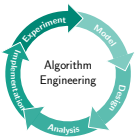

Scalable Distributed String Sorting Algorithms

Master's Thesis of

Pascal Mehnert

At the Department of Informatics
Institute of Theoretical Informatics
Karlsruhe Institute of Technology

First examiner: Prof. Dr. Peter Sanders

First advisor: M.Sc. Matthias Schimek
Second advisor: Dr. Florian Kurpicz

August 01, 2023 – February 01, 2024

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. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, February 01, 2024

. . . . . .
(Pascal Mehnert)

Abstract

String sorting algorithms have been studied extensively for sequential and shared-memory parallel models of computation. There has, however, been comparatively little and only very recent work covering string sorting in distributed-memory parallel systems. In this thesis, we directly build on the existing work to develop distributed algorithms that are more scalable with respect to two parameters: the number of processors used for sorting and the input size per processor in terms of characters. For the first aspect, we develop a multi-level generalization of existing multi-way string merge sort algorithms, based on a technique that has been applied successfully in atomic sorting. The developed algorithms are experimentally demonstrated to perform well for a range of inputs across a spectrum of magnitudes. We observe speedups up to five over the closest existing competitor on up to 24 576 processors.

For the second aspect, aimed at making distributed string sorting more scalable with respect to input size, we develop a space-efficient sorting framework which primarily distinguishes itself through the use of a compressed input representation. By deduplicating overlapping substrings and sorting the input in smaller chunks rather than as a whole, it is possible to create sorted permutations for inputs that would otherwise exceed the available working memory. We experimentally confirm this claim by demonstrating that an implementation of the framework is able to sort inputs of uncompressed size up to 22.4 GB per processor with only 2 GB memory available on average. Furthermore, an application of space-efficient sorting in suffix array construction, specifically as subroutine to DCX, is proposed. We show that our implementation is capable of sorting large difference cover samples, including for a difference cover modulo 8192, for texts comprising up to 1.23 TB in size.

Zusammenfassung

Sortieralgorithmen für Zeichenketten variabler Länge (Strings) wurden in der Vergangenheit für sequentielle und parallele Systeme mit gemeinsamem Speicher bereits umfassend untersucht. Hingegen existiert im Bezug auf Stringsortieralgorithmen für parallele Systeme mit verteiltem Speicher erst seit kurzem und bisher nur vergleichsweise wenig Forschung. Die vorliegende Masterarbeit baut direkt auf vorhandenen Ergebnissen auf und entwickelt verteilte Algorithmen, die in Bezug auf zwei Parameter besser skalierbar sind: die Anzahl der Prozessoren, die für das Sortieren verwendet werden können, und die Eingabegröße bezogen auf die Anzahl Zeichen pro Prozessor. Für den ersten Aspekt wird eine mehrstufige Verallgemeinerung von bestehenden Mehrwege-Mergesort Algorithmen entwickelt, basierend auf einer Methodik, die bereits erfolgreich für das atomare Sortieren eingesetzt wurde. Anhand einer experimentellen Evaluation wird empirisch gezeigt, dass sich die vorgestellten Algorithmen für eine Reihe von Eingaben von diverser Größe bewähren. Dabei werden Beschleunigungen von bis zu fünf gegenüber dem nächstgelegenen Konkurrenten auf bis zu 24 576 Prozessoren gemessen.

Bezogen auf den zweiten Aspekt, also zur Verbesserung verteilter Stringsortieralgorithmen hinsichtlich der Eingabegröße, wird eine Struktur für platzsparendes Sortieren entwickelt. Diese zeichnet sich in erster Linie durch die Verwendung eines komprimierten Eingabeformats aus, bei dem sich überlappende Strings dedupliziert gespeichert werden. Indem die Eingabe in kleineren Teilen verarbeitet wird, ist es möglich, sortierte Permutationen für Mengen von Strings zu erstellen, die andernfalls den verfügbaren Arbeitsspeicher überschreiten würden. Diese Behauptung wird experimentell dadurch bestätigt, dass eine Implementierung der Struktur in der Lage ist, Eingaben von dekomprimierter Größe bis zu 22.4 GB auf einem System mit nur 2 GB Arbeitsspeicher zu sortieren. Darüber hinaus wird auch eine mögliche Anwendung des Konzepts als Subroutine von DCX in der Suffix-Sortierung vorgeschlagen. Hierzu werden als Teil der experimentellen Evaluation Differenzabdeckungen mit großer Schrittweite für Texte von bis zu 1.23 TB sortiert.

Acknowledgments

I would like to thank my supervisors, Matthias Schimek and Florian Kurpicz, for guiding me through the process of writing this thesis and for sharing their expertise with me. I would also like to thank Prof. Dr. Sanders for providing me with the opportunity to work on such an interesting subject. Finally, I want to express my sincere gratitude towards my family for their encouragement and patience throughout my educational path.

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-NG at Leibniz Supercomputing Centre (`www.lrz.de`).

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Sorting, that is, finding a globally ordered permutation of a sequence of elements, is one of the most extensively studied problems in computer science. Conventional sorting algorithms treat input elements as *atomic* and assume they can be compared and swapped in constant time. However, at least the first assumption clearly does not hold if variable-length strings are to be sorted, since the worst-case complexity of comparing two strings is linear with respect to their length. It is therefore generally inadequate to treat strings as atomic objects and inefficient to sort sequences thereof using conventional sorting algorithms. Instead, specialized *string sorting algorithms* are required, which make use of the inherent structure of string sequences. By exploiting characteristic properties like *longest common prefixes*, such algorithms are able to yield improved runtime guarantees and can, ideally, bound the required work by the size of the *distinguishing prefix*.

Atomic sorting algorithms have been and continue to be the subject of extensive study across a wide variety of computational models, ranging from sequential, over shared- and distributed-memory parallel, to external-memory paradigms. Comparable effort has gone into designing and engineering string-specific sorting algorithms that are fast in practice for sequential and shared-memory parallel systems. Here too, external-memory algorithms have been developed to sort inputs that would otherwise exceed the available memory. However, the specific topic of string sorting algorithms for distributed-memory parallel systems has received surprisingly little attention in the past. The first publication, to our knowledge, that includes explicit work in this direction was published in 2019 [18]. The algorithm uses a multi-way merge sort approach which has since been refined by different authors [36, 11].

In distributed sorting algorithms, the input elements are usually assumed to be evenly distributed over p ordered *processing elements* (PEs) which receive *ranks* from 0 to $p-1$. A sorting algorithm must redistribute all input elements between PEs such that they are *sorted globally* and reorder them to ensure they are also *sorted locally*. The former condition, for a globally sorted output, requires that any element on a PE with rank i must be smaller than every element on PEs with higher rank $j > i$. Being sorted locally, simply refers to the order of elements on any PE. For strings, the order of elements is defined lexicographically. Sorting algorithms should also ensure that the output distribution remains balanced.

Distributed-memory parallel algorithms can typically expect to be executed on a much higher number of processors compared to their shared-memory counterparts. Furthermore, communication between PEs is performed using explicit *message passing*, which is generally more expensive than shared-memory techniques and comes with higher latency. Bottleneck communication volume and the number of communication rounds must therefore be considered as a crucial factor in the design and analysis of efficient distributed algorithms. Explicit communication also creates additional overhead in practice, for example in the form of send and receive buffers. On distributed systems, where memory is often sparse, particular care must therefore be taken to limit the amount of auxiliary space required by an algorithm.

1.1. Contribution

In this thesis, we develop two novel algorithms that make distributed string sorting more scalable. The first builds upon the multi-way merge sort algorithm from the existing work [18, 36, 11], allowing it to efficiently scale to a higher number of PEs. To this end, we develop a *multi-level* generalization of the algorithm based on the approach employed by Axtmann et al. for atomic sorting [6, 2]. The existing *single-level* variant works by partitioning the input into p pieces on every PE and sending strings to their final PE with a single exchange. The multi-level variant subdivides PEs into r groups, partitions the input into r pieces, and exchanges strings only between groups. This technique trades an additional exchange phase with every recursion level for a reduction in the overhead caused by partitioning. We demonstrate that this approach is not only an improvement in theory, as it is efficient for more PEs at a given input size, but also an improvement in practice. In an experimental evaluation, multi-level variants exhibit improved scaling behavior on up to 24 576 PEs for a range of input sizes and are shown to outperform competitors for sufficiently large values of p. Relating to multi-level merge sort, we also develop partitioning and load-balancing methods, as well as a multi-level Bloom filter for the purpose of distinguishing prefix approximation.

The second algorithm developed in this thesis aims to make distributed string sorting algorithms more *space-efficient*, thereby allowing them to accept larger inputs. This is primarily achieved through the use of a compressed input representation where strings may overlap and shared characters are not duplicated. Thus, inputs that would otherwise exceed the available memory can be instantiated, especially in cases with high duplication between strings. Because strings must be *materialized*, i.e., converted to an uncompressed format, before they can be exchanged using message passing, it is not possible to sort the entire input at once. Instead, we propose to partition strings into ordered *quantiles*, which individually fit into memory, sorting only one quantile at a time, and thereby determining the rank of each string within a sorted permutation of the input. For an implementation based on multi-level merge sort, we demonstrate that space-efficient sorting is able to process inputs of uncompressed size up to 22.4 GB per PE on a system with only 2 GB memory available on average. We also provide motivation for an application in suffix-sorting as subroutine to DCX, by sorting large difference cover samples for texts up to 1.23 TB in size.

1.2. Outline

Prior to any substantive work, Chapter 2 introduces some conventions and notations used throughout this thesis and discusses related work. Chapter 3 develops a number of building blocks, including a distributed string sorting algorithm for small inputs, as well as distributed partitioning and group assignment algorithms. The first major part of this thesis—multi-level merge sort—is covered in Chapter 4. Space-efficient string sorting as a framework, as well as a realization thereof, using multi-level merge sort, is proposed in Chapter 5. After an overview of implementation details and after briefly discussing the experimental setup, Chapter 6 mainly comprises an experimental evaluation. Finally, Chapter 7 summarizes the results of this thesis and gives pointers for possible future work on the topic.

2. Preliminaries

This chapter lays the necessary foundations for the rest of this thesis. Section 2.1 introduces commonly used definitions and a number of notational conventions. Section 2.2 gives specifics for the distributed-memory parallel model of computation and provides runtime bounds for important collective operations. Used as subroutine to various distributed algorithms, Section 2.3 concisely describes MSD radix sort—our sequential string sorting algorithm of choice. Finally, Section 2.4 discusses related work from the topic of distributed string sorting algorithms and from the subject of highly scalable atomic sorting.

2.1. Definitions and Notation

An *array* $\mathcal{A} \coloneqq [a_0, \ldots, a_{n-1}]$ is an ordered sequence of n elements where $|\mathcal{A}| \coloneqq n$ denotes the length of $\mathcal{A}$. A *string* $s \coloneqq [c_0, \ldots, c_{n-1}]$ is an array of characters from an ordered *alphabet* $\Sigma \coloneqq \{1, \ldots, \sigma\}$. Unless stated otherwise, we assume that strings are terminated by a *sentinel* element that is not part of the alphabet, i.e., $c_{n-1} = \$ \notin \Sigma$. The sentinel is defined as a null-terminator and therefore smaller than every character in Σ. It is possible—and necessary for the algorithms in Section 5—to replace the sentinel with other end-of-string indicators such as explicitly storing their length. String arrays are represented as arrays of pointers which is illustrated in Figure 2.1. When sorting strings, the lexicographical order of characters is used to define the binary relation "$<$". We use $+\infty$ and $-\infty$ to denote strings that are lexicographically larger and smaller than every other string. For an array of strings $\mathcal{S}$, we define the *character array* $\mathcal{C}(\mathcal{S}) \coloneqq s_0 + \cdots + s_{p-1}$ as the concatenation of all strings. For an array of strings $\mathcal{S}$, we define $\|\mathcal{S}\| \coloneqq |\mathcal{C}(\mathcal{S})| = \sum_{i=0}^{n-1} |s_i|$ as the total length of strings in $\mathcal{S}$.

The notations $[i, j)$ and $[i, j]$ are used to denote ranges of whole numbers $\{i, \ldots, j-1\}$ and $\{i, \ldots, j\}$ respectively. Analogously, in addition to regular array indexing $\mathcal{A}[i] \coloneqq a_i$, we define $\mathcal{A}[i, j) \coloneqq [a_x \mid x \in [i, j)]$ and $\mathcal{A}[i, j] \coloneqq [a_x \mid x \in [i, j]]$. Corresponding to regular *set comprehensions* we use *array comprehensions* such as $\left[x^2 \mid x \in \mathcal{A}\right] \coloneqq [a_0^2, a_1^2, \ldots, a_{n-1}^2]$ when the order of elements is unambiguous.

Let $\textsc{lcp}(s, t)$ denote the *longest common prefix* (LCP) of two strings s and t. The *LCP array* of a string array $\mathcal{S} = [s_0, \ldots, s_{n-1}]$ is defined as $\mathcal{H}(\mathcal{S}) \coloneqq [\perp, h_1, \ldots, h_{n-1}]$ with LCP values $h_i \coloneqq \textsc{lcp}(s_{i-1}, s_i)$ and undefined first value $\perp$. We define the sum of LCP values as $\mathcal{L}(\mathcal{S}) \coloneqq \sum_{i=1}^{n-1} h_i$. The *distinguishing prefix* $\textsc{dp}_{\mathcal{S}}(s)$ of a string s with respect $\mathcal{S}$ is the shortest prefix of s that is required to establish the rank of s in a lexicographical order of $\mathcal{S}$. Note that, while sentinel characters are never part of any longest common prefix, they can be part of distinguishing prefixes if necessary. We also refer to (the size of) the distinguishing prefix of string arrays $\mathcal{D}(\mathcal{S}) \coloneqq \sum_{i=0}^{n-1} \textsc{dp}_{\mathcal{S}}(s_i)$. The definition can be extended to use prefixes with respect to another string array $\mathcal{S}'$ which we denote as $\mathcal{D}_{\mathcal{S}'}(\mathcal{S})$. If $\mathcal{S}$ is sorted, then

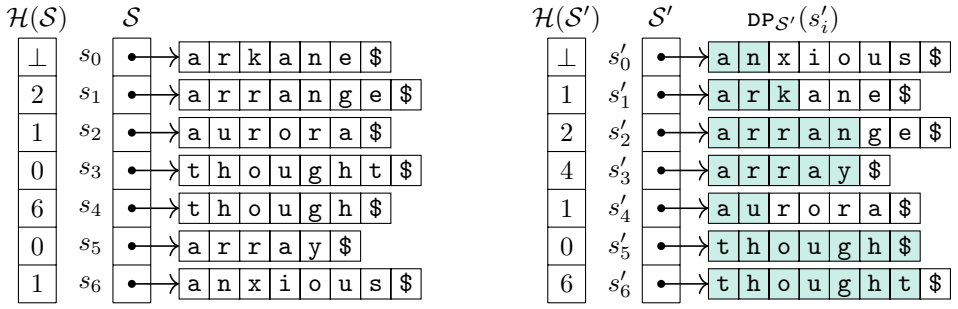

Figure 2.1.: Unsorted string array $\mathcal{S}$ with LCP array and the corresponding sorted string array $\mathcal{S}'$ with LCP array and distinguishing prefixes.

distinguishing prefix lengths are $|\text{DP}_{\mathcal{S}}(s_i)| = \max\{h_i + 1, h_{i+1} + 1\}$ with sentinels $h_0 = 0$ and $h_n = 0$.

We use a number of abbreviations for key characteristics of string arrays when it is clear which array is being referenced. For distributed algorithms, this will always be a global string array $\mathcal{S}$. The total number of strings is denoted $n \coloneqq |\mathcal{S}|$, the cumulative length of strings is $N \coloneqq \|\mathcal{S}\|$, the size of the distinguishing prefix is $D \coloneqq \mathcal{D}(\mathcal{S})$, and the sum of LCP values is $L \coloneqq \mathcal{L}(\mathcal{S})$. Furthermore, let $\hat{\ell} \coloneqq \max_{s\in\mathcal{S}}|s|$ and $\check{\ell} \coloneqq \min_{s\in\mathcal{S}}|s|$ be the lengths of the longest and shortest strings, and $\hat{d} \coloneqq \max_{s\in\mathcal{S}}|\text{DP}_{\mathcal{S}}(s)|$ the length of the longest distinguishing prefix of any string.

2.2. Model of Computation

Throughout this thesis we use a distributed-memory parallel machine with p ordered *processing elements* (PEs) numbered from 0 to $p - 1$ as our computational model. PEs are connected by a network and communicate using *single-ported* message passing where every PE can only exchange messages with a single other PE at a time. Communication between PEs is assumed to use the *linear model* of communication where sending a message of m units of data takes time $\alpha + \beta m$ [19]. The constant α-term is the so called *start-up latency* and β models the network's inverse bandwidth, i.e., the speed at which units of data can be transferred. For the parameters, it is generally possible to assume that $\alpha \gg \beta$. Somewhat unusually, we use bits, rather than machine words, as the standard unit of data, which makes it possible to differentiate between sending characters and larger integers. Sending m characters of an alphabet of size σ from one PE to another incurs total communication cost in $O(\alpha + \beta m \log \sigma)$.

We often rely on well-understood *collective communication* operations to state the communication complexity of algorithms. Here, all PEs participate in communication with simple, predictable, and uniform patterns of interaction. This makes analysis as simple as plugging characteristic values into existing formulas. Collective operations relevant to this thesis with accompanying bounds on complexity are introduced hereafter.

Broadcast A single PE—called the *root*—has a message m of size n bits which it wants to distribute, i.e., *broadcast*, to every other PE. A lower bound for latency in $\Theta(\alpha \log p)$ must

hold, since the number of PEs that have received m can at most double with every round of communication due to the single-ported requirement. By splitting messages into smaller chunks, *pipelining* can be used to properly utilize the network with large messages. Two-tree algorithms combine two pipelined binary trees to better use the available bandwidth while achieving optimal communication time in $O(\alpha \log p + \beta n)$ [34].

Reduce/All-Reduce Let $\oplus$ be an associative operator and assume that each PE has a message m_i of equal length n bits. A *reduction* computes the result of $\bigoplus_{i=0}^{p-1} m_i$ on a root PE. Messages are usually made up of smaller units, e.g., integers, that can be reduced individually and make pipelining possible. Similar to broadcasts, reductions can use pipelined binary trees for communication complexity in $O(\alpha \log p + \beta n)$ [34]. All-reduce operations make the result available to all PEs which can be implemented using an additional broadcast with the same asymptotic bound.

Prefix-Sum There is again an associative operator $\oplus$ and messages m_i of equal length n which are, as is the case for reductions, often made up of multiple smaller units. A *prefix-sum*, or *scan*, computes the result of $\bigoplus_{j=0}^{i} m_j$ on each PE i. The value of m_i is omitted on each PE for an *exclusive* prefix-sum. The operation can be implemented using similar techniques to reductions, which again requires communication time in $O(\alpha \log p + \beta n)$ [34].

Gather/All-Gather Here, PEs have a message m_i consisting of n bits which must all be sent to a single root PE. Hence, the root *gathers* and concatenates messages $m_0 \cdot m_1 \cdot \ldots \cdot m_{p-1}$. This can be seen as a reduction with concatenation as operator and thus needs communication time in $O(\alpha \log p + \beta pn)$ [35, p. 412]. An additional broadcast in time $O(\alpha \log p + \beta pn)$ suffices to obtain an all-gather operation. If messages have varying lengths n_i with sum $n \coloneqq \sum_{i=0}^{p-1} n_i$, then both operations need communication time in $O(\alpha \log p + \beta n)$.

All-To-All Initially, each PE i has p messages $m_i^0, \ldots, m_i^{p-1}$ that need to be delivered such that afterwards PE i has messages $m_0^i, \ldots, m_{p-1}^i$. Hence, every PE exchanges information with every other PE. If all messages have equal length n bits, then a *regular* all-to-all exchange can be implemented with $p - 1$ rounds of direct exchanges using the *1-factor algorithm* in time $O(\alpha p + \beta pn)$ [35, p. 414]. In the case of small messages, an indirect exchange using a hypercube communication pattern consisting of $\log p$ iterations with total communication in $O(\alpha \log p + \beta pn \log p)$ may be preferable [35, p. 415]. In the context of strings we often require *irregular* all-to-all exchanges where message have arbitrary lengths n_i^j. We can define the maximum number of bits sent or received by any PE, the so-called *bottleneck communication volume*, as $h \coloneqq \max_i \left\{ \sum_j n_i^j, \sum_j n_j^i \right\}$. Messages can be delivered using two successive uniform all-to-all exchanges. Each message m_i^j is split into p equally sized pieces $m_i^{j,k}$ for $k \in [0, p)$ which are then sent indirectly via PE k and reconstructed on PE j. Using the 1-factor algorithm for uniform all-to-all exchanges, this *two-phase algorithm*, has communication complexity in $O(\alpha p + \beta h)$ if message sizes are sufficiently large—at least $h = \Omega(p \log h)$ bits to dominate the exchange of piece sizes [35, p. 417]. Irregular all-to-all exchanges are the most general communication pattern, as they form a superset of other operations such broadcasts and gathers. However, the two-phase algorithm is not efficient in cases with sparse communication patterns, where many send and receive counts are equal to zero.

Irregular Exchanges Unfortunately, at least for the multi-level algorithms in Chapter 4, we also encounter cases of highly irregular communication where simple collective operations are not sufficient. In particular, we require exchange operations where each PE sends messages to a subset of PEs that is unknown before execution and whose size is usually only bounded asymptotically. Using direct exchanges, an obvious lower bound for this is given by $\Omega(\alpha r + \beta m)$ if each PE sends and receives messages to and from r other PEs with total length m. Unfortunately, it is not clear how such an exchange can be implemented optimally while observing the single-ported requirement. The problem here is that a schedule of exchanges needs to be determined such that each PE can be active during every round of communication. Naive solutions, like using all-to-all exchanges with empty messages, incur either latency in $O(\alpha p)$, which may be acceptable in practice but undesirable in theory, or factors up to $\log p$ in communication volume for indirect exchanges, which is unacceptable for large messages. More advanced algorithms are conceivable, for example using a graph coloring formulation [35, p. 417], though it is unclear how these can be implemented in practice with acceptable runtime overhead. In practice, sparse all-to-all exchanges where messages are sent in parallel may prove effective. Such approaches are, however, precluded from our theoretic analysis due the requirement for single-portedness. Conforming to the approach chosen in the paper on which our multi-level merge sort algorithms are based, we define a black-box function to encapsulate the cost of irregular exchanges [6]. The term $\text{Exch}(p, m, r)$ gives the time it takes for p participating PEs, where each PE sends and receives at most m bits, to exchange at most r messages in either direction. We also use $\widetilde{\text{Exch}}(p, m, r) \coloneqq (1 + o(1))\,\text{Exch}(p, m, r)$ to absorb terms that are dominated by data exchange phases. As previously noted, we assume direct exchanges and therefore $\text{Exch}(p, m, r) = \Omega(\alpha r + \beta m)$.

2.3. Sequential String Sorting

There exist a large variety of sequential sorting algorithms including multi-key quicksort, LCP-merge sort, and LCP-insertion sort—Bingmann provides a recent and comprehensive overview [8]. To sort local string arrays as subroutine to the distributed algorithms in this thesis, we use *most significant digit* (MSD) radix sort, a variant of regular radix sort for strings. Significant effort has gone into engineering radix sort for strings and the resulting implementations are some of the fastest string sorters in practice [29, 25].

With an initial string array $\mathcal{S}$, MSD radix sort works by repeatedly splitting $\mathcal{S}$ into σ subproblems. At each step, all strings in the current array have a common prefix of length h. Starting with an initial character depth $h = 0$, by looking at the character at position h and splitting $\mathcal{S}$ into σ arrays accordingly, the common prefix is increased to $h + 1$. For inputs with $o(\sigma)$ strings, maintaining σ buckets is no longer efficient and a different string sorting algorithm is required. We assume that multi-key quicksort [7] is used for these base cases, which has runtime in $O(D + n \log n)$ with the definitions previously introduced in this chapter. The radix steps have total runtime in $O(D)$. Sorting $O(n/\sigma)$ base cases with $O(\sigma)$ strings each and cumulative distinguishing prefix at most D, has runtime in $O(D)$ with multi-key quicksort. Therefore, MSD radix sort has time complexity in $O(D + n \log \sigma)$ which is good for small alphabets. The algorithm is less suited for large alphabet and may, depending on the implementation, struggle with large recursion stacks. Many algorithmic optimizations, such as *super-alphabets* and *character-caching*, are possible to improve empirical performance [8].

2.4. Related Work

Compared to sequential string sorting [29, 25, 7, 8], parallel string sorting in shared-memory machines [32, 14, 10, 15, 8], and even string sorting in external memory [1, 16], there has been surprisingly little work on sorting strings in distributed-memory parallel systems. The, to our knowledge, first distributed sorting algorithm explicitly designed for strings was proposed by Fischer and Kurpicz as a subroutine to a distributed suffix array construction algorithm [18]. The authors therein describe their algorithm as a distributed variant of shared-memory *string sample sort* [10]. In their master's thesis [36]—the results of which have since been published [11]—Schimek develops two new algorithms, *distributed String Merge Sort* (MS) and *distributed Prefix Doubling String Merge Sort* (PDMS), based on the approach chosen by Fischer and Kurpicz. Since the multi-level sorting algorithms in this thesis are directly based on MS and PDMS, the following provides a detailed description of the two algorithms. Figure 2.2 includes an illustration of single-level multi-way merge sort without string-specific details.

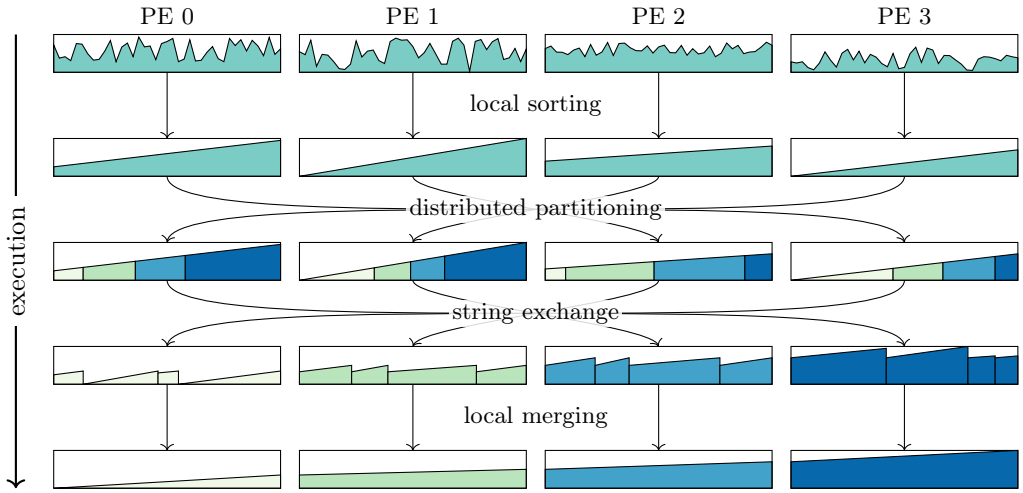

Figure 2.2.: Execution of single-level multi-way merge sort for an instance with $p = 4$ PEs. Colors indicate lexical distribution of elements.

Both MS and PDMS use a multi-way partitioning and merging scheme. For an input consisting of a string array $\mathcal{S}_i$ on each PE i, both algorithms first sort the arrays locally. MS then proceeds by globally partitioning the input into p buckets $\mathcal{B}^0, \ldots, \mathcal{B}^{p-1}$, by globally drawing $\Theta(p^2)$ samples at regular intervals, sorting the samples, and selecting $p-1$ *splitters*; again at regular intervals. Buckets are ordered, meaning that for any strings $s_i \in \mathcal{B}^i$ and $s_j \in \mathcal{B}^j$ from buckets $i < j$ the same order $s_i < s_j$ is guaranteed to hold for the strings. To ensure strings are globally ordered, the algorithm then assigns the strings of bucket i to PE i, by splitting local string arrays according to the global partition and exchanging the resulting sequences with an irregular all-to-all exchange. It only remains to merge all received sequences with a multi-way merging algorithm which restores local order of strings. PDMS enhances MS based on the observation that strings can be arbitrarily long and therefore cause high communication volume. In reality, depending on the string array, some characters, if not most, may be unnecessary to establish a correct order of strings, specifically in cases where $D \ll N$. By repeatedly applying a Bloom filter [12] to determine duplicates among

exponentially growing string prefixes, PDMS approximates the distinguishing prefix of each string. From here, the algorithm works like MS, expect using string arrays consisting of the approximated prefixes. The authors propose a number of string-specific optimizations for both MS and PDMS by exploiting LCP values. Notably, strings can be compressed for all-to-all exchanges by only sending common prefixes once, which we refer to as *LCP compression*. Local merging can use an LCP-ware multi-way merging algorithm, thereby bounding the number of character comparisons by D.

We also draw from a technique used in atomic sorting. Specifically, our multi-level merge sort algorithms are closely related to *Recurse Last Multi-way Merge sort* (RLM-sort) and, more distantly, to *Adaptive Multi-level Sample sort* (AMS-sort), both of which were proposed by Axtmann et al. [6, 3]. The ideas of RLM- and AMS-sort are similar: given an input distributed over p PEs, rather than partitioning local data into p pieces and delivering each piece to its final PE with a single exchange, instead divide PEs into groups of size p', only exchange elements between group, and proceed recursively. This approach needs to exchange data multiple times, which for k level of recursion results in a factor k more communication volume. However, by only needing to compute $r = p/p'$ splitters, the authors are able to improve the minimum size at which the algorithms are efficient from $\Omega(p^2 \log p)$ to $\Omega(p^{1+1/k} \log p)$ for RLM- and $\Omega(p^{1+1/k} / \log p)$ for AMS-sort. Similar approaches have been explored previously, for example, in the *bulk synchronous parallel* (BSP) model for sample sort by Gerbessiotis and Valiant [20].

3. Techniques

The following sections introduce a number of techniques and algorithms that are required as subroutines in later chapters. We first cover distributed sorting of small string sets in Section 3.1. A distributed ordered partitioning algorithm, as well associated sampling techniques are described in Section 3.2. Finally, Section 3.3 covers distributed assignment of strings to groups of PEs.

All algorithms in this chapter obtain similar inputs, which we formalize with the following conventions. Each PE i receives an array of strings $\mathcal{S}_i$ as input. The global input array is defined as $\mathcal{S} \coloneqq \mathcal{S}_0 + \cdots + \mathcal{S}_{p-1}$ using concatenation to allow for duplicate elements. Recall the definitions of n, N, $\hat{\ell}$, $\ell\!\!\!^{\smile}$, and $\hat{d}$ for $\mathcal{S}$ from Section 2.1.

3.1. LCP-Hypercube Quicksort

The distributed merge sort algorithms in this thesis require comparatively small sets of strings to be sorted as a subroutine. Once sorted, the strings are used to compute a globally ordered partition of a larger input. In their Master's thesis [36], Schimek proposes two algorithms for this scenario. The first—a centralized approach that gathers strings, sorts them sequentially on a single PE, and broadcasts them afterwards—proved a bottleneck for even moderate input sizes based on empirical data. To deal with lager numbers of PEs, the author proposes a distributed hypercube quicksort algorithm as an improvement which has been further analyzed in a subsequent paper [11]. The algorithm is directly based on RQuick—a "robust" version of hypercube quicksort for atomic objects proposed and implemented by Axtmann [2, 3]. What makes RQuick novel and distinguishes it from other distributed hypercube quicksort implementations is an initial random redistribution of inputs, the use of a tie-breaking scheme to cope with duplicates, and the details of the scheme used to approximate the median and thereby determine the splitter.

In an attempt to make RQuick more efficient for strings, we applied several string-related optimizations. Algorithm 1 shows pseudocode for the resulting variant—LCP-RQuick. First, during the local sorting phase, the LCP array of $\mathcal{S}_i$ is computed as by-product of sorting and saved as $\mathcal{H}_i$ on each PE. LCP arrays must be partitioned and sent alongside the string arrays during each iteration. This incurs additional communication, but does not actually change the algorithm's asymptotic complexity, which we show in Theorem 1. The main improvement over regular RQuick is that using the LCP arrays, we can apply a binary LCP-merge algorithm to combine the received string arrays. This also leaves us with an up-to-date LCP array for the result. It is also possible to use LCP values to speed up locating the splitter within local string arrays. When combined, these changes ideally improve the algorithm's required local work from $\log p$ scans of the entire character array to a single scan.

Algorithm 1: LCP-Hypercube Quicksort (LCP-RQuick) based on [3, 36]

Input: Local string array $\mathcal{S}_i$ on each PE i and number of PEs $p = 2^d$

1 $\mathcal{S}_i \leftarrow$ randomly redistribute $\mathcal{S}_i$ *// for details see [3]*
2 `Sort`$(\mathcal{S}_i) \rightarrow \mathcal{H}_i$ *// save LCP array during sorting*
3 **for** $h \leftarrow d-1$ **downto** 0 **do** *// hypercube dimension*
4 $s \leftarrow$ find approximate median *// for details see [3]*
5 $\mathcal{S}_i^{\leq} \cdot \mathcal{S}_i^{>},\ \mathcal{H}_i^{\leq} \cdot \mathcal{H}_i^{>} \leftarrow$ split $\mathcal{S}_i$ and $\mathcal{H}_i$ at s *// can exploit LCP information*
6 $j \leftarrow i \oplus 2^h$ *// communication partner*
7 **if** $j < i$ **then**
8 send $\mathcal{S}_i^{\leq}, \mathcal{H}_i^{\leq}$ to PE j and receive $\mathcal{S}_j^{>}, \mathcal{H}_j^{>}$ from j
9 $\mathcal{S}_i,\ \mathcal{H}_i \leftarrow$ `LCP-Merge`$(\left(\mathcal{S}_i^{>}, \mathcal{H}_i^{>}\right), \left(\mathcal{S}_j^{>}, \mathcal{H}_j^{>}\right))$
10 **else**
11 send $\mathcal{S}_i^{>}, \mathcal{H}_i^{>}$ to PE j and receive $\mathcal{S}_j^{\leq}, \mathcal{H}_j^{\leq}$ from j
12 $\mathcal{S}_i,\ \mathcal{H}_i \leftarrow$ `LCP-Merge`$(\left(\mathcal{S}_i^{\leq}, \mathcal{H}_i^{\leq}\right), \left(\mathcal{S}_j^{\leq}, \mathcal{H}_j^{\leq}\right))$

Output: Locally and globally sorted string array $\mathcal{S}_i$ with LCP array $\mathcal{H}_i$

Note that in Algorithm 1 the number of PEs is assumed to be a power of two. If this assumption does not hold, then the input must be reduced to a hypercube with dimension $d = \lfloor \log p \rfloor$ prior to the algorithm itself. Here, any PE with rank $i > 2^d$ moves their entire input array to PE $i - 2^d$ and thereafter no longer participates in sorting.

Lemma 1. *Using binary LCP-merge from Algorithm 2, LCP-RQuick, as described in Algorithm 1, requires at most* $L + n\lfloor \log p \rfloor + p\hat{\ell}\lfloor \log p \rfloor$ *character comparisons summed over all PEs and hypercube-dimensions, to merge string sequences.*

The following proof is closely based on the proof by Bingmann et al. of an analogous statement for two-way LCP-merge sort [10, Theorem 2]. Our version requires an additional term $p\hat{\ell} \log p$ to account for LCP values "lost" after splitting arrays during each iteration.

Proof. First of all, note that there are only three places in `LCP-Merge` where character comparisons can occur—specifically in lines 8 and 9. The first two, in the `while`-loop, are essentially part of an inlined `LCP-Compare` routine that scans over positions of s_0 and s_1 while the corresponding characters are equal, or until a null-terminator is found. The third comparison, in the `if`-statement, rechecks the characters at the last encountered position and thereby determines the order of s_0 and s_1. Because the same characters are compared in two cases and the third involves a comparison of one of them to zero, we can view the entire group as a single three-way comparison, and use status flags for the comparison to zero. Each time the condition of the `while`-loop is evaluated to `true`, a corresponding entry in an LCP array is increased by one. And for each string, there can only be a single `false`-outcome of the condition per call to `LCP-Merge`; specifically when a string is appended to $\mathcal{S}$. Thus, the former can be bounded by the sum of LCP values after merging and the latter contributes at most n comparisons per dimension of the hypercube, i.e., $n\lfloor \log p \rfloor$ in total. To arrive at a bound for the former term, we need to establish a number of additional observations about LCP-RQuick. After every iteration of the `for`-loop in Algorithm 1, the global sum of LCP

values is at most L, i.e., $\sum_{i=0}^{p-1} \mathcal{H}_i \leq L$. The only points at which LCP values are changed after the initial local sorting, are during merging and after splitting local arrays (the first entry of $\mathcal{H}_i^{>}$ is implicitly set to zero by the next call to `LCP-Merge`). Resetting a single LCP value per PE discards at most $\hat{\ell}$ characters which amounts to $p\hat{\ell}\lfloor \log p \rfloor$ over all PEs and dimensions of the hypercube. Otherwise, no LCP values are decreased or modified. Recalling that each `true`-outcome accounts for incrementing the sum of LCP values by one, we can bound the number of character comparisons incurred thereby with $L + p\hat{\ell}\lfloor \log p \rfloor$. Having accounted for all comparisons in `LCP-Merge`, this concludes the proof of Lemma 1. □

Algorithm 2: Binary LCP-Merge [10]

```
1  Function LCP-Merge((S_0, H_0), (S_1, H_1))
       Input: Sorted string arrays S_0, S_1 and LCP arrays H_0, H_1.
              String arrays have sentinels S_y[|S_y|] = ∞ for y ∈ {0,1}.
2      i_0 ← 0,  i_1 ← 0,  j ← 0                    // current indices into S_0, S_1, and S
3      h_0 ← 0,  h_1 ← 0                            // LCP values to the current head of S
4      while i_0 < |S_0| or i_1 < |S_1| do
           // Invariant: h_y = LCP(S_y[i_y], S[j-1]) for y ∈ {0,1}
5          s_0 ← S_0[i_0],  s_1 ← S_1[i_1]
6          if h_0 = h_1 then                        // need to compare additional characters
7              h' ← h_0                             // increment until h' = LCP(s_0, s_1)
8              while s_0[h'] ≠ $ and s_0[h'] = s_1[h'] do h' ← h' + 1
9              if s_0[h'] ≤ s_1[h'] then x ← 0 else x ← 1
10         else if h_0 < h_1 then x ← 1,  h' ← h_0          // s_1[h_0+1] < s_0[h_0+1]
11         else if h_0 > h_1 then x ← 0,  h' ← h_1          // s_0[h_1+1] < s_1[h_1+1]
           // Invariant: s_x ≤ s_{1-x} and h' = LCP(s_0, s_1)
12         S[j] ← s_x,  H[j] ← h_x,  j ← j + 1      // append smaller string to S
13         i_x ← i_x + 1,  h_x ← H_x[i_x],  h_{1-x} ← h'   // advance to next string
       Output: Sorted string array S with LCP array H
```

Lemma 1 only gives a bound for the character comparisons incurred by merging over all PEs and iterations of LCP-RQuick. This still leaves the possibility that local work is distributed unevenly between PEs and iterations which could lead to a longer critical path. We suspect that the total local work for merging on any PE can in fact be bounded further, e.g., by $O(n\hat{\ell}/p + \hat{\ell} \log p)$ over all iterations, but were unable to satisfactorily prove this. Somewhat unconventionally, we instead state the total local work summed over all PEs incurred by LCP-RQuick in the following theorem.

To determine the runtime of LCP-RQuick, we also need to consider the imbalance of local string arrays, which is influenced by the quality of the approximated median. For a version of the algorithm with a more complex median selection (RQuick$^+$), Axtmann is able to show that every PE has $O(n/p)$ elements with high probability, but is only able to conjecture, based on empirical evidence, that RQuick manages the same [2, Theorem 6.6, Conjecture 6.11]. The following theorem therefore includes an explicit precondition that the median approximation works well and manages to maintain $O(n/p)$ strings per PE. Because we do not make any general assumptions about the distribution of string lengths with respect to the lexicographical order, we assume that each string contributes $\hat{\ell}$ characters. This leads to several bounds containing the term $n\hat{\ell}/p$.

Theorem 1. *If Algorithm 1 maintains balanced workload for all* $\log p$ *iterations, then the algorithm can be implemented to run using local work* $O(n\hat{\ell}/p \log p + n/p \log \sigma)$*, latency* $O(\alpha \log^2 p)$*, and bottleneck communication volume* $O((\hat{\ell} \log^2 p + n\hat{\ell}/p \log p) \log \sigma)$ *bits. The sum of local work over all PEs is in* $O(D + n \log \sigma + (n + p\hat{\ell}) \log p)$*.*

Proof. Initial reduction to the nearest hypercube sends and receives at most one message of $O(n\hat{\ell}/p)$ characters on each PE. The subsequent random redistribution uses a hypercube communication pattern during which each string is sent at most once which yields latency $O(\alpha \log p)$ and communication volume $O(n\hat{\ell}/p)$ characters. Local sorting with MSD radix sort takes time $O(n\hat{\ell}/p + n/p \log \sigma)$ with a conservative estimate for the distinguishing prefix. Approximating the median in each iteration of the loop requires a binary tree reduction and single broadcast. Both have latency $O(\alpha \log p)$ per iteration which explains the overall polylogarithmic latency. The reduction only works on a constant number of strings and therefore contributes $O(\beta\hat{\ell} \log p \log \sigma)$ bits. Using LCP-aware linear search to locate the splitter requires local work $O(n/p + \hat{\ell})$ for a single scan of $\mathcal{S}_i$ and s in the worst case. Moving string arrays to the communication partner requires sending $O(n\hat{\ell}/p \log \sigma)$ bits in each iteration. LCP arrays only require $O(n/p \log \hat{\ell})$ additional bits to be sent which is dominated by the previous term. Finally, in the worst case merging requires local work in $O(n\hat{\ell}/p)$ per iteration.

For the sum of local work over all PEs, first note that the sum of LCP values is bounded by the size of the distinguishing prefix. Summed over all PEs, local sorting takes total local work in $O(D + n \log \sigma)$. Locating the median is in $O(n + p\hat{\ell})$ per iteration when multiplied by p. Finally, the total work for merging can be derived from Lemma 1 and is in $O(D + (n + p\hat{\ell}) \log p)$. All other local work is dominated by these terms and the bound from the theorem follows immediately. □

With Theorem 1, we are not able show an improvement in worst-case complexity of LCP-RQuick over regular RQuick. However, with the bound on local work over all PEs, if we assume that work for merging is distributed evenly over PEs, then this suggests that LCP-RQuick does in fact provide an improvement in expected merging time by a factor $\log p$. Though a theorem with an explicit bound on local work per PE is needed, to further support this claim. Even then, consider again that the main drawback of hypercube quicksort for strings is its communication-inefficiency since it has to send approximately half of the strings $\log p$ times. Exploiting LCP values to make local work more efficient does not change this and actually leads to additional communication. Given inputs with sufficiently large common prefixes, the communication overhead can be offset by the reduction in local work, which we demonstrate empirically with better sorting times as part of the experimental evaluation in Chapter 6. There are a number of ways the algorithm could be further improved by reducing the expected communication volume. Similar to PDMS, one could apply prefix approximation before sorting or even before the reduction to the nearest hypercube. By applying LCP compression it is possible to further reduce the number of characters sent per exchange. Also, median approximation could use character-based techniques to bound the expected number of characters per PE by $O(N/p)$.

3.2. Distributed Ordered Partitioning

The existing distributed string sorting algorithms—MS and PDMS—use a multi-way merge sort scheme which requires a global input array to be partitioned into p buckets $\mathcal{B}^0, \ldots, \mathcal{B}^{p-1}$ [18, 36, 11]. The buckets are required to be ordered, i.e., for $i < j$ all strings in bucket $\mathcal{B}^i$ must be smaller than any string in bucket $\mathcal{B}^j$. In the algorithm's next phase, elements are exchanged such that each bucket is assigned to its own PE. The relative bucket sizes directly influence the workload of PEs in subsequent processing steps and must therefore be bounded to ensure load balancing. For atomic objects, it is sufficient to consider only the number of elements to show that buckets are well-balanced. However, due to the multidimensionality of strings, where complexity also depends on the length of strings, it may be desirable to use the number of characters as criteria instead.

The multi-level merge sort variants presented in this thesis use the same partitioning scheme in a slightly more general form. Rather than being limited to yield a partition with p buckets, we may instead freely choose any divisor of p as the desired number of buckets. The space-efficient string sorting framework is yet more general and does not require that r is a divisor of p or even that $r \leq p$. Any bounds on worst-case imbalances must consider these relaxed preconditions. In Section 3.2.1, we first restate the algorithm from [36] to accept this new input specification. Sections 3.2.2 and 3.2.3 adapt string- and character-based regular sampling techniques and give amended bounds for buckets imbalance.

3.2.1. Partitioning Algorithm

In addition to the string array $\mathcal{S}_i$, each PE also receives the desired number of buckets r as input which must be the same value on all PEs. We assume all input strings to be globally unique which may be assured with an appropriate tie-breaking scheme, e.g., using the PE rank and local array index from which a string originates. The *oversampling factor* v gives the number of samples to be drawn per PE. For single-level merge sort, the oversampling factor is usually chosen to as $\Theta(p)$ in practice when using regular sampling. Input arrays are required to be sorted locally which is essential for regular sampling techniques. The following algorithm and the analysis thereafter assume perfectly balanced string and character arrays, i.e., $|\mathcal{S}_i| = n/p$ and $\|\mathcal{S}_i\| = N/p$ for every PE i. However, it is possible to modify both such that they remain valid for only asymptotically balanced arrays in $\Theta(n/p)$ and $\Theta(N/p)$. The idea is broadly to maintain uniform spacing between samples, by distributing the pv samples according to the number of strings and characters on each PE [36].

The partitioning algorithm consists of the following four main steps:

(1) **Local Sampling:** On each PE i, draw v samples from $\mathcal{S}_i$ using one of the sampling techniques from the following sections to obtain $\mathcal{V}_i$.

(2) **Distributed Sorting:** Globally sort the pv samples using a distributed string sorting algorithm, e.g., LCP-RQuick, to obtain the distributed array $\mathcal{V}$.

(3) **Splitter Selection:** Choose $r-1$ evenly spaced splitters $f_i = \mathcal{V}[(ipv/r) - 1]$ for $i = 1, \ldots, r-1$ using a distributed selection algorithm. For simplicity, we assume that r divides pv which means that no rounding is necessary. Using an all-gather operation the $r-1$ splitters are communicated to every PE.

(4) **Bucket Computation:** On each PE i, the (sorted) input array is split into intervals to obtain ordered buckets $\mathcal{B}_i^j := [s \in \mathcal{S}_i \mid f_j < s \leq f_{j+1}]$ with sentinels $f_0 = -\infty$ and $f_r = +\infty$. Splitter positions within $\mathcal{S}_i$ can be found using binary search or a single linear scan.

For convenience in later analysis, we define global buckets $\mathcal{B}^j := \mathcal{B}_0^j \cup \cdots \cup \mathcal{B}_{p-1}^j$ as the set of strings in the jth bucket on all PEs. The algorithm fulfills the postcondition that buckets $\mathcal{B}^0, \ldots, \mathcal{B}^{r-1}$ are ordered.

We do not provide combined runtime of the partitioning algorithm as it depends on the chosen sampling technique and distributed sorting algorithm. Splitter selection can be implemented in time $O(\alpha \log p + \beta r\hat{\ell} \log \sigma)$. This includes a prefix-sum to compute the offset of every PE into $\mathcal{V}$ and the all-gather which can be thought of as separate gather and broadcast operations in the same time. Locating the splitters in step (4) using $r-1$ individual binary searches requires time $O(r\hat{\ell} \log \frac{n}{p})$. Because the splitters are sorted we can also use a single linear scan over $\mathcal{S}_i$ in time $O(\max_i \mathcal{D}(\mathcal{S}_i) + r\hat{d})$ with an up-to-date LCP array. The term $r\hat{\ell}$ is required to find LCP values of splitters. More conservatively, with the reasonable assumption that $r\hat{d} = O(N/p)$, this can be simplified to $O(N/p)$ which we may use to absorb the term into more complex bounds.

3.2.2. String-Based Regular Sampling

Because the input arrays are sorted, it is possible to apply regular sampling. Here, as the name implies, samples are chosen at regular intervals which provides good worst-case guarantees for the maximum size of buckets after partitioning and has been applied successfully to parallel sorting algorithms in the past [37]. The following string- and character-based sampling techniques are directly based on existing work [36, 11]. Proofs of bucket sizes bounds are only slightly generalized to account for the choice of $r-1$ rather than $p-1$ splitters. For string-based regular sampling, we assume that the number of input strings $|\mathcal{S}_i|$ is divisible by $v+1$ on every PE. We define $\omega = |\mathcal{S}_i|/(v+1)$ as the distance between samples and choose

$$\mathcal{V}_i := \{\mathcal{S}_i[\omega j - 1] \mid j \in [1, v]\}$$

accordingly in time $O(n/p)$. Theorem 2 proves that with these samples, the resulting buckets are well-balanced with respect to the number of strings. It is, however, not possible to give general bounds for the number of characters per bucket. We first reiterate the following lemma on the density of samples from the existing literature.

Lemma 2 (String-Based Sampling Density [11, Lemma 1.1])**.** *Let $\mathcal{S}_i' = \{s \in \mathcal{S}_i \mid a \leq s \leq b\}$ be an arbitrary contiguous subsequence of $\mathcal{S}_i$ for $i \in [0, p)$. With $|\mathcal{S}_i' \cap \mathcal{V}| = k$ it must hold that $|\mathcal{S}_i'| \leq (k+1)\omega$.*

Theorem 2. *Using string-based regular sampling with v samples per PE, every global bucket $\mathcal{B}^j$ obtained using the partitioning algorithm contains at most $n/r + n/v$ strings.*

Proof. Let $\mathcal{V}_i^j = \mathcal{B}_i^j \cap \mathcal{V}_i$ be the samples on PE i that fall into the jth bucket. Using Lemma 2 it follows that $\mathcal{B}_i^j \leq (|\mathcal{V}_i^j| + 1)\omega$. By definition in step (3) of the partitioning algorithm,

splitters f_j and f_{j+1} are separated by $pv/r - 1$ strings and thus $\sum_{i=0}^{p-1} |\mathcal{V}_i^j| = pv/r$ when including f_{j+1}. The stated bound follows directly using the definition of $\mathcal{B}^j$ and assuming balanced input arrays as follows:

$$\begin{aligned} |\mathcal{B}^j| = \sum_{i=0}^{p-1} |\mathcal{B}_i^j| \quad &\leq \sum_{i=0}^{p-1} \Big(|\mathcal{V}_i^j| + 1 \Big) \omega \qquad = \omega \Big(\frac{pv}{r} + p \Big) \quad = \frac{|\mathcal{S}_i|}{v+1} \Big(\frac{pv}{r} + p \Big) \\ &= \frac{n}{p(v+1)} \Big(\frac{pv}{r} + p \Big) \quad \leq \frac{n}{pv} \Big(\frac{pv}{r} + p \Big) = \frac{n}{r} + \frac{n}{v} \end{aligned}$$

□

To summarize, using string-based regular sampling yields buckets with a maximum imbalance of n/v strings if the input is initially balanced: which is equivalent to an imbalance factor $1 + r/v$. A choice of $v = \Theta(r)$ means that the number of strings per bucket is in $\Theta(n/r)$.

3.2.3. Character-Based Regular Sampling

For character-based regular sampling, the distance between samples is chosen with respect to the character array $\mathcal{C}(\mathcal{S}_i)$. As with string-based sampling, we assume that the number of characters $\|\mathcal{S}_i\|$ is divisible by $v + 1$ on each PE and choose the appropriate sampling distance $\omega' = \|\mathcal{S}_i\|/(v+1)$. To ensure that samples are unique, we require that the length of the longest string $\hat{\ell}$ is not greater than the distance between samples, i.e., $\hat{\ell} \leq \omega'$. Using ω' we obtain positions in the character array at $\omega' j - 1$ for $j \in [1, v]$. These *character samples* do not necessarily line up with string boundaries and we therefore need to introduce an additional transformation to arrive at the final string samples. Positions that are not already at the first character of a string, are shifted right by at most $\hat{\ell} - 1$ characters to the beginning of the next string. This transformation can be formalized on PE i as a function

$$\phi_i \colon [0, \|\mathcal{S}_i\|) \to [0, |\mathcal{S}_i|) \quad \text{with} \quad m \mapsto \min \big\{ j \in [0, |\mathcal{S}_i|) \colon m \leq \|\mathcal{S}_i[0, j)\| \big\}$$

which maps character positions to string positions. The returned position is the leftmost string beginning at or after the mth character in $\mathcal{C}(\mathcal{S}_i)$. If the length of strings is known and can be queried in constant time, then sampling character positions and mapping them to strings only requires a single scan over $\mathcal{S}_i$ in time $O(n/p)$. Using ϕ_i the set of samples on PE i is defined as

$$\mathcal{V}_i \coloneqq \big\{ \mathcal{S}_i[\phi(\omega' j - 1)] \mid j \in [1, v] \big\}.$$

Following a similar approach to string-based sampling, we can bound the maximum number of characters in each bucket. Again, we first restate a lemma from the existing work about the density of samples with respect to the number of characters.

Lemma 3 (Character-Based Sampling Density [11, Lemma 2.2])**.** *Let $\mathcal{S}_i' = \{s \in \mathcal{S}_i \mid a \leq s \leq b\}$ be an arbitrary contiguous subsequence of $\mathcal{S}_i$ for $i \in [0, p)$. With $|\mathcal{S}_i' \cap \mathcal{V}| = k$ it must hold that $\|\mathcal{S}_i'\| = |\mathcal{C}(\mathcal{S}_i')| \leq (k+1)(\omega' + \hat{\ell}\,)$.*

Lemma 3 suffices to show the following theorem about the number of characters per bucket.

Theorem 3. *Using character-based regular sampling with v samples per PE, every bucket $\mathcal{B}^j$ obtained using the partitioning algorithm contains at most $N/r + N/v + (p + vp')\hat{\ell}$ characters.*

Proof. Let $\mathcal{V}_i^j = \mathcal{B}_i^j \cap \mathcal{V}_i$ be the local sample buckets as in the proof of Theorem 2. Using Lemma 3 yields the analogous bound $\|\mathcal{B}_i^j\| \leq (|\mathcal{V}_i^j| + 1)(\omega' + \hat{\ell}\,)$. The equalities for $\mathcal{V}_i^j$ and $\mathcal{B}^j$ remain identical which suffices to obtain the stated bound.

$$\begin{aligned} \|\mathcal{B}^j\| = \sum_{i=0}^{p-1} \|\mathcal{B}_i^j\| \quad &\leq \sum_{i=0}^{p-1} \Big(|\mathcal{V}_i^j| + 1\Big)(\omega' + \hat{\ell}\,) \quad &&= \Big(\frac{pv}{r} + p\Big)\left(\frac{\|\mathcal{S}_i\|}{v+1} + \hat{\ell}\right) \\ &\leq \Big(\frac{pv}{r} + p\Big)\left(\frac{N}{pv} + \hat{\ell}\right) \quad &&= \frac{N}{r} + \frac{N}{v} + \Big(\frac{pv}{r} + p\Big)\hat{\ell} \end{aligned}$$

□

Note the term $(p + pv/r)\hat{\ell}$ in Theorem 3 which can be simplified to $\Theta(p\hat{\ell})$ if $v = \Theta(r)$. This means that in contrast to string-based sampling, the imbalance does not only depend on the number of partitions, but also on the number of PEs. Intuitively, this is due to the fact that on each PE the string samples may have been shifted right by up to $\hat{\ell} - 1$ characters.

3.3. Distributed Group Assignment

As part of the multi-level merge sort algorithm in Chapter 4, we encounter the problem of assigning elements from partitioned arrays to groups of multiple PEs. Going forward, we refer to the problem as (distributed) group assignment. The resulting assignment needs to ensure that each PE receives approximately the same amount of data to ensure load balancing. Again, because our algorithms work with variable-length strings, an assignment of elements may be considered well-balanced with respect to either the number of strings or the number of characters on each PE. Additionally, the number of sent and received messages should be bounded, with an asymptotic lower bound being a constant number of messages per group.

In addition to the customary string array $\mathcal{S}_i$, each PE also obtains a partitioning of that array into r buckets $\mathcal{B}_i^0, \ldots, \mathcal{B}_i^{r-1}$. We only require that buckets are a partition of $\mathcal{S}_i$, i.e., the union of all buckets on PE i is equal to $\mathcal{S}_i$ and their intersection is empty, but make no assumption about the lexicographical order of contained strings. As before, strings are assumed to be unique and we can thus define global buckets $\mathcal{B}^j = \bigcup_{i=0}^{r-1} \mathcal{B}_i^j$ using set union. The number of buckets must divide the number of PEs into groups of equal size $p' \coloneqq p/r \geq 1$. Intuitively, group j consists of the p' consecutive PEs with ranks jp' to $(j+1)p' - 1$. The task of algorithms in this section is to assign all strings from the p buckets $\mathcal{B}_0^j, \ldots, \mathcal{B}_{p-1}^j$ to the PEs of group j. Our algorithms are, with one exception, not explicit on how such an assignment is represented, since we consider this to be an implementation detail. It can be conceptualized as further splitting the buckets into p' sets, one for each PE of the group. Based on the local string assignments, we globally define $\mathcal{A}_i^j$ as the set of all strings assigned to the ith PE of group j. Note that actually applying the assignment is *not* part of the algorithms described in this section and instead part of a subsequent and separate string exchange phase.

The case of $p' = 1$ is trivial as there is only the single valid assignment $\mathcal{A}_0^j = \mathcal{B}^j$ for every j. This is only used as convention to avoid special cases in the definition of multi-level merge sort. We assume that input arrays are initially perfectly balanced, i.e., each PE needs to send n/p strings and N/p characters. Using appropriate rounding behavior, the algorithms

can be adapted to work with only asymptotically balanced arrays. We also assume that perfect *intra-group* balance is possible, i.e., bucket sizes $|\mathcal{B}^j|$ and $\|\mathcal{B}^j\|$ are divisible by p' for every group j. However, because the partitioning algorithm does not yield perfectly uniform buckets, we introduce *inter-group* string and character imbalance ε and δ with corresponding bounds

$$\left|\mathcal{B}^j\right| \leq (1+\varepsilon)\frac{n}{r} \qquad \text{and} \qquad \left\|\mathcal{B}^j\right\| \leq (1+\delta)\frac{N}{r}. \tag{3.1}$$

The following sections introduce three algorithms that solve the assignment problem with increasing complexity. Section 3.3.1 introduces a first attempt that is entirely oblivious to the amount of data per PE, but always sends and receives exactly one message to every group on each PE. Conversely, the second approach, in Section 3.3.2, guarantees uniform distribution of data but cannot offer a nontrivial upper bound on the number of messages received per PE. Finally, Section 3.3.3 proposes an approach that maintains equal distribution of elements while bounding the number of sent and received messages by the number of groups at the cost of additional algorithmic complexity. In their paper on multi-level atomic sorting, Axtmann et al. [6] cover the same "data redistribution problem" for atomic objects. Compared to their work, the following algorithms are broadly similar, but somewhat more general, since we cannot assume that elements remain globally balanced after redistribution.

3.3.1. Grid-Wise Assignment

The first assignment technique is termed *grid-wise* as it views the PEs as a grid with dimensions $p' \times r$ and only allows communication along rows and columns thereof. Intuitively, groups of p' PEs make up rows, while columns join the lth PE of each group which yields coordinates

$$\operatorname{row}(i) \coloneqq \lfloor i/p' \rfloor \quad \text{and} \quad \operatorname{col}(i) \coloneqq i \bmod p'.$$

This approach is primarily intended to support 2-dimensional cartesian MPI communicators and works by exchanging buckets along columns using p' parallel all-to-all exchanges. Formally, PE i assigns all strings in bucket $\mathcal{B}_i^j$ to PE $jp' + \operatorname{col}(i)$. This leads to all buckets in a column being assigned to

$$\mathcal{A}_i^j \coloneqq \bigcup_{l=0}^{r-1} \mathcal{B}_{lp'+i}^j$$

on the ith PE of group j. It is immediately clear that grid-wise group assignment may yield undesirable imbalances between PEs, which we formalize with Theorem 4.

Theorem 4. *Using grid-wise group assignment, each PE needs to send and receive strings from exactly r PEs. There exist cases where a single PE receives $\Omega(n/p')$ strings or $\Omega(N/p')$ characters.*

Proof. The first part holds by definition and must be true to allow a column-wise all-to-all exchange. For the second part, consider the case where every PE in a column c sends all its strings to the same group j. This happens if $|\mathcal{B}_i^j| = |\mathcal{S}_i| = n/p$ for PE $i = lp' + c$ on every row l. We claim that such bucket sizes are actually possible without violating the

requirement for initially balanced arrays. The number strings assigned to PE c of group j is therefore

$$|\mathcal{A}_c^j| = \left|\bigcup_{l=0}^{r-1} \mathcal{B}_{lp'+c}^j\right| = \sum_{l=0}^{r-1} \left|\mathcal{B}_{lp'+c}^j\right| = r\frac{n}{p} = \frac{n}{p'}.$$

An analogous argument can be made for the number of characters by substituting the number of strings n for characters N and usages of $|\cdot|$ for $\|\cdot\|$. □

Note the difference between n/p and n/p' in Theorem 4. The latter directly implies that an intra-group string or character imbalance by a factor of $\Omega(r)$ is possible when using grid-wise assignment. The following approach aims to resolve this issue using a technique from the distributed algorithm *basic toolbox*.

3.3.2. Simple Assignment

As a simple approach to ensure well-balanced assignment, we view the problem as filling a sequence of *slots* using a set of items of possibly variable size. Each item, i.e., string, takes up a defined number of slots—either one per string or one per character depending on whether string- or character-based assignment is desired. For each group of PEs, we allocate a matching number of open slots, sufficient to receive all strings from the corresponding bucket in the partition. Slots and strings are both enumerated and we assign the ith string to the ith open slot. Now, slots are distributed over the p' PEs of each group in equally sized chunks to arrive at the final assignment of strings to PEs. PEs only need to determine the total number of slots and how many slots are used by PEs with smaller rank to compute the assignment. Both values can easily be computed using simple collective operations. Additional work is required to adapt the technique to a character-based approach, where the number of slots corresponds to the number of characters. Figure 3.1 illustrates the working principle of simple assignment for a system with six PEs and three groups.

String-Based Assignment

For the string-based variant, the number of slots in group j is equal to the number of strings in the group (i.e., $|\mathcal{B}^j|$) which can be computed using an all-reduce operation. The all-reduce is performed over r-sized vectors containing $|\mathcal{B}_i^0|, \ldots, |\mathcal{B}_i^{r-1}|$. Additionally, an exclusive prefix-sum operation over the same vectors is performed to determine the slot to which PE i should assign the first string in each of its buckets. The exclusive prefix-sum yields the number of slots used by PEs to the left of i. Algorithm 3 describes how the assignment can be computed using the results of all-reduce and prefix-sum operations.

The following theorems show that the algorithm can be efficiently implemented and yields a well-balanced assignment with respect to the number of strings per PE. However, Theorem 6 also shows that it is not possible to give nontrivial bounds on the number of messages received during a string exchange using the assignment.

Theorem 5. *Algorithm 3 requires local work $O(n/p)$, latency $O(\alpha \log p)$, and bottleneck communication volume $O(r \log n)$ bits.*

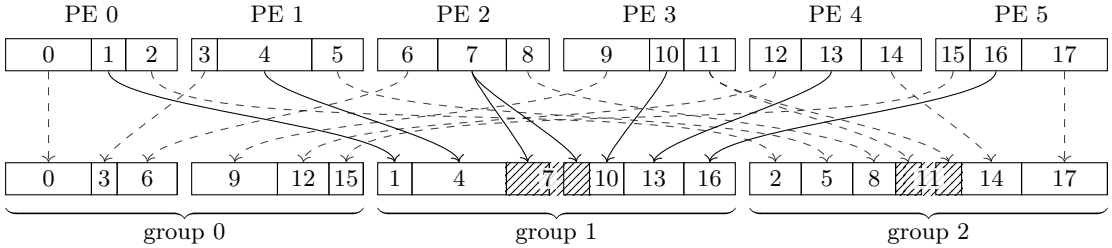

Figure 3.1.: An exchange using simple group assignment for $p = 6$ PEs with $r = 3$ groups of size $p' = 2$. Buckets are globally numbered sequentially from 0 to 17. Assignment is highlighted for the second group. Hatching indicates overlapping buckets.

Algorithm 3: Simple String-Based Assignment

Input: On each PE i string arrays $\mathcal{B}_i^0, \ldots, \mathcal{B}_i^{r-1}$

1 **for** $j \leftarrow 0$ **to** $r-1$ **do**
2 $\quad k \leftarrow \sum_{l=0}^{i-1} |\mathcal{B}_l^j|$ *// computed with a single r-ary exclusive prefix-sum*
3 $\quad m \leftarrow |\mathcal{B}^j|/p'$ *// and an r-ary all-reduce before the first iteration*
4 $\quad$ **for** $s \in \mathcal{B}_i^j$ **do**
5 $\quad\quad$ assign string s to PE $\lfloor k/m \rfloor$ of group j
6 $\quad\quad k \leftarrow k+1$

Output: A balanced assignment $\mathcal{A}_i^j$ of strings to PEs

Proof. Local work is determined by the inner loop of Algorithm 3 which makes $\sum_{j=0}^{r-1} |\mathcal{B}_i^j| = |\mathcal{S}_i| = n/p$ iterations. All-reduce and prefix-sum operations each incur latency $O(\alpha \log p)$. Both operate on vectors with r entries of $O(\log n)$ bits to store the global number of strings which gives the communication volume. □

Theorem 6. *If $|\mathcal{B}^j|$ is a multiple of p' for every $j \in [0, r)$, then Algorithm 3 assigns $|\mathcal{B}^j|/p'$ strings to the ith PE of group j. It is possible for a PE to be assigned strings from $\Omega(p)$ PEs.*

Proof. Let $|\mathcal{B}^j|$ be divisible by p' for every group j as required by the theorem. Note that the algorithm can be adapted to work without this assumption by using $m = \lceil |\mathcal{B}^j|/p' \rceil$ instead. The first part of Theorem 6 must hold because values of k in line 5 are globally unique. Therefore, exactly $|\mathcal{B}^j|/p'$ strings are assigned to every set $\mathcal{A}_i^j$. For the worst-case lower bound on received messages consider a case where $\Theta(p)$ consecutive PEs send small buckets containing $|\mathcal{B}_i^j| = \Theta(n/p^2)$ strings each to the same group j. The algorithm assigns the strings in these buckets to $O(1)$ PEs which therefore receive $\Omega(p)$ messages during a string exchange. □

Theorem 6 proves that Algorithm 3 computes an assignment with perfect intra-group string balance. Based on the string imbalance parameter ε, we also achieve optimal global imbalance

$$|\mathcal{A}_i^j| = \frac{|\mathcal{B}^j|}{p'} \leq \frac{1}{p'}(1+\varepsilon)\frac{n}{r} = (1+\varepsilon)\frac{n}{p}.$$

However, even for a constant number of partitions $r = O(1)$ it is possible that a single PE needs to receive $\Theta(p)$ messages. In Section 3.3.3 we provide an algorithm that reduces this to $O(r)$.

Character-Based Assignment

The preceding string-based approach can easily be adapted to yield an assignment that is balanced with respect to the number of characters instead. Here, the number of slots per group is $\|\mathcal{B}^j\|$ which can be implemented by using vectors containing character counts $\|\mathcal{B}_i^j\|$ rather than string counts for all-reduce and prefix-sum operations. Strings are assigned to a PE while at least one character slot is still available, regardless of the length of the assigned string. We show that this allows at most an imbalance of $\hat{\ell} - 1$ characters. Note that the corresponding slots taken up by a string on the previous PE remain unused on the following PE. Algorithm 4 gives a formal description of how this behavior can be implemented. The only changes when compared to string-based assignment are the usage of $\|\cdot\|$ where appropriate and incrementing k by the number of characters $|s|$ instead of 1 in line 6. To avoid edge cases where no strings are assigned to a PE, we require that the length of the longest string is less than the number of slots per PE, i.e., $\hat{\ell} < \|\mathcal{B}^j\|/p'$ for all groups j.

Algorithm 4: Simple Character-Based Assignment

Input: On each PE i string arrays $\mathcal{B}_i^0, \ldots, \mathcal{B}_i^{r-1}$

```
1 for j ← 0 to r − 1 do
2     k ← Σ_{l=0}^{i−1} ‖B_l^j‖        // computed with a single r-ary exclusive prefix-sum
3     m ← ‖B^j‖/p'                     // and an r-ary all-reduce before the first iteration
4     for s ∈ B_i^j do
5         assign string s to PE ⌊k/m⌋ of group j
6         k ← k + |s|
```

Output: A balanced assignment $\mathcal{A}_i^j$ of strings to PEs

We can provide similar bounds for local work and communication complexity of Algorithm 4 as for the string-based version. The following theorems assume that string lengths are known and can be queried in constant time; otherwise, a lower bound of $O(N)$ could not be improved upon. In practice, we store strings as pairs of pointer and length which provides the desired constant lookup time.

Theorem 7. *Algorithm 4 requires local work $O(n/p)$, latency $O(\alpha \log p)$, and bottleneck communication volume $O(r \log N)$ bits.*

Proof. The proof is identical to Theorem 5 except that collective operations use integers of $\log N$ bits. For local work also consider that local buckets sizes $\mathcal{B}_i^0, \ldots, \mathcal{B}_i^{r-1}$ can be computed in time $O(n/p)$. □

Theorem 8. *If $\hat{\ell}$ is the length of the longest string, then Algorithm 3 assigns less than $\|\mathcal{B}^j\|/p' + \hat{\ell}$ characters to each PE in every group j.*

Proof. Let s be a string from bucket $\mathcal{B}_i^j$ and let k be the associated value in line 5 of the algorithm. The string is assigned to PE $k/(\|\mathcal{B}^j\|/p')$ which must have less than $\|\mathcal{B}^j\|/p'$ assigned characters. Considering the length of the longest string, s adds at most $\hat{\ell}$ additional characters to the PE. The stated bound follows directly. □

Theorem 8 proves that with simple character-based assignment in contrast to the string-based version we only get close to maintaining intra-group character balance after a subsequent string exchange. For the global character imbalance, using the parameter for character-imbalance, we get a bound of

$$\left\|\mathcal{A}_i^j\right\| \leq \frac{\left\|\mathcal{B}^j\right\|}{p'} + \hat{\ell} \leq (1+\delta)\frac{N}{p} + \hat{\ell}.$$

As with the character-based sampling techniques from Section 3.2 the length of the longest string directly influences the worst-case bound. Thus, if there exist extremely long strings, for example of length $\Theta(N/p)$, it is not possible to make meaningful load-balancing guarantees.

3.3.3. Deterministic Assignment

With the previous approach to group assignment we already have an algorithm that yields optimal intra-group balance. To resolve that approach's main drawback—PEs being assigned strings from $\Omega(p)$ sources—we adapt a technique proposed by Axtmann and Sanders [3]. Consistent with the original naming convention, we call this approach *deterministic* assignment which stands in contrast to a randomized approach from the same publication. The following is not an exhaustive analysis of the algorithm but instead only serves to motivate the idea that an efficient algorithm exists which can compute an assignment with the desired properties. Our contribution exclusively consists in adapting the group assignment technique to a character-based approach.

General Approach

In this section, we describe the basic idea of deterministic assignment and define an according black-box function `DeterministicAssign` [6, 2]. This function provides relevant guarantees which we use to define deterministic string- and character-based assignment in the subsequent sections. Because of the more abstract perspective on the problem, we also require slightly altered definitions and conventions. Instead of receiving buckets and using functions $|\cdot|$ or $\|\cdot\|$, the function `DeterministicAssign` directly obtains bucket sizes b_i^j as input. We require that local input sizes are balanced and define the global input size $m := \sum_{i=0}^{p-1}\sum_{j=0}^{r-1} b_i^j$ to avoid ambiguities with n and N. The meanings of p, r, and p' remain unchanged.

The algorithm is based on the observation that, when many consecutive PEs send small buckets to the same group, the messages must be distributed among multiple PEs, to maintain $O(r)$ messages on each one. To handle such scenarios properly, we choose to assign small and large buckets separately, using different techniques. For our purposes, we consider a bucket to be small if it has size at most $m/2pr$. Small buckets are assigned first, with a naive technique using only local information. PE i simply assigns any small bucket b_i^j to PE

$\lfloor i/r \rfloor$ of group j. Any PE is assigned at most r buckets of cumulative size at most $m/2p$, leaving at least half its total capacity of m/p unused.

It now remains to distribute the leftover large buckets while taking the residual capacities into account. This is again accomplished by viewing the problem as filling a sequence of open slots, similar to the simple assignment from Section 3.3.2. However, because PEs have different residual capacities, it is not possible to find an assignment using a single prefix-sum as before. Each group must instead work individually to compute an assignment for the buckets assigned to it. Conceptually, this works by performing separate prefix-sums over residual capacities and sizes of unassigned buckets, in each group. The resulting sorted sequences of integers must then be merged such that the bucket beginning at the ith element is preceded by the PE containing the ith open slot. Buckets that do not line up exactly with PE boundaries are split, and simply wrap over to the next PE or, at most, to the two succeeding PEs. Axtmann et al. propose an algorithm that implements this assignment strategy, based on a merging technique for shared-memory systems [22]. The algorithm requires time $\text{Exch}(p, O(r \log m), r)$ to exchange bucket sizes and latency $O(\alpha \log p')$ for the merging operation. Crucially, each PE has residual capacity between $m/2p$ and m/p on one hand, and each leftover large bucket contains at least $m/2pr$ elements on the other hand. Thus, the number of large buckets assigned to any PE is less than $\frac{m}{p} / \frac{m}{2pr} = 2r$ and a single large bucket spans at most $1 + \frac{m}{p} / \frac{m}{2p} = 3$ PEs (considering the wrapping behavior described above). Hence, taking both small and large buckets into account, each PE receives messages from $O(r)$ other PEs. With the previous assumption that inputs are perfectly balanced, this yields the following lemma about the resulting assignment.

Lemma 4. *For perfectly balanced inputs, the function* `DeterministicAssign` *ensures that each PE is assigned exactly m/p elements from $O(r)$ other PEs.*

The multi-level merge sort algorithms introduced in Chapter 4 cannot generally assume that inputs are perfectly balanced. Instead, only asymptotic guarantees in $O(n/p)$ strings or worse $O(N/p + \hat{\ell})$ characters are possible. A version of `DeterministicAssign` can be implemented such that the resulting assignment maintains the guarantees from Lemma 4. Clearly, a bucket greater than m/p elements may wrap over to more than 2 additional PEs, but a bound of $O(n/p)$ guarantees that the result is in $O(1)$ at least in the case of strings. As is always the case for strings in a distributed context, extreme values of $\hat{\ell}$ may lead to unavoidable imbalances. Finally, based on the work by Axtmann et al. [6, Theorem 1], we derive the following statement about the runtime of `DeterministicAssign`:

Theorem 9. *For balanced inputs, the function* `DeterministicAssign` *requires runtime in $2 \cdot \widetilde{\text{Exch}}(p, r \log m, O(r))$.*

The factor two in Theorem 9 stems from having to first send bucket sizes to the assigning group and then needing to return the assignment back to the sender. Remaining terms for the merging of slot sequences are dominated by these exchanges; hence, the usage of $\widetilde{\text{Exch}}(\cdot)$. The preceding theorem can be used to assert that the communication of group assignment as a whole is dominated by the time of string exchange phases when used as part of multi-level string merge sort.

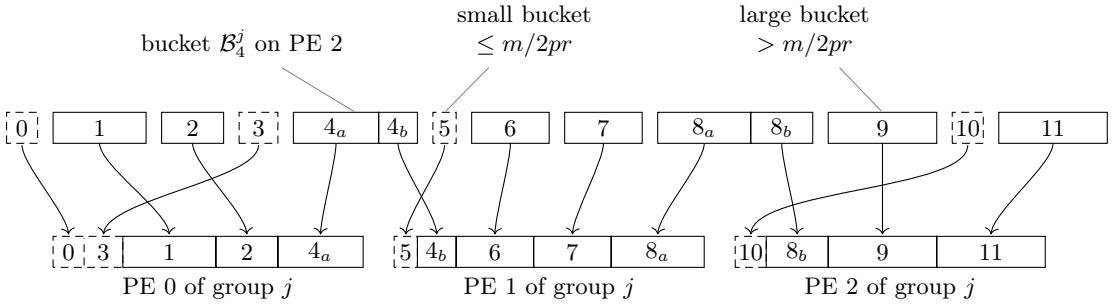

Figure 3.2.: An exchange using deterministic group assignment for $p = 12$ PEs with $r = 4$ groups of size $p' = 3$. Only buckets and communication for a single group j are shown. Buckets are numbered according to source PE. Small buckets are drawn as ⬚. Large buckets □ are labeled a and b if split during assignment.

String-Based Assignment

For a string-based approach, the result of `DeterministicAssign` can immediately be used as the final group assignment. We simply define bucket sizes as the number of strings, i.e., $b_i^j := |\mathcal{B}_i^j|$. The desired guarantees on balance and number of messages received can be derived directly from Lemma 4 with $m := n$. Complexity of the algorithm is identical and therefore subject to the bound from Conjecture 9.

Corollary 1. *If $|\mathcal{S}_i| = \Theta(n/p)$ for every $i \in [0, p)$, then deterministic string-based assignment ensures that each PE is assigned $\Theta(n/p)$ strings from $O(r)$ other PEs.*

As before, string-based techniques do not allow us to establish nontrivial guarantees on the balance of characters. The following section therefore proposes a character-based version of the algorithm.

Character-Based Assignment

Providing a character-based version of deterministic assignment only requires a number of minor alterations. First, we use character counts to define bucket sizes $b_i^j := \|\mathcal{B}_i^j\|$ and the number of elements $m := N$. We also need to map the result obtained using `DeterministicAssign` from an assignment of characters to one of strings. For this purpose, we define the function's return value as r sequences $\mathcal{D}_i^0, \ldots, \mathcal{D}_i^{r-1}$ on each PE. Each sequence $\mathcal{D}_i^j$ contains $O(r)$ elements from $[0, r) \times [0, N)$. Here the pair $(i', d) \in \mathcal{D}_i^j$ specifies that d characters from bucket $\mathcal{B}_i^j$ on PE i should be assigned to PE i' of group j. To avoid ambiguities, we say that PE i has been *allocated* d character slots for PE i' to draw a distinction between the final *assignment* of strings. For correctness of the following algorithm, we use the fact that the sum of interval sizes in each of these sequences is equal to the total bucket size, i.e., $\sum_{(\cdot,d)\in\mathcal{D}_i^j} d = b_i^j = \|\mathcal{B}_i^j\|$. We do not make any further assumptions about the order of entries in $\mathcal{D}_i^j$.

Algorithm 5 describes how the transformation from character to group assignment can be performed. Similar to simple assignment, strings are assigned to a PE while at least one

Algorithm 5: Deterministic Character-Based Assignment

Input: Buckets $\mathcal{B}_i^0, \ldots, \mathcal{B}_i^{r-1}$ and character assignments $\mathcal{D}_i^0, \ldots, \mathcal{D}_i^{r-1}$ obtained from `DeterministicAssign`$(\|\mathcal{B}_i^0\|, \ldots, \|\mathcal{B}_i^{r-1}\|)$ on each PE i.

```
for j ← 0 to r − 1 do
    l ← 0, c ← 0                          // index of current string, characters to the left
    b ← 0                                 // character boundary to the next PE
    for (i′, d) ∈ D_i^j do
        b ← b + d
        while c < b do                    // at least one slot must be unused
            assign string B_i^j[l] to PE i′ of group j
            c ← c + |B_i^j[l]|, l ← l + 1
```

Output: A balanced assignment $\mathcal{A}_i^j$ of strings to PEs

character slot defined by $\mathcal{D}_i^j$ is still available. Any excess characters, which wrap over PE boundaries, are deducted from the next PE. As before, this means that a PE may receive up to $\hat{\ell} - 1$ characters more than actually allocated. Note that if buckets were assigned as a whole by `DeterministicAssign`, then strings line up exactly with the assignment and no over-allocation of strings can occur. Because each PE receives at most one bucket whose end has been truncated, the over-allocation can only occur once. With $m \coloneqq N$ the rest of the following theorem follows directly from Lemma 4.

Theorem 10. *If $\|\mathcal{S}_i\| = O(N/p)$ for every $i \in [0, p)$, then deterministic character-based assignment ensures that each PE is assigned at most $O(N/p + \hat{\ell})$ strings from $O(r)$ PEs.*

Character-based assignment adds runtime in $O(n/p)$ to iterate over string arrays—first to compute buckets sizes and then to map assigned character slots to strings. However, because the number of buckets itself remains unchanged, so does the runtime of the call to `DeterministicAssign`. The bound from Conjecture 9 therefore still applies with communication in $2 \cdot \widetilde{\text{Exch}}(p, r \log N, O(r))$.

This concludes our investigation of distributed group assignment techniques. With deterministic character-based assignment we have an algorithm that ensures sufficient load balance for our multi-level merge sort algorithms. This will allow us to implement the string exchange phases in time $\text{Exch}(p, O(N/p + k^2 r\hat{\ell}), O(r))$ where k is the number of levels. However, in the experimental evaluation, we rely on grid-wise assignment as the technique of choice. So long as inputs can be assumed to be randomly distributed, no significant imbalance should be incurred in practice.

4. Multi-Level String Merge Sort

In Section 2.4, we already introduced two distributed string merge sort algorithms—MS and PDMS—from the existing work on distributed-memory string sorting algorithms [18, 36, 11]. Both algorithms work by computing an ordered partition of the input into p buckets and redistributing elements onto their final PE in a single pass. Hence, we refer to them as *single-level* algorithms. The ordered partitions are defined by $p-1$ splitters which are obtained using the partitioning algorithm from Section 3.2. Splitters are determined by sampling the input strings, globally sorting the samples, and finally drawing splitters at regular intervals. Using regular sampling techniques requires $\Theta(p)$ samples to be drawn on each PE to achieve well-balanced buckets. This results in $\Theta(p^2)$ global samples needing to be sorted in total. For inputs with small N/p ratio and less clearly for small n/p ratios, i.e., cases where proportionately few characters or strings are distributed over many PEs, the step of sorting samples can come to dominate the runtime of single-level MS and PDMS. Put more formally, if we assume a simple setting, where a generic hypercube quicksort algorithm is used to sort $\Theta(p)$ samples of length at most $\hat{\ell}$, then we expect communication time in $T_{\text{sort}} \coloneqq O(\alpha \log^2 p + \beta p \hat{\ell} \log p \log \sigma)$. A single-level merge sort algorithm that works by sending all characters using direct exchanges, has requires at least communication time in $T_{\text{exch}} \coloneqq O(\alpha p + \beta N/p \log \sigma)$ to sort an input of N equally distributed characters. It can easily be seen that for an instance with N/p ratio in $O(p\hat{\ell} \log p)$, the time required for a character exchange is already bounded by the time to sort samples.

$$T_{\text{exch}} = O\left(\alpha p + \beta \frac{N}{p} \log \sigma\right) = O\left(\alpha \log^2 p + \beta p \hat{\ell} \log p \log \sigma\right) = T_{\text{sort}}$$

Intuitively, we may infer that the single-level algorithm is only efficient for inputs of size $N = \Omega(p^2 \hat{\ell} \log p)$, and thus with N/p ratio in $\Omega(p\hat{\ell} \log p)$. The preceding analysis is clearly somewhat limited as it may overestimate the cost of sample sorting, outright ignores the exchange of LCP values, and disregards potential input imbalances among other factors. However, because behavior consistent with this analysis is also revealed by an experimental evaluation, the observation still serves as motivation for the multi-level algorithms introduced hereafter. Also note that similar limitations are often inherent to related distributed sampling- and partitioning-based sorting algorithms, and are not exclusive to our particular versions of multi-way string merge sort.

In their paper on "massively parallel sorting" [6], Axtmann et al. cover the same problem for the case of atomic distributed merge and sample sort algorithms. The authors propose multi-level generalizations of both algorithms to address the problem. In short, the idea is to partition the input into fewer than p buckets, distribute the elements of each bucket over groups of multiple PEs, and recurse independently on each group to sort assigned strings. By selecting appropriate splitting factors, it is possible to reduce the number of samples that need to be sorted by a factor p/r for a favorable choice of r. The main disadvantage of this approach is that with k levels of recursion, input elements need to be moved between PEs k

times which incurs increased overhead, in the form of additional communication volume. By bringing this argument to its logical conclusion, we would arrive at $\log p$ levels of recursion where the number of PEs is halved each time. This would require moving the data $\log p$ times which essentially completes the circle back to hypercube quicksort. In practice, we will therefore choose some midpoint between the two extremes as a trade-off between the cost of partitioning and the cost of communication. Axtmann et al. support the efficacy of this approach with an evaluation of their multi-level merge and sample sort implementations on up to 2^{15} PEs. In this chapter, we apply the same techniques to distributed string sorting by developing multi-level generalizations of MS and PDMS.

Throughout this chapter, we rely on the established input conventions and related definitions to describe and analyze our algorithms. As before, each PE i receives an array of strings $\mathcal{S}_i$ as input. We explicitly allow duplicate strings and therefore have to define the global input array $\mathcal{S} := \mathcal{S}_0 + \cdots + \mathcal{S}_{p-1}$ as the concatenation of local arrays. The existing definitions of n, N, D, $\hat{\ell}$, and $\hat{d}$ are assumed for $\mathcal{S}$.

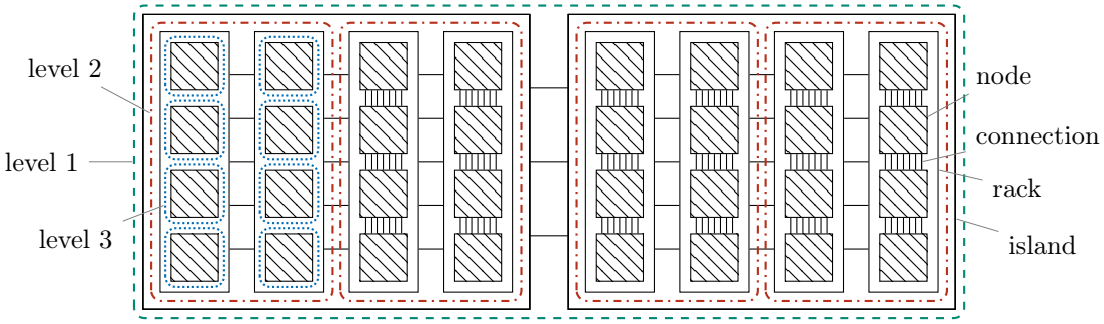

Figure 4.1.: A possible configuration of PE groups for $k = 3$ levels in a system with two islands, four racks per island, and four nodes per rack. The first level uses the entire system (- - -), the second level combines two racks (-·-·-), the final level is node internal (·····). The configuration assumes around eight PEs per node. Bandwidth is indicated by the number of connections between components.

The algorithms presented in this chapter work by recursively splitting PEs into multiple groups. We use k levels of recursion with arbitrary splitting factors between levels. To avoid special cases, we require that the final level splits the PEs into groups of size one. Thus, the original single-level variant of an algorithm may be trivially obtained by choosing $k = 1$. In the description of algorithms, we always assume the case of p PEs which can be perfectly subdivided into r groups of p' consecutive PEs. It follows naturally that the (zero-based) jth group consists of PEs jp' to $(j + 1)p' - 1$. While the splitting factors for a given number of recursions may notionally be chosen freely, a number of considerations can be used to help limit the space of available divisors of p. To simplify analysis of the multi-level algorithms, we generally assume approximately equal splitting factors on each level, i.e., $r = \Theta\left(\sqrt[k]{p}\right)$ which means that $p = \Theta\left(r^k\right)$. The group size on the lth level of recursion p/r^l can interchangeably be obtained as r^{k-l} or $p^{(k-l)/k}$. In practice, one also needs to consider architectural properties of any particular distributed system. For example, it may seem immediately beneficial to choose a level of recursion—most likely the final one—such that the number of PEs in each group matches the number of processor cores per *compute node* in the system. Thus, provided that processes are allocated correctly, no further inter-node communication is required for the remaining sorting process. Similarly, in

a system with multiple *islands*, where the network topology connecting nodes within a single island provides a higher bandwidth than the one between islands, parameters may be chosen such that every PE of a group is located in the same island. Figure 4.1 provides an example for a configuration of PE groups and splitting factors in a system with a hierarchy of islands, racks, and nodes.

Having established the preceding conventions and definitions, we can turn to the description of multi-level algorithms. Section 4.1 covers the generalization of the merge sort algorithm without prefix doubling to obtain multi-level MS. Building on the basic version, Section 4.2 introduces the necessary modifications to obtain an algorithm that only considers approximate distinguishing prefixes, i.e., multi-level PDMS. We also introduce a multi-level version of the Bloom filter used to approximate such prefixes.

4.1. Multi-Level Merge Sort

Multi-level distributed string Merge Sort, or multi-level MS for short, is closely based on the single-level algorithm by Schimek [36]. Processing steps are mostly identical, with the notable exception of a distributed group assignment phase before each string exchange. The same string-specific optimizations to local sorting, multi-way merging, compression, etc. may be applied. We provide a formal description of the algorithm in Section 4.1.1 and analyze its complexity in Section 4.1.2.

4.1.1. Algorithmic Details

The algorithm is split into a one-time initialization and a recursive phase which is invoked k times. The primary processing steps of both phases are listed and described below. Figure 4.2 provides a visual illustration of the multi-level merge sort scheme without string-specific details.

Initialization The algorithm first ensures that all input arrays are sorted locally.

(1) **Local Sorting:** On each PE sort the local input array $\mathcal{S}_i$. The LCP array is either obtained as by-product of sorting or needs to be computed explicitly.

Recursion The algorithm now proceeds recursively to establish a global order. As necessary precondition, we require that string arrays $\mathcal{S}_i$ are sorted locally. There are always p PEs and r groups of size $p' = p/r$ on each level of recursion; in other words, the values of p and p' are updated after every round of sorting.

(2) **Distributed Partitioning:** Globally determine $r - 1$ splitters and compute local buckets $\mathcal{B}_i^0, \ldots, \mathcal{B}_i^{r-1}$ on each PE using the distributed ordered partitioning algorithm from Section 3.2.

(3) **Distributed Group Assignment:** Using one of the algorithms from Section 3.3, assign the strings from bucket $\mathcal{B}_i^j$ to the PEs of group j on each PE.

(4) **String Exchange:** Using the assignment from the previous step, exchange strings and LCP values using direct messaging.

(5) **Local Merging:** On each PE i, merge the received string sequences to obtain locally sorted string arrays $\mathcal{O}_i$. We also obtain up-to-date LCP values for $\mathcal{O}_i$ during merging.

After each level of recursion, the following invariant must hold: For any two groups $i, j \in [0, r)$ any string on a PE of group i is smaller than all strings on the PEs of group j. On the final level we require that r is equal to the remaining number of PEs and thus groups have size $p' = 1$ which makes the invariant equivalent to the regular criteria for global sortedness. If the algorithm has not reached the final level of recursion, i.e., $p' > 1$, then it independently recurses on each group of PEs with the updated value of $p \leftarrow p'$ and $\mathcal{S}_i \leftarrow \mathcal{O}_i$. In other words, r instantiations of each step are executed in parallel.

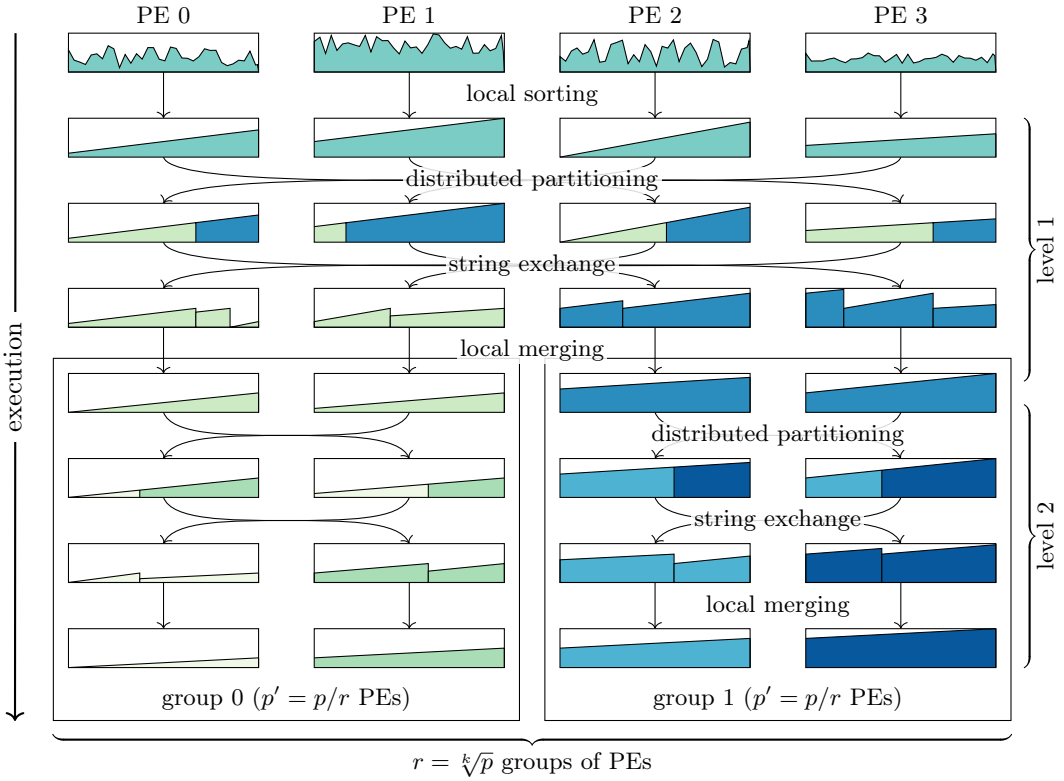

Figure 4.2.: Execution of multi-level merge sort for an instance with $p = 4$ PEs and $r = 2$ groups. Colors indicate lexical distribution of elements. Group assignment phases are omitted for brevity. String exchanges assume simple group assignment.

Several further string-specific optimizations are possible for each of the preceding steps, a selection of which is mentioned hereafter. Local sorting may use a sequential string sorting algorithm such as MSD radix sort. Similarly, an LCP-aware multi-way string merging algorithm, e.g., LCP-aware loser trees (a.k.a. tournament trees) [10], can be used for local merging. During distributed partitioning, samples may be truncated to an approximation of the average distinguishing prefix length to reduce communication volume at the cost of forfeiting worse-case guarantees. Communication volume during string exchange phases can be reduced by applying LCP compression. Schimek presents an algorithm that makes it possible to merge strings without unpacking LCP-compressed sequences. If the distributed partitioning phase was changed to work with compressed strings as well, then only a single round of compression during the first level of recursion would be required without a need

for decompression at any point during the algorithm. We did not further investigate this possibility and instead assume sequences to be decompressed after each string exchange phase.

4.1.2. Runtime and Communication

We now turn to analyzing the complexity of multi-level MS. As before, the analysis assumes that inputs are initially balanced with respect to the number of strings and characters except for constant factors, i.e., $|\mathcal{S}_i| = \Theta(n/p)$ and $\|\mathcal{S}_i\| = \Theta(N/p)$ for each PE i. To ensure that this balance is—at least approximately—maintained after every redistribution of strings, we first show that repeated partitioning does not lead to escalating imbalance.

Bounds on Workload Imbalance

The various theorems on imbalance introduced for partitioning from Section 3.2 are the basis for the following analysis. We aim to generalize these to make them applicable in the multi-level context.

Lemma 5. *On recursion level l with $r = \sqrt[k]{p}$ groups, in step (2) of multi-level MS using string-based regular sampling with oversampling factor v and string-based deterministic assignment, each bucket has size at most*

$$\left|\mathcal{B}^j\right| \leq \left(1 + \frac{r}{v}\right)^l \frac{n}{r^l}.$$

Proof. We give a proof by induction. For the first level of recursion, Theorem 2 immediately yields a maximum bucket size of

$$\left|\mathcal{B}^j\right| \leq \frac{n}{r} + \frac{n}{v} = \left(1 + \frac{r}{v}\right)\frac{n}{r}.$$

Using string-based deterministic assignment from Section 3.3.3 guarantees that each PE in a group receives the same number of strings. We can therefore apply Theorem 2 again, for recursion level $l - 1 \rightsquigarrow l$ to obtain the stated bound

$$\left|\mathcal{B}^j\right| \leq \left(1 + \frac{r}{v}\right)\frac{1}{r}\left(1 + \frac{r}{v}\right)^{l-1} \frac{n}{r^{l-1}} = \left(1 + \frac{r}{v}\right)^l \frac{n}{r^l}.$$

□

The preceding theorem is of little use in establishing nontrivial bounds on communication volume and local work for multi-level MS, since it cannot limit the number of characters per bucket. It nevertheless allows us to make general statements about the algorithm and informs our choice of oversampling factor for regular sampling. The term $(1 + r/v)^l$ means that imbalance between buckets multiplies with each level of recursion. We therefore need to choose the oversampling factor v such that the term remains asymptotically constant for the entire sorting process. For single-level MS $v = \Theta(p)$ samples per PE suffice which we generalize to $v = \Theta(kr)$ for any number of levels k. Notice the crucial difference between drawing kr rather than $r^k = p$ samples. With this oversampling factor, we get $(1 + 1/k)^k = \Theta(1)$ after

the final round of partitioning[1]. More precisely, to achieve an imbalance of $(1+\varepsilon)(n/p)$ for a constant ε, we require per-level imbalance ε' and an appropriate v that satisfy

$$\varepsilon' = \frac{r}{v} = \sqrt[k]{1+\varepsilon} - 1 = \Theta\Big(\frac{\varepsilon}{k}\Big).$$

Next, we provide a version of Lemma 5 that uses character-based sampling and group assignment.

Lemma 6. *On recursion level l with $r = \sqrt[k]{p}$ groups, in step (2) of multi-level MS using character-based regular sampling with oversampling factor v and a character-based assignment algorithm, each bucket has size at most*

$$\big\|\mathcal{B}^j\big\| \leq \Big(1+\frac{r}{v}\Big)^l \left(\frac{N}{r^l} + l\Big(1+\frac{v}{r}\Big)\frac{p}{r^{l-1}}\hat{\ell}\right).$$

Proof. We give a proof by induction as in the previous lemma. The base case of $l = 1$ can be derived immediately from Theorem 3 as follows:

$$\big\|\mathcal{B}^j\big\| \leq \Big(1+\frac{r}{v}\Big)\frac{N}{r} + \Big(1+\frac{v}{r}\Big)p\hat{\ell} \leq \Big(1+\frac{r}{v}\Big)\left(\frac{N}{r} + \Big(1+\frac{v}{r}\Big)p\hat{\ell}\right)$$

We now proceed with the inductive case $l-1 \rightsquigarrow l$ by applying the same theorem again on each group. Note that the value of p' needs to be changed to reflect the current group size, i.e., $p \rightsquigarrow p/r^{l-1}$. Recall that character-based group assignment in contrast to the string-based version does not yield perfect intra-group balance. Because string lengths may not exactly line up with PE boundaries, there is a possible imbalance of up to $O(\hat{\ell})$ characters between PEs in the same group with deterministic assignment (cf. Section 3.3.3). We can furthermore assume that $r/v \geq 0$ as both values must in fact be positive. This implies $(1+r/v)^{l-1} \geq 1$ which we use to derive the second inequality $(*)$ in the following equation and thereby suffices to show the stated bound:

$$\begin{aligned}
\big\|\mathcal{B}^j\big\| &\leq \Big(1+\frac{r}{v}\Big)\frac{\big(1+\frac{r}{v}\big)^{l-1}\frac{N}{r^{l-1}} + (l-1)\big(1+\frac{r}{v}\big)^{l-2}\big(1+\frac{v}{r}\big)\frac{p}{r^{l-2}}\hat{\ell}}{r} + \Big(1+\frac{v}{r}\Big)\frac{p}{r^{l-1}}\hat{\ell} \\
&= \Big(1+\frac{r}{v}\Big)^l\frac{N}{r^l} + (l-1)\Big(1+\frac{r}{v}\Big)^{l-1}\Big(1+\frac{v}{r}\Big)\frac{p}{r^{l-1}}\hat{\ell} + \Big(1+\frac{v}{r}\Big)\frac{p}{r^{l-1}}\hat{\ell} \\
&\overset{*}{\leq} \Big(1+\frac{r}{v}\Big)^l\frac{N}{r^l} + l\Big(1+\frac{r}{v}\Big)^{l-1}\Big(1+\frac{v}{r}\Big)\frac{p}{r^{l-1}}\hat{\ell} \\
&\leq \Big(1+\frac{r}{v}\Big)^l\left(\frac{N}{r^l} + l\Big(1+\frac{v}{r}\Big)\frac{p}{r^{l-1}}\hat{\ell}\right)
\end{aligned}$$

□

Lemma 6 may seem intractable at first sight and its usefulness may not be immediately apparent. The following seeks to interpret the result and derive asymptotic bounds for the maximum bucket size. First, note that the left summand of the inequality is equivalent to the lemma's string-based version which further reinforces our choice of $v = \Theta(kr)$. Using the stated oversampling factor and by applying the same argument as before, we can simplify the left summand to $O(\|\mathcal{S}\|/p)$. Because character-based regular sampling introduces an additional imbalance from shifting samples to the beginning of the next string, we get the

[1] Note that $\lim_{k\to\infty}(1+\alpha/k)^k = e^\alpha = O(1)$ for any constant $0 < \alpha \in \mathbb{R}$.

added right summand. By substituting $r = \sqrt[k]{p}$ and $(1 + 1/k)^{k-1} \leq (1 + 1/k)^k = O(1)$, we can simplify the term to

$$O\Big(k(1+k)\frac{p}{r^{k-1}}\hat{\ell}\Big) = O\Big(k^2 r\hat{\ell}\Big).$$

For the single level case with $k = 1$, this is immediately equivalent to $O(p\hat{\ell}\,)$ which is the same as the original algorithm. For the case of $k > 1$, we need to further interpret the result. Character-based regular sampling requires that the sampling distance ω' is greater than the length of the longest string. With asymptotically balanced local string arrays and oversampling factor $v = \Theta(kr)$, the sampling distance is in $O(N/pkr)$ (cf. Section 3.2.3). It is therefore possible to bound the term $k^2 r\hat{\ell}$ as follows:

$$k^2 r\hat{\ell} \leq k^2 r\omega' = O\left(\frac{k^2 rN}{pkr}\right) = O\left(k\frac{N}{p}\right)$$

If we additionally require that k is constant, then the term and therefore also bucket sizes overall are in $O(N/p)$. This suffices to demonstrate that, with character-based regular sampling, multiple level of partitioning and exchanging strings does not lead to unexpected imbalances.

In total, using character-based sampling with an oversampling factor in $\Theta(kr)$ yields bucket sizes in $O(\|\mathcal{S}\|/p + k^2 r\hat{\ell})$. This may seem slightly counter-intuitive, as a single round of character-based partitioning already introduces an imbalance of $O(p\hat{\ell})$ characters. However, consider that the assignment algorithm distributes this imbalance over p' PEs in each group. Because of the unwieldy form of Lemma 6—and in contrast to the string-based case—we are unable to give exact per-level imbalance parameters to achieve a particular final imbalance ε. It is also possible to derive bounds for the number of characters per PE after each level of recursion using similar arguments. Simply divide buckets sizes by the current value of $p' = r^{k-l}$ and add the imbalance introduced by deterministic character-based assignment as in Theorem 10 to bound the size of $\|\mathcal{O}_i\|$ and therefore $\|\mathcal{S}_i\|$ with

$$O\left(\frac{\|\mathcal{B}^j\|}{r^{k-l}} + \hat{\ell}\right) = O\left(\left(1+\frac{1}{k}\right)^l\left(\frac{N}{p} + l(1+k)r\hat{\ell}\right) + \hat{\ell}\right) = O\left(\frac{N}{p} + k^2 r\hat{\ell}\right). \tag{4.1}$$

With the additional assumptions $k = O(1)$ and $\hat{\ell} = O(N/pr)$, the term is in $O(N/p)$ which we later use to provide simplified runtime guarantees for well-behaved inputs.

Finally, it is necessary to bound the number of strings per PE in some way. While it is not possible to give general balance guarantees, we can at least try to improve on the worst-case assumption of $O(N/p)$ strings, i.e., one string per character. First consider the universal inequalities $N \leq n\hat{\ell}$ and $n \leq N/\check{\ell}$ if we assume that $\check{\ell} > 0$. By applying these relations to Equation (4.1), we can limit the number of strings $|\mathcal{S}_i|$ with

$$O\left(\frac{N/p + k^2 r\hat{\ell}}{\check{\ell}}\right) = O\left(\frac{n\hat{\ell}/p + k^2 r\hat{\ell}}{\check{\ell}}\right) = O\left(\frac{\hat{\ell}}{\check{\ell}}\left(\frac{n}{p} + k^2 r\right)\right)$$

using the ratio of longest to shortest string. This yields strong guarantees at least for benign instances where $\hat{\ell}$ is close to $\check{\ell}$. In the general case (i.e., if $\hat{\ell} \gg \check{\ell}$), we still require additional assumptions to guarantee that the number of strings per PE remains in $O(n/p)$. For the sake of simpler notation, we define shorthands

$$\widetilde{N} \coloneqq \frac{N}{p} + k^2 r\hat{\ell} \qquad \text{and} \qquad \widetilde{n} \coloneqq \frac{\hat{\ell}}{\check{\ell}}\left(\frac{n}{p} + k^2 r\right). \tag{4.2}$$

We can additionally require that $\widetilde{n} = O(\widetilde{N})$, because $n\hat{\ell}$ may otherwise overestimate the total number characters. For a proper worst-case analysis, we cannot reasonably improve on $O(N/p + k^2 r\hat{\ell})$ strings per PE.

Bounds on Complexity

With the preceding bounds on character imbalance, we are able to obtain meaningful guarantees for the runtime of multi-level MS. We first analyze the runtime for a single level of recursion by providing bounds for steps (2)–(5). Note that on recursion level l the number of PEs is p/r^{l-1}.

Distributed Partitioning First, each PE samples $v = \Theta(kr)$ strings locally with local work in $O(\widetilde{n})$ using character-based sampling. The samples are then sorted using a hypercube quicksort algorithm, e.g., LCP-RQuick. As stated in Theorem 1, this incurs polylogarithmic latency $O(\alpha \log^2 \frac{p}{r^{l-1}})$. For local work, we start with the bound from the theorem and substitute the current values for p and v for the number of elements per PE. We simplify the term using our choice of v to estimate runtime as

$$O\Big(\Big(\hat{\ell} + \log\sigma + \log\frac{p}{r^{l-1}}\Big)v + \hat{\ell}\log\frac{p}{r^{l-1}}\Big) = O\Big(kr\hat{\ell}\log\frac{p}{r^{l-1}}\log\sigma\Big).$$

Performing the same substitutions and similar simplifications, we can obtain an equivalent bound for communication volume. The factor $O(\hat{\ell}\log^2\frac{p}{r^{l-1}}\log\sigma)$ in communication for median determination is dominated by $O(r\hat{\ell}\log\frac{p}{r^{l-1}}\log\sigma)$ because $r = \sqrt[k]{p}$. Note that the bound is effectively naive hypercube quicksort and equal to sending v strings of $\hat{\ell}$ characters $\log p$ times. Once the samples have been sorted we select and distribute the final splitters using an all-gather operation. By conceptualizing this as separate gather and broadcast operations, we obtain communication time $O(\alpha\log\frac{p}{r^{l-1}} + \beta r\hat{\ell}\log\sigma)$. Finally, splitters are located in $\mathcal{S}_i$ to partition the array, using a single linear search in time $O(\widetilde{N})$ as described in Section 3.2.1. To summarize, overall complexity of the partitioning phase is almost entirely determined by the time and communication required to globally sort samples. In total, we get the following bound on runtime:

$$O\Big(\widetilde{n} + \alpha\log^2\frac{p}{r^{l-1}} + \beta kr\hat{\ell}\log\frac{p}{r^{l-1}}\log\sigma\Big) \tag{4.3}$$

Distributed Assignment We do not provide tight bounds for the runtime of deterministic assignment. Suffice to say that the algorithm's complexity is dominated by the actual string exchange. From Conjecture 9 [6, Theorem 1] we infer runtime in

$$2 \cdot \widetilde{\text{Exch}}(p, O(r\log N), O(r)). \tag{4.4}$$

We assert that this term is dominated by the string exchange phase for inputs with some reasonable restrictions. A multi-level algorithm only makes sense if the input size is polynomial in p, since otherwise a single-level algorithm would be better. We can therefore assume that $\log N = O(\log p)$ which, with $r = O(\sqrt{p})$, implies that $r\log N = O(p)$. This is dominated by the characters exchange phase, unless $N/p = o(p)$ in which case LCP-RQuick may be preferable.

String Exchange Little further analysis is needed for the string exchange phase. There are p/r^{l-1} PEs which all receive $O(\widetilde{N})$ characters from $O(r)$ other PEs according to the bounds established here and in Theorem 10. Exchanging LCP values additionally requires $O(\log n)$ bits per string. We do not consider gains achieved through LCP compression as they cannot improve worst-case complexity. These values can immediately be substituted into the Exch$(\cdot)$ black-box to obtain the following:

$$\mathrm{Exch}\Big(\frac{p}{r^{l-1}}, O\Big(\widetilde{N}\log\sigma + \widetilde{n}\log n\Big), O(r)\Big) \tag{4.5}$$

Local Merging Merging the $O(r)$ received string arrays without an LCP-aware algorithm requires local work in

$$O\Big(\widetilde{N} + \widetilde{n}\log r\Big). \tag{4.6}$$

Recursion With the bounds from Equations (4.3)–(4.6) and some simplifications, we can derive a bound for the runtime of a single recursion. Note the usage of $\widetilde{\mathrm{Exch}}(\cdot)$ to account for group assignment.

$$\begin{aligned} &O\Big(\widetilde{N} + \widetilde{n}\log r + \alpha\log^2\frac{p}{r^{l-1}} + \beta k r\hat{\ell}\log\frac{p}{r^{l-1}}\log\sigma\Big) \\ &\quad + \widetilde{\mathrm{Exch}}\Big(\frac{p}{r^{l-1}}, O\Big(\widetilde{N}\log\sigma + \widetilde{n}\log n\Big), O(r)\Big) \end{aligned} \tag{4.7}$$

Overall Runtime From this we can easily derive the algorithm's combined runtime. The initial local sorting with MSD radix sort takes time in $O(\max_i \mathcal{D}(\mathcal{S}_i) + n/p\log\sigma)$, of which the first term can be absorbed into $\widetilde{N}$. What remains are k repetitions of Equation (4.7). The first part of the equation can be simplified by substituting the original value of p. For the exchange phases we observe that the algorithms iterates over groups of $p^{k/k}, \ldots, p^{1/k}$ PEs and reformulate the $\widetilde{\mathrm{Exch}}$-terms accordingly. In total, this yields the following bound for the runtime of multi-level MS.

Theorem 11. *Multi-level MS with $r = \sqrt[k]{p}$, and using character-based sampling and deterministic assignment, can be implemented to run in time*

$$\begin{aligned} &O\Big(\frac{n}{p}\log\sigma + k\Big(\widetilde{N} + \widetilde{n}\log r + \alpha\log^2 p + \beta k r\hat{\ell}\log p\log\sigma\Big)\Big) \\ &\quad + \sum_{l=1}^{k}\widetilde{\mathrm{Exch}}\Big(p^{\frac{l}{k}}, O\Big(\widetilde{N}\log\sigma + \widetilde{n}\log n\Big), O(r)\Big). \end{aligned}$$

Interpretation of Results

The result from Theorem 11 makes high-level analysis difficult due to its inherent complexity. Further interpretation of the bound requires additional assumptions to simplify the term. First, we constrain the input such that the number of characters per PE is in $O(N/p)$. Considering the definition of $\widetilde{N}$ from Equation (4.2), this is equivalent to $k^2 r\hat{\ell} = O(N/p)$. An analogous bound for the number of strings $\widetilde{n}$ requires a much stronger assumption. We need to require a priori that multi-level MS manages to maintain $O(n/p)$ strings per PE. In practice, we can only guarantee this if, for example, all strings have equal length, or,

at least in expectation, if there is no statistical correlation between lexical order and the length of strings. However, without this constraint few meaningful worst-case guarantees are possible.

Next, we can reasonably assume that a string exchange phase in a smaller network is at least as fast as one in a large network, if PE groups for the algorithm are properly configured. This, together with the results from the previous section, allows us to bound the sum from Theorem 11 with

$$k\,\widetilde{\mathrm{Exch}}\left(p, O\left(\frac{N}{p}\log\sigma + \frac{n}{p}\log n\right), O(\sqrt[k]{p})\right).$$

If we further assume that exchange phases are implemented with a direct exchange, then the resulting latency of at least $\Omega(\alpha\sqrt[k]{p})$ dominates the term $O(\alpha k \log^2 p)$. To further simplify the runtime, we assert that α and β are constants and use this to consolidate local work and communication. We can thereby immediately absorb terms $O(kN/p)$ and $O(n/p \log\sigma)$. For the term $O(n/p \log r)$ we need to assume that the number of strings is a polynomial over p and therefore $\log n = \Theta(\log p)$. With $r = O(p)$, the term is dominated by the exchange of LCP values. This is reasonable for instances with $\hat{\ell} \approx \hat{\ell}$ because multi-level MS only makes sense for $N/\hat{\ell}$ polynomial over p as mentioned in the introduction to this chapter. In total, this leaves communication for hypercube quicksort and terms for data exchange phases. If we assume the best-case scenario for $\mathrm{Exch}(\cdot)$, then the runtime of multi-level MS can finally be estimated as

$$O\left(k^2\hat{\ell}\sqrt[k]{p}\log p\log\sigma + k\frac{N}{p}\log\sigma + k\frac{n}{p}\log n\right).$$

Disregarding the exchange of LCP values, this shows that the algorithm is only efficient for inputs with $N = \Omega(kp^{1+1/k}\hat{\ell}\log p)$—a factor $p^{(k-1)/k}/k$ better than the single-level algorithm.

4.2. Multi-Level Prefix Doubling Merge Sort

Multi-level Prefix Doubling string Merge Sort (multi-level PDMS) can be obtained from multi-level MS by applying a number of modifications to the algorithm, similar to the single-level variants. First, a step is added to the algorithm's initialization phase, which computes an approximation of the distinguishing prefixes:

(1.1) **Distinguishing Prefix Approximation:** Compute an approximation of the global distinguishing prefix of each string and replace each string in $\mathcal{S}_i$ with the resulting prefix for the rest of the algorithm.

This step uses a prefix doubling technique in combination with a Bloom filter to compute an approximation in at most $\lceil \log \hat{\ell} \rceil$ iterations. We adapt the single-level version slightly in Section 4.2.1 to avoid latency $O(\alpha p)$ per iteration. Additionally, the algorithm's output must be changed to use a different format. This requires auxiliary information to be associated with each string to reconstruct its original location after sorting. We discuss this point with more detail in Section 4.2.2.

4.2.1. Multi-Level Bloom Filter

Schimek proposes an approximation scheme that uses a prefix doubling technique to check exponentially growing string prefixes for uniqueness [36]. The approach includes a distributed duplicate detection algorithm based on a *distributed single shot Bloom filter* [33] as subroutine. The multi-level version of PDMS can reuse the prefix doubling technique without change, but requires a slightly refined version of the duplicate detection algorithm. The following will first give a short explanation of the single-level version and then highlight the changes needed to make it suitable for usage in the multi-level algorithm.

Conceptually, a Bloom filter works by defining a global bit array B with all bits initial set to 0. Let $h\colon \mathcal{S} \to \{0, \dots, |B| - 1\}$ be a hash function that maps strings of $\mathcal{S}$ to positions of B. A string s can be added to the Bloom filter by setting the bit at position $h(s)$ to 1. To query whether s has been previously added to the Bloom filter, simply check the bit at the same position $h(s)$. If the bit has value 0, then we can guarantee that s has not been added yet. If the value is 1, then s or another value with the same hash value has been added, i.e., there may be false positives due to hash collisions. To identify potential duplicates in $\mathcal{S}$, we first add all strings from $\mathcal{S}$ and then perform membership queries for all strings. This is guaranteed to mark every duplicate as such. Assuming a uniform hash function and using a bit array of size $|B| = |\mathcal{S}|c$ for some positive integer c yields false positive rate $1/c$. Going forward we define the size of B as $m \coloneqq |\mathcal{S}|c$ and require that m is divisible by p for simplicity.

The technique described above can be used to approximate distinguishing prefixes in a distributed context. Let $\ell \leq \hat{\ell}$ be the current length of prefixes to be tested. Hash values, i.e., positions of the bit array B, are distributed equally over PEs in chunks of size m/p. Hence, PE i is assigned hash values im/p to $(i+1)m/p - 1$. First, each PE computes the hash value for the ℓ-prefix of every string in its local string array—truncating strings that are too short—to obtain the array

$$\mathcal{H}_i \coloneqq \big[h\big(s[0, \min\{|s|, \ell\}]\big) \,\big|\, s \in \mathcal{S}_i\big].$$

The array is sorted while saving the original position of elements. Next, hash values are sent to the appropriate PE according to their value. In the single-level version, this is accomplished with a single all-to-all exchange after partitioning the array into p intervals. Received hash values are tagged with the rank of the sending PE and sorted again. Duplicate values can now be identified using a single scan over the local array. The result is a bit array with one entry per received hash value, where a 1-bit denotes a possible duplicate, while a 0-bit indicates a unique string. To return this information back to the sender, partition the bit array into p separate arrays, using the rank saved during the second round of sorting to restore the order in which the hash values were originally received. Now, the bit arrays are sent back with another all-to-all exchange. It is important to note that the order of elements from the same PE is maintained by all-to-all exchanges, the second round of sorting, and the partitioning of bits. Finally, the bits can be mapped back to the original strings using positions saved during the first round of sorting which yields the bit array $\mathcal{R}_i$.

The duplicate detection algorithm as described here has two relevant communication phases which both use all-to-all exchanges. With the 1-factor algorithms (cf. Section 2.2), this incurs latency $O(\alpha p)$ which is undesirable in a multi-level setting where latency in $O(\alpha \sqrt[k]{p})$ is the goal. This was actually observed during early experimental evaluation of multi-level

PDMS where the latency dominated the algorithm's runtime for sufficiently large values of p. We use a grid-based communication pattern as a well-understood technique to contend with this problem. What makes the implementation noteworthy is that, due to the special structure of the communicated information, the algorithm is able to entirely avoid sending any additional information, e.g., ranks to indicate the destination of a message.

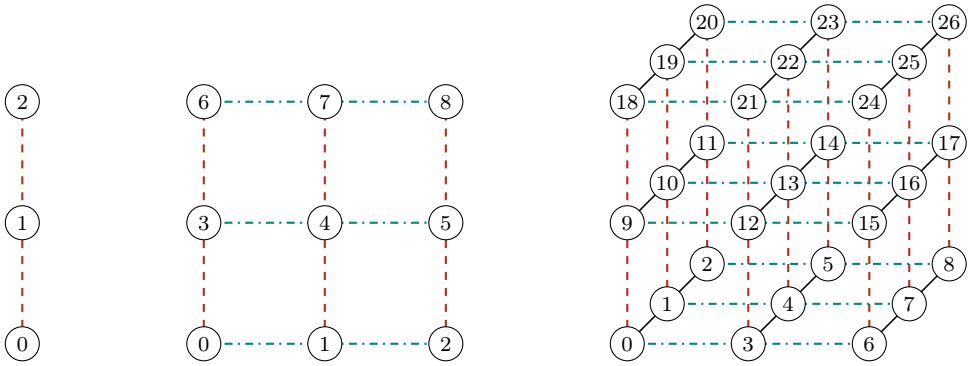

Figure 4.3.: From left to right, k-dimensional communicator grids for $k \in \{1, 2, 3\}$, $r = 3$, and $p = r^k$. PE groups for column-wise exchanges with $d = 1$ are highlighted as - - -. Groups for $d = 2$ are highlighted as -·- if applicable.

The following algorithm uses the same parameters as multi-level MS, though there exists no actual requirement for this. There are again $k \geq 1$ levels of recursion with equal splitting factor $r := \sqrt[k]{p}$ to define a k-dimensional grid. We define a function `ColumnAlltoall` to encapsulate all-to-all exchanges along conceptual column of the grid. The function is used as `ColumnAlltoall`$((m_i^0, \ldots, m_i^{r-1}), d)$, where m_i^j are the arrays to be sent and the second parameter d indicates the dimension along which to exchange. A dimension $d \in [1, k]$ defines a stride r^{k-d}, such that a PE i exchanges data with PEs $i + jr^{k-d} \mod p$ for appropriate values of $j \in \mathbb{Z}$. This is illustrated for simple values of r and k in Figure 4.3. The function returns a single concatenated array of received values, ordered by the ranks of sending PEs. Received values may optionally be tagged with an index from $[0, r)$ to indicate the sender. The idea is to partition the data into r buckets k times and exchange them among only r PEs with each iteration. During the first exchange, hash values can be partitioned and routed according to their numerical value. By saving the sender of received hash values, it is possible to route the bits during the second exchange. This approach is formalized as function `SendRec` in Algorithm 6. The algorithm's main processing steps are shown for a small instance in Figure 4.4.

We first consider the algorithm's recursive case, i.e., the `else`-block. Hash values are routed to the correct PE through incremental refinement. Prior to the first level of recursion, the array $\mathcal{H}_i$ may contain hash values from the entire range $[0, m)$. The array is partitioned into buckets of size m/r and redistributed by the first all-to-all exchange. Afterwards, the updated array $\mathcal{H}_i^{\text{recv}}$ only contains values from the range starting at $\lfloor i/r^{k-1} \rfloor m/r$. This can be extrapolated to an arbitrary recursion depth $d \in [1, k]$. For every PE i and every hash value h in $\mathcal{H}_i^{\text{recv}}$ on that PE, it holds that

$$\left\lfloor \frac{i}{r^{k-d}} \right\rfloor \frac{m}{r^d} \leq h < \left(\left\lfloor \frac{i}{r^{k-d}} \right\rfloor + 1 \right) \frac{m}{r^d}.$$

Algorithm 6: Multi-Level Bloom Filter Exchange

1 **Function** `SendRec`($\mathcal{H}_i$, $d \coloneqq 1$)
  Input: A sorted array of hash values $\mathcal{H}_i$ and recursion depth d on each PE i
2 **if** $d = k + 1$ **then** *// final level of recursion*
3 locally compute bit arrays of duplicates $\mathcal{R}_i$
4 **return** $\mathcal{R}_i$
5 **else** *// exchange hash values and recurse*
6 partition $\mathcal{H}_i$ into r buckets $\mathcal{H}_i^0, \ldots, \mathcal{H}_i^{r-1}$
7 $\mathcal{H}_i^{\text{recv}} \leftarrow$ `ColumnAlltoall`$\left(\left(\mathcal{H}_i^0, \ldots, \mathcal{H}_i^{r-1}\right), d\right)$ *// save source PE*
8 `Sort`$(\mathcal{H}_i^{\text{recv}})$ *// sort tuples by hash value*
9 $\mathcal{R}_i \leftarrow$ `SendRec`$([\, h \mid (h, \cdot) \in \mathcal{H}_i^{\text{recv}}], d + 1)$
10 **for** $(\cdot, j)$, b **in** $\mathcal{H}_i^{\text{recv}}, \mathcal{R}_i$ **do** *// Invariant:* $|\mathcal{H}_i^{\text{recv}}| = |\mathcal{R}_i|$
11 append b to bit array $\mathcal{R}_i^j$
12 **return** `ColumnAlltoall`$\left(\left(\mathcal{R}_i^0, \ldots, \mathcal{R}_i^{r-1}\right), d\right)$
  Output: A bit array $\mathcal{R}_i$ containing 1-bits for possible duplicates.

All values are therefore on the correct PE after $d = k$ levels of recursion, since $p = r^k$ which simplifies the expression to $im/p \leq h < (i + 1)m/p$. During the first all-to-all exchange, we tag hash values with an integer from the range $[0, r)$ to indicate the PE from which they were sent. This means that $\mathcal{H}_i^{\text{recv}}$ actually contains pairs (h, j) of hash and index. Only the first entry is considered during the sorting step in line 8, which can therefore be performed in expected linear time. Hash values being in sorted order also guarantees that the partitioning step only requires a single, linear time scan of the array to obtain contiguous intervals thereof.

Once hash values have been delivered to the correct PE, the local duplicate detection works identically to the single-level algorithm. To deliver bits back to the respective hash value's sender, we simply follow the indices from the first round of all-to-all exchanges. This uses the same arguments as the single-level version and relies on the fact that the order of hash values sent from the same PE is never altered after the first round of sorting. Local duplicate detection and each round of partitioning can be performed in linear time with a single scan of the respective array. It only remains to map the received bits back to the original strings, using the permutation saved during the first round of sorting. However, this exceeds the scope of Algorithm 6. Observe that the algorithm is a generalization of the single-level variant, which may be obtained by choosing $k = 1$.

Analyzing the complexity of Algorithm 6 and the prefix approximation scheme more broadly requires significant effort. This includes obtaining expected values for the size of distinguishing prefixes and using Golomb coding [31] to compress hash values for all-to-all exchanges. We refer to the analysis by Schimek [36], a more concise version by Bingmann et al. [11], and the original work by Sanders et al. [33] for more details. Because the multi-level Bloom filter only changes the data-delivery phases, we simply alter existing bounds and conjecture the resulting runtime.

Broadly, each iteration of duplicate detection incurs latency $O(\alpha r)$ and $O(\log p)$ bits for each string where a distinguishing prefix has not yet been found. Provided an appropriate false positive rate, i.e., a large enough value of m, the expected number of iterations is in

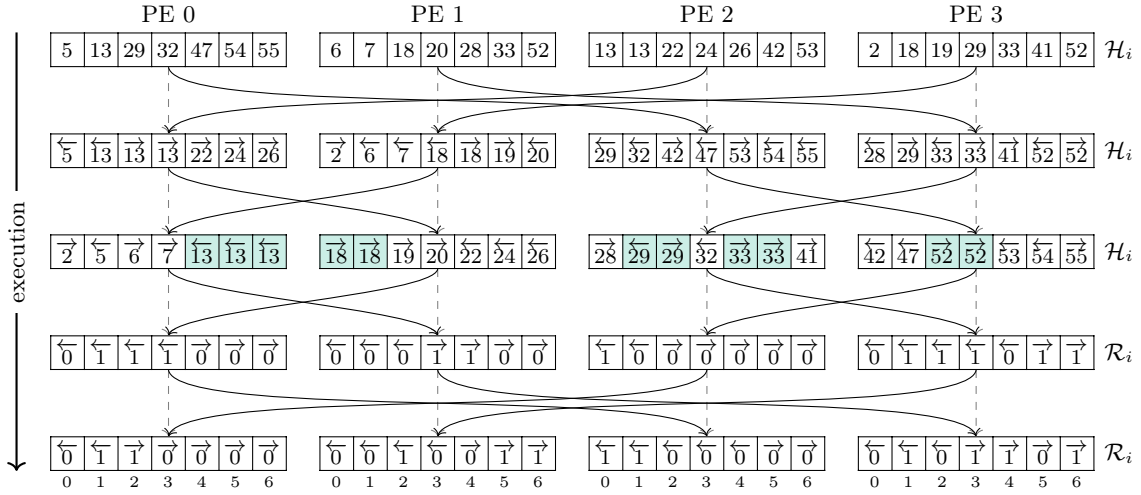

Figure 4.4.: Execution of the multi-level Bloom filter for an instance with $n = 28$, $m = 2n$, $p = 4$, and $r = 2$. Hash value arrays $\mathcal{H}_i$ are shown as passed to the function, bit arrays $\mathcal{R}_i$ are shown as returned. Arrows over hash values or bits (i.e., $\overleftarrow{x}$ and $\overrightarrow{x}$) indicate origin PE. Duplicates found during local detection are highlighted.

$O(\log \hat{d})$. Furthermore, let $\widehat{D} \coloneqq \max_i \mathcal{D}_{\mathcal{S}}(\mathcal{S}_i)$ be the largest *global distinguishing prefix*, i.e., distinguishing prefix of local string arrays $\mathcal{S}_i$ with respect to the global array $\mathcal{S}$. And let $\hat{n} \coloneqq \max_i |\mathcal{S}_i|$ be the maximum number of strings on any PE. This yields the following runtime for the updated prefix approximation algorithm with k levels of recursion and columns of size r.

Conjecture 1. *Distributed distinguishing prefix approximation using Algorithm 6 for data exchange can be implemented with expected runtime in*

$$O\Big(\widehat{D} + \Big\lceil \log \hat{d} \Big\rceil (\alpha k r + \beta k \hat{n} \log p)\Big).$$

As already stated, reaching the claimed communication volume requires compression of hash values using Golomb coding and only if a sufficient number of strings participate in the algorithm. It is not entirely clear whether repeated compression and decompression on every level of Algorithm 6 maintains the required runtime.

4.2.2. Distributed Permutations

Like the single-level variant, multi-level PDMS reduces communication volume by only exchanging string prefixes. The information discarded thereby makes it impossible to output the entire original strings after sorting without additional communication and thereby defeating the point of PDMS. Schimek therefore proposes to output the resulting permutation, in the form of references to the strings original position instead [36]. We refer to this format as a *distributed permutation* and provide the following definition:

Definition 1 (Distributed Permutation)**.** *A distributed permutation is an array $\mathcal{P}_i$ containing tuples from $[0, p) \times [0, n)$ on each PE i. The pair $(j, l) \in \mathcal{P}_i$ references the string $\mathcal{S}_j[l]$ at position l on PE j.*

The global permutation is defined using concatenation as $\mathcal{P} \coloneqq \mathcal{P}_0 + \cdots + \mathcal{P}_{p-1}$. For every PE j and any string $l \in [0, |\mathcal{S}_h|)$, a correct distributed permutation $\mathcal{P}$ must contain the pair (j, l) exactly once. Furthermore, the permutation must uphold the established criteria for sortedness. Building such a permutation requires the PE rank and local index, from which each string originates, to be exchanged during sorting. Though the PE rank may be omitted for the single-level algorithm as it can be inferred from the all-to-all exchange.

To remove the need for additional communication, and in preparation for the space-efficient sorting algorithm in Chapter 5, we introduce a second output format. The idea is to store the rank, in a globally sorted, order of each string at its original location—similar to the difference between a suffix array and an *inverse* suffix array. Hence, we designate this format as a *distributed inverse permutation.*

Definition 2 (Distributed Inverse Permutation)**.** *A distributed inverse permutation is an array $\overline{\mathcal{P}}_i$ containing integers from $[0, n)$ on each PE i. An entry $l = \overline{\mathcal{P}}_i[j]$ states that there exists an ordered permutation of $\mathcal{S}$ where the string $\mathcal{S}_i[j]$ is located at position l.*

We again define the concatenation $\overline{\mathcal{P}} \coloneqq \overline{\mathcal{P}}_0 + \cdots + \overline{\mathcal{P}}_{p-1}$ and require that the result is a permutation of $[0, n)$. To ensure sortedness, it must hold that for any indices $i, j \in [0, n)$, if the corresponding strings have order $\mathcal{S}[i] < \mathcal{S}[j]$, then the same must be true for their respective ranks $\overline{\mathcal{P}}[i] < \overline{\mathcal{P}}[j]$. Note that the order of ranks for duplicate strings in $\mathcal{S}$ may be chosen arbitrarily in this case. For the space-efficient sorting algorithm, in Section 5.3, we provide a version of inverse permutations which guarantees that equal strings receive identical ranks. It is clearly possible to *invert* a distributed permutation $\mathcal{P}$ to obtain $\overline{\mathcal{P}}$ by using a prefix-sum to compute ranks and sending the result back to the referenced PEs with an all-to-all exchange. Formally, the jth string on PE i in sorted order has global rank $j + \sum_{l=0}^{i-1} |\mathcal{O}_i|$, where $\mathcal{O}_i$ are the string arrays returned by PDMS. Depending on the all-to-all exchange algorithm, this approach has time complexity in

$$O(\alpha p + \beta \widetilde{n} \log n) \quad \text{or} \quad O(\alpha \log p + \beta \widetilde{n} \log p \log n).$$

Neither term can easily be absorbed into the existing runtime of multi-level MS. This approach may also be undesirable in practice, as it incurs additional communication during sorting to compute the permutation.

A less naive technique to obtain inverse permutations uses an approach very similar to the multi-level Bloom filter from Algorithm 6. As with the duplicate detection algorithm, each PE saves the resulting permutation during local sorting. Then, during each string exchange phase, received strings are tagged with the rank of the sending PE. These tags are reordered alongside the strings during merging and saved for later use. Once the final order has been established, global string ranks are determined using a prefix-sum as before. Now, the indices saved during the string exchanges are used to send ranks back to the string's original PE. Like in Algorithm 6, this exploits the fact that the local order of strings from the same PE does not change after initial sorting. Finally, the received ranks need to be reordered using the permutation obtained during local sorting. Because the exchange of string ranks uses the same sets of communication partners as the sorting process, we need to use the $\text{Exch}(\cdot)$ black-box to obtain runtime in

$$\sum_{l=1}^{k} \text{Exch}\Big(p^{\frac{l}{k}}, O(\widetilde{n} \log n), O(r)\Big).$$

This is asymptotically equivalent to existing LCP value exchanges. However, the improvement comes at the cost of space complexity in $O(k\widetilde{n}\log p)$ to store arrays of PE ranks on each level of recursion.

4.2.3. Runtime and Communication

Computing approximate distinguishing prefixes using a Bloom filter and prefix doubling reduces the expected number of characters on each PE to $O(\mathcal{D}_{\mathcal{S}}(\mathcal{S}_i))$. For the purposes of multi-level merge sort, we can therefore say that each PE initially contains $O(\widehat{D})$ characters with the previous definition of $\widehat{D}$ as the largest global distinguishing prefix. Furthermore, the length of the longest string is reduced to $O(\hat{d}\,)$ which applies to the runtime of partitioning phases and the character imbalance introduced thereby. Analogous to $\widetilde{N}$, we introduce the shorthand $\widetilde{D} \coloneqq \widehat{D} + k^2 r\hat{d}$ to include the additional characters. Combining runtimes for prefix approximation and merge sort using a regular distributed permutation yields the following overall runtime:

Theorem 12. *If prefix approximation has the runtime stated in Conjecture 1, then multi-level PDMS with $r = \sqrt[k]{p}$, and using character-based sampling and deterministic assignment, has expected runtime in*

$$O\Bigg(\frac{n}{p}\log\sigma + \overbrace{\Big(\alpha k r + \beta\frac{n}{p}\log p\Big)\log\hat{d}}^{\text{prefix approximation}} + k\Bigg(\overbrace{\widetilde{D} + \widetilde{n}\log r}^{\text{merging}} + \overbrace{\beta k r \hat{d}\log p\log\sigma}^{\text{partitioning}}\Bigg)\Bigg) + \underbrace{\sum_{l=1}^{k} \widetilde{\text{Exch}}\Big(p^{\frac{l}{k}}, O\Big(\widetilde{D}\log\sigma + \widetilde{n}\log n\Big), O(r)\Big)}_{\text{string exchange and group assignment}}.$$

Observe that the factor r in latency of multi-level Bloom filter dominates the factor $\log^2 p$ of hypercube quicksort. Furthermore, note the additional factor $\log\hat{d}$ in latency due to prefix doubling. The term $O(\widehat{D})$ local work for prefix approximation is dominated by the local work for a single level of sorting.

5. Space-Efficient String Sorting

A well-established characteristic of distributed-memory systems is founded in their limitations in terms of working memory provided per PE. While the system as a whole may provide massive amounts of memory in total, individual PEs only have access to a small portion thereof. Whereas shared-memory systems might nowadays provide in the order of multiple terabytes of memory to their processes, the same is usually not feasible in systems with many thousands of independent nodes. For example, the system used in our experimental evaluation only provides 96 GB per compute node, which amounts to 2 GB memory per PE (cf. Section 6.2.1). Applied to the setting of string sorting, where a single string may easily contain thousands of characters, this imposes substantial constraints on the size of inputs that can be processed with a given number of PEs.

Considering these memory limitations, it is only natural to explore ways of reducing the memory-footprint of string sorting algorithms. To this end, it can be observed that certain families of string sets requiring sorting, do not consist of wholly independent strings. Instead, there is often a significant overlap between strings in the same input, especially if strings are sections of a larger text. A symptomatic—even pathological—example for such inputs are suffix arrays, where an instance of n characters represents a set of strings with a combined length of approximately $n^2/2$ characters. However, a generic string sorting algorithm is unlikely to prove competitive with specialized suffix sorters. More relevant instances might consist of overlapping (possibly fixed-sized) substrings taken at larger intervals, e.g., taking strings of length ℓ starting at every kth character where $\ell > k$. A real-world example of this principle are *difference cover samples* used by the DCX suffix sorting algorithm—which we cover in greater detail in Section 5.3 and use in the experimental evaluation. Figure 5.1 illustrates compressed and uncompressed character arrays for a difference cover sample of a short text.

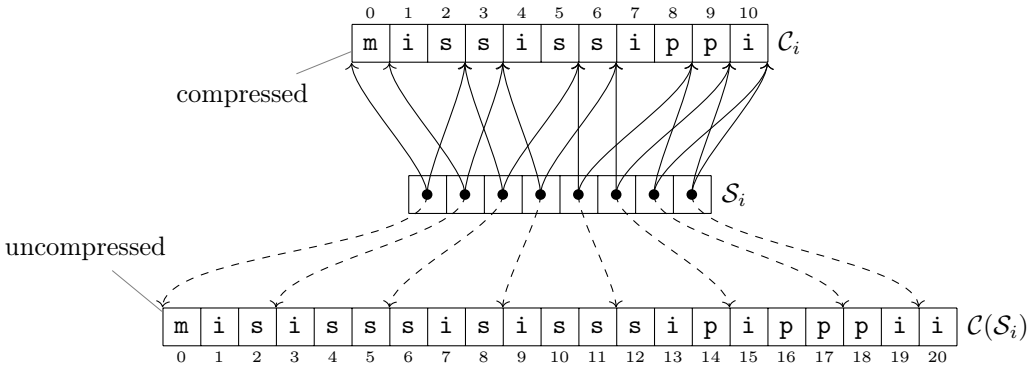

Figure 5.1.: Compressed and uncompressed character arrays for a given string array. The string array is a difference cover sample of $\mathcal{C}_i$ for DC3.

By using a compressed input format, where shared sections of character arrays are not duplicated, it is possible to construct instances that would ordinarily far exceed the available memory. For sequential algorithms and parallel algorithms in shared-memory systems such a format is easily implemented, because character arrays are usually not modified during sorting. In distributed systems however, strings must be redistributed onto different PEs according to their lexicographical order. This requires *materializing* strings to an uncompressed format and thereby duplicating shared sections. It follows that any compression by deduplication cannot be maintained outside the initial PE, which means that inputs may no longer fit into memory after being exchanged. In this chapter, we introduce an algorithmic framework, as well as concrete implementations therefore, to overcome this limitation. We broadly term the resulting algorithms as *space-efficient* distributed string sorting algorithms.

To formalize compressed inputs, let $\mathcal{C}_i$ denote an array of character on each PE i. Strings in the string arrays $\mathcal{S}_i$ are *backed* by subsequences of $\mathcal{C}_i$. This may alternately be realized using pairs of index and length, pairs of start and end index, or with pointers instead of indices. Though we leave the exact details open, suffice it to say that constant time access to characters and length are required for any representation. Crucially, strings can no longer be assumed to be terminated by a sentinel. Note the difference between the implicitly defined character array $\mathcal{C}(\mathcal{S}_i)$ which contains the concatenation of all strings in $\mathcal{S}_i$ and the concrete array $\mathcal{C}_i$. For reasonable instances, we may generally assume that all characters in $\mathcal{C}_i$ are part of at least one string or, at the very least, that $|\mathcal{C}_i| \leq |\mathcal{C}(\mathcal{S}_i)|$. Global string and character arrays are defined as $\mathcal{S} \coloneqq \mathcal{S}_0 + \cdots + \mathcal{S}_{p-1}$ and $\mathcal{C} \coloneqq \mathcal{C}_0 + \cdots + \mathcal{C}_{p-1}$. Definitions of n, N, ℓ, and $\hat{\ell}$ remain unchanged. We additionally define the global size of compressed character arrays as $M \coloneqq |\mathcal{C}|$ in contrast to the uncompressed size N.

Section 5.1 proposes a high-level framework for space-efficient distributed string sorting algorithms. It reduces large instances to multiple smaller ones and assumes a black-box distributed sorter to solve subproblems. The framework is realized in Section 5.2 using the previously established merge sort implementations, i.e., (multi-level) MS and PDMS. Several merge sort-specific optimizations are developed to make better use of the available preconditions. Finally, Section 5.3 develops a slight variation of the algorithm with the aforementioned application in distributed suffix sorting.

5.1. Algorithmic Framework

As already stated, our goal is to be able to sort inputs where the uncompressed character array $\mathcal{C}(\mathcal{S}_i)$ would not fit into memory. Let X be the size in characters of the largest instance that can be sorted with some distributed string sorting algorithm $\mathcal{A}$. In other words, any input with at most X characters in $\mathcal{C}(\mathcal{S}_i)$ on every PE, can be sorted directly using $\mathcal{A}$. We also assert that the inverse is true; i.e., if the size of any local character array exceeds X, then sorting the input with $\mathcal{A}$ fails. To arrive at a *space-efficient* algorithm, the idea is to split larger instances into multiple smaller subproblems of size at most X and sort each individually using the sorting algorithm $\mathcal{A}$. Thus, only the strings of a single instance need to be materialized at a time. If subproblems are chosen appropriately, then it is possible to obtain a globally sorted permutation for the original instance by combining the separate results. The following paragraphs elaborate on how the required subproblems are obtained and how the results must be combined to ensure a correct overall permutation. Figure 5.2

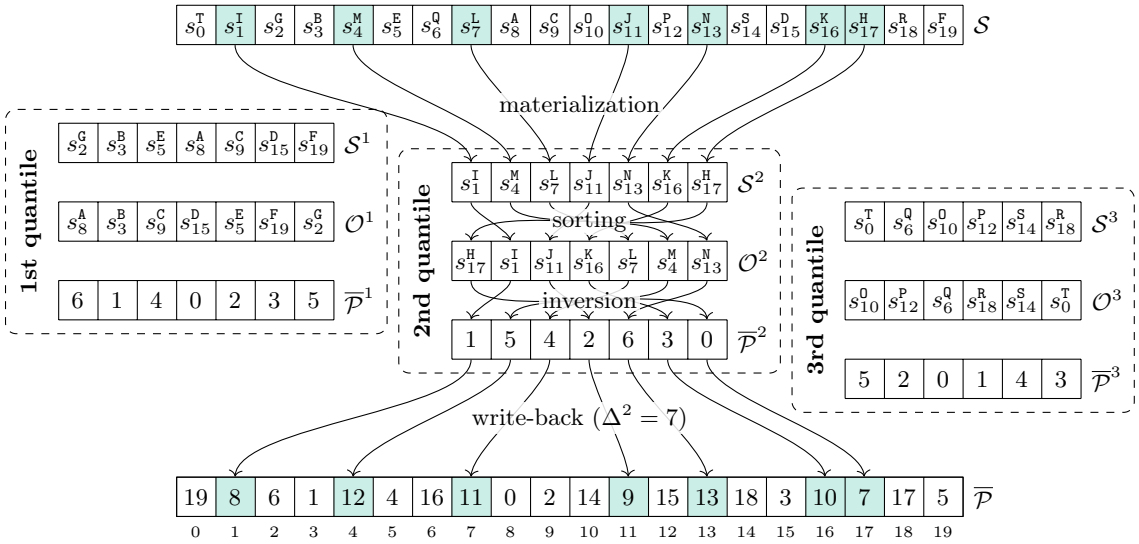

Figure 5.2.: Simplified illustration of the space-efficient sorting scheme without details of distributed execution. The string array $\mathcal{S}$ contains elements s_i^c where i indicates initial position and c indicates lexicographical order. Execution steps are detailed for the second round of sorting.

illustrates the principle of space-efficient string sorting for a simple example. In practice, the precise value of X may depend on a multitude of factors and is unlikely to be available for any realistic sorting algorithm. It can, in fact, not be assumed to be a constant at all, but rather to vary between inputs, for example, due to differing levels of prefix repetition and resulting gains through LCP compression. Even more removed from a theoretical viewpoint, are considerations such as multiple PEs in the same node sharing memory and, as a result, being subject to a cumulative limit on input size. With the aforementioned caveats in mind, the value of X can be seen as the primary tuning parameter for the space-efficient algorithms in this chapter.

To arrive at the desired subproblems, we compute a globally ordered partition of the local string arrays, similar to the partition used in distributed merge sort. Formally, for some as, yet undefined, value ψ, the arrays $\mathcal{S}_i$ are split into *local quantiles* $\mathcal{S}_i^1, \ldots, \mathcal{S}_i^\psi$ which yields ψ global subproblems $\mathcal{S}^j := \mathcal{S}_0^j + \cdots + \mathcal{S}_{p-1}^j$, a.k.a. *global quantiles*. Per the requirement for an ordered partition, the familiar invariant for all quantiles $i < j$ and strings $s^i \in \mathcal{S}^i$ and $s^j \in \mathcal{S}^j$, the order $s^i < s^j$ must hold[1]. By sorting each quantile individually, a globally sorted order for the global instance is therefore induced via simple concatenation in order. It remains to find a viable format that can be used to incrementally build the order of $\mathcal{S}$. Storing entire strings is clearly not an option due to the established memory limitations. More generally, any format where the result is distributed according to lexicographical order is also not feasible, since it would require that a single PE gather results for a round of sorting from all other PEs in the worst case. This rules out a distributed permutation per Definition 1. We therefore propose to use an inverse permutation as described in Definition 2. Recall that an inverse permutation is an array $\overline{\mathcal{P}}_i$ of the same size as $\mathcal{S}_i$ that stores the

[1]To make some of the notation in this chapter easier to follow, note that, when used in reference to an array, indices in the superscript always refer to a quantile, while indices in the subscript always refer to a PE.

globally sorted rank of each string. The working principle for space-efficient algorithms follows intuitively:

(1) Compute an ordered partition of $\mathcal{S}$ into ψ parts and derive local quantiles $\mathcal{S}_i^1, \ldots, \mathcal{S}_i^\psi$.

(2) For each quantile $j = 1, \ldots, \psi$:

  (2.1) Globally sort the jth quantile $\mathcal{S}^j$ using algorithm $\mathcal{A}$ to obtain a distributed inverse permutation $\overline{\mathcal{P}}_i^j$ on each PE.

  (2.2) Use $\overline{\mathcal{P}}_i^j$ to determine the rank of each string in $\mathcal{S}_i^j$ with respect to the original array $\mathcal{S}$ and write them into an inverse permutation $\overline{\mathcal{P}}_i$ at their position in $\mathcal{S}_i$.

To formalize how ranks are determined for the combined inverse permutation $\overline{\mathcal{P}}_i$, we require a mapping from quantile to the original string array. This can be realized with functions ϕ_i^j that map positions in the jth quantile $\mathcal{S}_i^j$ to positions in $\mathcal{S}_i$ on PE i. Thus, for every position l the corresponding strings $\mathcal{S}_i[\phi_i^j(l)]$ and $\mathcal{S}_i^j[l]$ must be equal. The transformation of ranks—which basically adds the number of strings in previous quantiles to the rank—can easily be formalized for PE i and quantile j as

$$\overline{\mathcal{P}}_i\big[\phi_i^j(l)\big] = \Delta^j + \overline{\mathcal{P}}_i^j[l] \qquad \text{where} \qquad \Delta^j \coloneqq \sum_{l=1}^{j-1} \big|\mathcal{S}^l\big|.$$

Functions ϕ_i^j can be realized by storing the permutation computed during local sorting. The value of Δ^j requires global knowledge and can be computed using a single all-reduce operation per round of sorting to compute the sum $|\mathcal{S}_0^j| + \cdots + |\mathcal{S}_{p-1}^j|$.

It remains to determine a suitable number of a quantiles ψ for given set of strings. Ideally, if the runtime of the employed string sorting algorithm $\mathcal{A}$ is roughly linear in the input size, one may be inclined to minimize ψ—and thus the number of sorting rounds—by ensuring that each local quantile contains the maximum X characters, except for rounding errors. In this case, the number of quantiles would be $\lceil N/pX \rceil$ with each containing approximately pX characters globally. This approach, however, is not feasible in general and may not even be desirable, given the established constraints on knowledge of X. An ordered partition where each local quantile contains the same number of elements requires a uniformly random input distribution, completely independent of lexicographical order. This reveals a fundamental limitation of our space-efficient algorithms: Even for initially balanced inputs, where each PE starts out with an equal workload, it is possible to construct instances that induce arbitrarily imbalanced local quantiles. Simply require that local inputs arrays $\mathcal{S}_i$ are already globally ordered and ensure character size $\|\mathcal{S}_i\|$ is greater than X. Every ordered partition with no more than X characters in any local quantile will contain strings from at most two consecutive PEs in any global quantile. Consequently, the number of quantiles would need to be in $\Theta(N/X)$, a factor p worse than the minimum. Such a scenario could usually be resolved through an initial random redistribution of elements—as is the case for RQuick. However, due to the compressed input format, it is not possible to redistribute strings independently without materializing the input first and losing any benefit of compression.

The preceding observation leads us to require a number of constraints on inputs to space-efficient algorithms. As with the analysis of multi-level algorithms from Chapter 4 local arrays must be balanced, i.e., $|\mathcal{S}_i| = \Theta(n/p)$ and $\|\mathcal{S}_i\| = \Theta(N/p)$ on every PE i. We also

assume that the input is randomly distributed and choose ψ such that the probability of local quantiles greater than X is sufficiently small. Hence, there is a small probability that our algorithm fails because $\mathcal{A}$ is unable to sort at least one of the computed quantiles. It would be possible to handle such cases correctly, for example by further partitioning offending quantiles. We did however not investigate this possibility further, due to the established uncertainty around the value of X and to allow reasonable runtime analysis.

For the following proofs, we introduce notation $(\mathcal{T})_a^b \coloneqq [s \in \mathcal{T} \mid a \leq s < b]$ to denote the quantile of any string array $\mathcal{T}$ defined by lower bound a and upper bound b. The following lemma and theorem assume that in addition to strings, characters are also distributed randomly, i.e., for a global quantile $(\mathcal{S})_a^b$, the expected value for the number of characters on each PE is $\|(\mathcal{S})_a^b\|/p$.

Lemma 7. *Let a and b be strings from $\mathcal{S} \cup \{-\infty, +\infty\}$, and let $\varepsilon \in \mathbb{R}$ with $0 < \varepsilon < 1$. If the characters in $(\mathcal{S})_a^b$ are distributed over PEs uniformly at random and $(\mathcal{S})_a^b \leq \varepsilon p X$, then the probability $\mathbf{P}\big(\big\|(\mathcal{S}_i)_a^b\big\| > X\big)$ is less than $e^{-(1-\varepsilon)^2/(\varepsilon+1)X}$ for all $i \in [0, p)$.*

Proof. Let a, b, and ε be defined as required by the lemma. Characters in $(\mathcal{S})_a^b$ are assumed to be randomly distributed over PEs. The number of characters on a specific PE can therefore be seen as the sum of independent and identically distributed random binary variables, i.e., Bernoulli variables. We can therefore apply a Chernoff bound [21] to estimate the desired probability. It can be seen that the probability is maximized for $(\mathcal{S})_a^b = \varepsilon p X$ which yields an expected value of εX for every $(\mathcal{S}_i)_a^b$. The stated bound follows by rewriting with simple arithmetic identities:

$$\begin{aligned}\mathbf{P}\Big(\Big\|(\mathcal{S}_i)_a^b\Big\| > X\Big) &= \mathbf{P}\left(\Big\|(\mathcal{S}_i)_a^b\Big\| > \frac{1}{\varepsilon}\,\mathbf{E}\Big(\Big\|(\mathcal{S}_i)_a^b\Big\|\Big)\right) \\ &\leq \exp\left(-\frac{(1/\varepsilon - 1)^2}{1 + 1/\varepsilon}\varepsilon X\right) = \exp\left(-\frac{(1-\varepsilon)^2}{1+\varepsilon}X\right)\end{aligned}$$

□

Lemma 7 states the probability of a single local quantile exceeding X characters for a given value of ε. Using this result, we can show that space-efficient sorting is unlikely to fail for a particular choice of parameters. The following theorem assumes a quantile imbalance due to imperfect partitioning of at most two and estimates the desired probability for a choice of $\varepsilon \coloneqq \frac{1}{2}$. The parameters yield and *overpartitioning factor* for quantiles of $\gamma \coloneqq 4$.

Theorem 13. *For quantile imbalance $\delta \coloneqq 2$ and parameter $\varepsilon \coloneqq \frac{1}{2}$, let $\gamma \coloneqq \delta/\varepsilon$ and $\varepsilon' \coloneqq (1+\varepsilon)/(1-\varepsilon)^2$. If characters are distributed over PEs uniformly at random, $\mathcal{A}$ can sort at least $X \geq \varepsilon' \log(N/p)$ characters, there are $\psi \coloneqq \gamma N/pX$ quantiles, $N = \omega(p)$ characters in total, and at most $\delta p X/\gamma$ characters per global quantile, then space-efficient sorting succeeds with high probability.*

Proof. Space-efficient sorting succeeds if no local quantile exceeds X characters—or equivalently if every local quantile contains at most X characters. The success-probability $\mathbf{P}_{\text{succ}}$ can therefore be estimated using the bound from Lemma 7. With overpartitioning factor γ and imbalance parameter δ, each quantile contains at most $\delta p X/\gamma$ characters. Hence, let a

and b be defined as stated in Lemma 7 with ε from the theorem. Noting that the probability is negated, this yields inequality (i). Furthermore, the success probability is minimized for the smallest value of X, since the number of quantiles is maximized for this choice. With $(1-\varepsilon)^2/(1+\varepsilon)$ canceling out ε' in the scaling factor of X, inequality (ii) follows.

$$\begin{aligned}\mathbf{P}_{\text{succ}} &= \mathbf{P}\left(\forall i \left\|(\mathcal{S}_i)_a^b\right\| \leq X\right) = \left(1 - \mathbf{P}\left(\exists i \left\|(\mathcal{S}_i)_a^b\right\| > X\right)\right)^{\psi} \\ &\overset{\text{(i)}}{\geq} \left(1 - \exp\left(-\frac{(1-\varepsilon)^2}{1+\varepsilon} X\right)\right)^{\frac{\gamma N}{pX}} \\ &\overset{\text{(ii)}}{\geq} \left(1 - \exp\left(-\log\frac{N}{p}\right)\right)^{\frac{\gamma N/p}{\varepsilon' \log(N/p)}} = \left(1 - \frac{1}{N/p}\right)^{\frac{6N/p}{\log(N/p)}}\end{aligned}$$

It remains to show that the stated lower bound is sufficient to demonstrate high success probability. Here, we can apply the precondition $N = \omega(p)$, i.e., N dominates p asymptotically. We can therefore see the term N/p as increasing towards infinity and, by substituting for x, obtain the following limit:

$$\lim_{x\to\infty}\left(1 - \frac{1}{x}\right)^{x/\log x} = 1$$

The additional factor 6 in the exponent does not change this result, which finally justifies the claim that no local quantile exceeds X characters with high probability. □

The usefulness of Theorem 13 may not be immediately clear due to the number of necessary preconditions. To summarize, given a partition of $\mathcal{S}$ into a number of quantiles that is only worse than the optimal value by a constant factor. With additional preconditions on the input size and the minimum capability of $\mathcal{A}$, and if global quantiles are imbalanced by a factor at most two, then space-efficient sorting is likely to succeed with high probability. Equivalent statements could be shown for quantiles with greater imbalance parameters δ.

5.2. Space-Efficient Merge Sort

Having established the basic framework for space-efficient string sorting, we can now use distributed string merge sort instead of the black-box $\mathcal{A}$ to realize it. We call the result *Space-Efficient String Merge Sort* (SEMS) regardless of whether single- or multi-level, MS or PDMS is used. Rather than a straightforward implementation of the framework, we also propose a number of optimizations to improve runtime. Section 5.2.1 focuses on the partitioning phase of SEMS and Section 5.2.2 aims to reduce the number of quantiles by applying distinguishing prefix approximation. Finally, the resulting algorithm's complexity is analyzed in Section 5.2.3.

5.2.1. Sorting First versus Partitioning First

The framework for space-efficient sorting leaves the details of ordered partitioning unspecified. Here, the opportunity presents itself to reuse the distributed ordered partitioning algorithm

from Section 3.2 for SEMS. Recall that this requires sampling $\mathcal{S}_i$, sorting the samples, and drawing the final splitters at regular intervals. To apply the bounds on bucket imbalance for regular sampling, we need to ensure that local arrays are sorted before sampling. Alternatively, we could restrict ourselves to using a random sampling technique without the same precondition. This yields two possible variants of SEMS:

(a) Locally sort string arrays $\mathcal{S}_i$, then compute an ordered partition of $\mathcal{S}$, and obtain quantiles as contiguous intervals of $\mathcal{S}_i$.

(b) Compute an ordered partition of $\mathcal{S}$, obtain quantiles as non-contiguous subsets of $\mathcal{S}_i$, and locally sort each quantile individually.

Sorting first has the advantage that quantiles consist of successive element in the string array. We may therefore find intervals of $\mathcal{S}_i$ using as a single scan thereof, which can exploit available LCP information. However, sorting the string array as a whole, instead of only sorting quantiles individually, may be undesirable, especially for algorithms with superlinear complexity. On the other hand, partitioning an unsorted array cannot use LCP values to speed up string comparisons. Rather than being able to find the position of splitters within $\mathcal{S}_i$ to obtain intervals thereof, we instead need to locate each string in the sequence of $\psi - 1$ splitters. A naive approach could compute LCP values of splitters and use linear search for a runtime in $O(\|\mathcal{S}_i\| + \psi|\mathcal{S}_i|)$. The second term can be improved to a factor $\log \psi$ using binary search. This requires constant time range minimum queries (RMQs) to efficiently compute LCP values for any pair of splitters. An appropriate data structure can be constructed with linear preprocessing time [17]. Table 5.1 compares runtimes of both variants with MSD radix sort as local string sorter. We assume that both approaches use the same distributed algorithm to obtain splitters and omit the corresponding runtime.

phase	— Variant (a) — sort first	— Variant (b) — partition first
local sorting	$O(\mathcal{D}(\mathcal{S}_i) + \lvert\mathcal{S}_i\rvert \log \sigma)$	$O\left(\sum_{j=1}^{\psi}\left(\mathcal{D}\left(\mathcal{S}_i^j\right) + \lvert\mathcal{S}_i^j\rvert \log \sigma\right)\right)^{\dagger}$
partitioning	$O\left(\mathcal{D}(\mathcal{S}_i) + \lvert\mathcal{S}_i\rvert + \psi\hat{\ell}\right)^{\ddagger}$	$O\left(\|\mathcal{S}_i\| + \lvert\mathcal{S}_i\rvert \log \psi + \psi\hat{\ell}\right)^{\ddagger}$
total	$O\left(\mathcal{D}(\mathcal{S}_i) + \lvert\mathcal{S}_i\rvert \log \sigma + \psi\hat{\ell}\right)$	$O\left(\|\mathcal{S}_i\| + \lvert\mathcal{S}_i\rvert \log(\psi\sigma) + \psi\hat{\ell}\right)$

† the sum simplifies to $O(\mathcal{D}(\mathcal{S}_i) + |\mathcal{S}_i| \log \sigma)$ because $\sum_{j=1}^{\psi} \mathcal{D}(\mathcal{S}_i^j) \leq \mathcal{D}(\mathcal{S}_i)$

‡ the term $O(\psi\hat{\ell})$ accounts for computation of splitter LCP values and construction of an RMQ data structure (assumes $\hat{\ell} > 0$)

Table 5.1.: Comparison of local work to obtain quantiles on a specific PE i by sorting before or after partitioning.

We can conclude that it is preferable to sort the entire, compressed string arrays prior to partitioning if MSD radix sort is used. Partitioning string arrays first incurs character comparisons outside the distinguishing prefix and adds a component $O(|\mathcal{S}_i| \log \psi)$ to the runtime. Accordingly, the remainder of this chapter, as well as the experimental evaluation thereafter, will use Variant (a) of the algorithm.

5.2.2. Prefix Approximation First versus Partitioning First

Similar considerations to the preceding section on local sorting can be made for distinguishing prefix approximation when it is applied to space-efficient sorting. It is again possible to partition the input into quantiles first and apply duplicate detection to each subset of strings. Alternatively, prefixes can be approximated with respect to the complete array $\mathcal{S}$, which is then partitioned based on the resulting prefixes.

(c) Compute an ordered partition of $\mathcal{S}$ to obtain quantiles $\mathcal{S}^1, \ldots, \mathcal{S}^\psi$. Separately approximate distinguishing prefixes of each quantile and sort the resulting prefixes.

(d) Approximate distinguishing prefixes of $\mathcal{S}$ to obtain $\widetilde{\mathcal{S}}$. Compute an ordered partition of the resulting prefixes into ψ' quantiles as $\mathcal{S}^1, \ldots, \mathcal{S}^{\psi'}$ and sort the result.

We can again consider advantages and disadvantages of either variant. By approximating distinguishing prefixes first and using the shortened strings to compute quantiles, we reduce the total number of characters and therefore the number of required quantiles. Otherwise, especially if $D \ll N$, each quantile may contain significantly fewer characters than could be sorted with the available memory. If the number of quantiles is in $\Theta(N/pX)$ for the original strings, then it is in $\Theta(\mathcal{D}(\mathcal{S})/pX)$ for the reduced strings. Hence, the usage of ψ and ψ' with $\psi' \leq \psi$ to disambiguate both cases. We also need to take the size of resulting distinguishing prefixes into account. For any partition $\mathcal{S}^1, \ldots, \mathcal{S}^\psi$, regardless of whether it is ordered, the sum of distinguishing prefixes of individual quantiles is no larger than for the original array. However, because the partition is ordered, each additional quantile decreases the sum by at most $\hat{\ell}$ characters. Thus, at least for small values of $\hat{\ell}$, partitioning first does not yield significantly improved distinguishing prefixes. Partitioning first means that the prefix approximation algorithm needs to be run ψ times which adds a corresponding factor to latency. Individual executions, however, can use smaller Bloom filters, i.e., hash functions with fewer possible values, to achieve the same false positive rates. On the other hand, approximating distinguishing prefixes first has the advantage that we may use the length of the longest distinguishing prefix to bound the runtime of partitioning. Runtimes for distinguishing approximation and ordered partitioning are summarized in Table 5.2.

Table 5.2 shows that Variant (c) has overall worse runtime. Aside from bounds for partitioning using $\hat{\ell}$ rather than $\hat{d}$, distinguishing prefix approximation also results in unfavorable worst-case bounds. It is possible to construct instances where individual quantiles concentrate strings on single PEs. This results in a longer critical path during execution. Therefore, using the maximum distinguishing prefix to bound the sum of maxima for individual quantiles would not be valid. Thus, the following analysis assumes that distinguishing are approximated first, i.e., Variant (d).

5.2.3. Runtime and Communication

Having considered different variants of SEMS and having already analyzed the complexity of individual phases in preceding sections, the following provides bounds for the algorithm's runtime and communication in their entirety. The following analysis uses basic single-level merge sort without prefix approximation to keep bounds simple and because we do not aim to evaluate the complexity of individual merge sort variants. We use the established distributed

	— Variant (c) — partition first	— Variant (d) — approximate first
	prefix approximation	
local work	$O\left(\log \hat{d} \sum_{j=1}^{\psi} \max_i \mathcal{D}_{\mathcal{S}^j}(\mathcal{S}_i^j)\right)$	$O\left(\log \hat{d} \max_i \mathcal{D}_{\mathcal{S}}(\mathcal{S}_i)\right)$
latency	$O\left(\alpha\psi k \sqrt[k]{p} \log \hat{d}\right)$	$O\left(\alpha k \sqrt[k]{p} \log \hat{d}\right)$
communication	$O\left(\sum_{j=1}^{\psi} \beta k \max_i \lvert\mathcal{S}_i^j\rvert \log \hat{d} \log p\right)$	$O\left(\beta k \frac{n}{p} \log \hat{d} \log p\right)$
	ordered partitioning	
local work	$O\left(\frac{n}{p} \log \sigma + \psi\hat{\ell} \log p\right)$	$O\left(\frac{n}{p} \log \sigma + \psi' \hat{d} \log p\right)$
latency	$O\left(\alpha \log^2 p\right)$	
communication	$O\left(\beta\psi\hat{\ell} \log p \log \sigma\right)$	$O\left(\beta\psi' \hat{d} \log p \log \sigma\right)$

Table 5.2.: Comparison of runtimes for distinguishing prefix approximation and distributed ordered partitioning depending on the order of execution. Runtimes assume the multi-level Bloom filter from Section 4.2.1 and LCP-RQuick. Order in the table *does not* correspond to the order of execution.

ordered partitioning algorithms with character-based regular sampling to determine quantiles and to determine buckets during each sorting round.

As discussed in Section 5.1, space-efficient sorting by definition only succeeds if no local quantile exceeds X characters. Theorem 13 states that the algorithm succeeds with high probability for $\psi := 4N/pX$ quantiles if each global quantiles contains at most $2N/\psi$ characters. We therefore need to ensure that partitioning yields quantiles that fulfill this requirement. Recall Theorem 3 which states that character-based regular sampling yields buckets of size at most $N/\psi + N/v + (pv/\psi + p)\hat{\ell}$ with v samples per PE. Unfortunately, the final term cannot be meaningfully bounded without additional assumptions. If the number of samples is chosen as $\Theta(\psi)$, say $v = \mu\psi$ for some $1 < \mu \in \mathbb{N}$, then the bound simplifies to $(1 + 1/\mu)N/\psi + (\mu + 1)p\hat{\ell}$. To stay below $2N/\psi$ characters per local quantile, the second term must therefore be bound by

$$(\mu + 1)p\hat{\ell} \leq \frac{\mu - 1}{\mu} \cdot \frac{N}{\psi} = \frac{\mu - 1}{\mu} \cdot \frac{NpX}{4N} = \frac{\mu - 1}{4\mu} pX.$$

Thus, if the length of the longest string is bounded by X as follows, then global quantiles are guaranteed not to exceed $pX/2$ characters, and the algorithm succeeds with high probability.

$$\hat{\ell} \leq \frac{\mu - 1}{4(\mu^2 + \mu)} X$$

Obviously, the term μ^2 in the denominator means that increasing the number of samples places tighter restrictions on $\hat{\ell}$. Due to the nature of string sorting, where it is always possible

to construct instances with disproportionately long strings, this is the most general statement we were able prove. Keep in mind that this does not imply that SEMS is necessarily likely to fail in cases where the guarantees provided here do not hold. For the following runtime analysis, we simply choose a value ψ and assume that the algorithm succeeds.

Theorem 14. *If algorithm SEMS succeeds with $\psi = O(N/pX)$ quantiles, then it has expected running time in*

$$O\left(\frac{N}{p} + \left(\frac{n}{p} + \psi p\hat{\ell}\right)\log(\sigma p) + \alpha\psi p + \beta\left(\frac{N}{p}\log\sigma + \frac{n}{p}\log n + \psi p\hat{\ell}\log p\log\sigma\right)\right).$$

Proof. As ever, MSD radix sort can be used to sort local string arrays in time $O(N/p + n/p\log\sigma)$. Obtaining quantiles with the ordered partitioning algorithm from Section 3.2 requires time in

$$O\left(\psi\log\sigma + \psi\hat{\ell}\log p + \alpha\log^2 p + \beta\left(\hat{\ell}\log^2 p + \psi\hat{\ell}\log p\right)\log\sigma\right)$$

to sort $\Theta(\psi)$ samples using hypercube quicksort. It can be seen that this is dominated by the bound stated in the theorem. Additional work to obtain samples and to locate splitters is in $O(N/p + \psi\hat{\ell}\,)$.

For the actual sorting phase of the algorithm, it is necessary to establish bounds for the size of quantiles. It is known that the number of characters in local quantiles, i.e., local character arrays during sorting rounds, is in $O(X) = O(N/p\psi)$ since we can assume that the algorithm succeeds. With character-based regular sampling, this yields buckets containing $O(N/p + p\hat{\ell})$ characters. For space-efficient sorting, unlike for multi-level sorting in Section 4, we make the assumption that there exists no correlation between string length and lexicographical order. We can therefore bound the number of strings, at least in expectation, by $O(n/p\psi)$. By substituting the bounds for local quantile sizes into the established runtime for single-level string merge sort we obtain the following expected complexity for the ψ sorting rounds of SEMS:

$$O\left(\psi\left(\overbrace{\frac{N}{p\psi} + p\hat{\ell} + \frac{n}{p\psi}\log p}^{\text{local merging}} + \overbrace{\alpha p + \beta\left(\left(\frac{N}{p\psi} + p\hat{\ell}\right)\log\sigma + \frac{n}{p\psi}\log n\right)}^{\text{string exchange}} \right.\right.$$
$$\left.\left.\underbrace{+\,p\log\sigma + p\hat{\ell}\log p + \alpha\log^2 p + \beta\left(\hat{\ell}\log^2 p + p\hat{\ell}\log p\right)\log\sigma}_{\text{partitioning}}\right)\right)$$

The bound from the theorem follows with a number of simplifications and by combining the terms $\psi p\log\sigma$ and $\psi p\hat{\ell}\log p$. Note that imbalance introduced by imperfect partitioning, i.e., $p\hat{\ell}$ in string exchange, is dominated by the term for partitioning. Finally, in the single-level context, writing the inverse permutation back to original PEs requires time in $O((1 + \beta\log n)(n/p\psi))$ which is the same as LCP-value exchanges. □

At least for this simple variant of SEMS, the factor ψ introduced by multiple sorting rounds is partially canceled out, in terms of local work and communication, by the factor $1/\psi$ in quantile size. For multi-level variants using the Exch(·) black-box for communication or variants using prefix approximation, similar guarantees cannot be established as easily.

Obviously, latency and the overhead for distributed partitioning increase linearly with ψ in all cases. In scenarios where performance is dominated by string exchange and merging times, it is therefore not inherently beneficial to reduce the number of quantiles.

5.3. Application in Suffix Sorting

The preceding sections have already introduced space-efficient string sorting and have shown how it can be used to sort inputs that would otherwise exceed the memory limits of PEs in a distributed system. In this section, we describe how the scheme can be used, with a slightly tweaked output format, as subroutine to the DCX suffix sorter. Provided additional future work, this could make it possible to use DCX with much larger difference covers than previously possible on realistic distributed systems. Larger difference covers use longer substrings during sorting which means fewer duplicates and therefore faster overall suffix array construction.

For the new context of suffix sorting, we require a number of additional definitions. A *text* $\mathcal{T}$ is an array of m characters which is distributed over PEs in consecutive slices of $\Theta(m/p)$ characters. The *suffix array* of $\mathcal{T}$ defines a sorted permutation of $\mathcal{T}$'s suffixes. Proposed by Kärkkäinen and Sanders [26, 27], DC3 and its generalization DCX are linear time suffix array construction algorithms. They broadly work by sampling fixed length substrings of $\mathcal{T}$ at regular intervals, sorting the samples, and using the resulting ranks to obtain the order of corresponding suffixes. If samples are not unique, then the suffix array over the resulting ranks is computed recursively using the same principle. This section describes how the required samples are obtained for a given text such that they can be sorted using space-efficient string sorting. It is also shown how the existing output format can be adapted to conform to the requirements set out by DCX. We explicitly do not cover how the result needs to be processed to compute ranks for sampled suffixes. Furthermore, details for recursive sorting steps are left unspecified. Sorting ranks rather than characters, i.e., texts over an alphabet of size $O(m)$ rather than of small, independent size σ, may require additional considerations in the implementation of sorters. For example, any algorithm storing arrays with entries for every characters of the alphabet, such as MSD radix sort, are no longer viable due to memory limitations. On realistic, byte-addressable systems, where string ranks are represented as machine words, an alternative to having a separate sorting variant for the recursive case is obtained by exploiting the memory representation of integers. Given an appropriate byte-order, it is possible to interpret integer arrays as character arrays and therefore to sort both types of strings with the same implementation of space-efficient sorting.

Sampling positions for DCX are obtained from a *difference cover modulo* X—hence the algorithm's name. Since we already use X to refer to the character limit for quantiles in space-efficient sorting, we adhere to convention and use v to refer to the modulus.

Definition 3 (Difference Cover [27, Definition 4.1])**.** *A set* $\mathsf{DC} \subseteq [0, v)$ *is a difference cover modulo* v *if* $\{(i - j) \bmod v \mid i, j \in \mathsf{DC}\} = [0, v)$.

A difference cover DC modulo v is optimal if there exists no difference cover DC' modulo v with $|\mathsf{DC}'| < |\mathsf{DC}|$. For example, $\mathsf{DC} = \{0, 1, 3\}$ is optimal for $v = 7$. Optimal difference

covers are known for at least $v \leq 111$ [28] and difference covers with no optimality guarantees can be obtained for any value of v [13]. If a range of numbers is sampled at intervals defined by a difference cover, we obtain a *difference cover sample*.

Definition 4 (Difference Cover Sample [27, Definition 4.2])**.** *A difference cover sample of* $[0, n)$ *for a difference cover* DC *modulo* v *is a* v*-periodic sample* $\mathsf{CS} \coloneqq \{i \in [0, n) \mid i \bmod v \in \mathsf{DC}\}$.

For a given difference DC modulo v and difference cover sample CS, we can define the global string array $\mathcal{S}$ to contain the string $\mathcal{T}[i, \min\{i + v, m\})$ for every sampled position $i \in \mathsf{CS}$. Disregarding shortened samples near the end of the text, the resulting string array has uncompressed size of roughly $m|\mathsf{DC}|$ characters which yields a compression ratio of $|\mathsf{DC}| : 1$ for the resulting character array $\mathcal{C}$. Local string arrays $\mathcal{S}_i$ can be obtained by assigning each string to the PE where its first referenced character in $\mathcal{T}$ is located. Strings therefore reference up to $v - 1$ characters succeeding PEs. The additionally required characters can be received with constant latency, through "shifting" copies of characters to the left by one PE, assuming that each slice of $\mathcal{T}$ contains at least v characters.

It is necessary for the correctness of DCX, that identical strings receive equal ranks in the sorted order. Essentially, the result is still an inverse permutation per Definition 2, but with a different meaning of a string's rank. The desired property for an array of ranks $\mathcal{R}$ is sufficiently defined by the following condition:

$$\forall i, j \in [0, n) \colon \mathcal{S}[i] < \mathcal{S}[j] \iff \mathcal{R}[i] < \mathcal{R}[j]$$

Note that $\mathcal{R}$ is not generally a permutation of $[0, n)$. Observe also that the result is unique if we additionally require that the maximum of $\mathcal{R}$ is minimal. For a single quantile of space-efficient sorting the desired ranks can be computed by checking sorted local string arrays $\mathcal{O}_i$ for duplicates. This can be integrated into the procedure for obtaining an inverse permutation described in Section 4.2.2, by only incrementing the current rank if strings are distinct. If up-to-date LCP values are available—as is the case for SEMS—no characters need to be compared and it instead suffices to check if a string's LCP value is equal to its length. The result is still correct if distinguishing prefix approximation has been applied and only shortened strings are available. Comparison of strings on different PEs is not necessary for implementations of space-efficient sorting, such as SEMS, that guarantee strict ordering between PEs. Otherwise, the last string on each PE would additionally need to be compared to the first string on the next non-empty PE with appropriate handling for empty string arrays. Regardless of implementation, the lexicographically smallest string of each quantile must be compared to the largest string from the preceding one. Here it should be noted that prefixes must be approximated with respect to the complete input, not individual quantiles, i.e., using Variant (d) from Section 5.2.2. If prefixes are only unique among the strings of a given quantile, then comparisons between shortened strings from different quantiles may be incorrectly classified as equal or vice versa.

Finally, it is important to note that the ranks in $\mathcal{R}$ are not in the correct order for suffix sorting and would require additional reordering. The global string array is ordered such that strings start out on the same PE as referenced characters, which interleaves members of the difference cover. For usage with DCX, the ranks must be reordered with every sample for a given $k \in \mathsf{DC}$ in series and resulting subsequences delimited by sentinel characters. This completes the topic of applying space-efficient sorting to suffix sorting with DCX.

6. Experimental Evaluation

For distributed-memory algorithms in general, and for string sorting algorithms in particular, a purely asymptotic analysis is not sufficient to satisfactorily establish performance characteristics of any particular algorithm. This chapter therefore comprises an experimental evaluation of practical implementations of the algorithms introduced throughout this thesis. Section 6.1 gives an overview of what has been implemented and discusses some details thereof. Section 6.2 introduces the experimental setup with specifics regarding the systems, algorithms, and inputs employed. The evaluation itself is split into two parts according to the major topics of this thesis: Section 6.3 is primarily concerned with multi-level merge sort, while Section 6.4 focuses on space-efficient merge sort.

6.1. Implementation Details

We implemented a number of algorithms and concepts proposed in this thesis for the purposes of the experimental evaluation. Our code is written in C++ with MPI [30] as standard for interprocess communication and uses the kaMPIng [24] library to wrap around MPI's interface. The implementation is available in a public repository[1] and directly based on Schimek's implementation of single-level MS and PDMS [36]. In the following sections we give a short overview of what has been implemented with additional details for noteworthy aspects.

6.1.1. String Layout and Communication

Unlike with atomic objects, it is not possible to compactly store strings in a contiguous array while allowing efficient implementations of operations like indexing or swapping of elements. Instead, we represent a given string array $\mathcal{S}$ using a data structure with two separate memory allocations. The first allocation stores a contiguous array of characters—compressed or uncompressed depending on the context. Strings may likewise be null-terminated or left without sentinel depending on the use case. The data structure used for regular string sorting assumes null-terminators which gives some additional room for optimization. For space-efficient sorting, due to the compressed input format, strings cannot be assumed to be null-terminated. The second allocation contains an array of structures with an entry for each string consisting of at least a pointer to the string's first character and its length. Depending on the exact sorting algorithm used—specifically, the desired output format—additional information is stored per entry. A distributed permutation requires two additional integers to store the initial rank and index of each string. Inverse permutations need an integer to compute the permutation obtained from local sorting and an integer to record the sending PE

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

during local merging after every all-to-all exchange. Notably, for an inverse permutation, it is never necessary to store both pieces of information concurrently, since they are transferred to and stored in separate arrays after local sorting or merging. It therefore suffices to reserve only a single integer field in the structure. This field, as well as the length, can be stored using 32-bit unsigned integers because it is unrealistic for the length of a single string or the total number of strings to exceed $2^{32} - 1$ on the given system (cf. Section 6.2.1). With these layout optimizations, the structure can be stored using only two 8-byte machine words. Figure 6.1 shows the memory layout of strings for sorting without permutation, with regular permutation, and with inverse permutation.

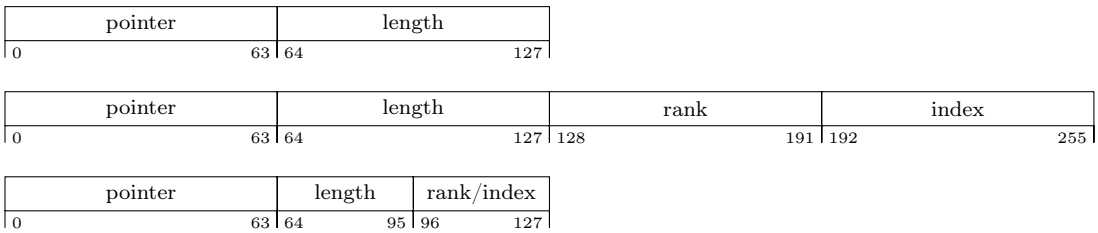

Figure 6.1.: From top to bottom, layout and size in bits of strings without permutation, with distributed permutation, and with distributed inverse permutation.

Strings are exchanged using MPI operations by converting them to a *raw* format consisting of a single character array. Here, strings are always null-terminated and common prefixes may be omitted for LCP compression. String arrays are reconstructed after the exchange using null characters to determine string boundaries. Communication uses either built-in all-to-all exchanges with `MPI_Alltoallv` or direct exchanges with `MPI_Isend` and `MPI_Irecv`. The latter can be used if the size of any sent or received character array exceeds the maximum allowed by collective operations, which is defined by the size of a C `int` in practice (i.e., usually $2^{31} - 1$). Direct exchanges are also preferable if nontrivial group assignment strategies are used, as built-in all-to-all exchanges are likely not optimized for sparse communication with roughly one out of p' PEs being sent to or received from.

Constructing permutations may require additional communication during string exchanges. Specifically, for a distributed permutation, the original rank and string index need to be exchanged using separate all-to-all (or direct) exchanges. For inverse permutations, the required ranks can be inferred from receive counts and do not need to be explicitly communicated. Note that Schimek's original implementation uses a special case of distributed permutations where string indices refer to the locally sorted order of string arrays. The author is thereby able to obtain ranks from receive counts using the same optimization we applied to inverse permutations. By determining the index of the first string received from any PE, the remaining indices can be assigned sequentially. This approach does not work for a multi-level implementation, since the string indices can only be inferred for single round of string exchanges.

6.1.2. LCP-Hypercube Quicksort

We implemented hypercube quicksort in the vein of RQuick and LCP-RQuick from Section 3.1. Our implementation supports sorting with and without LCP-based optimizations which can be configured by passing an appropriate type of string array. The implementation

also supports a tie-breaking scheme to handle duplicate strings. Local string sorting employs the CI3 variant of MSD radix sort from `tlx` [9], which uses a 16-bit super-alphabet and therefore requires 2^{16} words for each entry of the recursion stack. Merging during recursion level uses either `std::merge` from the C++ standard library or our own binary LCP-merge implementation. Similarly, either `std::lower_bound` or an implementation of the same logic which exploits LCP information is used to locate the median. The implementation uses *range-based communicators* (RBC) as proposed and implemented by Axtmann et al. [4]. These wrap around built-in MPI communicators, allowing the creation of subcommunicators without communication overhead through the manual reimplementation of collective communication operations. Our implementation uses RBC to obtain communicators for the current hypercube, which simplifies addressing. String and LCP-value exchanges use (non-blocking) point-to-point communication, and thus do not strictly require any particular communicator. However, median approximation uses a broadcast to distribute the result which is made easy using RBC communicators.

6.1.3. Multi-Level Merge Sort

The first major part of our implementation consists of an implementation of multi-level MS and PDMS. Our implementation works for an arbitrary number of levels and only requires that every r is greater than one and is a divisor of (the current value of) p. The size and total number sorting levels is determined by a sequence of descriptors where each defines a sorting level. For a given PE and level, such a descriptor consists of (references to) three MPI communicators containing the following sets of PEs:

1. All PEs in the same group on the current sorting level
2. Only PEs that need to communicate during a string exchange
3. All PEs in the same group on the next sorting level

The first and third communicators are uniquely defined for a particular set of parameters, whereas the constituents of the second may vary depending on the employed group assignment strategy. If grid-wise assignment is used, the communicator consists of PE rows; otherwise, it contains the entire group of PEs. The final level of sorting is handled separately, without group assignment, and only uses a single communicator. All subcommunicators are constructed prior to sorting, though we include construction times in overall sorting times. Albeit, if string sorting is used as a subroutine to a more extensive algorithm, appropriate subcommunicators may already be available. Construction times were generally negligible using `MPI_Comm_create`, which only requires communication between PEs that belong to the new group, unlike `MPI_Comm_split`. Template metaprogramming can be used to entirely disable intermediate sorting levels.

Initial local sorting and LCP-array construction again uses the CI3 variant of MSD radix sort from `tlx`. Local merging after string exchanges uses an implementation of LCP-aware loser trees by Bingmann et al. [10]. String exchanges may optionally use LCP compression on strings.

We implemented a number of the assignment strategies from Section 3.3. Grid-wise assignment and naive strategy where strings on PE i are assigned to PE $\lfloor i/r \rfloor$ of each group are trivial. String- and character-based versions of simple assignment were also implemented and work

by manipulating send counts to match the assignment. We only provide a proof of concept for deterministic assignment. This implementation does not use the merging algorithm described by Axtmann et al. and has latency in $O(p)$—defeating the algorithm's point compared to simple assignment.

For PDMS, we implemented the multi-level Bloom filter as described in Algorithm 6. Hash values are generated using a C++ port of `xxHash`[2,3] (specifically 64-bit `XXH3`). Initial sorting of hash values uses sequential *In-place Parallel Super Scalar Sample Sort* (IPS^{4}o) [5]. We employ a multi-way merging implementation using loser trees provided by `tlx` to merge received sequences of hash values after each round of exchanges. Local and remote duplicate detection remain conceptually unchanged from Schimek's implementation.

6.1.4. Space-Efficient Merge Sort

The second major part of our effort consists of an implementation of space-efficient string sorting, specifically, SEMS using multi-level string merge sort as subroutine. The user specifies the desired size of quantiles X, from which the required number of quantiles $\psi := \max_i \lceil \|\mathcal{S}_i\|/X \rceil$ is derived. Quantiles are computed using the implementation of distributed ordered partitioning that is also employed to compute buckets during sorting. Inputs are sorted locally first and partitioned afterwards, i.e., variant (a) from Section 5.2.1. We provide implementations with and without prefix approximation, in the latter case, prefixes are approximated before partitioning according to variant (d) from Section 5.2.2. Quantiles are sorted with the implementations of multi-level MS and PDMS, using template metaprogramming to add logic necessary for deriving inverse permutations for the whole input from sorted orders of individual quantiles. Our implementation supports building an inverse permutation using the procedure described in Section 5.1. Alternatively, the variant necessary for DCX, where equal strings receive equal ranks, can also be configured. Because we optionally use a tie-breaking scheme during partitioning, where string are made unique using their global position in the input, it is possible that lexicographically equal strings end up on different PEs after sorting. It is therefore also necessary to compare the last string on every PE to the globally subsequent one. We use an implementation that is optimal when no PEs are empty, but incurs latency in $O(p)$ in the worst case. The implementation requires at least two arrays with an integer entry for each string in the original input, to store the inverse permutation and the permutation obtained from local sorting—which needs to be considered when configuring the quantile size.

6.2. Experimental Setup

We evaluate the implemented algorithms in a number of configurations and on a variety of inputs to analyze their performance. This section aims to give an overview of what is being evaluated. Section 6.2.1 describes the systems used to run experiments. Section 6.2.2 lists the evaluated variants of each algorithm. Finally, Section 6.2.3 introduces inputs used in our experiments and establishes important properties.

[2] `https://github.com/RedSpah/xxhash_cpp`
[3] `https://xxhash.com/`

6.2.1. Platforms

All experiments with distributed-memory algorithms were performed on *thin nodes* of the supercomputer SuperMUC-NG Phase 1 [23] at Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences and Humanities (BAdW). Distributed over eight islands, the system comprises 6336 thin nodes with 792 nodes per island. Each node consists of two Intel Skylake Xeon Platinum 8174 processors with 24 cores each, for a total of 48 cores. Of the system's total 304 128 cores we used up to 24 576 in our experiments. The processors run at 2.7 GHz base frequency in regular operational mode and have 33 MiB L3, 24×1 MiB L2, 24×32 KiB L1d, and 24×32 KiB L1i cache. Nodes also contain 96 GiB of main memory shared between both processors. Communication between nodes is provided via a 100 Gbit/s Omni-Path network. The network uses a fat tree topology within islands, with inter-island connections pruned at a ratio of 1 : 4. For our experiments, we ensured that all nodes were allocated within a single island. Furthermore, *Energy Aware Runtime* (EAR) was disabled for all benchmarks. We used Open MPI v4.0.7 as implementation of the MPI standard. Programs were compiled using GCC 11.2.0 with flags `-O3` and `-march=native`. Additionally, link time optimizations (i.e., flag `-flto`) were enable for code in `tlx`.

Experiments with shared-memory algorithms were performed on one of two mostly identical systems at the Institute of Theoretical Informatics, Algorithm Engineering. Both machines consist of a single AMD Epyc Rome 7702P processor with 64 cores running at 2 GHz base and up to 3.35 GHz boost frequency. The processor contains 256 MiB L3, 64×512 KiB L2, 64×32 KiB L1d, and 64×32 KiB L1i cache. In total, 1024 GiB of DDR4 ECC main memory of differing speeds are available to either system. Programs were compiled using GCC 12.1.0, again with flags `-O3 -march=native` and `-flto` for `tlx`.

6.2.2. Algorithms

Our experimental evaluation compares the performance of a variety of algorithms and versions thereof. This section provides an overview of the algorithms, their configuration, and the notation used to refer to them. First up are the following variants of hypercube quicksort:

- `RQuick`/`LCP-RQuick`: Implementation of hypercube quicksort with or without LCP-based optimizations. The input is distributed over all available PEs.
- `RQuick`*/`LCP-RQuick`*: Much like `[LCP-]RQuick` except that the input is only distributed over 32 PEs per node to make the algorithms more efficient.

A fundamental drawback of hypercube-based algorithms is that the number of PEs must be a power of two. For hypercube quicksort this entails an initial data reduction phase (cf. Section 3.1), after which up to half of PEs no longer participate in sorting. For our particular experimental setup on SuperMUC-NG, with 48 PEs per node and with the number of nodes a power of two, this means that one out of three PEs is inactive for the majority of the algorithm[4]. Even worse, due to the sequential assignment of ranks, a third of nodes will be entirely idle. To make the algorithm more competitive and to measure only the actual sorting phase, we add variants `RQuick`* and `LCP-RQuick`*. Here, the system is configured with only 32 active PEs per node using `mpiexec`. Provided the number of nodes is a power of two,

[4]For 2^h nodes we get $p = 48 \cdot 2^h = (2^4 + 2^5) \cdot 2^h = 2^{4+h} + 2^{5+h}$ PEs of which 2^{4+h} are idle.

so is the number of PEs, and therefore no data reduction phase is necessary. This obviously does not correspond to how RQuick is used to sort samples during the partitioning phase of MS and PDMS. It may however be argued that similar effects could be achieved in practice, by providing an implementation of partitioning that uses appropriate subcommunicators to achieve a comparable effect.

Next up are the following variants of single- and multi-level string merge sort with and without prefix approximation.

- $\texttt{MS}_k$: Multi-level string merge sort with k levels of recursion. Single-level variants are obtained by configuring $k = 1$.
- $\texttt{PDMS}_k$: Multi-level prefix doubling string merge sort including prefix approximation using grid-wise Bloom filter with k levels of recursion.
- $\texttt{PDMS}_k^{\triangledown}$: Much like $\texttt{PDMS}_k$, except prefix approximation does not use the multi-level algorithm. This variant is equivalent to $\texttt{PDMS}_k$ for $k = 1$.

All variants use string-based regular sampling with oversampling factor two. We generally use the hypercube quicksort implementation with LCP optimizations to sort samples. If necessary, tie braking is used during partitioning for datasets that otherwise yield poor bucket imbalance. Only naive, grid-wise group assignment is used because we can generally assume that inputs are randomly distributed (cf. Section 3.3.1). LCP compression is used during string exchange phases for datasets where significant common prefixes can be expected.

Multi-level variants always ensure one group per node, i.e., groups of size 48, on the final level of sorting. This means that k-level variants fall back to only $k-1$ or $k-2$ levels of sorting for $p < 2^{k-1}48$. For three-level variants, group sizes for the first two levels are chosen such that splitting factors are as close as possible with a preference for fewer groups on the first level. Table 6.1 lists the resulting values of r for all tested values of p.

		nodes (p/48)									
k	level	1	2	4	8	16	32	64	128	256	512
1	1	48	96	192	384	768	1536	3072	6144	12 288	24 576
2	1	—	2	4	8	16	32	64	128	256	512
	2	48	48	48	48	48	48	48	48	48	48
3	1	—	—	2	2	4	4	8	8	16	16
	2	—	2	2	4	4	8	8	16	16	32
	3	48	48	48	48	48	48	48	48	48	48

Table 6.1.: Configured splitting factors, i.e., values of r, for each level of multi-level merge sort with $k \in \{1, 2, 3\}$ levels and relevant choices of p. Missing entries indicate cases where p is too small to split k times.

The following variants of the space-efficient sorting merge sort implementation are used:

- $\texttt{SEMS}_k$: Implementation of SEMS using multi-level MS with k levels of recursion to sort quantiles.
- $\texttt{PDSEMS}_k$: Implementation of SEMS with prefix approximation and using multi-level PDMS with k levels of recursion to sort quantiles.

The configuration of PE groups, sampling parameters, and LCP optimizations are equivalent to $\texttt{MS}_k$ and $\texttt{PDMS}_k$. By default, quantiles are configured to 100 MiB and obtained using the same parameters used to compute buckets during sorting.

Finally, a number of experiments use a shared-memory implementation of *parallel Super Scalar String Sample Sort* ($\texttt{pS}^5$) [10] from the `tlx` library. To our knowledge, the implementation is still highly competitive in practice.

6.2.3. Inputs

Our experimental evaluation uses a number of real-world and generated inputs. This section lists the used datasets, how they were obtained, and states some basic characteristics.

Fixed D/N Ratio with Uncompressed Strings

The first class of inputs aims to generate string sets with a predictable and configurable D/N ratio (using the established definitions of D and N). Here, the intention is to influence runtimes of local sorting, the effectiveness of LCP compression, and the length of distinguishing prefixes by varying the D/N ratio. For "normal" string sorting, i.e., not space-efficient sorting, we use the *DNGenerator* proposed by Schimek [36]. Given a number of strings n, a string length ℓ, and a desired D/N ratio $q \in (0, 1)$, the idea is to assign a number i from $[0, n)$ to each string and use the unique σ-ary representation of i to distinguish them. Let $\text{repr}_\Sigma(i)$ be the desired representation mapped to the characters of Σ. If the requested length and D/N ratio are sufficiently large, i.e., if $\lfloor ql \rfloor \geq \lceil \log_\sigma(n-1) \rceil$, then the ith string s_i is defined as

$$s_i \coloneqq \underbrace{\mathtt{a} + \cdots + \mathtt{a} + \text{repr}_\Sigma(i)}_{\lfloor q\ell \rfloor} + \underbrace{\mathtt{a} + \cdots + \mathtt{a}}_{\lceil (1-q)\ell \rceil} \quad \text{where } \mathtt{a} \coloneqq \min \Sigma.$$

In our experiments we refer to $D/N = 0$: this is not possible for nonzero n by definition and is simply a placeholder for the smallest achievable ratio. Every string's distinguishing prefix with respect to $\mathcal{S} \coloneqq \{s_0, \ldots, s_{n-1}\}$ consists of its first $\lfloor q\ell \rfloor$ characters. The global distinguishing prefix of $\mathcal{S}$ is therefore $D = n\lfloor q\ell \rfloor$ which, if ℓ is divisible by $1/q$, is equal to qN as desired. Schimek provides more details and proves that the stated distinguishing prefixes are correct. Local string arrays are obtained by distributing strings to PEs uniformly at random and shuffling locally. Improving on the implementation by Schimek, we developed a simple distributed algorithm to distribute strings in time $O(n/p)$ rather than $O(n)$ which provides a significant speedup for large values of p. We refer to the resulting generated datasets as DNData inputs.

Fixed D/N Ratio with Compressed Strings

To exploit the capabilities of space-efficient sorting, we propose DNDataSE inputs as a compressed counterpart to DNData. The idea is to construct a sequence of characters such that, by moving a fixed size window over the characters, the resulting strings have the desired D/N ratio. Formally, let n be the number of strings, ℓ be the length of strings, and $q \in (0, \frac{1}{2}]$ the desired D/N ratio. Note that ratios greater than $1/2$ are not supported. The local character arrays consist of blocks, each of which is used to generate $\lfloor 2q\ell \rfloor \geq 1$ strings. Each block is $\lfloor 2q\ell \rfloor + \ell - 1$ characters long and, similar to DNData, has $\lfloor 2ql \rfloor - 1$ leading padding characters. The remaining ℓ characters are chosen randomly. Let $\mathtt{a} \coloneqq \min \Sigma$ be the padding character as before. We can define the $\lceil n/\lfloor 2q\ell \rfloor \rceil$ blocks b_i using random variables $r_i^0, \ldots, r_i^{\ell-1}$ with uniform distribution over the remaining characters $\Sigma \setminus \{\mathtt{a}\}$ as

$$b_i \coloneqq \underbrace{\mathtt{a} + \cdots + \mathtt{a}}_{\lfloor 2q\ell \rfloor - 1} + \underbrace{r_i^0 + \cdots + r_i^{\ell-1}}_{\ell}.$$

The corresponding strings are be defined as $s_i^j \coloneqq b_i[j, j + \ell)$ for $j \in [0, 2q\ell)$ with $\mathcal{S}$ the set of all strings thus obtained. Local string and characters arrays can be derived by evenly distributing blocks and associated strings to PEs. Strings with equal length are randomly distributed according to the trailing characters. However, note that strings from the same block are ordered according to the number of leading `a`-characters which makes it necessary to shuffle local string arrays. Figure 6.2 illustrates DNDataSE inputs for two combinations of parameters. Observe that the compression ratio of generated strings is determined by the combination of ℓ and q, i.e., ratio $\lfloor 2q\ell \rfloor \ell : \lfloor 2q\ell \rfloor + \ell - 1$.

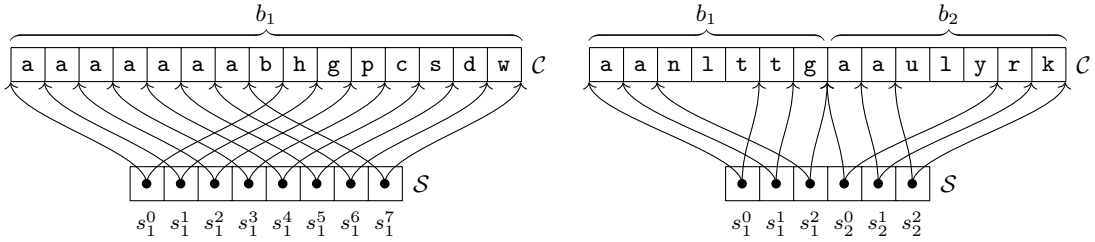

Figure 6.2.: Character and string arrays for two DNDataSE instances over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ for $(n, \ell, q) = \left(8, 8, \frac{1}{2}\right)$ on the left and $\left(6, 5, \frac{1}{3}\right)$ on the right.

Distinguishing prefixes consist of the leading `a`-characters and some of the random characters. Because q is by definition at most $1/2$, and thus $\lfloor 2q\ell \rfloor \leq \ell - 1$, the sum of leading characters is $\sum_{i=0}^{\lfloor 2q\ell \rfloor - 1} i$ per block. Assuming strings divide evenly into blocks (i.e., $2q\ell$ divides n) the sum of leading characters for the whole string array is

$$\frac{n}{\lfloor 2q\ell \rfloor} \sum_{i=0}^{\lfloor 2q\ell \rfloor - 1} i = \frac{n}{\lfloor 2q\ell \rfloor} \cdot \frac{(\lfloor 2q\ell \rfloor - 1)\lfloor 2q\ell \rfloor}{2} = \frac{n(\lfloor 2q\ell \rfloor - 1)}{2}$$

which also provides a lower bound for the size of D. We can estimate the expected size of the distinguishing prefix of character blocks. The probability that the random part of two blocks b_i and b_j share exactly k leading characters is

$$\mathbf{P}\left(\arg\min_{h \in [0,\ell)} \left(r_i^h \neq r_j^h\right) = k\right) = \left(\frac{1}{\sigma - 1}\right)^k \left(1 - \frac{1}{\sigma - 1}\right) = \frac{\sigma - 2}{(\sigma - 1)^{k+1}}.$$

Let the result of $\arg\min(\cdot)$ be defined as ℓ in the case where all characters are equal. Using the sum of the infinite series $\sum_{k=1}^{\infty} kr^k = r/(1-r)^2$ where $|r| < 1$, the expected value can be bound as follows:

$$\mathbf{E}\left(\underset{k\in[0,\ell)}{\arg\min}\left(r_i^k \neq r_j^k\right)\right) = \sum_{k=0}^{\ell} k\frac{\sigma-2}{(\sigma-1)^{k+1}} \leq \frac{\sigma-2}{\sigma-1}\sum_{k=1}^{\infty} k\frac{1}{(\sigma-1)^k} = \frac{1}{\sigma-2}$$

Because leading and random characters are distinct, the expected value for the distinguishing part of random characters is $n/(\sigma-2)$. If we additionally assume that $2q\ell$ is a whole number, then we can establish both a lower bound for and the expected value of the D/N ratio:

$$\frac{D}{N} \geq \frac{n(2q\ell-1)}{2n\ell} = q - \frac{1}{2\ell} \qquad \mathbf{E}\left(\frac{D}{N}\right) = q - \frac{1}{2\ell} + \frac{1}{\sigma-2}$$

This is sufficient for our purposes with large enough values of ℓ and σ and was confirmed empirically, by measuring D/N ratios of the resulting instances. Though, for a more precise analysis, one may chose q to compensate for the error.

Real-World Datasets

Next to the generated inputs with fixed D/N ratios, we also use the following large, real-world datasets to evaluate performance:

- **CommonCrawl** A dataset consisting of WET files from the September/October 2023 Common Crawl archive (`CC-MAIN-2023-40`)[5]. Files in the WET format consist mostly of plain text with only small headers containing minimal metadata. We were unable to obtain consecutive files from the archive due to unpredictable rate limiting behavior during retrieval. The dataset contains 2.5 TB in total.

- **WikipediaFull** A dataset consisting of a prefix of a dump, from 2023-10-01, of all pages in the English Wikipedia with full edit history (`pages-meta-history`) in XML format[6]. Like CommonCrawl, the dataset contains 2.5 TB in total.

- **Wikipedia** A dataset consisting of a dump, from 2023-12-20, of all pages in the English Wikipedia in XML format *without* edit history (`pages-articles`)[7]. The dataset comprises 97.7 GB and is less repetitive than WikipediaFull.

- **WikipediaText** A dataset derived from Wikipedia by deleting all XML metadata, i.e., any lines starting with "<".

More information on the parameters of each dataset, such as D/N ratios, is provided in the evaluation where appropriate.

[5] `https://index.commoncrawl.org/CC-MAIN-2023-40/`
[6] `https://dumps.wikimedia.org/enwiki/20231001/`
[7] `https://dumps.wikimedia.org/enwiki/20231220/`

Difference Cover

In the experimental evaluation, difference cover inputs are used in conjunction with the preceding real word datasets, both for their variable compression ratio and their relevancy to suffix sorting. We use the procedure described in Section 5.3 to construct difference cover samples for a global text by duplicating $v - 1$ characters and shifting them to the previous PE. To our knowledge, no difference covers with accompanying proofs for minimality have been found for $v > 128$. In their paper on *quorums*, Colbourn and Ling show that it it is possible to construct nontrivial difference covers for any v, though without claiming to maintain minimal size [13].

Theorem 15 ([13, Corollary 2.3])**.** *For nonzero $r, v \in \mathbb{N}$, there exists a difference cover modulo v of size $6r + 4$ if $v \leq 24r^2 + 36r + 13$.*

Theorem 15 is constructive, as the authors provide a method for constructing the claimed difference cover. We define members of the difference cover as $d_0, \ldots, d_{6r+3}$ using matching offsets $\delta_0, \ldots, \delta_{6r+2}$. With initial value $d_0 \coloneqq 0$, each subsequent value is obtained by adding the corresponding offset, i.e., $d_{i+1} \coloneqq d_i + \delta_i$. Offsets are defined in six chunks as follows:

$$\begin{array}{llll} \delta_i \coloneqq 1 & i \in [0, r) & \delta_i \coloneqq r + 1 & i = r \\ \delta_i \coloneqq 2r + 1 & i \in [r + 1, 2r + 1) & \delta_i \coloneqq 4r + 3 & i \in [2r + 1, 4r + 2) \\ \delta_i \coloneqq 2r + 2 & i \in [4r + 2, 5r + 3) & \delta_i \coloneqq 1 & i \in [5r + 3, 6r + 3) \end{array} \tag{6.1}$$

The authors show that $\mathsf{DC} \coloneqq \{d_0, \ldots, d_{6r+3}\}$ is a difference cover modulo $24r^2 + 36r + 13$, and thus also for any smaller value of v by extension. We use difference covers modulo v for powers of two 512, 1024, 2048, 4096, and 8192 in our experiments; suitable parameters r and resulting difference cover sizes are listed in Table 6.2. Smaller values of v were ruled out due to the overhead of storing sampled strings. For each element of a difference cover, a strings needs to be constructed every v characters of a text, which yields a string to character ratio of roughly $|\mathsf{DC}|/v$. With the memory layout from Section 6.1.1, each string requires sixteen bytes of memory. For $v = 512$ with a difference cover of size 28 and one byte per character, the size of the string array is 7/8 that of the original text. This ratio improves for larger values of v. Going beyond $v = 8192$ is possible and may be desirable to obtain fewer duplicate strings—the computed difference cover for 16 384 has size 160. However, recall that this is also the compression ratio for the resulting set of strings and therefore results in proportionally more work during sorting.

	difference cover (v)				
	512	1024	2048	4096	8192
r	4	6	9	13	18
$\lvert\mathsf{DC}\rvert$	28	40	58	82	112

Table 6.2.: Size and parameter r of difference covers modulo v computed using Equation (6.1).

6.3. Multi-Level String Merge Sort

Having established our implementation and the experimental setup, we proceed with the actual evaluation. This section focuses on multi-level merge sort, with a number of experiments to evaluate the influence of PE count, n/p ratio, and D/N ratio. Section 6.3.1 includes two weak-scaling experiments using DNData inputs with variable n/p and D/N ratios. Section 6.3.2 comprises a strong-scaling experiment on real-word data sets. Finally, our observations are summarized in Section 6.3.3.

6.3.1. Fixed D/N Ratio Weak-Scaling

Our first experiment uses a weak-scaling setup with DNData inputs and strings of length 500. The experiment is split into two series of runs to compare different aspects of performance.

Comparing n/p Ratios

For the first series, the D/N ratio is fixed at $1/2$ and the number of strings per PE, i.e., the n/p and N/p ratios, is chosen from 10^4, 10^5, and 10^6. This represents a selection of small, medium, and large inputs which serves to illustrate the relative scaling behavior of single- and multi-level merge sort. With character arrays containing 5 MB, 50 MB, and 500 MB per PE, the largest input is near the realistic limit for this system with roughly 2 GB RAM per PE. We compare runtimes for $\texttt{MS}_k$ and $\texttt{PDMS}_k$ with $k \in \{1, 2, 3\}$ on up to 512 nodes (24 576 PEs) which is the most of any of our experiments. The experiments also includes all four variants of hypercube quicksort. For the largest instances and using all nodes, this yields a total input size of more than 12 TB distributed over the system. Figure 6.3 shows the result for the first series of runs. Table A.1 in the Appendix contains exact sorting times. The topmost row of plots shows the overall sorting time for all six variants of $\texttt{MS}_k$ and $\texttt{PDMS}_k$. To gain further insight into the performance of each variant, the sorting times are broken down for each major sorting phase and per level of sorting. Phases correspond to the algorithmic steps from Sections 4.1 and 4.2 which include local sorting, distinguishing prefix approximation (a.k.a. Bloom filter), ordered partitioning, string exchange, and local merging. Wall times per phase were measured during execution on each PE, with a barrier after each measurement to ensure synchronization. The final measurement for a single run is the maximum value of any PE per phase; which means that the sum of phase times can exceed the listed overall sorting time. All measurements are chosen as the median of five runs. Results for $\texttt{MS}_k$ are shown in the middle row and for $\texttt{PDMS}_k$ in the bottom row of the figure. For reasons of compactness, phase breakdowns are only shown for $p/48 \in \{4, 32, 256\}$ nodes.

The results of Figure 6.3 broadly reveal the expected relation between input size and scaling behavior of algorithms. Two-level merge sort significantly outperforms the single-level version on all input sizes for sufficiently large values of p. Adding a third level only leads to further improvements in few cases and substantially deteriorates performance otherwise. As expected, the improvement is most obvious for the smallest inputs with $n/p = 10^4$. Here, the single level algorithm scales roughly linearly with the number of PEs, as runtimes approximately double for every doubling of p. For $\texttt{MS}_1$ the scaling behavior can mostly be attributed to

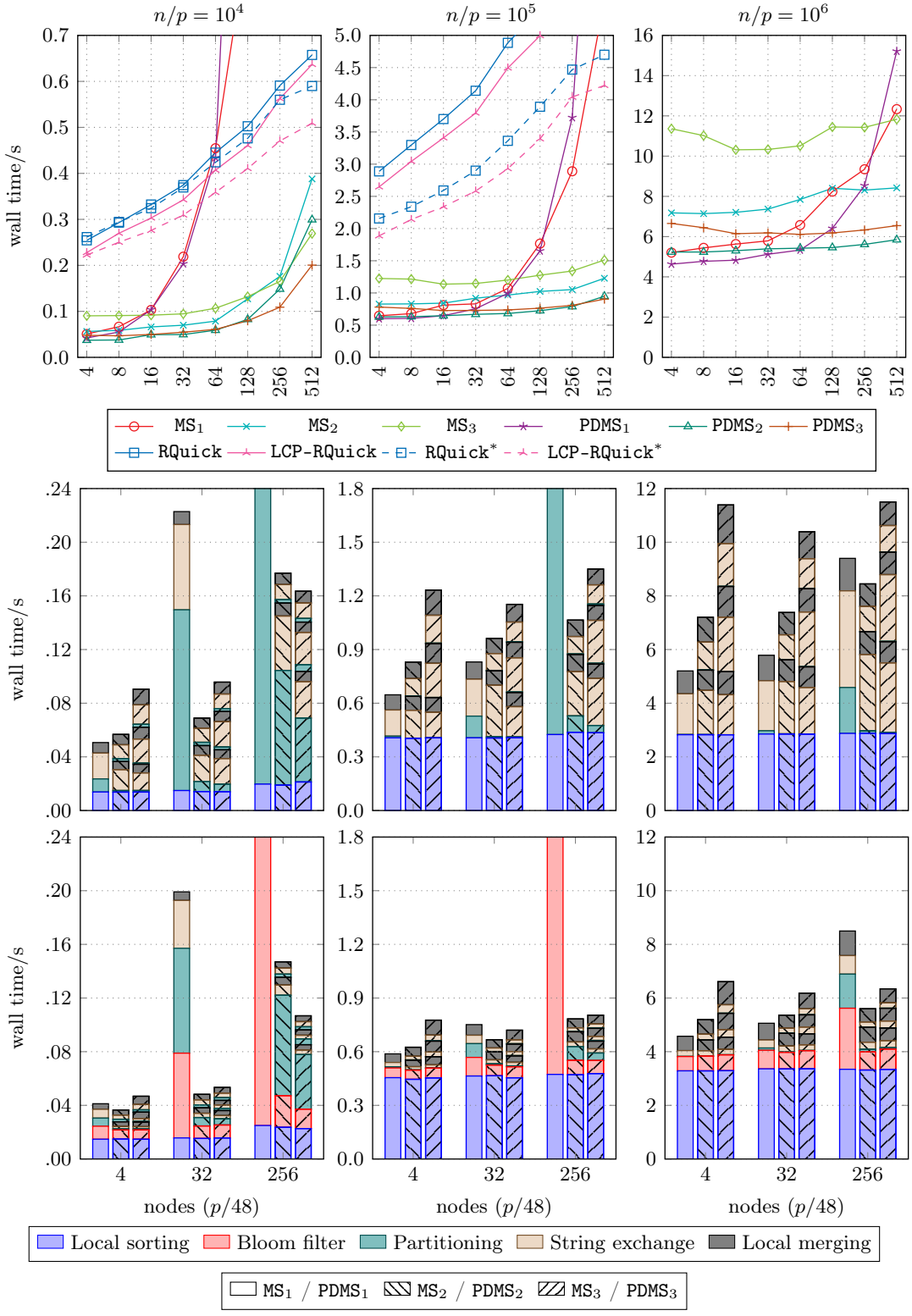

Figure 6.3.: Median overall sorting times (top row) and runtimes per sorting phase for MS$_k$ (middle row) and PDMS$_k$ (bottom row) for the weak-scaling experiment using DNDATA inputs with $\ell = 500$, $D/N = 1/2$, and $n/p \in \{10^4, 10^5, 10^6\}$. Sorting phases are in order of execution starting from the bottom.

the time required for partitioning, with $\Theta(p)$ samples on each PE needing to be sorted. String exchange phases however also account for a notably larger proportion of overall runtime, as latency in $O(p)$ appears to dominate the time for actual data exchange. Prefix approximation dominates runtime for $\texttt{PDMS}_1$ which means that using whole strings is actually faster on more than 64 nodes. At 256 nodes, approximately 54 % of overall runtime is used for prefix approximation. Already for 8 nodes, two-level merge sort yields improved runtimes regardless of whether distinguishing prefix approximation is applied. Looking at runtimes of the prefix approximation phases, it is clear that the multi-level variant scales well. With a two-level Bloom filter, runtimes less than quadruple for 64 times increase in the number of PEs. Three-level versions provide an additional improvement in overall sorting times for 256 and 512 nodes with a speedup of roughly 1.5 for the latter. It is however also apparent that for more than 128 nodes, the time for partitioning also dominates the runtime for multi-level variants. Especially for $\texttt{PDMS}_2$ and $\texttt{PDMS}_3$, the runtime is mostly determined by partitioning during the first round of sorting. Compared to single-level variants, hypercube quicksort performs worse up to 32 nodes but outperforms both $\texttt{MS}_1$ and $\texttt{PDMS}_1$ otherwise. With speedups up to four for $\texttt{MS}_3$ and five for $\texttt{PDMS}_3$, multi-level variants are always better, but a trend is clearly visible, indicating that, for around 1024 to 2048 nodes, they are likely to be inferior. LCP-based optimizations provide speedups up to 1.12 for `LCP-RQuick` over `RQuick`.

On medium-sized inputs ($n/p = 10^5$), the behavior of single- and two-level versions are similar, with a crossover point between 16 and 64 nodes depending on whether $\texttt{MS}_2$ or $\texttt{PDMS}_2$ is used. A third level never yields an unambiguous improvement and instead clearly degrades performance for $\texttt{MS}_3$ with slowdowns ranging form 1.2 to 1.5. For the algorithm without prefix approximation, this can be attributed to the communication overhead caused by an additional string exchange phase. If $\texttt{PDMS}_k$ is used, local sorting contributes the greatest proportion to overall runtimes and local merging dominates additional sorting levels. Both observations can be explained by the make-up of DNData instances, with only small parts of each string needing to be exchanged. For strings of 500 characters and D/N ratio 1/2, all distinguishing prefixes can be expected to be 256 characters long because of prefix doubling. Together with LCP compression, the algorithm ends up exchanging less than 15 characters per string. In either case, partitioning barely contributes to runtimes for multi-level variants. Hypercube quicksort only outperforms single-level variants on at least 512 nodes. Multi-level variants are unambiguously superior, with speedups up to five for $\texttt{MS}_2$ and up to seven for $\texttt{PDMS}_3$ compared to `LCP-RQuick` on 48 PEs per node. Speedups for LCP-based optimizations, hovering around 1.1, are similar to the smaller instances.

Finally, for the largest input size ($n/p = 10^6$), the disadvantage of single-level merge sort is less pronounced, as partitioning proportionally accounts for less of the overall runtime. The two-level version still exhibits better scaling behavior and, at least for $\texttt{PDMS}_2$, never leads to a slowdown of more than 0.85. Meanwhile, three sorting levels are always worse compared to two levels, with even more pronounced slowdowns between 1.3 to 1.6 for $\texttt{MS}_3$. It is plainly visible that the first and second sorting-levels require roughly equal time and are at most marginally faster than the first level of two-level variants. Interestingly, there is marked decrease in runtime for $\texttt{MS}_3$ between 4 and 32 nodes which seems to be caused by faster merging times in the latter case. Given that at 4 nodes the first two levels use only two groups each, we disregarded this observation. Runs for the hypercube quicksort variants consistently failed for the largest input size due to memory exhaustion. An implementation with a more conservative approach to buffer allocations could likely overcome this limitation.

Comparing D/N Ratios

The first series of runs has already shown that multi-level merge sort exhibits improved scaling behavior and that two sorting level are generally preferable for our input sizes. We now aim to further evaluate performance with a second series of runs using DNData inputs. Here, the goal is also to gauge the influence of different D/N ratios. Strings are again 500 characters long, with 10^5 strings per PE. This is equivalent to the medium sized inputs from the first series of runs, which should provide relevant sorting times and clearly show the effects of multi-level variants. Global D/N ratios are varied by choosing the parameter q of DNData inputs in steps of 1/4 which yields five classes with $q \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$. The preconditions for n, q, and ℓ never permit $q = 0$ and we actually use the minimum values of q such that $\lfloor q\ell \rfloor \geq \lceil \log_\sigma(n-1) \rceil$ instead. For the remainder of this section we refer directly to the desired D/N ratio rather than the parameter q to make statements more clear. The set of evaluated algorithms includes single- and two-level versions of $\texttt{MS}_k$ and $\texttt{PDMS}_k$. No three-level variants are part of this or any other experiment going forward. The additional level only yields clearly better performance for 512 nodes, which is also not included in any further experiments due to computing budget constraints. We again evaluate all four variants of `RQuick` as an alternative distributed string sorting algorithm. To provide a baseline for the performance of distributed- compared to shared-memory algorithms, runtimes for $\texttt{pS}^5$ are also included. Here, the number of available cores is left constant (at 64) and the input size is chosen to match the combined input for a given number of PEs.

Results for the experiment on 4 to 128 nodes (192 to 6144 PEs) are shown in Figure 6.4. Exact sorting times are listed in Table A.2 in the Appendix. Due to memory limitations, $\texttt{pS}^5$ could only be run for the equivalent of up to 64 nodes, i.e., $48 \times 64 \times 10^5$ strings totaling 153.6 GB in characters. As before, the topmost row shows overall sorting times for each algorithm. The bottom row again shows a breakdown of sorting times for $\texttt{MS}_k$ and $\texttt{PDMS}_k$ into major phases; this time only for 4 and 128 nodes. The middle row is new and gives an approximation for the number of bytes sent per string for the distributed algorithms. Communication volume is measured on each PE during execution using the size of buffers passed to MPI routines and summed afterwards to arrive at the final value. Different rules are applied depending on the used routine: For example, calls to `MPI_Alltoall` and `MPI_Alltoallv`, as well as reduce and scan operations count the size of send buffers on all PEs. Calls to `MPI_Bcast` only count the send buffer on the root PE and multiply its size by p. The result is only an approximation of the actual communication volume for at least two reasons. Most importantly, the measurements are idealized and do not exactly correspond to actual communication performed by MPI—especially for broadcast and reduction operations. Communication of collective exchange operations may be overestimated if send buffers include data which is already on the correct PE and does not need to be sent. With these caveats in mind, the resulting data still provides useful insight into performance.

We first consider the results for $\texttt{MS}_k$ and $\texttt{PDMS}_k$. The influence of D/N ratio on overall sorting times is clearly visible, as variants with prefix approximation outperform variants without on instances with ratio up to 1/2. For ratios 3/4 and 1, the Bloom filter only adds runtime without providing any benefit as can be seen in the phase breakdowns. Prefix doubling cannot yield shortened strings for the given input sizes, because the nearest tested prefix length is 512 characters. Instead, LCP compression is highly effective due to the nature of DNData inputs. As before, multi-level variants outperform single-level counterparts, usually with a

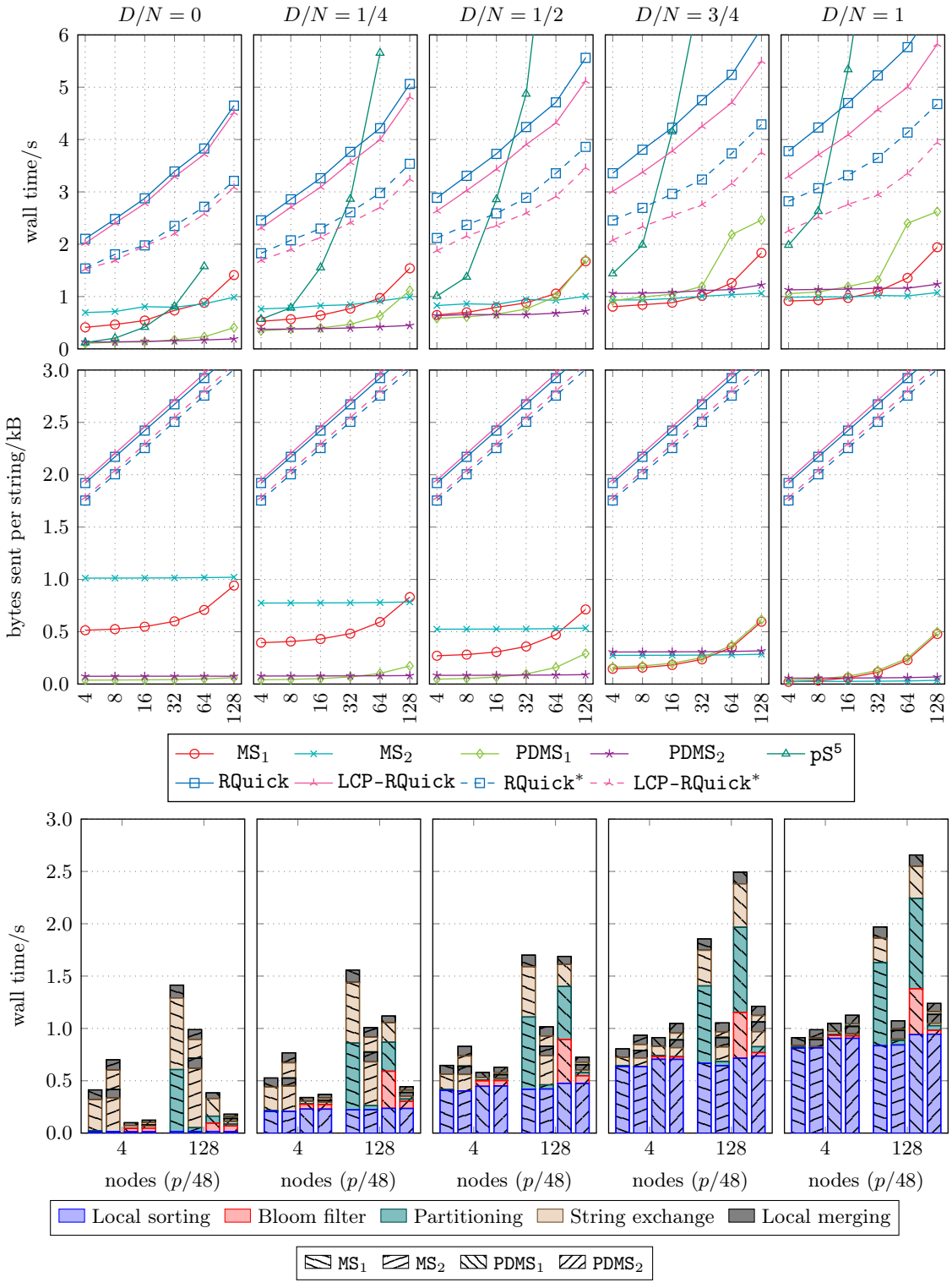

Figure 6.4.: Median overall sorting times (top row), bytes sent per string (middle row), and runtimes per sorting phase (bottom row) for the weak-scaling experiment using DNData inputs with $\ell = 500$, $n/p = 10^5$, and $D/N \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$. Sorting phases are in order of execution starting from the bottom.

crossover point at 32 nodes. The gap seemingly increases for larger D/N ratios, e.g., for $\texttt{MS}_k$ on 128 nodes the speedups are 1.56 and 1.78 for ratios 0 and 1 respectively. This can partially be explained if communication volume is considered. On fewer PEs, two-level variants cause roughly twice the communication volume because string exchange phases dominate. At $D/N = 0$, sending 0.5 kB and 1 kB per string is roughly equivalent to exchanging every string once and twice respectively. Communication increases for single-level variants with the number of PEs as partitioning requires more samples to be sorted. String exchanges send fewer characters for larger D/N ratios due to LCP compression, sample sorting remains roughly constant. A hypercube quicksort implementation with LCP compression could improve this communication inefficiency.

It is notable that for multi-level variants and large D/N ratios, sorting times are almost entirely determined by the time required for local sorting. Here, MSD radix sort causes almost 250 redundant 16-bit sorting steps to scan over the padding characters of each string. With per-phase runtimes of $\texttt{MS}_k$ and $\texttt{PDMS}_k$ next to each other, it also becomes evident that local sorting is slightly slower for $\texttt{PDMS}_k$. This discrepancy may be explained with the datatype used to store strings. For this experiment, $\texttt{PDMS}_k$ uses the less efficient representation with 32 bytes per strings, whereas $\texttt{MS}_k$ only stores string and length in 16 bytes.

A note on an irregularity in our results. For single-level merge sort with prefix approximation, i.e., for $\texttt{PDMS}_1$, a discontinuity in overall sorting times is clearly visible in Figure 6.4 for D/N ratios 3/4 and 1 at 64 nodes. Here, the runtime to sort samples during partitioning fluctuates, seemingly at random, between "normal" values and ones that are three times higher (roughly 330 ms and 1 s). This exact behavior was observed in multiple experiments, with two entirely distinct implementations of `RQuick`, and only at 64 nodes. It only occurs if prefix approximation is run prior to partitioning even though, with the given parameters, distinguishing prefixes are equal to the original strings. Thus, in practice, there should not exist a difference in execution to $\texttt{MS}_1$. We were unable to find the cause of this issue and can only speculatively attribute it to an MPI performance bug.

Next we consider the results for `RQuick`. Overall, for the given input sizes, all variants are slower than basic multi-level merge sort by factors between 2 to 5.4. Runtimes roughly show the expected logarithmic scaling with the number of PEs, as determined by the factor $\log p$ communication volume (note that the x-axis has a logarithmic scale). Viewed in isolation, it is clear that, at least for DNData inputs, the LCP-based optimizations are effective, with LCP variants consistently performing better. With increasing D/N ratio and corresponding LCP values, the gap between `RQuick` and `LCP-RQuick` increases. The reduction in local work results in a speedup of approximately 1.15 at $D/N = 1$. Increases in runtimes with larger D/N ratio are also due to higher local sorting times (cf. phase breakdowns of merge sort). As expected with long strings, the additional communication caused by sending LCP values is negligible—roughly 1.5 % which closely matches the ratio of bytes per LCP value to bytes per string ($8/500 \approx 0.016$). Interestingly, even for $D/N = 0$ there exists a small but measurable improvement in sorting times even though LCP values should be minimal. This could be down to more favorable memory access behavior of procedures using LCP values, as fewer cache misses are incurred for accesses to the character arrays. Comparing runs with 48 to runs with 32 PEs per node, i.e., `RQuick` vs. $\texttt{RQuick}^*$, there is a clear advantage in more evenly distributing active PEs over the available nodes. Avoiding the initial redistribution phase results in a 5 % to 10 % reduction in communication volume and a speedup between

1.3 to 1.4. It may be worth considering system-specific optimizations for the partitioning phase of merge sort implementations to exploit this observation.

Comparing runtimes of merge sort and quicksort to shared-memory $\texttt{pS}^5$ again shows a correlation to D/N ratios. Running on a machine with only 64 cores, $\texttt{pS}^5$ outperforms merge sort without prefix approximation on up to 1536 PEs at $D/N = 0$. These instances are particularly difficult for $\texttt{MS}_k$, since almost the entire character arrays still need to be exchanged, whereas the runtime $\texttt{pS}^5$ is more tightly bounded by the distinguishing prefix. A slight advantage for $\texttt{pS}^5$ still exists at $D/N = 1/4$ but only up to around 4 to 8 nodes. At higher D/N ratios, the shared-memory sorter never outperforms $\texttt{MS}_k$, even on 48 or 96 PEs. If prefix approximation is applied, $\texttt{pS}^5$ only outperforms at least one variant of $\texttt{PDMS}_k$ in three cases, at $D/N = 0$ and $p \leq 192$ (cf. Table A.2). In all other cases, $\texttt{PDMS}_1$ and $\texttt{PDMS}_2$ both yield faster sorting times than $\texttt{pS}^5$ for all tested values of p. In contrast, runtimes of hypercube quicksort are much less competitive with $\texttt{pS}^5$. For inputs with minimal D/N ratio, no variant of `RQuick` manages faster sorting times at any tested value of p. In fact, $\texttt{pS}^5$ requires roughly the same time to sort 153.6 GB using 64 cores, as the fastest variant of hypercube quicksort needs to sort 9.6 GB on 192 PEs. Even extrapolating runtimes of $\texttt{pS}^5$ to the equivalent of 6144 PEs, the shared-memory algorithm is still expected to be competitive. The picture improves for higher D/N ratios and all variants of hypercube quicksort outperform $\texttt{pS}^5$ somewhere between 8 to 64 nodes.

6.3.2. Real-World Strong-Scaling

Our final experiment, to evaluate the performance of multi-level merge sort, uses a strong-scaling setup with two real-world datasets. The first dataset is derived from CommonCrawl, by only using the first 100 GB. The second and third datasets are Wikipedia and WikipediaText. Strings are defined using the lines of each dataset. Local string arrays are obtained by distributing characters over all PEs in equally sized chunks and shifting overlapping lines to the previous PE. Table 6.3 lists key characteristics, such as the average string length N/n, average LCP length L/n, and D/N ratio of each dataset. CommonCrawl contains the largest number of strings, has the shortest average string length, and highest D/N ratio. Being obtained by removing XML metadata, WikipediaText obviously contains fewer strings and characters than Wikipedia with higher average string length and lower D/N ratio. The longest string (i.e., line) in each dataset is over a megabyte in size. We evaluated sorting times on 4 to 256 nodes for single- and two-level $\texttt{MS}_k$ and $\texttt{PDMS}_k$ as well as $\texttt{PDMS}_2^{\triangledown}$ which uses the single-level Bloom filter. At 256 nodes, the 100 GB of CommonCrawl only account for roughly 8.1 MB of characters on each PE.

	$n/10^9$	$N/10^9$	N/n	L/n	D/N	$\hat{\ell}/10^6$
CommonCrawl	2.13	100.0	46.98	31.27	.726	1.04
Wikipedia	1.42	97.7	68.59	25.82	.415	2.07
WikipediaText	0.97	81.7	84.21	25.27	.336	2.07

Table 6.3.: Characteristics of real-world datasets used in the strong-scaling experiment.

Results of the experiment are shown in Figure 6.5 and exact sorting times are listed in Table A.3 in the Appendix. As ever, overall sorting times are provided by the topmost plots and sorting times per phase are in the bottom row for 8, 32, and 128 nodes. The two middle rows contain speedups in overall sorting times relative to $\texttt{PDMS}_1$—values greater than 1 mean shorter runtimes.

We first consider the results for CommonCrawl. Note that $\texttt{PDMS}_2$ failed to sort the input on 4 node likely due to memory limitations caused by imbalances introduced during the first string exchange. With 100 GB characters in total, each PE holds roughly 500 MB which is already near the limit for 2 GB memory per PE. Comparing single- and two-level $\texttt{MS}_k$ reveals no clear advantage for either variant. String exchanges and local merging dominate runtime in both cases which means that reduced partitioning times do not yield substantial speedups even at 128 nodes. For $\texttt{PDMS}_k$ the difference is more pronounced with two levels clearly outperforming a single level at a speedup of roughly 1.9. The improvement can mostly be attributed to the prefix approximation phase which is made clear by looking at runtimes per sorting phase and also by the overall sorting times for $\texttt{PDMS}_2^{\triangledown}$. All three variants exhibit disproportionately high local merging times during the second sorting level. This can be partially explained by the make-up of CommonCrawl with many, relatively short strings, and long common prefixes on average (cf. Table 6.3). However, the majority of runtime (approximately 75 %) is actually spent undoing LCP compression after merging. Further to explaining these runtimes, it should be noted that there exists a significant character imbalance after sorting for this dataset. For two-level sorting, the maximum number of characters on any PE is more than 2 GB. Compared to the 8 MB per PE for an even distribution, this goes a long way in explaining the high variance in merging times between PEs. While different sampling and group assignment techniques could improve character balance, the CommonCrawl dataset inherently contains many duplicated strings—such as standard legal disclaimers—which are always assigned to a single PE and cannot be shortened with prefix approximation.

Changing focus to Wikipedia and WikipediaText we see a wider spread in runtimes between variants. Sorting failed to complete successfully for $\texttt{MS}_2$ on 4 nodes. Clearly, prefix approximation is more effective for both datasets due to lower D/N ratios. As before, $\texttt{PDMS}_2$ is slightly slower than $\texttt{PDMS}_1$ on fewer PEs (slowdown less than 1.25) but is significantly faster otherwise (speedup up to 3.9). The poor scaling behavior of the single-level Bloom filter is even more apparent, with a clear increase in overall runtimes on more than 64 nodes. As a result, even variants of $\texttt{MS}_k$ outperform single-level $\texttt{PDMS}_1$ at 128 nodes. String exchange and local merging times are also much reduced due to the less repetitive nature of the datasets. At 128 nodes, sorting times are split roughly equally between initialization (local sorting and Bloom filter) and distributed sorting phases.

6.3.3. Summary

The preceding experiments show that multi-level MS and PDMS clearly outperform single-level variants for sufficiently many PEs, and, on at least medium sized inputs, exhibit desirable scaling behavior. While the gap is more pronounced and occurs at fewer PEs for smaller inputs, it also exists at the largest tested input size, which is also near the limit of what is possible on the given system. In our experiments, three-level variants did not yield clearly better runtimes compared to two levels and generally come with significant overhead.

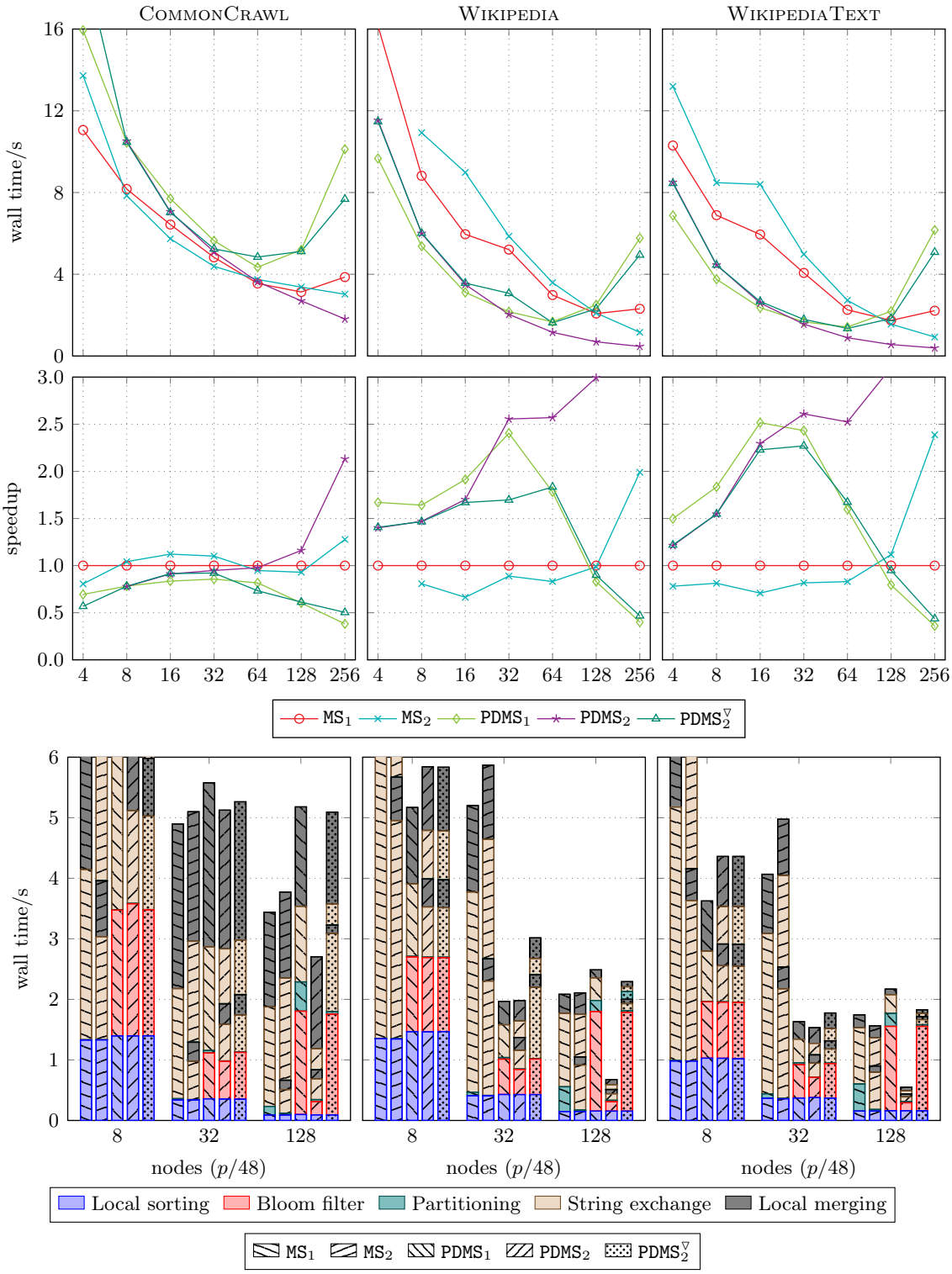

Figure 6.5.: Median overall sorting times (top row), speedup in overall sorting time relative to $\texttt{MS}_1$ (middle row), and runtimes per sorting phase (bottom row) for the strong-scaling experiment using real-world inputs. Sorting phases are in order of execution starting from the bottom.

Notably, partitioning still contributes disproportionately for the smallest tested inputs on more than 256 nodes. On generated DNData inputs, local sorting and merging usually dominate overall sorting times of multi-level variants for large values of p. For real-world data sets the results are less clear, though multi-level variants still provide an improvement and especially the multi-level Bloom filter is highly effective. As expected, hypercube quicksort is up to multiple times slower than merge sort variants, though LCP optimizations do provide a small but noticeable improvement.

6.4. Space-Efficient String Sorting

This section comprises an experimental evaluation of space-efficient merge sort. The main goal is to determine whether the algorithm scales well with regard to input size per PE and to quantify the influence on overall sorting times of the number of quantiles. As before, we start, in Section 6.4.1, with experiments using DNDataSE instances with fixed D/N ratio. Section 6.4.2 focuses on difference cover samples of real-word datasets. The limiting factor for the experiments in this section are often overall runtimes as they can be expected to grow at least linearly with the number of characters per PE.

6.4.1. Fixed D/N Ratio Weak-Scaling

The first experiment with space-efficient sorters uses DNDataSE inputs to vary both the D/N ratio and the number of strings per PE. Varying the D/N ratio can be used to demonstrate the efficacy of using prefix approximation to reduce the number of required quantiles and the total work during sorting. Using instances with more strings per PE, i.e., increasing the n/p ratio, highlights the capability of space-efficient sorting to handle higher data volumes. The experiment uses inputs consisting of 2, 4, and 8 million strings per PE, 1000 characters per strings, and with D/N ratios 1/8, 1/4, and 1/2. Recall that DNRatioSE does not permit D/N ratios greater than 1/2 and that compression and D/N ratios are proportional. All input sizes would either exceed or entirely fill the available memory of roughly 2 GB per PE in their uncompressed form. For the largest inputs, each PE holds roughly 8 GB of uncompressed data with distinguishing prefixes of roughly 4 GB. Figure 6.6 shows results for single- and two-level $\texttt{SEMS}_k$ and $\texttt{PDSEMS}_k$ on 2 to 64 nodes. Overall sorting times are listed in Table A.4 in the Appendix. Experiments were not performed with more PEs, to limit expenditure of computing budget due to high runtimes and the large number of variants ($3 \times 3 \times 4$).

The results broadly confirm the expected behavior for the given input parameters. Space-efficient sorting appears to be effective, since the tested implementations are able to process inputs that would otherwise exceed the available memory. As expected, sorting times increase at least proportionally to input size which places clear limits on the inputs we can sort with reasonable runtimes. The longest sorting time in this experiment is almost 170 s to sort an input which globally contains approximately 24.6 TB. Table 6.4 lists the number of quantiles used during sorting for every combination of D/N and n/p ratio, with and without prefix approximation. Clearly, $\texttt{SEMS}_k$ always yields an equal number of quantiles for the given inputs—exactly $(n/p) \times 10^{-5}$ in this instance. For $\texttt{PDSEMS}_k$ the number could vary, at least in theory, though this is did not occur and is exceedingly unlikely given the expected

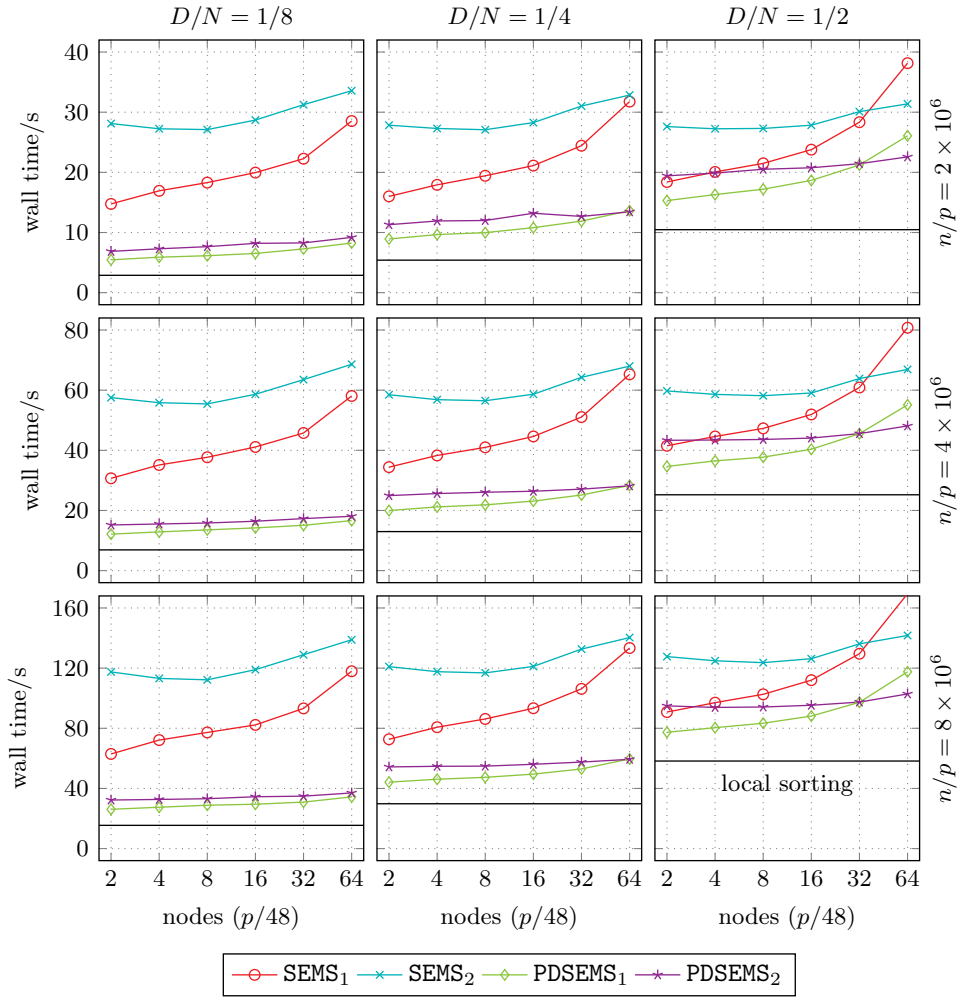

Figure 6.6.: Median overall sorting times for the weak-scaling experiment using DNDataSE inputs with $\ell = 1000$, $D/N \in \{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}\}$, and $n/p \in \{2, 4, 8\} \times 10^6$. Sorting requests quantiles of size 100×2^{20} characters. Horizontal lines indicate average local sorting times.

distinguishing prefix size for DNDataSE and the probability of false positives with the used hash function.

We first compare results for different input parameters before focusing on individual variants of the algorithm. Per doubling of the number of strings per PE, the sorting times for each algorithm and at every D/N ratio increase by factors between 1.97 to 2.26. Interestingly, slowdown factors are consistently higher for variants with prefix approximation (median slowdown 2.08 compared to 2.14) even though the lengths of computed distinguishing prefixes barely vary between n/p ratios. For increasing D/N ratios, average lengths are 172, 342, and 667 characters. Observe that prefix doubling overestimates the actual size of distinguishing prefixes. Comparing relative runtimes for varying D/N ratios reveals a somewhat different picture. As can be expected, performance of SEMS$_k$ is generally less affected by increases in D/N ratio. SEMS$_1$ yields average slowdowns 1.15 (maximum 1.27) per doubling of D/N ratio and for every n/p ratio, while SEMS$_2$ only exhibits slowdowns of 1.01 on average (maximum

	$D/N = 1/8$		1/4		1/2	
n/p	$\texttt{SEMS}_k$	$\texttt{PDSEMS}_k$	$\texttt{SEMS}_k$	$\texttt{PDSEMS}_k$	$\texttt{SEMS}_k$	$\texttt{PDSEMS}_k$
2×10^6	20	4	20	7	20	14
4×10^6	40	7	40	14	40	27
8×10^6	80	14	80	28	80	54

Table 6.4.: Number of quantiles used during sorting for the weak-scaling experiment with DNDataSE inputs. Values are identical for every run and all values of p and k.

1.06). The difference is at least partially due to the single-level variant being proportionally more affected by increased local work of `RQuick`. Slowdowns are between 1.61 and 1.97 (average 1.72) for $\texttt{PDSEMS}_1$ and slightly lower for $\texttt{PDSEMS}_2$ (1.46–1.75, average 1.65).

Shifting focus to the results for individual algorithms reveals the recurring trade-off between different variants of merge sort. First, note that local sorting contributes significantly to overall sorting times, as can be expected given the size of distinguishing prefixes involved. For $\texttt{PDSEMS}_k$, local sorting local makes up more than 50 % of overall sorting in most cases and never less than 40 %. Generally, variants with prefix approximation clearly outperform those without. The relative runtimes clearly correlate to D/N ratio with much reduced speedups of $\texttt{PDSEMS}_k$ over $\texttt{SEMS}_k$ for larger ratios, as average approximated prefixes comprise a greater proportion of each string. This observation is reinforced if we again consider Table 6.4 and assume that the expected cost of sorting a single quantile for a fixed number of PEs is roughly constant. With higher D/N ratio, the relative numbers of quantiles—and thus the expected sorting time—converge. There also exists a relationship between the D/N ratio and relative runtimes of single- and two-level variants roughly opposite to the correlation with prefix approximation. For instances with small ratios less than 1/2, multi-level variants never provide a benefit compared to the base variants for the tested PE numbers. With $D/N = 1/2$ at 64 nodes, we observe speedups of roughly 1.22 for $\texttt{SEMS}_2$ and 1.15 for $\texttt{PDSEMS}_2$. For $\texttt{SEMS}_k$, the trend is already apparent at 1/4. The observed behavior is likely due to LCP compression being less effective at lower D/N ratios.

We also performed an experiment to gauge the influence of configured quantile sizes on overall sorting times. The setup is mostly identical to the preceding experiment—DNDataSE inputs with strings of length 1000, $n/p = 4 \times 10^6$ string per PE, and D/N ratios 1/8, 1/4, and 1/2. Runtimes were only measured for single-level $\texttt{PDSEMS}_1$ to provide a reasonable range for the number of quantiles, but general trends should translate to other variants too. Runs for the preceding experiment were all performed by requesting quantiles of size 100 MiB, which we increased and decreased by factors 2 and 4—yielding sizes 25, 50, 100, 200, and 400 MiB. Sorting quantiles significantly larger than 400 MiB is not feasible with the available memory. Results for the experiment are shown in Figure 6.7 for 1 to 64 nodes. Runtimes using quantiles less than 100 MiB are generally worse for all D/N ratios. While there are small speedups at low PE counts, the overhead of additional partitioning phases clearly shows for higher values of p. At $D/N = 1/2$ the algorithm requires 54 and 107 quantiles, resulting in slowdowns up to 1.24 and 1.71. Results are not as clear for quantiles larger than 100 MiB. While we observe noticeable speedups for larger PE counts (up to 1.14), at lower

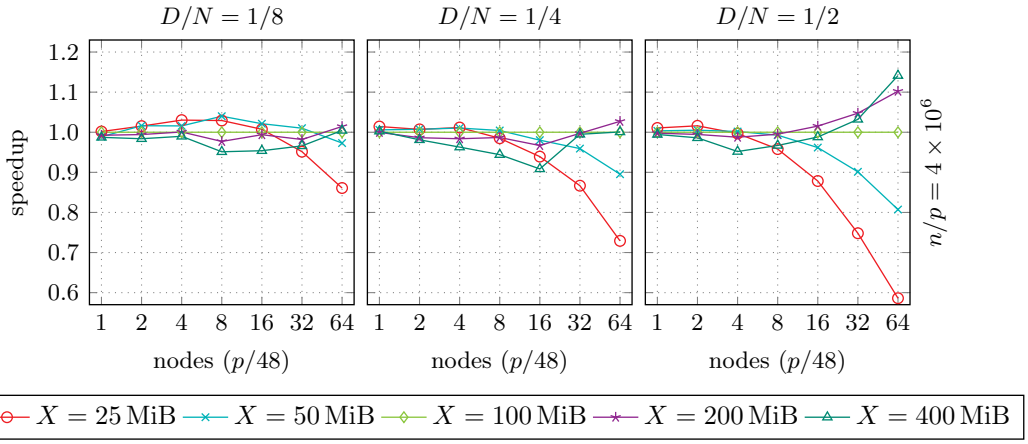

Figure 6.7.: Speedup in overall sorting time for the weak-scaling experiment with PDSEMS$_1$ requesting quantiles of local size $\{\frac{1}{4}, \frac{1}{2}, 1, 2, 4\} \times 2^{22}$ bytes on DNDATASE inputs with $\ell = 1000$, $D/N \in \{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}\}$, and $n/p = 4 \times 10^6$.

values the opposite is true with slowdowns up to 1.1. For the following experiment with real-world datasets, we resolve to use quantiles of 100 MiB as a safe choice. Larger values may often be possible, but can become impractical due to the non-uniform nature of input distribution, where local quantile sizes vary significantly and are often much larger than the configured value.

6.4.2. Difference Cover Weak-Scaling

With the final experiment, we aim to evaluate the performance of space-efficient sorting on realistic inputs. As a secondary goal, the experiment should also try to approach the limit of input sizes that can reasonably be sorted on the given system. To this end, we use difference covers samples of real-word datasets for generated difference covers modulo 512, 1024, 2048, 4096, and 8192 (cf. Section 6.2.3). The experiment uses a weak-scaling setup, where a text, consisting of a prefix of each dataset, is distributed over PEs in fixed-size chunks and a difference cover is constructed as previously described. Inputs are limited to 100 MB and 200 MB chunks per PE on up to 256 nodes for the smaller size and 128 nodes for the larger. These parameters are constrained by several factors: Most immediate is the problem of obtaining and storing sufficiently large and suitable datasets. With 256 nodes and 100 MB per PE, the experiment requires a dataset consisting of almost 1.23 TB and the same for 128 nodes at 200 MB. We use COMMONCRAWL and WIKIPEDIAFULL as introduced in Section 6.2.3. Note that 2.5 TB were obtained for both datasets which could facilitate a further doubling of p or the size of text chunks, but we elected not to perform corresponding experiments due to unreasonably high running times. Nevertheless, with the given number of characters per PE, each uncompressed local character array contains 11.2 GB and 22.4 GB for the largest difference cover. At 256 and 128 nodes respectively, this yields global uncompressed character arrays of size 137.6 TB. The experiment only includes multi-level variants, since runs are performed beyond 64 nodes, which makes single-level algorithms impractical (cf. Figure 6.6). Measurements of variants without prefix approximation are limited to 100 MB and up to 128 nodes. All runs use a combination of built-in all-to-all

exchanges and direct messaging, where the former is used if possible and the latter serves as a fallback if the total number of elements sent to or received from any PE exceeds the numeric limits of a 32-bit C `int`. Using only native all-to-all exchanges, we would regularly exceed this limit and encounter numeric overflows as a result. Measurement with CommonCrawl were preformed without LCP compression, because it proved ineffective; runs with WikipediaFull do use LCP compression. We elaborate on the reason for this difference later in this section. Runtimes were also measured for pS^5, to gauge the efficiency of distributed- compared to shared-memory sorting for texts up to $64 \times 48 \times 100\,\text{MB}$ (roughly 307 GB)—equivalent to 64 and 32 nodes respectively.

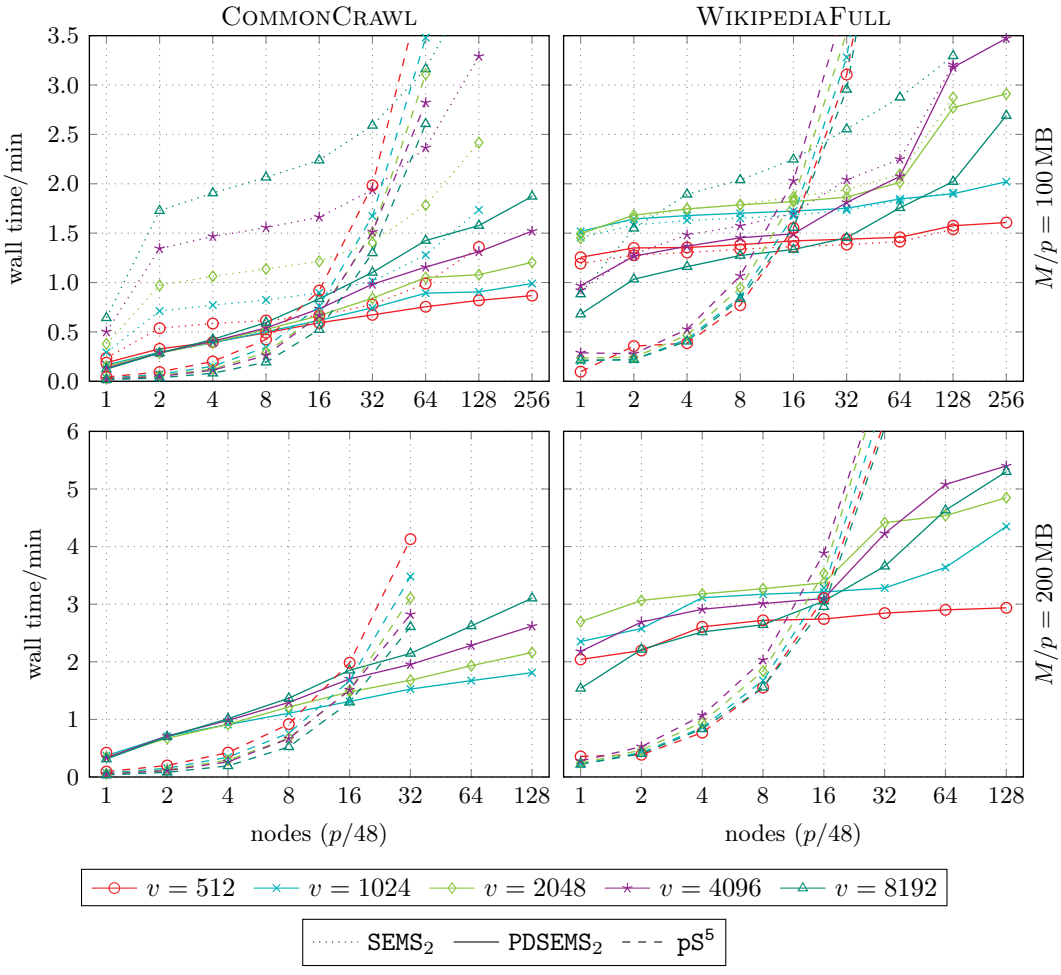

Figure 6.8.: Median overall sorting times for the weak-scaling experiment using difference cover samples modulo $v \in \{2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}\}$ of real-world datasets with chunks of $M/p = 100\,\text{MB}$ per PE.

Figure 6.8 shows results for the experiment with CommonCrawl on the left, WikipediaFull on the right, 100 MB at the top, and 200 MB below. Overall sorting times are listed in Table A.5 in the Appendix. It is immediately obvious that the two datasets yield remarkably different results. Sorting times for CommonCrawl broadly increase with the number of PEs and the size of difference covers. In comparison, absolute runtimes for WikipediaFull are significantly higher in general and exhibit much more unpredictable scaling behavior.

$M/100\,\text{MB}$		CommonCrawl					WikipediaFull				
/48	/96	512	1024	2048	4096	8192	512	1024	2048	4096	8192
1	—	.189	.141	.105	.077	.055	.983	.962	.918	.824	.657
2	1	.237	.178	.134	.100	.074	.982	.961	.914	.814	.642
4	2	.304	.231	.173	.128	.093	.978	.951	.895	.784	.603
8	4	.382	.298	.226	.166	.118	.979	.955	.901	.795	.620
16	8	.454	.372	.291	.216	.154	.975	.945	.884	.767	.587
32	16	.504	.431	.353	.272	.199	.971	.938	.872	.751	.570
64	32	.512	.439	.361	.281	.208	.973	.942	.878	.761	.583
128	64	.518	.445	.368	.288	.216	.972	.939	.874	.755	.575
256	128	.522	.449	.373	.296	.224	.972	.940	.876	.757	.579

Table 6.5.: List of D/N ratios for difference cover samples modulo $v \in \{2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}\}$ of real-word datasets for M equal to multiples of $48 \times 100\,\text{MB}$ and $48 \times 200\,\text{MB}$.

In an attempt to explain these discrepancies, we computed D/N ratios of difference cover samples at every used text size. Table 6.5 lists the resulting ratios for CommonCrawl and WikipediaFull. Values in the two leftmost columns correspond to the number of PEs at 100 MB and 200 MB chunks. Both datasets can, in some sense, be classified as highly repetitive based on the measured ratios.

We first consider results for CommonCrawl in more detail. Note that runs with difference cover sample modulo 512 for 200 MB chunks on more than one node were consistently terminated by the job scheduler. While no specific reason was forthcoming, we generally observed this behavior in cases of memory exhaustion. Also note the "bump" in runtimes for $\texttt{SEMS}_2$ between one and two nodes, which is due to the algorithm using only one sorting level at 48 PEs and two levels at 96 PEs (cf. Section 6.2.2). As already pointed out, overall sorting times clearly correlate to the difference cover size, with larger difference covers resulting in higher runtimes. Looking only at the results for $\texttt{SEMS}_2$, each doubling in difference cover size yields average slowdowns between 1.31 to 1.38. This lines up closely with the increases in compression ratio between difference covers, e.g., going from $v = 4096$ with 82 members to $v = 8192$ with 112 members is a factor 1.37. With average slowdowns between 1.1 and 1.17, the relation is less pronounced for $\texttt{PDMS}_2$, since sorting times are also dependent on the length of approximate prefixes. Interestingly, results for $\texttt{pS}^5$ broadly exhibit the opposite correlation between the size of difference cover and sorting times. Being a shared-memory algorithm, $\texttt{pS}^5$ has no need to exchange strings and therefore sorting times are much less determined by the total (character) input size. Instead, distinguishing prefix sizes are of much greater importance, and thus longer strings from samples for larger difference covers are less detrimental to performance, due to their lower D/N ratios.

For the CommonCrawl dataset, prefix approximation is generally highly effective. This can be sufficiently explained, considering the D/N ratios listed in Table 6.5. As one would expect from shorter strings, samples for smaller difference covers have higher D/N ratios, with only $v = 512$ exceeding $1/2$ while $v = 8192$ is never more than $1/4$ on the tested input sizes. For all difference covers, D/N ratios steadily increase as more chunks of the dataset are used. This is primarily due to the semi-random make-up of CommonCrawl,

where duplicate strings can be expected to be spread out evenly over the entire dataset. As previously mentioned, duplicates often consists of legal disclaimers, policies, and similar standard passages, where little local coherence exists. From higher D/N ratios follow longer approximated prefixes, which directly leads to more quantiles needing to be sorted. For 100 MB chunks, the average number of quantiles roughly doubles ($15 \rightsquigarrow 34$) for $v = 512$ and more than triples ($22 \rightsquigarrow 72$) for $v = 8192$ going from one node to 256 nodes.

Focusing instead on results for WikipediaFull, none of the preceding observations seem to apply. There is no clear correlation between difference cover size and sorting times, nor a steady increase for larger values of p. This can again be explained with the D/N ratios from Table 6.5. Overall, D/N ratios are much—in fact strictly—greater than for CommonCrawl and never fall below 0.57. For $v = 512$, distinguishing prefixes are virtually equal to the original strings and hence, sorting times of $\texttt{SEMS}_2$ and $\texttt{PDSEMS}_2$ barely differ. The same is true, to a similar extent, for the difference covers modulo 1024, 2048, and 4096. Only $v = 8192$ shows a significant speedup for $\texttt{PDSEMS}_2$ over $\texttt{SEMS}_2$. Running entirely contrary to CommonCrawl, D/N ratios actually slightly decrease for samples of larger chunks. This is again an inherent characteristic of the dataset. Whereas CommonCrawl consists of a mostly random sequence of web pages, WikipediaFull has high local coherence where the entire edit history for a page is listed consecutively. Interestingly, this also means that LCP compression is much more effective for WikipediaFull because common prefixes are likely to be located on the same PE before the first string exchange. Regardless, using larger chunks of the dataset is unlikely to significantly increase the number of duplicates or the D/N ratio.

A note on the conspicuous jump of sorting times at 128 nodes for difference covers of WikipediaFull modulo 2048 and 4096 and chunk size 100 MB. Here, runtimes increase by 45.4 s (37.6 %) and 66.3 s (53.2 %) respectively for a single doubling of p. Further investigation of the results revealed that this is closely matched by increases in maximum local sorting times (40.8 s and 57.1 s). We can only speculate that a section of WikipediaFull, which is first part of the input here, is especially repetitive. There also exist significant increases in local sorting times going from 16 to 32 and 32 to 64 nodes whereas they are otherwise fairly stable. In general, D/N ratios no longer decrease consistently at 32 nodes and above (cf. Table 6.5).

6.4.3. Summary

Overall, our experiments demonstrate that space-efficient string sorting is effective in increasing the scalability of distributed string sorting, by allowing for a higher number of characters per PE. We have shown that it is possible to handle inputs with up to 22.4 GB uncompressed characters on a system with only 2 GB memory. This is more than a twentyfold improvement compared to a normal distributed sorting algorithm, where a realistic implementation, to allow for send and receive buffers, can only be expected to be capable of sorting inputs up to 1 GB with the available memory. Other than proportionally higher sorting times, there is no reason to expect that further increases in local input size should not be achievable. It is also clear that the number of quantiles plays a crucial role in determining sorting performance. Aside from aiming to configure quantiles as large as possible, this also means that prefix approximation can be especially effective at reducing sorting times.

7. Conclusion

In this thesis, we have presented two new techniques to increase the scalability of string sorting algorithms in distributed-memory parallel systems. By applying a multi-level approach [6] to existing multi-way string merge sort algorithms MS and PDMS [11], we have been able to reduce the minimum number of characters per PE at which both algorithms are efficient. This makes it feasible to sort a given input using more PEs without incurring disproportionate costs for partitioning. At least for multi-level MS, we were able to formally quantify this improvement under a number of simplifying assumptions. In order to provide bounds on workload imbalance in the multi-level context, we have also slightly generalized existing regular sampling techniques, as well as adapting group assignment methods to allow character-based worst-case guarantees. We also proposed a multi-level Bloom filter as an approach to achieve sublinear latency for prefix approximation. In our experiments we were able to demonstrate that the multi-level algorithms generally outperform their single-level counterparts for large values of p. On up to 24 576 PEs, we observed speedups up to five over the closest existing competitors. While multi-level merge sort proved especially effective for synthetic instances in a weak-scaling setup, we nonetheless observed an improvement for real-word datasets in a strong-scaling experiment.

The second primary contribution of this thesis is to introduce the concept of space-efficient string sorting for distributed systems. We proposed an algorithmic framework, which accepts a deduplicated representation of string arrays, partitions inputs into smaller parts of limited size so-called quantiles, and only works on a single quantile at a time. By not having to materialize the entire input at once, it is possible to derive sorted (inverse) permutations for string arrays that would otherwise not fit into the available memory. The result no longer fulfills the strictest definition of a sorting algorithm, yet still solves a closely related problem. With the aforementioned distributed multi-level merge sort algorithms to sort quantiles, and by applying some additional optimizations, as well as optional prefix approximation, we obtained a family of algorithms which we collective refer to as space-efficient merge sort (SEMS). In our experiments, an implementation of SEMS proved effective at sorting inputs of uncompressed size up to eleven times greater than the actually available memory per PE. We furthermore proposed an application for space-efficient sorting as a subroutine to suffix array construction with DCX. To demonstrate the viability of SEMS in this context, we performed experiments using large difference cover samples for real-world datasets. Space-efficient sorting seems particularly well-suited for such instances, given their high level of overlap between strings and corresponding compression ratio.

As a tertiary result, we also reevaluated hypercube quicksort for small string arrays, by applying LCP-based optimizations to local merging and to the process of locating the splitter. The resulting variant—LCP-RQuick—cannot overcome the basic communication inefficiency of the algorithm, but did provide a slight improvement in practice for inputs with long common prefixes.

7.1. Future Work

Because distributed string sorting algorithms have only received limited attention so far, there exists significant room for future work on the topic. First of all, the multi-level string merge sort algorithms developed in this thesis could benefit from a thorough worst-case analysis to simplify some of the more convoluted runtime bounds. In the context of PDMS, it is still an open question whether our multi-level Bloom filter actually manages to achieve the conjectured runtime, and whether it, in fact, provides any benefit over a solution using regular hypercube or grid all-to-all exchanges. Originally intended as part of this thesis, random (character-based) sampling techniques still require theoretical analysis to determine whether they could be used as part of the partitioning algorithm to obtain improved runtime guarantees in expectation. While not included in the experimental evaluation, we noticed that multi-level variants performed especially poorly using character-based regular sampling on real-world data sets. In our assessment, this is due to a disproportionate number of strings being sent to the same PE by the partitioning algorithm. Incurring string imbalances in the order of multiple magnitudes in exchange for balanced character arrays seems to be an unfavorable trade-off, since it introduces high variance in sorting times between PEs on subsequent sorting levels. Future work could attempt to reconcile this gap between theory and practice, by acknowledging the importance of the number of strings in determining performance.

The distribution of strings relative to their lexicographical order poses a significant hindrance for the analysis of space-efficient string sorting. Skewed input distributions directly degrade the algorithm's performance, increasing latency and the cost for partitioning by up to an additional factor p, and can cause the algorithm to fail outright if too many strings are concentrated on a single PE. In this thesis we made strong assumption about the uniformity of input distributions in order to apply a Chernoff bound and thereby show that the algorithm succeeds with high probability for a very limited range of inputs and parameters. We also have to contend with the additional imbalance introduced by partitioning via regular sampling. A more extensive analysis could seek to better classify and describe inputs that can be processed using space-efficient sorting with a particular set of parameters. For practical applications, the implementation of SEMS could be improved with special handling for quantiles over a certain threshold, for example, by recursive partitioning or by redistributing strings from overpopulated PEs to those with free capacity.

Building on the foundations laid in this thesis, a full-fledged distributed suffix sorter based on DCX could be implemented in the future and would require additional effort in several aspects: First, the ranks computed during sorting need to be reordered to create the supertext required for recursive sorting. Next, we need to consider the aforementioned recursive sorting step, which requires at least a different variant of SEMS or, more likely, an altogether distinct algorithm. Finally, the merging step—to sort *sample* and *nonsample* suffixes—is more involved in the general case of $v \neq 3$ and requires work to be implemented efficiently.

In Section 1 we already discussed several improvements that could be applied to LCP-RQuick. From a theoretical standpoint, bounding the local work for merging and using character-based median approximation could yield improved runtime guarantees. More practically speaking, prefix approximation and LCP compression could prove useful in reducing the algorithm's communication inefficiency.

Bibliography

[1] Lars Arge et al. "On Sorting Strings in External Memory (Extended Abstract)". In: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. ACM, May 1997, pages 540–548. DOI: 10.1145/258533.258647.

[2] Michael Axtmann. "Robust Scalable Sorting". PhD thesis. Karlsruhe Institute of Technology (KIT), May 2021. DOI: 10.5445/IR/1000136621.

[3] Michael Axtmann and Peter Sanders. "Robust Massively Parallel Sorting". In: *2017 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. Proceedings. SIAM, January 2017, pages 83–97. DOI: 10.1137/1.9781611974768.7.

[4] Michael Axtmann, Armin Wiebigke, and Peter Sanders. "Lightweight MPI Communicators with Applications to Perfectly Balanced Quicksort". In: *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2018, pages 254–265. DOI: 10.1109/IPDPS.2018.00035.

[5] Michael Axtmann et al. *Engineering In-Place (Shared-Memory) Sorting Algorithms*. Computing Research Repository (CoRR). September 2020. arXiv: 2009.13569.

[6] Michael Axtmann et al. "Practical Massively Parallel Sorting". In: *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, June 2015, pages 13–23. DOI: 10.1145/2755573.2755595.

[7] 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*. SIAM, January 1997, pages 360–369. ISBN: 0-89871-390-0.

[8] Timo Bingmann. "Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools". PhD thesis. Karlsruhe Institute of Technology (KIT), July 2018. DOI: 10.5445/IR/1000085031.

[9] Timo Bingmann. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers*. 2018.

[10] Timo Bingmann, Andreas Eberle, and Peter Sanders. "Engineering Parallel String Sorting". In: *Algorithmica* 77.1 (January 2017), pages 235–286. DOI: 10.1007/s00453-015-0071-1.

[11] Timo Bingmann, Peter Sanders, and Matthias Schimek. "Communication-Efficient String Sorting". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2020, pages 137–147. DOI: 10.1109/IPDPS47924.2020.00024.

[12] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (July 1970), pages 422–426. DOI: 10.1145/362686.362692.

[13] Charles J. Colbourn and Alan C. H. Ling. “Quorums from Difference Covers”. In: *Information Processing Letters* 75.1 (2000), pages 9–12. DOI: 10.1016/S0020-0190(00)00080-6.

[14] Andrew Davidson et al. “Efficient Parallel Merge Sort for Fixed and Variable Length Keys”. In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pages 1–9. DOI: 10.1109/InPar.2012.6339592.

[15] Jonas Ellert, Johannes Fischer, and Nodari Sitchinava. “LCP-Aware Parallel String Sorting”. In: *26th International European Conference on Parallel and Distributed Computing (Euro-Par)*. LNCS. Springer, 2020, pages 329–342. DOI: 10.1007/978-3-030-57675-2_21.

[16] Rolf Fagerberg, Anna Pagh, and Rasmus Pagh. “External String Sorting: Faster and Cache-Oblivious”. In: *23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS)*. LNCS. Springer, 2006, pages 68–79. DOI: 10.1007/11672142_4.

[17] Johannes Fischer and Volker Heun. “Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE”. In: *Combinatorial Pattern Matching*. LNCS. Springer, 2006. DOI: 10.1007/11780441_5.

[18] Johannes Fischer and Florian Kurpicz. “Lightweight Distributed Suffix Array Construction”. In: *2019 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. SIAM, January 2019, pages 27–38. DOI: 10.1137/1.9781611975499.3.

[19] Pierre Fraigniaud and Emmanuel Lazard. “Methods and Problems of Communication in Usual Networks”. In: *Discrete Applied Mathematics* 53.1 (September 1994), pages 79–133. DOI: 10.1016/0166-218X(94)90180-5.

[20] Alexandros V. Gerbessiotis and Leslie G. Valiant. “Direct Bulk-Synchronous Parallel Algorithms”. In: *Journal of Parallel and Distributed Computing* 22.2 (August 1994), pages 251–267. DOI: 10.1006/jpdc.1994.1085.

[21] Torben Hagerup and Christine Rüb. “A Guided Tour of Chernoff Bounds”. In: *Information Processing Letters* 33.6 (February 1990), pages 305–308. DOI: 10.1016/0020-0190(90)90214-I.

[22] Torben Hagerup and Christine Rüb. “Optimal Merging and Sorting on the EREW PRAM”. In: *Information Processing Letters* 33.4 (December 1989), pages 181–185. DOI: 10.1016/0020-0190(89)90138-5.

[23] *Hardware of SuperMUC-NG Phase 1*. Leibniz-Rechenzentrum (LRZ). URL: https://doku.lrz.de/hardware-of-supermuc-ng-phase-1-11482553.html (visited on 01/19/2024).

[24] “KaMPIng: Karlsruhe MPI next Generation”. Unpublished software. Institute of Theoretical Informatics, Algorithm Engineering – Karlsruhe Institute of Technology, 2024.

[25] Juha Kärkkäinen and Tommi Rantala. “Engineering Radix Sort for Strings”. In: *String Processing and Information Retrieval (SPIRE)*. LNCS. Springer, 2009, pages 3–14. DOI: 10.1007/978-3-540-89097-3_3.

[26] Juha Kärkkäinen and Peter Sanders. “Simple Linear Work Suffix Array Construction”. In: *Automata, Languages and Programming*. LNCS. Springer, 2003, pages 943–955. DOI: 10.1007/3-540-45061-0_73.

[27] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. "Linear Work Suffix Array Construction". In: *Journal of the ACM* 53.6 (2006), pages 918–936. DOI: 10.1145/1217856.1217858.

[28] Wai-Shing Luk and Tien-Tsin Wong. "Two New Quorum Based Algorithms for Distributed Mutual Exclusion". In: *Proceedings of 17th International Conference on Distributed Computing Systems*. 1997, pages 100–106. DOI: 10.1109/ICDCS.1997.597862.

[29] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. "Engineering Radix Sort". In: *Computing Systems* 6 (1993), pages 5–27.

[30] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. manual. November 2023. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[31] Alistair Moffat and Andrew Turpin. *Compression and Coding Algorithms*. Springer US, 2002. DOI: 10.1007/978-1-4615-0935-6.

[32] Peter Sanders and Thomas Hansch. "Efficient Massively Parallel Quicksort". In: *Solving Irregularly Structured Problems in Parallel*. Volume 1253. LNCS. Springer, 1997, pages 13–24. DOI: 10.1007/3-540-63138-0_2.

[33] Peter Sanders, Sebastian Schlag, and Ingo Müller. "Communication Efficient Algorithms for Fundamental Big Data Problems". In: *2013 IEEE International Conference on Big Data*. October 2013, pages 15–23. DOI: 10.1109/BigData.2013.6691549.

[34] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. "Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan". In: *14th European PVM/MPI Users Group Meeting* 35.12 (December 2009), pages 581–594. DOI: 10.1016/j.parco.2009.09.001.

[35] Peter Sanders et al. *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer International Publishing, 2019. DOI: 10.1007/978-3-030-25209-0.

[36] Matthias Schimek. "Distributed String Sorting Algorithms". Master's thesis. Karlsruhe Institute of Technology (KIT), July 2019. DOI: 10.5445/IR/1000098432.

[37] Hanmao Shi and Jonathan Schaeffer. "Parallel Sorting by Regular Sampling". In: *Journal of Parallel and Distributed Computing* 14.4 (April 1992), pages 361–372. DOI: 10.1016/0743-7315(92)90075-X.

A. Appendix

A.1. Running Times – Multi-Level Merge Sort

	wall time/s								
$p =$	96	192	384	768	1536	3072	6144	12 288	24 576
	$n/p = 10^4$								
MS$_1$	0.040	0.051	0.067	0.103	0.220	0.455	0.911	1.170	1.988
MS$_2$	0.056	0.057	0.059	0.067	0.070	0.079	0.127	0.175	0.387
MS$_3$	0.056	0.090	0.091	0.092	0.095	0.107	0.132	0.165	0.269
PDMS$_1$	0.040	0.042	0.055	0.104	0.203	0.434	1.968	3.217	7.086
PDMS$_2$	**0.037**	**0.037**	**0.038**	**0.049**	**0.050**	**0.059**	0.083	0.148	0.299
PDMS$_3$	0.039	0.048	0.047	0.050	0.055	0.061	**0.079**	**0.109**	**0.199**
RQuick	0.193	0.254	0.293	0.332	0.374	0.445	0.502	0.591	0.657
LCP-RQuick	**0.171**	0.228	0.270	0.303	0.342	0.407	0.460	0.563	0.637
RQuick*	0.230	0.261	0.295	0.324	0.369	0.424	0.476	0.567	0.590
LCP-RQuick*	0.194	**0.222**	**0.250**	**0.276**	**0.309**	**0.358**	**0.411**	**0.491**	**0.509**
	$n/p = 10^5$								
MS$_1$	0.603	0.646	0.679	0.811	0.831	1.071	1.766	2.890	5.550
MS$_2$	0.838	0.827	0.830	0.842	0.920	0.966	1.024	1.053	1.228
MS$_3$	0.836	1.225	1.215	1.136	1.148	1.199	1.276	1.342	1.511
PDMS$_1$	**0.570**	**0.600**	**0.605**	0.647	0.754	0.994	1.653	3.723	11.393
PDMS$_2$	0.650	0.628	0.632	**0.646**	**0.668**	**0.681**	**0.724**	**0.788**	0.948
PDMS$_3$	0.655	0.781	0.760	0.727	0.729	0.736	0.763	0.807	**0.903**
RQuick	2.179	2.888	3.298	3.701	4.140	4.884	5.403	6.266	6.888
LCP-RQuick	1.968	2.647	3.043	3.408	3.793	4.493	4.995	5.798	6.384
RQuick*	1.809	2.157	2.339	2.592	2.900	3.363	3.892	4.468	4.699
LCP-RQuick*	**1.723**	**1.887**	**2.141**	**2.339**	**2.583**	**2.930**	**3.393**	**4.043**	**4.223**
	$n/p = 10^6$								
MS$_1$	4.849	5.204	5.440	5.628	5.790	6.575	8.236	9.346	12.336
MS$_2$	7.385	7.180	7.143	7.207	7.368	7.848	8.402	8.312	8.421
MS$_3$	7.389	11.360	11.020	10.316	10.332	10.512	11.447	11.430	11.832
PDMS$_1$	**4.560**	**4.631**	**4.773**	**4.829**	**5.123**	**5.333**	6.408	8.533	15.213
PDMS$_2$	5.387	5.235	5.234	5.301	5.393	5.423	**5.456**	**5.615**	**5.845**
PDMS$_3$	5.377	6.654	6.444	6.146	6.185	6.112	6.180	6.329	6.546

Table A.1.: Median overall sorting times for the weak-scaling experiment using DNData inputs with $\ell = 500$, $D/N = 1/2$, and $n/p \in \{10^4, 10^5, 10^6\}$. Runtime of the best variant is printed bold for every combination of p and n/p ratio per family of algorithms.

	wall time/s							
$p =$	48	96	192	384	768	1536	3072	6144
	$D/N = 0$							
MS$_1$	0.265	0.342	0.412	0.466	0.541	0.735	0.881	1.410
MS$_2$	0.265	0.717	0.694	0.714	0.807	0.796	0.861	0.986
PDMS$_1$	0.097	**0.097**	**0.105**	**0.129**	**0.134**	0.173	0.230	0.406
PDMS$_2$	**0.087**	0.131	0.129	0.133	0.143	**0.153**	**0.170**	**0.191**
RQuick	0.916	1.528	2.104	2.481	2.876	3.390	3.829	4.647
LCP-RQuick	**0.880**	1.492	2.018	2.402	2.777	3.288	3.715	4.515
RQuick*	1.187	1.444	1.537	1.807	1.978	2.349	2.716	3.210
LCP-RQuick*	1.123	**1.276**	**1.524**	**1.687**	**1.967**	**2.203**	**2.576**	**3.083**
pS^5	0.061	0.075	0.120	0.205	0.416	0.812	1.576	—
	$D/N = 1/4$							
MS$_1$	0.414	0.477	0.527	0.567	0.641	0.770	0.973	1.541
MS$_2$	0.421	0.789	0.763	0.784	0.826	0.844	0.915	0.992
PDMS$_1$	0.322	**0.333**	**0.347**	**0.377**	0.399	0.468	0.631	1.118
PDMS$_2$	**0.321**	0.382	0.374	0.380	**0.387**	**0.399**	**0.421**	**0.448**
RQuick	1.185	1.847	2.456	2.858	3.262	3.765	4.217	5.062
LCP-RQuick	**1.077**	1.704	2.310	2.707	3.089	3.561	3.997	4.812
RQuick*	1.322	1.573	1.828	2.074	2.299	2.609	2.980	3.536
LCP-RQuick*	1.432	**1.538**	**1.691**	**1.898**	**2.133**	**2.413**	**2.703**	**3.239**
pS^5	0.463	0.452	0.568	0.786	1.553	2.864	5.654	—
	$D/N = 1/2$							
MS$_1$	0.563	0.605	0.647	0.699	0.794	0.880	1.057	1.673
MS$_2$	0.567	0.841	0.828	0.861	0.850	0.944	0.928	1.006
PDMS$_1$	**0.555**	**0.576**	**0.584**	**0.610**	0.662	0.762	1.001	1.710
PDMS$_2$	0.555	0.654	0.634	0.658	**0.654**	**0.655**	**0.682**	**0.723**
RQuick	1.501	2.183	2.887	3.306	3.726	4.239	4.711	5.561
LCP-RQuick	**1.316**	1.986	2.635	3.025	3.432	3.903	4.316	5.116
RQuick*	1.589	1.795	2.119	2.368	2.589	2.886	3.355	3.860
LCP-RQuick*	1.456	**1.683**	**1.878**	**2.154**	**2.360**	**2.595**	**2.911**	**3.458**
pS^5	1.076	1.065	1.010	1.375	2.856	4.872	9.657	—
	$D/N = 3/4$							
MS$_1$	0.753	**0.776**	**0.805**	**0.844**	**0.882**	1.013	1.258	1.835
MS$_2$	**0.753**	0.948	0.935	0.936	0.962	**1.005**	**1.036**	**1.064**
PDMS$_1$	0.861	0.890	0.922	0.991	1.046	1.194	2.184	2.461
PDMS$_2$	0.863	1.074	1.060	1.064	1.080	1.114	1.134	1.220
RQuick	1.830	2.614	3.357	3.806	4.225	4.752	5.236	6.124
LCP-RQuick	**1.570**	2.300	3.009	3.374	3.778	4.251	4.703	5.494
RQuick*	1.801	2.167	2.456	2.694	2.961	3.236	3.736	4.289
LCP-RQuick*	1.637	**1.842**	**2.078**	**2.336**	**2.544**	**2.756**	**3.154**	**3.747**
pS^5	1.222	1.233	1.439	1.988	4.152	7.029	13.749	—

Table A.2.: Median overall sorting times for the weak-scaling experiment using DNDATA inputs with $\ell = 500$, $n/p = 10^5$, and $D/N \in \left\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\right\}$. Runtime of the best variant is printed bold for every combination of p and D/N ratio per family of algorithms.

	wall time/s							
$p =$	48	96	192	384	768	1536	3072	6144
	$D/N = 1$							
MS$_1$	0.892	**0.898**	**0.916**	**0.932**	**0.973**	1.096	1.356	1.942
MS$_2$	**0.889**	1.004	0.987	0.994	0.991	**1.021**	**1.012**	**1.075**
PDMS$_1$	1.023	1.038	1.054	1.088	1.191	1.312	2.401	2.623
PDMS$_2$	1.023	1.148	1.129	1.136	1.143	1.159	1.160	1.236
RQuick	2.127	2.965	3.779	4.229	4.698	5.225	5.766	6.611
LCP-RQuick	**1.779**	2.568	3.293	3.707	4.087	4.569	5.000	5.822
RQuick*	2.069	2.311	2.821	3.071	3.317	3.649	4.134	4.678
LCP-RQuick*	1.845	**2.083**	**2.257**	**2.517**	**2.763**	**2.944**	**3.349**	**3.937**
pS^5	1.691	1.652	1.984	2.632	5.340	9.023	17.452	—

Table A.2.: (Continued)

	wall time/s						
$p =$	192	384	768	1536	3072	6144	12 288
	COMMONCRAWL						
MS$_1$	**11.059**	8.172	6.433	4.828	**3.549**	3.134	3.863
MS$_2$	13.722	**7.844**	**5.736**	**4.387**	3.746	3.372	3.024
PDMS$_1$	15.938	10.449	7.702	5.640	4.346	5.197	10.120
PDMS$_2$	**	10.514	7.060	5.092	3.631	**2.700**	**1.812**
PDMS$_2^\nabla$	19.474	10.466	7.031	5.241	4.844	5.118	7.677
	WIKIPEDIA						
MS$_1$	16.145	8.819	5.957	5.207	2.984	2.079	2.309
MS$_2$	**	10.926	8.983	5.861	3.590	2.105	1.160
PDMS$_1$	**9.666**	**5.374**	**3.113**	2.166	1.675	2.506	5.769
PDMS$_2$	11.533	6.001	3.503	**2.037**	**1.161**	**0.695**	**0.470**
PDMS$_2^\nabla$	11.478	6.014	3.568	3.069	1.628	2.312	4.945
	WIKIPEDIATEXT						
MS$_1$	10.299	6.892	5.947	4.068	2.261	1.742	2.218
MS$_2$	13.190	8.483	8.399	4.980	2.725	1.560	0.929
PDMS$_1$	**6.876**	**3.756**	**2.363**	1.673	1.416	2.191	6.163
PDMS$_2$	8.500	4.456	2.592	**1.559**	**0.895**	**0.562**	**0.396**
PDMS$_2^\nabla$	8.452	4.455	2.669	1.792	1.351	1.835	5.080

Table A.3.: Median overall sorting times for the strong-scaling experiment using real-world inputs. Runtime of the best variant is printed bold for all datasets and values of p. Failed runs (due to memory exhaustion) are marked "**".

A.2. Running Times – Space-Efficient Merge Sort

$\frac{n}{p}$	$\frac{D}{N}$	$p =$	wall time/s						
			48	96	192	384	768	1536	3072
2×10^6	$\frac{1}{8}$	$\texttt{MS}_1$	11.97	14.75	16.93	18.30	19.96	22.27	28.53
		$\texttt{MS}_2$	12.05	28.11	27.24	27.11	28.69	31.25	33.57
		$\texttt{PDMS}_1$	5.01	**5.46**	**5.90**	**6.14**	**6.52**	**7.27**	**8.26**
		$\texttt{PDMS}_2$	**4.98**	6.87	7.29	7.65	8.20	8.27	9.19
2×10^6	$\frac{1}{4}$	$\texttt{MS}_1$	13.45	16.01	17.92	19.43	21.12	24.43	31.76
		$\texttt{MS}_2$	13.52	27.84	27.30	27.08	28.27	31.02	32.85
		$\texttt{PDMS}_1$	**8.21**	**8.93**	**9.66**	**9.98**	**10.80**	**11.89**	13.62
		$\texttt{PDMS}_2$	8.21	11.30	11.92	12.00	13.20	12.68	**13.44**
2×10^6	$\frac{1}{2}$	$\texttt{MS}_1$	16.64	18.40	20.07	21.49	23.77	28.34	38.15
		$\texttt{MS}_2$	16.64	27.61	27.24	27.30	27.84	30.10	31.38
		$\texttt{PDMS}_1$	14.27	**15.30**	**16.31**	**17.20**	**18.66**	**21.23**	26.07
		$\texttt{PDMS}_2$	**14.16**	19.42	19.88	20.51	20.76	21.44	**22.56**
4×10^6	$\frac{1}{8}$	$\texttt{MS}_1$	25.12	30.69	35.14	37.69	41.11	45.76	58.07
		$\texttt{MS}_2$	25.36	57.52	55.83	55.43	58.62	63.51	68.60
		$\texttt{PDMS}_1$	11.38	**12.14**	**12.87**	**13.55**	**14.21**	**15.07**	**16.62**
		$\texttt{PDMS}_2$	**11.27**	15.18	15.51	15.83	16.43	17.29	18.07
4×10^6	$\frac{1}{4}$	$\texttt{MS}_1$	29.43	34.41	38.29	40.98	44.60	51.05	65.25
		$\texttt{MS}_2$	29.61	58.48	56.84	56.49	58.65	64.28	67.99
		$\texttt{PDMS}_1$	18.88	**19.97**	**21.19**	**21.87**	**23.12**	**25.14**	28.32
		$\texttt{PDMS}_2$	**18.85**	24.94	25.62	26.06	26.39	27.09	**28.19**
4×10^6	$\frac{1}{2}$	$\texttt{MS}_1$	37.89	41.51	44.60	47.28	51.92	60.88	80.75
		$\texttt{MS}_2$	37.86	59.74	58.60	58.16	59.06	63.86	66.89
		$\texttt{PDMS}_1$	32.92	**34.64**	**36.48**	**37.73**	**40.35**	**45.53**	55.11
		$\texttt{PDMS}_2$	**32.87**	43.33	43.43	43.61	44.08	45.56	**48.11**
8×10^6	$\frac{1}{8}$	$\texttt{MS}_1$	52.15	62.93	72.15	77.24	82.24	93.21	117.91
		$\texttt{MS}_2$	52.42	117.47	113.17	112.22	118.97	128.92	138.86
		$\texttt{PDMS}_1$	24.66	**26.11**	**27.51**	**28.83**	**29.51**	**30.91**	**34.38**
		$\texttt{PDMS}_2$	**24.58**	32.30	32.69	33.17	34.48	34.87	37.01
8×10^6	$\frac{1}{4}$	$\texttt{MS}_1$	62.84	72.68	80.72	86.20	93.32	106.22	133.37
		$\texttt{MS}_2$	63.13	121.01	117.63	116.83	121.12	132.65	140.31
		$\texttt{PDMS}_1$	**41.85**	**44.17**	**46.13**	**47.38**	**49.53**	**52.94**	59.63
		$\texttt{PDMS}_2$	41.90	54.31	54.71	54.83	56.06	57.54	**59.36**
8×10^6	$\frac{1}{2}$	$\texttt{MS}_1$	83.57	90.79	97.04	102.59	111.99	129.58	169.50
		$\texttt{MS}_2$	83.68	127.67	124.88	123.65	126.26	136.12	141.73
		$\texttt{PDMS}_1$	**73.93**	**77.36**	**80.39**	**83.36**	**88.13**	**97.29**	117.56
		$\texttt{PDMS}_2$	73.94	94.90	93.87	94.16	95.30	97.42	**102.83**

Table A.4.: Median overall sorting times for the weak-scaling experiment using DNDataSE inputs $\ell = 1000$, $n/p \in \{2, 4, 8\} \times 10^6$, and $D/N \in \left\{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}\right\}$. Runtime of the best variant is printed bold per value of p and every combination of n/p and D/N ratios.

algorithm (M/p)	v	$p =$	wall time/s 48	96	192	384	768	1536	3072	6144	12 288
CommonCrawl											
pS^5 (100 MB)	512		2.6	5.6	12.0	25.4	55.1	119.0	247.8	—	—
	1024		2.0	4.2	9.1	20.7	45.7	100.4	208.8	—	—
	2048		1.6	3.4	7.5	17.5	40.4	90.3	186.6	—	—
	4096		1.3	3.0	6.7	15.7	40.0	90.8	169.3	—	—
	8192		0.9	2.1	4.9	11.6	31.4	78.0	156.4	—	—
SEMS$_2$ (100 MB)	512		14.2	32.3	35.1	36.9	39.9	46.6	59.3	81.6	—
	1024		17.6	42.7	46.3	49.4	53.8	60.8	76.7	104.1	—
	2048		22.7	58.3	63.8	68.3	72.9	84.4	107.0	145.0	—
	4096		30.1	80.6	88.0	93.4	99.7	116.3	141.9	197.5	—
	8192		38.5	103.7	114.4	123.9	134.3	155.5	189.7	250.0	—
PDSEMS$_2$ (100 MB)	512		11.3	19.7	23.8	29.5	35.5	40.4	45.3	49.2	52.0
	1024		9.9	17.7	23.3	30.0	37.0	44.6	53.5	54.3	59.4
	2048		9.1	17.2	23.7	31.5	39.9	50.5	63.0	64.7	72.3
	4096		8.0	17.3	24.4	32.4	44.1	59.0	69.4	78.7	91.2
	8192		7.4	17.1	25.4	35.7	50.1	66.1	85.5	94.6	112.4
PDSEMS$_2$ (200 MB)	512		25.3	**	**	**	**	**	**	**	—
	1024		22.3	42.5	54.6	66.4	78.7	91.6	100.4	108.7	—
	2048		20.9	40.3	54.9	72.9	88.6	100.7	115.9	129.7	—
	4096		20.0	42.7	58.9	77.7	102.1	117.2	137.1	157.2	—
	8192		18.7	41.9	60.7	82.1	111.2	128.7	157.4	186.3	—
WikipediaFull											
pS^5 (100 MB)	512		5.8	21.4	23.1	46.2	93.1	186.4	371.0	—	—
	1024		12.4	13.2	25.5	51.5	100.5	196.8	395.5	—	—
	2048		14.1	14.5	28.0	56.9	110.2	212.0	427.6	—	—
	4096		17.3	16.7	31.5	64.0	121.8	233.0	429.2	—	—
	8192		12.8	13.2	24.3	50.1	93.3	177.3	363.5	—	—
SEMS$_2$ (100 MB)	512		71.5	76.6	78.2	80.2	81.8	83.0	84.9	92.3	—
	1024		87.4	94.8	97.8	99.1	101.0	104.0	109.8	114.6	—
	2048		86.9	99.3	104.7	107.3	112.0	116.5	125.7	172.4	—
	4096		55.2	77.2	88.9	94.4	102.0	122.5	134.9	192.3	—
	8192		53.0	92.9	113.6	122.3	134.9	153.2	172.5	197.7	—
PDSEMS$_2$ (100 MB)	512		75.4	81.0	81.3	83.0	85.3	86.3	87.5	94.5	96.5
	1024		91.0	98.6	100.7	102.1	103.4	105.1	110.8	113.9	121.2
	2048		89.6	101.0	104.7	107.2	109.2	111.9	120.9	166.3	174.6
	4096		58.0	76.0	82.1	87.2	89.6	108.6	124.5	190.7	208.3
	8192		40.8	62.0	69.6	76.4	80.1	87.2	105.4	121.3	161.5
PDSEMS$_2$ (200 MB)	512		122.4	131.7	156.4	163.1	164.5	170.7	174.1	176.2	—
	1024		141.1	154.6	186.7	190.3	192.7	196.9	218.2	260.9	—
	2048		161.8	183.9	190.7	196.1	202.2	264.9	272.2	290.9	—
	4096		130.8	161.4	174.7	180.6	185.9	253.5	304.7	324.0	—
	8192		92.3	132.9	151.1	158.5	183.1	219.4	277.9	318.0	—

Table A.5.: Median overall sorting times for the weak-scaling experiment using difference cover samples modulo v for real-world datasets with chunks of M/p characters per PE. Failed runs are denoted as "**", missing runs are indicated "—".