Scalable Distributed String Sorting

Florian Kurpicz
Karlsruhe Institute of Technology, Germany

Pascal Mehnert
Independent, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Matthias Schimek
Karlsruhe Institute of Technology, Germany

Abstract

String sorting is an important part of tasks such as building index data structures. Unfortunately, current string sorting algorithms do not scale to massively parallel distributed-memory machines since they either have latency (at least) proportional to the number of processors p or communicate the data a large number of times (at least logarithmic). We present practical and efficient algorithms for distributed-memory string sorting that scale to large p. Similar to state-of-the-art sorters for atomic objects, the algorithms have latency of about $p^{1/k}$ when allowing the data to be communicated k times. Experiments indicate good scaling behavior on a wide range of inputs on up to 49 152 cores. Overall, we achieve speedups of up to 4.9 over the current state-of-the-art distributed string sorting algorithms.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Theory of computation → Massively parallel algorithms; Computing methodologies → Distributed algorithms; Theory of computation → Bloom filters and hashing

Keywords and phrases sorting, strings, distributed-memory computing, distributed membership filters, scalability

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.83

Related Version This paper is based on the master's thesis of Pascal Mehnert [21].
Full Version: `https://arxiv.org/abs/2404.16517` [19]
Brief Announcement: `https://dl.acm.org/doi/10.1145/3626183.3660256` [18]

Supplementary Material
Software (Source Code): `https://github.com/mschimek/scalable-distributed-string-sorting` [22], archived at `swh:1:dir:1d60272c5beeb821650519f3f0ce805434b705fa`

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).
The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`).
This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

1 Introduction

Sorting strings is a fundamental building block of many important string-processing tasks such as the construction of index data structures for databases and full-text phrase search [10, 16, 25]. The problem differs from *atomic* sorting – where keys are treated as indivisible objects that can be compared in constant time. Strings on the other hand can have variable lengths and the time needed to compare two strings depends on the length of their longest common prefix. Therefore, string sorting algorithms try to avoid the repeated comparison of whole strings. Instead, they only inspect the *distinguishing prefixes* of the strings, i.e., the characters needed to establish the global ordering, once. The sum of the lengths of all distinguishing prefixes is usually denoted by D whereas N is the total number of characters and we often find $D \ll N$. The lower bound for sequential string sorting based on character comparisons is $\Omega(n \log n + D)$ with existing algorithms matching this bound [4].

While the problem has been extensively researched in the sequential and (shared-memory) parallel setting – both in theory and practice – the results for distributed-memory string sorting algorithms are limited. Bingmann et al. present the two state-of-the-art distributed string sorting algorithms [8].

For the first one, they follow the standard (distributed-memory) parallel merge sort scheme [31, 2] in which the input is first sorted locally on each of the p processing elements (PEs). Then $p-1$ splitter elements are determined globally and used to partition the locally sorted input into p parts on each PE. Subsequently, the i^{th} such part is communicated to the i^{th} PE. Finally, a local p-way merging step is performed to obtain the globally sorted output. Bingmann et al. augment every step of this scheme with string-specific optimization like LCP-compression and LCP-aware merging [24, 8, 6]. Their second algorithm takes one step further towards the goal of a communication-efficient string sorting algorithm by approximating the distinguishing prefix of each string before the actual sorting process. Then only the approximated distinguishing prefixes are sorted using the distributed string merge sort algorithm. Furthermore, they also adapt the distributed hypercube quicksort algorithm [2, 3] to variable length keys but without incorporating further string-related optimizations.

While currently representing the state-of-the-art, these algorithms are only efficient for very small or large inputs, because they have a prohibitively high communication volume (hypercube quicksort) or do not scale to the largest available machines due to their latency which is (at least) proportional to the number of processors p (merge sort algorithms).

Our Contribution. Our new algorithms close this gap in the current landscape of distributed string sorting by providing a viable trade-off between latency and communication volume. This is achieved by adapting a multi-level approach known from distributed *atomic* sorting to Bingmann et al.'s string sorting algorithms, i.e., we use k levels where processor groups work on independent sorting problems. The resulting *multi-level mergesort* has internal work and communication volume close to N for each level (Theorem 7). This is significantly improved with *prefix-doubling mergesort* where internal work and communication volume per level are close to D (Theorem 11). As a side result of independent interest, we present a *multi-level distributed single-shot Bloom filter* (Theorem 9). We also improve the variable-length hypercube quicksort algorithm (Theorem 1) by including some previously missing string-related optimizations such as exploiting LCP values during sorting.

While the idea to combine the best string sorting and atomic sorting algorithm is simple in principle, we view the analysis of a quite complicated overall algorithm in a realistic model of distributed-memory computing as a significant contribution. In particular, because this guides an efficient, highly scalable implementation. In an experimental evaluation, on up to 49 152 cores, the multi-level algorithms are up to 4.9× faster than the single-level ones.

Related Work. There has been extensive research on string sorting in the sequential setting. For a systematic overview and a good starting point in this field, we refer to [5, 15, 30]. Here, however, we focus on parallel sorting algorithms. Let n be the total number of strings and

Table 1 Symbols used in this paper.

Machine Model		**String Properties**	
p	number of processing elements (PEs)	n	total number of strings
r	number of groups in each recursion level	N	total number of characters
k	number of recursion levels	$\hat{\ell}$	length of longest string
α	message start-up latency	$\underset{\vee}{\ell}$	length of shortest string
β	time to communicate a bit	$\hat{d}$	length of longest distinguishing prefix

N be the total number of characters. See Table 1 for a full list of variables and notation used throughout this paper. For the PRAM model, there are (comparison-based) algorithms solving the string sorting problem in $\mathcal{O}(n \log n + N)$ work and $\mathcal{O}(\log^2 n / \log\log n)$ time [14]. For integer alphabets Hagerup proposes an algorithm with $\mathcal{O}(N \log\log N)$ work and running time in $\mathcal{O}(\log N / \log\log N)$ [12]. Recently, Ellert et al. proposed an algorithmic framework for the PRAM model to turn any string sorting algorithm into a *D-aware* sorting algorithm, i.e., an algorithm whose (work) complexity depends on D instead of N, increasing the time complexity only by a logarithmic factor in the length of the overall longest distinguishing prefix $\hat{d}$ [9].

For the distributed setting, the algorithms by Bingmann et al. that we briefly discussed in the introduction are the current state-of-the-art. They surpass the first dedicated string sorting algorithm that was utilized for suffix sorting by Fischer and Kurpicz [11]. We are not aware of other publications targeting distributed string sorting algorithms. However, string sorting is also considered in other parallel settings, e.g., on the GPU [23].

2 Preliminaries

Machine Model and Communication Primitives. We assume a distributed-memory machine model consisting of p processing elements (PEs) allowing single-ported point-to-point communication. The cost of exchanging a message of h bits between any two PEs is $\alpha + \beta h$, where α accounts for the message start-up overhead and β quantifies the time to exchange one bit. Let h be the maximum number of bits a PE sends or receives, then collective operations *broadcast*, *prefix sum*, *(all-)reduce*, and *(all-)gather* can be implemented in time $\mathcal{O}(\alpha \log p + \beta h)$ [26]. For personalized all-to-all communication, there is a trade-off between communication volume and start-up latency. When data is delivered directly, we obtain a time complexity in $\mathcal{O}(\alpha p + \beta h)$, while using a maximum degree of indirection yields $\mathcal{O}(\alpha \log p + \beta h \log p)$. In our analysis, we resort to a more abstract view introduced by Axtmann et al. – a black box data exchange function $\mathrm{Exch}(h, r)$ yielding the time complexity of exchanging data , when each PE sends or receives at most h bits in total from at most r PEs [2]. We find $\alpha r + \beta h$ as a lower bound for the time complexity of $\mathrm{Exch}(h, r)$, and although matching upper bounds are not known, there are indications suggesting that one can come close to this (see [2] for a brief discussion). We use $\widetilde{\mathrm{Exch}}(h, r) = (1 + o(1))\mathrm{Exch}(h, r)$ to sum up multiple exchanges by the dominant one.

String Properties and Input Format. The input to our algorithms is a string array $S = [s_0, s_1, \ldots, s_{n-1}]$ consisting of $n = |S|$ unique strings. A string s of length $|s| = \ell$ is a sequence of characters from an alphabet Σ with $s = [s[0], \ldots, s[\ell - 2], \perp]$ where $\perp \notin \Sigma$ is a sentinel character; $\hat{\ell}$ denotes the length of the longest string in S. Note that due to the sentinel character, all strings are prefix-free. By $N = \|S\|$, we denote the total number of characters in S. The ℓ-prefix of a string s comprises the first ℓ characters of s. The *longest common prefix (LCP)* of two strings $s \neq t$ is the prefix of s with length $\operatorname{lcp}(s,t) = \arg\min s[i] \neq t[i]$. For a sorted string array, the corresponding LCP array $[\perp, h_1, h_2, \ldots, h_{n-1}]$ contains the LCP values $h_i = \operatorname{lcp}(s_{i-1}, s_i,)$. For sorting the string array, we do not necessarily have to look at all characters in S. The *distinguishing prefix* of a string s (with length $\textsc{dist}(s)$) are the characters that need to be inspected to rank s in S. The sum of the lengths of all distinguishing prefixes of S is denoted by D. By $\hat{d}$, we denote the length of the longest distinguishing prefix. A string array is usually represented as an array of pointers referring to the start of the corresponding character sequence. This allows for moving or swapping strings in constant time. The concatenation of all the character sequences forms the *character* array $\mathcal{C}(S)$ with $|\mathcal{C}(S)| = \|S\|$.

In our distributed setting, we assume that each PE i obtains a local subarray S_i of S as input such that S is the concatenation of all local string arrays S_i. Furthermore, we assume the input to be well-balanced, i.e., $|S_i| = \Theta(n/p)$, $\|S_i\| = \Theta(N/p)$, and $\sum_{s \in S_i} \textsc{dist}(s) = \Theta(D/p)$.

Algorithmic Building Blocks. We make use of an *r-way LCP loser tree* [6] for merging r sorted sequences of strings with associated LCP arrays. This data structure combines the binary LCP-merging technique proposed by Ng and Kakehi [24] with the (atomic) r-way loser tree [17]. An r-way loser tree is a binary tree with r leaves. Each leaf is associated with the current head of a sorted sequence of elements. For initialization, this head element is passed up the tree. In each internal node of the tree, the arriving elements are compared and the smaller one is forwarded up while the larger element, i.e., the *loser*, is stored at the node. Continuing this process, the overall smallest element w finally arrives at the root. We now have the invariant that in each node of the tree the smaller element originating from its two children is stored. When removing w and advancing the corresponding sequence by one element, this invariant can be re-established in $\mathcal{O}(\log r)$ time as only the states of internal nodes on w's path from its leaf to the root have to be repaired. Loser trees can be adapted to strings by additionally storing the intermediate LCP values of compared strings in each internal node. These values can then be exploited in the next round to save character comparisons as both strings know their LCP value to a common smaller string – the previous winner string at the node. This allows us to merge m strings in r sorted sequences augmented with LCP values in time $\mathcal{O}(m \log r + D)$. While merging, we obtain the LCP values of the resulting merged string sequence as a by-product.

Furthermore, we use LCP compression, i.e., we send the longest common prefix of two consecutive strings (in a sorted sequence) only once. While being very useful for many inputs in practice, LCP compression cannot substantially reduce the communication volume in the worst case [8, 28].

Robust Hypercube Quicksort (RQuick) is a sorting algorithm initially proposed for atomic sorting [3, 1] that has been adapted to handle strings [8]. RQuick only works for a number of PEs that is a power of two. Hence, with $d = \lfloor \log p \rfloor$, PEs with index $i \geq 2^d$ send their data to a PE within the d-dimensional hypercube. Then the input data is randomly permuted among the PEs to make imbalances less likely. Subsequently, each PE sorts its local data. Then the actual d hypercube (quick)sorting rounds are executed, where the input is recursively partitioned into sub-hypercubes using a pivot element. Hence, the elements are communicated at least $d = \lfloor \log p \rfloor$ times.

▶ **Theorem 1** (String RQuick, [8, Theorem 1] combined with [1, Theorem 6.6]). *If all input strings are unique, RQuick runs in time $\mathcal{O}\left(\hat{\ell}\frac{n}{p}\log n + \alpha \log^2 p + \beta\left(\frac{n}{p}\hat{\ell}\log p + \hat{\ell}\log^2 p\right)\log\sigma\right)$ with probability $\geq 1 - p^{-c}$ for any constant $c > 0$.*

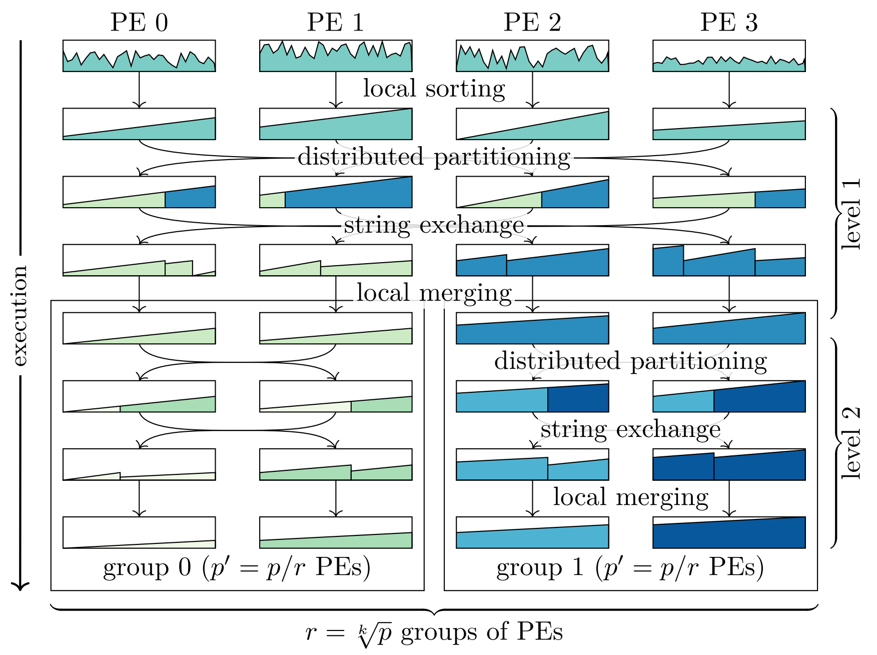

Figure 1 Overview of the main steps in the multi-level string sorting scheme with $k = 2$ levels.

RQuick can be easily improved by exploiting the LCP values computed during the local sorting phase for LCP-aware merging. Our implementation `RQuick+` uses this optimization. Therefore, the overall work performed by all PEs for these two steps is $\mathcal{O}(n \log n + D)$.

3 Multi-Level String Sorting

Our *multi*-level **m**erge **s**ort (MS) algorithm adapts ideas of Axtmann et al. for multi-level *atomic* sorting [2, 3] to string sorting. The key point is to allow each string to be moved multiple (k) times between PEs instead of only once and therefore trade communication volume for latency with k being a tunable parameter. In the algorithm, we partition the input into independent (string) sorting problems, split the PEs into groups, and let each group recursively work on one of the sorting problems until the input is globally sorted. Figure 1 provides an overview of the main phases of the approach.

There are k levels of recursion with arbitrary splitting factors between levels. To simplify the analysis, we assume that p can be perfectly subdivided into r groups of p/r consecutive PEs, i.e., on the first level the j^{th} group consists of PEs $jp/r, \dots, (j+1)p/r - 1$. Furthermore, we generally assume approximately equal factors on each level, i.e., $r = \Theta(\sqrt[k]{p})$ which implies $p = \Theta(r^k)$, and that the final level splits the PEs into groups of size 1. To obtain the single-level variant [8] of the algorithm choose $k = 1$.

We now discuss the main steps of our approach in more detail. The algorithm consists of a one-time initialization and a recursive phase which is invoked k times.

Initialization. On each PE i, sort the local input array $\mathcal{S}_i$. The LCP array can be obtained as a by-product of sorting.

Recursion. A global order is established recursively. Initially, all string arrays $\mathcal{S}_i$ must be locally sorted. On each level of recursion, we have p' PEs and r groups of size $p'' = p'/r$.

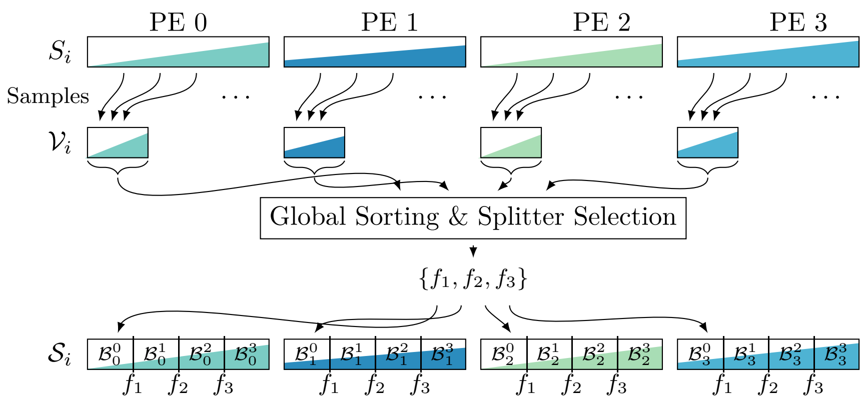
PE 0
PE 1
PE 2
PE 3
S_i
Samples
$\mathcal{V}_i$
Global Sorting & Splitter Selection
$\{f_1, f_2, f_3\}$
$\mathcal{S}_i$
$\mathcal{B}_0^0$ $\mathcal{B}_0^1$ $\mathcal{B}_0^2$ $\mathcal{B}_0^3$
$\mathcal{B}_1^0$ $\mathcal{B}_1^1$ $\mathcal{B}_1^2$ $\mathcal{B}_1^3$
$\mathcal{B}_2^0$ $\mathcal{B}_2^1$ $\mathcal{B}_2^2$ $\mathcal{B}_2^3$
$\mathcal{B}_3^0$ $\mathcal{B}_3^1$ $\mathcal{B}_3^2$ $\mathcal{B}_3^3$
f_1 f_2 f_3

Figure 2 Overview of the partitioning phase.

1. **Distributed Partitioning.** Globally determine $r - 1$ splitter strings f_j and on each PE i compute local *buckets* $\mathcal{B}_i^0, \dots, \mathcal{B}_i^{r-1}$ with $\mathcal{B}_i^j = \{s \in S_i \mid f_j < s \leq f_{j+1}\}$ for $j \in \{0, \dots, r-1\}$ using sentinels $f_0 = -\infty$ and $f_r = \infty$.
2. **String Assignment and Exchange.** On PE i, the strings in bucket $\mathcal{B}_i^j$ are assigned to PEs belonging to group j. By $\mathcal{B}^j = \bigcup_i \mathcal{B}_i^j$, we denote the union of all strings assigned to group j. Then, all strings and LCP values are exchanged using direct messaging.
3. **Local LCP-aware Merging.** On PE i, the received string sequences are merged to obtain locally sorted string arrays $\mathcal{O}_i$ (using the LCP values). We also update the LCP values for $\mathcal{O}_i$ during merging. We then set $p' \leftarrow p'/r$ and $S_i \leftarrow O_i$ in the subsequent recursive step.

We now describe the distributed partitioning (Section 3.1) and the distributed assignment and exchange phase (Section 3.2) in more detail. An analysis of the overall running time will be given in Section 3.3.

3.1 Distributed Partitioning

Due to the multidimensionality of the string sorting problem, determining balanced partitions is more challenging than in atomic sorting. Some of the steps of our merge sort algorithm depend on the number of strings in the local string array while others depend on the number of characters or the size of the distinguishing prefix. We therefore adapt *string-based* and *character-based* partitioning [8] to our multi-level approach. These schemes bound the number of strings and characters, respectively. The general approach is to draw and globally sort a number of samples on each PE. Then, $r - 1$ splitters f_j are chosen from the global sample array, see Figure 2. Since all strings are locally sorted before the partitioning step, we can make use of a *regular* sampling approach [20, 29] in which the samples are drawn equidistantly.

3.1.1 String-Based Partitioning

On the t^{th} recursion level, there are r^{t-1} groups of PEs working on independent sorting problems, see Figure 1. We now describe the partitioning process from the point of view of one such group. Let $\mathcal{S}'$ be the concatenation of the local string arrays of the PEs of one such group of size p'. Unlike for single-level MS, now, we cannot assume an equal number of strings on each PE as from the second level of recursion on, this number is subject to the result of a previous partitioning round which is not exact. For multi-level MS, string-based partitioning consists of the following two steps:

Local Sampling: Let $v > 0$ be the *sampling* factor. In total, there will be $p'(v+1)$ samples drawn from the local string arrays. To simplify the discussion, we assume $|\mathcal{S}'|$ to be divisible by $p'(v+1)$. Let $\omega = |\mathcal{S}'|/(p'(v+1))$. PE i then draws $\lceil |\mathcal{S}_i|/\omega \rceil - 1$ samples $\mathcal{V}_i$ from its local string array spaced as evenly as possible. $\mathcal{V}$ is the union of all local samples. If $|\mathcal{V}| < p'(v+1)$, then the first $p'(v+1) - |\mathcal{V}|$ PEs draw one additional sample. This ensures that at most ω strings are between two local samples on each PE. Also, the global number of samples is a multiple of p' and of r as we find $p' = r^{k+1-t}$ on recursion level t.

Splitter Computation: The samples $\mathcal{V}$ are globally sorted using hypercube quicksort. Then $r-1$ splitters $f_j = \mathcal{V}[j|\mathcal{V}|/r - 1]$ for $0 < j < r$ are determined using a prefix sum. Subsequently, the $r-1$ splitters are communicated to all PEs using an all-gather operation.

A sampling factor $v = \Theta(r)$ yields a maximum number of strings per bucket in $\Theta(|\mathcal{S}'|/r)$. This is shown in detail by using a generalization of the sample density lemma from [8] in the full version [19]. The proof of the following Lemma 2 can also be found there.

▶ **Lemma 2.** *On recursion level t with $r = \sqrt[k]{p}$ in step 1 of multi-level MS, string-based regular sampling with sampling factor v yields a maximum bucket size of $|\mathcal{B}^j| \leq \left(1 + \frac{r}{v}\right)^t \frac{n}{r^t}$.*

The term $(1 + r/v)^t$ signifies that the imbalance between the buckets multiplies with each level of recursion. Therefore, we need to choose $v = \Theta(kr)$ for any number of k levels to keep the term asymptotically constant during the entire sorting process. For single-level MS, $v = \Theta(r) = \Theta(p)$ samples per PE are sufficient. Note the large difference between drawing kr samples per PE (on average) here and $r^k = p$ samples in the single-level case. Using the assignment strategy described in Section 3.2, which equally distributes the strings over the PEs in each group, we arrive at Theorem 3.

▶ **Theorem 3** (String-Based Sampling). *Using a sampling factor of $v = \Theta(kr) = \Theta(k\sqrt[k]{p})$ the number of strings per PE is in $\mathcal{O}(n/p)$ in each level of the algorithm.*

3.1.2 Character-Based Sampling

We generalize *character*-based regular sampling [8] to our multi-level approach to achieve tighter bounds on the number of characters per PE than the conservative $\mathcal{O}(\hat{\ell}n/p)$. Now, each PE of the considered group draws $\lceil \|\mathcal{S}_i\|/\omega' \rceil - 1$ equally spaced samples from its character array with sampling distance $\omega' = \|\mathcal{S}'\|/(p'(v+1))$. To arrive at the final string samples, we shift the sampled character positions by at most $\hat{\ell} - 1$ characters to the beginning of a string. If the total number of samples is smaller than $p'(v+1)$, the first PEs draw one additional sample.

In Lemma 4, we give bounds on the number of characters per bucket over the course of our algorithm when using character-based partitioning. This is shown in detail by using a generalization of the (character-) sample density lemma [8] in the full version [19].

▶ **Lemma 4.** *On recursion level t with $r = \sqrt[k]{p}$ in step 1 of multi-level MS using character-based regular sampling with a sampling factor of v, each bucket contains at most $\|\mathcal{B}^j\| \leq \left(1 + \frac{r}{v}\right)^t \left(\frac{N}{r^t} + t\left(1 + \frac{v+1}{r}\right)\frac{p}{r^{t-1}}\hat{\ell}\right)$ characters.*

The additional term depending on $\hat{\ell}$ stems from shifting the sampled positions to the beginning of strings. By distributing the characters equally over the PEs in each group up to additional $\mathcal{O}(\hat{\ell})$ characters (see Section 3.2), we can limit the maximum number of characters per PE in Theorem 5.

▶ **Theorem 5** (Character-Based Sampling). *Using a sampling factor in $\Theta(kr)$ the maximum number of characters per PE in each level is in $\mathcal{O}\left(\frac{N}{p} + k^2 r\hat{\ell}\right)$.*

For the single-level case with $k = 1$, this is equivalent to $O(N/p + p\hat{\ell})$ which is the bound in the original algorithm [8]. For $k > 1$, we even have an improvement over the single-level algorithm. Assuming k in $\mathcal{O}(\log p / \log\log p)$, we find $k^2 r\hat{\ell} = \mathcal{O}(\log^2(p) \sqrt[k]{p}\hat{\ell}) = o(p\hat{\ell})$. This may seem counter-intuitive at first as we introduce a potential imbalance already in the first recursion level. However, the subsequent assignment step distributes this imbalance equally over p'/r PEs.

3.2 String Assignment and Exchange

We now assign the strings in bucket $\mathcal{B}^j$ to the j^{th} (sub-)group consisting of PEs jp'',…, $(j+1)p'' - 1$ with $p'' = p'/r = r^{k-t}$ on the t^{th} level. The resulting assignment needs to ensure that each PE receives approximately the same amount of data. Additionally, the number of sent and received messages per PE should be bounded by the number of groups r. We generalize an approach proposed by Axtmann et al. [2] for distributed atomic sorting to string sorting. For the sake of simplicity, we assume that each PE contributes the same number of strings (characters) in the assignment process. With (slightly) imbalanced data (due to the partitioning) the below-stated results hold up to a small factor.

For *string-based* partitioning, we want to balance the number of strings per PE. Since the assignment algorithm does not rely on internal properties of the elements, we can treat a string as an *atomic* object and apply Axtmann et al.'s assignment algorithm directly.

For *character-based* partitioning, we want to achieve a balanced number of characters but cannot split strings. Therefore, we reiterate the steps of the algorithm and describe the necessary adjustments. A local bucket $\mathcal{B}_i^j$ is *small* if it contains at most $\|\mathcal{B}^j\|/(2rp'')$ characters. Small buckets are separately enumerated for each group j using a prefix sum where each PE contributes its number of small buckets for group j. The q^{th} small bucket belonging to group j is then assigned to PE $\lfloor q/r \rfloor$ of group j. This way, each PE gets assigned no more than half of its final capacity and receives messages from at most r different PEs.

Then, a description of each large bucket located on PE i and destined for group j is first sent to PE $\lfloor i/r \rfloor$ in group j and each group computes a balanced assignment independently of each other. Conceptually, this works by performing separate prefix sums over residual capacities (remaining after assigning small buckets) and sizes of unassigned buckets, in each group. The resulting sorted sequences of integers R (residual capacities) and U (unassigned buckets) must then be merged such that the bucket beginning at the i^{th} element is preceded by the PE containing the i^{th} open slot. A subsequence of $\langle r_i, u_j, \ldots, u_{j+h}, r_{i+1}, z\rangle$ in the merged sequence of R and U means that the local buckets $u_j, \ldots, u_{j+h}$ are assigned to PE i. The last bucket u_{j+h} potentially needs to be split up (respecting string boundaries in the character-based assignment) and partly assigned to r_{i+1} or even r_{i+2} if $z = r_{i+2}$. Since we cannot split up strings, we may end up with a PE obtaining up to $\hat{\ell} - 1$ additional characters. Strings from one local bucket cannot be assigned on more than 3 PEs as the residual capacity on each PE is at least $\|\mathcal{B}^j\|/(2p'')$. Since large buckets contain more than $\|\mathcal{B}^j\|/(2rp'')$ elements, a single PE can store at most $2r$ of them. Hence, each PE receives $\mathcal{O}(r)$ messages. We refer to [2] for details on the actual group-local merging process.

▶ **Theorem 6** (Bounded Assignment). *Using the bounded assignment algorithm [2], we obtain a message assignment where each PE sends and receives $\mathcal{O}(r)$ messages and each PE in group j obtains $\frac{|\mathcal{B}^j|}{p''}$ strings (string-based) or at most $\frac{\|\mathcal{B}^j\|}{p''} + \hat{\ell}$ characters (character-based).*

Hence, even for character-based sampling, we can find an assignment, such that the number of characters per PE remains in the bounds of Theorem 5. Afterwards, each PE sends its strings according to their assignment. The time for computing the assignment is dominated by the actual data exchange [2]. [1]

3.3 Overall Running Time

Let $\Theta(kr)$ be the sampling factor of our algorithm with character-based sampling. By Theorem 5, the maximum number of characters per PE at any time is $\mathcal{O}(\widetilde{N}/p)$ with $\widetilde{N} = N + k^2 r\hat{\ell}p$. Since $n \leq N/\check{\ell}$ and $N \leq n\hat{\ell}$, the number of strings per PE then is $\mathcal{O}(\tilde{n}/p)$ with $\tilde{n} = \hat{\ell}/\check{\ell}(n + k^2 rp)$.

We now combine the running time of the three phases of our algorithm at each level, including $\mathcal{O}(n/p \log n/p + D/p)$ time for the initial local string sorting [9].

1. **Distributed Partitioning.** Here, we have to globally sort $\mathcal{O}(rk)$ local sample strings of length $\leq \hat{\ell}$ per PE. With RQuick, this is possible in $\mathcal{O}(rk\hat{\ell} \log p(1 + \beta \log \sigma) + \alpha \log^2 p)$ expected time. Allgathering the $r - 1$ splitters needs $\mathcal{O}(\alpha \log p + \beta r\hat{\ell} \log \sigma)$ time. As we have $r\hat{\ell} = \mathcal{O}(\widetilde{N}/p)$ by definition and the local string array as well as the splitters are sorted, computing the local buckets $\mathcal{B}^i_j$ is possible in time $\mathcal{O}(\widetilde{N}/p)$.
2. **String Assignment and Exchange.** The string assignment is dominated by the data exchange which is possible in time $\widetilde{\mathrm{Exch}}(\mathcal{O}(\widetilde{N}/p \log \sigma), \mathcal{O}(r))$ on level t. This holds as a PE stores $\mathcal{O}(\widetilde{N}/p)$ characters (Theorem 5) encoded in $\log \sigma$ bits each. By the bounded assignment algorithm each PE exchanges strings with at most $\mathcal{O}(r)$ other PEs (Theorem 6).
3. **Local (LCP-aware) Merging.** The $\mathcal{O}(r)$ sorted sequences of strings received in the data exchange now need to be merged to restore our invariant on the local string array. While being beneficial for many inputs in practice, asymptotically, LCP-aware merging does not yield substantial advantage. Thus, we resort to uninformed merging for our analysis. Processing the $\mathcal{O}(r)$ sorted sequences with $\mathcal{O}(\widetilde{N}/p)$ characters in total is possible in $\mathcal{O}(\widetilde{N}/p + \tilde{n}/p \log r)$ time, dominating the time required for computing the local buckets during partitioning.

▶ **Theorem 7.** *Multi-level MS with $r = \sqrt[k]{p}$, $\widetilde{N} = N + k^2 r\hat{\ell}p$, and $\tilde{n} = \hat{\ell}/\check{\ell}(n + k^2 rp)$, using character-based sampling and bounded assignment, runs in expected time*

$$\mathrm{T}_{\mathrm{MS}}(n, N, \hat{\ell}) = \mathcal{O}\left(\overbrace{\frac{n}{p}\log\frac{n}{p} + \frac{D}{p}}^{\textit{local sorting}} + \overbrace{k\frac{\widetilde{N}}{p} + \frac{\tilde{n}}{p}\log p}^{\textit{merging}} + \overbrace{k\left(\alpha \log^2 p + k\sqrt[k]{p}\hat{\ell}\log p(1 + \beta\log\sigma)\right)}^{\textit{partitioning}}\right)$$
$$+ \underbrace{k \cdot \widetilde{\mathit{Exch}}(\mathcal{O}\left(\frac{\widetilde{N}}{p}\log\sigma\right), \mathcal{O}(\sqrt[k]{p}))}_{\textit{assignment + exchange}}.$$

[1] For string-based assignment, this naturally transfers from the atomic sorting case. For character-based assignment, we have to compensate a communication volume in $\mathcal{O}(r \log N)$. Assuming unique strings, we find $\hat{\ell} \geq \hat{d} = \Omega(\log n/\log\sigma)$ and $N \leq n\hat{\ell}$ and therefore $\log(N) = \mathcal{O}(\hat{\ell}\log\sigma)$. Since our bound for the number of (received) characters per PE contains an imbalance of at least $r\hat{\ell}$ characters (which require an encoding of $\log\sigma$ bits each), the data exchange dominates the assignment also for character-assignment.

Algorithm 1 Approximate Distinguishing Prefix Computation.

```
Input: Local string array $S_i = [s_0, s_1, \dots, s_{n_i-1}]$ with $n_i = |\mathcal{S}_i|$ on each PE $i$.
1  distPrefix $\leftarrow [|s_0|, |s_1|, \dots, |s_{n_i-1}|]$
2  $l \leftarrow \left\lceil \frac{\log p}{\log \sigma} \right\rceil$                // current approx. prefix length
3  $\mathcal{C}_i \leftarrow \{0, \dots, n_i - 1\}$                // indices of candidate strings
4  while $\sum_i |\mathcal{C}_i| > 0$ do
5      $\mathcal{H}_i \leftarrow \{(j, \texttt{Hash}(s_j[0, l-1])) \mid j \in C_i \wedge |s_j| \geq l\}$      // hash prefix
6      $\mathcal{D}_i \leftarrow \texttt{FindDuplicates}(\mathcal{H}_i)$
7      for $j \in C_i \setminus \mathcal{D}_i$ do
8          distPrefix$[j] \leftarrow l$                // set length for unique prefixes
9      $\mathcal{C}_i \leftarrow \mathcal{D}_i$                // continue with duplicates
10     $l \leftarrow l \cdot (1 + \epsilon)$                // increase prefix length
Output: Array distPrefix storing for each string in $S_i$ its approximate distinguishing
        prefix length.
```

As we only draw $\mathcal{O}(k\sqrt[k]{p})$ local samples and receive $\sqrt[k]{p} - 1$ splitter strings for $k > 1$ as opposed to $\mathcal{O}(p)$ samples and $p - 1$ splitters in the single-level case we no longer have to compensate the mediocre scalability of the single-level partition phase with a huge amount of data and are capable of sorting small to medium sized inputs on a large number of PEs. Assuming the exchange primitive Exch$(h, \sqrt[k]{p})$ to run in $\mathcal{O}(\alpha\sqrt[k]{p} + \beta h)$ [2], we achieve a latency in $\mathcal{O}(\alpha k \sqrt[k]{p}) = o(\alpha p)$ at the cost of an k times higher communication volume. If we additionally assume $\hat{\ell} \leq N/(k^2 \sqrt[k]{p} p \log p)$ and $k \leq \log p/(2 \log\log p)$, we can state a simplified running time of multi-level MS in Corollary 8.

▶ **Corollary 8.** *With the above assumptions, we obtain a running time of multi-level MS in* $\mathcal{O}\left(\frac{N}{p}\log n + \alpha k\sqrt[k]{p} + \beta k \frac{N}{p} \log\sigma\right)$ *in expectation.*

4 Multi-Level Prefix Doubling Merge Sort

The distinguishing prefix of S is usually much smaller than the total number of characters N. In a distributed algorithm, we can use this property to reduce the communication volume by only exchanging the distinguishing prefixes. By doing so, instead of explicitly sorting the input strings we obtain the information on where to find the i^{th} smallest string of the input. This, however, is sufficient in many use cases where string sorting is used, e.g., for suffix sorting [16].

Bingmann et al. approximate the distinguishing prefix of each string by an upper bound in an iterative doubling process [8] using a distributed single-shot Bloom filter (dSBF) [27]. Conceptually, a distributed single-shot Bloom filter of size m is a bit array of the same size which is equally distributed over the p PEs. To insert an element e, the element is hashed to a random position $h(e)$ within the interval $[0, m)$ and the corresponding bit is set on the PE responsible for this position, where h denotes a random hash function. For querying whether an element e is contained in the filter, one simply has to check whether the corresponding bit at position $h(e)$ is set. This may result in a false positive result. Note that for duplicate detection in our setting, the bit array does not necessarily need to be materialized as a PE only has to report back whether it has received a hash value multiple times.

For the distinguishing prefix approximation, in each round prefixes with geometrically increasing length of the strings are hashed and globally checked for uniqueness of the hash values. If the hash value of a prefix with length d of string s is unique, we find $\text{DIST}(s) \leq d$ and s no longer needs to participate in the process. That way for each string s an approximate distinguishing prefix with length $\text{DIST}_{\approxeq}(s) \geq \text{DIST}(s)$ can be determined in expected $\mathcal{O}(\log(\hat{d}))$ rounds assuming a constant false positive probability of the Bloom filter. Algorithm 1 shows pseudo-code for this computation. This can be achieved with expected latency in $\mathcal{O}(\alpha p \log \hat{d})$ and expected bottleneck communication volume in $\mathcal{O}(n/p \log p) + o(D/p \log \sigma)$ [8].[2] By employing a k-level Bloom filter for duplicate detection which communicates along a k-dimensional grid, we generalize this approach to arbitrary levels of indirection.

▶ **Theorem 9.** *Using communication on a k-dimensional grid, performing at most $\hat{n}$ operations (insertions, queries) per PE on a dSBF of size $m \geq en$ can be done in time $\mathcal{O}\left(k\left(\alpha p^{1/k} + \beta \hat{n} \log \frac{mp}{n} + \hat{n} \log k\right)\right)$ in expectation and with probability $\geq 1 - 1/p^{\omega(1)}$ assuming the total number of operations $n = \omega(k^2 p^{1+1/k} \log p)$ and additionally $m = \text{poly}(n)$.*

A proof of Theorem 9 can be found in the full version [19]. A problem of using Bloom filters in the prefix doubling process is that the precondition on the overall number of operations required in Theorem 9 might not hold when more and more strings drop out of the process because their distinguishing prefixes have already been determined. We therefore switch to duplicate detection using (atomic) hypercube quicksort for sorting the hash values if there are too few strings left. Once the hash values are globally sorted, a scan over the local elements and two additional message exchanges for the first and last local hash values suffices to identify duplicates. These can then be reported back to their issuing PEs.

Combining duplicate detection using multi-level dSBF and sorting results in Theorem 10.

▶ **Theorem 10.** *For each string $s \in \mathcal{S}$ with $\text{DIST}(s) \geq \log p / \log \sigma$ an approximation $\text{DIST}_{\approxeq}(s)$ with $\mathbb{E}[\text{DIST}_{\approxeq}(s)] = \mathcal{O}(\text{DIST}(s))$ can be computed in time*

$$\mathcal{O}\left(\overbrace{\alpha k \sqrt[k]{p} \log \hat{d}}^{\text{latency}} + \overbrace{\beta k \left(\frac{n}{p} \log p + \frac{D}{p} \log \sigma\right)}^{\text{communication volume}} + \overbrace{k \frac{n}{p} \log k \log \log \sigma + \frac{D}{p}}^{\text{local work}}\right)$$

in expectation. We assume a balanced distribution of strings and their distinguishing prefixes, i.e., $\Theta(n/p)$ strings and $\Theta(D/p)$ per PE, and an overall number of strings $n = \mathcal{O}(\text{poly}(p))$. Additionally, we assume $n/p = \omega(k^2 \sqrt[k]{p} \log p \log \log p)$ and $k \leq \log p / (2 \log \log p)$.

A proof for Theorem 10 can be found in the full version [19].

Since we assume all strings to be unique, we can also bound the average distinguishing prefix length $D/n = \Omega(\log n / \log \sigma) = \Omega(\log p / \log \sigma)$. Hence, there are only few ($\leq n/\sigma$) strings with small distinguishing prefixes for which Theorem 10 does not yield an (expected) constant factor approximation of the actual distinguishing prefix. Therefore, we find the expected value of the sum $D_{\approx}$ of the approximate distinguishing prefixes determined with Theorem 10 to be in $\mathcal{O}(D)$. In conjunction with the assumed balanced distribution of D over the PEs, we find $\mathcal{O}(\widetilde{D}/p)$ with $\widetilde{D} = D + k^2 r \hat{d} p$ as an upper bound on the expected number of characters per PE in the multi-level merge sort algorithm when executed on the approximated

[2] The latency can be reduced by increasing the communication volume by a factor $\Theta(\log p)$ [8].

distinguishing prefixes only. Furthermore, since $\hat{d} = \Omega(\log n / \log \sigma) = \Omega(\log p / \log \sigma)$, and we therefore approximate $\hat{d}$ up to an expected constant factor, we find $\mathrm{T_{MS}} = (n, D, \hat{d})$, as an upper bound for the running time of the merge sort part of multi-level **p**refix **d**oubling **m**erge **s**ort (PDMS) in Theorem 11.

▶ **Theorem 11.** *Multi-level PDMS with $r = \sqrt[k]{p}$, using character-based sampling and assignment, and assuming the preconditions of Theorem 10, runs in expected time*

$$\underbrace{\mathcal{O}\left(\overbrace{\log \hat{d}(\alpha k \sqrt[k]{p})}^{\textit{add. latency}} + \overbrace{k\beta\frac{n}{p}\log p}^{\textit{add. comm. volume}} + \overbrace{k\frac{n}{p}\log k \log\log\sigma}^{\textit{add. local work}}\right)}_{\textit{prefix doubling}} + \mathrm{T_{MS}}(n, D, \hat{d}).$$

Note that the $\mathcal{O}(\beta k D/p \log\sigma + D/p)$ part of the running time of the prefix doubling process in Theorem 10 is subsumed by $\mathrm{T_{MS}}(n, D, \hat{d})$ and thus not explicitly stated in Theorem 11.

5 Experimental Evaluation

We now discuss the experimental evaluation of the following distributed-memory algorithms.

`MS`$_k$ Our new multi-level string merge sort with k levels of recursion, see Section 3.

`PDMS`$_k$ Our new multi-level doubling string merge sort including prefix approximation using grid-wise Bloom filter with k levels of recursion, see Section 4.

`RQuick+` `RQuick` [8] with our string-specific optimization, see Section 2.

The single-level variants `MS`$_1$, `PDMS`$_1$, and `RQuick` are implementations by Bingmann et al. [8] that we improved slightly. We also include the state-of-the-art shared-memory parallel algorithm pS^5 [7]. We ran pS^5 on an AMD Epyc Rome 7702P CPU with 64 cores (2 GHz base and 3.5 GHz boost frequency) equipped with 1024 GB DDR4 ECC RAM. All distributed-memory experiments were performed on SuperMUC-NG[3] consisting of 6336 nodes (792 nodes per island). Each node is equipped with two Intel Skylake Xeon Platinum 8174 CPUs (24 cores each, 3.1 GHz base frequency) and 96 GB RAM. Communication between nodes uses a 100 Gbit/s Omni-Path network. All algorithms are implemented in C++ and compiled with GCC 11.2.0 with flags `-O3` and `-march=native`. For interprocess communication, we use the MPI-wrapper KaMPIng [13] and Open MPI v4.0.7. Reported times are the average of five runs excluding the first iteration (MPI warm-up phase).

All variants use string-based regular sampling with a sampling factor of 2. We use `RQuick+` to sort the samples. For simplicity, we use a grid-wise group (string) assignment in the implementation of our multi-level algorithms. Here, the PEs are arranged in a $p' \times r$ grid and

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

Table 2 Characteristics of real-world data sets used for the strong scaling experiment.

	n	N	N/n	L/n	D/N	$\hat{\ell}$
CCrawl	2.13 G	100 G	46.98	31.27	0.726	1.04 M
Wiki	1.42 G	97.7 G	68.59	25.82	0.415	2.07 M
WikiText	0.97 G	81.7 G	84.21	25.27	0.336	2.07 M

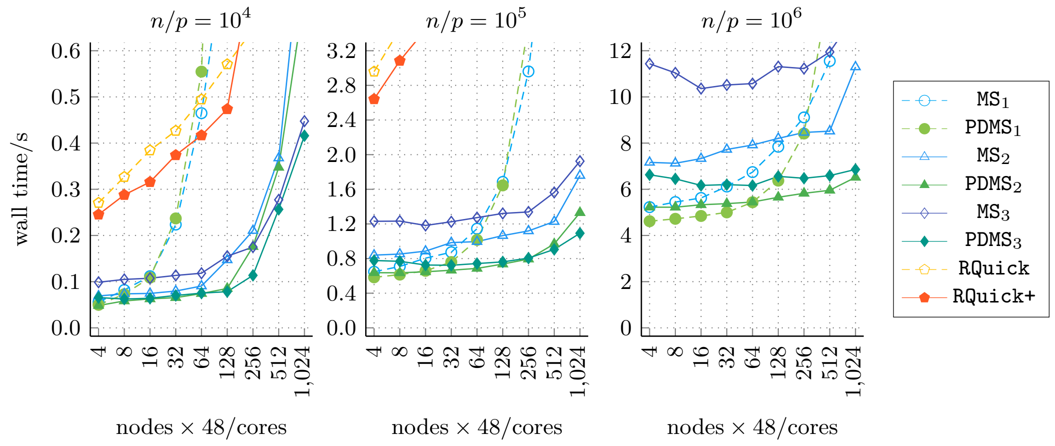

Figure 3 Average sorting times (weak scaling) using DNDATA with $\ell = 500$ and $D/N = 0.5$.

PEs exchange buckets only along the rows, i.e., PE i sends its bucket for subgroup j along its row to the PE in the j^{th} column. Also, for all-to-all exchanges a simple k-dimensional grid all-to-all is used which provides a latency in $\mathcal{O}(\alpha \sqrt[k]{p})$. LCP compression is used during string exchange phases for data sets where significant common prefixes can be expected.

Multi-level variants always ensure one group per node on the final level of sorting, i.e., k-level variants fall back to $k-1$ or $k-2$ levels 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.

For strong-scaling experiments, we use real-world data sets, see Table 2 for details. Each line of the data sets is interpreted as a string.

CommonCrawl (CCrawl) consists of the first 100 GB of WET files from the Sep./Oct. 2023 Common Crawl archive (`https://index.commoncrawl.org/CC-MAIN-2023-40/`). The WET format consists mostly of plain text with small meta data headers.

Wikipedia (Wiki) consists of a dump, from 2023-12-20, of all pages in the English Wikipedia in XML format *without* edit history (`https://dumps.wikimedia.org/enwiki/20231220/`).

WikipediaText (WikiText) is from dumps of WIKIPEDIA without any XML metadata.

For our weak-scaling experiments, we use the *DNGenerator* string generator [28, 8] to generate string sets with configurable D/N ratio (denoted by DNDATA). This allows us to influence the running times of local sorting, the effectiveness of LCP compression, and the length of distinguishing prefixes.

Additional results are given in the full version [19] for the following experiments.

5.1 Sorting Small to Medium Sized String Sets

In this weak-scaling experiment, we evaluate the algorithms on DNDATA with a string length of 500 characters, $D/N = 0.5$, and $n/p \in \{10^4, 10^5, 10^6\}$. See Figure 3 for the results. The largest input is near the realistic limit for this system with roughly 2 GB RAM per PE. We could not run the `RQuick` variants on it, due to their memory consumption.

The running times broadly confirm 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. For small inputs, adding a third level leads to further improvements from 256 nodes on. As expected, the improvement is most obvious

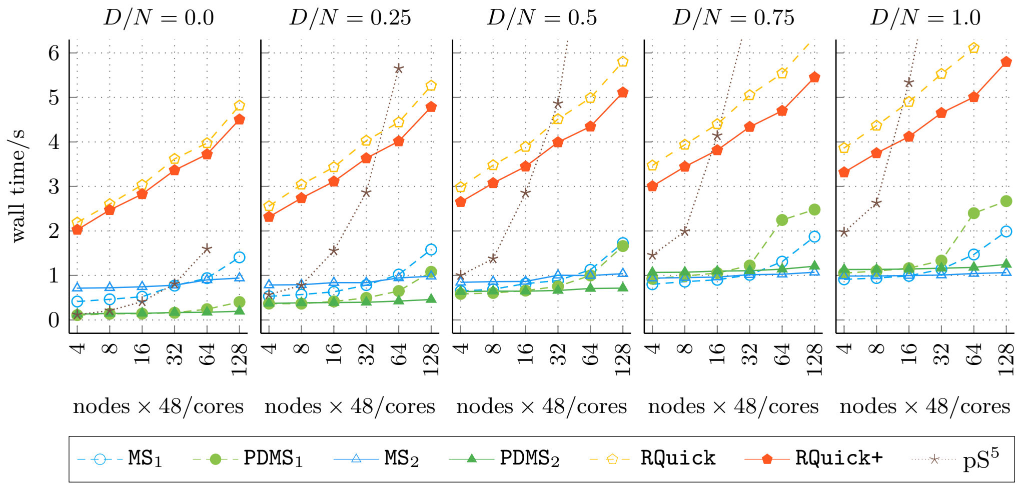

Figure 4 Average sorting times (weak scaling) using DNDATA with $\ell = 500$ and $n/p = 10^5$.

for the smallest inputs with $n/p = 10^4$. Here, the single-level algorithms scale roughly linearly with the number of PEs, as the running times approximately double for every doubling of p. For MS$_1$ the scaling behavior can mostly be attributed to the time required for partitioning, with $\Theta(p)$ samples on each PE needing to be sorted. For PDMS$_1$ a significant amount of time is also spent in the distinguishing prefix approximation. The `RQuick` variants perform significantly worse than our (multi-level) merge sort algorithms.

5.2 Influence of D/N Ratios

Our first experiment already shows that multi-level merge sort exhibits improved scaling properties. Now, we have a closer look at the influence of the D/N ratio on the running times of the different algorithms. As before, we use strings with a length of 500 characters and 10^5 strings per PE. Figure 4 shows the average running times. We evaluate `MS` and `PDMS` on one and two levels and the `RQuick` variants. Three-level variants are not part of this experiment as the additional level only yields clearly better performance for ≥ 512 compute nodes which we did not include in any further experiments due to computing budget constraints.

The influence of D/N ratio is clearly visible for MS$_k$ and PDMS$_k$, as variants with prefix approximation are superior on instances with a ratio up to 0.5. For ratios 0.75 and 1, the prefix approximation encompasses the whole string and is detrimental to the running time.

LCP compression is highly effective due to the nature of DNDATA inputs. As before, multi-level variants outperform their single-level counterparts, usually with a crossover point at 32 nodes. The gap seemingly increases for larger D/N ratios, e.g., for MS$_k$ on 128 nodes the speedups are 1.49 and 1.86 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 or 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.

The `RQuick` variants perform worse than the merge sort based algorithms. However, we can also see that for an increasing D/N ratio improving the algorithms by using LCP-aware merging pays off with `RQuick+` being up to 20% faster for $D/N = 1$ than plain `RQuick`.

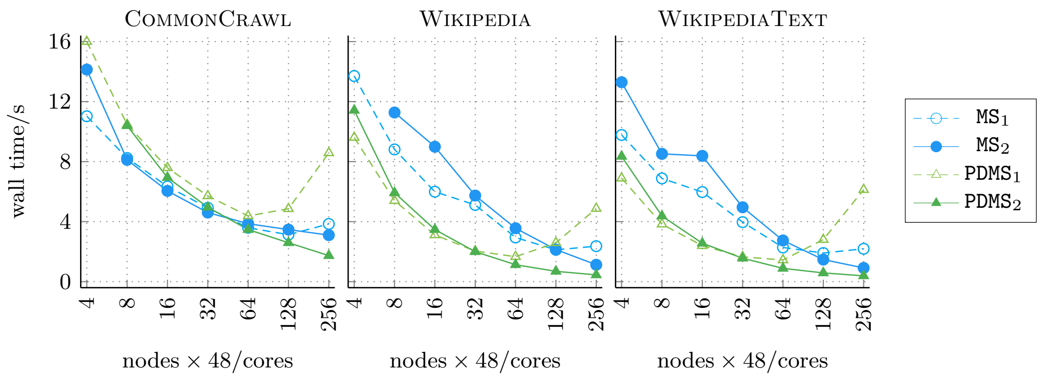

Figure 5 Average running times (strong scaling) using real-world inputs (see Table 2).

The running times of pS^5(*shared-memory* using 64 cores) are given for the input of all p processors. We could not run pS^5 on a node of SuperMUC-NG as these are only equipped with 96 GB RAM. With $D/N \geq 0.25$, we need 384 cores to match the performance of pS^5 and only 192 cores for $D/N \geq 0.5$. Data with $D/N \approx 0$ is difficult for distributed algorithms since the time spent in local sorting hardly compensates for the communication overheads and start-up latencies. Here, we only need 1536 cores to be on par with pS^5. This is of interest when string sorting is part of more complex distributed tasks as it indicates that sorting the strings directly on the distributed system is faster than transferring them to a sufficiently large shared-memory machine from a very modest number of cores on.

5.3 Evaluation on Real-World Data

To evaluate our algorithms on real-world data we use a strong-scaling experiment. On all three data sets, we can observe that the two-level algorithms scale better than their single-level counterparts and finally outperform them. On the largest PE configuration utilizing 256 compute nodes, $\texttt{PDMS}_2$ is the fastest algorithm on all three data sets. This highlights the usefulness of our multi-level approaches – even on very skewed real-world inputs. However, the differences in the running times are less pronounced. Also, $\texttt{PDMS}_2$ fails on CommonCrawl and $\texttt{MS}_2$ fails on Wikipedia when using only 4 compute nodes, due to imbalances of the data. This shows that string inputs can be inherently hard to partition.

Despite not all strings having the same length, we only report results for string-based sampling. Character-based sampling proved to be not feasible on these data sets. Preliminary experiments showed a significant imbalance on the second level, where the time spent in the merging phase varied by a large factor among the PEs.

6 Conclusion And Future Work

We demonstrate – in theory and practice – that string sorting can be scaled to a very large number of processors. Our best algorithm, a multi-level prefix-doubling merge sort, only requires internal work and communication volume close to the optimum (the total length of all distinguishing prefixes) per level. In practice, all our multi-level algorithms outperform their single-level counterparts on a wide range of inputs on up to 49 152 cores (from a modest number of cores on). This is especially important in scenarios where string sorting is part of a distributed application, i.e., it is not feasible to sort the data on a large shared-memory machine because of the transfer costs. Hence, we see our work as an important building block to enable more complex string-processing tasks at a massively parallel scale.

One problem we want to tackle in the future is suffix sorting based on sorting strings of equal length [16]. To this end, we plan to extend our algorithms to support *space-efficient* string sorting. Some inputs are highly compressible with lots of overlapping strings, e.g., the suffixes of a text with length n with a combined length of approximately $n^2/2$ can be represented using only n characters. However, due to the scarcity of main memory on most supercomputers, we cannot easily materialize *all* strings at the same time during sorting.

References

1 Michael Axtmann. *Robust Scalable Sorting.* PhD thesis, Karlsruhe Institute of Technology, Germany, 2021. doi:10.5445/IR/1000136621.

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

3 Michael Axtmann and Peter Sanders. Robust massively parallel sorting. In *ALENEX*, pages 83–97. SIAM, 2017. doi:10.1137/1.9781611974768.7.

4 Jon Louis Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *SODA*, pages 360–369. ACM/SIAM, 1997.

5 Timo Bingmann. *Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools.* PhD thesis, Karlsruhe Institute of Technology, Germany, 2018. doi:10.5445/IR/1000085031.

6 Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *Algorithmica*, 77(1):235–286, 2017. doi:10.1007/S00453-015-0071-1.

7 Timo Bingmann and Peter Sanders. Parallel string sample sort. In *ESA*, volume 8125 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2013. doi:10.1007/978-3-642-40450-4_15.

8 Timo Bingmann, Peter Sanders, and Matthias Schimek. Communication-efficient string sorting. In *IPDPS*, pages 137–147. IEEE, 2020. doi:10.1109/IPDPS47924.2020.00024.

9 Jonas Ellert, Johannes Fischer, and Nodari Sitchinava. LCP-aware parallel string sorting. In *Euro-Par*, volume 12247 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2020. doi:10.1007/978-3-030-57675-2_21.

10 Paolo Ferragina and Roberto Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999. doi:10.1145/301970.301973.

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

12 Torben Hagerup. Optimal parallel string algorithms: sorting, merging and computing the minimum. In *STOC*, pages 382–391. ACM, 1994. doi:10.1145/195058.195202.

13 D. Hespe, L. Hübner, F. Kurpicz, P. Sanders, M. Schimek, D. Seemaier, C. Stelz, and T. N. Uhl. KaMPIng: Flexible and (near) zero-overhead C++ bindings for MPI. *CoRR*, abs/2404.05610, 2024. doi:10.48550/arXiv.2404.05610.

14 Joseph F. JáJá, Kwan Woo Ryu, and Uzi Vishkin. Sorting strings and constructing digital search trees in parallel. *Theor. Comput. Sci.*, 154(2):225–245, 1996. doi:10.1016/0304-3975(94)00263-0.

15 Juha Kärkkäinen and Tommi Rantala. Engineering radix sort for strings. In *SPIRE*, volume 5280 of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2008. doi:10.1007/978-3-540-89097-3_3.

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

17 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.

18 Florian Kurpicz, Pascal Mehnert, Peter Sanders, and Matthias Schimek. Brief announcement: Scalable distributed string sorting. In *SPAA*, pages 375–377. ACM, 2024. doi:10.1145/3626183.3660256.

19 Florian Kurpicz, Pascal Mehnert, Peter Sanders, and Matthias Schimek. Scalable distributed string sorting. *CoRR*, abs/2404.16517, 2024. `doi:10.48550/arXiv.2404.16517`.

20 Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Comput.*, 19(10):1079–1103, 1993. `doi:10.1016/0167-8191(93)90019-H`.

21 Pascal Mehnert. Scalable distributed string sorting algorithms. Master's thesis, Karlsruher Institut für Technologie (KIT), 2024. `doi:10.5445/IR/1000170222`.

22 Pascal Mehnert and Matthias Schimek. mschimek/scalable-distributed-string-sorting. Software, European Research Council (ERC)(grant agreement No. 882500), swhId: `swh:1:dir:1d60272c5beeb821650519f3f0ce805434b705fa` (visited on 2024-07-09). URL: `https://github.com/mschimek/scalable-distributed-string-sorting`.

23 Bayyapu Neelima, Anjjan S. Narayan, and Rithesh G. Prabhu. String sorting on multi and many-threaded architectures: A comparative study. In *ICHPCA*, pages 1–6. IEEE, 2014.

24 Waihong Ng and Katsuhiko Kakehi. Merging string sequences by longest common prefixes. *IPSJ Digital Courier*, 4:69–78, 2008.

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

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

27 Peter Sanders, Sebastian Schlag, and Ingo Müller. Communication efficient algorithms for fundamental big data problems. In *IEEE BigData*, pages 15–23. IEEE Computer Society, 2013. `doi:10.1109/BIGDATA.2013.6691549`.

28 Matthias Schimek. Distributed string sorting algorithms. Master's thesis, Karlsruher Institut für Technologie (KIT), 2019. `doi:10.5445/IR/1000098432`.

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

30 Ranjan Sinha and Anthony Wirth. Engineering burstsort: Towards fast in-place string sorting. In *WEA*, volume 5038 of *Lecture Notes in Computer Science*, pages 14–27. Springer, 2008. `doi:10.1007/978-3-540-68552-4_2`.

31 Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. Merging multiple lists on hierarchical-memory multiprocessors. *J. Parallel Distributed Comput.*, 12(2):171–177, 1991. `doi:10.1016/0743-7315(91)90022-2`.