Brief Announcement: Scalable Distributed String Sorting

Florian Kurpicz
Karlsruhe Institute of Technology
Karlsruhe, Germany
kurpicz@kit.edu

Pascal Mehnert
Independent
Karlsruhe, Germany
pascalmehnert@posteo.de

Peter Sanders
Karlsruhe Institute of Technology
Karlsruhe, Germany
sanders@kit.edu

Matthias Schimek
Karlsruhe Institute of Technology
Karlsruhe, Germany
schimek@kit.edu

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 show good scaling behavior on a wide range of inputs on up to 49 152 cores. We achieve speedups of up to 5 over the current state-of-the-art distributed string sorting algorithms.

KEYWORDS

distributed-memory algorithm; string sorting; multi-level algorithm; communication-efficient algorithm

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 [9, 15]. 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 comparison of whole strings. Instead, they inspect the *distinguishing prefixes* of the strings, i.e., the characters needed to establish the global ordering, ideally only once. The sum of the lengths of all distinguishing prefixes is usually denoted by D. The lower bound for sequential string sorting based on character comparisons is $\Omega(n \log n + D)$ with existing algorithms matching this bound [4].

1.1 Related Work

There exists extensive research on string sorting in the sequential setting. For a systematic overview, we refer to [5, 14, 19]. We focus on parallel sorting algorithms. Let n be the total number of strings and N be the total number of characters. For the PRAM model, there are (comparison-based) algorithms solving the string sorting problem in $O(n \log n + N)$ work and $O(\log^2 n / \log \log n)$ time [13]. For integer alphabets, there exists an algorithm with $O(N \log \log N)$ work and running time in $O(\log N / \log \log N)$ [11]. The work of any string sorting algorithm can be made to depend on D instead of N by increasing its time complexity in the PRAM model [8].

While the problem has been extensively studied in the sequential and (shared-memory) parallel setting, we are only aware of the following results in the distributed-memory setting. Bingmann et al. present the two state-of-the-art distributed string sorting algorithms [7]. The first one follows the standard distributed-memory merge sort scheme (local sorting, partitioning, message exchange, and merging). Every step is augmented with string-specific optimizations, e.g., LCP-compression and LCP-aware merging [6, 7, 17], see Section 2. The second algorithm is more communication-efficient, as it only sorts approximations of the distinguishing prefixes using the first algorithm. They also adapt the distributed hypercube quicksort algorithm [2, 3] to variable-length keys (without string-related optimizations). These algorithms improve the first dedicated distributed string sorting algorithm by Fischer et al. [10].

However, the algorithms are only efficient for very small or large inputs, as 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.

2 PRELIMINARIES

Machine Model. We assume $1 \dots p$ processing elements (PEs) in a network allowing single-ported point-to-point communication. Exchanging h bits between two PEs requires $\alpha + \beta h$ time, where α accounts for the message start-up overhead and β quantifies the time to exchange one bit.

String Properties. The input to our algorithms is a string array $S = [s_0, s_1, \ldots, s_{n-1}]$ consisting of $n = |S|$ unique strings over an alphabet Σ of size σ. By $N = \|S\|$, we denote the total number of characters in S. The ℓ-prefix of a string s are the first ℓ characters of s. The *longest common prefix (LCP)* of two strings $s \neq t$ is the prefix of s with length $\mathrm{lcp}(s, t) = \arg\min s[i] \neq t[i]$. 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 $\hat{\ell}$ and $\hat{d}$, we denote the length of the longest string and the longest distinguishing prefix, respectively. 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., $|S_i| = \Theta(n/p)$, $\|S_i\| = \Theta(N/p)$.

Building Blocks. We make use of an *r-way LCP loser tree* to merge r sorted sequences of in total m strings augmented with LCP values in $O(m \log r + D)$ time [6]. 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 [7]. (Robust) Hypercube Quicksort (RQuick) is a sorting algorithm initially proposed for atomic distributed sorting [1, 3]. It communicates all data (at least) a logarithmic number of times but has a latency in $O(\alpha \log^2 p)$ and is therefore especially suited for small inputs.

3 MULTI-LEVEL STRING SORTING

Our *multi*-level merge sort (MS) approach adapts algorithmic ideas of Axtmann et al. for multi-level *atomic* sorting [2, 3] to string sorting. It works by recursively splitting PEs into multiple groups each of which solves an independent (string) sorting problem. By using k levels of recursion, we can reduce the latency to about $r = \sqrt[k]{p}$ as PEs now only communicate strings with $O(r)$ instead of p PEs directly. This comes at the cost of increasing the communication volume by a factor k as we communicate all strings in each level. The algorithm is split into a one-time initialization and a recursive phase which is invoked k times. Fig. 1 provides an overview of the main steps of the multi-level merge sort scheme which are discussed in more detail in the following.

Initialization: On each PE i, the input $\mathcal{S}_i$ is sorted locally using a string sorting algorithm [5, 8, 11].

Recursion: This phase comprises three main steps.

1) Distributed Partitioning: Globally determine $r - 1$ splitters and locally partition the data into r buckets. Use *string-based* or *character-based* partitioning to either balance the number of strings or characters per group (see full version [16] for details).

2) String Assignment and Exchange: On each PE, assign the strings in bucket j to PEs in group j such that no PE sends or receives more than $O(r)$ messages. Once this assignment is computed, exchange strings and LCP values using direct messaging.

3) Local LCP-aware Merging: On each PE, merge the received string sequences with LCP values to obtain locally sorted string arrays which form the input for the subsequent recursive step.

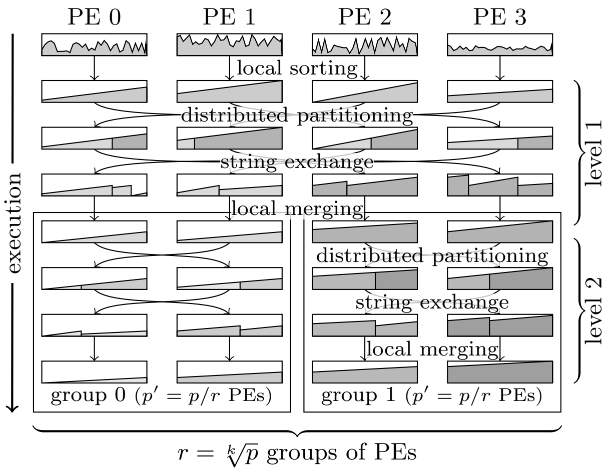

Figure 1: Overview of multi-level string sorting.

With *character-based* partitioning, message assignment, and assuming $\hat{\ell} \leq N/(k^2 p \sqrt[k]{p} \log p)$, it can be shown that the number of characters per PE remains in $O(N/p)$ over the course of the algorithm. Further assuming that an all-to-all operation where each PE exchanges at most $O(N/p)$ characters with $O(r)$ other PEs is possible in $O(\alpha r + \beta N/p \log \sigma)$ (see [2]), we achieve an overall latency in $O(\alpha k \sqrt[k]{p}) = o(\alpha p)$ at the cost of a k times higher communication volume compared to the single-level algorithm for $k \leq \log p / (2 \log \log p)$.

Theorem 3.1. *With the above assumptions, multi-level MS runs in* $O\left(\frac{N}{p} \log n + \alpha k \sqrt[k]{p} + \beta k \frac{N}{p} \log \sigma\right)$ *time in expectation.*

We refer to the full paper [16] for a more detailed analysis of the algorithm's running time stated in Theorem 3.1.

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 [15]. Bingmann et al. approximate the distinguishing prefix of each string by an upper bound in an iterative doubling process [7] using a distributed single-shot Bloom filter [18]. In each round, they hash prefixes with geometrically increasing length of the strings and globally check for uniqueness of the hash values. If the hash value of a prefix with length d of string s is unique, we find $\textsc{dist}(s) \leq d$ and s no longer needs to participate in the process. By introducing k-level Bloom filters for duplicate detection, we generalize this approach to arbitrary levels of indirection and achieve an expected overall latency in $O(\alpha k \sqrt[k]{p} \log \hat{d})$ and expected communication volume in $O(k(n/p \log p + D/p \log \sigma))$ for large enough n/p and $k \leq \log p / (2 \log \log p)$. Note that the term $O(kD/p \log \sigma)$ is dominated by the subsequent multi-level merge sort performed on the approximated distinguishing prefixes only.

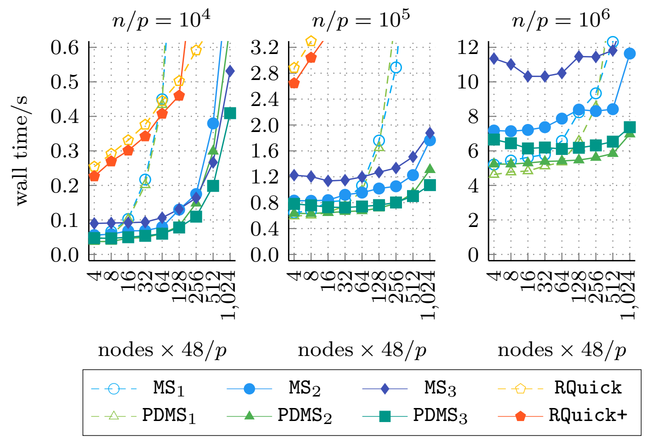

Figure 2: Average times for `RQuick`, `MS`$_k$, and `PDMS`$_k$ (weak scaling) using DNDATA inputs with $\ell = 500$ and $D/N = 0.5$.

4 EXPERIMENTAL EVALUATION

The source code is publicly available at https://github.com/pmehnert/distributed-string-sorting/. All algorithms are implemented in C++. For interprocess communication, we use the MPI-wrapper KaMPIng [12]. The experiments were performed on SuperMUC-NG.

We evaluated the following algorithms: Our new multi-level string merge sort `MS`$_k$ with k levels. Our new multi-level prefix doubling string merge sort `PDMS`$_k$ including prefix approximation using Bloom filters with k levels. String sorting enabled `RQuick` [7] (and `RQuick+` with further string-specific optimizations) is also part of the evaluation. Note that `MS`$_1$, `PDMS`$_1$, and `RQuick` are the implementations by Bingmann et al. [7] that we improved slightly.

Fig. 2 shows the running times of three weak-scaling experiments on up to 49 152 cores (1024 compute nodes) where we evaluate the algorithms on strings of length 500 with a D/N ratio of 0.5 with $\{10^4, 10^5, 10^6\}$ strings per PE. We use *DNGenerator* [7] to generate the inputs. Further experiments can be found in the full paper [16]. The experiments reveal the expected relation between input size and scaling behavior of the algorithms. Two-level merge sort significantly outperforms the single-level version on all input sizes for sufficiently large values of p. As expected, the improvement is most obvious 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 small inputs, adding a third level leads to further improvements from 256 cores on. The different variants of `RQuick` perform significantly worse than our (multi-level) merge sort algorithms. We were not able to run any `RQuick` variant on the largest data set due to memory consumption.

5 CONCLUSION AND FUTURE WORK

We demonstrate—in theory and practice—that string sorting can be scaled to a very large number of processors. Our fastest algorithm, a multi-level prefix doubling merge sort, only requires 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 up to 49 152 cores (from a modest number of cores on). Scalable algorithms are especially important when string sorting is part of a more complex distributed application where it is not feasible to sort the data on a large shared-memory machine due to the transfer costs. Hence, we see our work as a building block to enable more complex string-processing tasks at a massively parallel scale. Next, we plan to improve the robustness of our algorithms by handling short and long strings separately and only merging them in the end.

ACKNOWLEDGMENTS

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

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882500).

REFERENCES

[1] M. Axtmann. 2021. *Robust Scalable Sorting.* Ph. D. Dissertation. Karlsruhe Institute of Technology, Germany.
[2] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz. 2015. Practical Massively Parallel Sorting. In *SPAA*. ACM, 13–23. https://doi.org/10.1145/2755573.2755595
[3] M. Axtmann and P. Sanders. 2017. Robust Massively Parallel Sorting. In *ALENEX*. SIAM, 83–97. https://doi.org/10.1137/1.9781611974768.7
[4] J. L. Bentley and R. Sedgewick. 1997. Fast Algorithms for Sorting and Searching Strings. In *SODA*. ACM/SIAM, 360–369.
[5] T. Bingmann. 2018. *Scalable String and Suffix Sorting: Algorithms, Techniques, and Tools.* Ph. D. Dissertation. Karlsruhe Institute of Technology, Germany.
[6] T. Bingmann, A. Eberle, and P. Sanders. 2017. Engineering Parallel String Sorting. *Algorithmica* 77, 1 (2017), 235–286. https://doi.org/10.1007/S00453-015-0071-1
[7] T. Bingmann, P. Sanders, and M. Schimek. 2020. Communication-Efficient String Sorting. In *IPDPS*. IEEE, 137–147. https://doi.org/10.1109/IPDPS47924.2020.00024
[8] J. Ellert, J. Fischer, and N. Sitchinava. 2020. LCP-Aware Parallel String Sorting. In *Euro-Par (Lecture Notes in Computer Science, Vol. 12247)*. Springer, 329–342. https://doi.org/10.1007/978-3-030-57675-2_21
[9] P. Ferragina and R. Grossi. 1999. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *J. ACM* 46, 2 (1999), 236–280. https://doi.org/10.1145/301970.301973
[10] J. Fischer and F. Kurpicz. 2019. Lightweight Distributed Suffix Array Construction. In *ALENEX*. SIAM, 27–38. https://doi.org/10.1137/1.9781611975499.3
[11] T. Hagerup. 1994. Optimal parallel string algorithms: sorting, merging and computing the minimum. In *STOC*. ACM, 382–391. https://doi.org/10.1145/195058.195202
[12] D. Hespe, L. Hübner, F. Kurpicz, P. Sanders, M. Schimek, D. Seemaier, C. Stelz, and T. N. Uhl. 2024. KaMPIng: Flexible and (Near) Zero-overhead C++ Bindings for MPI. *CoRR* abs/2404.05610 (2024). https://doi.org/10.48550/ARXIV.2404.05610 arXiv:2404.05610
[13] J. F. JáJá, K. W. Ryu, and U. Vishkin. 1996. Sorting Strings and Constructing Digital Search Trees in Parallel. *Theor. Comput. Sci.* 154, 2 (1996), 225–245. https://doi.org/10.1016/0304-3975(94)00263-0
[14] J. Kärkkäinen and T. Rantala. 2008. Engineering Radix Sort for Strings. In *SPIRE (Lecture Notes in Computer Science, Vol. 5280)*. Springer, 3–14. https://doi.org/10.1007/978-3-540-89097-3_3
[15] J. Kärkkäinen, P. Sanders, and S. Burkhardt. 2006. Linear work suffix array construction. *J. ACM* 53, 6 (2006), 918–936. https://doi.org/10.1145/1217856.1217858
[16] Florian Kurpicz, Pascal Mehnert, Peter Sanders, and Matthias Schimek. 2024. Scalable Distributed String Sorting. arXiv:2404.16517
[17] W. Ng and K. Kakehi. 2008. Merging string sequences by longest common prefixes. *IPSJ Digital Courier* 4 (2008), 69–78. https://doi.org/10.2197/ipsjdc.4.69
[18] P. Sanders, S. Schlag, and I. Müller. 2013. Communication efficient algorithms for fundamental big data problems. In *IEEE BigData*. IEEE Computer Society, 15–23. https://doi.org/10.1109/BIGDATA.2013.6691549
[19] R. Sinha and A. Wirth. 2008. Engineering Burstsort: Towards Fast In-Place String Sorting. In *WEA (Lecture Notes in Computer Science, Vol. 5038)*. Springer, 14–27. https://doi.org/10.1007/978-3-540-68552-4_2