Communication-Efficient String Sorting

Timo Bingmann, Peter Sanders, Matthias Schimek

Karlsruhe Institute of Technology, Karlsruhe, Germany

{*bingmann,sanders*}*@kit.edu, matthias_schimek@gmx.de*

Abstract—There has been surprisingly little work on algorithms for sorting strings on distributed-memory parallel machines. We develop efficient algorithms for this problem based on the multi-way merging principle. These algorithms inspect only characters that are needed to determine the sorting order. Moreover, communication volume is reduced by also communicating (roughly) only those characters and by communicating repetitions of the same prefixes only once. Experiments on up to 1280 cores reveal that these algorithm are often more than five times faster than previous algorithms.

Index Terms—distributed-memory algorithm, string sorting, communication-efficient algorithm

I. INTRODUCTION

Sorting, i.e., establishing the global ordering of n elements $s_0, \ldots s_{n-1}$, is one of the most fundamental and most frequently used subroutines in computer programs. For example, sorting is used for building index data structures like B-trees, inverted indices or suffix arrays, or for bringing data together that needs to be processed together. Often, the elements have string keys, i.e., variable length sequences of characters, or, more generally, multiple subcomponents that are sorted lexicographically. For example, this is the case for sorted arrays of strings that facilitate fast binary search, for prefix B-trees [1], [2], or when using string sorting as a subroutine for suffix sorting (i.e., the problem of sorting all suffixes of *one* string). Using string sorting for suffix sorting can mean to directly sort the suffixes [3], or to sort shorter strings as a subroutine. For example, the difference cover algorithm [4] is theoretically one of the most scalable suffix sorting algorithms. An implementation with large difference cover could turn out to be the most practical variant but it requires an efficient string sorter for medium length strings.

Sorting strings using conventional *atomic* sorting algorithms (that treat keys as indivisible objects) is inefficient since comparing entire strings can be expensive and has to be done many times in atomic sorting algorithms. In contrast, efficient string sorting algorithms inspect most characters of the input only once during the entire sorting process and they inspect only those characters that are needed to establish the global ordering. Let D denote the *distinguishing prefix*, which is the minimal number of characters that need to be inspected. Efficient sequential string sorting algorithms come close to the lower bound of $\Omega(D)$ for sorting the input. When characters are black boxes, that can only be compared but not further inspected, we get a lower bound of $\Omega(D + n \log n)$. Such comparison-based string sorting algorithms will be the main focus of our theoretical analysis. Our implementations also include some optimizations for integer alphabets.

Surprisingly, there has been little previous work on parallel string sorting using p processors, a.k.a. processing elements (PEs). Here one would like to come close to time $\mathcal{O}(D/p)$ – at least for sufficiently large inputs. Our extensive previous work [5]–[7] concentrates on shared-memory algorithms. However, for large data sets stored on nodes of large compute clusters, distributed-memory algorithms are needed. While in principle the shared-memory algorithms could be adapted, they neglect that *communication volume* is the limiting factor for the scalability of algorithms to large systems [8]–[10].

The present paper largely closes this gap by developing such communication-efficient string sorting algorithms. After discussing preliminaries (Section II) and further related work (Section III), we begin with a very simple baseline algorithm based on Quicksort that treats strings as atomic objects (Section IV). We then develop genuine string sorting algorithms that are based on multi-way mergesort that was previously used for parallel and external sorting algorithms [11]–[15]. The data on each PE is first sorted locally. It is then partitioned into p ranges so that one range can be moved to each PE. Finally, each PE merges the received fragments of data. The appeal of multi-way merging for communication-efficient sorting is that the local sorting exposes common prefixes of the local input strings. Our *Distributed String Merge Sort* (MS) described in Section V exploits this by only communicating the length of the common prefix with the previous string and the remaining characters. The LCP values also allow us to use the multiway LCP-merging technique we developed in [7] in such a way that characters are only inspected once. In addition, we develop a partitioning scheme that takes the length of the strings into account in order to achieve better load balancing.

Our second algorithm *Distributed Prefix-Doubling String Merge Sort* (PDMS) described in Section VI further improves communication efficiency by only communicating characters that may be needed to establish the global ordering of the data. The algorithm also has optimal local work for a comparison-based string sorting algorithm. See Theorem 5 for details. The key idea is to apply our communication-efficient duplicate detection algorithm [10] to geometrically growing prefixes of each string. Once a prefix has no duplicate anymore, we know that it is sufficient to transmit only this prefix.

In Section VII, we present an extensive experimental evaluation. We observe several times better performance compared to previous approaches; in particular for large machines and strings with high potential for saving communication bandwidth. Section VIII concludes the paper including a discussion of possible future work.

II. Preliminaries

Our input is an array $\mathcal{S} := [s_0, ..., s_{n-1}]$ of n strings with total length N. Sorting $\mathcal{S}$ amounts to permuting it so that a lexicographical order is established. A string s of length $\ell = |s|$ is an array $[\,s[0], \ldots, s[\ell-2], 0\,]$ where 0 is a special end-of-string character outside the alphabet.[1] String arrays are usually represented as arrays of pointers to the beginning of the strings. Thus, entire strings can be moved or swapped in constant time. The first ℓ characters of a string are its length ℓ *prefix*. Let $\text{LCP}(s_1, s_2)$ denote the length of the *longest common prefix* (LCP) of s_1 and s_2. For a sorted array of strings $\mathcal{S}$, we define the corresponding *LCP array* $[\bot, h_1, h_2, \ldots, h_{|\mathcal{S}|-1}]$ with $h_i := \text{LCP}(s_{i-1}, s_i)$. The string sorting algorithms we describe here produce the LCP-array as additional output. This is useful in many applications. For example, it facilitates building a search tree that allows searching for a string pattern s in time $\mathcal{O}(|s| + \log n)$ [1], [2].

The distinguishing prefix length $\text{DIST}(s)$ of a string s is the number of characters that must be inspected to differentiate it from all other strings in the set $\mathcal{S}$. We have $\text{DIST}(s) = \max_{t \in \mathcal{S}, t \neq s} \text{LCP}(s, t) + 1$. The sum of the distinguishing prefix lengths D is a lower bound on the number of characters that must be inspected to sort the input.

Our model of computation is a distributed-memory machine with p PEs. Sending a message of m bits from one PE to another PE takes time $\alpha + \beta m$ [16], [17].[2] Analyzing the communication cost of our algorithms is mostly based on plugging in the cost of well-known collective communication operations. When h is the maximum amount of data sent or received at any PE, we get $\mathcal{O}(\alpha \log p + \beta h)$ for broadcast, reduction, and all-to-all broadcast (a.k.a. gossiping). For personalized all-to-all communication we have a tradeoff between low communication volume (cost $\mathcal{O}(\alpha p + \beta h)$) and low latency (cost $\mathcal{O}(\alpha \log p + \beta h \log p)$); e.g., [17].

Table I summarizes the notation, concentrating on the symbols that are needed for the result of the algorithm analysis.

A. Sequential String Sorting for the Base Case

In [6] an extensive evaluation of sequential string sorting algorithms is given in which a variant of *MSD String Radix Sort* has been found to be among the fastest algorithms on many data sets. We are using this algorithm for our implementations. This recursive algorithm considers subproblems where all strings have a common prefix length ℓ. The strings are then partitioned based on their $(\ell+1)$-st character. The recursion stops when the subproblem contains less then σ strings. This takes time $\mathcal{O}(D)$ (not counting the base case problems). These small subproblems are sorted using Multikey Quicksort [18]. This is an adaptation of Quicksort to strings that needs expected time $\mathcal{O}(D + n \log n)$. Our implementation, in turn, uses LCP insertion sort [6] as a based case for constant size inputs.

Symbol	Meaning
n	total number of input strings
N	total number of input characters
σ	alphabet size
D	total distinguishing prefix size
$\hat{n}$	max. number of strings on any PE
$\hat{N}$	max. number of characters on any PE
$\hat{D}$	max. number of distinguishing prefix characters on any PE
$\hat{\ell}$	length of the longest input string
$\hat{d}$	length of the longest distinguishing prefix
p	number of processing elements (PEs)
α	message startup latency
β	time per bit of communicated data

This algorithm has complexity $\mathcal{O}(D + n^2)$. Putting these components together leads to a base case sorter with cost $\mathcal{O}(D + n \log \sigma)$. We have modified these algorithms so that they produce an LCP array as part of the output at no additional cost. The modified implementations have been made available as part of the tlx library [19].

Our study [6] identifies several other efficient sequential string sorters. Which ones are best depends on the characteristics of the input. For example, for large alphabets and skewed inputs strings, sample sort [5] might be better. The resulting asymptotic complexity for such purely comparison-based algorithms is $\mathcal{O}(D + n \log n)$ which represents a lower bound for string sorting based on character comparisons.

B. Multiway LCP-Merging

We are using our *LCP loser tree* [7]. This is a generalization of the binary merging technique proposed by Ng and Kakehi [20] building on the (atomic) loser tree data structure [21].

A K-way (atomic) loser tree (a.k.a. tournament tree) is a binary tree with K leaves. Each leaf is associated with one current element of a sorted sequence of objects – initially the smallest element in that sequence. This current element is passed up the tree. Internal nodes store the larger of the elements passed up to them (the loser) and pass up the smaller element (the winner) to the next level. The element passed up by the root is the globally smallest element. This element is output in each step. The sequence corresponding to the winner's leaf is advanced to the next element. The data structure invariant of the loser tree can be reestablished in logarithmic time by repairing it step by step while going upwards from the winner's leaf to the root. This also determines the next element to be output.

Loser trees are adapted to strings by associating each sorted sequence with its LCP array. Moreover, internal nodes store the intermediate LCP length of the compared strings. The output is the sorted sequence representing all input sequences plus the corresponding LCP array. The number of character comparisons for multiway LCP-merging of m strings is bounded by $m \log K + \Delta L$ where ΔL is the total increment of the LCP-array entries of the input strings. Embedded into a string sorting algorithm this leads to total complexity $\mathcal{O}(D + n \log n)$ for sorting n strings.

---

[1] Our algorithms can also be adapted to the case without 0-termination where the inputs specify string lengths instead.

[2] Usually, the unit is a different one, e.g., machine words. Here we use bits in order to be able to make more precise statements with respect to the number of characters to be communicated.

C. Distributed Multiway Mergesort

A starting point for our algorithms is the distributed-memory mergesort algorithm by Fischer and Kurpicz [15] as a subroutine for suffix array construction. The data is first sorted locally using a sequential string sorting algorithm. It is then partitioned globally by $p-1$ splitter strings $f_1, \ldots, f_{p-1}$ such that PE i gets all the strings s with $f_i < s \leq f_{i+1}$ (with f_0 denoting an "infinitely" small string and f_p an "infinitely" large one). Fischer and Kurpicz choose these splitters based on a deterministic sampling technique where each PE chooses $p-1$ samples equidistantly from its sorted local input. After gathering the samples on PE 0, the splitters are chosen equidistantly from the globally sorted sample. They use an ordinary (not LCP-aware) loser tree for merging strings.

III. More Related Work

This paper is based on the master's thesis of Matthias Schimek [22]. There has been intensive work on sequential string sorting. Refer to [6], [23], [24] for an overview of results and comparative studies. There are very fast PRAM algorithms for sorting strings of total length N with work $\mathcal{O}(N \log N)$ ($\mathcal{O}(N \log \log N)$ for integer characters), e.g., [25], [26]. Note that our results need only *linear* work in the (possibly much smaller) distinguishing prefix length D rather than in the total input size N. The previous algorithms use a doubling technique similar to the one used by Manber and Myers [27] for suffix sorting: Use integer sorting to build lexicographic names of substrings with a length that doubles in every iteration. The doubling technique in our PDMS-algorithm is much simpler – it only requires hashing of prefixes of the strings. Also, doubling is not inherent in this technique but only one special case. To achieve better approximation of distinguishing prefix lengths one can also uses smaller multipliers. Neelima et al. [28] study string sorting on GPUs.

IV. Parallel String Sorting Based on Atomic Parallel Quicksort

This section serves two purposes. We describe a simple parallel string sorting algorithm whose analysis can serve as a basis for comparing it with the more sophisticated algorithms below. We also use this algorithm as a subroutine in the others.

This algorithm – hQuick– is a rather straightforward adaptation of an atomic sorting algorithm based on a Quicksort variant introduced in [29]. We therefore only outline it, focusing on the changes needed for string sorting. Let $d = \lfloor \log p \rfloor$. The algorithm employs only $2^d \geq p/2$ PEs which it logically arranges as a d-dimensional hypercube. The algorithm starts by moving each input string to a random hypercube node. hQuick proceeds in d iterations. In iteration $i = d, \ldots, 1$, the remaining task is to sort the data within i-dimensional subcubes of this hypercube. To establish the loop invariant for the next iteration, a pivot string s is determined as a good approximation of the median of the strings within each subcube. This is done using a special kind of tree reduction. One subcube will then work on the strings $\leq s$ and one works on the strings $> s$. A tie breaking scheme enforces that the pivot is unique. When
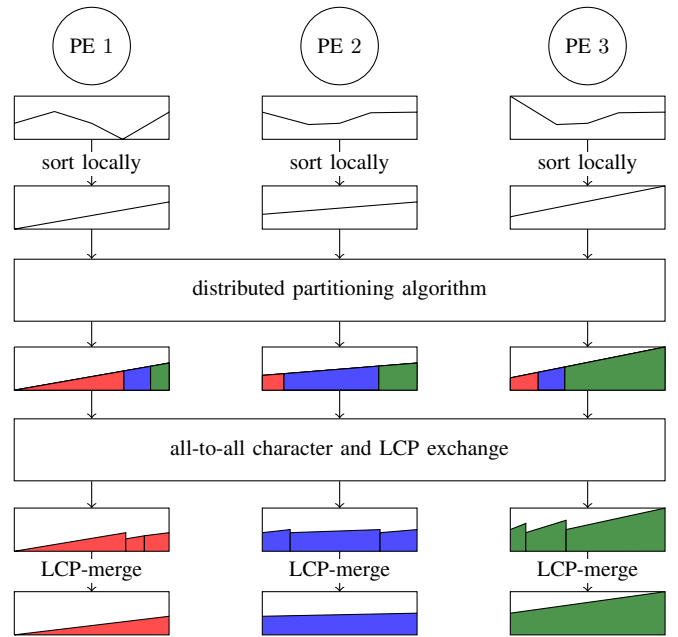

Fig. 1. Standard distributed mergesort scheme, which we augmented in every step with string-specific optimizations.

the loop has terminated, the remaining problem is to locally sort the data on one PE.

Theorem 1. *With the notation from Table I, Algorithm* hQuick *needs local work* $\mathcal{O}(n\hat{\ell}/p \log n)$, *latency* $\mathcal{O}(\alpha \log^2 p)$, *and bottleneck communication volume* $\mathcal{O}((\hat{N} + n\hat{\ell}/p \log p + \hat{\ell} \log^2 p) \log \sigma)$ *bits in expectation.*[3]

Before turning to the analysis, we interpret what this theorem says. Algorithm hQuick is not very efficient because all the data is moved a logarithmic number of times and because using an approximation of the median as a pivot only balances the *number* of strings but not their total length. Also, string comparisons do not exploit information on common prefixes that may be implicitly available. On the other hand, the algorithm has only polylogarithmic latency which makes it a good candidate for sorting small inputs.

Proof. The term $\hat{N} \log \sigma$ is due to the initial random placement of the input. We assume here that afterwards the pivot selection ensures (in expectation) that in each iteration, each PE works on $\mathcal{O}(n/p)$ strings in expectation. See [29] for the details which transfer from the atomic algorithm. We make the conservative assumption that each string incurs work and communication volume $\hat{\ell}$ in each iteration. Similarly, we assume that local sorting takes time $\mathcal{O}(n\hat{\ell}/p \log n)$. The term $\hat{\ell} \log \sigma \log^2 p$ in the communication volume stems from the reduction operation that, in each iteration, needs to transmit up to $\hat{\ell}$ characters along a reduction tree of logarithmic depth. □

---

[3]This conservative bound ensues if n/p strings of length $\hat{\ell}$ are concentrated on a single PE. Randomization makes this unlikely. However, the ordering of the strings might enforce such a distribution at the end. Hence, it may be possible to improve the work and communication bounds by a factor $\log p$.

V. Distributed String Merge Sort

Algorithm MS is based on the standard mergesort scheme (see Figure 1) for distributed memory but we need to augment it in every step with string-specific optimizations. Each PE i starts with a string array $\mathcal{S}_i$ as input and the goal is to sort the union $\mathcal{S}$ of all inputs such that afterwards strings of PE i are larger than those on PE $i-1$, smaller than those on PE $i+1$, and locally sorted. We also output the LCP array. The MS algorithm follows the following four steps (see Fig. 2 for an illustration):

1) Sort the string set locally using a sequential string sorting algorithm which also saves the local LCP array (see Section II-A for details).
2) Determine $p-1$ global splitters $f_1, \ldots, f_{p-1}$ such that PE i gets bucket b_i containing all strings s with $f_i < s \le f_{i+1}$ assuming sentinels $f_0 = -\infty$ and $f_p = +\infty$.
3) Perform an all-to-all exchange of string and LCP data, optionally applying LCP compression.
4) Merge the p received sorted subsequences locally with our efficient LCP-aware loser tree.

The following subsections discuss details of these steps.

A. String-Based or Character-Based Partitioning

When determining the splitters f_i in step 2) the goal is to balance the result among all PEs. In the case of strings, this can mean to balance the number of strings or the number of characters each PE receives. For MS we thus devised a *string*-based and an alternative *character*-based partitioning step to determine splitters. Both assume a given oversampling parameter v. Furthermore, because we sort the local string sets $\mathcal{S}_i$ in step 1), we can use *regular* sampling [30], [31] instead of randomly selecting samples.

String-based partitioning performs the following steps:

1) Each PE i chooses v evenly spaced samples $\mathcal{V}_i$ from its strings $\mathcal{S}_i$. Assuming $|\mathcal{S}_i|$ is divisible by $v+1$, then we can choose the strings $\mathcal{S}_i[\omega j - 1]$ with $\omega = |\mathcal{S}_i|/(v+1)$ for $j = 1, \ldots, v$.
2) The pv samples are globally sorted into array $\mathcal{V}$. Then, $p-1$ splitters $f_i = \mathcal{V}[vj - 1]$ are selected for $i = 1, \ldots, p-1$. This sorting and selection can be implemented trivially by sending all samples to one PE, or using distributed sorting and selection algorithms. In both cases the complete set of splitters is communicated to all PEs.

To prove that buckets are well-balanced, we first show a lemma reformulating the density of samples in a subsequence.

Lemma 1.1. *For $i = 1, \ldots, p$ let $\mathcal{S}'_i = \{s \in \mathcal{S}_i \mid a \le s \le b\}$ be an arbitrary contiguous subarray of $\mathcal{S}_i$. If $|\mathcal{S}'_i \cap \mathcal{V}_i| = k$, then $|\mathcal{S}'_i| \le (k+1)\omega$ with $\omega = |\mathcal{S}_i|/(v+1)$.*

Proof. If $k = 0$, then all elements of $\mathcal{S}'_i$ are fully contained between two consecutive sample elements of $\mathcal{V}_i$, thus $|\mathcal{S}'_i| < \omega$. If $k = 1$, then let x be the element in $\mathcal{S}'_i \cap \mathcal{V}_i$ and we have $\mathcal{S}'_i = \mathcal{S}'_{i,<} \cup \{x\} \cup \mathcal{S}'_{i,>}$. For $\mathcal{S}'_{i,<}$ and $\mathcal{S}'_{i,>}$ the case $k = 0$ applies and thus we have $|\mathcal{S}'_i| \le (\omega - 1) + 1 + (\omega - 1) \le 2\omega$.

Step 1: sort locally with LCP array output

Step 2: sample regularly: { alpha, snow, organ }, select splitters: { alpha, organ }, and find splits.

Step 3: all-to-all exchange with LCP compression

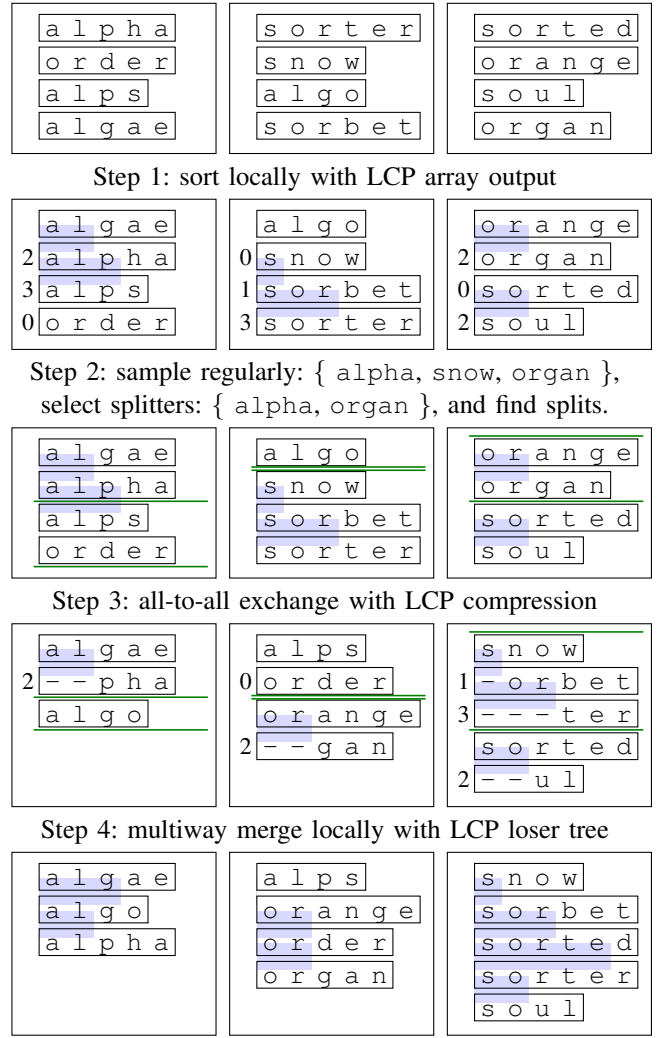Step 4: multiway merge locally with LCP loser tree

Fig. 2. Steps of Algorithm MS shown on example strings. The small numbers and shaded blue area after Step 1 are the calculated LCPs. The green lines after Step 2 are the two splitting positions. In Step 3, characters shown as "–" are omitted due to LCP compression.

If $k \ge 2$, then we can split $\mathcal{S}'_i$ into $(k+1)$ parts such that the first and last contain at most $\omega - 1$ elements and the others exactly $\omega - 1$ between the splitters. Thus $|\mathcal{S}'_i| \le k((\omega - 1) + 1) + (\omega - 1) \le (k+1)\omega$. $\qquad\square$

Theorem 2. *All buckets b_j contain at most $\frac{n}{p} + \frac{n}{v}$ elements.*

Proof. Let $\mathcal{B}^j_i := \{s \in \mathcal{S}_i \mid f_{j-1} < s \le f_j\}$ be the elements in bucket b_j on PE i, $\mathcal{V}^j_i := \{s \in \mathcal{V}_i \mid f_{j-1} < s \le f_j\}$ the samples therein, and $v^j_i := |\mathcal{V}^j_i|$ their number. By definition $|\mathcal{B}^j_i \cap \mathcal{V}_i| = v^j_i$ and by applying Lemma 1.1 we get $|\mathcal{B}^j_i| \le (v^j_i + 1)\omega$. Since f_{j-1} and f_j are globally separated by $v-1$ samples, $\sum_{i=1}^p v^j_i = v$. We can now bound b_j by summing over all PEs: $|b_j| = \sum_{i=1}^p |\mathcal{B}^j_i| \le \sum_{i=1}^p (v^j_i + 1)\omega = \omega(v + p) = \frac{|\mathcal{S}_i|}{(v+1)}(v + p) = \frac{|\mathcal{S}|}{(v+1)p}(v + p) < \frac{|\mathcal{S}|}{vp}(v + p) = \frac{n}{v} + \frac{n}{p}$. $\qquad\square$

For character-based partitioning, we have to switch our focus from the string arrays $\mathcal{S}_i$ to the underlying character arrays $\mathcal{C}(\mathcal{S}_i)$. For simplicity assume that $|\mathcal{C}(\mathcal{S}_i)|$ is divisible by $v+1$

and let $\omega' = |\mathcal{C}(\mathcal{S}_i)|/(v+1)$. We furthermore assume $\hat{\ell} \leq \omega'$, otherwise strings are very long and too few to draw v samples.

For character-based partitioning, each PE i again chooses v sample strings $\mathcal{V}_i$ from its string set, but this time the strings are regularly sampled such that $\mathcal{C}(\mathcal{S}_i)$ is evenly spaced between them. For this chose the first strings *starting at or following* the characters at ranks $j\omega' - 1$ in $\mathcal{C}(\mathcal{S}_i)$ for $j = 1, \ldots, v$. This can be efficiently calculated by keeping an array containing the length of each string while sorting $\mathcal{S}_i$ in Step 1. As before, the pv sample strings are globally sorted into $\mathcal{V}$ and $p - 1$ splitters f_i are selected.

The following lemma states that at most an imbalance of $\hat{\ell}$ is introduced due to the shift to the next string. Using it we can then show a character-based lemma equivalent to Lemma 1.1.

Lemma 2.1. *If $\hat{\ell} \leq \omega'$, then the splitters $\mathcal{V}_i$ selected by character-based partitioning split $\mathcal{S}_i$ into $v + 1$ non-empty local buckets $\mathcal{S}_i^j$. The number of characters in each bucket $\mathcal{S}_i^j$ is at most $\omega' + \hat{\ell}$.*

Proof. We have $\mathcal{S}_i^j := \{s \in \mathcal{S}_i \mid \mathcal{V}_i[j-1] < s \leq \mathcal{V}_i[j]\}$ assuming $\mathcal{V}_i$ is sorted. Since the initially chosen equally spaced characters have a distance of $\omega' \geq \hat{\ell}$, the splitters in $\mathcal{V}_i$ are distinct and thus each $\mathcal{S}_i^j$ contains at least the splitter. On the other hand, the furthest possible shift from the character-based split point to the next string is $\hat{\ell}$, hence each bucket contains at most $\omega' + \hat{\ell}$ characters. $\qquad\square$

Lemma 2.2. *For $i = 1, \ldots, p$ let $\mathcal{S}_i' = \{s \in \mathcal{S}_i \mid a \leq s \leq b\}$ be an arbitrary contiguous subarray of $\mathcal{S}_i$. If $|\mathcal{S}_i' \cap \mathcal{V}_i| = k$, then $|\mathcal{C}(\mathcal{S}_i')| \leq (k+1)(\omega' + \hat{\ell})$ with $\omega' = |\mathcal{C}(\mathcal{S}_i)|/(v+1)$.*

Proof. Let $\hat{\omega} := \omega' + \hat{\ell}$. If $k = 0$, then all elements of $\mathcal{S}_i'$ are fully contained in one of the sets $\mathcal{S}_i'$, hence $|\mathcal{C}(\mathcal{S}_i')| \leq \hat{\omega}$ by Lemma 2.1. The remaining proof for $k = 1$ and $k \geq 2$ is analogous to the proof of Lemma 1.1 with $\hat{\omega}$ taking the role of ω due to Lemma 2.1. $\qquad\square$

With the two lemmas we can reiterate Theorem 2 to bound the size of buckets for characters-based partitioning.

Theorem 3. *All buckets b_j contain at most $\frac{N}{p} + \frac{N}{v} + (p+v)\hat{\ell}$ characters.*

Proof. Applying Lemma 2.2 with the same arguments as in the proof of Theorem 2 yields $|b_j| = \sum_{i=1}^{p} |\mathcal{B}_i^j| \leq \sum_{i=1}^{p} (v_i^j + 1)(\omega' + \hat{\ell}) = (\omega' + \hat{\ell})(v + p) = \frac{|\mathcal{C}(\mathcal{S}_i)|}{(v+1)}(v+p) + \hat{\ell}(v+p) = \frac{|\mathcal{C}(\mathcal{S})|}{(v+1)p}(v+p) + \hat{\ell}(v+p) < \frac{|\mathcal{C}(\mathcal{S})|}{vp}(v+p) + \hat{\ell}(v+p) = \frac{N}{v} + \frac{N}{p} + \hat{\ell}(v+p)$. $\qquad\square$

B. Data Exchange

Lemma 3.1. *The data exchange phase of Algorithm MS (Step 3) with LCP compression has bottleneck communication volume $\mathcal{O}((\hat{N} + p\hat{\ell})\log\sigma + \hat{n}\log\hat{\ell})$ bits when character-based sampling is used.*

Proof. The term $\hat{n}\log\hat{\ell}$ stems from the LCP values. $\hat{N}$ is an obvious upper bound for the string data on each PE. By Theorem 3, character-based sampling with $v = \Theta(p)$ samples

per PE guarantees $\mathcal{O}(N/p + p\hat{\ell}) \leq \mathcal{O}(\hat{N} + p\hat{\ell})$ characters on the receiving side. $\qquad\square$

Note that LCP compression is of no help in establishing non-trivial worst case bounds on the communication volume. The reason is that local LCP values may be very short even if every string has long LCPs with strings located on other PEs. The situation is even worse with string based sampling since it may happen that some PE gets n/p very long strings.

C. Overall Analysis of Algorithm MS

We now analyze Algorithm MS with character-based sampling and using algorithm hQuick for sorting the sample.

Theorem 4. *With the notation from Table I, Algorithm MS can be implemented to run using local work $\mathcal{O}(\hat{N} + \hat{n}\log n + p\hat{\ell}\log n)$, latency $\mathcal{O}(\alpha p)$, and bottleneck communication volume $\mathcal{O}((\hat{N} + p\hat{\ell}\log p)\log\sigma)$ bits.*

Once more, we first interpret this result. When the input is balanced with respect to the number of strings and number of characters (i.e., $\hat{n} = \mathcal{O}(n/p)$ and $\hat{N} = \mathcal{O}(N/p)$), and if it is sufficiently large (i.e., $N = \Omega(p^2\hat{\ell})$), we get an algorithm that is as efficient as we can expect from a method that communicates all the data. Hence, for large inputs this is a big improvement over hQuick. In the worst case, we have no advantage from LCP compression even if $D \ll N$. However, by using character-based sampling, we achieve load balancing guarantees. Using parallel sorting of the sample saves us a factor p in the minimal efficient input size compared to [15] since a deterministic sampling approach needs samples of *quadratic* size.

Proof. After local sorting (in time $\mathcal{O}(\hat{D} + \hat{n}\log\hat{n})$), each PE samples $v = \Theta(p)$ strings locally which have maximal length $\hat{\ell}$. These $\mathcal{O}(p^2)$ strings are then sorted using algorithm hQuick. By Theorem 1, this incurs local work $\mathcal{O}(p\hat{\ell}\log n)$, latency $\mathcal{O}(\alpha\log^2 p)$, and communication volume $\mathcal{O}(p\hat{\ell}\log\sigma\log p)$ bits. The splitter strings are then gossiped to all PEs. This contributes latency $\alpha\log p$ and communication volume $p\hat{\ell}\log\sigma$ bits.

The local data is then partitioned in time $\mathcal{O}(p\log(\hat{n})\hat{\ell})$ using binary search. By Theorem 3, each of the resulting $p \times p$ messages has size $\mathcal{O}(\hat{N}/p + p\hat{\ell})$. Moreover, no PE receives more than $\mathcal{O}(N/p + p\hat{\ell})$ characters. Hence, the ensuing all-to-all data exchange contributes latency $\mathcal{O}(\alpha p)$ and communication volume $\mathcal{O}((\hat{N} + p\hat{\ell})\log\sigma)$ bits. Finally, the received data is merged in time $\mathcal{O}(N/p\log p)$. Adding all these terms and making some simplifications yields the claimed result. $\qquad\square$

VI. DISTRIBUTED PREFIX-DOUBLING STRING MERGE SORT

We now refine algorithm MS so that it can take advantage of the case $D \ll N$. The idea is to find an upper bound for the distinguishing prefix length of each input string. We do this as a Step $(1 + \varepsilon)$ after local sorting (Step 1) but before determining splitters (Step 2). The required global communication is expensive but it pays off in theory and in Section VII we will see that this algorithm also works well in practice. We not only save in communication volume in

Step 3 but knowing the distinguishing prefix lengths also aids (character-based) splitter determination in finding splitters that balance the actual amount of work that needs to be done.

Algorithm PDMS does not solve exactly the same problem as Algorithm MS. Whereas MS permutes the strings into sorted order, PDMS only computes the permutation without completely executing it – it only permutes the distinguishing prefixes (and can indicate the origin of these prefixes). Note that some applications do not need the complete information; for example, when string sorting is used as a subroutine in suffix sorting [3], [4], [15]. The locally available information also suffices to build a sorted array of the strings for pattern search or to build a search tree [1], [2]. The resulting search data structures support many operations (e.g., counting matches) based on local information.

Theorem 5. *With the notation from Table I, Algorithm PDMS can be implemented to run using local work $\mathcal{O}(\hat{D} + \hat{n}\log n)$, latency $\mathcal{O}(\alpha p \log \hat{d})$, and bottleneck communication volume*

$$(1 + \varepsilon)\hat{D}\log\sigma + \mathcal{O}(\hat{n}\log p + p\hat{d}\log\sigma\log p)$$

bits, in expectation and assuming that the all-to-all communication in Step 3 incurs bottleneck communication volume h when the maximum sum of local message sizes is h.[4] The latency can be reduced to $\mathcal{O}(\alpha(p + \log p\log\hat{d}))$ when increasing the term $\hat{n}\log p$ in the communication volume to $\hat{n}\log^2 p$.

Again, we interpret the result before proving it. Compared to Algorithm MS, we now achieve local work and bottleneck communication volume that is close to a worst case lower bounds if the input is sufficiently large and sufficiently evenly distributed over the PEs. The price we pay is a logarithmic factor in the latency which correspondingly increases the input size that is required to achieve overall efficiency.

Proof. We only discuss the differences to Algorithm MS and refer to Theorem 4 for remaining details. The work for Step 1 is $\mathcal{O}(\hat{D} + \hat{n}\log\hat{n})$ using any efficient sequential comparison-based string sorting algorithm.

The analysis of Step 2 is similar to that in Theorem 4 except that we are now using samples and splitter strings of length at most $\hat{d}$. Also, we do not use the total string lengths as the basis for sampling but the length of the approximated distinguishing prefix lengths. Using Algorithm hQuick on the sample now incurs local work $\mathcal{O}(p\hat{d}\log p)$, latency $\mathcal{O}(\alpha\log^2 p)$, and communication volume $\mathcal{O}(p\hat{d}\log\sigma\log p)$ bits.

Refer to Theorem 6 for the analysis of Step $1 + \varepsilon$.

The all-to-all exchange in Step 3 incurs latency $\alpha\log p$ and communication volume $(1 + \varepsilon/2)\hat{D}\log\sigma$ for those strings whose prefix length has been successfully approximated within a factor $1 + \varepsilon/2$. We add an additional volume $\varepsilon\hat{D}/2 \cdot \log\sigma$ to account for the $o(1)$ term in the analysis of Step $1 + \varepsilon$ and for the prefix lengths that are overestimated due to false positives in the duplicate detection. This works out by setting

<hr>

[4]It seems to be an open problem whether there is an algorithm achieving this. We make this assumption in order to be able to concisely work out the impact of the tuning parameter ϵ.

an appropriate false positive rate $\approx 1/2$. Overall, we calculate bottleneck communication volume $(1 + \varepsilon)\hat{D}$ for Step 3.

In Step 4, the received data is merged in time $\mathcal{O}(D/p\log p)$. Adding all these terms and making some simplifications yields the claimed result. $\qquad\square$

A. Approximating Distinguishing Prefix Lengths

Determining whether a prefix of an input string is a distinguishing prefix is equivalent to finding out whether there are any duplicates of it. Duplicate detection is a well studied problem. There is no known deterministic solution to the problem apart from communicating the entire prefix. However, we can use randomization. We calculate hash values (a.k.a. fingerprint) of the prefixes to be considered and determine which fingerprints are unique. The corresponding prefixes are now certain to be distinguishing prefixes. Errors are on the safe side – two fingerprints may be accidentally identical which would lead to falsely declaring their corresponding prefixes to be non-distinguishing. By judiciously choosing the fingerprint size, by compressing fingerprints, and by iterating the process with a short fingerprint in the first iteration and a long fingerprint in the second iteration (where only few candidates remain), we can do duplicate detection using only $\mathcal{O}(\log p)$ bits of communication volume for each prefix to be checked [10].

To approximate the distinguishing prefix length of a string s, we start from some initial guess ℓ_s and then let the guessed length grow geometrically by a factor $(1 + \varepsilon)$ in each iteration. With our default value of $\varepsilon = 1$ we get *prefix doubling* which we use to name sorting algorithm PDMS. Fig. 3 shows an illustration of PDMS and we fill in the remaining details in the proof of the following theorem.

Theorem 6. *With the notation from Table I, distinguishing prefix lengths can be found using local work $\mathcal{O}(\hat{D})$, latency $\mathcal{O}(\alpha p \log \hat{d})$, and bottleneck communication volume $\mathcal{O}(\hat{n}\log p)$ bits, in expectation. The latency term can be reduced to $\mathcal{O}(\alpha\log p\log\hat{d})$ at the price of increasing the term $\hat{n}\log p$ in the communication volume to $\hat{n}\log^2 p$.*

Proof. Determining approximate distinguishing prefix sizes starts with an initial guess $\ell = \Theta(\lceil\log p/\log\sigma\rceil)$ bits and iteratively multiplies ℓ by a factor $1 + \varepsilon/2$ taking all strings into account whose first ℓ characters are not proven to be unique yet. Hence, the overall number of iterations is $\log_{1+\varepsilon/2}\hat{d} = \mathcal{O}(\log\hat{d})$. Each iteration incurs a latency of αp and communication volume $\mathcal{O}(\log p)$ for each string that has not been eliminated yet. Summing the communication volume for a particular string s with distinguishing prefix length $\text{DIST}(s)$ yields communication volume $\mathcal{O}(\log p) + o(\text{DIST}(s))$. Overall, we account communication volume $\mathcal{O}(\hat{n}\log p)$.

The above discussion assumes that the all-to-all communication of fingerprints is done by directly delivering them to their destination. We can reduce the latency of this all-to-all to $\alpha\log p$ by delivering the data indirectly, e.g., using a hypercube based all-to-all [17]. This increases the communication volume by a factor $\log p$ however. $\qquad\square$

Step 1: sort locally (with LCP array output)

algae	algo	orange
alpha	snow	organ
alps	sorbet	sorted
order	sorter	soul

Step 1 + ε (depth 1): approximate distinguishing prefix

algae	**a**lgo	**o**range
alpha	**s**now	**o**rgan
alps	**s**orbet	**s**orted
order	**s**orter	**s**oul

Step 1 + ε (depth 2): using distributed duplicate detection

algae	**al**go	**or**ange
alpha	**sn**ow	**or**gan
alps	**so**rbet	**so**rted
order	**so**rter	**so**ul

Step 1 + ε (depth 4): repeat until all prefixes

algae	**algo**	**oran**ge
alpha	**sn**ow	**orga**n
alps	**sorb**et	**sort**ed
order	**sort**er	**soul**

Step 1 + ε (depth 8): are unique.

algae	**algo**	**oran**ge
alpha	**sn**ow	**orga**n
alps	**sorb**et	**sorted**
order	**sorter**	**soul**

Step 2: sample regularly: { alph, sn, orga },
select splitters: { alph, orga }, and find splits.

algae	**algo**	**oran**ge
alpha	**sn**ow	**orga**n
alps	**sorb**et	**sorted**
order	**sorter**	**soul**

Step 3: all-to-all exchange with LCP compression

algae	alps	snow
--pha	order	-orbet
algo	orange	---ter
	--gan	sorted
		--ul

Step 4: multiway merge locally with LCP loser tree

algae	alps	snow
algo	orange	sorbet
alpha	order	sorted
	organ	sorter
		soul

Fig. 3. Steps of Algorithm PDMS shown on example strings. String prefixes marked blue are duplicates, while red prefixes are unique. In Step 3, only the approximate distinguishing prefix is transmitted, the omitted characters are marked gray.

Theorem 6 may also be useful outside string sorting algorithms in order to analyze the input with respect to its distinguishing prefixes. A simple application might be to choose an algorithm for suffix sorting based on approximations of D – when D/n is small, we can use string sorting based algorithms, otherwise, more sophisticated algorithms are better. We might also use this information to choose the difference cover size in an implementation of the DC algorithm [4].

B. Average Case Analysis of Algorithm PDMS and Beyond

Neither Algorithm MS, nor Algorithm PDMS can profit from LCP compression in the worst case. This is because there may be inputs where all input strings have only very short local LCP values but very long distinguishing prefix lengths due to similar strings on other PEs. In order to understand why LCP-compression is nevertheless useful in practice, we now outline an average case analysis. To keep things simple let us first consider random bit strings where 0s or 1s are chosen independently with probability $1/2$. Among n strings uniformly distributed over p PEs, the distinguishing prefix lengths will be about $\log n$. Locally, the LCP values will be about $\log(n/p)$. Hence, LCP compression saves us $\log(n/p)$ bits per string. Thus, only about $\log n - \log(n/p) = \log p$ bits actually need to be transferred.

Therefore, for random inputs, the communication volume of Algorithm PDMS is dominated by the $\mathcal{O}(\log n)$ bits communicated for LCP-values, string IDs, etc. We now outline how to obtain an algorithm beyond PDMS that reduces this cost by data compression. Local LCP-values "on-the-average" only differ by a constant requiring $\mathcal{O}(1)$ bits to communicate them using a combination of difference encoding and variable-bit-length codes. We also cannot afford to transfer string IDs ($\log n$ bits) or long associated information. However, we can still view this as a sorting algorithm with a similar API as Algorithm PDMS: To reconstruct an output string s and its associated information, a PE remembers from which PE i string s was received and at which position j in the array of strings received from PE i it was located. PE i can then be queried for the suffix and associated information of s. This complication also explains why the logarithm of the number of permutations of the inputs ($\log n! \approx n \log n$, i.e., about $\log n$ bits per input string), is *not* a lower bound for our view on the sorting problem – we do not compute a full permutation but only a data structure that allows querying this permutation at a cost of $\mathcal{O}(\log n)$ bits of communication per query.

Let us now turn to more general input models. Assume now that the characters come from a random source with entropy H. Distinguishing prefix sizes are now about $\log_{1/H} n$ and LCP values are $\log_{1/H} n/p$ so that only $\log_{1/H} p$ characters need to be transmitted. By additionally compressing those, we can get down to about $\log p$ bits once more. This argument not only works for random sources where characters are chosen independently but also, e.g., for Markov chains.

VII. Experiments

A. Inputs

We now present experiments based on two large real world data sets (COMMONCRAWL and DNAREADS) and a synthetic data set (D/N) with tunable ratio $r = D/N$; see [22] for details. In Section VII-E, we summarize results for further inputs. The i-th string from the D/N input consists of an appropriate number of repetitions of the first character of Σ followed by a base σ encoding of i followed by further characters to achieve the desired string length (500 in the numbers reported here). Value $r = 0$ means that i begins immediately and $r = 1$ means that i stands at the end of the string.

Input COMMONCRAWL consists of the concatenation of the first 200 files from CommonCrawl (2016-40)[5] This data consists of 82 GB of text dumps of websites. Each line of these files represents one input string. Here we have $D/N = 0.68$, alphabet size 242, average line length 40 characters, and average LCP 23.9 (60 % of each line).

As an example for small alphabets and bioinformatics applications, we consider input DNAREADS which consists of reads of DNA sequences from the 1000 Genomes Project[6]. Sorting such inputs is relevant as preprocessing for genome assembly or for building indices on the raw data. We concatenated the low coverage whole genome sequence (WGS) reads from the lexicographically smallest six samples (HG00099, HG00102, HG00107, HG00114, HG00119, HG00121). We extracted the reads from the FastQ files discarding quality information and concatenated them in lexicographic order of their accession identifier. Reads containing any other character than A, C, G, and T were dropped. The resulting data set contains 125 GB base pairs in 1.27 million read strings with an alphabet size of four and $D/N = 38$ %. On average a DNA read line is 98.7 base pairs long with an average LCP of 29.2 (30 % of each line). Compared to the COMMONCRAWL input, DNAREADS has a considerably lower percentage of characters in the LCPs and distinguishing prefix. This is due to the DNA base pair sequences being more random than text on web pages.

The COMMONCRAWL and DNAREADS data was split such that each PE gets about the same number of characters. The strings from D/N are randomly distributed over the PEs.

B. Hardware

All experiments were performed on the distributed-memory cluster ForHLR I. This cluster consists of 512 compute nodes. Each of these nodes contains two 10-core Intel Xeon processors E5-2670 v2 (Sandy Bridge) with a clock speed of 2.5 GHz and have 10×256 KB of level 2 cache and 25 MB level 3 cache. Each node possesses 64 GB of main memory and an adapter to connect to the InfiniBand 4X FDR interconnect.[7] Intel MPI Library 2018 was used as implementation of the MPI standard. All programs were compiled with GCC 8.2.0 and optimization

[5]commoncrawl.s3.amazonaws.com/crawl-data/CC-MAIN-2016-40/wet.paths.gz
[6]www.internationalgenome.org/data-portal/sample
[7]wiki.scc.kit.edu/hpc/index.php/ForHLR_-_Hardware_and_Architecture

flags -O3 and -march=native. We create one MPI process on each available core, i.e., hardware threads are not used.

C. Algorithms

We compare the following algorithms:

FKmerge is the distributed multiway string mergesort of Fischer and Kurpicz [15]; see also Section II-C. This is the only distributed-memory string sorter that we could find.

hQuick: As an example for a fast atomic sorting algorithm we use our adaptation of distributed hypercube atomic Quicksort by Axtmann et al. [29]. We adapted its original implementation [32] by replacing point-to-point communication of fixed length with point-to-point communication of variable length. See also Section IV.

MS-simple is our Distributed String Merge Sort from Section V with no LCP related optimizations at all.

MS is our Algorithm MS with LCP compression.

PDMS-Golomb is an implementation of our Distributed Prefix-Doubling String Merge Sort (PDMS) from Section VI that uses Golomb coding for communicating hash values.

PDMS is the same as PDMS-Golomb except without using Golomb compression for communicating hash values.

All these algorithms use string based sampling. Our C++ implementations of all these algorithms is available as open source from https://github.com/mschimek/distributed-string-sorting.

D. Results

Fig. 4 shows a weak scaling experiment with 250 MB of data on each core using different ratios D/N. As expected, the atomic sorting algorithm hQuick is outclassed by the string sorting algorithms. The only previous distributed string sorter FKmerge works well up to 320 cores (16 nodes) but scalability then quickly deteriorates. We attribute this to high communication costs and a bottleneck due to centralized sorting of samples. This is also consistent with the increasing communication volume observed in the lower part of the plot that shows communication volume. Already the most simple variant of our MS algorithm MS-simple consistently outperforms FKmerge and hQuick, and scales reasonably well – the execution time with 64 nodes (1280 cores) is only about twice that of the execution time with 2 nodes (40 cores). This ratio gets smaller as D/N grows since there is more internal work to be done. Enabling the LCP optimization in MS yields further consistent improvements. Not surprisingly, the advantages get more pronounced with increasing D/N since this implies longer common prefixes. The prefix doubling algorithms (PDMS-Golomb and PDMS) yield a further large improvement when D/N is not too large because we get a large saving in communication volume. For large D/N, the prefix doubling yields no useful bounds on the distinguishing prefix length and hence the moderate overheads for finding these values makes the algorithms slightly slower than MS. Using or not using Golomb compression of hash values is of little consequence on running time. We see on the lower part of the figure that it also has little influence on communication volume. Apparently, the communication overhead for finding
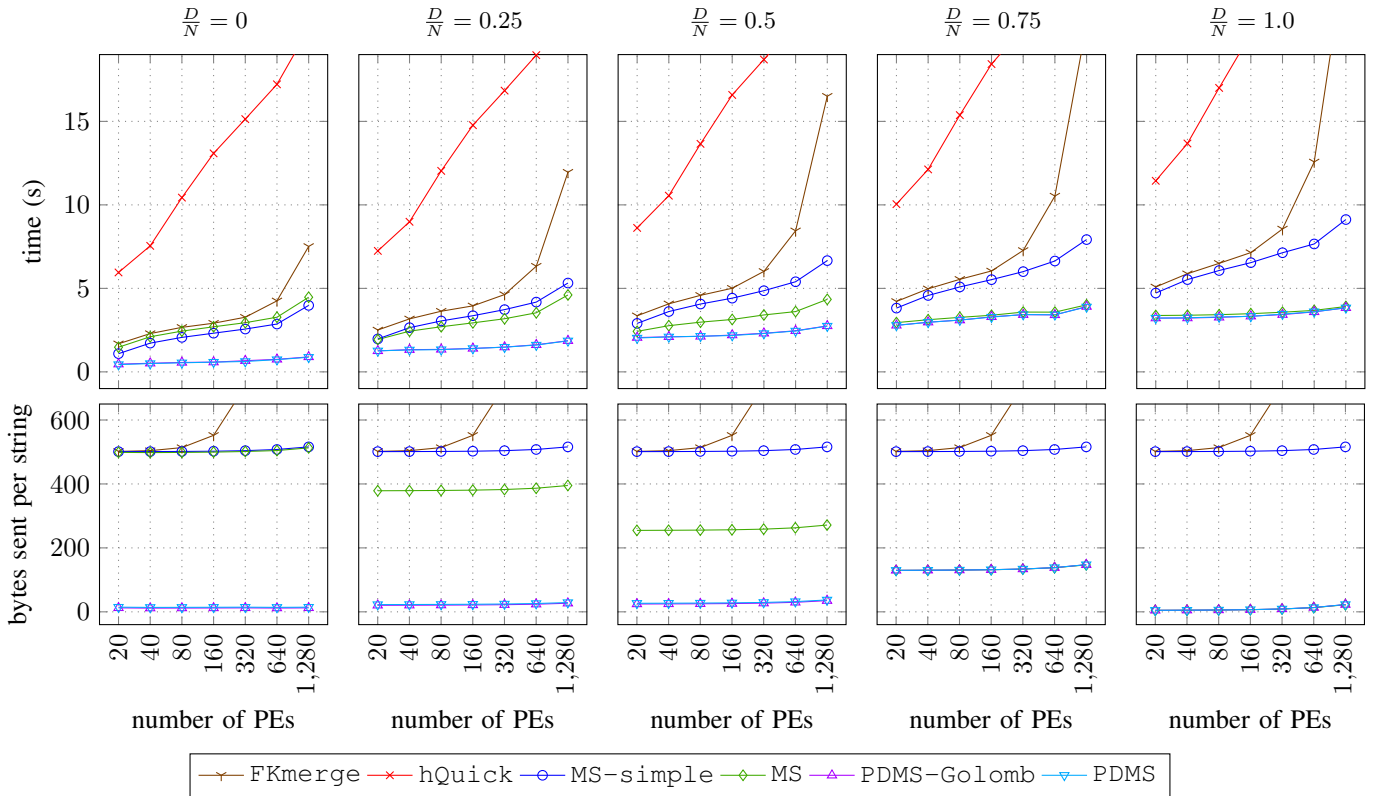
8

Fig. 4. Running times and bytes sent per string for the weak-scaling experiment with five generated $\frac{D}{N}$ inputs with 500 000 strings of length 500 per PE.

distinguishing prefixes is rather small anyway. On the largest configuration (1280 PEs, 64 nodes, 320 GB of data) the best shown algorithm is 5.3–8.6 times faster than FKmerge.

A look at the communication volumes in the lower part of the plot underlines the great communication efficiency of combining LCP compression with prefix doubling (algorithms PDMS-Golomb and PDMS). Using LCP compression only (Algorithm MS) is only effective when the LCPs are long.

Fig. 5, left panel, shows strong scaling results for the COMMONCRAWL instance. Here we cannot show results for the competing code FKmerge since it crashes. Apparently it does not correctly handle inputs with many repeated input strings. The ranking of the remaining algorithms is similar as for the D/N-instances. For $p \geq 480$, the algorithms based on prefix doubling are 5.4–6.1 times faster than hQuick and MS is a factor 4.5–4.6 faster. The algorithms with LCP compression are 2.6–3.5 times faster than MS-simple. This indicates that the LCP optimizations are very effective for the COMMONCRAWL-instance while prefix doubling does not help here. This is consistent with the large D/N-ratio of 0.68 for this instance where prefix doubling cannot be effective. The running times keep going down until the largest configuration tried. However, efficiency is rapidly deteriorating. The reason for the difference to the above weak scaling result may be that the COMMONCRAWL-instance is a factor four smaller for $p = 1280$. Hence, experiments with larger real world inputs are interesting topics for future work.

Fig. 5, right panel, shows the corresponding results for the DNAREADS input. Here, algorithms MS and MS-simple are slightly faster than the prefix doubling algorithms, despite considerable savings in communication volume. Algorithm FKmerge works now but does not scale so well.

E. Summary of Further Experiments

We now outline the results of further experiments in [22]. Character-based sampling is inconsequential for the D/N instances since their uniform length and random distribution makes load balancing easy. For COMMONCRAWL, our initial implementation is detrimental indicating further improvement potential. In particular, we have to carefully handle repeated short substrings. A tendency of character-based sampling to select long splitter keys indicates that one should perhaps consider using prefixes of samples as splitters.

Besides the COMMONCRAWL and DNAREADS instances, we also tried an instance consisting of 71 GB of Wikipedia pages. The results are similar to the COMMONCRAWL instance so that we do not show them here.

As a first attempt in the direction of suffix sorting, we considered the first 3000 lines of the above Wikipedia instance as a single string and used all their suffixes as input. This instance has $N \approx 104 \cdot 10^9$ and $D \approx 10.4 \cdot 10^6$, i.e., $D/N \approx 0.0001$. This is a very easy instance (execution time about 0.2 seconds on 160 PEs) for algorithm PDMS and a fairly difficult instance for all the other algorithms. Algorithm PDMS is about
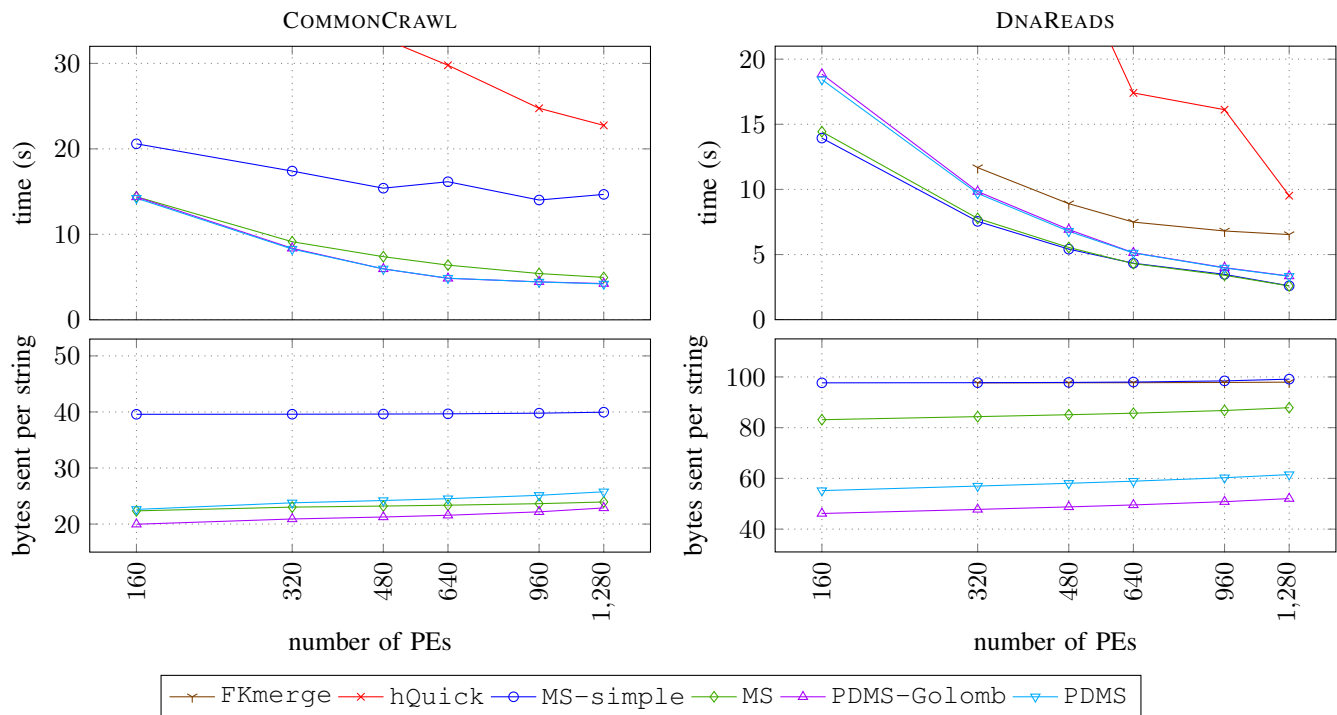
Fig. 5. Running times and number of bytes sent per string in strong-scaling for COMMONCRAWL (82 GB) and DNAREADS (125 GB) inputs.

30 times faster than the other algorithms for $p = 160$. For larger p this advantage shrinks because larger inputs would be needed to achieve scalability. However, these inputs would be very expensive for the other algorithms so that we did not pursue suffix instances given our limited compute resources.

We also generated *skewed* variants of our D/N-instances as follows: The 20 % smallest of these strings are padded with additional characters that make them 4 times longer (now 2000 characters) but without contributing to the distinguishing prefixes. The relative ranking of the algorithms hQuick, FKmerge, MS, and PDMS remains the same. Among the variants of MS, those with character-based sampling now profit because they avoid deteriorating load balance due to the skewed distribution of output string lengths.

VIII. CONCLUSIONS AND FUTURE WORK

With Algorithm PDMS, we have developed a distributed-memory string sorting algorithm that efficiently sorts large data sets both in theory and practice. The algorithm is several times faster than the best previous algorithms and scales well for sufficiently large inputs.

One approach to further optimize the algorithm is to improve splitter selection. Analogous to the work done for atomic sorting [29] one could remove load balancing problems due to duplicate strings by tie breaking techniques. One could also consider whether it helps to look for short splitter strings. The bounds could also be improved by going from deterministic sampling to random sampling. This requires less samples and, in expectation, the sample strings have average length rather than length $\hat{\ell}$. Adapting the techniques from [11], [14] for

perfect splitting in atomic sorting to string sorting could also be interesting. Probably this only makes sense if we also use a refined cost model that takes both the number of strings and their distinguishing prefix lengths into account.

To speed up sorting of the sample but also for other applications, it is interesting to look for parallel string sorting algorithms for small inputs that are faster than hQuick. One approach would be to adapt the key idea of Multikey Quicksort to look not at entire strings as pivots [18]. In [6] this is refined by looking at several characters at once. Probably for a distributed algorithm one should look at up to $\mathcal{O}(\alpha \log p/(\beta \log \sigma))$ characters at a time to find the right balance between latency and communication volume.

An interesting observation is that algorithms based on data partitioning rather than merging are successful both for atomic sorting [29] and shared-memory string sorting [6]. We have chosen a merging based algorithm here since it is not clear how to do LCP compression without locally sorting first.

The large D/N values in our practical inputs are in part due to many repeated keys. Perhaps one could design string sorting algorithms that do not communicate duplicate keys. One could modify Algorithm PDMS so that it detects likely duplicates and decides to communicate only one copy of each duplicate to their common destination PE. A problem with this approach is that we cannot guarantee that these strings are duplicates. We could enforce a very small false positive rate but we would end up with a result that is only approximately sorted.

The average-case upper bound of $\mathcal{O}(\log p)$ bits per string from Section VI-B is intriguing from a theoretical point of view. For atomic sorting, the average case and the worst case

10

running time share the same upper and lower bounds. Does this extend to the communication complexity of string sorting? Are there algorithms that need $o(D/n)$ communication volume in the worst case? What are the lower bounds? D/n? $\log p$? Something in between? The answer will likely depend on the small print in how we define our sorting problem. It should also be noted here that the lower bound for the easier problem of duplicate detection is conjectured to be $\log p$ [10] but that this is also still an open problem for distributed communication complexity with point-to-point communication.

The algorithm for approximating distinguishing prefixes from Section VI-A is an overkill if we only need information on global values like D/n or its variance. These values can be approximated more efficiently by sampling. A simple approach is to gossip a small sample of the input strings. Then, without further communication, their distinguishing prefix sizes can be computed locally. However, this way we can only process small samples which might be insufficient when $\hat{d} \gg D/n$ – a small sample is insufficient if D/n is dominated by a small number of strings with very large $\text{DIST}(s)$.[8] More efficiently, we can take a Bernoulli sample of prefixes of *keys* rather than input strings. This allows us to still use distributed hashing and thus makes the algorithm more scalable. Also this might reduce the amount of local work.

Acknowledgment

References

[1] R. Bayer and K. Unterauer, "Prefix B-trees," *ACM Transactions on Database Systems (TODS)*, vol. 2, no. 1, pp. 11–26, 1977.

[2] G. Graefe and P. . Larson, "B-tree indexes and CPU caches," in *17th Int. Conference on Data Engineering (ICDE)*, 2001, pp. 349–358.

[3] N. Futamura, S. Aluru, and S. Kurtz, "Parallel suffix sorting," in *9th International Conference on Advanced Computing and Communications*. McGraw-Hill, 2001, pp. 76–81.

[4] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *Journal of the ACM*, vol. 53, no. 6, pp. 1–19, 2006.

[5] T. Bingmann and P. Sanders, "Parallel string sample sort," in *21st European Symposium on Algorithms (ESA)*, ser. LNCS, vol. 8125. Springer, 2013, pp. 169–180.

[6] T. Bingmann, "Scalable string and suffix sorting: Algorithms, techniques, and tools," Ph.D. dissertation, Karlsruhe Institute of Technology, 2018.

[7] T. Bingmann, A. Eberle, and P. Sanders, "Engineering parallel string sorting," *Algorithmica*, vol. 77, no. 1, pp. 235–286, Jan 2017.

[8] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill *et al.*, "Exascale software study: Software challenges in extreme scale systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, pp. 1–153, 2009.

[9] S. Borkar, "Exascale computing – a fact or a fiction?" in *IEEE 27th Int. Symposium on Parallel and Distributed Processing*, May 2013.

[10] P. Sanders, S. Schlag, and I. Müller, "Communication efficient algorithms for fundamental big data problems," in *IEEE Int. Conference on Big Data*, 2013, pp. 15–23.

[11] P. J. Varman, S. D. Scheufler, B. R. Iyer, and G. R. Ricard, "Merging multiple lists on hierarchical-memory multiprocessors," *Journal on Parallel & Distributed Computing*, vol. 12, no. 2, pp. 171–177, 1991.

[12] M. Rahn, P. Sanders, and J. Singler, "Scalable distributed-memory external sorting," in *26th IEEE International Conference on Data Engineering (ICDE)*, 2010, pp. 685–688.

[13] H. Sundar, D. Malhotra, and G. Biros, "HykSort: A new variant of hypercube quicksort on distributed memory architectures," in *27th ACM Int. Conference on Supercomputing (ICS)*, 2013, pp. 293–302.

[14] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz, "Practical massively parallel sorting," in *27th ACM Symposium on Parallelism in Algorithms and Architectures*, 2015, pp. 13–23.

[15] J. Fischer and F. Kurpicz, "Lightweight distributed suffix array construction," in *212st Meeting on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2019, pp. 27–38.

[16] P. Fraigniaud and E. Lazard, "Methods and problems of communication in usual networks," *Disr. Appl. Math.*, vol. 53, no. 1–3, pp. 79–133, 1994.

[17] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev, *Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox*. Springer, 2019.

[18] J. L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings," in *8th ACM-SIAM Symp. on Discr. Alg. (SODA)*, 1997, pp. 360–369.

[19] T. Bingmann, "TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers," 2018, panthema.net/tlx.

[20] W. Ng and K. Kakehi, "Merging string sequences by longest common prefixes," *IPSJ Digital Courier*, vol. 4, pp. 69–78, 2008.

[21] D. Knuth, "Sorting and searching," *The art of computer programming*, vol. 3, p. 513, 1998.

[22] M. Schimek, "Distributed string sorting algorithms," Master's thesis, Karlsruhe Institute of Technology, 2019, doi:10.5445/IR/1000098432.

[23] J. Kärkkäinen and T. Rantala, "Engineering radix sort for strings," in *Symp. on String Processing and Inf. Retrieval (SPIRE)*, 2008, pp. 3–14.

[24] R. Sinha and A. Wirth, "Engineering burstsort: Toward fast in-place string sorting," *J. of Exp. Algorithmics (JEA)*, vol. 15, pp. 2–5, 2010.

[25] T. Hagerup, "Optimal parallel string algorithms: sorting, merging and computing the minimum," in *26th ACM Symposium on Theory of Computing (STOC)*, 1994, pp. 382–391.

[26] J. F. JáJá and K. W. Ryu, "An efficient parallel algorithm for the single function coarsest partition problem," *Theoretical Computer Science*, vol. 129, no. 2, pp. 293–307, 1994.

[27] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[28] B. Neelima, A. S. Narayan, and R. G. Prabhu, "String sorting on multi and many-threaded architectures: A comparative study," in *Int. Conf. on High Perf. Comp. & Appl. (ICHPCA)*, Dec 2014, pp. 1–6.

[29] M. Axtmann and P. Sanders, "Robust massively parallel sorting," in *19th Meeting on Alg. Eng. & Exp. (ALENEX)*. SIAM, 2017, pp. 83–97.

[30] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of parallel and distributed computing*, vol. 14, no. 4, pp. 361–372, 1992.

[31] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi, "On the versatility of parallel sorting by regular sampling," *Parallel Computing*, vol. 19, no. 10, pp. 1079–1103, 1993.

[32] M. Axtmann, "Karlsruhe distributed sorting library (KaDiS)," 2019, github.com/MichaelAxtmann/KaDiS, retrieved Oct. 7, 2019.

---

[8]Initial calculations indicate that a sample of size $\Theta(\varepsilon^{-2} n\hat{d}/D)$ is needed to approximate D with standard deviation εD.