucketing(Q, q):
2      determine global splitter elements −∞ = s_0 < s_1 < ··· < s_{q−1}
3      B = ⟨⟩
4      S = ⟨⟩
5      for 0 ≤ k < q do
6          materialize elements x ∈ Q with s_k < x ≤ s_{k+1} into Q_k
7          globally sort Q_k
8          store order of Q_k into B_k
9          append B_k to B
10         append |B_k| to S
11     return Rearrange (B, S)
```

In Step 2., $g_0 + g_1 + \cdots + g_{k-1}$ is the global index of the bucket k in the sorted order of buckets and $|B_k^0| + |B_k^2| + \ldots |B_k^{j-1}|$ offsets the index to the beginning of B_k^j. Then, in Step 3., we clamp z with the remaining capacity of elements y on target PE r, and update z and the global index i accordingly. If $z = 0$ in Step 4, we do not increment r, since PE r might receives some elements of the next block $k + 1$.

Now, we exchange B using an irregular alltoall communication with the computed send counts. However, the SA entries do not arrive in sorted order, since the received elements on a PE are sorted by the sending PE first. Let's examine a fixed PE, where C_k^j is the received part of the bucket B_k^j of the sending PE j. For 3 PEs with 3 Buckets, the situation looks as follows:

C_0^0	C_1^0	C_2^0	C_0^1	C_1^1	C_2^1	C_0^2	C_1^2	C_2^2

Depending on the choice of q and the bucket sizes, many C_k^j might be empty. To reorder the receiver buffer, we allocate another buffer into which we copy the parts C_k^j by bucket number k and then by PE number j. In addition to C_k^j, each PE sends $|C_k^j|$ in another

C_0^0	C_0^1	C_0^2	C_1^0	C_1^1	C_1^2	C_2^0	C_2^1	C_2^2

alltoall communication to be able to identify the regions in the receive buffer.

Bucketing in Phase 1 and Phase 2. To apply bucketing in Phase 1, we have to interleave the sorting with the naming procedure of Phase 2. We require the X-prefix of the difference cover samples in text-order and compare adjacent elements to determine $\Delta_j = S[j] \neq S[j+1]$, $j \in [0, |S| - 1)$. Instead of storing the order of the X-prefixes, like in the general formulation, we compute Δ_j within the current bucket. Since we only access adjacent elements in the sorting, we can process each bucket separately. It is only necessary to keep track of the last X-prefix of the previous bucket to compare it with the first element of the current bucket. This requires a single point-to-point communication between the

first and the last PE. After the rearranging procedure, the ranks are computed as before based on Δ_j.

In addition to the input text, we store the ranks and indices of the sample suffixes in P, one fully materialized bucket of X-prefixes with an index, and a receive buffer for sorting a bucket of X-prefixes. In total, this yields a memory consumption of

$$2\lambda^{r+1}n(w_r + (Xw_c + w_r)/q).$$

Depending on the choice of X and q, the alltoall communication, which requires a receive buffer for P, might take more memory than the bucketing

$$4\lambda^{r+1}nw_r.$$

Bucketing in Phase 4. Bucketing can be directly applied to sorting the vector M in Phase 4. However, in this case, the space-efficient representation consists of two arrays, the text T and the sample ranks R, to materialize the triple consisting of a $(X-1)$-prefix of the suffix, a $|D_x|$-prefix of sample ranks and the global suffix starting position. After sorting a bucket, we extract the starting position of the suffix, which corresponds to the final SA entry.

We store the rank vector R, the full suffix array SA and one fully materialized bucket of triples with a receive buffer of the same size. This requires

$$\begin{aligned}&\lambda^{r+1}nw_r + \lambda^r nw_r + 2(\lambda^r n \cdot ((X-1)\cdot w_c + (|D_X|+1)\cdot w_r))/q\\ =&\lambda^r n(w_r(1+\lambda) + 2((X-1)\cdot w_c + (|D_X|+1)\cdot w_r)/q)\end{aligned}$$

of memory in addition to the text. Rearranging the suffix array in the last step requires

$$2\lambda^r nw_r$$

memory.

5.7. Chunking

If we take a sufficiently large sample to determine the global splitter elements, the bucket sizes $|B_k|$, $k \in [0, q)$, are approximately of the same size n/q. However, the suffixes within a bucket might not be distributed equally among the PEs due to the input text. For example, consider a text in which the suffixes are already in lexicographic order with $q < p$ buckets. In this setting, the first PE has to materialize n/p elements when processing the first bucket, while the last PE materializes no elements. This results in poor load-balancing and in high memory consumption. In our experiments, we observe imbalances between 170% and 376% caused by this effect on real world texts of 15.36 GB using 768 PEs and 128 buckets.

A standard technique to deal with this problems is to *randomly redistribute* the elements to be sorted. Yet, this is not directly possible for suffixes stored in a space-efficient manner, since the compressed representations overlap. Therefore, we propose to randomly redistribute whole *chunks* of the text instead of single elements.

Let Y' be the local array that stores the elements Y in a space-efficient way. Formally, each PE divides its local array into C chunks of consecutive data

$$Y' = X_1 \cup X_2 \cup \cdots \cup X_C.$$

Each element $x \in Y$ belongs to exactly one chunk. Depending on the elements, the chunks have to overlap to be able to materialize the elements at the edge of the chunk. For example, when using chunking in Phase 4 of DCX, there is an overlap $X - 2$ characters for the $X - 1$-prefixes of suffixes and an overlap of $|D_X| - 1$ ranks for the $|D_X|$-tuple of ranks. Each chunk is send to a PE uniformly at random. We concatenate the local chunks into a single send buffer and exchange them in an alltoall communication. Table 5.9 shows an example of random redistribution with none-overlapping chunks.

	PE 0				PE 1				PE 2			
Y'	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}
target PE	2	1	1	3	1	3	2	2	2	3	1	1
send buffer	X_2	X_3	X_1	X_4	X_5	X_7	X_8	X_6	X_{11}	X_{12}	X_9	X_{10}

	PE 0					PE 1				PE 2		
receive buffer	X_2	X_3	X_5	X_{11}	X_{12}	X_1	X_7	X_8	X_9	X_4	X_6	X_{10}

Table 5.9.: Example random redistribution with none-overlapping chunks. Send/receive buffers are sorted by sending/receiving PE and then by their original order.

In addition to the information required to reconstruct an element, we also send bookkeeping information. We send the size of a chunk, to identify each chunk in the receive buffer. Further, we send the global index of the start of the chunk to be able to identify the elements in the global array.

In a sorting step of bucketing, a PE iterates over all received chunks instead of its local array and materializes all elements of the current bucket. Here, a suffix is not identified by its global index, but by its local position in the chunk and global starting position of the chunk. Other than that, bucketing works the same as described before.

Additionally, we proved the following theorem on the probabilistic guarantees of random chunk redistribution. The proof can be found in Appendix A.1 and in [40].

Theorem 2 (Random Chunk Redistribution [40])**.** *When redistributing chunks of size c uniformly at random across p PEs, with q 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 $\gamma > 0$.*

5.8. Further Optimizations

In distributed DCX, we require distributed sorting in various parts of the algorithm. There are three types of sequence that are sorted.

1. Tuples of integers to sort ranks and global indices (Line 6, Line 9).
2. X-prefixes of suffixes paired with a global index to sort the sample suffixes (Line 3).
3. Tuples of the X-prefixes of suffixes, D rank entries and a global index to sort all suffixes (Line 12).

All types of data can be sorted with comparison-based sorters. Alternatively, for Type 2 and Type 3 it might be beneficial to consider using string sorting, to exploit the structure of the keys. A third option is to pack the characters of the X-prefix of Type 2 and Type 3 into computer words to accelearate comparison-based sorting. We will discuss the string sorting and packing optimization in Section 5.8.1 and Section 5.8.2

5.8.1. Incorporating String Sorting

Specialized algorithms for the problem of string sorting have been shown to be more efficient than comparison-based sorting algorithms [12]. Usually it is assumed that the input strings have arbitrary length. However, in distributed DCX we sort short strings of constant size (Line 3 and Line 12). A natural questions is whether in this case straightforward atomic sorting or string sorting technique performs better.

To answer this question, we implemented our own prototypical version of (single-level) sample sort, where local sorting is performed with highly-tuned sequential string sorters [13] and k-way merging uses LCP-aware Loser Trees [15]. Optionally, LCP-compression and prefix doubling optimization can be activated to reduce the communication volume of exchanged characters [17].

Globally sorting all suffixes in Line 12 is a hybrid between string sorting and comparison-based sorting, since the order of equal X-prefix is determined by comparing the sample ranks. We implemented two ways to incorporate this tie-breaking mechanism into string sorting.

Tie-Breaking Afterwards. A simple way to perform tie-breaking is to run two separate sorting phases. First, the X-prefixes are sorted using string sample sort and then each interval of equal X-prefixes is sorted by their sample ranks. To avoid intervals that span over multiple processes, we send equal strings to the same PE. This can lead to severe imbalances between the PEs, which we also observed on real data in our preliminary testing. Alternatively, overlapping intervals can be sorted using multiple parallel sorting processes similar to how the authors of PSAC [32] perform bucket sorting for each h-group in parallel. However, we chose to include the tie-breaking process directly into string sample sort.

Tie-Breaking in String Sample Sort. In this variant, we already perform tie-breaking after local sorting and use the full comparison function when computing splitter elements and partitioning the elements. We perform one final round of local tie-breaking after k-way merging, since Loser Trees do not directly support a tie-breaking mechanism.

5.8.2. Packing

Packing is an optimization proposed by Flick et al. [32] for distributed prefix doubling exploiting small-sized alphabets. In the first prefix doubling iteration, instead of sorting the suffixes by the first character, they sort them by the first k characters and pack k characters into a single 64-bit word, where k is as large as possible. For example, DNA data uses only 4 characters. Another character is required to encode the padding. Therefore, a single character can be encoded with 3-bits and prefix doubling can be started with a prefix length of $k = 21$.

In the DCX algorithm, packing can be utilized when sorting the X-prefixes. However, we can only exploit the small alphabet on level 0, because on deeper levels the characters are ranks that have much greater range. There are three ways we can benefit from packing.

Cheaper Comparison. Usually, each character is stored in a single byte and string comparison are done byte-by-byte. However, if we pack multiple characters into 16, 32 or 64-bit words, we can compare multiple characters at once with a single machine instruction. For example, consider DC21 with the usual alphabet size of 8-bit per character. The 21 characters can be packed into three 64-bit words, reducing the maximum number of comparison from 21 to 3. This slightly increase the memory usage to 24 bytes instead of 21 bytes per X-prefix to be sorted.

Less Memory. If the alphabet can be encoded with less than 8-bits, we can store the X-prefixes utilizing less memory. A 4-bit alphabet can be stored using only half of the bytes. This can also be combined with packing into larger words, taking advantage of cheaper comparisons.

More Unique Ranks. Instead of packing X characters into less bytes, we can also pack more characters than necessary. This preserves correctness and yields more unique ranks in the naming phase, which allows us to discard more ranks in Phase 3. Consider for example DC21 with 4-bits per character. We can pack 32 characters into two 64-bit words, which takes 16 bytes, instead of the 21 bytes without packing. Additionally, it only takes two machine instructions to compare all characters.

5.9. Implementation Details

5.9.1. Static String Containers

Generally strings are assumed to have different sizes and are stored with a pointer to a character array containing the concatenated set of strings. In DCX, the size of the X-prefix is fixed and know at compile time. Thus, we can simply statically allocate containers to store X-prefixes. This facilitates exchanging strings in MPI, since we can treat strings as atomic objects. Additionally, we can directly sort X-prefixes using comparison-based sorters like AMS.

5.9.2. Smaller Data Types for Ranks

Another optimization is to choose smaller data types to store ranks and indices. This reduces the memory consumption and the total amount of bytes that have to be communicated between PEs. Instead of using 64-bit words, like for example PSAC [32], we use a 40-bit integer that requires 37.5% less memory. The 40-bit integer is implemented as a pair of 32-bit and 8-bit integer. This allows us to process texts of sizes up to 1 TB.

One can take this one step further and choose the smallest data type that can still fit the largest rank computed in the naming phase. Using template programming in C++, this can be realized at the cost of longer compilation times. The largest rank depends on the alphabet size of the input and the period length X used. For example, consider a DNA dataset with 4 characters and DC3. Then, there are at most $4^3 = 2^6 = 64$ unique ranks on level 1 and at most $64^3 = 2^{18} = 262144$ unique ranks on level 2. Therefore, on level 1 we could use 8-bit integers and on level 2 possibly 16-bit integers, if less ranks are present, or 32-bit integers. However, this optimization works only for small X and small alphabets. The improvements in running time we observed were small. Since we generally, use larger period lengths X, we deactivated this optimization.

5.9.3. Bucketing

Here, we want to outline three details we found important when implementing bucketing space-efficiently for Phase 4 of DCX.

Estimating the Output Size. As a result of the k-th sorting step in bucketing we get the SA entries B_k of the suffixes in the current bucket. These entries are stored into a single vector B. We can not exactly know $|B|$ after bucketing in advance, since $|B_k|$ depends on the imbalance that the distributed sorter introduces. To avoid unnecessary reallocation and too much wasted memory, we allocate $(n/p) \cdot 1.03$ entries for B locally on each PE. In practice, the imbalance of B is close to the average size n/p. This is because usually the imbalances of B_k cancel each other out when using a sufficiently large number of buckets. Distributed sorter usually try to keep the imbalance low or can give imbalance guarantees like AMS. Should the preallocated space be not enough during bucketing, we allocate just enough memory to store the next bucket. Otherwise, the default vector reallocation policy would double the vector capacity.

Mapping the Bucket IDs. In the k-th sorting step, we have to efficiently determine the suffixes to be materialized. To avoid repeated comparison of suffixes with the splitters, we do the comparisons once before bucketing to determine the bucket sizes and the mapping of suffixes to buckets IDs and use binary search to find the bucket for each suffix. A straightforward approach to store the mapping is to concatenate the suffix-IDs of the k-th bucket for $0 \leq k < q$ into a single vector C_1. When materializing the k-th bucket, we scan the part of C_1 belonging to the current bucket. This however, requires $w_r \cdot n = 5n$ additional bytes. Instead, we store the mapping in a vector with $C_2[i] = b_i$, where $0 \leq i < n$ and b_i is the bucket-ID of suffix i. In the k-th materialization step, we scan the complete vector C_2 and materialize all entries i with $C_2[i] = k$. This variant only requires n additional bytes,

if we assume $q < 256$. In our implementation, we use $2n$ bytes and allow $q < 2^{16} = 65536$. The downside is that C_2 has to be scanned q times.

Storing the Suffix-IDs in-place. Increasing the number of buckets q, the previous method using C_2 spends significant amount of time in materializing the suffixes. However, we can store the suffix-IDs within the SA entries of B and use the same method we described for C_1. In the beginning, B is equivalent to the aforementioned vector C_1. Suffixes are materialized as if we would use C_1. After materializing the k-th bucket, we can use the space previously occupied by the suffix-IDs of bucket k to store the SA entries.

One has to make sure that the SA entries never overwrite the suffix-IDs of later buckets. First of all, we allocate B with 25% extra space instead of 3%. The first 25% of B are empty and serve as an extra buffer between the SA entries and suffix-IDs. Secondly, this techniques assumes that the distribution of Q_k across the PEs is well-balanced before and after sorting. Consider for example the extreme case, in which all elements of Q_k are located on a single PE j. The PEs $i \neq j$ do not free any suffix-IDs, but all PEs write approximately n/pq SA entries into B. As mentioned earlier, the distributed sorter used can usually guarantee a small imbalance after sorting. Using our Chunking technique, we can guarantee a good balance before sorting with high probability. Should the regions overlap, we have to restart bucketing and use the previous method as a fallback. In our experiments, we never had to use the fallback method when we use 10^4 chunks per PE.

When combining Chunking with the in-place storage management, we do not store the suffix-ID to identify a suffix, but its chunk-ID and its local position in its chunk. We pack both numbers into the first and second 20 bits respectively of the 40-bit integer that is used to store a SA entry. This limits us to use at most $2^{20} = 1048576$ chunks and elements per chunk, which is reasonable for the input sizes we process. For more flexibility in the configuration of chunking, this packing could be improved by selecting the number of bits used for the number of chunks and the chunk size at run time.

5.10. Suffix Array Checking Algorithm

To ensure the correctness our algorithms, a fast distributed suffix array checker is required. A straightforward comparison of suffixes in suffix array order is infeasible, since each comparison requires linear time in the worst case. However, there is simple sequential algorithm that only requires scans and sorting routines. We adapt it to the distributed setting. It is based on the following lemma.

Lemma 5 ([21, 25])**.** An array $\mathrm{SA}[0, n)$ is the suffix array of a text T if and only if the following conditions are satisfied:

1. SA contains a permutation of $[0, n)$.
2. $\forall i, j : r_i \leq r_j \Leftrightarrow (T[i], r_{i+1}) \leq (T[j], r_{j+1})$ where r_i denotes the rank of the suffix S_i according to the suffix array.

Proof. Clearly, both conditions are necessary. We show that they are also sufficient. By way of contradiction, assume there exists a pair $i \neq j$ that violates the second condition.

That is $r_i < r_j$, but $(T[i], r_{i+1}) > (T[j], r_{j+1})$. If $T[i] > T[j]$ holds, this contradicts the fact $S_i < S_j$. Otherwise, $T[i] = T[j]$ and $r_{i+1} > r_{j+1}$. Since the first characters are equal and suffix are prefix-free $S_i < S_j \Leftrightarrow S_{i+1} < S_{j+1}$, which is a contradiction to $r_{i+1} > r_{j+1}$. □

Distributed Checker. Now, we describe how to implement the checker in distributed memory. Algorithm 6 shows the pseudocode for the distributed checker. To check the first condition, we create a vector P of tuples $\langle i, SA[i] \rangle$ for $i \in [0, n)$, sort it by the second component and check that the second component is equal to $\langle 0, 1, \ldots, n-1 \rangle$ (Line 2 - Line 5). Each PE builds a slice of P based on its local array SA. Converting from a local index i of a distributed array X to a global index is done by adding the result of $exscan(|X|)$ to i.

Now, we check the second condition. The first component of P contains the ranks $R = \langle r_0, r_1, \ldots r_{n-1} \rangle$ of each suffix, i.e. the ISA. First, we build a vector V of triples $\langle r_i, r_{i+1}, T[i] \rangle$ for $i \in [0, n)$. We make all required data available to each PE by aligning R and T between the PEs and shift the first element of PE $i+1 < p$ to PE i. Secondly, we sort V by the first component and check that V is sorted lexicographically by (c, r_2), where $(r_1, r_2, c) \in V$ (Line 7 - Line 11). For the final check, we require the last triple of PE $i-1$ (if $i > 0$) and the first triple of PE $i+1$ (if $i < p-1$) to check triples at the edges.

A straightforward implementation requires in addition to the input text T and SA 2 bytes and 4 machine words per input character to store to store V and a receive buffer for V. Assuming 5 bytes per machine word, the checker requires 22 bytes additional bytes per input character.

Algorithm 6: Distributed Suffix Array Checking.

```
1  Function SA_Check(T, SA):
2      P = ⟨(i, SA[i]) | 0 ≤ i < n⟩                    // zip SA entries with ranks
3      sort P by the second component
4      J = ⟨j | (r, j) ∈ P⟩
5      check J = ⟨0, 1, 2, ... n − 1⟩                  // check if SA is a permutation
6
7      R = ⟨r | (r, j) ∈ P⟩                                          // R[i] = r_i
8      V = ⟨(R[i], R[i + 1], T[i]) | 0 ≤ i < n − 1⟩
9      sort V by the first component
10     check that V is sorted by:
11     (c, r_2) where (r_1, r_2, c) ∈ V                  // check second condition
```

6. Experimental Evaluation

In this chapter, we conduct an experimental evaluation of our distributed DCX implementation. We explain our experimental setup in Section 6.1. Then, we show our results consisting of two parts. First in Section 6.2, we evaluate the influence of various configurations of distributed DCX on time and memory usage. Second in Section 6.3, we compare our best configuration with the current state-of-the-art distributed suffix array algorithm.

6.1. Experimental Setup

We implemented distributed DCX in `C++20` using MPI for interprocess communication and the (zero-overhead) MPI Wrapper *KaMPIng*[1] [91]. Our code is compiled with `IntelMPI 2021.11` and `gcc 12.2.0` using the optimization flags `-O3 -march=native`. The implementation is available in a public repository[2]. In addition, we set the following environment variables for lowering the memory footprint of MPI [1].

```
// size of forward cells
export I_MPI_SHM_CELL_FWD_NUM=0

// total number of extended cells per computational node
export I_MPI_SHM_CELL_EXT_NUM_TOTAL=0

// size of backward cells
export I_MPI_SHM_CELL_BWD_SIZE=65536

// number of backward cells per rank
export I_MPI_SHM_CELL_BWD_NUM=64

// disable Intel MPI custom allocator of private memory
export I_MPI_MALLOC=0

// disable Intel MPI custom allocator of shared memory
export I_MPI_SHM_HEAP_VSIZE=0
```

Machine. We perform our experiments on up to 128 compute nodes of SuperMUC-NG. Each node consists of an Intel Skylake Xeon Platinum 8174 processor with 48 cores and 96GB of main memory. The nodes are connected with an OmniPath network of 100 Gbit/s.

[1] `https://github.com/kamping-site/kamping`
[2] `https://github.com/HaagManuel/distributed_suffix_sorting`

We also run a sequential and shared-memory suffix array algorithm. The RAM of a single node of SuperMUC-NG limits their application to inputs up to 10 GB. To process larger inputs, we use a second machine, we refer to as MACHINE B. It has an AMD EPYC 9684X 96-Core processor clocked at 3.715 GHz with 2 Threads per core, 1.5 TB RAM, L1, L2 and L3 caches of 3 MiB, 96 MiB and 1.1 GiB respectively.

Inputs. To evaluate our algorithms, we use four real-world instances. The inputs consist of fixed-sized prefixes of these text.

- **CommonCrawl (CC).** WET files that only contain the plain text of crawled web pages. HTML code, images and other media are excluded. Additionally, we removed meta information added by the CommonCrawl corpus. The files can be downloaded here: `https://data.commoncrawl.org/crawl-data/CC-MAIN-2019-09/index.html`
- **DNA Data (DNA).** FASTQ files from the 1000 Genomes project, where we only kept the raw sequence consisting of the letters `A, C, G` and `T`. The data can be accessed here: `https://www.internationalgenome.org/`
- **Protein Data (Protein).** FASTA files from the Universal Protein Resource. We removed all lines not containing sequence data. The files can be downloaded here: `https://ftp.uniprot.org/pub/databases/uniprot/current_release/uniparc/fasta/active/`
- **Wikipedia (Wiki).** Current version of each article on Wikipedia in multiple languages stored in XML-format. The files are available at `https://dumps.wikimedia.org/mirrors.html`.

Table 6.1 shows basic LCP statistics and Table 6.2 quantiles of the LCP-values of our inputs. Higher LCP-values indicate that the algorithms have to examine longer parts of the suffixes to determine their sorted order.

Name	Mean	SD	Max	Alphabet Size
CC	10396.42	50428.5	1838814	243
DNA	25.17	17.89	3570	4
Proteins	179.31	644.43	34340	26
Wiki	396.07	9342.32	1582472	213

Table 6.1.: LCP statistics and alphabet size of our texts. LCPs were computed on the first 50 GB of the data.

Measurements. We measure the wall-clock time of each algorithm. The timing starts as soon as the local parts of the text are available in RAM. To measure the memory-footprint, we record the maximum resident size on each PE (*PE-MaxRss*) after the algorithms termination. Since PEs of the same node share the RAM, for each node, we sum up the PE-MaxRss of its corresponding PEs (*node-MaxRss*). Further, we define the *node-blowup* as

Name	$q_{0.1}$	$q_{0.2}$	$q_{0.3}$	$q_{0.4}$	$q_{0.5}$	$q_{0.6}$	$q_{0.7}$	$q_{0.8}$	$q_{0.9}$	$q_{0.95}$	$q_{0.99}$
CC	10	14	20	33	124	484	1350	3677	15010	45922	208833
DNA	16	17	18	19	20	22	26	31	37	47	98
Proteins	7	8	9	10	11	18	33	73	258	924	3674
Wiki	10	13	16	19	22	28	40	70	184	490	3013

Table 6.2.: Quantiles of LCPs of our texts. LCPs were computed on first 50 GB of the data.

the node-MaxRss divided by the total size of the input of its PEs. By *blowup* we refer to the maximum node-blowup aggregated over all nodes. The blowup measures how many bytes per input character are required by the algorithm.

Additionally, we measure imbalances of distributed vectors. Let m be the total number of elements in a distributed vector, m_i for $0 \leq i < p$ the local number of elements on PE i and $m_{avg} = m/p$ the average number of elements. We define the *imbalance* of the distributed vector as $\max_{0 \leq i < p}(m_i/m_{avg}) - 1$. In our experiments, we report the arithmetic mean of 3 runs with different seeds. Each PE adds its rank to the global seed to avoid using the same source of randomness for all PEs.

6.2. Evaluating Distributed DCX

We use the IPS40[3] algorithm [8] for local and AMS[4] [6, 7] for distributed comparison-based sorting. For sorting X-prefixes (Line 3 and Line 12 in Algorithm 3), we can either use comparison-based sorting or string sorting. The remaining sorting routines to sort tuples of integers (Line 6 and Line 9 in Algorithm 3) use comparison-based sorting. AMS is configured to use two levels and otherwise uses default parameters, which guarantee an imbalance of at most 10% of the distributed vector after sorting.

For distributed string sorting, we implemented our own prototypical version of (single-level)-sample sort (see Section 5.8.1). We were not able to easily include already existing distributed string sorters[5] [55, 68], since we also require the sample ranks in the comparison of suffixes.

On the first level of DCX, we use MSD RadixSortCE3[6] for local sorting and on later levels MultiKeyQuickSort[7]. Bingmann [12] showed in his extensive evaluation of string sorters that MSD RadixSortCE3 performs best among all string sorters tested. RadixSort is optimized for 8-bit and 16-bit alphabets and crashes on larger alphabets. Therefore, we can only use it on the first level, since on later levels the characters represents ranks in the range of the text size. We use the LCP-aware Loser Tree implementation by the authors [15].

Based on a preliminary set of experiments we set the initial configuration of DCX. If not stated otherwise, we configure DCX with a discarding threshold of 70%, $2 \cdot 10^4$

[3] https://github.com/SaschaWitt/ips4o

[4] https://github.com/MichaelAxtmann/KaDiS

[5] https://github.com/pmehnert/distributed-string-sorting/

[6] https://github.com/tlx/tlx/blob/master/tlx/sort/strings/radix_sort.hpp

[7] https://github.com/tlx/tlx/blob/master/tlx/sort/strings/multikey_quicksort.hpp

random global samples to determine bucket splitters, $2000 \cdot p$ random samples sorted with RQUICK to determine the splitter elements in our sample sort implementation. We apply the bucketing technique on level 0 and 1 with 32 buckets for Phase 1 and 128 buckets for Phase 4 and $X \geq 13$. In subsequent recursions, the input is small enough such that space-efficient sorting is not required. Since the reduction for $X = 3$ and $X = 7$ is smaller, we use bucketing on the first four and three level respectively. In Section 6.2.5, we fine-tune these parameters for concrete difference covers. We enable chunking in the bucketing technique and use equal chunk sizes. Let n_j be the number of locals characters on PE $j < p$ and n the number of global characters when we want to use the bucketing technique. To configure chunking, we set the average number of chunks C, clamp C between 1 and $\min_{j<p} n_j$ and set the global chunk size to (n/pC). Assuming a global chunk size facilitates the implementation. We configure the average number of chunks on a PE rather than chunk size directly, to adapt the chunk size to different sized inputs. If the characters are distributed equally, each PE uses exactly C chunks.

For this part of the experiments, we run our algorithm on 16 nodes with a total of 768 PEs and 15.36 GB of text input (20 MB per PE). The difference covers we use are shown in Table 5.1.

First, we will evaluate the influence of the different optimizations for distributed string such as LCP-aware Loser Tress, LCP-compression and distinguishing prefix approximation (Section 6.2.1). We evaluate comparison and string based sorters for sorting the X-prefixes in Phase 1 and Phase 4 (Section 6.2.2). Then, we show the benefit of the discarding technique (Section 6.2.3) and fine-tune the number of chunks and bucket sizes (Section 6.2.4 and Section 6.2.5).

6.2.1. String Sorting Variants

LCP-aware k-way merging with Loser Trees (LT) and reduction of communication volume by LCP-compression (LC) and a distinguishing prefix approximation (DP) are key components in state-of-the-art distributed string sorters [17, 55]. In this experiment, we apply four variants of string sample sort to sort X-prefixes in Phase 1 and Phase 4 of DCX with varying values of X. All other distributed sorting routines use AMS.

- STRING-SORT: String sample sort without additional optimizations. Local merging is performed with a local sort.
- STRING-SORT-LT: STRING-SORT with a LCP-aware Loser Tree to merge received strings.
- STRING-SORT-LC: STRING-SORT with LCP-compression.
- STRING-SORT-LC-DP: STRING-SORT-LC with distinguishing prefix approximation.

Table 6.3 shows the reduction in communication volume and the average length of the distinguishing prefix for DC133 in Phase 4 on level 0. We show the values for DC133, since here the effects of the reduction can be demonstrated the best. On deeper levels, the LCP-value are very low (between 0 and 2) and LC would actually send more data than the standard variant. For this reason, we deactivate LC if the reduction of characters is

less than 5%. LC reduces the total communication volume by about 6% to 9% and LC-DP between 30% and 55%.

Figure 6.1 shows the total time and the blowup of DCX with varying values of X and the four string sorter configurations. In terms of running time, STRING-SORT is the fastest on $X = 3, 7$, being on average 7% faster STRING-SORT-LT. Here, the strings are very short, such that the LCP optimizations can not take effect. There is a sweet spot around $X = 13, 21, 31$, where STRING-SORT and STRING-SORT-LT achieve the best running times of all values of X, while STRING-SORT-LC is 6% slower. For $X \geq 39$, STRING-SORT-LT is on average 23% faster than the second best variant STRING-SORT. On longer X-prefix, merging with the LT is much more efficient than locally sorting the received strings.

Across all values of X, LC and DP slow down the algorithm significantly with greater difference for larger values of X. LC and DP create an additional overhead for copying the text that should be communicated into a separate send buffer and copying back the results into the static data types. DP requires additional time for the distinguishing prefix approximation. In Phase 4 of DCX, the $|D_X|$ sample ranks to break ties of equal X-prefixes create significant part of the communication volume, which is not affected by the reduction.

Memory consumption does not vary significantly between the variants. DCX with $X = 3$ has a higher blowup (around 33) than the remaining DCX variants (around 23). This occurs because DC3 reduces the text size by only 66.66%, while the data type for ranks is changed from 1 byte to 5 bytes. Secondly, the bucket imbalances in Phase 4 are very high ($> 80\%$ for DC3, $< 30\%$ for DC21). The bucket splitter consists only of 3 characters, consequently there can be many equal strings that are not split between the PEs. To resolve this issue, we implemented a variant that breaks ties by using the full comparison function including the sample ranks when comparing a suffix with a splitter element. However, it slowed down the other faster DCX variants significantly, therefore we deactivated tie-breaking for the splitter elements.

Overall, the computational overhead of LC and DP does not outweigh the time gained by the reduction in communication volume. LT brings some improvements in running time for larger values of X.

Input	LCP		LCP-PD		
	Char	Total	Char	Total	AvgPrefix
CC	0.10	0.07	0.45	0.30	86.68
DNA	0.09	0.06	0.82	0.55	35.33
Proteins	0.09	0.06	0.76	0.51	43.29
Wiki	0.14	0.09	0.74	0.50	52.49

Table 6.3.: Reduction of communication volume in Phase 4 on level for DC133. The Char columns shows the reduction of send characters and Total the overall reduction. The last column is the average length of the approximated distinguishing prefix.

6.2.2. Comparison-based-Sorting vs. String Sorting

In this experiment, we want to determine the best sorting variant for the X-prefixes and the best value of X. We run four configurations:

- ATOMIC-SORT: Comparison-based sorting with AMS.
- PACKED: ATOMIC-SORT and additionally characters are packed into the minimal number of 64-bit words required. Table 6.4 shows the number of packed words and characters for PACKED with various values of X.
- PACKED+: PACKED with one additional word used for packing.
- STRING-SORT-LT: String sample sort with LCP-aware Loser Tree.

Examining the running times in Figure 6.2, we see that the packed variants are the fastest and STRING-SORT-LT is faster than ATOMIC-SORT. PACKED and PACKED+ differ not significantly, except on DNA. For $X \leq 21$, PACKED+ is faster than PACKED. Instead of 21 characters, 42 characters are packed such that the reduction of the text size is much higher. DC39-PACKED uses 42 characters without the extra word and, thus it performs similar to the lower PACKED+ variants. The packed variants exploits the smaller alphabet size on DNA ($\sigma = 4$) and PROTEINS ($\sigma = 26$), causing the larger gap to STRING-SORT-LT. DC39-PACKED uses 2 and 4 words on DNA and PROTEINS instead of 5 as on CC and WIKI. The sweet spot for running time lies on medium-sized values $X = 21, 31, 39$. We choose DC39-PACKED as the default configuration from now on, since it performs the best on DNA and on the other inputs DC21, DC31, DC39 perform similarly. The memory plot is almost the same as in the previous experiment. Here, DC39-PACKED has a blowup close to 20 on all inputs, which is only 1 - 2 bytes per input character higher than the lowest achieved blowup.

Figure 6.3 shows the time spent in the different Phases of DC39-PACKED on the first two levels and on the remaining levels. Clearly, SORT ALL (Phase 4) is the most expensive. All suffixes have to be sorted at once and characters and ranks have to be materialized. For $X \geq 21$, between 70% and 80% of the total time is spent in this phase. Furthermore, we observe that for $X \geq 21$ more than 75% of the time is spend on the first level. Higher values of X lead to greater reduction in text-size. DCX for $X \geq 21$ takes between 2 and 3 levels in total to construct the SA. For $X \geq 57$, the additional overhead caused by longer X-prefix does not outweigh the greater reduction in text size. This however, depends on the characteristic of the text instances. In our evaluation, medium-sized values of X are the best choice across all inputs.

Table 6.5 shows the time spent on sorting in Phase 4 on level 0 and 1 for DC39 with different sorting variants. We observe that sorting on both levels on average makes about up 79% of the time spent in Phase 4. PACKED outperforms STRING-SORT-LT and ATOMIC-SORT significantly. On level 0, PACKED is on average 40% faster than STRING-SORT-LT in the sorting part of Phase 4 and on DNA even 49%. This performance gain arises from the more effective comparison function of PACKED, which allows to compare multiple characters at once. STRING-SORT-LT and ATOMIC-SORT take similar time for sorting on level 0, but in total STRING-SORT-LT is 17% faster than ATOMIC-SORT on level 0. On level 1, PACKED and

ATOMIC-SORT need similar time, since we use the packing optimization only on level 0. The differences come from the higher reduction in text size of PACKED. We pack more than 39 characters to utilize every bit in the packed words. STRING-SORT-LT needs between 2 and 3 seconds longer, than the other variants for sorting on level 1. The LCPs on deeper levels are quite low (between 0 and 2). Thus, comparison-based sorting only has to compare the first few characters. MULTIKEYQUICKSORT can not benefit from the low LCP-values.

Summing up, DC39-PACKED has the best trade-off between running time and memory consumption on our inputs. The majority of the time is spent on sorting all suffixes in Phase 4 on level 0.

Input	Bits per Char	Metric	period length X										
			3	7	13	21	31	39	57	73	91	95	133
DNA	3	#words	1	1	1	1	2	2	3	4	5	5	7
		#chars	21	21	21	21	42	42	63	84	105	105	147
Proteins	5	#words	1	1	2	2	3	4	5	7	8	8	12
		#chars	12	12	24	24	36	48	60	84	96	96	144
CC, Wiki	8	#words	1	1	2	3	4	5	8	10	12	12	17
		#chars	8	8	16	24	32	40	64	80	96	96	136

Table 6.4.: Number of packed 64-bit words and chars in PACKED.

Input	Algorithm	Total Time	Level 0		Level 1	
			Total	Sorting	Total	Sorting
CC	packed	54.79	31.93	24.51	8.28	5.82
	string-sort-LT	68.29	42.89	38.23	10.77	9.39
	atomic-sort	71.1	49.36	38.28	8.23	5.93
DNA	packed	35.6	24.59	18.02	0.88	0.74
	string-sort-LT	57.27	40.94	35.48	4.53	4.37
	atomic-sort	63.13	51.29	42.9	1.06	0.88
Proteins	packed	45.9	29.66	21.82	3.6	2.48
	string-sort-LT	60.23	41.08	35.93	7.04	6.42
	atomic-sort	67.32	51.1	35.81	4.23	2.84
Wiki	packed	51.97	33.08	25.35	5.26	3.66
	string-sort-LT	66.95	45.48	40.27	8.31	7.47
	atomic-sort	72.04	54.42	40.78	5.36	3.68

Table 6.5.: Time spent on sorting in Phase 4 on level 0 and 1 for DC39.

6.2.3. Discarding

Now, we demonstrate the influence of the discarding optimization. We run DCX-PACKED with a discarding threshold of 70% (DCX-DISCARD) and without discarding (DCX-NO-DISCARD). Recall, that a discarding threshold of 70% means that if the size of reduced string

Input	Text Size		
	Level 1	Level 2	Level 3
CC	8.9564%	0.3713%	0.004%
DNA	0.2872%	-	-
Proteins	3.2308%	0.12%	-
Wiki	5.2949%	0.0675%	0.0002%

Table 6.6.: Text size on level 1, 2, 3 of DC39-DISCARD. Empty cells indicate that the algorithm already finished. For DC39-NO-DISCARD, the text-sizes are $(7/39)^i$ for $i \in \{1, 2, 3\}$: 17.95%, 3.22%, 0.58% independent of the input text.

is less than 70% we apply discarding, otherwise we call the normal recursion procedure of Phase 3.

Figure 6.4 shows the time and memory usage of both variants on varying values of X. On CC and WIKI, DCX-DISCARD is 4-10% faster for $X \leq 13$ and 13-21% faster for $X > 13$. The effect is stronger for larger X-prefixes, since more samples become unique and can be discarded. On DNA and PROTEINS with $X \geq 31$, DCX-DISCARD is 25-39% faster than DCX-NO-DISCARD. These data sets have on average lower LCP-values than WIKI and CC, which leads to more unique suffixes that can be discarded. Comparing the number of packed chars of DC39-PACKED in Table 6.4 on the DNA, PROTEINS, CC and WIKI (42, 48, 40, 40) with the next lower quantile in Table 6.2 ($q_{0.9}$, $q_{0.7}$, $q_{0.4}$, $q_{0.7}$), we see that after the first level more than 70% of the ranks are already unique, except on CC only 40%.

Looking at the blowup values of WIKI, CC and PROTEINS, we observe that for $X \leq 57$ DCX-DISCARD requires 24-40% less memory. For larger X, the reduction in memory is slightly smaller. On DNA, the memory consumption is reduced by 23-33% for $X \leq 13$. For $X > 13$, the blowup is almost the same as in both variants. The memory peak in DCX-NO-DISCARD arises from level 2 of DCX. Since we only apply bucketing on level 0 and 1, on level 2 of Phase 4 all suffix are materialized at once. Table 6.6 shows the remaining text sizes of DC39-DISCARD. The remaining string size is still large enough (3.22% of the input text) that it causes the memory peak, while for DC39-DISCARD the sizes are smaller than 0.38% of the input text on all inputs. On DNA, this effect does not occur for $X \geq 21$, because all ranks are unique before level 2 is reached. The memory peak could be reduced by using bucketing also on level 2. However, discarding is always faster and does not require bucketing on deeper levels, which makes it the better choice. Discarding synergies well with the packing optimization. Packing compares longer X-prefix than non-packing variants, which creates more unique ranks that can be discarded.

6.2.4. Chunk Sizes

In this experiment, we evaluate the influence of the average number of chunks C per PE on time and memory usage. We run DC39-PACKED without chunking ($C = 0$), and $C = 10, 100, 1000, 10000, 100000$ without the in-place optimization (see Section 5.9.3). We enable the in-place optimization in a separate configuration for $C = 10000$. Recall, that we can only use this variant if the bucket imbalances are low.

Input	Metric	Chunks C						Chunks C (in-place)
		0	10	100	1000	10000	100000	10000
CC	time	66.76	81.24	70.0	64.68	64.85	65.66	**53.92**
	blowup	23.97	21.0	20.08	**19.82**	20.12	21.07	20.65
	imbalance	2.99	3.05	1.59	0.73	0.32	**0.23**	0.3
DNA	time	44.13	60.83	48.75	44.94	44.8	46.34	**36.09**
	blowup	23.33	19.98	18.96	**18.93**	19.35	20.29	20.36
	imbalance	1.7	1.75	0.69	0.35	0.27	**0.24**	0.25
Proteins	time	52.2	72.02	58.79	54.76	55.1	56.8	**45.35**
	blowup	23.23	20.26	**19.41**	19.65	19.95	21.25	20.75
	imbalance	2.42	1.69	0.63	0.37	0.24	**0.22**	0.27
Wiki	time	62.54	78.2	65.23	63.6	61.0	62.92	**51.47**
	blowup	23.62	20.63	19.85	**19.82**	20.39	21.44	21.07
	imbalance	3.76	3.14	0.83	0.37	0.32	**0.23**	0.25

Table 6.7.: Time, memory and imbalance for chunking experiment with DC39. The best values in each row are marked in bold.

Table 6.7 shows the total time, blowup and the bucket imbalances. We observe that chunking with $C = 1000, 10000, 100000$ has similar time to $C = 0$ and takes 15-20 seconds longer with $C = 10$. Thus, there is an overhead caused by copying and exchanging the chunks. However, smaller initial bucket imbalances accelerate the sorting step, since the work is distributed better among the PEs. With $C = 10000, 100000$, the imbalance is reduced to 22% to 32% while without chunking the imbalance is between 170% and 376%. For a proper choice of C, the gains of a more balanced distribution of data outweighs the overhead created by chunking and additionally reduces the blowup from around 23.5 to 19.5 bytes per input character. For $C = 10000, 100000$, the blowup increases slightly. This occurs because we send more chunks and thus more padding characters in total.

Enabling the in-place optimization with $C = 10000$ reduces the running time around 10 seconds. Now, the suffix-ID can be read continuously and we do not have to scan the whole array q times. The blowup increase slightly by 0.5-1 bytes per input character, because we allocate 25% extra space for the vector to simultaneously store suffix-IDs and SA entries. We chose the in-place variant with $C = 10000$ for our default configuration, since it has the lowest running time with very low blowup.

6.2.5. Bucket Sizes

Lastly, we fine-tune the bucket sizes of DCX-PACKED for $X = 21, 31, 39$. As we have shown, these values of X achieve the fastest running times and good memory-efficiency. The reductions of the text-size for $X = 21, 31, 39$ are 23.8%, 19.4%, 17.4% respectively. We set the buckets size in Phase 4 to $b = 4, 8, 16, 32, 64, 128, 256, 512$. The buckets sizes of Phase 1 are set to the corresponding values divided by four, since the number of difference cover samples is roughly a fourth or a fifth of the input text.

Figure 6.5 shows the resulting running times and blowups. For $b \leq 64$, the running time does not vary greatly. Using even larger block sizes slows down the algorithm. Block sizes of $b = 256$ take 4-5 seconds and $b = 512$ about 11-13 seconds longer than $b = 128$. Looking at the blowups, we observe that for after $b = 64$ the memory usage does not decrease further and stagnates around 20 bytes per input character. Thus, for our best configuration we use $b = 64$. Table 6.8 summarizes our best DCX configuration.

DCX-Configuration	
X	39
discarding threshold	70%
sorter Phase 1, Phase 4	AMS-lv2-packed
atomic-sorter	AMS-lv2
#global samples bucket splitters	$2 \cdot 10^4$
#buckets Phase 1	16 (on level 0 and level 1)
#buckets Phase 4	64 (on level 0 and level 1)
avg #chunks per PE	10^4
use in-place optimization	True

Table 6.8.: Our best configuration of DCX.

6.3. Comparison with State-of-the-Art

We compare the following distributed suffix array construction algorithms:

- **DC39** the best configuration of our DCX implementation.
- **PSAC** the distributed prefix doubling algorithm by Flick et al. [32]

The prefix doubling algorithm with discarding and distributed DivSufSort by Fischer and Kurpicz [31] computed incorrect suffix arrays on some inputs, which is why we excluded them from our evaluations. We are also aware of the distributed DCX implementation in FEMTO [28]. However, we were not able to get it to run on our system. In future work, we want to compare DCX with those implementations as well.

PSAC Variants. PSAC uses a threshold on the number non-singleton suffixes to decide when to switch to the second algorithm. The second algorithm trades more memory usage for faster running time. In the original paper [32], the authors use a threshold of 10%. However, they only evaluated PSAC on genomic datasets. To see the full potential of PSAC, we also evaluate a version of PSAC with the threshold set to 100% (PSAC+). This means, that PSAC performs only one round of prefix doubling with the first algorithm and than switches to the second one.

Internally, PSAC uses 64-bit integers for indices and ranks, while we use 40-bit integers. For a fair comparison of memory usage, we evaluate PSAC and PSAC+with 64-bit and a versions, in which we replaced all necessary 64-bit integers by 40-bit integers in the source

code. In the plots, we indicate the integer type used in the subscript. The authors use their own sample sort implementation for distributed sorting. We also tried to replace sample sort by AMS with varying number of levels, but could not improve the performance.

Libsais. LIBSAIS [44, 76, 90, 93] is a library for linear time suffix array, longest common prefix array and Burrows-Wheeler transform construction based on induced sorting algorithm. We run the single-threaded (-ST) and multi-threaded (-MT) suffix array construction algorithm of LIBSAIS[8] on a single node of SuperMUC. Due to the limited RAM of 96 GB on a SuperMUC node, we can only run LIBSAIS up to 7.68 GB (input to 8 nodes). To process larger inputs, we run LIBSAIS on MACHINE B as well (indicated by *). In the parallel version, we use the maximum number of cores available (48 cores on SuperMUC and 96 on MACHINE B) and do not use hyperthreading. LIBSAIS supports 32-bit and 64-bit integers for indices. For our input sizes, we require the 64-bit version.

6.3.1. Weak-Scaling Experiments

We perform *weak-scaling* experiments to evaluate the scalability of the distributed suffix array algorithms. The input per PE is fixed at 20 MB and we increase the number of nodes N up to 128 nodes (6144 cores in total). We can not use more input per PE, because PSAC_{64} goes out of memory for larger inputs (see breakdown test in Section 6.3.2). LIBSAIS is configured with the same total input sizes as the distributed algorithms, but we leave the number of used threads unchanged.

Figure 6.6 shows the time and memory consumption of our weak-scaling experiments. On WIKI, CC and PROTEINS, DC39 outperforms PSAC_{40}/PSAC_{64} significantly and scales better to larger number nodes. It is 1.13× - 2.4× faster on WIKI, more than 3.2× faster on CC and 1.4× - 2.2× faster on PROTEINS. For more than 16, 1 and 16 nodes respectively, DC39 overtakes PSAC+_{40}/PSAC+_{64} in terms of running time. PSAC_{64} is slightly faster than PSAC_{40} on WIKI and PROTEINS, while on CC they perform similarly. PSAC+_{64} and PSAC_{64} are the fastest on DNA and are 1.8× - 2× faster than DC39 for $N \leq 8$, 1.5× faster for $N = 16, 32$ and 1.15× faster for $N = 64, 128$. DC39 performs similar to PSAC_{40}, but is 1.2× faster for $N \geq 64$.

The difference between the 40-bit and 64-bit PSAC variant mainly arises from the initial k-mer packing. More characters can be packed with 64-bit, which allows prefix-doubling to start with a longer prefix length. On the 3-bit alphabet of DNA, this difference is especially prominent (21 packed chars for 64-bit and 13 packed chars for 40-bit). Except on DNA, PSAC_{40}/PSAC_{64} perform many rounds of expensive global sorting, while PSAC+_{40}/PSAC+_{64} only perform one initial round of global sorting before switching to the second algorithm. PSAC_{64} requires 4-7 rounds of global sorting on WIKI and PROTEINS, and 10-12 rounds on CC.

Clearly, the single-threaded and multi-threaded LIBSAIS algorithms can not scale, since they do not get additional compute resources. We observe that on the faster hardware of MACHINE B we require 4 compute nodes (192 PEs, 3.84 GB total input) to outperform LIBSAIS-MT*. On the largest input of 122.88 GB, DC39 with 128 nodes (6144 cores) takes

[8] https://github.com/IlyaGrebnov/libsais

between 50 and 80 seconds, Libsais-MT* (192 threads) around 30 minutes, Libsais-ST* (1 thread) between 95 and 115 minutes, depending on the input text. Executing Libsais-MT (48 threads) and Libsais-ST on the SuperMUC hardware is about twice as slow as on machine B.

Now, we examine the memory usage of the compared algorithms. Libsais requires around 9 bytes per input character, which is just enough to store the input text and the SA. DC39 requires around 20 bytes per input character, which slightly increase up to 25 when using more compute nodes. Approximately 4.7 bytes per input character are used after initializing MPI (94 MB) and before starting DC39. In theory, we except a blowup of approximately 9.4 for the bucketing phase and 11.0 for rearranging the SA (see Section 5.6). The deviation may be to internal allocations of MPI or a limitation of our conservative method of measuring memory. According to our measurements, the memory peak of DC39 with 64 nodes still occurs on level 0 in Phase 4 and the imbalance are not higher than for lower number of nodes (around 20%). We suppose that internally MPI may allocates more buffers when using additional PEs. Additionally, we inspected the heap allocations of individual PEs in a run with 64 nodes using heaptrack[9] and found that we allocated approximately 15 bytes per input character. As we will see in our breakdown test (Section 6.3.2), for larger inputs per PE (up to 130MB), these effects are mitigated and the blowup is on average 14.2 bytes per input character, which is closer to the theoretical optimum.

PSAC_{64} and PSAC_{40} use around 60 and 40 bytes per character on CC, Wiki and Proteins and around 53 and 40 bytes per character on DNA. From theory we expect a blowup of 48 and 30 (6 words).

On Wiki and CC, PSAC+_{64} and PSAC+_{40} require 60 bytes per input character for 1 node, which gradually increase up to 80 and 67 for 128 nodes. On Proteins, PSAC+_{64} requires less memory than PSAC+_{40} for $N \leq 32$. Again, the reasons lies in the initial k-mer packing. The alphabet of Proteins can be coded with 5-bits, thus PSAC+_{64} starts with a prefix length of 12 and PSAC+_{40} with a prefix length of 8. PSAC+_{64} has to process less non-singleton suffixes in the second algorithm, which explains the lower memory peak.

Overall, DC39 is the most memory-efficient. It uses 1.5× - 2× less memory than the next best distributed algorithm. We showed that DC39 scales well to large number cores and is the fastest on 3 out of 4 texts for inputs larger than 30.72 GB. PSAC+_{40} is slightly faster for $N \leq 16$ on Wiki and Proteins, but requires 3× and 1.5× the memory of DC39. In terms of running time, PSAC_{64} and PSAC+_{64} remain unbeaten on DNA. As a side result, we proposed alternative configurations to PSAC_{64} that offer different trade-offs between running time and memory usage.

6.3.2. Breakdown Test

Finally, we conducted a *breakdown test*. We run each distributed algorithm with 768 PEs and increase the input per PE starting from 10 MB until the program crashes. Figure 6.7 shows the throughput and memory consumption on different input sizes. DC39 can handle inputs up to 120 MB per PE (92.16 GB in total), PSAC_{64}, PSAC+_{64} and PSAC+_{40} up to 20 MB per PE (15.36 GB in total) and PSAC_{40} up to 40 MB per PE (30.72 GB in total). Depending

[9] `https://github.com/KDE/heaptrack`

on the characteristic of the input the algorithms can handle slightly larger inputs. For example, PSAC+$_{40}$ can process up to 40 MB per PE on Proteins and DNA.

When looking at the throughput, we see a decrease in efficiency of DC39 for increasing input size. For inputs larger than 110 MB, DC39 is between 10% and 15% less efficient on Wiki, CC and Proteins, while performing the same on DNA. On average, DC39 has a throughput of 365 MB per second. For comparison, Libsais-ST* has an average throughput of 20 MB per second and Libsais-MT* of 70 MB per second. Using the same hardware, Libsais-ST/Libsais-MT are about half as efficient.

The memory curve decreases with larger inputs, since the constant overheads outside of our algorithm, like additional allocations of MPI, relatively becomes less significant. On average, DC39 requires 14.2 bytes per input character if the input is larger than 100 MB per PE.

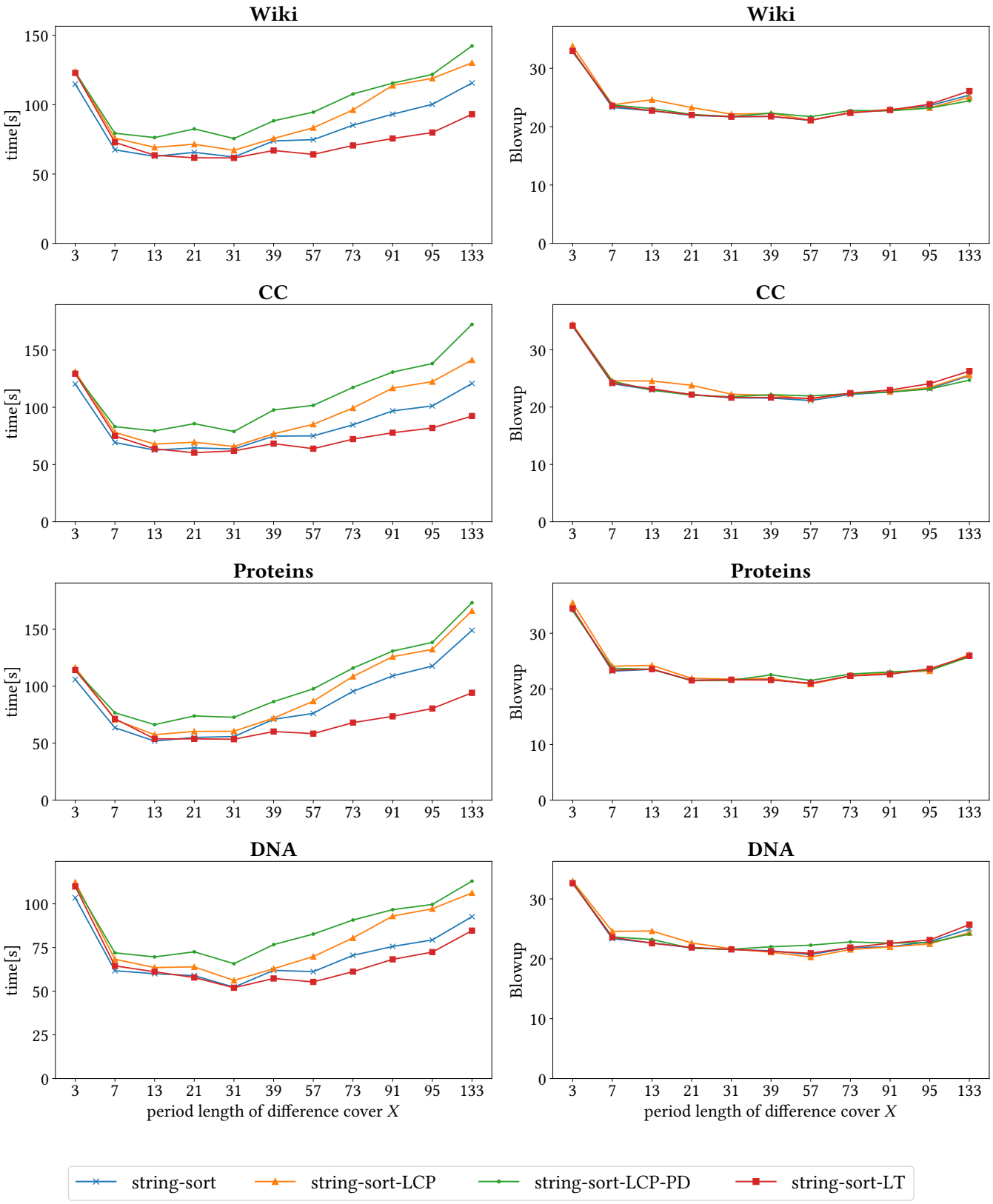


Figure 6.1.: Running times and blowup of DCX for different period lengths X with different string sort optimizations, 15.36 GB input and 768 PEs.

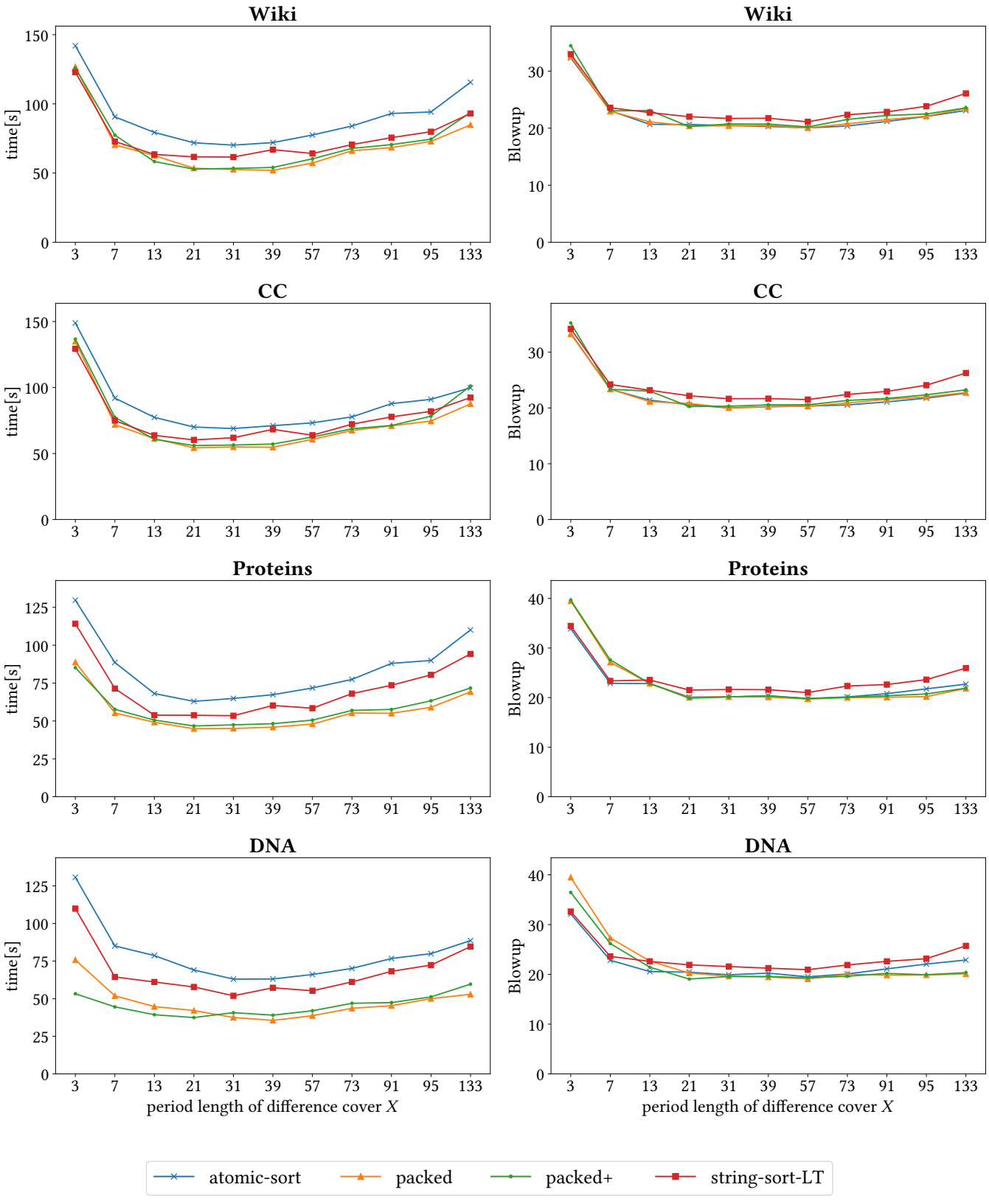


Figure 6.2.: Running times and blowup of DCX for different period lengths X with different sorter configurations, 15.36 GB input and 768 PEs.

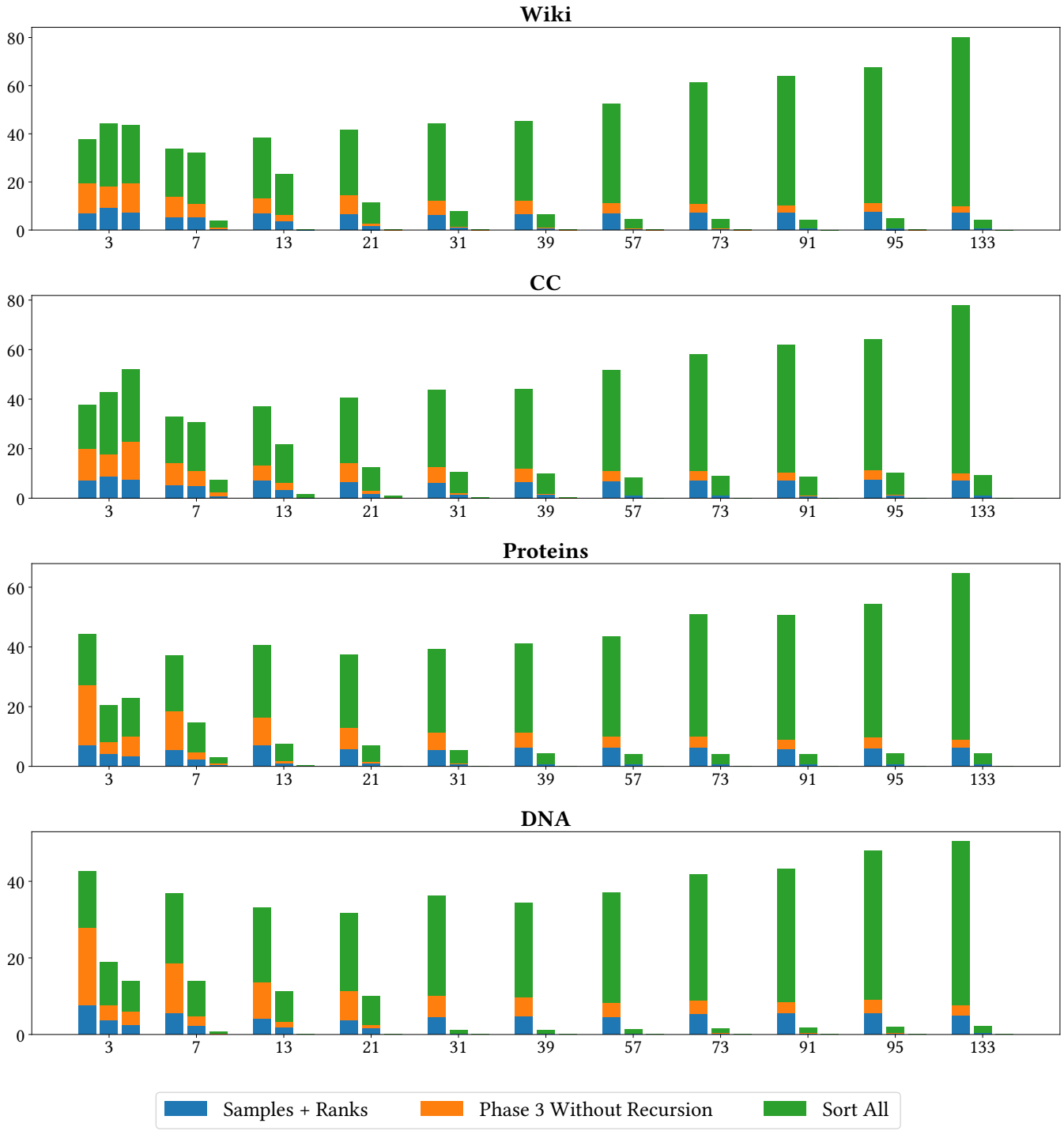


Figure 6.3.: Barplot shows time spent on the on the first level (first bar), the second level (second bar) and time spent on the remaining levels (third bar) for different period lengths X with 15.36 GB input and 768 PEs.

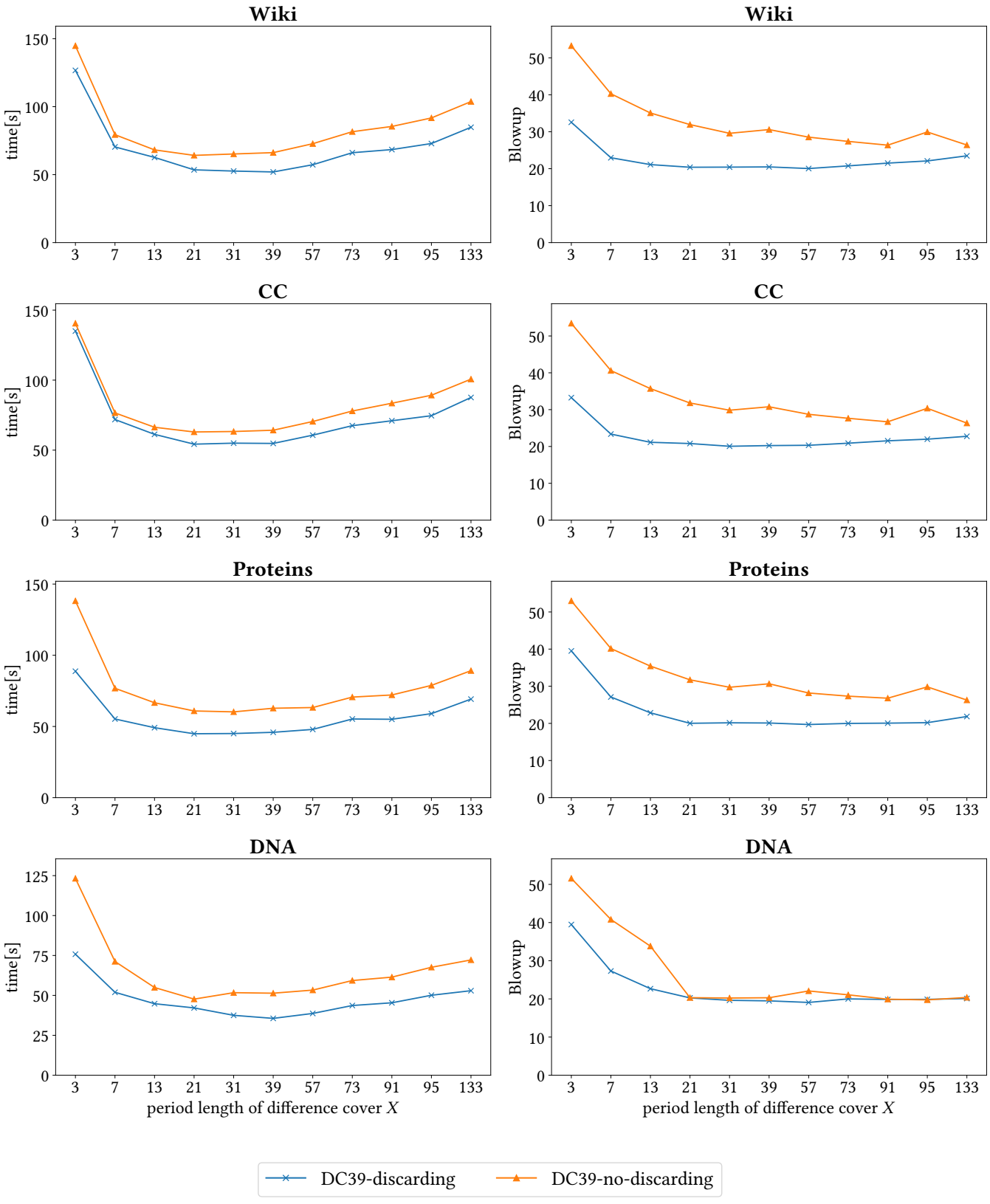


Figure 6.4.: Influence of discarding with different period lengths X on 15.36 GB input and 768 PEs.

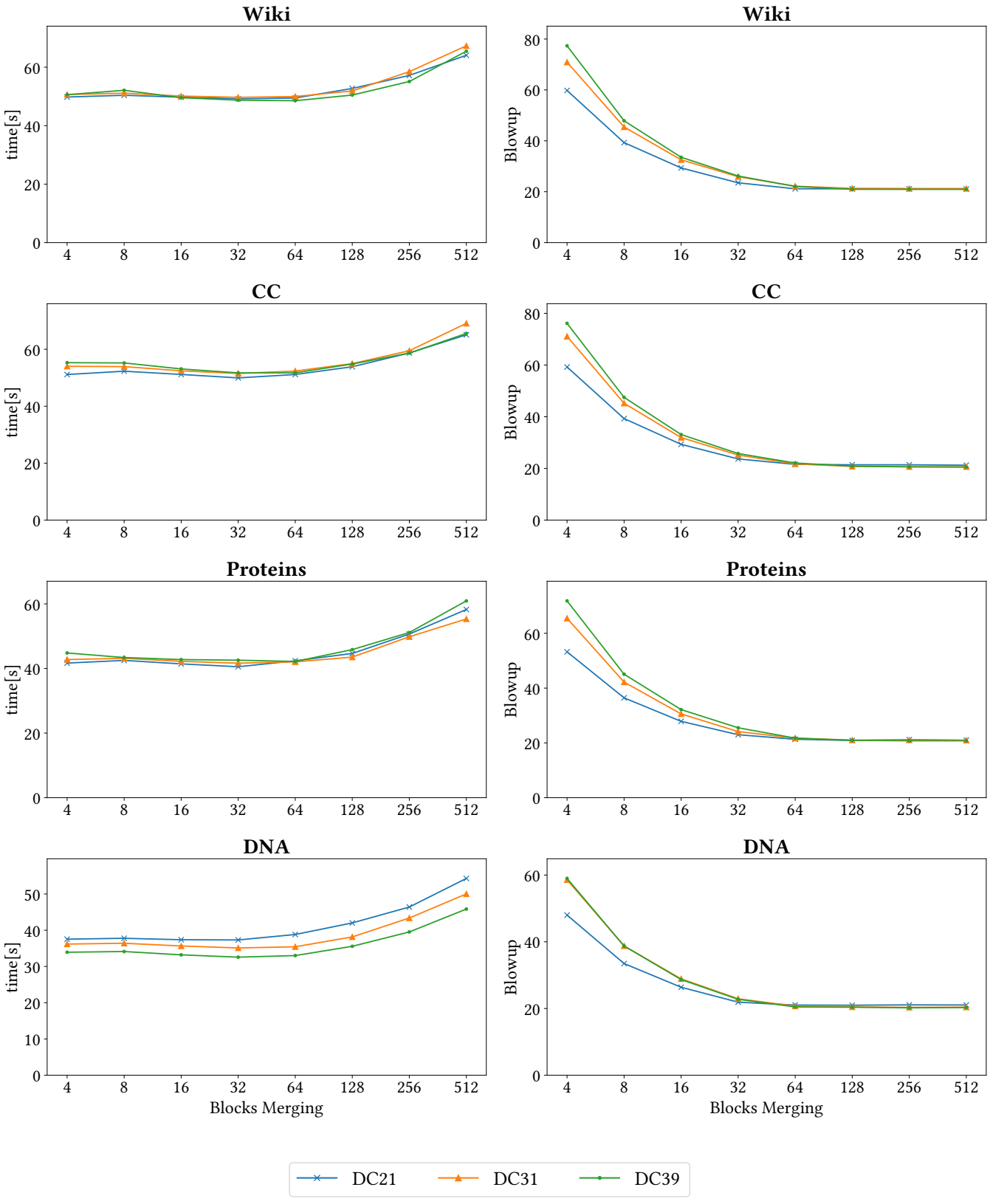


Figure 6.5.: Influence of block sizes on 15.36 GB input and 768 PEs.

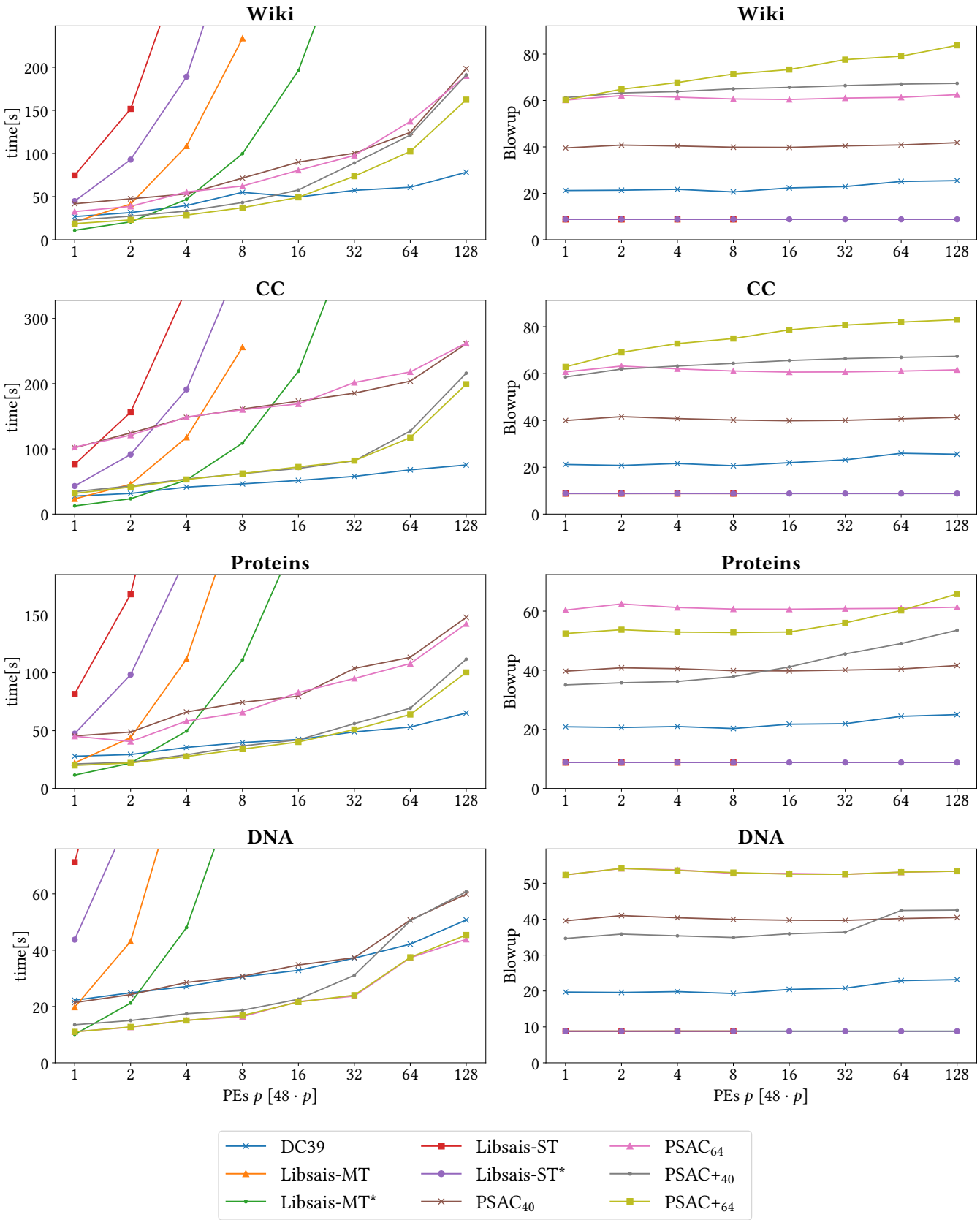


Figure 6.6.: Running times and blowup of the suffix array algorithms in our weak scaling experiments with 20MB per PE.

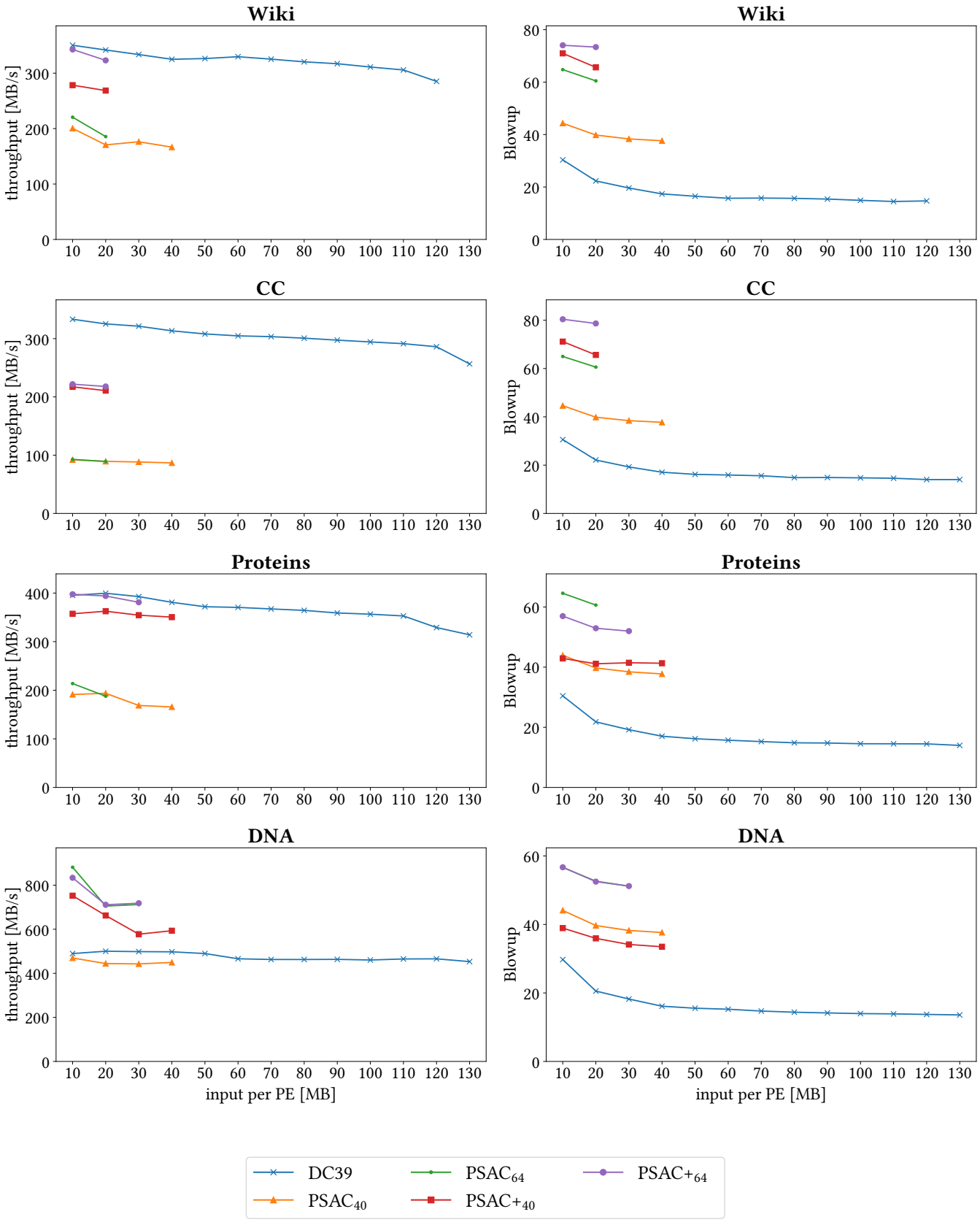


Figure 6.7.: Throughput and blowup during our breakdown test on 768 PEs.

7. Conclusion

In this thesis, we have developed a distributed suffix array algorithm. Our algorithm adapts the sequential DCX algorithm [46] to the distributed setting, generalizing the distributed DC3 implementation of Kulla and Sanders [52]. Additionally, we proposed and implemented multiple algorithmic improvements to distributed DCX. These are the most important insights of our experimental evaluations of DCX.

1. Packing characters together with comparison-based sorting proves to be the most efficient approach for sorting X-prefixes, outperforming string sorting techniques with on average 25% lower running times.

2. Discarding significantly improves running times by 13-39% depending on the LCP-values of the inputs.

3. Bucketing, combined with the novel load-balancing method chunking, achieves blowups around 20 bytes per character and even 14.2 bytes if the input per PE is large (92.16 GB).

We compared DCX, with the current state-of-the-art distributed suffix array algorithm PSAC [32] on up to 6144 PEs. On 3 out of 4 inputs, our algorithm achieves the fastest running times for more than 768 PEs with speedups up to 3.2×, while being the most memory-efficient across all inputs. Although DCX is slower on a DNA dataset (1.5× slower for 768 PEs, 1.15× slower for 6144 PEs), our algorithm offers a good time-space trade-off, enabling it to handle much larger text inputs. Moreover, DCX is able to process inputs 3× as large as our competitor. Overall, we showed that our DCX implementation is scalable, fast and still space-efficient.

Future Work

There are more aspects of DCX that can be inspected in future research. An open question is, whether distributed DCX can be combined with prefix doubling or inducing approaches to design an even more efficient hybrid-algorithm. Furthermore, one can consider to implement DCX in a *semi-external* setting. It combines the distributed with the external approach. Communication over a network between compute nodes works as before. In addition to its RAM, each compute node can now also use an external hard drive with much more memory, but slower access speeds. This allows to construct suffix arrays for even larger amounts of data, for example in the order of TB. Space-efficient sorting techniques using bucketing and chunking can possibly reduce the I/O transfers necessary by utilizing the RAM more efficiently. Optimizations proven useful in the distributed

setting, like discarding and packing, might also improve DCX in the semi-external setting. Another option is to consider implementing DCX for the GPU, benefiting from faster specialized hardware. The comparison of our algorithm with the two distributed suffix array algorithms of Kurpicz and Fischer [31] and the DCX implementation of Ferguson [28] remains open.

Bibliography

[1] LRZ documentation, Intel MPI, Settings for low memory footprint. https://doku.lrz.de/intel-mpi-10746523.html#IntelMPI-Settingsforlowmemoryfootprint. Accessed: 2025-03-09.

[2] 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.

[3] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. Optimal exact string matching based on suffix arrays. In *SPIRE*, 2002. URL: https://api.semanticscholar.org/CorpusID:15230729.

[4] Donald A. Adjeroh and Fei Nan. Suffix-sorting *via* shannon-fano-elias codes. *Algorithms*, 3(2):145–167, 2010. doi:10.3390/A3020145.

[5] Michael Axtmann. *Robust Scalable Sorting*. PhD thesis, Karlsruhe Institute of Technology, 2021. doi:http://dx.doi.org/10.5445/IR/1000136621.

[6] 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.

[7] Michael Axtmann and Peter Sanders. Robust massively parallel sorting. In *Workshop on Algorithm Engineering and Experimentation*, 2016. URL: https://api.semanticscholar.org/CorpusID:17464934.

[8] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.9.

[9] 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.

[10] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, page 360–369, USA, 1997. Society for Industrial and Applied Mathematics.

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

[12] Timo Bingmann. Scalable string and suffix sorting: Algorithms, techniques, and tools. *CoRR*, abs/1808.00963, 2018. URL: http://arxiv.org/abs/1808.00963, arXiv:1808.00963.

[13] Timo Bingmann. TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers, 2018. https://panthema.net/tlx, retrieved Oct. 7, 2020.

[14] 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.

[15] Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *CoRR*, abs/1403.2056, 2014. URL: http://arxiv.org/abs/1403.2056, arXiv:1403.2056.

[16] 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.

[17] Timo Bingmann, Peter Sanders, and Matthias Schimek. Communication-efficient string sorting. *CoRR*, abs/2001.08516, 2020. URL: https://arxiv.org/abs/2001.08516, arXiv:2001.08516.

[18] 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. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, page 3–16, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/113379.113380.

[19] 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.

[20] Florian Büren, Daniel Jünger, Robin Kobus, Christian Hundt, and Bertil Schmidt. Suffix array construction on multi-gpu systems. *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019. URL: https://api.semanticscholar.org/CorpusID:195325871.

[21] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *CPM*, volume 2676 of *LNCS*, pages 55–69. Springer, 2003. doi:10.1007/3-540-44888-8_5.

[22] Michael Burrows, D J Wheeler D I G I T A L, Robert W. Taylor, David J. Wheeler, and David Wheeler. A block-sorting lossless data compression algorithm. 1994. URL: https://api.semanticscholar.org/CorpusID:2167441.

[23] Charles J. Colbourn and Alan C. H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75:9–12, 2000. URL: https://api.semanticscholar.org/CorpusID:28681324.

[24] Sajal K. Das. Book review: Introduction to parallel algorithms and architectures : Arrays, trees, hypercubes by f. t. leighton (morgan kauffman pub, 1992). *SIGACT News*, 23:31–32, 1992. URL: https://api.semanticscholar.org/CorpusID:30516556.

[25] 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.

[26] Mrinal Deo and Sean Keely. Parallel suffix array and least common prefix for the gpu. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 2013. URL: https://api.semanticscholar.org/CorpusID:6442592.

[27] Martin Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143. IEEE, 1997. doi:10.1109/SFCS.1997.646102.

[28] Michael Ferguson. DCX implementation in FEMTO, an indexing and search system. https://github.com/femto-dev/femto/tree/main/src/ssort_chpl, 2025. Last accessed: 2025-03-18.

[29] 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.

[30] 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.

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

[32] Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2015. URL: https://api.semanticscholar.org/CorpusID:207228993.

[33] Pierre Fraigniaud and Emmanuel Lazard. Methods and problems of communication in usual networks. *Discret. Appl. Math.*, 53:79–133, 1994. URL: https://api.semanticscholar.org/CorpusID:9369364.

[34] Natsuhiko Futamura, Srinivas Aluru, and Stefan Kurtz. Parallel suffix sorting. 2001. URL: https://api.semanticscholar.org/CorpusID:18752570.

[35] 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.

[36] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel Distrib. Comput.*, 22(2):251–267, August 1994. doi:10.1006/jpdc.1994.1085.

[37] Andreas Gogol-Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9:11 – 11, 2008. URL: https://api.semanticscholar.org/CorpusID:8244335.

[38] 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.

[39] Ilya Grebnov. libsais. https://github.com/IlyaGrebnov/libsais/, 2021.

[40] Manuel Haag, Florian Kurpicz, Peter Sanders, and Matthias Schimek. Fast and lightweight distributed suffix array construction – first results, 2024. URL: https://arxiv.org/abs/2412.10160, arXiv:2412.10160.

[41] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38(6):2162–2178, 2009. doi:10.1137/070685373.

[42] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware (Cat. No.PR00268)*, pages 81–88, 1999. URL: https://api.semanticscholar.org/CorpusID:32790278.

[43] Juha Kärkkäinen and Dominik Kempa. Engineering a lightweight external memory suffix array construction algorithm. *Mathematics in Computer Science*, 11:137 – 149, 2017. URL: https://api.semanticscholar.org/CorpusID:12284533.

[44] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Annual Symposium on Combinatorial Pattern Matching*, 2009. URL: https://api.semanticscholar.org/CorpusID:12841864.

[45] Juha Kärkkäinen and Tommi Rantala. Engineering radix sort for strings. In *SPIRE*, 2008. URL: https://api.semanticscholar.org/CorpusID:33834889.

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

[47] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. *Proceedings of the fourth annual ACM symposium on Theory of computing*, 1972. URL: https://api.semanticscholar.org/CorpusID:16652988.

[48] Dong Kyue Kim, Junha Jo, and Heejin Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In *WEA*, volume 3059 of *LNCS*, pages 301–314. Springer, 2004. doi:10.1007/978-3-540-24838-5_23.

[49] 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. doi:10.1016/J.JDA.2004.08.019.

[50] Donald Ervin Knuth. The art of computer programming: Volume 3: Sorting and searching. 1998. URL: https://api.semanticscholar.org/CorpusID:5451516.

[51] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005. doi:10.1016/J.JDA.2004.08.002.

[52] 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.

[53] Vipin Kumar, A. Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing. Design and analysis of algorithms*, volume 2. 01 1994.

[54] Florian Kurpicz. *Parallel Text Index Construction.* PhD thesis, Technical University of Dortmund, Germany, 2020. doi:http://dx.doi.org/10.17877/DE290R-21114.

[55] 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.

[56] Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and fm-index construction. *2016 Data Compression Conference (DCC)*, pages 33–42, 2016. URL: https://api.semanticscholar.org/CorpusID:756320.

[57] Bin Lao, Ge Nong, Wai Hong Chan, and Jing Yi Xie. Fast in-place suffix sorting on a multicore computer. *IEEE Transactions on Computers*, 67:1737–1749, 2018. URL: https://api.semanticscholar.org/CorpusID:53284729.

[58] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theor. Comput. Sci.*, 387(3):258–272, 2007. doi:10.1016/J.TCS.2007.07.017.

[59] Hui Li and Kenneth C. Sevcik. Parallel sorting by over partitioning. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, page 46–56, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/181014.192329.

[60] Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. *Inf. Comput.*, 285(Part):104818, 2022. doi:10.1016/J.IC.2021.104818.

[61] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. Genome-scale algorithm design: Biological sequence analysis in the era of high-throughput sequencing. 2015. URL: https://api.semanticscholar.org/CorpusID:46273870.

[62] Udi Manber and Eugene Wimberly Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22:935–948, 1993. URL: https://api.semanticscholar.org/CorpusID:5074629.

[63] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *SODA*, pages 319–327. SIAM, 1990.

[64] Michael A. Maniscalco and Simon J. Puglisi. An efficient, versatile approach to suffix sorting. *JEA*, 12:1.2:1–1.2:23, 2007. doi:10.1145/1227161.1278374.

[65] Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *SWAT*, volume 3111 of *LNCS*, pages 372–383. Springer, 2004. doi:10.1007/978-3-540-27810-8_32.

[66] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004. doi:10.1007/S00453-004-1094-1.

[67] Peter M. McIlroy, Keith Bostic, and Douglas McIlroy. Engineering radix sort. *Comput. Syst.*, 6:5–27, 1993. URL: https://api.semanticscholar.org/CorpusID:37702205.

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

[69] Ahmed A. Metwally, Ahmed Hisham Kandil, and Mohamed Ibrahim Abouelhoda. Distributed suffix array construction algorithms: Comparison of two algorithms. *2016 8th Cairo International Biomedical Engineering Conference (CIBEC)*, pages 27–30, 2016. URL: https://api.semanticscholar.org/CorpusID:22460641.

[70] Yuta Mori. libdivsufsort. https://github.com/y-256/libdivsufsort, 2015.

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

[72] Waihong Ng and Katsuhiko Kakehi. Cache efficient radix sort for string sorting. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 90-A:457–466, 2007. URL: https://api.semanticscholar.org/CorpusID:33681633.

[73] Waihong Ng and Katsuhiko Kakehi. Merging string sequences by longest common prefixes. *Inf. Media Technol.*, 3:236–245, 2008. URL: https://api.semanticscholar.org/CorpusID:62616115.

[74] Ge Nong. Practical linear-time O(1)-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15, 2013. doi:10.1145/2493175.2493180.

[75] Ge Nong and Sen Zhang. Optimal lightweight construction of suffix arrays for constant alphabets. In *WADS*, volume 4619 of *LNCS*, pages 613–624. Springer, 2007. doi:10.1007/978-3-540-73951-7_53.

[76] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear time suffix array construction using d-critical substrings. In *Annual Symposium on Combinatorial Pattern Matching*, 2009. URL: https://api.semanticscholar.org/CorpusID:38578392.

[77] 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.

[78] Enno Ohlebusch. Bioinformatics algorithms: Sequence analysis, genome rearrangements, and phylogenetic reconstruction. 2013. URL: https://api.semanticscholar.org/CorpusID:39512900.

[79] Vitaly Osipov. Parallel suffix array construction for shared memory architectures. In *SPIRE*, 2012. URL: https://api.semanticscholar.org/CorpusID:13120089.

[80] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. arXiv:https://doi.org/10.1137/0216062, doi:10.1137/0216062.

[81] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. In *CSUR*, 2007. URL: https://api.semanticscholar.org/CorpusID:2653529.

[82] Tommi Rantala. Library of string sorting algorithms in c++. http://github.com/rantala/string-sorting, 2007. Accessed: 2012.

[83] Peter Sanders. Fast priority queues for cached memory. *ACM J. Exp. Algorithmics*, 5:7, 1999. URL: https://api.semanticscholar.org/CorpusID:11590125.

[84] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. Sequential and parallel algorithms and data structures: The basic toolbox. *Sequential and Parallel Algorithms and Data Structures*, 2019. URL: https://api.semanticscholar.org/CorpusID:201692657.

[85] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.*, 35:581–594, 2009. URL: https://api.semanticscholar.org/CorpusID:16366981.

[86] Matthias Schimek. Distributed string sorting algorithms. Master's thesis, Karlsruhe Institute of Technology, Germany, 2019. URL: http://dx.doi.org/10.5445/IR/1000098432.

[87] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007. doi:10.1002/SPE.768.

[88] Julian Seward. On the performance of BWT sorting algorithms. In *DCC*, pages 173–182. IEEE, 2000. doi:10.1109/DCC.2000.838157.

[89] 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, 07 2015. doi:10.1371/journal.pbio.1002195.

[90] Nataliya Timoshevskaya and Wu chun Feng. Sais-opt: On the characterization and optimization of the sa-is algorithm for suffix array construction. *2014 IEEE 4th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pages 1–6, 2014. URL: https://api.semanticscholar.org/CorpusID:322646.

[91] 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.

[92] David Weese. Indices and applications in high-throughput sequencing. 2013. URL: https://api.semanticscholar.org/CorpusID:5280467.

[93] Jing Yi Xie, Ge Nong, Bin Lao, and Wentao Xu. Scalable suffix sorting on a multicore machine. *IEEE Transactions on Computers*, 69:1364–1375, 2020. URL: https://api.semanticscholar.org/CorpusID:213350404.

A. Appendix

A.1. Theoretical Guarantees of Random Chunk Redistribution

Here, we state a theoretical result and its proof [40] on the imbalance guarantees of random chunk redistribution.

Theorem 3 (Random Chunk Redistribution)**.** *When redistributing chunks of size c uniformly at random across p PEs, with q 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 $0 \leq j < n/c$, PE i with $0 \leq i < p$, and bucket k with $0 \leq k < q$. 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 $\mathbb{P}[Y_i^k \geq 2n/(pq)]$ that PE i receives two times its expected number of elements or more. We have

$$\begin{aligned} \mathbb{P}\left[Y_i^k \geq \frac{2n}{pq}\right] &= \mathbb{P}\left[\sum_{j=0}^{n/c-1} X_{j,i}^k \geq \frac{2n}{pq}\right] \\ &= \mathbb{P}\left[\sum_{j=0}^{n/c-1} X_{j,i}^k - \frac{n}{pq} \geq \frac{n}{pq}\right] = \mathbb{P}\left[\sum_{j=0}^{n/c-1} X_{j,i}^k - \mathbb{E}[X_{j,i}^k] \geq \frac{n}{pq}\right]. \end{aligned} \tag{A.1}$$

As the value of $X_{i,j}^k$ is bounded by the chunk size c, the Bernstein inequality [19, Theorem 2.10, Corollary 2.11] yields the following bound

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

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

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

as the sum of the squares of a set of elements $0 \leq a_i \leq c$ 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 (A.2)

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

Combining these estimations, we obtain the bound

$$\mathbb{P}\left[Y_i^k \geq \frac{2n}{pq}\right] \overset{(A.1),(A.3)}{\leq} \exp\left(-\frac{3n}{8pqc}\right) \leq \exp\left(-(\gamma+2)\ln p\right) = \frac{1}{p^{\gamma+2}}$$

for $n \geq 8pqc\ln(p)(\gamma+2)/3$.

Although the random variables Y_i^k are dependent on each other, using the union-bound argument yields the following estimation

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

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 $q \leq p$. □