Karlsruhe Institute of Technology

Distributed String Sorting Algorithms

Master's Thesis of

Matthias Schimek

Department of Informatics
Institute of Theoretical Informatics
Karlsruhe Institute of Technology

Reviewer:	Prof. Dr. Peter Sanders
Advisor:	Dr. Timo Bingmann

Time Period: 7th January 2019 – 8th July 2019

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 8th July 2019

This work was performed on the supercomputer ForHLR funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

Abstract

Although there has been extensive work on sequential and shared-memory parallel string sorting, the problem has not yet been thoroughly studied for distributed systems. In this thesis we present two new distributed string sorting algorithms. Our first algorithm *distributed Merge String Sort* extends the – to our knowledge – only distributed string sorting algorithm with longest common prefix-related optimizations. Furthermore, we present a new approach to compute an ordered partition of a distributed string array such that the number of characters in each set of the partition has about the same value.

Our second algorithm *distributed Prefix-Doubling String Sort* addresses a major problem of distributed string sorting: the communication of characters which are not required to establish a lexicographical order of the input. Distributed Bloom filters are applied to approximately compute the distinguishing prefixes of the input without exchanging the actual strings. Afterwards, the algorithm operates on the (approximate) distinguishing prefixes only. By this means, a significant amount of communication can be saved provided that the distinguishing prefixes are short compared to the strings themselves.

Furthermore, we introduce a new string generator producing string data sets with the ratio of the distinguishing prefix length to the entire string length being an input parameter. Our evaluation on up to 1280 processors shows that the presented algorithms clearly outperform their competitors on both generated and real-world data.

Deutsche Zusammenfassung

Obwohl in den letzten Jahren das Sortieren von Zeichenketten sowohl im Sequentiellen als auch auf parallelen Systemen mit gemeinsamen Speicher intensiv untersucht wurde, ist das Problem auf verteilten Systemen ohne gemeinsamen Speicher noch weitgehend unbehandelt. In der vorliegenden Arbeit werden zwei Algorithmen für das verteilte Sortieren von Zeichenketten vorgestellt. Der erste Algorithmus, *distributed Merge String Sort*, erweitert den einzigen bisher publizierten verteilten Algorithmus zum Sortieren von Zeichenketten um Optimierungen, welche die Eigenschaften längster gemeinsamer Präfixe der Eingabe ausnutzen. Darüberhinaus wird ein neuer Algorithmus zum Berechnen einer geordneten Partition verteilter Zeichenketten präsentiert, welcher sicherstellt, dass die Anzahl an Zeichen in jeder Teilmenge der Partition annähernd identisch ist.

Der zweite Algorithmus, *distributed Prefix-Doubling String Sort*, geht eines der Hauptprobleme beim Sortieren von Zeichenketten auf verteilten Systemen an, das darin besteht, nur die für das Sortieren der Zeichenketten tatsächlich benötigten Teile derselben zu kommunizieren. Zu diesem Zweck werden verteilte Bloomfilter verwendet, die es erlauben, die für die Sortierung relevanten Präfixe der Zeichenketten approximativ zu berechnen, ohne diese als ganze auszutauschen. Ab diesem Zeitpunkt werden im Algorithmus nur noch Operationen auf den zuvor berechneten Präfixen durchgeführt. Hierdurch kann ein erheblicher Anteil des Kommunikationsvolumens eingespart werden – vorausgesetzt, dass die für die Sortierung relevanten Präfixe im Vergleich zu den (Gesamt-)Zeichenketten kurz sind.

Weiterhin wird ein Zeichenketten-Generator vorgestellt, welcher es erlaubt, Datensätze mit einem vorgegebenen Verhältnis zwischen Länge der für die Sortierung relevanten Präfixe und der Länge der Zeichenketten zu erzeugen. In der abschließenden Evaluation auf bis zu 1280 Prozessoren wird gezeigt, dass distributed Merge String Sort und distributed Prefix-Doubling String Sort deutlich bessere Laufzeiten als die Konkurrenzalgorithmen besitzen. Diese werden sowohl auf generierten als auch nicht-künstlichen Datensätzen erreicht.

Acknowledgements

I would like to thank my advisor, Dr. Timo Bingmann, for his guidance, helpful advice and encouraging support through each stage of this thesis. I also would like to thank Michael Axtmann for the interesting discussions about MPI, related problems on distributed systems and possible solutions to them. Special thanks go to Philipp Ronecker for his help in proofreading this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

Sorting is one of the most important algorithmic operations with a large variety of applications [1, 3]. Most sorting algorithms assume their input elements to be *atomic*, i.e. elements of fixed size that can be compared or swapped in constant time. These assumptions do not hold for strings as input to these algorithms. Comparing two strings is a potentially costly operation whose time complexity can be linear in the length of the strings. Therefore, sorting strings is an algorithmic challenge different from atomic key sorting. Dedicated string sorting algorithms try to use the internal structure of strings like *longest common prefixes* to reduce the number of such expensive operations.

There has been been extensive work on sequential and also on shared-memory parallel string sorting [5]. However, the rise of Big Data makes it necessary to parallelize algorithms beyond the limited scope of shared-memory systems. Therefore, we investigate *distributed* string sorting in this thesis. This topic has not been given much attention yet, we are only aware of one publication [11] discussing distributed string sorting.

Figure 1.1 shows the general flow of a distributed string sorting algorithm. Unlike in sequential or shared-memory-parallel string sorting, the input string array is assumed to be distributed over the processing elements (PE) such that only a fraction $\mathcal{S}_i$ of the global string array is located on PE i at the beginning. Then the distributed string sorting algorithm is executed, communication between different PEs must be done using explicit messages as different PEs do not share memory. In the end, the output of the algorithm is also distributed over all PEs, i.e. on PE i an output array $\mathcal{O}_i$ is located. The output arrays are *locally* and *globally* sorted. In the context of Figure 1.1 this means that all strings in $\mathcal{O}_1$ are smaller than or equal to the strings in $\mathcal{O}_2$ and all strings in $\mathcal{O}_2$ are smaller than or equal to the strings in $\mathcal{O}_3$. Locally refers to the sortedness of each of the string arrays. The number of strings in each of the output arrays depends on the used algorithm.

The main differences between shared-memory and distributed parallelism are the larger number of processors and the necessity to communicate via messages in distributed systems, which is usually more expensive. This implies that an efficient distributed string sorting algorithm must on the one hand exploit the properties of strings to avoid repeated costly operations on the strings in local operations. On the other hand it should also try to reduce the communication volume as much as possible.

1.1 Contribution

In this thesis we present two new distributed string sorting algorithms. In Section 4 we introduce distributed Merge String Sort (dMSS), an extension of [11]. Opposed to the original algorithm, we

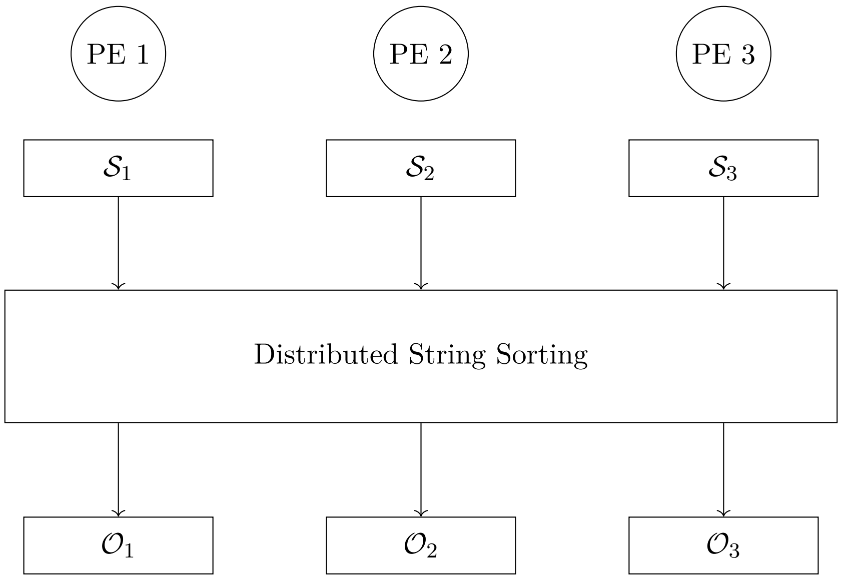


Figure 1.1: Illustration of a distributed string sorting algorithm.

also compute the longest common prefixes while sorting the input. These are then used to avoid repeated costly comparisons of the whole strings. Additionally, the knowledge of these prefixes enables us to apply a simple and fast compression on the strings in order to reduce the amount of communication. Furthermore, we present a new distributed partitioning approach that enables us to redistribute the input over the PEs such that each PE has about same amount of characters in the output sets.

In Section 5 distributed Prefix-Doubling String Sort (dPDSS) is presented. In this algorithm characters that are not required to sort the strings are cut off before the strings are exchanged. Applying additionally the same compression technique and the partitioning approach already used in dMSS the communication volume of our algorithm can be considerably smaller than the total number of characters of the input string array. Since only the distinguishing prefixes of the input strings are communicated, this algorithm's output is not the sorted string array itself but the permutation defining the sorted order of the strings.

The presented algorithms will be evaluated on various generated and real-world data sets in Section 7.

2. Preliminaries and Related Work

2.1 Definitions

A string array $\mathcal{S}$ is an array $\mathcal{S} \coloneqq [s_0, ..., s_{n-1}]$ of $|\mathcal{S}| \coloneqq n$ strings. A string s of length $l = |s|$ is a (zero-based) array of $l-1$ characters over the alphabet $\Sigma \coloneqq \{c_0, \ldots, c_{\sigma-1}\}$ of size σ, i.e. $s \coloneqq [c_0, c_1, \ldots, c_{l-2}, 0]$ with $c_i \in \Sigma$. The last character of a string is always the end-of-string character 0. The end-of-string character is not part of the alphabet. String arrays are usually represented as arrays of pointers to the beginning of the strings. Using this indirection, moving one or swapping two strings can be achieved in constant time by moving or swapping the corresponding pointers. The expression c^i with $0 < i$ and $c \in \Sigma$ denotes i repetitions of the character c. By $\|\mathcal{S}\|$ we denote the total sum of all characters of the strings contained in $\mathcal{S}$. The length of the longest string in a string array is denoted by l_{max}.

We use the access operator on a string s in two different ways: $s[i]$ returns the i^{th} character of a string, $s[i,j]$ denotes the character subarray $[s[i], \ldots, s[j]]$ for $0 \leq i \leq j < |s|$. Furthermore, we define $s[i,j) \coloneqq [s[i], \ldots, s[j-1]]$ and $s[i,] \coloneqq [s[i], \ldots, s[|s|-1]]$. The concatenation of two arrays of characters $a \coloneqq [a_0, \ldots, a_i]$ and $a_2 \coloneqq [a'_0, \ldots, a'_j]$ is denoted by $a_1 + a_2 \coloneqq [a_0, \ldots, a_i, a'_0, \ldots, a'_j]$. A *prefix* of a string s of length l is the subarray $s[0,l)$, provided $|s| \geq l$. *Sorting* a string array means to permute the strings of the array such that a *lexicographical* order is established.

Let s_1 and s_2 be two strings. By $\textsc{lcp}(s_1, s_2)$ we denote the length of their *longest common prefix*. If $s_1 = s_2$ we set $\textsc{lcp}(s_1, s_2) = |s_1| - 1$. For a *sorted* array of strings $\mathcal{S}$, we define the corresponding LCP array $\mathcal{H}(\mathcal{S}) \coloneqq [\perp, h_1, h_2, \ldots, h_{|\mathcal{S}|-1}]$ with $h_i \coloneqq \textsc{lcp}(s_{i-1}, s_i)$ for $1 \leq i < |\mathcal{S}|$. The sum of all values in $\mathcal{H}(\mathcal{S})$ excluding the first entry is denoted by $\mathcal{L}(\mathcal{S})$.

The distinguishing prefix $\textsc{dp}_{\mathcal{S}}(s)$ of a string s in an array $\mathcal{S}$ is the number of characters that must be inspected to differentiate it from any other string t in $\mathcal{S}$, i.e. $\textsc{dist}_{\mathcal{S}}(s) = \max\limits_{t \in \mathcal{S}, t \neq s} (\textsc{lcp}(s,t) + 1)$. The sum of the lengths of the distinguishing prefixes of a set $\mathcal{S}$ is $\mathcal{D}(\mathcal{S})$. This value $\mathcal{D}(\mathcal{S})$ is a lower bound on the number of characters that must be inspected to sort the array $\mathcal{S}$. Hence, it is a lower bound on the number of character comparisons in comparison-based string sorting algorithms. Figure 2.1 depicts a string array $\mathcal{S}$ consisting of 8 strings. The distinguishing prefixes are coloured red and the values of the LCP array $\mathcal{H}(\mathcal{S})$ are shown. This example reveals that $\mathcal{L}(\mathcal{S})$ and $\mathcal{D}(\mathcal{S})$ are related, but not equal. For a string array $\mathcal{S}$ the sum $\mathcal{D}(\mathcal{S})$ comprises all characters counted in $\mathcal{L}(\mathcal{S})$ plus further characters. Lemma 2.1 describes this relation quantitatively.

Lemma 2.1 (Relation between $\mathcal{L}(\mathcal{S})$ and $\mathcal{D}(\mathcal{S})$ [5])**.** *For any string array $\mathcal{S}$ with $n \coloneqq |\mathcal{S}|$*

$$n + \mathcal{L}(\mathcal{S}) \leq \mathcal{D}(\mathcal{S}) \leq 2\mathcal{L}(\mathcal{S}) + n$$

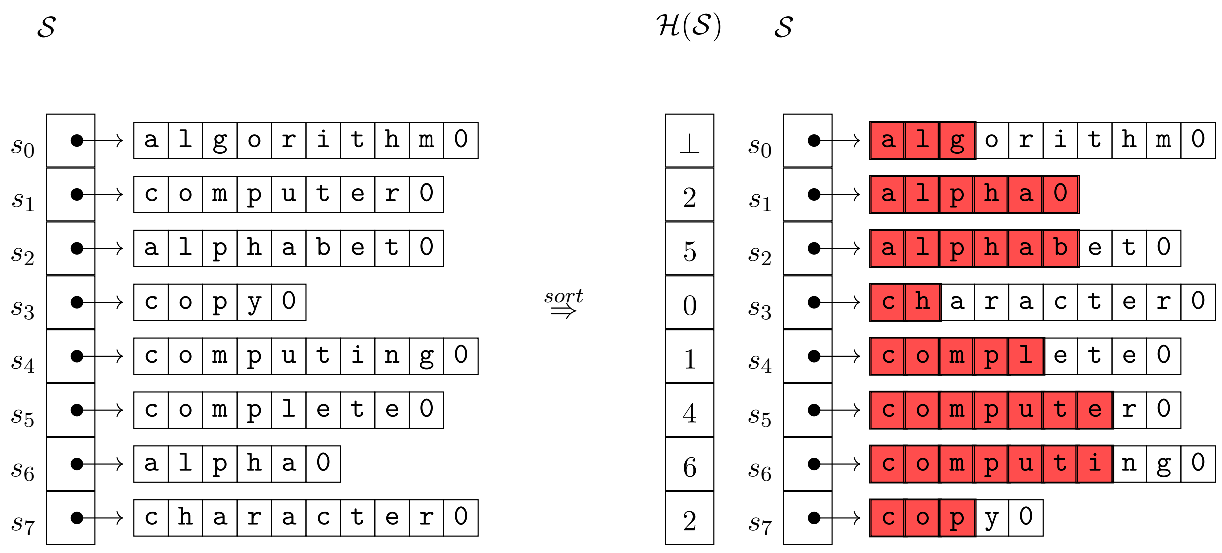


Figure 2.1: String array $\mathcal{S}$ consisting of 8 strings. The distinguishing prefixes are coloured red. We have $\mathcal{L}(\mathcal{S}) = 20$, $\mathcal{D}(\mathcal{S}) = 39$ and $\|\mathcal{S}\| = 68$.

holds.

Note that the only relationship between $\mathcal{D}(\mathcal{S})$ and $\|\mathcal{S}\|$ is $\mathcal{D}(\mathcal{S}) \leq \|\mathcal{S}\|$. The number of characters is not bounded by $\mathcal{D}(\mathcal{S})$ and there are data sets with $\|\mathcal{S}\| \gg \mathcal{D}(\mathcal{S})$.

A last remark on pseudo-code used in the thesis: we use a mathematical set notation style when appropriate, e.g. an expression like $A \leftarrow [i^2 \mid 0 \leq i \leq 3]$ sets A to the array $[0, 1, 4, 9]$. Furthermore, arrays or variables are not declared or allocated explicitly before their first usage, e.g. an expression like $\mathcal{S}[i] \leftarrow s$ allocates an array $\mathcal{S}$ and sets its i^{th} entry to s.

2.2 Model of Communication

In this thesis the *linear* model of communication [13] is assumed. Sending a message of m data units (usually bits or bytes) from one Processing Element (PE) to another PE takes $\alpha + \beta m$ time. The constant α is the so called *start-up* latency and β the speed at which one data unit can be transferred. Furthermore, we define that a PE is *single ported*, i.e. it can only communicate with *one* other PE at a time. We assume such a communication to be *full duplex*, i.e. a PE can send to and receive from its communication partner simultaneously. Additionally, we define c to be the number of bits required to transmit one character and w bits are required to communicate an integer.

2.3 Collective Communication Operations

Collective communication operations are operations in which more than two PEs participate. Since we make use of some collective communication operations in our algorithms, we will give a brief overview of the time complexity in which these operations can be executed. We will only state the time complexity of the operations up to constant factors. The number of PEs in the distributed system is denoted by p.

2.3.1 Broadcast

In the *broadcast* operation one PE wants to send a message of length m to all other PEs. There is a lower bound on the running time of $\alpha \log(p) + \beta m$ since the PEs reached by the initial PE can only double in each round and the whole message must leave the sending PE at least once. This lower bound is reached by algorithms like the *23-broadcast* algorithm up to a constant factor [23].

2.3.2 (All-)Reduce

Let $\oplus$ be an associative binary operator on a datatype `T`. Assume that PE i possesses a vector M_i with m elements of type `T`. In a reduce operation $M \coloneqq \oplus_{i \leq p} M_i$ is computed. After the computation the result M is located on a dedicated root PE. An adaption of the 23-broadcast algorithm can be used to execute a reduce operation in $\mathcal{O}\left(\alpha \log(p) + \beta m\right)$ [23]. The allreduce operation has the same asymptotic complexity as we can simply combine one reduce with one broadcast operation.

2.3.3 Prefix Sum

Again, let $\oplus$ be an associative operator. As above, PE i possesses a vector M_i consisting of m elements of type `T`. We want to calculate the prefix sum $s_i \coloneqq \oplus_{i \leq p} M_i$ for PE i and $1 \leq i \leq p$. As for the last two collective operations a variant of the 23-algorithm can be applied. Hence, this problem has a time complexity in $\mathcal{O}\left(\alpha \log(p) + \beta m\right)$ [23].

2.3.4 (All-)Gather

In the *gather* operation one designated PE (often called the *root*) collects a message from all other PEs. A lower bound for this problem is given by $\alpha \log(p) + \beta(p-1)m$. The summand $\beta(p-1)m$ is due to the fact that the root must receive all $p-1$ messages and can only receive one data unit at a time. Since the number of messages aggregated on any PE can at most double in each communication round and the root must receive all p messages, there are at least $\log(p)$ start-ups necessary for this task.

In the *allgather* operation each PE i wants to communicate its message m_i to all other PEs. Thus, after the allgather operation each PE possesses all messages $m_1, \ldots, m_p$. Since an allgather algorithm also solves the gather communication task, the same lower bound holds.

The gather problem can be solved in $\alpha \log(p) + \beta(p-1)m$ using a binomial tree communication graph [10]. As combining one gather and one broadcast operation yields an algorithm for the allgather operation, we find the time complexity for this problem to be in $\mathcal{O}(\alpha \log(p) + \beta p m)$ for arbitrary p.

2.3.5 Alltoall

In this problem each PE i has a message m_{ij} of length m for every PE j. There is a lower bound on the time complexity of $\beta(p-1)m$ since each PE has to send at least $(p-1)$ messages of size m. For arbitrary p the *1-factor* algorithm can be used with a time complexity in $\mathcal{O}(p(\alpha + \beta m))$ [22]. This algorithm basically consists of p communication rounds in which each PE exchanges a message with a dedicated partner.

The 1-factor algorithm solves the *regular* alltoall operation, i.e. each message has the same length. In our algorithms we will not always have this situation. Instead, we are confronted with the *irregular* alltoall problem. As before, each PE has a message for any other PE but this time the length of these messages can vary. A naïve solution is to pad each message to the length of longest message and apply the algorithm for the regular problem. However, this might incur an unnecessary increase in runtime. If instead we assume that the length of a message can vary, but the overall amount of data to be sent and received from each PE M is identical, the *2-phase* algorithm can be applied [21]. This algorithm partitions the data into regular messages of length M/p and exchanges/recomposes them in two rounds using an algorithm for the regular alltoall problem. Hence, the time complexity in this case is also in $\mathcal{O}(p(\alpha + \beta m))$ if the 1-Factor algorithm is used as the base algorithm.

2.4 Related Work

In the following we will describe the – to our knowledge – only published distributed string sorting algorithm by Fischer and Kurpicz [11], on which our dMSS algorithm is based. Figure 2.2 illustrates

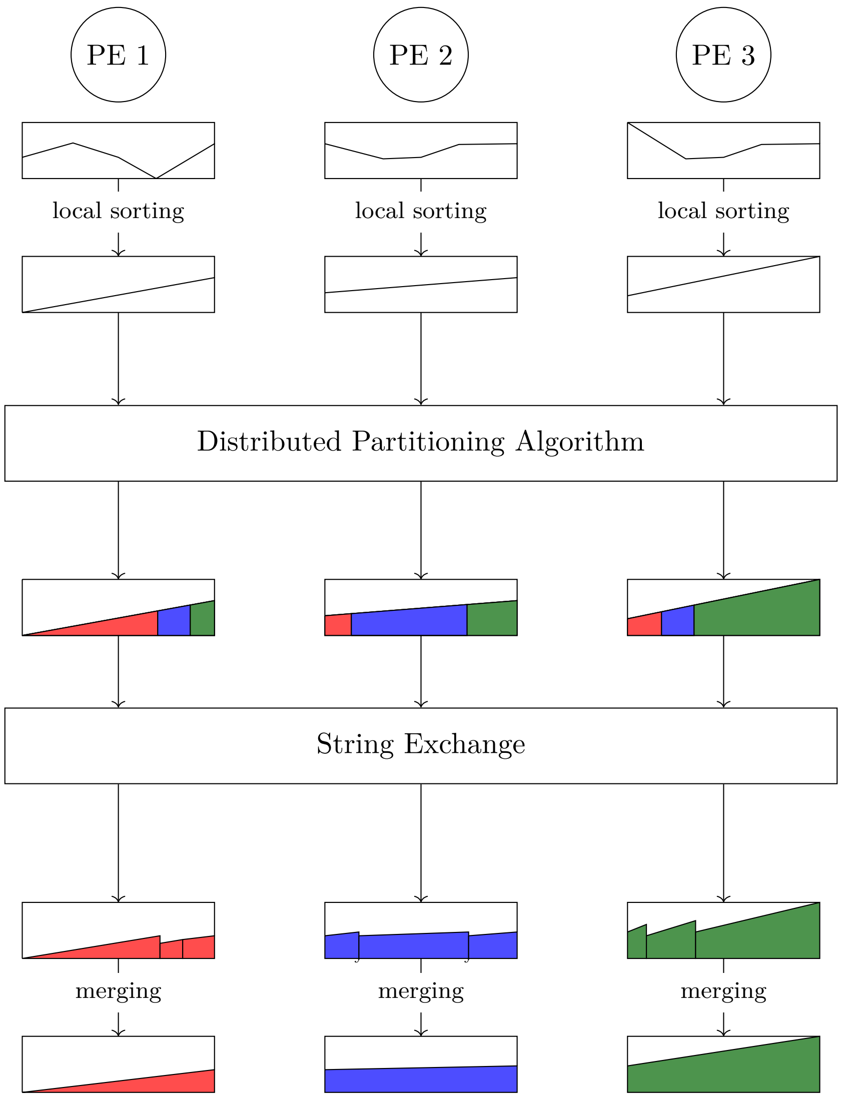


Figure 2.2: Illustration of the distributed string sorting algorithm by Fischer and Kurpicz [11].

the execution of the algorithm. At the beginning we assume that the input is already distributed over the PEs. In a first step the input to each PE is sorted locally. Then a distributed partitioning algorithm divides the input into p buckets. More precisely, the partitioning algorithm determines $p-1$ splitters *globally* such that the strings on each PE can be assigned to p buckets $\mathcal{B}^i$ for $1 \leq i \leq p$. These buckets have the property that each string in bucket $\mathcal{B}^i$ is smaller than or equal to each string in bucket $\mathcal{B}^j$ for $1 \leq i < j \leq p$. In Figure 2.2 the assignment of a string to a bucket is depicted by the colours in which the local string array is coloured after the distributed partitioning step. Note that up to this step the buckets are distributed over the PEs. In the next step PE i packs all strings assigned to bucket j into a message m_{ij}. Hence, there are p messages on each PE. These are exchanged by an (irregular) alltoall operation. After this step all strings assigned to bucket i are located on PE i, i.e. each PE has p sorted sequences of strings and the strings are already *globally* sorted. Therefore, it is sufficient to apply a multiway-merge algorithm to sort the strings also locally.

For the local sorting of the input string arrays any sequential string sorter can be used. In the distributed partitioning algorithm each PE samples $p-1$ strings equidistantly from its local input array. Then an allgather operation is executed on the local sample sets so that all PEs possess all p local sample sets. Subsequently, each PE sorts the sample sets using a sequential string sorting algorithm and chooses $p-1$ final splitters equidistantly from the sorted array. Based on these final splitters the p buckets are determined. For the merging step Fischer and Kurpicz use a losertree (as described in Section 3.5). Note that apart from the sequential string sorter in the first step the algorithm does not apply any techniques exploiting the special properties of strings sets to accelerate the sorting process.

3. Techniques and Subroutines

In this section algorithmic building blocks that are used in both algorithms – dMSS and dPDSS – will be presented. We will cover sequential string sorting, parallel (string) sorting for small local sets, distributed partitioning (for redistribution), LCP compression, and efficient multiway-merging of sorted string sequences. In Section 3.2 and 3.3 we will introduce distributed algorithms with string arrays as input to each PE. For these inputs we state the following conventions:

Input

1. PE i obtains a string array $\mathcal{S}_i$ as input. By $\mathcal{S}$ we denote the union of all local input arrays, i.e. $\mathcal{S} = \bigcup\limits_{1 \leq i \leq p} \mathcal{S}_i$. Although the term "union" is used, we allow duplicates in $\mathcal{S}$.
2. For the number of strings and the number of characters in the local input array
$$|\mathcal{S}_i| \leq \delta \frac{|\mathcal{S}|}{p} \qquad \text{and} \qquad \|\mathcal{S}_i\| \leq \Delta \frac{\|\mathcal{S}\|}{p}$$
holds with $\delta, \Delta \geq 1$. The factors δ and Δ describe the imbalance of the input. Note that for $\delta = \Delta = 1$ the input data is *perfectly* balanced over all PEs.

3.1 Sequential String Sorting

Sorting the input sets of the PEs locally is the first step in both of our main algorithms (see Section 4 and Section 5). In [5] 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. Therefore, we decided to use this sorting algorithm for the local sorting step in our two algorithms. In the following we briefly describe MSD string radix sort and its base sorters multikey quicksort and LCP-insertion sort with a string array $\mathcal{S}$ as input.

3.1.1 Multikey Quicksort

Multikey Quicksort by Bentley and Sedgewick [4] is an adaption of the common quicksort algorithm to strings (which are called multikeys in the original publication). Assume that all strings in $\mathcal{S}$ have a common prefix of length l, i.e. the first l characters of all strings are identical. Note that the algorithm is started with $l = 0$. As in the traditional quicksort algorithm, a pivot s_p is chosen to partition the input into the three sets $\mathcal{S}_<, \mathcal{S}_=$ and $\mathcal{S}_>$. The difference is that only the $(l+1)^{th}$ character of the strings is compared to $c_p := s_p[l]$ (recall that strings are zero-based arrays) to determine the partitions instead of the whole strings. The algorithm is applied recursively on the three partitions, unless $c_p = 0$. In this case the strings in $\mathcal{S}_=$ are all identical and, thus, sorted. Hence, the algorithm only continues on the partitions $\mathcal{S}_<$ and $\mathcal{S}_>$. For constant size sets *insertion sort* is used as base sorter.

The key insight is that the strings in $\mathcal{S}_=$ have a common prefix of length $l + 1$. Therefore, characters found equal to c_p will never take part in any character comparison again. This leads to an expected running time in $\mathcal{O}(|\mathcal{S}| \log |\mathcal{S}| + \mathcal{D}(\mathcal{S}))$.

3.1.2 (LCP-) Insertion Sort

Bingmann et al. [6] propose LCP-Insertion Sort as the string variant of the common insertion sort algorithm. The algorithm is similar to insertion sort, however the associated LCP-array of the string array to be sorted is kept up-to-date to reduce the number of necessary character comparisons. This yields a running time in $\mathcal{O}\left(\mathcal{D}(\mathcal{S}) + |\mathcal{S}|^2\right)$.

3.1.3 Most-Significant-Digit String Radix Sort

(MSD) string radix sort [14, 18] adapts the common radix sort algorithm to string sorting by inspecting strings characterwise. In the common variant of radix sort integers are inspected digit per digit. Since strings over an alphabet Σ can be seen as σ-ary integers, this approach is quite intuitive.

Suppose the string array $\mathcal{S}$ has a common prefix of length l. Then the algorithm looks at the $(l+1)^{th}$ character of each string to sort them into $\sigma + 1$ buckets. Recall that the end-of-string character is not part of the alphabet. The algorithm is applied recursively to each bucket but the bucket dedicated to the end-of-string character. It can be seen as a generalization of multikey quicksort that uses the maximum amount of information of a single character. If the size of the input string arrays is in $o(\sigma)$, the algorithm is no longer efficient as there are more buckets than strings in this case. Hence, we need another algorithm for the base case. Using multikey quicksort for buckets smaller than σ yields an expected running time for the base case in $\mathcal{O}(|\mathcal{S}| \log(\sigma) + \mathcal{D}(\mathcal{S}))$. The repetitive assignment to buckets is in $\mathcal{O}(\mathcal{D}(\mathcal{S}))$. Hence, the total expected time complexity is in $\mathcal{O}(|\mathcal{S}| \log(\sigma) + \mathcal{D}(\mathcal{S}))$.

3.2 Distributed Sorting of Small Sets

Both of our main algorithms contain a sub-step which needs to sort a relatively small array of strings globally. In the following we present two algorithms solving this task.

3.2.1 Centralized Sequential Sorting

The first method is very simple and rather sequential than parallel. We first gather all local inputs on one PE. This PE sorts the data sequentially using MSD string radix sort as described in Section 3.1. The concrete redistribution of the sorted data depends on the context in which the algorithm is used. The time complexity of the gather operation is in $\mathcal{O}(\alpha \log(p) + \beta \Delta c \|\mathcal{S}\|)$ as the longest message contains at most $\Delta \|\mathcal{S}\| / p$ characters. The sorting can be done in $\mathcal{O}(|\mathcal{S}| \log(\sigma) + \mathcal{D}(\mathcal{S}))$ time. Since $\mathcal{D}(\mathcal{S}) \leq \|\mathcal{S}\|$, we have a total time complexity in

$$\mathcal{O}(\alpha \log(p) + (\beta c + 1)\Delta \|\mathcal{S}\| + |\mathcal{S}| \log(\sigma)).$$

This running time might be prohibitive for large inputs, but it can be dominated by other steps of the overall algorithm in which this subroutine is used. In those cases this simple procedure is a reasonable alternative to more complicated distributed sorting algorithms (see [16]).

3.2.2 Hypercube Quicksort

During our experimental evaluation it turned out that the above-mentioned approach to sort small sets does not scale in our setting. Therefore, we decided to parallelize the sorting of the smaller sets with the hypercube quicksort algorithm presented by Axtmann and Sanders [1]. The algorithm is designed for fixed-length data. We adapted its implementation to be able to handle elements of variable length. From a conceptual point of view, however, the algorithm remains the same.

The algorithm can be split into an initialization phase followed by $\log(p)$ recursion levels. As with all other string sorting algorithms, each PE obtains a string array $\mathcal{S}_i$ as input.

Initialization

Let $h = \lfloor \log(p) \rfloor$.

1. **Reduction to hypercube:** In the case that p is not a power of two, i.e. $2^h < p$, all PE with an index $i \geq 2^h$ send their elements to the PE with index $i - 2^h$ and return. They do not further participate in the algorithm. All other PEs build an (implicit) hypercube H_{init} of dimension h.

2. **Random Redistribution:** All remaining 2^h PEs randomly exchange their data using a hypercube communication pattern (see [1] for details).

3. **Local Sorting:** The data on each PE is sorted locally.

Then we start the recursion step $i = 0$ with H_{init} as input.

Recursion

In recursion step i a hypercube H of dimension $h - i$ is taken as input, i.e. a hypercube with 2^{h-i} PEs. The precondition is that the data on each PE is already locally sorted, which holds for $i = 0$ because of the initialization phase. If $h = i$, i.e. the hypercube only contains one PE, the algorithm returns immediately. Otherwise, the following steps are executed:

1. **Pivot Selection:** A pivot element is selected by all PEs using a binary-tree-reduction (again, see [1] for more details).

2. **Exchange:** On each PE the local elements are partitioned into two sets $S_{\leq}$ containing all elements smaller than or equal to the pivot and $S_{>}$ containing all elements greater than the pivot. Then, each PE determines whether it is part of the 0-subcube H_0 of H or part of the 1-subcube H_1 by evaluating the i^{th} bit of its PE index. If a PE is part of H_0, it sends the set $S_{>}$ to its corresponding PE in the 1-subcube H_1 of H. If a PE is in H_1, it keeps the set $S_{>}$ and sends $S_{\leq}$ to its corresponding partner in H_0. The corresponding PE in the other subcube of a PE with index j is the one with index $2^i \oplus j$, with $\oplus$ being the bitwise `XOR`-operator.

3. **Merging:** On each PE, the received elements and the elements that were not sent in the exchange step are merged.

Thus, after exchanging and merging, all elements in H equal to or smaller than the pivot are on PEs in H_0, and elements greater than the pivot are in H_1. Furthermore, the elements on all PEs are locally sorted. We then recurse on the hypercubes H_0 and H_1.

3.3 Distributed Ordered Partitioning

In this section we develop the algorithm `partition` which computes a partition of the global string array $\mathcal{S}$ into p buckets such that all strings assigned to a bucket $\mathcal{B}^i$ are smaller than any of the strings in bucket $\mathcal{B}^j$ for $i < j$, i.e. the buckets of the partition are ordered. In both of our main algorithms – dMSS and dPDSS – each of these buckets will be assigned to one PE. Therefore, the main goal is to build buckets such that the workload associated with them in the subsequent steps of the algorithms is evenly distributed. When sorting atomic keys (i.e. integers or other fixed-length data) the workload usually correlates very well with the number of elements. Thus, one tries to choose such buckets that about the same number of elements are distributed to each PE. Sorting strings differs in this respect. Especially the time complexity of the distribution of the buckets in our distributed environment does not only depend on the number of elements but on the total number of characters in each bucket. This difference motivates the analysis of methods trying to achieve nearly equal amounts of characters in each bucket. The algorithm described in this section computes the buckets but does not yet distribute them to the PEs, i.e. after the execution of this partitioning algorithm the PEs only know which of their local strings are assigned to which bucket.

3.3.1 General Approach

As with the other distributed algorithms considered so far, PE i obtains a string array $\mathcal{S}_i$ as input. Furthermore, an integer s is part of the input. This integer is identical for all PEs. In addition to the conventions specified above, assume that the string array is already locally sorted. Furthermore, let each element in $\mathcal{S}$ (the union of the local input string arrays) be unique. This can be easily achieved by adding a unique identifier to each string. Although this might be prohibitive when handling atomic keys, regarding string sorting the additional identifier is neglectable as the elements themselves are assumed to be “long”. As a last assumption, let $\delta = \Delta = 1$. In Section 3.3.5 we will discuss how different amounts of strings or characters per PE can be handled by the algorithm. The computation of the buckets is done in four main steps:

1. Local Sampling

PE i chooses s elements from $\mathcal{S}_i$ forming the local sample $\mathcal{S}_i^{sample}$.

2. Sorting

Let $\mathcal{S}^{sample}$ be the union of all local samples $\mathcal{S}_i^{sample}$ (with $|\mathcal{S}^{sample}| = ps$). In the next step the *final splitters* f_i are chosen as the elements of $\mathcal{S}^{sample}$ with rank $is-1$ for $1 \leq i \leq p-1$. Note that we define the smallest element of a set to have rank 0. Therefore, we need $\mathcal{S}^{sample}$ to be sorted. This can be achieved by using one of the distributed sorting algorithms introduced in Section 3.2.

3. Extraction and Distribution of Splitters

The $p-1$ final splitters must be determined and communicated to all PEs. If the centralized sequential sorting algorithm described in Section 3.2.1 is used, the PE at which all data is sorted simply determines the final splitters and broadcasts them to all other PEs. If the hypercube quicksort algorithm (or any other sorting algorithm with a distributed output) is applied, we first compute a prefix sum over the number of elements in the output set $\mathcal{O}_i$ of the distributed sorting algorithm. By that, PE i knows the global rank of each element in $\mathcal{O}_i$. All elements with global rank $sj-1$ for $1 \leq j \leq p-1$ are inserted into $\mathcal{F}_i$. Then an allgather operation is performed on the local final splitter arrays $\mathcal{F}_i$ so that each PE possesses the complete set $\mathcal{F}$ of final splitters with $|\mathcal{F}| = p-1$.

4. Determination of Buckets

The $p-1$ final splitters are used to determine p buckets $\mathcal{B}^j$ into which the elements of $\mathcal{S}_i$ are partitioned. The buckets are defined as

$$\mathcal{B}^j = \begin{cases} \{s \in \mathcal{S} \mid s \leq f_1\} & \text{for } j = 1, \\ \{s \in \mathcal{S} \mid f_{j-1} < s \leq f_j\} & \text{for } 2 \leq j \leq p-1, \\ \{s \in \mathcal{S} \mid f_{p-1} < s\} & \text{for } j = p. \end{cases}$$

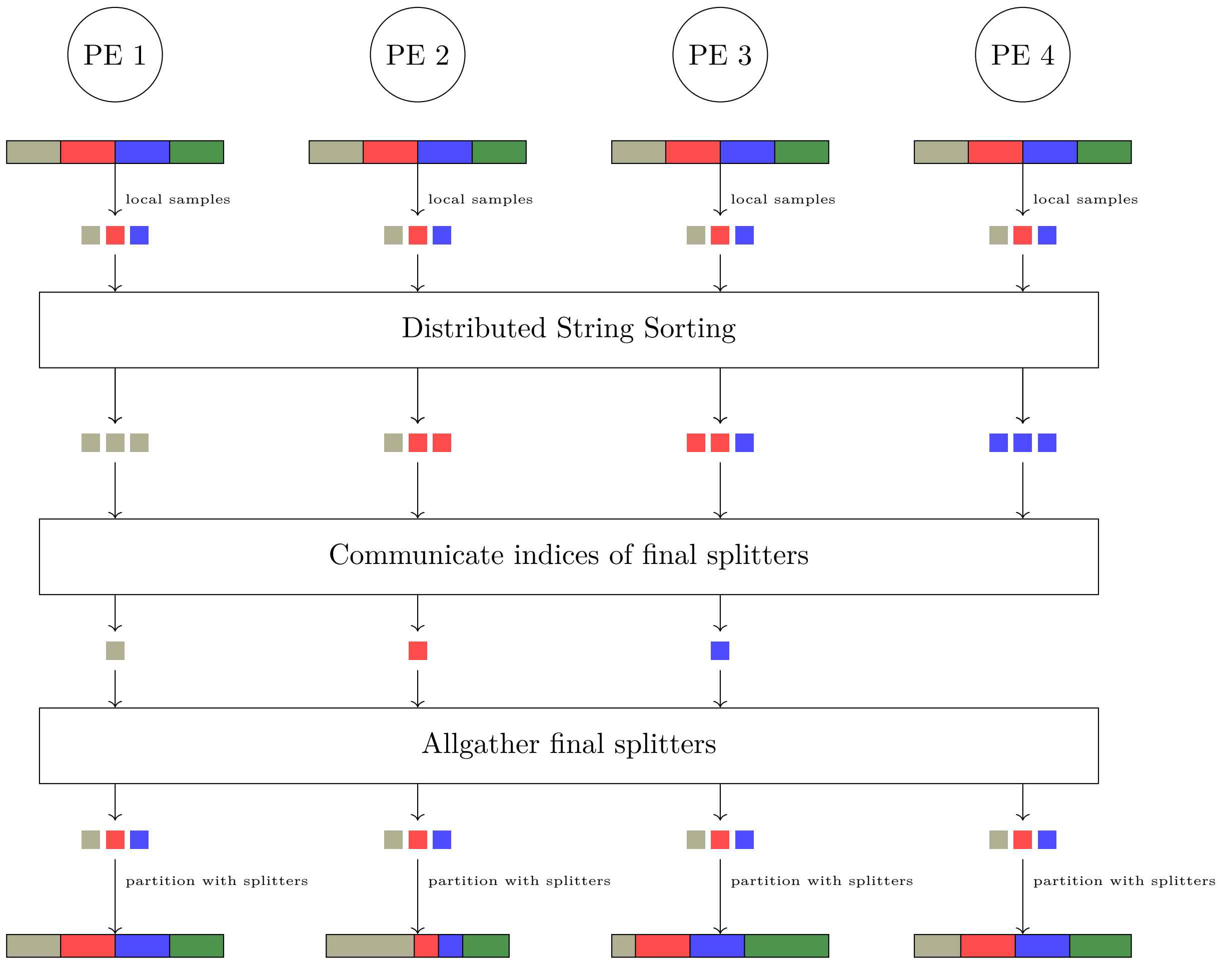

PE 1
PE 2
PE 3
PE 4
local samples
Distributed String Sorting
Communicate indices of final splitters
Allgather final splitters
partition with splitters

Figure 3.1: Illustration of the distributed partitioning algorithm.

The determination of the buckets can be done using $p-1$ binary searches on each PE. Figure 3.1 illustrates the execution of the `partition` algorithm.

There are different techniques to select the local samples in step 1. Two commonly used approaches are *random* and *regular* sampling. In the former s elements are picked from $\mathcal{S}_i$ at random. If s is chosen big enough, random sampling yields good guarantees for the maximum size of the buckets with a high probability (see [8] for further details).

However, there are no worst-case guarantees since the elements in the sample sets are picked at random. This can be overcome by using *regular sampling*. Here, we need the arrays $\mathcal{S}_i$ to be sorted. The sortedness is exploited by choosing the local samples evenly spaced. Doing so, each interval into which $\mathcal{S}_i$ is divided by the elements of $\mathcal{S}_i^{sample}$ is of equal size. This can be used to give worst-case bounds. The concept of regular sampling for sorting was first introduced by Shi and Schaeffer [25] and their follow-up paper [16]. We give a slightly different algorithm and a different analysis for the upper bounds since we have to be able to adapt the procedure also to character-based sampling in 3.3.4, too. The analysis presented in the paper is not applicable to this case.

3.3.2 Regular Sampling

In this approach PE i determines a set $\mathcal{S}_i^{sample}$ of s evenly spaced elements. Therefore, we assume that $|\mathcal{S}_i|$ is divisible by $s+1$. In this setting, we choose s elements with a distance of $\omega = |\mathcal{S}_i|/(s+1)$

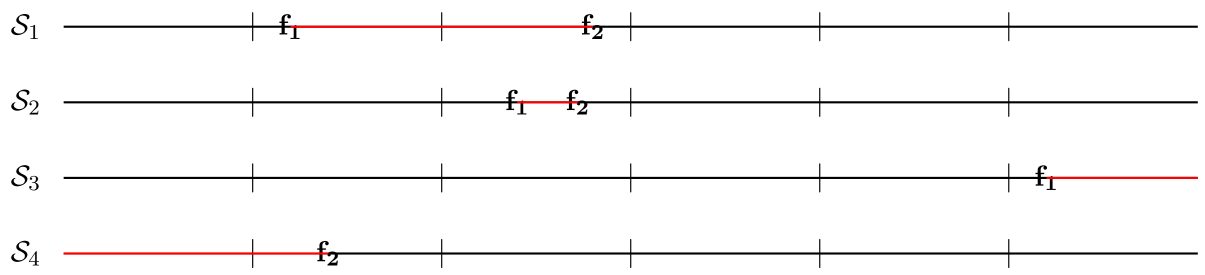


Figure 3.2: Illustration of the proof of Theorem 3.2.

from the set $\mathcal{S}_i$. More precisely, we select the elements $s_i^j = \mathcal{S}_i[\omega j - 1]$ for $1 \leq j \leq s$ as local sample set $\mathcal{S}_i^{sample}$.

$$\mathcal{S}_i : \underbrace{s_0, s_1, \ldots, s_{\omega-2}, s^1_{\omega-1}}_{\omega}, \underbrace{s_\omega, \ldots, s_{2\omega-2}, s^2_{2\omega-1}}_{\omega}, \ldots \underbrace{s_{(s-1)\omega}, \ldots, s_{s\omega-2}, s^s_{s\omega-1}}_{\omega}, \underbrace{s_{s\omega}, \ldots, s_{(s+1)\omega-1}}_{\omega}$$

The s elements from $\mathcal{S}_i^{sample}$ partition $\mathcal{S}_i$ into $s+1$ sets each of ω elements as depicted above. Theorem 3.2 gives an upper bound on the number of strings per bucket computed by `partition` using regular sampling. To prove this theorem, we need the following Lemma 3.1 giving an upper bound on the number of elements in a contiguous subarray of $\mathcal{S}_i$ if this subarray contains k elements of the local sample set.

Lemma 3.1 (Number of Sample Elements)**.** *For $1 \leq i \leq p$ let $\mathcal{S}' = \{s \in \mathcal{S}_i \mid a \leq s \leq b\}$ be a contiguous subarray of $\mathcal{S}_i$. If $|\mathcal{S}' \cap \mathcal{S}_i^{sample}| = k$, then $|\mathcal{S}'| \leq (k+1)\omega$.*

Proof. We distinguish three cases:

1. $k = 0$: There are three possibilities:

   a) All elements of $\mathcal{S}'$ are smaller than s_i^1. Then $|\mathcal{S}'| < \omega$ since s_i^1 is the ω^{th} element of $\mathcal{S}_i$.

   b) All elements of $\mathcal{S}'$ are contained completely between two consecutive elements of $\mathcal{S}_i^{sample}$. As above, we have $|\mathcal{S}'| < \omega$ since the number of elements of $\mathcal{S}_i$ between two consecutive elements of $\mathcal{S}_i^{sample}$ is ω.

   c) All elements of $\mathcal{S}'$ are greater than s_i^s. As there are ω elements greater than s_i^s in $\mathcal{S}_i$, this is an upper bound for $|\mathcal{S}'|$.

2. $k = 1$: Let s_i^j be the element of the local sample set contained in $\mathcal{S}'$. We split $\mathcal{S}'$ into $\mathcal{S}_< = \{s \in \mathcal{S}' \mid s < s_i^j\}$ and $\mathcal{S}_> = \{s \in \mathcal{S}' \mid s > s_i^j\}$. We then have $\mathcal{S}' = \mathcal{S}_< \cup \{s_i^j\} \cup \mathcal{S}_>$. For $\mathcal{S}_<$ the cases 1a or 1b can be applied. For $\mathcal{S}_>$ 1b or 1c is applicable. This yields the upper bound $|\mathcal{S}'| \leq (\omega - 1) + 1 + \omega \leq 2\omega$.

3. $k > 1$: Again, we split $\mathcal{S}'$. This time not only into 2 subsets as above but into $(k+1)$. We find $|\mathcal{S}'| \leq k\left((\omega-1)+1\right) + \omega \leq (k+1)\omega$.

□

With Lemma 3.1 we can prove the following Theorem 3.2.

Theorem 3.2 (Bucket Sizes for Regular Sampling)**.** *If regular sampling is used in* `partition` *with s as the number of locally sampled elements, then all buckets $\mathcal{B}^j$ contain less than $|\mathcal{S}_i|/s + |\mathcal{S}_i|/p$ elements.*

Figure 3.2 illustrates the idea of the proof. To give an upper bound for any bucket, say bucket $\mathcal{B}^2$, one can sum the number of elements of each set $\mathcal{S}_i$ that will be assigned to this bucket. Since $\mathcal{B}^2$ contains the elements between f_1 and f_2, we just have to give bounds for the number of elements between these two splitters in every array $\mathcal{S}_i$. In Figure 3.2 the thin black ticks represent the elements of $\mathcal{S}_i^{sample}$; the intervals between f_1 and f_2 are coloured in red.

Proof. We define the sets

$$\mathcal{B}_i^j := \begin{cases} \{s \in \mathcal{S}_i \mid s \le f_j\} \\ \{s \in \mathcal{S}_i \mid f_{j-1} < s \le f_j\} \\ \{s \in \mathcal{S}_i \mid f_{p-1} < s\} \end{cases} \text{ and } C_i^j := \begin{cases} \{s \in \mathcal{S}_i^{sample} \mid s \le f_j\} & \text{for } j = 1, \\ \{s \in \mathcal{S}_i^{sample} \mid f_{j-1} < s \le f_j\} & \text{for } 2 \le j \le p-1 \\ \{s \in \mathcal{S}_i^{sample} \mid f_{p-1} < s\} & \text{for } j = p \end{cases}$$

for $1 \le i \le p$. Let $c_i^j := \left|C_i^j\right|$ for $1 \le i \le p$ and $1 \le j \le p-1$. The set $\mathcal{B}_i^j$ contains all elements that are in $\mathcal{S}_i$ and bucket j. The variable c_i^j counts the number of elements in the sample set that are in $\mathcal{B}_i^j$. To give an upper bound on the number of elements in $\mathcal{B}^j$, we state upper bounds on $|\mathcal{B}_i^j|$ and sum over all PEs.

1. $j = 1$: We have $f_1 = \mathcal{S}^{sample}[s-1]$. Therefore, there are $s-1$ elements of the local sample sets smaller than f_1 plus the element itself, which leaves s elements of the local sample sets that are smaller than or equal to f_1. Hence, $\sum_{i=1}^p c_i^1 = s$.
2. $2 \le j \le p-1$: We have $f_{j-1} = \mathcal{S}^{sample}[(j-1)s-1]$ and $f_j = \mathcal{S}^{sample}[js-1]$. Hence, there are s elements of the local sample sets greater than f_{j-1} and smaller than or equal to f_j. We obtain $\sum_{i=1}^p c_i^j = s$.
3. $j = p$. Since $f_{p-1} = \mathcal{S}^{sample}[(p-1)s-1]$ there are s elements in $\mathcal{S}^{sample}$ greater than f_{p-1}. Therefore, $\sum_{i=1}^p c_i^p = s$.

In each set $\mathcal{B}_i^j$ there are c_i^j elements in $\mathcal{S}_i^{sample}$ by definition. Hence, Lemma 3.1 yields the upper bound of $(c_i^j+1)\omega$ elements for $|\mathcal{B}_i^j|$. Summing over all PEs yields

$$|\mathcal{B}^j| = \sum_{i=1}^p |\mathcal{B}_i^j| \le \sum_{i=1}^p (c_i^j + 1)\omega = \omega(p+s) = \frac{|\mathcal{S}|}{p(s+1)}(p+s) < \frac{|\mathcal{S}|}{ps}(p+s) = \frac{|\mathcal{S}|}{s} + \frac{|\mathcal{S}|}{p}.$$

for $1 \le j \le p$. □

3.3.3 String-Based Sampling

String-based sampling aims at choosing final splitters such that the number of strings in the constructed buckets $\mathcal{B}^j$ is equal. We can directly apply regular sampling as described in 3.3.2. In the following the algorithm `partition` with string-based regular sampling as sampling method will be denoted by `partitionSB`.

3.3.4 Character-Based Sampling

Character-based sampling as opposed to string-based sampling asks for buckets that contain about the same number of characters. Our character sampling approach is also based on regular sampling exploiting the sortedness of the input. Instead of sampling the string array, we switch to a view on the character-level and sample the according character array $\mathcal{C}(\mathcal{S}_i)$. Whereas a string array contains pointer to strings, the corresponding character array $\mathcal{C}(\mathcal{S}_i)$ contains directly the characters of the strings of $\mathcal{S}_i$ as a sequence. Therefore, we find $|\mathcal{C}(\mathcal{S}_i)| = \|\mathcal{S}_i\|$. Figure 3.3 shows an example of a character array.

For the sake of simplicity we assume – as above – that $\mathcal{C}(\mathcal{S}_i)$ is divisible by $s+1$ and contains at least s elements. In string-based sampling we have simply chosen the elements s_i^j at rank $j\omega - 1$ from the

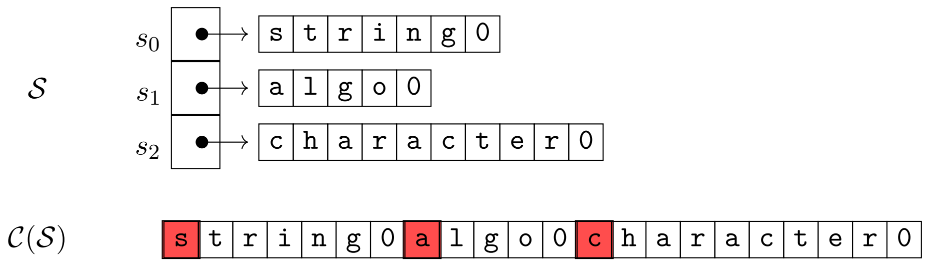

s_0
s_1
s_2
$\mathcal{S}$
s t r i n g 0
a l g o 0
c h a r a c t e r 0
$\mathcal{C}(\mathcal{S})$
s t r i n g 0 a l g o 0 c h a r a c t e r 0

Figure 3.3: String array $\mathcal{S}$ with corresponding character array $\mathcal{C}(\mathcal{S})$.

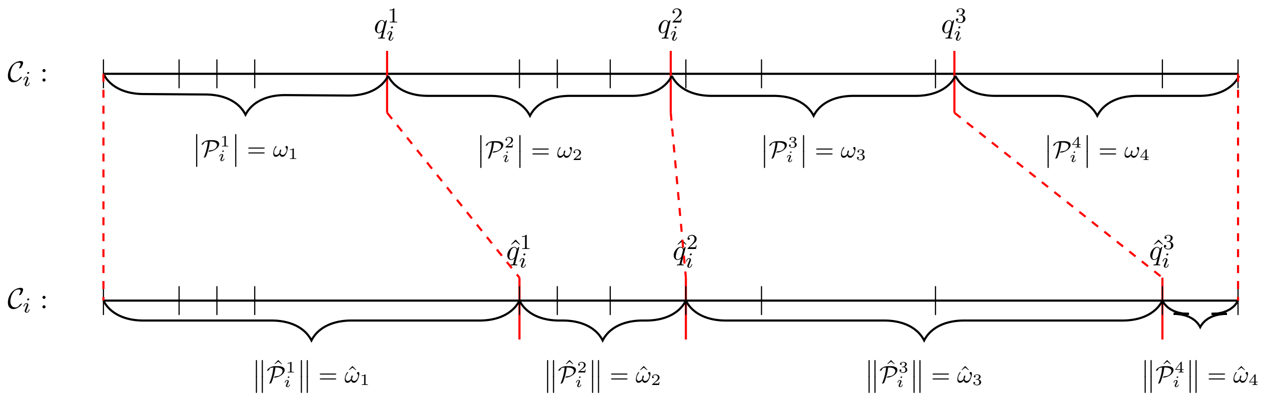

q_i^1
q_i^2
q_i^3
$\mathcal{C}_i$:
$|\mathcal{P}_i^1| = \omega_1$
$|\mathcal{P}_i^2| = \omega_2$
$|\mathcal{P}_i^3| = \omega_3$
$|\mathcal{P}_i^4| = \omega_4$
$\hat{q}_i^1$
$\hat{q}_i^2$
$\hat{q}_i^3$
$\mathcal{C}_i$:
$\|\hat{\mathcal{P}}_i^1\| = \hat{\omega}_1$
$\|\hat{\mathcal{P}}_i^2\| = \hat{\omega}_2$
$\|\hat{\mathcal{P}}_i^3\| = \hat{\omega}_3$
$\|\hat{\mathcal{P}}_i^4\| = \hat{\omega}_4$

Figure 3.4: Character-based sampling: Shift of the number controlled by the sample set elements when respecting string boundaries.

local input array as the elements of the local sample set $\mathcal{S}_i^{sample}$. For character-based sampling this procedure is changed. We define $q_i^j := j|\mathcal{C}(\mathcal{S}_i)|/(s+1) - 1$ for $1 \leq j \leq s$ and $1 \leq i \leq p$, i.e. q_i^j is an index in $\mathcal{C}_i$. Using these indices we partition the character array $\mathcal{C}(\mathcal{S}_i)$ into the following $s+1$ sets:

$$\mathcal{P}_i^j := \begin{cases} \mathcal{C}(\mathcal{S}_i)[0, q_i^1] & \text{for } j = 1, \\ \mathcal{C}(\mathcal{S}_i)(q_i^{j-1}, q_i^j] & \text{for } 2 \leq j \leq s, \\ \mathcal{C}(\mathcal{S}_i)(q_i^s, |\mathcal{C}(\mathcal{S}_i)| - 1] & \text{for } j = s+1. \end{cases}$$

Note that $\mathcal{P}_i^j$ does not necessarily contain whole strings only. The first and last string in these sets might be broken since the indices q_i^j do not respect string boundaries. This can be overcome by defining $\hat{q}_i^j$ for each q_i^j as the index of the last character in $\mathcal{C}(\mathcal{S}_i)$ of the string to which the character $\mathcal{C}(\mathcal{S}_i)[q_i^j]$ belongs. Since $\hat{q}_i^j$ indexes the last character of a string, we can take these strings as the elements of the sample set $\mathcal{S}_i^{sample}$. Figure 3.4 shows an example. The thin black ticks represent the string boundaries of $\mathcal{C}(\mathcal{S}_i)$. The larger red ticks q_i^1, q_i^2 and q_i^3 mark the splitting indices used to partition $\mathcal{C}(\mathcal{S}_i)$ without taking string boundaries into consideration. The sets $\mathcal{P}_i^j$ into which the characters of $\mathcal{C}(\mathcal{S}_i)$ would be divided by q_i^1, q_i^2 and q_i^3 are of the same size, i.e. they all consist of exactly ω characters. When we shift the splitters such that string boundaries are respected (now the shifted splitters $\hat{q}_i^1, \hat{q}_i^2$ and $\hat{q}_i^3$ are at the end of real strings); the partition obtained is $\hat{\mathcal{P}}_i^1, \ldots, \hat{\mathcal{P}}_i^4$ with varying number of characters $\hat{\omega}_1, \ldots, \hat{\omega}_4$. Note the difference between $\mathcal{P}_i^j$ and $\hat{\mathcal{P}}_i^j$. The former is a character array, the latter can be interpreted as a string array. This example illustrates that the number of characters of each set of the partition depends heavily on the string length when applying (regular) sampling on a character-level. Note that the sets $\hat{\mathcal{P}}_i^j$ contain the same strings as $\mathcal{S}_i^j$ defined by

$$\mathcal{S}_i^j := \begin{cases} \{s \in \mathcal{S}_i \mid s \leq s_i^1\} & \text{for } j = 1, \\ \{s \in \mathcal{S}_i \mid s_i^{j-1} < s \leq s_i^j\} & \text{for } 2 \leq j \leq s, \\ \{s \in \mathcal{S}_i \mid s_i^s < s\} & \text{for } j = s+1 \end{cases},$$

with s_i^j being the j^{th} element of the $\mathcal{S}_i^{sample}$ for $1 \leq i \leq p$. Algorithm 3.1 describes the sampling process in pseudo-code. It is clear that the maximum string length directly interferes with the number of characters per bucket $\mathcal{B}^j$. To get an upper bound on this number, we need to bound the length of the longest string in $\mathcal{S}_i$.

Lemma 3.3 (Maximum Partition Size)**.** *Let l_{max} be the length of the longest string in $\mathcal{S}_i$ and let $\|\mathcal{S}_i\|$ be divisible by $s+1$. If $l_{max} \leq \|\mathcal{S}_i\|/(s+1)$, the sample set obtained by Algorithm 3.1 with the parameters $(\mathcal{S}_i, s)$ partitions $\mathcal{S}_i$ into $s+1$ (non-empty) buckets. The maximum number of characters in each partition $\mathcal{S}_i^j$ is less than or equal to $\|\mathcal{S}_i\|/(s+1) + l_{max}$.*

Proof. Let $l_{max} \leq \|\mathcal{S}_i\|/(s+1) := \omega$. The initially chosen indices q_i^j have a distance of ω. Therefore, there are no two characters $\mathcal{C}(\mathcal{S}_i)[q_i^j]$, $\mathcal{C}(\mathcal{S}_i)[q_i^{j+1}]$ belonging to the same string. It follows that $\hat{q}_i^j \neq \hat{q}_i^k$ for $1 \leq j, k \leq s$ and $k \neq j$. This implies that the loop in lines $7-9$ of the algorithm is executed for one iteration at a time only. Furthermore, we have $q_i^j \leq \hat{q}_i^j \leq q_i^j + l_{max}$ as we shift the initial indices to the end of the string. With this, it holds that

- $\hat{q}_i^1 \leq \omega + l_{max}$.
- $\hat{q}_i^j - \hat{q}_i^{j-1} \leq q_i^j + l_{max} - q_i^{j-1} = \omega + l_{max}$, for $2 \leq j \leq s$.
- $\|\mathcal{S}_i\| - \hat{q}_i^s \leq (s+1)\omega - s\omega = \omega$.

Hence, $\left\|\hat{\mathcal{P}}_i^j\right\| \leq \omega + l_{max}$ as we assume that each string has at least length 1 (its end-of-string character) for all $1 \leq j \leq s+1$. It follows that $\left\|\mathcal{S}_i^j\right\| \leq \omega + l_{max}$, i.e. all sets into which $\mathcal{S}_i^{sample}$ divides $\mathcal{S}_i$ contain less than or equal to $\omega + l_{max}$ characters, if $l_{max} \leq \omega$. Furthermore, they also contain at least one string as all strings in the sample set are distinct. □

Now we can state Lemma 3.4 – a character-based version of Lemma 3.1.

Lemma 3.4 (Character-Based Number of Sample Elements)**.** *We define $\omega := \|\mathcal{S}_i\|/(s+1)$. Let $l_{max} \leq \omega$ be the length of the longest string in $\mathcal{S}_i$. Let $\mathcal{S}' = \{s \in \mathcal{S}_i \mid a \leq s \leq b\}$ be a contiguous subarray of $\mathcal{S}_i$. If $\left|\mathcal{S}' \cap \mathcal{S}_i^{sample}\right| = k$, then $\|\mathcal{S}'\| \leq (k+1)(\omega + l_{max})$.*

Proof. First, we define $\omega_{max} := \omega + l_{max}$.

1. $k = 0$: If $k = 0$, then all elements of $\mathcal{S}'$ are completely contained within one of the sets $\mathcal{S}_i^j$ for $1 \leq j \leq s+1$. Hence, $\|\mathcal{S}'\| \leq \omega_{max}$ by Lemma 3.3.
2. $k = 1$: Let s_i^j be the element of the local sample set contained in $\mathcal{S}'$. We split $\mathcal{S}'$ into the sets $\mathcal{S}'_{\leq}$ containing all strings less than s_i^j and s_i^j and the set $\mathcal{S}'_{>}$ containing the string greater than s_i^j. By Lemma 3.3, $\mathcal{S}'_{\leq}$ can contain at most ω_{max} characters. The same holds for $\mathcal{S}'_{>}$ as the first case is applicable. Hence, $\mathcal{S}'$ contains at most $2\omega_{max}$ characters.
3. $k \geq 1$: Let $s_i^{j+1}, \ldots, s_i^{j+k}$ be the local sample set elements contained in $\mathcal{S}'$. We define the following sets:

$$\mathcal{S}'_t = \begin{cases} \{s \in \mathcal{S}' \mid s \leq s_i^{j+1}\} & \text{for } t = 1 \\ \{s \in \mathcal{S}' \mid s_i^{j+t-1} < s \leq s_i^{j+t}\} & \text{for } 1 < t \leq k \\ \{s \in \mathcal{S}' \mid s_i^t < s\} & \text{for } t = k+1. \end{cases}$$

   With these we can split $\mathcal{S}'$ in $k+1$ sets, i.e. $\mathcal{S}' = \mathcal{S}'_j \cup \cdots \cup \mathcal{S}'_k \cup \mathcal{S}'_{j+k+1}$. For all the sets $\mathcal{S}'_t$ with $1 \leq t \leq k+1$ Lemma 3.3 can be applied as they are all subsets of one of the partition-sets $\mathcal{S}_i^j$. Thus, we have $\|\mathcal{S}'\| = \sum_{i=j+1}^{j+k+1} \|\mathcal{S}'_i\| \leq (k+1)(\omega_{max})$.

Algorithm 3.1: Character Sampling

```
Input: On each PE: a sorted string array $\mathcal{S}_i$ and the number $s$ of elements to be drawn
       from $\mathcal{S}_i$ .
1 $j \leftarrow 0,\ c \leftarrow 0$
2 $\omega \leftarrow \frac{\|\mathcal{S}_i\|}{s+1}$                                   // splitter distance
3 for $k \leftarrow 0$ to $s-1$ do
4     while $c < (k+1)\omega$ do
5         $c \leftarrow c + |\mathcal{S}_i[j]|$
6         $j \leftarrow j+1$
7     while $(k+1)\omega \leq c \wedge k < s$ do   // if $l_{max} \leq \omega$ only one iteration within loop
8         $\mathcal{S}_i^{sample}[k] \leftarrow \mathcal{S}_i[j-1]$
9         $k \leftarrow k+1$
Output: A sample set $\mathcal{S}_i^{sample}$ of size $s$.
```

□

With Lemma 3.4 the following Theorem 3.5 can be proven similarly to Theorem 3.2.

Theorem 3.5. *Let s be the number of elements in $\mathcal{S}_i^{sample}$. If we have the same number of characters on all PEs, i.e. $\|\mathcal{S}_i\| = \|\mathcal{S}\|/p$ and $l_{max} \leq \|\mathcal{S}_i\|/(s+1)$, then the maximum number of characters in a bucket $\mathcal{B}^j$ is less than $\|\mathcal{S}\|/p + \|\mathcal{S}\|/s + (p+s)l_{max}$.*

Proof. Let with $\omega := ||\mathcal{S}||/(p(s+1))$. Using Lemma 3.4 within the same argument as in Theorem 3.2 yields

$$||\mathcal{B}^j|| = \sum_{i=1}^{p} ||\mathcal{B}_i^j|| \leq \sum_{i=1}^{p} (c_i^j + 1)(\omega + l_{max}) = (\omega + l_{max})(p+s) < \frac{||\mathcal{S}||}{s} + \frac{||\mathcal{S}||}{p} + (p+s)l_{max}.$$

□

In Theorem 3.5 we demand $l_{max} \leq \omega$. If we define $\gamma = l_{max}/\omega$, we can rewrite the upper bound of this theorem as

$$||\mathcal{B}^j|| < (1+\gamma)\left(\frac{||\mathcal{S}||}{p} + \frac{||\mathcal{S}||}{s}\right).$$

In the following we will refer to `partition` using character-based regular sampling in the sampling step as `partitionCB`.

3.3.5 Local Input Arrays of Different Size

Thus far, we have assumed that the number of strings/the number of characters is the same on all local input arrays $\mathcal{S}_i$. We will now discuss how the algorithm must be adapted to handle local input arrays of varying size while preserving the above-stated guarantees for the bucket size/the number of characters per bucket. We describe the adaption of the algorithm for `partitionSB` only. Character-based sampling can be adapted analogously. For the sake of simplicity assume that the number of strings on each PE is a multiple of $\omega := |\mathcal{S}|/(p(s+1))$. As before the number of elements in $\mathcal{S}^{sample}$ remains the same. The key difference is that the PEs no longer choose s strings for their local sample set but $|\mathcal{S}_i|/\omega$ elements. This way, every element in $\mathcal{S}^{sample}$ "controls" ω strings as before and we obtain the same guarantees on the number of strings per bucket. To compute the value of ω, an allreduce is executed in which the number of strings of each PE's input string array is summed up. Let S be this sum. Then we have $\omega = S/p(s+1)$. Therefore, the root PE divides this S by $p(s+1)$ and broadcasts the result back to the other PEs.

3.3.6 Running Time

We have not discussed the running time of the algorithm `partition` as described in Section 3.3.1 yet. In order to get an upper bound on the running time of a distributed algorithm, we have to analyse its critical path (also called depth), i.e. the longest sequence of operations on any PE that needs to be executed sequentially (see [7] for details). The length of the longest string in $\mathcal{S}$ is l_{max}. Although we use hypercube quicksort as distributed string sorting algorithm within `partition` in our implementation, we will not use it for this analysis. The reason is that Axtmann and Sanders [1] do not prove any bounds on the running time of the algorithm on average or in the worst case. Therefore, we assume a black box distributed sorter with a running time of $T_{sort}(n, l)$ if each PE has an input string array consisting of at most n strings with a maximum length of l characters.

1. **Local Sampling:** The sampling of the input to obtain the local sample set can be done in $\mathcal{O}(\delta s)$ for string-based sampling as at most δs strings are drawn from the input and as the construction of the string array representing the sample set is in $\mathcal{O}(1)$ per inserted string as only pointers need to be copied. For character-based sampling we must know the length of the strings. Therefore, this step is in $\mathcal{O}(\Delta\|\mathcal{S}\|/p)$ for character-based sampling. However, if the string length of all strings is already determined, sampling can be executed in $\mathcal{O}(\delta\mathcal{S}/p)$ as a scan over the local string array is sufficient (see Algorithm 3.1).
2. **Sorting $\mathcal{S}^{sample}$:** In this step all elements of the local samples must be augmented with their respective ranks in $\mathcal{S}^{sample}$ such that the final splitters can be extracted in the next step. Using a black box distributed sorter yields a running time of $T_{sort}(\delta s, l)$ for string-based sampling and a running time of $T_{sort}(\Delta s, l)$ for character-based sampling as δs (and Δs respectively) are the maximum number of strings belonging to a local sample set.
3. **Extraction and Distribution of Splitters:** The sorted set $\mathcal{S}^{sample}$ is distributed over the PEs. As describe above, a prefix sum over the number of elements in the output of the previous sorting step is sufficient to calculate the positions of the final splitters. This is in $\mathcal{O}(\alpha \log(p) + \beta w)$. We assume the black box sorter to deliver a balanced output, i.e. each PE contains only a constant number of the $p-1$ final splitters. The allgather operation to make all final splitters available to all PEs is therefore in $\mathcal{O}(\alpha \log(p) + \beta cpl_{max})$
4. **Determination of Buckets:** On PE i we use $p-1$ binary searches to determine the intervals into which the final splitters f_j divide the string array $\mathcal{S}_i$. Since the comparison of two strings s_1 and s_2 is in $\mathcal{O}(\min(|s_1|, |s_2|))$, the time complexity of this step is in $\mathcal{O}((p-1)\log(\delta|\mathcal{S}|/p) l_{max})$.

3.4 LCP Compression

In our model of distributed computing every sent data unit of size m incurs costs of at least βm. To reduce the amount of communication, compression techniques can be used. However, there is always a trade-off between the (local) time complexity of a compression/decompression algorithm and the time that is saved by the reduced communication volume. In the following we describe a simple technique which compresses the common prefixes of a sorted string array $\mathcal{S}$ using the associated LCP array $\mathcal{H}(\mathcal{S})$.

The idea is to send each common prefix of $\mathcal{S}$ only once. Let $s_i = \mathcal{S}[i]$ and $h_i = \mathcal{H}(\mathcal{S})[i]$ for $1 \le i < |\mathcal{S}|$. Then h_i denotes the length of the longest common prefix of s_i and its predecessor in $\mathcal{S}$. Hence, $s[h_i]$ is the first character in which s_i and is predecessor differ. Algorithm 3.2 uses this idea and computes the compressed string array

$$\mathcal{S}' = [s_0, s_1[h_1,], s_2[h_2,] \ldots, s_{|\mathcal{S}|-1}[h_{|\mathcal{S}|-1},]].$$

By construction, we have $\|\mathcal{S}'\| = \|\mathcal{S}\| - \mathcal{L}(\mathcal{S})$. The first string is always uncompressed as we interpret the undefined first entry of $\mathcal{H}(\mathcal{S})$ as 0.

Algorithm 3.2: LCP Compression

```
Input: A string array $\mathcal{S}$ and the corresponding LCP array $\mathcal{H}(\mathcal{S})$ with $\mathcal{H}(\mathcal{S})[0] = 0$ instead
       of $\perp$.
1 if $|\mathcal{S}| > 0$ then
2     $\mathcal{S}'[0] \leftarrow \mathcal{S}[0]$;
3 for $i \leftarrow 1$ to $|\mathcal{S}| - 1$ do
4     $(s, h) \leftarrow (\mathcal{S}[i], \mathcal{H}(\mathcal{S})[i])$;
5     $s' \leftarrow s[h, ]$ ;          // copy string without common prefix to previous string
6     $\mathcal{S}'[i] \leftarrow s'$;
Output: Compressed string array $\mathcal{S}'$ with $\|\mathcal{S}'\| = \|\mathcal{S}\| - \mathcal{L}(\mathcal{S})$
```

Algorithm 3.3: LCP Decompression

```
Input: An LCP-compressed string array $\mathcal{S}'$ and the LCP array $\mathcal{H}(\mathcal{S})$ of the corresponding,
       uncompressed string array $\mathcal{S}$ of $\mathcal{S}'$.
1 if $|\mathcal{S}'| > 0$ then
2     $\mathcal{S}^{out}[0] \leftarrow \mathcal{S}'[0]$ ;                    // first string is always uncompressed
3 for $i \leftarrow 1$ to $|\mathcal{S}'| - 1$ do
4     precondition: $\mathcal{S}^{out}[i-1] = \mathcal{S}[i-1]$;
5     $h \leftarrow \mathcal{H}(\mathcal{S})[i]$;
6     $(a, b) \leftarrow (\mathcal{S}^{out}[i-1], \mathcal{S}'[i])$;
7     $s \leftarrow a[0, h) + b[0, ]$;
8     $\mathcal{S}^{out}[i] \leftarrow s$;
Output: String array $\mathcal{S}^{out}$ with $\mathcal{S}^{out} = \mathcal{S}$.
```

The original string array S can be reconstructed with $\mathcal{S}'$ and the original LCP array $\mathcal{H}(\mathcal{S})$ as inputs to Algorithm 3.3. This algorithm iterates over all strings in $\mathcal{S}'$ beginning with the second string in the compressed string array. Say $s_i' := \mathcal{S}'[i]$ with $1 \leq i \leq |\mathcal{S}| - 1$ is the next string to be unpacked. We assume the precondition of the loop in lines 3 – 8 to be true. Then we find $\mathcal{S}[i] := \mathcal{S}^{out}[0, h_i) + \mathcal{S}'[0,]$ with $h_i = \mathcal{H}(\mathcal{S})[i]$ as $\mathcal{S}[i-1][0, h_i)$ is the longest common prefix of $\mathcal{S}[i]$ and $\mathcal{S}[i-1]$. The string $\mathcal{S}^{'}[i-1]$ is already restored as the precondition is true for i. Hence, we can restore string $\mathcal{S}[i]$. After that the precondition holds for $i+1$, too. Since the first string of $\mathcal{S}'$ is always uncompressed, the precondition is also true for $i = 1$. By the principle of induction, this proves that Algorithm 3.3 reconstructs $\mathcal{S}$ correctly.

Since both, compression and decompression, do just one iteration over the data and copy each character at most once, the running time is in $\mathcal{O}(\|\mathcal{S}\|)$.

3.5 Multiway-(LCP)-Merging

In both of our main algorithms each PE will receive a bucket consisting of up to p sorted string arrays that need to be merged (see Section 4 and Section 5). This problem is known as multiway merging and can be solved by a losertree [15]. In our setting we additionally know the associated LCP array to each of the received sorted sequences. In the following section we describe the *LCP losertree* introduced by Bingmann et al. [6] and its essential subroutine *LCPCompare* proposed by Ng and Kakehi [19]. This adaption of the commonly used losertree exploits the information from the LCP arrays to save character comparisons in the merging process.

Algorithm 3.4: LCP Compare

```
Input: Two strings $s_0$ and $s_1$ and additional LCP values $h_0 = \text{LCP}(s, s_0)$ and
       $h_1 = \text{LCP}(s, s_1)$ with $s \leq s_0, s_1$.
if $h_0 < h_1$ then
    return $(1, h_0)$
else if $h_1 < h_0$ then
    return $(0, h_1)$
$h \leftarrow h_1$;
while $s_0[h] = s_1[h] \wedge s_0[h] \neq 0$ do
    $h \leftarrow h + 1$;
if $s_0[h] \leq s_1[h]$ then
    return $(0, h)$
else
    return $(1, h)$
Output: $(x, h)$ with $x \in \{0, 1\}$ and $s_x \leq s_{1-x}$ and $h = \text{lcp}(s_x, s_{1-x})$
```

3.5.1 LCP Compare

One of the key difference between atomic key sorting and string sorting is the time needed for one element comparison. With atomic keys this is clearly in $\mathcal{O}(1)$ but comparing two strings s_0 and s_1 can take up to $\min(|s_0|, |s_1|)$ character comparisons. However, these comparisons can be considerably accelerated if there is more information about the strings. Assume we know $\text{LCP}(s_0, s)$ and $\text{LCP}(s_1, s)$ where s is a string with $s \leq s_0$ and $s \leq s_1$. There are three cases:

1. $\text{LCP}(s, s_0) = \text{LCP}(s, s_1)$: Let $h := \text{LCP}(s, s_0)$. It holds that $s_0[0, h) = s_1[0, h)$ as both strings share the prefix $s[0, h)$. Therefore, we can start comparing both strings at the $(h + 1)^{th}$ character saving h character comparisons.

2. $\text{LCP}(s, s_0) < \text{LCP}(s, s_1)$: Let $h := \text{LCP}(s, s_0)$. Since $s \leq s_0$, we know that $|s_0| > h$. As we have $s \leq s_0$, it holds that $s[h] < s_0[h]$. Otherwise $\text{LCP}(s, s_0) \geq h + 1$ would hold. Since $\text{LCP}(s, s_1) > h$, we have $s[h] = s_1[h]$ and therefore $s_1[h] < s_0[h]$. With $s_0[0, h) = s_1[0, h)$ (see case 1) and $s_1[h] < s_0[h]$, it follows that $s_1 < s_0$.

3. $\text{LCP}(s, s_1) < \text{LCP}(s, s_0)$: The same argument as above can be applied vice versa. Thus, we find $s_0 < s_1$.

The `LCPCompare` subroutine exploits this observation. See Algorithm 3.4 for a detailed description.

Using this tool we can augment the losertree data structure with LCP values.

3.5.2 Losertree

A loser- or tournament tree is a data structure allowing the efficient merging of up to K sorted sequences of elements. If K is not a power of two, additional empty sequences are added until the number of sequences is a power of two. The last element of each sequence is a sentinel element greater than all other elements. Our setting is comparable to a tournament with K players, in which we want to determine the best player (i.e. the smallest element). This can be done using a complete binary tree with the players being the leaves of this tree. Figure 3.5 shows such a tree with 8 players s_1 to s_8 (the various variables denoted by characters h or H become important only in the LCP-extension of the losertree described in the next paragraph). Each inner vertex of the tree represents a match of two players (vertices v_8 to v_2). The winner of a match is promoted to the next level and the loser y_i stays in the vertex v_i in which it has lost the match. After playing the match of the root vertex v_2, we know the best of all K players, i.e. in our case the smallest element.

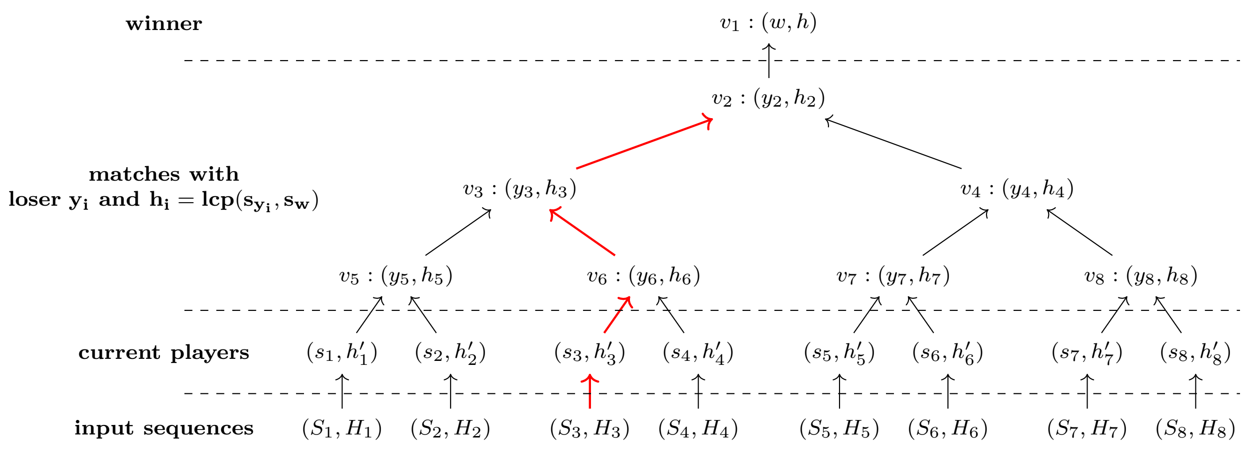


Figure 3.5: LCP Losertree with eight input sequences.

Unlike in a normal tournament, the procedure described thus far is just the initialization phase of our merging algorithm. Instead of K players, we have K *sequences* (S_1 to S_8) of players ordered increasingly.

After the initialization phase, we have determined the overall smallest element of all sequences. Say this element is from sequence i. It can be removed from its sequence, be written to the output, and the next element from sequence i takes part in the tournament. Furthermore, we know that the loser element in each vertex is the smallest element of the subtree of the losertree from which it was promoted to its current position. Therefore, this time, only the $\log(K)$ games corresponding to vertices in the path of the former winner from its leaf to the root must be replayed to find the second smallest element. As before, the loser of each match stays on the vertex and the winner is promoted to the next level. To make things more clear, we take the losertree from Figure 3.5 as example. Say the winner w of the last round is from sequence 3. Then only the matches along w's path from sequence 3 up to the root vertex v_2 depicted in red must be replayed. The vertices v_8 to v_2 plus the leaf s_3 currently hold the K remaining smallest elements of all sequences (recall that the sequences are sorted). Furthermore, for each inner vertex the loser element represents the smallest element currently in the subtree from which it has been promoted to its current position. For example the element y_2 has been promoted from v_4 to its current position. Additionally, the subtree of vertices not on path p does not change with the removal of w and the insertion of element in leaf s_3. Therefore, the only necessary new comparisons are on this path. By replaying this path bottom up we assure the element winning this tournament is the second smallest overall element. The algorithm continues until all sequences are empty. Thus, to merge all elements $K - 1$ (initialization phase) plus $(n - 1) \cdot \log(K)$ (remaining elements) string comparisons are needed to merge the K sorted sequences.

3.5.3 LCP Losertree

Now, we want to integrate the idea of the `LCPCompare` algorithm into the plain losertree. The changes required in the plain losertree to enable the `LCPCompare` subroutine are rather simple. As input we do not only need the sorted sequences of strings but also the corresponding LCP array H_i of each sequence. Additionally, we do not only store the loser of a comparison between the strings x, y_i in a vertex v_i of the tree but also the information $h = \text{LCP}(x, y_i)$, i.e. each vertex contains a tuple $(\min(x, y_i), h)$. Consider the winner string w of a round and its path p from the leaf associated with its sequence to the root. In the next round of the losertree only vertices of this path p are replayed. Let v_i be the next vertex being played on p, y_i the loser string stored on v_i and $h_i = \text{LCP}(y_i, w)$ the corresponding LCP value. By c we denote the contender arriving at v_i. The contender is either the next string from the origin sequence of w or a loser string, i.e. a string against w has won while traversing path p. In both cases we already know the LCP value $h' = \textsc{lcp}(c, w)$:

1. c is the next element from the origin sequence of w: the LCP value $\textsc{lcp}(c, w)$ is the corresponding entry in H_i by definition.
2. c is a loser string from a vertex v_j on p: the LCP value stored in v_j is $h = \textsc{lcp}(c, w)$. Therefore, h cannot be an other LCP value since the last match played on vertex v_j has been w against c as it is on p.

Hence, in all necessary matches we know the values of $\textsc{lcp}(c, w)$ and $\textsc{lcp}(y_i, w)$ and $w \leq c, y_i$. Therefore, we can apply the `LCPCompare` subroutine on the pairs $(c, \textsc{lcp}(c, w))$ and $(y_i, \textsc{lcp}(y_i, w))$ to save character comparisons. The question remains, how the procedure must be changed during initialization as there is no former winner which can be used as a reference. But we have $\textsc{lcp}(\epsilon, s) = 0$ for all strings s. Therefore, considering the empty string ϵ as the overall smallest elements enables us to use the same comparison function even during the initialization phase. Theorem 3.6 states the maximum number of character comparisons executed while merging K sequences of sorted string with a LCP losertree.

Theorem 3.6 (Complexity of LCP-K-way-merging [6]). *A LCP losertree needs at most $\Delta L + |\mathcal{S}_0| \log(K) + K$ character comparisons where $\mathcal{S}_0$ is the merged output sequence and $\Delta L = \mathcal{L}(\mathcal{S}_0) - \sum_{k=1}^{K} \mathcal{L}(\mathcal{S}_k)$ is the sum of increments to LCP array entries.*

Using the LCP losertree algorithm instead of a plain losertree not only saves characterwise comparisons, it can also be used without unpacking LCP compressed sequences as described in Section 3.4.

4. Distributed Merge String Sort

Before describing our distributed Merge String Sort algorithm (dMSS), we state some conventions about the input and output of the algorithm:

Input

1. PE i obtains a string array $\mathcal{S}_i$ as input. By $\mathcal{S}$, we denote the union of all local input arrays, i.e. $\mathcal{S} = \bigcup_{1 \le i \le p} \mathcal{S}_i$. Although the term “union” is used, duplicates are allowed in $\mathcal{S}$.
2. The number of strings and characters can be different in the local string arrays. We assume that
$$|\mathcal{S}_i| \le \delta \frac{|\mathcal{S}|}{p} \text{ and } \|\mathcal{S}_i\| \le \Delta \frac{\|\mathcal{S}\|}{p}$$
holds for $\delta, \Delta \ge 1$, i.e. δ and Δ characterize the maximum imbalance of the input.
3. By l_{max}, we denote the length of the longest string in $\mathcal{S}$.

Output

PE i outputs a string array $\mathcal{O}_i$, which has the following two properties:

1. Let $0 \le j < j' < |\mathcal{O}_i|$. Then $\mathcal{O}_i[j] < \mathcal{O}_i[j']$ holds, i.e. the output string array is locally sorted.
2. Let j be the index of another PEs with $i < j$, then for all strings $s_i \in \mathcal{O}_i$ and all strings $s_j \in \mathcal{O}_j$ we find $s_i \le s_j$, i.e. the output sets are also globally sorted.

The number of strings and characters in the output arrays depends on the sampling method used in the distributed ordered partitioning sub-step of dMSS. However, the union of all output arrays contains exactly the same strings as $\mathcal{S}$.

4.1 Description of the Algorithm

The four main steps of the algorithm are identical to the distributed string sorting algorithm by Fischer and Kurpicz [11] (see Figure 2.2 for an illustration of the algorithm). However, we will apply multiple string-related optimizations to the original algorithm.

1. **Local Sorting:** Each PE sorts its input string array locally. Additionally, the LCP array of the local string array is computed as a by-product of the sorting process. These LCP values will be used later to omit repeated costly character comparisons and to reduce the communication volume.
2. **Distributed Ordered Partitioning:** Afterwards, all strings are assigned to one of p buckets $\mathcal{B}^j$ with $1 \leq j \leq p$. All strings in bucket $\mathcal{B}^{j'}$ are smaller than or equal to any string in bucket $\mathcal{B}^j$ for $j' < j$. Either the algorithm `partitionSB` or the algorithm `partitionCB` can be used for this task.
3. **String Exchange**
   Then the PEs perform an alltoall exchange of the strings. On PE i all strings and the associated LCP values that are assigned to $\mathcal{B}^j$ are packed in a message and are sent to PE j. Hence, after the alltoall exchange all strings assigned to bucket $\mathcal{B}^i$ in the previous step are located on PE i. These strings build the output array $\mathcal{O}_i$ of PE i. The strings are already globally sorted by construction of the buckets, however, they are not locally sorted yet, as each bucket consists of up to p sorted sequences of strings (and the associated LCP values).
4. **Merging:** Since the buckets contain up to p sorted sequences of strings and the associated LCP values, we can apply a LCP-multiway-merge algorithm to sort the string array $\mathcal{O}_i$ also locally.

In the following we describe the four steps of the dMSS in more detail. Additionally we state the time complexity for each of the sub-steps. Since there is communication and, thus, synchronization across PEs in or after each sub-step we will state the maximum time complexity over all PEs for each of the sub-steps.

4.1.1 Local Sorting

We use MSD string radix sort as described in Section 3.1 to sort the local string array. After this step the local arrays $\mathcal{S}_i$ are sorted and we know $\mathcal{H}(\mathcal{S}_i)$. The expected time complexity on any PE is in

$$\mathcal{O}\left(\delta\frac{|\mathcal{S}|}{p}\log(\sigma) + \max_{i=1}^{p}\mathcal{D}(\mathcal{S}_i)\right).$$

4.1.2 Distributed Ordered Partitioning

In the original algorithm by Fischer and Kurpicz each PE samples its input set by drawing $p-1$ evenly spaced strings and then applies *centralized sequential sorting* to them as described in Section 3.2.1 with the only difference that they use an allgather instead of a gather operation. Afterwards, the final splitters are selected and used to determine the partition buckets. With `partitionSB` and $s = p-1$ we have an conceptually equivalent algorithm; however, we use a distributed sorter instead of centralized sequential sorting to sort the local sample sets. This becomes especially important if p is large as the global sample set consists of $ps = p(p-1)$ elements. Note that for $|\mathcal{S}| = o(p^3)$ centralized sorting requires more time than the local sorting of the input performed in the first step of the algorithm (assuming that strings and distinguishing prefixes are evenly distributed over all PEs). In our evaluation it will become clear that the original algorithm does not scale on various data sets because of the centralized sequential sorting approach within its partitioning algorithm.

The algorithm `partitionSB` computes a partition of $\mathcal{S}$ such that all sets (also called buckets) of the partition contain about the same number of strings. More precisely, we find $|\mathcal{B}^j| \leq |\mathcal{S}|/p + |\mathcal{S}|/s$ for bucket $\mathcal{B}^j$ and $1 \leq j \leq p$ where s is the second parameter of `partitionSB`.

However, our goal is to compute buckets such that the workload per PE is nearly equal in the subsequent steps – string exchange and merging – of the algorithm. The time complexity of these steps does not depend on the number of strings alone. Especially the time required for string exchange largely depends on the number of characters each PE receives.

Therefore, we designed `partitionCB` – a variant of `partition` that uses character-based sampling. With this sampling method the number of characters per bucket can be bounded. We obtain $\|\mathcal{B}^j\| \leq (\|\mathcal{S}\|/p + \|\mathcal{S}\|/s) \cdot (1+\gamma)$ if the longest string in $\mathcal{S}$ has a length smaller than or equal to $\gamma\|\mathcal{S}\|/(p(s+1))$ for $1 \leq j \leq p$.

Recall that the number of strings (with `partitionSB`) and the number of characters (for `partitionCB`) per bucket is independent of the imbalance factors δ and Δ of the input.

As already pointed out in the introduction, a major problem of distributed string sorting is posed by very long strings, especially, if their distinguishing prefix is short. For distributed ordered partitioning it would be sufficient to communicate the distinguishing prefixes of the sample sets only as they contain all the information required for the computation of the buckets. However, we do not know these prefixes. What we do know is the value of $\mathcal{D}(\mathcal{S}_i)$. As an approximation of the actual distinguishing prefix length we compute $\tilde{d} = \sum_{1 \leq i \leq p} \mathcal{D}(\mathcal{S}_i)$ and only communicate the prefix of length $\psi\tilde{d}/|\mathcal{S}|$ of the locally sampled strings, where ψ is a tuning parameter instead of the whole string. However, this can deteriorate the bounds on the number of strings and characters per bucket arbitrarily if the value of our approximation is too small for many elements of the sample sets. Therefore, this heuristic should only be applied if $\mathcal{D}(\mathcal{S}_i) \approx \mathcal{D}(\mathcal{S})/p$ for all $1 \leq i \leq p$. The running time of this step is as stated in Section 3.3.6.

4.1.3 String Exchange

After the determination of the buckets they must be sent to their destination. This is achieved by one (irregular) alltoall exchange. There are two variants of this exchange. The first simply exchanges the strings as they are. In the second variant we exploit our knowledge of the LCP values of the input string array after the sorting step. With that information we can compress strings with common prefixes such that the amount of communication can be reduced. The actual amount of communication that can be saved by this step depends on the characteristics of the input. Alongside the strings we also communicate the associated LCP values. The algorithm by Fischer and Kurpicz simply exchanges the strings without LCP values or compression.

Alltoall without compression:

The amount of bits sent by a PE is bounded by

$$C := \Delta\frac{\|\mathcal{S}\|}{p}c + \delta\frac{|\mathcal{S}|}{p}w,$$

where w is the number of bits required to encode one LCP value and c is the number of bits necessary to encode a character.

The number of bits received by a PE i depends on the number of characters in bucket $\mathcal{B}^i$. Using string-based sampling, we can only give upper bounds for the number of *strings* in such a bucket. The maximum number of strings in a bucket is expressed as $|\mathcal{S}|/p \cdot (1+\epsilon)$, i.e. as the *relative deviation* from the optimal value. For regular sampling with a sampling rate of $s \geq p$ we have $\epsilon \leq 1$ by Theorem 3.2. Therefore, each PE receives at most $|\mathcal{S}|/p \cdot (1+\epsilon)\,(cl_{max} + w)$ bits. This upper bound is tight if all strings are of the same length l_{max}.

For character-based sampling we can give better upper bounds even if there is a high variance in string length. The number of *characters* is at most $\|\mathcal{S}\|/p \cdot (1+\epsilon)$. For regular sampling with $s \geq p$ and $l_{max} \leq \|\mathcal{S}\|/(p(s+1))$ we find $\epsilon \leq 2$ by Theorem 3.5. Thus, the maximum number of received bits is bounded by

$$\frac{||\mathcal{S}||}{p}(1+\epsilon)c + \max_{1 \leq i \leq p}(|\mathcal{B}^i|)w \leq \frac{||\mathcal{S}||}{p}(1+\epsilon)(c+w).$$

This holds as each string has length at least 1 (end-of-string character). Using the 2-phase algorithm for irregular alltoall communication we obtain a communication time in

$$\mathcal{O}\left(\alpha p + \beta\left(\max\left(\frac{|\mathcal{S}|}{p}(1+\epsilon)(cl_{max}+w), C\right)\right)\right)$$

for buckets computed with string-based sampling and

$$\mathcal{O}\left(\alpha p + \beta\left(\max\left(\frac{\|\mathcal{S}\|}{p}(1+\epsilon)(c+w), C\right)\right)\right)$$

for buckets computed using character-based sampling.

Alltoall with LCP compression:

During the local sorting step $\mathcal{H}(\mathcal{S}_i)$ is computed by PE i. These LCP values can be used to compress common prefixes with Algorithm 3.2. The amount of characters that must be sent by PE i is then bounded by

$$\|\mathcal{S}_i\| - \max(\mathcal{L}(\mathcal{S}_i) - (p-1)h_{max}, 0)$$

where h_{max} is the maximum value of $\mathcal{H}(\mathcal{S}_i)$. Using Algorithm 3.2 on the whole string array would result in a reduction of the number of sent characters by $\mathcal{L}(\mathcal{S}_i)$. However, we do not compress one string array but p string arrays as we send p messages. The term $(p-1)h_{max}$ accounts for the maximum number of characters we cannot compress since the first string of each message has to remain uncompressed. However, we see that for data with a high $\mathcal{L}(\mathcal{S}_i)/\|\mathcal{S}_i\|$-ratio LCP compression is quite effective in reducing the number of bits to send.

To analyse the number of received characters that can be saved with LCP compression, we must investigate the effect of LCP compression on the number of characters in a bucket $\mathcal{B}^i$. Unfortunately, even for input sets in which $\mathcal{L}(\mathcal{S}_i)/\|\mathcal{S}_i\| \approx 1$ for all $1 \leq i \leq p$ there can be buckets determined through string-based or character-based sampling for which the effect of LCP compression is vanishingly small. Let L_i^{recv} be the number of characters that can be saved by applying LCP compression on the sorted sequences in $\mathcal{B}^i$.

Lemma 4.1. *Let Σ' be an alphabet of size $\sigma > 1$ and let n be divisible by p. For all $l > \log_{\sigma-1}(n/p)$ there is is a distributed input consisting of p string arrays $\mathcal{S}_i$ (with $|\mathcal{S}_i| = n/p$ and $\|\mathcal{S}_i\| = \|\mathcal{S}\|/p$) such that the following properties are fulfilled if we apply* `partitionSB` *with a local sample size of $s = p-1$ to this input:*

1. $\mathcal{L}(\mathcal{S}_i) \geq l \cdot (n/p - n/p^2)$
2. $L_1^{recv} \leq n/p \cdot \log_{\sigma'}(n/p)$,

Proof. Let Σ be an alphabet of size at least three and let c be the greatest character in Σ. By Σ', we denote $\Sigma \backslash \{c\}$. Let $n > p$ and let n be divisible by p^2. Furthermore, let $l > \log_{\sigma'}(n/p)$ and let $\omega := n/p^2$. We construct a set $\mathcal{S}'$ containing n/p strings of length $l+1$ over the alphabet Σ' with the property $\mathcal{L}(\mathcal{S}') < (n/p) \cdot \log_{\sigma'}(n/p)$. This can be achieved using DNGenerator (see Algorithm 7.1). We then construct another string $s_> = [c, c, \dots, c, 0]$ consisting of l repetitions of the character c. Note that $s_>$ is greater than every string in $\mathcal{S}'$. Now, we construct the string arrays $\mathcal{S}_i$. Each such array contains ω strings of $\mathcal{S}'$; the remaining $n/p - \omega$ strings are copies of s. With this construction we have $|\mathcal{S}_i| = n/p$ and $\|\mathcal{S}_i\| = \|\mathcal{S}\|/p$. Furthermore, when applying (string-based) regular sampling with $p-1$ local samples to the (locally sorted) string arrays, bucket $\mathcal{B}^1$ contains only strings of $\mathcal{S}'$. This holds since the first element of each local sample set is the greatest element of the local share of $\mathcal{S}'$; the first final splitter is the $(p-1)^{th}$ element of $\mathcal{S}^{sample}$, i.e. one of the aforementioned local samples, and all other elements are greater than the elements of $\mathcal{S}'$. Therefore, we have $L_1^{recv} < (n/p)\log_{\sigma'}(n/p)$ and $\mathcal{L}(\mathcal{S}_i) \geq (n/p - n/p^2)l$. □

For the example in the proof of Lemma 4.1 we have

$$\frac{\mathcal{L}(\mathcal{S}_i)}{\|\mathcal{S}_i\|} \geq \frac{(n/p - n/p^2) \cdot l}{n/p \cdot (l+1)} = \left(1 - \frac{1}{p}\right) \frac{l}{l+1}$$

whereas

$$\frac{L_1^{recv}}{\|\mathcal{B}^1\|} \leq \frac{\log_{\sigma'}(n/p)}{l}$$

holds. With fixed values for n and p and an increasing length l the first expression tends to $1 - 1/p$ whereas the latter ratio tends to 0. This shows that even for input sets with a high $\mathcal{L}(\mathcal{S}_i)/\|\mathcal{S}_i\|$-ratio the effect of LCP compression can be neglectable for buckets created with string-based sampling. Since all strings in the proof of Lemma 4.1 are of equal length, the same applies to character-based sampling.

Since for effective communication both the number of sent bits and the number of received bits must be small, the above example shows that there is no guarantee that LCP compression can achieve this for buckets computed with string- or character-based sampling. For these two sampling methods LCP compression remains merely heuristic. There are no reasonable guarantees for the reduction of communication volume. Nevertheless, in conjunction with `partitionCB` and `partitionSB` LCP compression shows good performance on many data sets (see Section 7).

However, this problem can be solved by considering LCP compression during the sampling process. Let $\mathcal{S}'_i$ be the LCP compressed array of $\mathcal{S}_i$ using Algorithm 3.2. Applying character-based regular sampling to the compressed string arrays $\mathcal{S}'_i$ yields buckets with at most $(N'/p)(1+\epsilon)$ characters where $N' := \sum_{1 \leq i \leq p} \|\mathcal{S}'_i\|$. We find $\epsilon \leq 2$ provided that the size of the local sample sets is $\geq p$ and the longest compressed string is sufficiently small (see Lemma 3.4). Due to time limitations the evaluation of this approach remains future work.

4.1.4 Merging

In this step PE i must merge the sorted sequences of which bucket $\mathcal{B}^i$ is composed. Bucket $\mathcal{B}^i$ contains up to p sorted sequences, i.e. we have $\mathcal{B}^i = \bigcup_{1 \leq j \leq p} \mathcal{S}^i_j$ where $\mathcal{S}^i_j$ contains the strings of $\mathcal{S}_j$ that have been sorted into the i^{th} bucket. Alongside each sorted sequence we also receive the associated LCP array $\mathcal{H}(\mathcal{S}^i_j)$. Therefore, the LCP losertree described in section 3.5 can be applied. This algorithm requires

$$M(\mathcal{B}^i) = \underbrace{\mathcal{L}(\mathcal{B}^i) - \sum_{1 \leq j \leq p} \mathcal{L}(\mathcal{S}^i_j)}_{L_i^{merge}} + \left|\mathcal{B}^j\right| \lceil \log(p) \rceil + 2^{\lceil \log(p) \rceil}$$

character comparisons. Hence, its time complexity is in $\mathcal{O}\left(M(\mathcal{B}^i)\right)$. If LCP compression has been applied before the alltoall exchange, we do not need to unpack the LCP-compressed arrays for the merging step as the compressed prefixes are skipped by the algorithm. We have $\left|\mathcal{B}^i\right|$ bound by $|\mathcal{S}|/p(1+\epsilon)$ or $\|\mathcal{S}\|/p(1+\epsilon)$, depending on the used sampling method in the previous step of the algorithm. For L_i^{merge} the only upper bound we can give is $\|\mathcal{B}^i\|$ as this aspect is not considered in the sampling method. For this reason LCP-enhanced merging remains a heuristic which never shows worse performance than merging without considering LCP values, but for which we cannot give reasonable guarantees on the reduction of character-comparisons either.

4.2 Total Running Time

In the following we give an upper bound on the expected length of the critical path of dMSS. The upper bound is on the expected length only as the stated time complexity of MSD string radix sort used in the first step only holds in the expected case. For convenience we define the following

$$L_s^{merge} := \min(\mathcal{L}(\mathcal{S}), \frac{|\mathcal{S}|}{p}(1+\epsilon) l_{max}) \qquad \text{and} \qquad L_c^{merge} := \min(\mathcal{L}(\mathcal{S}), \frac{\|\mathcal{S}\|}{p}(1+\epsilon)).$$

Table 4.1: Upper bound on the expected length of the critical path of dMSS.

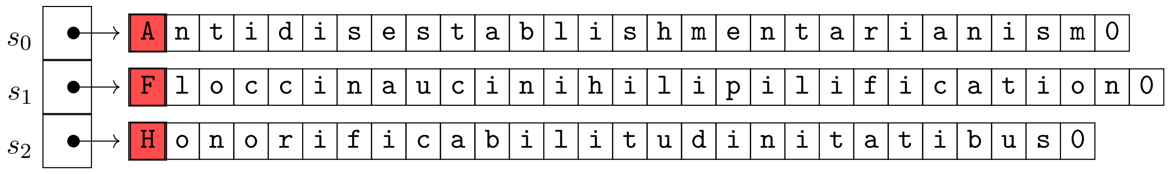
step	string-based sampling	character-based sampling
1	$\mathcal{O}\left(\delta \frac{\lvert\mathcal{S}\rvert}{p} \log(\sigma) + \max_{i=1}^{p}(\mathcal{D}(\mathcal{S}_i))\right)$	
2.1	$\mathcal{O}(\delta s)$	$\mathcal{O}\left(\Delta \frac{\lVert\mathcal{S}\rVert}{p}\right)$
2.2	$T_{sort}(\delta s, l_{max})$	$T_{sort}(\Delta s, l_{max})$
2.3	$\mathcal{O}(\alpha \log(p) + \beta c p l_{max})$	
2.4	$\mathcal{O}\left((p-1) \log\left(\frac{\delta\lvert\mathcal{S}\rvert}{p}\right) l_{max}\right)$	
3	$\mathcal{O}\left(\alpha p + \beta \max\left(\frac{\lvert\mathcal{S}\rvert}{p}(1+\epsilon)(c l_{max} + w), C\right)\right)$	$\mathcal{O}\left(\alpha p + \beta \max\left(\frac{\lVert\mathcal{S}\rVert}{p}(1+\epsilon)(c + w), C\right)\right)$
4	$\mathcal{O}\left(L_s^{merge} + \frac{\lvert\mathcal{S}\rvert}{p}(1+\epsilon) \cdot \lceil\log(p)\rceil + 2^{\lceil\log(p)\rceil}\right)$	$\mathcal{O}\left(L_c^{merge} + \frac{\lVert\mathcal{S}\rVert}{p}(1+\epsilon) \cdot \lceil\log(p)\rceil + 2^{\lceil\log(p)\rceil}\right)$

Table 4.1 shows an upper bound on the length of the critical path of the algorithm. Step one denotes the local sorting. The second step summarize the length of the critical path for the distributed order partitioning algorithm. Here, 2.1 accounts for sampling the local input array, 2.2 is the running time required to sort the sample sets. Sub-step 2.3 is the maximum time for the exchange of the final splitters and step 2.4 is an upper bound on the time complexity for the computation of the partition of the input array induced by the final splitters. Step three represents the string exchange. Step four accounts for the final merging.

Table 4.1 is rather complicated and the stated bound on the running time might not be tight at all. We feel this is a general problem of string sorting and due to the multidimensionality of string arrays. The time required to sort a string array does not only depend on the number of strings and characters in the input but also on the sum of the lengths of the distinguishing prefixes. In a distributed string sorting algorithm the situation is even more complicated as the data is partitioned over p PEs and is redistributed in the course of the algorithm. We have not been able to define concise characteristics for distributed string arrays that tightly bound the time complexity required to sort them. This is an open problem for future work.

5. Distributed Prefix-Doubling String Sort

What makes string sorting particularly "expensive" in a distributed environment is that strings may be very long and, thus, incur a high communication volume. This might be bearable if all characters of the strings are needed for sorting the input but this is often not the case. As already mentioned in Section 3.1 for sorting a string array $\mathcal{S}$ it is sufficient to look at its distinguishing prefixes. These distinguishing prefixes might – depending on the string array – be just a fraction of the total amount of characters.



s_0
s_1
s_2
A n t i d i s e s t a b l i s h m e n t a r i a n i s m 0
F l o c c i n a u c i n i h i l i p i l i f i c a t i o n 0
H o n o r i f i c a b i l i t u d i n i t a t i b u s 0

Figure 5.1: String array $\mathcal{S}$ consisting of 3 words existing in the English language. The distinguishing prefixes are coloured in red. We have $\mathcal{D}(\mathcal{S}) = 3$ and $\|\mathcal{S}\| = 87$.

Figure 5.1 illustrates the potential gap between the length of the distinguishing prefixes and the total number of characters in a string array. In the following we will introduce the distributed Prefix-Doubling String Sort algorithm (dPDSS) that tries to communicate the distinguishing prefixes of a string array only instead of the whole strings. In the context of Figure 5.1 this means that the algorithm attempts to exchange three characters (+ three end-of-string characters) instead of 87.

Before describing the algorithm we state Definition 5.1

Definition 5.1 (Distributed Distinguishing Prefixes)**.** *Let $\mathcal{S}_i$ be a string array for $1 \le i \le p$ and let $\mathcal{S}$ be the union of these string arrays. We define*

$$\hat{\mathcal{D}}(\mathcal{S}_i) = \sum_{s \in \mathcal{S}_i} |\textsc{dp}_{\mathcal{S}}(s)|.$$

Note the difference between $\textsc{dp}_{\mathcal{S}_i}(s)$ and $\textsc{dp}_{\mathcal{S}}(s)$ for a string $s \in \mathcal{S}_i$. We will refer to $\textsc{dp}_{\mathcal{S}_i}(s)$ as the *local* distinguishing prefix of s and to $\textsc{dp}_{\mathcal{S}}(s)$ as the *global* distinguishing prefix of s. Since $|\textsc{dp}_{\mathcal{S}_i}(s)| \le |\textsc{dp}_{\mathcal{S}}(s)|$, we have $\mathcal{D}(\mathcal{S}_i) \le \hat{\mathcal{D}}(\mathcal{S}_i)$.

As in Section 4 we first specify the input of the algorithm.

Input

1. PE i obtains a string array $\mathcal{S}_i$ as input. By $\mathcal{S}$, we denote the union of all local input sets, i.e. $\mathcal{S} = \bigcup\limits_{1 \le i \le p} \mathcal{S}_i$. As in dMSS, we allow duplicates.
2. The single string arrays can differ in number of strings and characters. We only assume that
$$|\mathcal{S}_i| \le \delta \frac{|\mathcal{S}|}{p} \text{ and } \|\mathcal{S}_i\| \le \Delta \frac{\|\mathcal{S}\|}{p}$$
holds for $\delta, \Delta \ge 1$, i.e. these two factors control the imbalance of the input.
3. Similarly, the sum of the length of the global distinguishing prefixes can differ among the PEs but is bounded by $\Delta' \ge 1$, i.e.
$$\hat{\mathcal{D}}(\mathcal{S}_i) \le \Delta' \frac{\mathcal{D}(\mathcal{S})}{p}$$
holds for all $1 \le i \le p$.
4. By l_{max} we denote the length of the longest string in $\mathcal{S}$, d_{max} is the length of the longest global distinguishing prefix.

As the algorithm attempts to reduce the communication volume by only sending the distinguishing prefixes of the strings, the distributed output arrays cannot contain the sorted strings as a whole. Instead, we only output the permutation which defines the sorted order of the input strings:

Output

PE i outputs an array $\mathcal{O}_i$ containing tuples (i, j). A tuple (i, j) identifies the i^{th} string on the j^{th} PE. We have the following properties:

1. Array $\mathcal{O}_i$ is locally sorted: Let $t = \mathcal{O}_i[j]$ and $t' = \mathcal{O}_i[j']$ with $0 \le j < j' < |\mathcal{O}_i|$. Then the string represented by t is smaller than or equal to the string represented by t'.
2. The arrays are globally sorted: Let j be the index of another PE with $i < j$, let $t \in \mathcal{O}_i$ and let $t' \in \mathcal{O}_j$. Then the string represented by t is smaller than or equal to the string represented by t'.

The dPDSS algorithm has the same principal structure as dMSS described in Section 4. The main difference is the (approximate) computation of the global distinguishing prefixes.

We now enumerate the five main sub-steps of dPDSS:

1. **Local Sorting:** Each PE sorts its input string array locally. We also determine the LCP values of the local string array as a by-product.
2. **Distinguishing Prefix Computation:** In this step a distributed Bloom filter is used to determine the (approximate) global distinguishing prefix of each string. From now on the algorithm is conceptually equivalent to dMSS with the strings being replaced by its approximate distinguishing prefixes.
3. **Distributed Ordered Partitioning:** The algorithms `partitionSB` or `partitionCB` are used to compute an ordered partition of the input. In `partitionCB` we do not sample the strings but the corresponding character array where the strings are replaced by its approximate distinguishing prefixes.
4. **String Exchange:** The PEs perform an alltoall exchange of the approximate distinguishing prefixes of the strings. On PE i the approximate global distinguishing prefixes of the strings assigned to bucket j are put in a message and sent to PE j.

5. **Merging:** Finally, the sequences of locally sorted distinguished prefixes are merged.

5.1 Description of the Algorithm

In the following section we describe the single sub-steps in more detail and also state their time complexity. As we want to give an upper bound on the expected time complexity of the whole algorithm in Section 5.1.2 and as there is synchronization by communication after each step, we state the maximum expected time over all PEs for each step.

5.1.1 Local Sorting

The local sorting step is exactly the same as in dMSS. Hence, we obtain the same expected running time in

$$\mathcal{O}\left(\delta\frac{|\mathcal{S}|}{p}\log(\sigma) + \Delta'\frac{\mathcal{D}(\mathcal{S})}{p}\right)$$

on any PE.

5.1.2 Distinguishing Prefix Computation

In the following Section 5.1.2.1 we will describe the distributed duplicate detection algorithm `distDuplicateDetection` capable of detecting global duplicates in the local input sets. This algorithm will then be used to (approximately) compute the distinguishing prefixes.

5.1.2.1 Distributed Duplicate Detection

The `distDuplicateDetection` algorithm is based on a distributed Single Shot Bloom Filter (dSBF) introduced by Sanders et al. [24]. A Bloom filter [9] is a probabilistic data structure answering *set membership queries.*

Definition 5.2 (Set Membership Query)**.** *Let $\mathcal{U}$ be a universe (i.e. a set) and let $\mathcal{S}$ be a subset of $\mathcal{U}$. A membership query $(e, \mathcal{S})$ for an element $e \in U$ is the question whether $e \in \mathcal{S}$ holds.*

An empty Bloom filter is conceptually a bit array B in which all bits are set to zero. The Bloom filter has k hash functions $h_i : U \to [0, |B| - 1]$ for $1 \leq i \leq k$. To answer member ship queries for the set $\mathcal{S}$, all elements of this set must be added to the Bloom filter. For $e \in S$ this is done by setting the bits in $\mathcal{B}$ at the positions $h_i(e)$ for $1 \leq i \leq k$ to 1. To answer the membership query $(e', \mathcal{S})$, the Bloom filter looks at the positions $h_i(e')$ in B for all k hash functions. If all those bits are set, the Bloom filter returns `true`, otherwise `false`. If the answer is `false`, we can be sure that the element e' is not part of $\mathcal{S}$. However, if the answer is positive, it is not clear whether the element is part of $\mathcal{S}$ or not as the investigated positions of $\mathcal{B}$ could be set to 1 due to hash collisions. The false positive rate f^+, i.e. the probability that the answer of a query $(e', \mathcal{S})$ is `true` although $e' \notin S$, depends on the size of $\mathcal{B}$, $\mathcal{S}$ and the number of used hash functions (see [26] for details).

A single shot Bloom filter (introduced in [20]) is a specialization of a Bloom filter with $k = 1$. To achieve a false positive rate of f^+ the size of the bit array $\mathcal{B}$ must be $|\mathcal{S}|/f^+$. In other words, for a false positive rate of $f^+ = \frac{1}{c}$ the corresponding bit array must be of size $|\mathcal{S}|c$.

In a distributed single shot Bloom filter the plain bit array B is distributed over all PEs, i.e. PE i handles the sub-array $[(i-1)\frac{|B|}{p}, i\frac{|B|}{p})$ for $1 \leq i \leq p$. Each PE i holds a subset $\mathcal{S}_i$ of the set $\mathcal{S}$ that shall be inserted into the Bloom filter. The insertion is done in *batch* mode, i.e. each PE computes the hash values of its local elements and partitions the hash values according to the boundaries of the subarrays of the filter's distributed bit array B. Then the hash values are sent to the according PE and the specified indices are set to 1. The upper part of Figure 5.2 shows a distributed single shot Bloom filter with $m = 30$. The elements of the distributed sets $\mathcal{S}_1, \mathcal{S}_2$ and $\mathcal{S}_3$ are inserted using a hash function h. If h is a random function, the false positive rate is

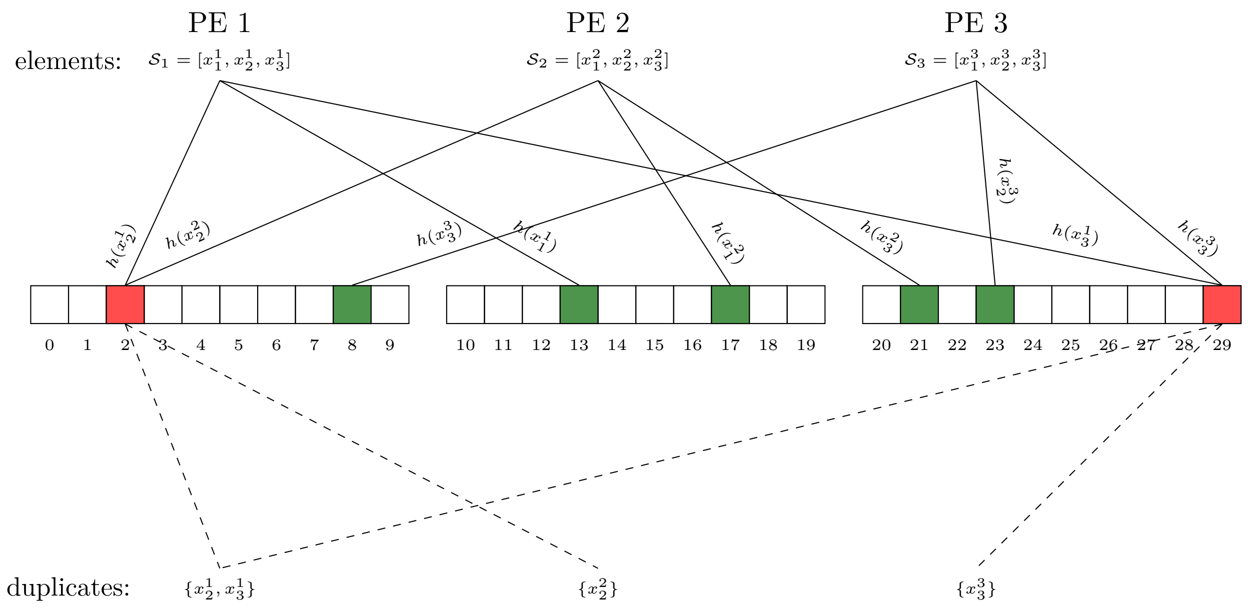


Figure 5.2: Distributed single shot Bloom filter: the bit array with 30 entries is distributed over the 3 PEs such that each PE is responsible for 10 indices. Each PE has 3 elements that are inserted into the Bloom filter via the hash function h at positions marked with 🟩 (unique hash value) or 🟥 (at least two hash values). The duplicate hash values are communicated back to their origin PEs.

$(|\mathcal{S}_1| + |\mathcal{S}_2| + |\mathcal{S}_3|)/|B| = (3 + 3 + 3)/30 = 3/10$. The coloured cells of the bit array indicate an entry specified by a received hash value, i.e. an entry that is set to 1. Conceptually, the algorithm we use for `distDuplicateDetection` is just a batch insertion of the distributed sets in which we want to identify duplicates into a distributed single shot Bloom filter. The only additional step is that we keep track of the origin PE of each hash value. If a PE receives the same hash value h' multiple times, this is communicated back to the PEs from which the hash value h' has been received. In Figure 5.2 a cell for which multiple times the same hash value has been received is coloured red. The corresponding hash values are communicated back to their sending PEs as they signal duplicates (or hash collisions of the hash function h). In contrast to the simplified illustration of Figure 5.2 the Bloom filter is not materialized in the algorithm as there is no need to post further queries to the filter. Instead, the received hash values are merged and a scan over them is performed to find duplicate values. These are communicated back. Additionally, the hash values are not sent as plain integers, but sent using an information-theoretical optimal encoding to reduce the communication volume. After sketching the idea of the algorithm, we describe it in more detail (and already adapted to strings) and discuss its running time and communication volume. The following description is based on [24]; the analysis is directly taken from the same paper.

The algorithm consists of four main steps:

1. **Local Preprocessing**
   Let m be the size of the Bloom filter, i.e. $m := |\mathcal{B}|$. PE i obtains a candidate array $\mathcal{C}_i$ consisting of tuples (j, s) where the first entry j is an index and the second entry s is a string prefix. All string prefixes are hashed using h and stored in an array $\mathcal{H}_i$, i.e. $\mathcal{H}_i = [(j, h(s)) \mid (j, s) \in \mathcal{S}_i]$. Afterwards, the array $\mathcal{H}_i$ is sorted with respect to the second entry of the tuples. Since the hash values are integers of maximum value $m - 1$, this can be achieved in expected linear time. Then tuples with duplicate hash values are removed such that all hash values in $\mathcal{H}_i$ are unique. Since the array is sorted with respect to the hash values, this can be done in a single scan over

the array. The first entry of the removed tuples is stored in $\mathcal{L}_i$. In a next step the hash values in $\mathcal{H}_i$ are partitioned into p messages $\mathcal{M}_k$ with

$$\mathcal{M}_k = \{h \mid (j,h) \in \mathcal{H}_i \wedge (k-1)(m/p) \leq h < k(m/p)\}$$

for $1 \leq k \leq p$. Since h is a random function, the hash values can be seen as drawn at random from $[0,m)$, i.e. they are uniformly distributed over this interval. Hence, the difference between two consecutive hash values are geometrically distributed. This allows to use Golomb encoding for an optimal encoding of the messages $\mathcal{M}_k$ (see [17, pg. 32]).

2. **Exchange**
   The messages are distributed in an alltoall exchange, i.e. message $\mathcal{M}_i$ is sent to PE i.

3. **Duplicate Detection**
   Each PE receives p messages. These are decoded and each hash value is tagged with the index of the PE from which it has been sent. Then the p sequences of hash values are again sorted (in expected linear time). This allows to identify duplicate hash values by scanning over the merged array. Next we specify how the duplicates are sent back. Assume that the j^{th} received messages contained n' hash values. Then a bit array of size n' is sent back to PE j with the indices corresponding to duplicates set to 1.

4. **Local Postprocessing**
   PE i receives p bit arrays. By scanning over the messages sent in the exchange step, the duplicate hash values corresponding to the entries set to 1 in the received bit arrays can be obtained and then be stored in an array $\mathcal{H}_i^{dup}$. These are sorted as the hash values were sorted before sending. Therefore, a parallel scan over $\mathcal{H}_i^{dup}$ and $\mathcal{H}_i$ suffices to compute the indices of the prefixes whose hash values are not globally unique. These indices are appended to the indices of the strings $\mathcal{L}_i$, whose prefixes have not even been locally unique . Hence, this array contains the indices of all strings whose prefixes are not unique. It is returned as result of the algorithm.

We now give an upper bound on the expected running time of the algorithm. Assume that the maximum size of the candidates array is at most n_{max} for all PEs and let l be the length of the prefixes. Furthermore, let n be the sum over the number of candidates on all PEs. Assuming that hashing a string s of length l is in $\mathcal{O}(l)$, we obtain an expected time complexity for the local preprocessing in $\mathcal{O}(n_{max}l + n_{max})$. Sanders et al. [24] prove that the expected communication volume of the exchange step is $n_{max}(\log(mp/n) + \mathcal{O}(1)) = n_{max}(\log(p/f^+) + \mathcal{O}(1))$ bits if $n = \omega(p^2 \log(m))$. Hence, this step can be executed in $\mathcal{O}\left(\alpha p + \beta(\log(p/f^+) + 1)\right)$. The local work of the duplicate detection can be done in expected $\mathcal{O}(n_{max})$ time. The back communication of the found duplicates is dominated by the time complexity of the first exchange step. The local postprocessing is in $\mathcal{O}(n_{max})$ as we only need to scan over the indices and hash values a constant number of times. Summing all steps yields an expected total time complexity in

$$\mathcal{O}\left(n_{max}l + \alpha p + n_{max}\beta\left(\log\left(p/f^+\right) + 1\right)\right).$$

For simplicity we will in the following assume that `distDuplicateDetection` receives already hashed prefixes in the candidate array and does not hash the prefixes itself. This yields an expected running time of `distDuplicateDetection` in

$$\mathcal{O}\left(\alpha p + n_{max}\beta\left(\log\left(p/f^+\right) + 1\right)\right).$$

5.1.2.2 Prefix-Doubling Algorithm for Computing Distinguishing Prefixes

As already pointed out only the distinguishing prefixes of strings are required to sort them. We use a prefix-doubling approach based on the duplicate detection algorithm `distDuplicateDetection` described above to (approximately) determine these. Algorithm 5.1 describes our approach to compute the distinguishing prefixes. During the algorithm we maintain a candidates set $\mathcal{C}_i$ which contains the indices of all strings in $\mathcal{S}_i$ whose (approximate) distinguishing prefix length has not been determined yet. The algorithm works in rounds (i.e. the iterations of the `while`-loop in lines 4 – 15). In each round `distDuplicateDetection` returns the indices of the strings s whose prefix $s[0, l)$ is not unique in $\mathcal{S}$. Only these duplicate strings participate in the next round. Since `distDuplicateDetetction` is based on a distributed Bloom filter, there can be false positives, i.e. strings whose prefix $s[0, l)$ is actually unique but that are returned anyway. The false positive rate f^+ is a tuning parameter.

Algorithm 5.1: Distinguishing Prefix Computation

```
Input: On PE $i$: string array $\mathcal{S}_i$.
1  $\mathcal{C}_i \leftarrow [0, \dots, |\mathcal{S}_i| - 1]$;                       // indices of candidates
2  $\mathcal{P}_i \leftarrow [0 \mid j \in \mathcal{C}_i]$;      // length of dist.  prefix, initially set to 0
3  $l \leftarrow 2$;                                   // currently tested prefix length
4  while $\mathcal{C}_i \neq \emptyset$ do
5      $j' \leftarrow 0$;
6      for $j \leftarrow 0$ to $|\mathcal{C}_i| - 1$ do
7          $s \leftarrow \mathcal{S}_i[\mathcal{C}_i[j]]$;
8          if $|s| \leq l$ then
9              $\mathcal{P}_i[\mathcal{C}_i[j]] \leftarrow |s|$;    // candidate length is at most tested prefix length
10         else
11             $\mathcal{P}_i[\mathcal{C}_i[j]] \leftarrow l$;
12             $\mathcal{C}'_i[j'] \leftarrow (\mathcal{C}_i[j]$,hash($s[0, l)$) ;
13             $j' \leftarrow j' + 1$;
14     $\mathcal{C}_i \leftarrow$distDuplicateDetection($\mathcal{C}'_i$) ;     // returns only indices of duplicates
15     $l \leftarrow 2l$;
Output: $\mathcal{P}_i$ contains at least the length of the distinguishing prefix of each string in $\mathcal{S}_i$.
```

The maximum number of rounds is bounded by $\lceil \log(l_{max}) \rceil$ (see line 8). Since `distDuplicate-Detection` may return false positives but never indicates erroneously that a non-unique prefix is unique, the approximate length of the distinguishing prefix of a string s computed by the algorithm is at least the length of its (global) distinguishing prefix.

To further quantify the quality and running time of Algorithm 5.1 we need the following Definition 5.3.

Definition 5.3 (Approximate Sum of Distinguishing Prefixes)**.** *We define*

$$\tilde{\mathcal{D}}(\mathcal{S}_i) := \sum_{j=0}^{|\mathcal{S}_i|-1} \mathcal{P}_i[j]$$

with $\mathcal{P}_i$ being the result of Algorithm 5.1 with local input $\mathcal{S}_i$ for $1 \leq i \leq p$. For subsets $\mathcal{S}'_i \subset \mathcal{S}_i$ we define

$$\tilde{\mathcal{D}}(\mathcal{S}'_i) := \sum_{j \in I'} \mathcal{P}_i[j]$$

where I' are the indices of the strings of $\mathcal{S}'_i$ in $\mathcal{S}_i$.

In other words, $\tilde{\mathcal{D}}(\mathcal{S}_i)$ is the approximate value of $\hat{\mathcal{D}}(\mathcal{S}_i)$ computed by Algorithm 5.1. Lemma 5.4 bounds its expected value.

Lemma 5.4. *The expected value of $\tilde{\mathcal{D}}(\mathcal{S}_i)$ for a false positive rate of $f^+ \in (0, \frac{1}{2})$ is smaller than*

$$\left(1 + \frac{1}{1-2f^+}\right)\hat{\mathcal{D}}(\mathcal{S}_i).$$

Proof. Let $\mathcal{S}_{i|j}$ be the strings of $\mathcal{S}_i$ whose global distinguishing prefix length is in the interval $(2^{j-1}, 2^j]$ for $j \geq 2$ and in $[2^{j-1}, 2^j]$ for $j = 1$, respectively. Let m be the greatest number for which $\left|\mathcal{S}_{i|m}\right| > 0$, i.e. $2^{m-1} < d_{max} \leq 2^m$. Since the sets $\mathcal{S}_{i|j}$ partition $\mathcal{S}_i$,

$$\tilde{\mathcal{D}}(\mathcal{S}_i) = \sum_{j=1}^{m} \tilde{\mathcal{D}}(\mathcal{S}_{i|j})$$

holds. In the case that there are no false positive detections, i.e. $f^+ = 0$, the approximate distinguishing prefix length of a string in $\mathcal{S}_{i|j}$ is at most 2^j as the prefix lengths $2, 4, 8\ldots$ are tested in the algorithm. Hence, we obtain

$$\tilde{\mathcal{D}}(\mathcal{S}_i) \leq 2\left|\mathcal{S}_{i|1}\right| + 4\left|\mathcal{S}_{i|2}\right| + \cdots + 2^m\left|\mathcal{S}_{i|m}\right| \leq 2\hat{\mathcal{D}}(\mathcal{S}_i).$$

In this case the algorithm is deterministic and $\tilde{\mathcal{D}}(\mathcal{S}_i)$ is not a random variable. The situation is different for $f^+ > 0$. We then have false positives and the algorithm itself is no longer deterministic. Let s be a string with a global distinguishing prefix in $(2^{j-1}, 2^j]$. The probability that it is erroneously detected as a duplicate in round j is f^+, the probability that the corresponding entry in $\mathcal{P}_i$ is set to 2^j, i.e. $s's$ approximate distinguishing prefix length is set to 2^j, is $(1 - f^+)$. The probability that string s remains further j' rounds in the process is $f^{+j'}$ (provided that the string is long enough, otherwise this value is not exact but an upper bound). Hence, the probability that the corresponding entry in $\mathcal{P}_i$ is set to $2^{j+j'}$ is $f^{+j'}(1 - f^+)$. Therefore, we obtain

$$\begin{aligned}
\mathbb{E}(\tilde{\mathcal{D}}(\mathcal{S}_{i|j})) &< \left|\mathcal{S}_{i|j}\right| \left(2^j(1-f^+) + f^+2^{j+1}(1-f^+) + (f^+)^2 2^{j+2}(1-f^+) + \ldots\right) \\
&= \left|\mathcal{S}_{i|j}\right|(1-f^+)2^j(1 + 2f^+ + (2f^+)^2 + \ldots) \\
&= \left|\mathcal{S}_{i|j}\right|(1-f^+)2^j\left(\sum_{k=0}^{\infty}(2f^+)^k\right) = \left|\mathcal{S}_{i|j}\right|(1-f^+)2^j\frac{1}{1-2f^+}
\end{aligned}$$

for $1 \leq j \leq m$.

By the linearity of expected values this yields

$$\mathbb{E}(\tilde{\mathcal{D}}(\mathcal{S}_i)) = \sum_{j=1}^{m}\mathbb{E}(\tilde{\mathcal{D}}(\mathcal{S}_{i|j})) < \frac{1-f^+}{1-2f^+}\sum_{j=1}^{m}2^j\left|\mathcal{S}_{i|j}\right| \leq \frac{1-f^+}{1-2f^+}2\hat{\mathcal{D}}(\mathcal{S}_i) = \left(1 + \frac{1}{1-2f^+}\right)\hat{\mathcal{D}}(\mathcal{S}_i).$$

□

The proof of Lemma 5.4 also shows that the expected value of the approximate distinguishing prefix length of any string $s \in \mathcal{S}$ is smaller than $(1 + 1/(1-2f^+))\textsc{dp}_{\mathcal{S}}(s)$. Furthermore, Lemma 5.4 implies that the expected amount of characters sent in the subsequent string exchange step is also in $\mathcal{O}\left(\hat{\mathcal{D}}(\mathcal{S}_i)\right)$ and not $\|\mathcal{S}_i\|$ as in dMSS. However, the approximate computation of $\hat{\mathcal{D}}(\mathcal{S}_i)$ incurs additional costs in running time. These are addressed in Lemma 5.5.

Lemma 5.5. *The expected time complexity of Algorithm 5.1 is in*

$$\mathcal{O}\left(\lceil\log(l_{max})\rceil\left(\left(1 + \frac{1}{1-2f^+}\right)\Delta'\frac{\hat{\mathcal{D}}(\mathcal{S})}{p} + \delta\frac{|\mathcal{S}|}{p}\beta\left(\log\left(\frac{p}{f^+}\right) + 1\right) + \alpha p\right)\right).$$

for $f^+ < 1/2$ and assuming $|\mathcal{S}| = \omega(p^2\log(|\mathcal{S}|/f^+))$.

Proof. Let n^j_{max} be the maximum number of strings participating in iteration j of the algorithm, i.e. $n^j_{max} = \max_{i=1}^{p}(|\mathcal{C}_i|)$. Furthermore, we define $\tilde{D}_{max} := \max_{i=1}^{p}(\tilde{\mathcal{D}}(\mathcal{S}_i))$. Note that in iteration j (starting with iteration $j = 1$) it holds that $n^j_{max} \leq \tilde{D}_{max}/2^{j-1}$ as the approximate distinguishing prefix length of a string participating in iteration j of the algorithm is at least 2^{j-1} and $\tilde{D}_{max}$ is the maximum of the sums of all these approximate distinguishing prefix lengths.

The time complexity of lines 4 – 13 is dominated by hashing the prefix $s[0, 2^j)$ in line 12. Therefore, we obtain $\mathcal{O}\left(2^j \tilde{D}_{max}/2^{j-1}\right)$ as upper bound over all PEs on the running time of these lines in iteration j.

What remains to be considered is the time complexity of `distDuplicateDetection` within iteration j. However, the bound on the expected time complexity of `distDuplicateDetection` stated in the previous section only holds if the sum over the number of participating strings over all PEs n is in $\omega(p^2 \log(n/f^+)$. Unfortunately, we cannot guarantee this to be true for all iterations a priori as the number of participating strings decreases with growing number of iterations. Let $T(n_{max}, n, m)$ be the expected time complexity of `distDuplicateDetection` for a maximum of n_{max} strings on any PE, n strings in total over all PEs and a Bloom filter size of m. In the following we will outline why it is valid to assume that $T(n'_{max}, n', m) \leq T(n_{max}, n, m)$ holds for $n'_{max} \leq n_{max}$ and $n' \leq n$. Since a random hash function is used to compute the hash values of the input strings, the expected number of hash values sent to one PE decreases with an decreasing input size. It follows that the expected time complexity of all work done locally on the PEs equally decreases. The only problem that remains to be considered is that the message lengths does not depend on the number of hash values per message alone but also on the used encoding scheme. Let $\textsc{bits}(i, m)$ be the number of bits required to encode i distinct random hash values within the range $[0, m-1]$. Our `distDuplicateDetection` relies on $\textsc{bits}(i, m) := i(\log(m/i) + \mathcal{O}(1))$ [24]. Let c be the hidden constant in the above function. The derivative of $\textsc{bits}(i, m)$ with respect to i is

$$\frac{\partial}{\partial i}\textsc{bits}(i, m) = \frac{c\ln(2) + \ln(m/i) - 1}{c\ln(2)}.$$

The zero of the derivative is $i = m2^c/e$. Since the derivative is positive for $i = 1$ (provided that $m \geq 3$), this implies that $\textsc{bits}(\cdot, m)$ is (strictly) monotone increasing in the interval $[1, 2^c m/e]$.

Assuming $c = 1.5$ (which is the case for Golomb encoding [24]) and $m > 2n$ (since we want $f^+ < 1/2$), we obtain $\textsc{bits}(n'_{max}, m) \leq \textsc{bits}(n_{max}, m)$ for $n'_{max} \leq n_{max} \leq n$. This implies that in our case the Golomb encoding of i' hash values needs at most the same number of bits as the encoding of i hash values with $i' \leq i$. Therefore, we can conclude $T(n'_{max}, n', m) \leq T(n_{max}, n, m)$. Hence, the expected time complexity of `distDuplicateDetection` in each iteration is bounded by the expected time required for `distDuplicateDetection` with a maximum number of locally participating strings equal to $\delta\frac{|\mathcal{S}|}{p}$, which is

$$\mathcal{O}\left(\alpha p + \delta\frac{|\mathcal{S}|}{p}\beta\left(\log\left(p/f^+\right) + 1\right)\right)$$

as stated in Section 5.1.2.1.

Therefore, the expected running time of iteration j is in

$$\mathcal{O}\left(\alpha p + \delta\frac{|\mathcal{S}|}{p}\beta\left(\log\left(p/f^+\right) + 1\right) + \frac{2^j\mathbb{E}(\tilde{D}_{max})}{2^{j-1}}\right).$$

Since $\mathbb{E}(\tilde{\mathcal{D}}(\mathcal{S}_i)) \leq \left(1 + \frac{1}{1-2f^+}\right)\hat{\mathcal{D}}(\mathcal{S}_i) \leq \left(1 + \frac{1}{1-2f^+}\right)\Delta'\frac{\mathcal{D}(\mathcal{S})}{p}$ and since there are at most $\lceil\log(l_{max}\rceil)$ iterations within the algorithm the claimed bound on the expected time complexity holds. □

The analysis of Algorithm 5.1 shows that the computation of the (approximate) distinguishing prefix lengths is not for free. But this computation can be still worthwhile, especially, if $\|\mathcal{S}\| \gg \hat{\mathcal{D}}(\mathcal{S})$. In this case a significant amount of communication in the alltoall string exchange can be saved.

Algorithm 5.2: Reduction to Distinguishing Prefixes

```
Input: An array of strings $\mathcal{S}$ and an array $\mathcal{P}$ containing positive integers of the same size.
1 for $i \leftarrow 0$ to $|\mathcal{S}| - 1$ do
2     $s \leftarrow (\mathcal{S}[i],\ l \leftarrow \min(\mathcal{P}[i], |s| - 1))$;
3     $s' \leftarrow s[0, l) + [0]$;
4     $\mathcal{S}'[i] \leftarrow s'$;
Output: String array $\mathcal{S}'$ containing the prefixes of $\mathcal{S}$ specified by $\mathcal{P}$.
```

5.1.3 Distributed Ordered Partitioning

For the computation of the buckets the same methods as for dMSS (see Section 4.1.2) can be applied. The only difference is that we now know the (approximate) values of the strings' global prefix lengths. Applying `partitionSB` does not change the upper bound on the number of strings per bucket. However, the maximum number of characters per bucket can now be bounded using $\tilde{d}_{max}$ – the maximum approximate distinguishing prefix length – computed in the previous step instead of l_{max} as in dMSS. Hence, we obtain the following upper bounds

$$\left|\mathcal{B}^i\right| \leq \delta\frac{|\mathcal{S}|}{p}(1+\epsilon) \text{ and } \left\|\mathcal{B}^i\right\| \leq \delta\frac{|\mathcal{S}|}{p}(1+\epsilon)\tilde{d}_{max}.$$

for $1 \leq i \leq p$. The expected value of $\tilde{d}_{max}$ is in $\mathcal{O}(d_{max})$ for $f^+ < 1/2$. The value of ϵ depends on the number of locally sampled elements in `partitionSB` and is smaller than one if p or more elements are chosen locally.

Also, character-based sampling can be used in this algorithm. The only change to be made is that we do not sample $\mathcal{S}_i$ directly but a string array $\mathcal{S}'_i$ that is computed by Algorithm 5.2 with $\mathcal{S}_i$ and the approximate distinguishing prefix lengths as input. Applying `partitionCB` to $\mathcal{S}'_i$ results in buckets with

$$\left|\mathcal{B}^i\right| \leq \left\|\mathcal{B}^i\right\| \leq \left(1+\frac{1}{1-2f^+}\right)\frac{\mathcal{D}(\mathcal{S})}{p}(1+\epsilon)$$

for $1 \leq j \leq p$ in the expected case, provided that the longest string in $\mathcal{S}'$ is sufficiently short as required in Theorem 3.5. The value of ϵ is smaller than two if the size of the local sample sets is at least p.

Additionally, LCP compression (Algorithm 3.2) could be used on the distinguishing prefixes in $\mathcal{S}'_i$ resulting in an array of LCP compressed strings $\mathcal{S}''_i$ on PE i on which `partitionCB` could be applied. We did not further investigate this variant and only want to mention it here for the sake of completeness.

In Section 3.3 we only stated the worst-case running time of the two variants of `partition`. However, in dPDSS the input to `partition` is a random variable since the computation of the distinguishing prefixes is not deterministic. We want to briefly discuss how this influences the expected running time of the partitioning algorithm. The sampling step for string-based sampling is in

$$\mathcal{O}\left(\delta s\left(1+\frac{1}{1-2f^+}\right)d_{max}\right)$$

for all PE as at most δs times the maximal approximate distinguishing prefix must be copied. Note that we cannot simply copy the string-pointer as in dMSS to construct the string array $\mathcal{S}_i^{sample}$ on PE i since we only want to use the distinguishing prefix not the strings as a whole.

Let us now consider character-based sampling. As Algorithm 5.1 outputs the approximate distinguishing prefix lengths, we do not construct the array $\mathcal{S}'_i$ explicitly to execute `partitionCB`. It is sufficient to scan over the output array $\mathcal{P}_i$ containing the approximate distinguishing prefix lengths to obtain the desired sample set. Therefore, the expected time complexity is in

$$\mathcal{O}\left(\delta\frac{|\mathcal{S}|}{p} + \Delta' s\left(1+\frac{1}{1-2f^+}\right)d_{max}\right).$$

To give a bound on the sorting of the sample sets, we cannot refer to hypercube quicksort as we do not know any bounds on its expected running time. Instead we account the expected time complexity to a black box distributed sorter with an expected running time $T_{sort}(n, l)$ where n is the maximum number of strings in any of the p input sets and l is the expected value of the length of the longest string in the input. Furthermore, we assume the black box sorter to balance its output such that the number of strings in each output set is in $\mathcal{O}(n)$. For string-based sampling we obtain $T_{sort}(\delta s, \left(1+\frac{1}{1-2f^+}\right)d_{max})$ and $T_{sort}(\Delta' s, \left(1+\frac{1}{1-2f^+}\right)d_{max})$ for character-based sampling. The redistribution of the final splitters can be done in expected $\mathcal{O}\left(\alpha\log(p) + \beta cp\left(1+\frac{1}{1-2f^+}\right)d_{max}\right)$ time since the number of final splitters per PE is constant. The expected time required to determine the intervals of the partition of the input string arrays is in $\mathcal{O}\left((p-1)\log\left(\delta\frac{|\mathcal{S}|}{p}\right)\left(1+\frac{1}{1-2f^+}\right)d_{max}\right)$ as at most the distinguishing prefix must be compared during the $p-1$ binary searches which are executed to determine the intervals of the partition induced by the final splitters.

5.1.4 String Exchange

This time, the (expected) amount of bits sent by any PE is bounded by

$$C := \delta\frac{|\mathcal{S}|}{p}w + \Delta'\left(1+\frac{1}{1-2f^+}\right)\frac{\mathcal{D}(\mathcal{S})}{p}c$$

for all $1 \le i \le p$ as we only send the approximate distinguishing prefixes (and the corresponding LCP arrays). In contrast to dMSS the term $\|\mathcal{S}\|$ can be omitted completely. For string-based sampling and the 2-phase alltoall algorithm, we obtain an expected communication time in

$$\mathcal{O}\left(\alpha p + \beta\max\left(\frac{|\mathcal{S}|}{p}(1+\epsilon)\left(c\left(1+\frac{1}{1-2f^+}\right)d_{max} + w\right), C\right)\right).$$

For character-based sampling the expected communication time is in

$$\mathcal{O}\left(\alpha p + \beta\max\left(\left(1+\frac{1}{1-2f^+}\right)\frac{\mathcal{D}(\mathcal{S})}{p}(1+\epsilon)(c+w), C\right)\right).$$

Additional LCP compression is always beneficial in terms of communication time/volume reduction, but we are not able to state better upper bounds.

5.1.5 Merging

As in dMSS the sorted sequences of which bucket $\mathcal{B}^i$ is composed must be merged. Bucket $\mathcal{B}^i$ contains up to p sorted sequences, i.e. we have $\mathcal{B}^i = \bigcup_{1\le j\le p} \mathcal{S}_j^i$, where $\mathcal{S}_j^i$ contains the strings of $\mathcal{S}_j$ that were sorted into the i^{th} bucket. Alongside each sorted sequence we also receive the associated LCP array $\mathcal{H}(\mathcal{S}_j^i)$. Therefore, the LCP losertree algorithm can be applied and requires

$$M(\mathcal{B}^i) = \underbrace{\mathcal{L}(\mathcal{B}^i) - \sum_{1\le j\le p}\mathcal{L}(\mathcal{S}_j^i)}_{L_i^{merge}} + \left|\mathcal{B}^j\right| \cdot \lceil\log(p)\rceil + 2^{\lceil\log(p)\rceil}$$

character comparisons. Hence, its time complexity is in $\mathcal{O}\left(M(\mathcal{B}^i)\right)$. The only upper bound we can give for L_i^{merge} is $\|\mathcal{B}^i\|$ as this aspect is not considered during the computation of the ordered partition. Therefore, we obtain

$$\mathcal{O}\left(\frac{|\mathcal{S}|}{p}(1+\epsilon)\left(\left(1+\frac{1}{1-2f^+}\right)d_{max}+\log(p)\rceil\right)+2^{\lceil\log(p)\rceil}\right)$$

as upper bound on the expected running time for string-based sampling. Character-based sampling yields an upper bound of

$$\mathcal{O}\left(\left(1+\frac{1}{1-2f^+}\right)\frac{\mathcal{D}(\mathcal{S})}{p}(1+\epsilon)(1+\log(p)\rceil)+2^{\lceil\log(p)\rceil}\right)$$

on the expected running time of this step. During the merging process, we need to keep track of the index k of a string s within the message $\mathcal{S}_j^i$ in which it was sent. Additionally, the offset of the position at which $\mathcal{S}_j^i$ begins within $\mathcal{S}_j$ on PE j needs to be known. This can be achieved by a regular alltoall-exchange of these offsets. The time complexity of this exchange is dominated by the complexity of the previous string exchange as we also send LCP values in this step. With this information the required permutation defining the sorted order of the input can be computed.

5.2 Total Running Time

Table 5.1 summarizes the upper bounds on the expected running time of the single sub-steps. By the linearity of expected values the sum of all these bounds is an upper bound on the length of the critical path of the whole dPDSS algorithm. To make this table more concise we define $b := \left(1+\frac{1}{1-2f^+}\right)d_{max}$ and $B := \left(1+\frac{1}{1-2f^+}\right)\frac{\mathcal{D}(\mathcal{S})}{p}$.

Table 5.1: Upper bound on the expected length of the critical path of dPDSS.

step	string-based sampling	character-based sampling
1	$\mathcal{O}\left(\delta\frac{\lvert\mathcal{S}\rvert}{p}\log(\sigma)+\Delta'\frac{\mathcal{D}(\mathcal{S})}{p}\right)$	
2	$\mathcal{O}\left(\lceil\log(l_{max})\rceil\left(\Delta' B+\delta\frac{\lvert\mathcal{S}\rvert}{p}\beta\left(\log\left(\frac{p}{f^+}\right)+1\right)+\alpha p\right)\right)$	
3.1	$\mathcal{O}\left(\delta s b\right)$	$\mathcal{O}\left(\delta\frac{\mathcal{D}(\mathcal{S})}{p}+\Delta' s b\right)$
3.2	$T_{sort}(\delta s, b)$	$T_{sort}(\Delta s, b)$
3.3	$\mathcal{O}\left(\alpha\log(p)+\beta c p b\right)$	
3.4	$\mathcal{O}\left((p-1)\log\left(\delta\frac{\lvert\mathcal{S}\rvert}{p}\right)b\right)$	
4	$\mathcal{O}\left(\alpha p+\beta\max\left(\frac{\lvert\mathcal{S}\rvert}{p}(1+\epsilon)(cb+w),C\right)\right)$	$\mathcal{O}\left(\alpha p+\beta\max\left(B(1+\epsilon)(c+w),C\right)\right)$
5	$\mathcal{O}\left(\frac{\lvert\mathcal{S}\rvert}{p}(1+\epsilon)(b+\log(p)\rceil)+2^{\lceil\log(p)\rceil}\right)$	$\mathcal{O}\left(B(1+\epsilon)(1+\log(p)\rceil)+2^{\lceil\log(p)\rceil}\right)$

Step one denotes the local sorting. The second step accounts for the time complexity of the computation of the approximate length of the distinguishing prefixes. Step three summarizes the computation of the buckets. Here, 3.1 accounts for sampling the local input array, 3.2 is an upper bound on the time required for the determination of the final splitter in the case that a black box sorter with the above-specified properties is used and 3.3 is the maximum time needed to compute the intervals assigned to the buckets in the local string arrays. Step four represents the string exchange and step five the final merging.

6. Implementation Details

To be capable of evaluating our algorithms on real machines we have implemented them in C++ with MPI as communication standard. In the following we specify some parts of the implementation that might not have become clear in the high-level algorithmic descriptions so far. Before going into the details we want to point out that all of our parallelization is done via MPI, i.e. no shared-memory parallelism has been applied. If not stated otherwise we use the MPI built-in implementations of collective operations like `MPI_Alltoall`, `MPI_Alltoallv` or `MPI_Allgather`.

6.1 Memory Layout of Strings

In atomic key sorting the memory layout of the elements to sort is rather straight forward. The elements are stored in a contiguous array. This approach is prohibitive for string sorting. In most sorting algorithms elements must be swapped/moved after each element comparison. This is a constant time operation in atomic key sorting since the element are of fixed length and commonly "small". In string sorting this is not the case. Therefore, we use a pointer-based approach in our implementation. A string array $\mathcal{S}$ is represented by two arrays:

- A *raw-string* array of size $\|\mathcal{S}\|$ that contains the strings as contiguous sequences of characters.
- A *pointer* array of size $|\mathcal{S}|$ in which each entry is a pointer to the first character of the corresponding string in the raw-strings array.

With this memory layout strings can be swapped by swapping the corresponding pointer. This can be achieved in constant time like in the atomic key case. Figure 6.1 shows an example of the memory layout of a string array consisting of 4 strings and 25 characters.

6.2 Sequential String Sorting

We use the optimized inplace variant CI3 of MSD string radix sort (see [5]) as sequential string sorter within all of our algorithms. The implementation has been taken from the `tlx`-library[1] written by Bingmann. The original implementation did not support the computation of the LCP array during the sorting process. The LCP array can be computed by comparing all pairs of strings (s_{i-1}, s_i) for $0 < i < |\mathcal{S}|$ of the *sorted* string array $\mathcal{S}$ again. However, this approach is not efficient especially if the string array has a high (D/N)-ratio. Bingmann et al. [6] therefore propose to perform this computation during the sorting process itself. In our implementation we followed their approach, which will be outlined in the following. Meanwhile, the LCP-extended sorters are also part of the `tlx`-library.

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

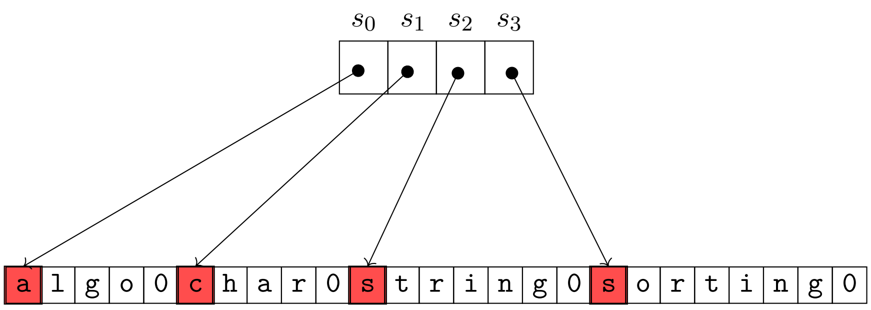


Figure 6.1: Memory layout of a string array consisting of 4 strings.

MSD String Radix Sort

Let $\mathcal{S}$ be the string array that is given to MSD string radix sort as input in the first iteration. The input to MSD string radix sort in the l^{th} iteration (first iteration is $l = 0$) are the $|\Sigma|$ buckets computed in the previous iteration (the bucket corresponding to the end-of-string character does not further participate in the process). This implies that the input has a common prefix of length l. Furthermore, we assume that the offset o at which the smallest string of the currently investigated input will be located in the *sorted* string array $\mathcal{S}$ is known. This can easily be achieved by computing the prefix sum of the sizes of the buckets of the previous iteration $l - 1$ and the offset of the previous iteration. The offset in the first iteration is 0. In iteration l the input is partitioned into $|\Sigma| + 1$ buckets based on the $(l + 1)^{th}$ character of each string. Let $i > 0$ be the position within the sorted array $\mathcal{S}$ at which the smallest element of a non-empty bucket computed in iteration l will be located after the execution of the whole algorithm. Then we know that the string that will be (finally) located at position i and the string that will be finally located at position $i - 1$ have the longest common prefix length l. Otherwise the partitioning into buckets based on the $(l + 1)^{th}$ character would have put them into the same bucket. Therefore, we can already set the LCP array at position i to l even without knowledge about the strings that will finally be sorted at the positions $i - 1$ and i. Hence, it will be sufficient to execute Algorithm 6.1 after the partitioning in each iteration of MSD string radix sort to compute the correct LCP array entries.

Algorithm 6.1: LCP Computation within MSD String Radix Sort

Input: Offset of input o, size of buckets b, depth of currently investigated character l, LCP array $\mathcal{H}(\mathcal{S})$.

```
1  $B \leftarrow 0$
2  for $i \leftarrow 0$ to $|b| - 1$ do
3      $B \leftarrow B + b[i]$
4  for $i \leftarrow 1$ to $b[0] - 1$ do          // strings within end-of-string bucket have LCP $l$
5      $\mathcal{H}(\mathcal{S})[o + i] \leftarrow l$
6  $i \leftarrow 0,\ j \leftarrow 0$
7  while $i < |b|$ do
8      while $i < |b| \wedge b[i] = 0$ do                        // skip empty buckets
9          $i \leftarrow i + 1$
10     if $i = |b| \vee j + b[i] = B$ then
11         break
12     $j \leftarrow j + b[i]$
13     $\mathcal{H}(\mathcal{S})[o + j] \leftarrow l$
```

Multikey Quicksort

Since multikey quicksort is the base sorter of string radix sort, we also have to describe how the LCP array can be obtained within this algorithm. The approach is very similar to the one used for string radix sort. The only difference is that we do not partition into $|\Sigma| + 1$ but only into 3 buckets. Refer to Algorithm 6.2 for the details.

Algorithm 6.2: LCP Computation within Multikey Quicksort

Input: Offset of input o, sizes of the three buckets b_1, b_2 and b_3, pivot string p, depth of currently investigated character l, LCP array $\mathcal{H}(\mathcal{S})$.

```
1 if $p[l-1] = 0$ then                // pivot character is end-of-string character.
2     for $i \leftarrow 1$ to $b_1 - 1$ do
3         $\mathcal{H}(\mathcal{S})[o+i] \leftarrow l$
4 if $b_1 > 0$ then                   // first bucket not empty
5     $\mathcal{H}(\mathcal{S})[o+b_1] \leftarrow l$
6 if $b_3 > 0$ then     // third bucket not empty, note $b_1 + b_2 > 0$ does always hold
7     $\mathcal{H}(\mathcal{S})[o+b_1+b_2] \leftarrow l$
```

LCP-Insertion Sort

LCP-Insertion sort is the base sorter of multikey quicksort. However, this algorithm is already designed to compute the LCP array while sorting the input. We therefore simply refer to the pseudo-code of the algorithm in [6], which we followed in our implementation.

6.3 Sending Strings/LCP Values with MPI

There are three main aspects that need to be considered for exchanging strings together with LCP values via MPI.

Encoding of Strings:

The MPI standard defines data types for integers and characters (`MPI_Int`, `MPI_Char`, ...) but does not offer a data type for strings [12]. Therefore, we exchange arrays of strings as raw-string arrays, i.e. as arrays of characters using the `MPI_Byte` data type. On the receiving PE the hybrid structure of raw-string/pointer array is reconstructed by iterating over the received raw-string array and setting the pointers to characters following an end-of-string character.

Maximal Message Length:

The type of the message count in the C-binding of MPI is `int` - a signed type for which it is likely to be represented by 4 bytes. Hence, the maximum number of elements that can be sent in a message is limited to $2^{31} - 1$ on such systems. Since we encode strings as raw-character arrays this implies that the maximum amount of string data that can be sent in one message is restricted to about 2GB. For larger messages we use MPI's *custom types*. MPI gives the possibility of constructing own data types based on the built-in data types like `MPI_Byte`. With `MPI_Type_contiguous(int count, MPI_Datatype old, MPI_Datatype new)` a data type `new` which represents `count` continuous copies of the original data type `old` can be created. Since this method can be also applied on own data types, we can create *big types* representing more than $2^{31} - 1$ contiguous bytes. One drawback of this solution is that each message bigger than 2^{31} now has its own type. Hence, the MPI collective communication operations like `MPI_Alltoallv` can no longer be used. We adopted the approach

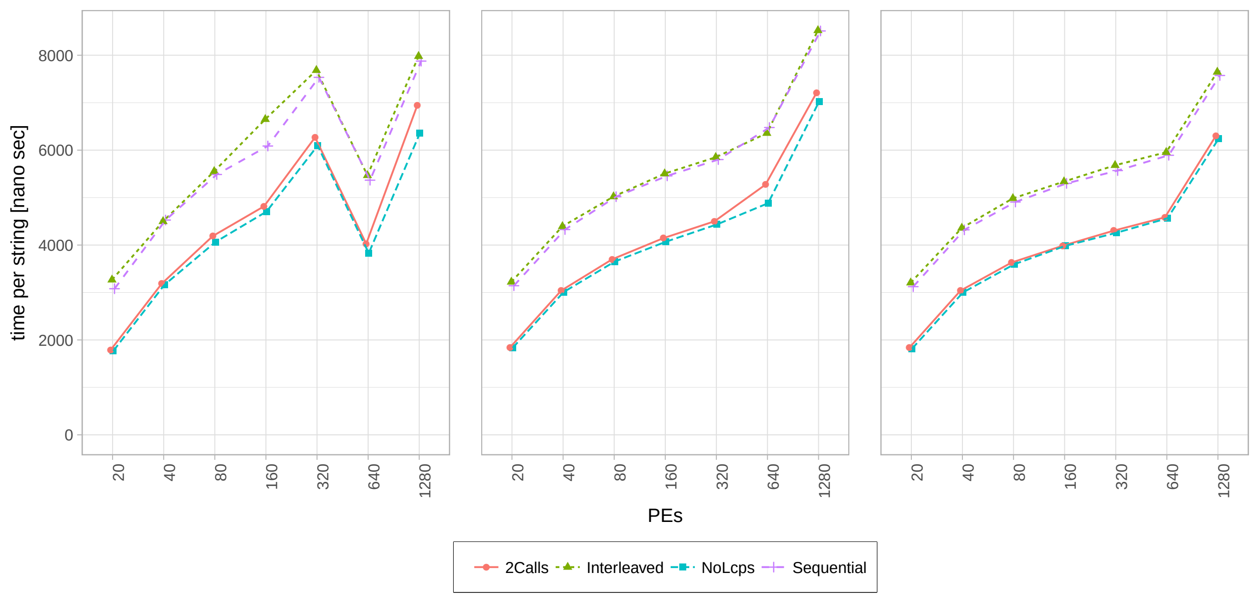


Figure 6.2: Comparison of different variants how strings and LCP values can be sent together. Plots shows (from left to right) the running times for 100000, 500000 and 1000000 strings per PE.

used by Fischer and Kurpicz [11] in the implementation[2] of their distributed string sorting algorithm: The alltoall exchange is executed by sending the messages directly to the other PEs using MPI's asynchronous communication methods `MPI_Isend` and `MPI_Irecv`.

Sending Strings together with LCP Values:

In the string exchange step we do not only send strings but also the LCP array. The easiest way to achieve this are two subsequent calls to `MPI_Alltoallv` – sending the raw-string array first and then the LCP values. We denote this approach `2calls`. However, this induces a factor two of additional start-up latency, since twice as many messages must be sent. Therefore, we also tried out two other patterns. In `sequential` only one byte array is sent. The first part of the array contains the strings as character sequences followed by the LCP array entries cast to bytes. In `interleaved` also only one byte array is sent. However, every string is directly followed by its LCP array entry. Figure 6.2 shows the performance of the different approaches in a weak-scaling setting with data generated by DNGenerator (see Section 7.1.1). Additionally, we also show the performance of `noLcps` in which we do not send the LCP array. We generated three data sets $\mathcal{DN}(s, 500, 0)$ with $s \in \{10^5, 5 \cdot 10^5, 10^6\}$, i.e. each PE obtained $10^5, 5 \cdot 10^5$, and 10^6 strings each of length 500 characters and a minimal (D/N)-ratio. The plot shows the running time of the string exchange including the operations required to copy the strings/LCP entry in the send buffer and to establish the string/LCP array from the receive buffer. The running time is divided by the number of strings of the input to each PE. The plots show the average over 16 iterations of which the first is discarded due to MPI-*warm-up* effects. The experiment has been performed on the cluster computer ForHLR I (see Section 7.2.2). We can observe that on each of the three data sets the variants using only one call to `MPI_Alltoallv` perform worse than the `2call`-approach. This seems to imply that the start-up latency of an additional alltoall exchange is neglectable compared to the local work overhead induced by reestablishing the string/LCP array from one receive buffer. Apart from this, the plot on the left shows a surprising decrease of the communication time for $p = 640$. We believe this to be due to a change in the algorithm MPI uses in `MPI_Alltoallv`.

[2]`https://github.com/kurpicz/dsss`

PE internally we use the `uint64_t` type to store the entries of the LCP array allowing to store integers of a value up to $2^{64}-1$. However, the values of the lengths of the longest common prefixes are usually much smaller incurring a large relative communication overhead when sending these values to other PEs. Hence, we decided to apply 7-bit compression to the LCP values. Algorithm 6.3 describes the compression technique for integers smaller than 2^{16}, i.e. integers that can be represented by two bytes. The compression for four- or eight-byte integers works analogously. By applying 7-bit compression to a `uint64_t`-integer v we can save 7 byte if $v < 127$, 6 bytes for $127 \leq v < 16384$, etc. Only for integers $v \geq 2^{56}$ 7-bit compression yields a representation that requires more than 8 bytes. Since the LCP values in our application are usually quite small in practice, we can save a considerable amount of memory using 7-bit compression.

Algorithm 6.3: 7-Bit Compression

```
Input: Integer $v < 2^{16}$.
1 if $v < 2^7$ then
2     $b[0] \leftarrow v$;
3 else if $v < 2^{14}$ then
4     $b[0] \leftarrow (((v \gg 7) \,\&\, \texttt{0x7F}) \mid \texttt{0x80})$,   $b[1] \leftarrow ((v \gg 0) \,\&\, \texttt{0x7F})$;
5 else
6     $b[0] \leftarrow (((v \gg 0) \,\&\, \texttt{0x7F}) \mid \texttt{0x80})$,   $b[1] \leftarrow (((v \gg 7) \,\&\, \texttt{0x7F}) \mid \texttt{0x80})$;
7     $b[2] \leftarrow ((v \gg 14) \,\&\, \texttt{0x7F})$;
Output: Byte array $b$ containing the 7-bit encoding of $v$.
```

6.4 LCP Losertree

We use the implementation by the authors of [6].

6.5 Distinguishing Prefix Computation

6.5.1 Hashing

The hashing of the string prefixes is done using the fast `xxHash`[3] algorithm. An optimization that we did not apply (and thus could not evaluate) due to time limitations is the following: Let s be a string that remains in the candidate set of Algorithm 5.1 for j rounds. Then for this string the prefixes $p_1 := s[0, 2), p_2 := s[0, 4), \ldots, p_j := s[0, 2^j)$ must be hashed. By storing the hash value h_i of prefix p_i in iteration i, this value can be used to accelerate the computation of the hash value h_{i+1} of prefix $p_{i+1} := s[0, 2^{i+1}]$ with the following formula:

$$h_{i+1} = h_i \oplus \texttt{xxHash}(s[2^i, 2^{i+1}))$$

where $\oplus$ is the bitwise `XOR` operator.

6.5.2 Local Duplicate Detection

In order to reduce the communication volume, strings located on the same PE whose currently investigated prefixes of length l are identical participate only once in the global `distDuplicateDetection`. In the description of the distinguishing prefix computation (Section 5.1.2) it is stated that these local duplicates are detected by scanning over the sorted hash values. This approach implies that the prefixes of length l of all candidate strings must be first hashed and then the created hash values must be sorted. However, local strings s with an equal prefix $s[0, l)$ can also be identified by their LCP array entries using Algorithm 6.4.

[3]`https://github.com/Cyan4973/xxHash`

Algorithm 6.4: Local Duplicate Detection

```
Input: Sorted string array $\mathcal{S}$, associated LCP array $\mathcal{H}(\mathcal{S})$ and the currently investigated
       prefix length $l$.
1 if $|\mathcal{S}| > 0$ then
2     $s \leftarrow \mathcal{S}[0]$;
3     $h[0] \leftarrow$xxHash($s[0, l]$);
4 for $i \leftarrow 1$ to $|\mathcal{S}| - 1$ do
5     if $\mathcal{H}(\mathcal{S})[i] < l$ then
6         $s \leftarrow \mathcal{S}[i]$;
7         $h[i] \leftarrow$xxHash($s[0, l]$);
Output: Array $h$ containing only one hash value per unique prefix $s[0, l)$.
```

Doing so, there is no need to create the hash values for local duplicates at all. Although `xxHash` is claimed to be as fast as `memcpy`, this is nonetheless an operation linear in the currently investigated prefix length. Comparing two LCP values, however, can be done in one machine instruction. The second advantage is that detecting local duplicates before hashing implies that the number of hash values that need to be sorted is reduced.

6.5.3 Sorting/Merging of Hash Values

Due to time limitations we did not develop a special integer sorting algorithm as proposed in [24]. Instead we use the authors' implementation of the fast comparison-based *ips*^{4}*o* [4] algorithm by Axtmann et al. [3]. For merging a losertree implementation of the `tlx`-library is used.

6.6 Hypercube Quicksort

In order to adapt hypercube quicksort to strings, i.e. elements of variable length, we have changed the following aspects of its implementation:

- **Local Sorter:** The local sorter is replaced by the MSD string radix sort variant CI3 without LCP extension.
- **Point-to-Point-Communication/Broadcasts:** Instead of exchanging n elements of fixed length, we apply the same procedure as described in Section 6.3. However, we do not send LCP values as none are computed.

[4]`https://github.com/SaschaWitt/ips4o`

7. Experimental Evaluation

7.1 Data

Given an (unsorted) string array $\mathcal{S}$ it is not possible to predict the time complexity of sorting this array based on the number of strings and the number of characters only. Also, aspects like the sum of the lengths of the distinguishing prefixes and the variance of the string lengths itself can be significant. Therefore, it is not easy to choose data sets for the evaluation of string sorting algorithms. We decided to evaluate the algorithms' performance with particular focus on the (D/N)-ratio, i.e. the ratio between the sum of the lengths of the distinguishing prefixes and the total number of characters. This is an interesting property of a string data set since it has two opposing effects on the running time of our distributed string sorting algorithms. On the one hand, a large value of the (D/N)-ratio makes string comparisons more expensive as more characters of the strings must be inspected. On the other hand, LCP compression is more effective for large values of the (D/N)-ratio than for small values. This can reduce the time needed for communication. To give a systematic evaluation, we designed the so called DNGenerator - a string generator capable of generating string data sets with the number, length and the (D/N)-ratio of the strings as parameters. Additionally, we conducted experiments on real-world string data sets that were chosen according to their (D/N)-ratio.

7.1.1 DNGenerator

The *DNGenerator* is a string generator which creates n strings of length l over an alphabet Σ with a desired (D/N)-ratio of $r \in (0, 1)$. The general idea of this generator is to use the n integers $0, \ldots, n-1$ in an $|\Sigma|$-ary representation as the strings with a (possibly) additional padding at the start and the end of each string in order to reach the requested length. Therefore, the generated strings are pairwise distinct. A detailed description is given in algorithm 7.1. If the tuple of parameters (n, l, Σ, r) fulfills the following conditions

1. $|\Sigma| > 1$
2. $l > \lfloor rl \rfloor \geq \lceil \log_{|\Sigma|}(n-1) \rceil$
3. n is divisible by $|\Sigma|$,

then the sum of the distinguishing prefixes of the generated string set is $D = n\lfloor rl \rfloor$.

Proof. Let $\mathcal{S}$ be the set of strings generated by Algorithm 7.1 using parameters (n, l, Σ, r) that fulfill the above-mentioned conditions. We define $\sigma \coloneqq |\Sigma|$. All strings in $\mathcal{S}$ are of length l (see lines 2 and 4). Let $k \coloneqq \lfloor lr \rfloor$. To represent an integer i in the σ-ary system, exactly $k_i \coloneqq \lceil \log_\sigma(i) \rceil$ digits are required. Since $l > \lfloor lr \rfloor \geq \lceil \log_\sigma(n-1) \rceil$ holds (condition 2), the generated strings are long enough to contain these representations for all integers in $\{0, \dots, n-1\}$.

In lines 5 – 9 the σ-ary representation of integer i is computed and written to $s_i[k - k_i, k)$. The remaining characters of s_i are not modified. Therefore, they still contain the character c – the smallest character of Σ. The last character of s_i is the end-of-string character. Hence, s_i contains the σ-ary representation of i with (possibly) an additional padding at the start and the end. Therefore, the strings generated by Algorithm 7.1 are already sorted.

We now want to determine the sum of the distinguishing prefix lengths of the generated string array. For s_i with $0 \leq i < n$ we define

$$j := \begin{cases} i+1 & \text{if } i = 0 \pmod{\sigma}, \\ i-1 & \text{otherwise} \end{cases}.$$

Note that $0 \leq j \leq n-1$ as n is divisible by σ.

Then $s_i[0, k-1) = s_j[0, k-1)$ as $\lfloor i/\sigma \rfloor = \lfloor j/\sigma \rfloor$ and $s_i[k-1] \neq s_j[k-1]$ as $|i - j| = 1$ and $\sigma > 1$ (condition 1). This implies that at least k characters have to be inspected to differentiate s_i from s_j. Therefore, the distinguishing prefix of s_i is at least k. Since the generated string array is sorted, the distinguishing prefix of a string s_i cannot be longer than the maximum of the length of the longest common prefix of s_i and its neighbour(s) in $\mathcal{S}$ plus one character. However, for $0 \leq i < n$ we find $s_i[k-1] \neq s_{i+1}[k-1]$ as already pointed out above. Therefore, the length of the distinguishing prefix of each string in $\mathcal{S}$ is exactly k. Thus, we have $D = nk = n\lfloor rl \rfloor$. □

As the total number of characters in the generated set equals $N = nl$, we obtain a D/N-ratio of $D/N = n\lfloor rl \rfloor/(nl) = \lfloor rl \rfloor / l \approx r$.

Algorithm 7.1: DNGenerator

```
Input: number of strings $n$, length of string $l$, $D/N$-ratio $r$, alphabet $\Sigma$
1  $k \leftarrow \lfloor l \cdot r \rfloor$, $c \leftarrow \Sigma[0]$
2  $s \leftarrow [c^{l-1}] + [0]$      // string $s$ consists of $l-1$ repetitions of the smallest
                                       // character in $\Sigma$.
3  for $i \leftarrow 0$ to $n-1$ by 1 do
4      $s_i \leftarrow s$                                        // current string
5      $m \leftarrow i$
6      for $j \leftarrow k-1$ to 0 by 1 do    // compute the $|\Sigma|$-ary representation of $m$.
7          $c' \leftarrow \Sigma[m \% |\Sigma|]$
8          $m \leftarrow \lfloor \frac{m}{|\Sigma|} \rfloor$
9          $s_i[j] \leftarrow c'$
10     $\mathcal{S}[i] \leftarrow s_i$
Output: A set $\mathcal{S}$ of $n$ strings each of length $l$ with a $D/N$-ratio of $r$.
```

As already pointed out above, the generated string array is already sorted due to its construction process. Therefore, we randomly permute the strings in the array before it is distributed over the PEs as input to the string sorting algorithms. In the following we will use the notation $\mathcal{DN}(n, l, r)$ to specify a string array generated by Algorithm 7.1 and the parameters n, l and r. Note that we cannot reach a D/N-ratio of 0 or 1 with the DNGenerator. In our evaluation a (D/N)-ratio of 0 or 1 simply represents the minimum of maximum (D/N)-ratio that can be generated with Algorithm 7.1 and the given parameters n and l.

Figure 7.1 shows an example of a generated string array using DNGenerator with $n = 9$ and a (D/N)-ratio of 0.4 on the left and 0.7 on the right.

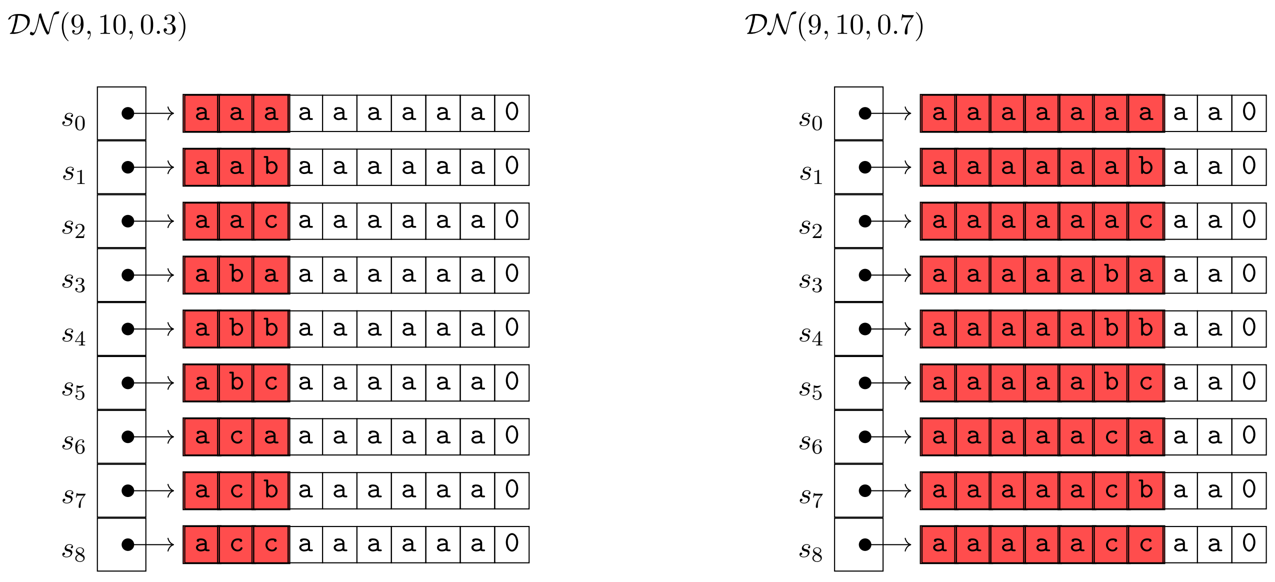


Figure 7.1: Two string arrays generated with DNGenerator. The distinguishing prefix is coloured red.

7.1.2 Skewed DNGenerator

One drawback of DNGenerator is its uniformity. Since all strings are of equal length, string- and character-based sampling yield the same results. We therefore constructed an extension of DNGenerator called SkewedDNGenerator. The smallest p percent of the generated strings are padded with additional characters such that they have e times the length of the other strings. Hence, the SkewedDNGenerator does not change the distinguishing prefix length of any string. However, the (D/N)-ratio of the generated string array is reduced as the value of N increases. In the following we will denote string arrays generated by SkewedDNGenerator with parameters (n, l, r, p, e) by $\mathcal{SDN}(n, l, r, p, e)$. Figure 7.2 shows a string array generated by SkewedDNGenerator consisting of 9 strings with an original (D/N)-ratio of 0.3.

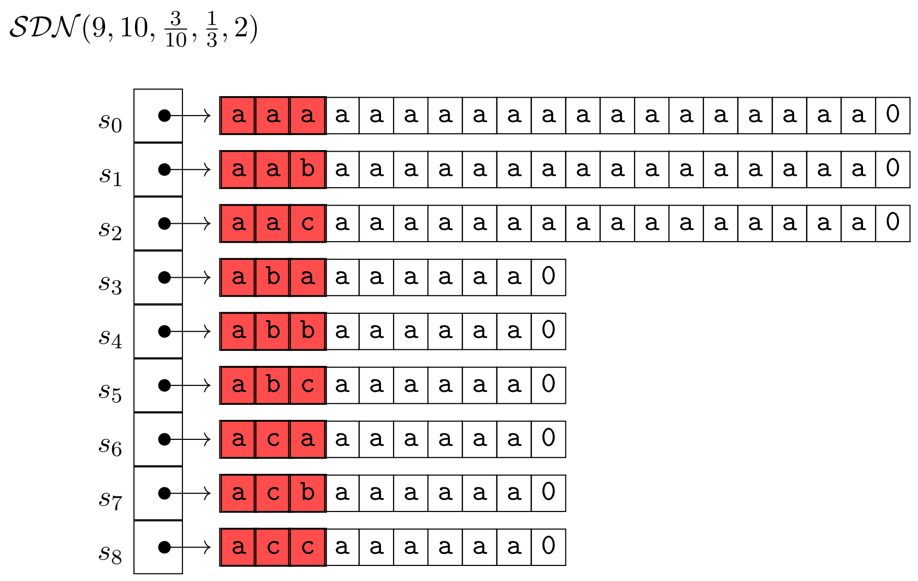


Figure 7.2: String array generated by SkewedDNGenerator. The distinguishing prefixes are coloured red.

7.1.3 Other Data Sets

Apart from string sets generated by DNGenerator and SkewedDNGenerator we have tested our algorithms on three further data set

- CommonCrawl: The data set CommonCrawl consists of the first 200 database files from CommonCrawl (2016-40) [1] and contains text dumps of websites. Each line is interpreted as a string. This data set contains 82 GB of characters in total. Furthermore, we created the set CommonCrawlR based on the original data set. In this data set all strings containing less than 10 characters were deleted.
- Wiki: This data set is a dump of all articles of the English Wikipedia in XML-format from 01/03/19 [2]. Its contains 71 GB of characters. Again, each line is interpreted as a string. Additionally, we created WikiReducedbased on the original data set.

  As above,WikiReduced does not contain any strings of length less than 10.
- Suffixes: The data set Suffixes contains all suffixes of the first 3000 lines of Wiki interpreted as a single string. The suffixes contain about 104 GB characters.

Table 7.1 shows detailed information on all of the above-mentioned data sets.

Table 7.1: Internals of the data sets CommonCrawl,Wiki, Suffixes and the data sets derived from them.

	n	N	(D/N)	$\bar{l}$	$Q_{0.25}$	$Q_{0.5}$	$Q_{0.75}$	$Q_{1.0}$
CommonCrawl	2.0G	82G	0.68	39.54	7	14	26	2,340,013
CommonCrawlR	1.7G	81.1G	0.67	45.27	10	16	31	2,340,013
Wiki	1.1G	71.3G	0.43	65.25	14	25	49	1,521,854
WikiReduced	0.9G	70.4G	0.42	80.46	20	33	54	1,521,854
Suffixes	0.46M	104.5G	0.00	228,566	114,282	228,565	342,847	457,130

The first two columns show the number of strings and number of characters of the data sets respectively. Column $\bar{l}$ contains the average string length. The last columns Q_i shows the string length l such that a fraction i of all strings of the data set contains l or less characters. Hence, $Q_{1.0}$ is the maximum string length of the data set.

7.2 Evaluation Setup

All experiments were performed on the distributed-memory cluster ForHLR I. This cluster consists of 512 20-way Intel Xeon compute nodes. Each of these nodes contains two Deca-core Intel Xeon processors E5-2670 v2 (Sandy Bridge) with a clock speed of 2.5 GHz and have 10x256 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.[3] Intel MPI Library 2018 was used as implementation of the MPI standard. All programs were compiled with GCC 8.2.0 and optimization flags `-O3` and `-march=native`.

7.2.1 Distribution of the Data

In Section 7.1 the data sets used for the experiments are described but we have not yet specified how the data is distributed over the PEs. The input data distribution differs between data sets generated

[1] `https://commoncrawl.s3.amazonaws.com/crawl-data/CC-MAIN-2016-40/wet.paths.gz`
[2] `https://dumps.wikimedia.org/enwiki/20190301/`
[3] `https://wiki.scc.kit.edu/hpc/index.php/ForHLR_-_Hardware_and_Architecture`

by DNGenerator, by SkewedDNGenerator, and the other data sets. The former two data sets are uniformly distributed over all PEs using a random generator, i.e. the probability of a string s to be assigned to PE i is $\frac{1}{p}$ for all strings $s \in \mathcal{S}$. The other data sets (CommonCrawl, Wiki, Suffixes and the sets derived from those) are divided into p parts (respecting string boundaries) such that the amount of characters is nearly the same in each part. These parts are then distributed to the PEs.

7.2.2 Algorithms

We have compared four algorithms against each other on the above-mentioned data sets. The first algorithm is dMSS introduced in Section 4. In this algorithm there are two main strategies to choose. The first one is the sampling strategy to be used in the `partition` algorithm (step 2 of dMSS). The second one is the question whether LCP compression is applied in the string exchange (step 3). We tested all four combinations of these parameters:

- **`msLCPS`**: This variant uses LCP compression and string-based sampling.
- **`msNoLCPS`**: Here LCP compression is not exploited, the string-based sampling approach is used as above.
- **`msLCPC`**: This variant LCP compresses its strings and uses character-based sampling.
- **`msNoLCPC`**: In this variant there is no LCP compression and character-based sampling is used.

The oversampling factor in `partition` in all variants and experiments is 2, i.e. each PE chooses $s = 2(p-1)$ elements for the local sample sets in the second step of the algorithm. Furthermore, we used $100(\tilde{l}+5)$ where $\tilde{l} := \sum_{1\le i\le p} \mathcal{L}(\mathcal{S}_i)$ as maximal string length that is communicated in `partition`. Chosen strings longer than this value were cut off.

The next algorithm is dPDSS (Section 5). In principle, there are three main strategies that can be chosen in this algorithm. The first choice is the question whether Golomb encoding is used in the computation of the distinguishing prefixes. The second and third choices are the sampling method and whether LCP compression is used in the string exchange as above. Since the goal of this algorithm is minimizing the communication volume, we have decided to only test the variant with LCP compression enabled. Another justification for this choice is that no (potentially) expensive decompression is needed as we only compute the sorting permutation and are not interested in the whole strings in this algorithm. Therefore, we have four evaluated dPDSS-variants:

- **`pdNoGolombS`**: In this variant we apply string-based sampling within `partition`. The computation of the (approximate) distinguishing prefix lengths (step 2 of the algorithm) is done without using Golomb encoding by sending the hash values as uncompressed integers.
- **`pdGolombS`**: Here we use Golomb encoding and string-based sampling.
- **`pdNoGolombC`**: This variant applies character-based sampling but no Golomb encoding.
- **`pdGolombC`**: The last variant uses Golomb encoding and character-based sampling.

Again, an oversampling factor of 2 was used in all experiments within the `partition` algorithm. The size of the distributed Bloom filter used to be $2^{64}-1$ in all experiments.

So far, we have only introduced the evaluated variants of our own algorithms dMSS and dPDSS. To thoroughly evaluate new algorithms, it is necessary to compare them against the fastest known algorithms for the considered problem. Although there are many distributed sorting algorithms for atomic keys, we are only aware of one other distributed string sorting algorithm, the algorithm by Fischer and Kurpicz [11]. The algorithm's principal structure is similar to dMSS but no LCP-related optimization is applied and during the computation of the ordered partition (step 2) the local sample sets are not sorted in parallel, but allgathered on the PEs and are then sorted sequentially. In the evaluation we used the authors' original source code.[4] In the following the algorithm will be denoted

[4] `https://github.com/kurpicz/dsss`

fkss. To be able to differentiate between the influence of the sequential sorting of the sample sets and the effect of the LCP-related optimizations in dMSS compared to fkss, we implemented a base variant of dMSS with the following properties:

- **Local Sorting:** For the local sorting of the input arrays the base MSD radix sort string sorter is used. There is no computation of the LCP array.
- **Distributed Order Partitioning:** The computation of the ordered partition is the same as in the other variants of dMSS; in particular hypercube quicksort is used to sort the local sample sets in parallel.
- **String Exchange:** Since we do not know the LCP values, we only communicate the strings.
- **Merging:** We use a normal tournament tree as we do not know LCP values and therefore cannot apply LCP-augmented merging.

This variant is denoted msSimpleS.

The other algorithm against which we compared our two string sorting algorithms is our adaption of the hypercube quicksort by Axtmann and Sanders [1] to string sorting (see Section 3.2). We adapted its original implementation[5] by replacing point-to-point communication of fixed length with point-to-point communication of variable length. In the following the algorithm will be denoted hQuick.

7.3 Evaluation

In the following we present multiple experiments to evaluate the performance of the aforementioned algorithms. We will especially analyse the effect of LCP-related optimization for compression and merging and the influence of string- and character-based sampling. For dPDSS the effect of Golomb encoding on running time and communication volume will be of interest. The reported values for running times and communication volume in the experiments are averages over 15 successive runs of the algorithms. In total we executed 16 runs. However, the first iteration always used to be discarded due to MPI *warm-up* effects. Since our algorithms are likely to be a subroutine of another algorithm with previous communication, we feel that this decision is valid.

7.3.1 DN-Data Weak Scaling

In this experiment we ran the algorithms in a weak-scaling setup on data generated by DNGenerator and SkewedDNGenerator.

Regular Data

For a number of p PEs the algorithms have been evaluated on the data set $\mathcal{DN}(5p \cdot 10^5, 500, r)$ with $r \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. In other words we created 5 data sets each consisting of 500.000 strings of length 500 per PE with (D/N)-ratios from 0.0 to 1.0 with a stepping of 0.25. The upper part of Figure 7.3 shows the running time of the algorithms. In the lower part of the figure the total number of bytes sent divided by the number of strings is shown. To obtain this number, we summed up all bytes sent by any PE during the execution of an algorithm. In point-to-point communication and in the alltoall operations, we simply added the length of the messages. In the allgather and broadcast operations, we multiplied the length of the sent messages with the number of PEs. The plot in between these two shows the running time of the evaluated variants of dMSS and dPDSS in more detail. Table A.1 contains the exact numbers. Table B.6 shows the speed-up of the algorithms against the fastest of our competitors for each data set/number of PEs. Since all strings in the data set are of equal length, there is no difference between string- and character-based sampling (see Table A.1). Hence, we do not show either msLCPC/msNoLCPC or pdGolombC/pdNoGolombC in the plots.

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

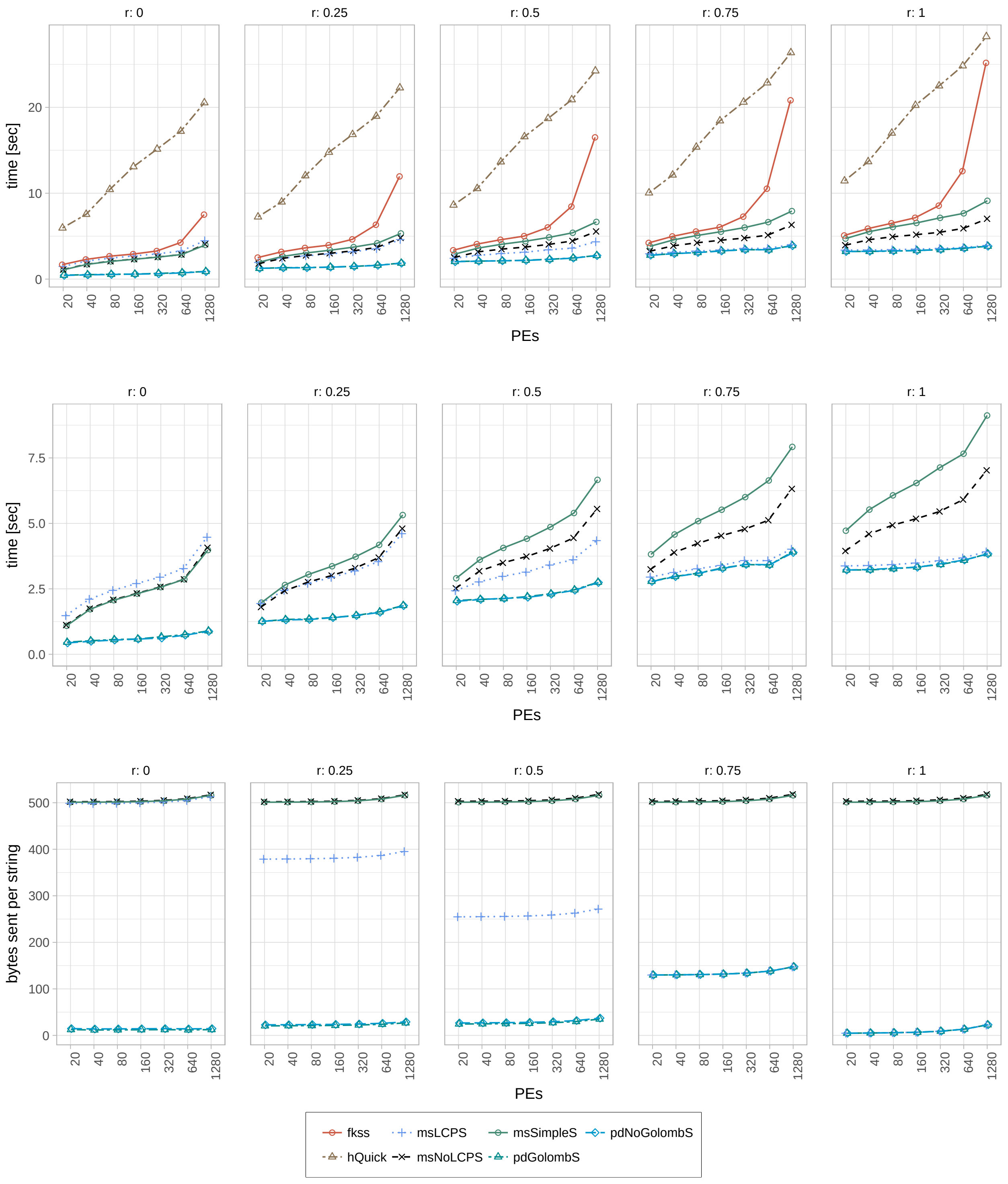


Figure 7.3: Running times and number of bytes sent per string in the DToN weak-scaling experiment.

The figure reveals that all of our dMSS- and dPDSS-variants clearly outperform their competitors **`hQuick`** and **`fkss`**. The algorithm **`hQuick`** shows the worst performance of all algorithms. This comes at no surprise as it has not been designed for string sorting and its (theoretical) communication volume is $\Theta(\log(p))$ times greater than the communication volume of dMSS even without LCP compression enabled. Furthermore, **`hQuick`** effectively uses only $2^{\lfloor \log(p) \rfloor}$ PEs. Since p is not a power of two in our experiments, this is another reason for its poor performance. The running time of **`fkss`** surprises as it does not scale at all for an increasing number of PEs. This becomes even worse with an increasing (D/N)-ratio r. The base variant msSimpleS of dMSS does not show this behaviour. Since the only algorithmic difference to fkss is its handling of the sorting of the local sets within partition, we believe the sequential sorting approach of fkss for this sub-step to be the reason for its low performance.

The comparison of the running time of msSimpleS, msNoLCPS and msLCPS reveals some interesting insights into the effect of LCP-based optimizations. For $r = 0$ msNoLCPS and msSimpleS, which nearly show the same performance, are on average faster than msLCPS by a factor of 1.2 over all PEs. This is reasonable since LCP compression incurs additional costs for compressing and decompressing the strings and since the reduction of communication volume is neglectable for this data set. For the other data sets ($r \geq 0.25$) we observe two effects. The first one being that LCP-augmented merging pays off as msNoLCPS is faster than msSimpleS. For $r = 1$ the variant with LCP-merging is on average faster than msSimpleS by a factor of 1.3 over all PEs. The other effect is even stronger: LCP compression improves the running time for an increasing (D/N)-ratio considerably. For $r = 1$ msLCPS (LCP values used for compression and merging) shows on average an speed-up of 1.9 compared to msSimpleS . Another effect of LCP compression is the following: All algorithms but msLCPS become slower for increasing r. This is as expected since the length of the distinguishing prefixes increases and more characters must be inspected during string comparisons. However, an increase in (D/N)-ratio also implies greater LCP values for the data set. Hence, LCP compression becomes more effective. This compensates the greater amount of time for local string comparisons and explains why the total running time of msLCPS increases more slowly than that of msNoLCPS and even decreases (!) for $p = 1280$ with increasing r.

Our prefix-doubling-based algorithm dPDSS has the best running time on all data sets and numbers of PEs. For $r = 0$ it is faster than the best variant of dMSS by a factor of 3.6 on this data set. With increasing r the advantage of dPDSS over the best variant of dMSS, which is msLCPS for $r \geq 0.25$, becomes smaller as the distinguishing prefixes increases, but the total length of the strings remains the same. This has two effects: First, dPDSS must execute more rounds in the approximate computation of the distinguishing prefix lengths (second step of the algortihm and described in Section 5.1.2). Second, the number of characters saved by communicating the distinguishing prefixes only becomes smaller. This explains that for $r = 0.75$ and $r = 1$ there is almost no difference between either dPDSS-variant and msLCPS. Another observation is that there is almost no difference between the two dPDSS-variants in terms of running time. Considering the total number of sent bytes, we observe a very slight advantage of pdGolombS over pdNoGolombS. Hence, Golomb encoding does reduce the communication volume, but this effect is not significant enough to also reduce the overall running time.

The lower part of Figure 7.3 shows the number of bytes communicated per string during a run of the dMSS and dPDSS algorithms. All strings are of length 500 and one character can be sent in one byte for the used alphabet. The number of sent bytes grows with increasing p. This is as expected as the total number of elements chosen for the global sample set in partition is in $\Theta(p^2)$. This increase is rather small in the data sets with $r = 0$ and $r = 0.25$ for dPDSS as the computation of the buckets only considers the distinguishing prefixes, which are rather short. For $r = 0.75$ the plot shows that about 135 bytes are communicated per string. This corresponds to the number of characters that can be saved by LCP compression (plus some additional bytes sent in steps 2 and 3). Here we do not gain any advantage by the computation of the distinguishing prefixes. This can be explained as we only test whether the prefixes of lengths $2, 4, 8, 16, \ldots$ are unique. Since the (D/N)-ratio is 0.75 and the string length is 500 characters, the distinguishing prefix length is 375 and we have $2^8 < d_{max} = 375 < l_{max} = 500 < 2^9$. Hence, our algorithm cannot detect the actual distinguishing prefix length of 375 and sets the distinguishing prefix length to 500 – the complete string length. The same happens for $r = 1$. Thus, only LCP compression is applied which yields nearly the same communication volume per string for pdNoGolombS, pdGolombS and msLCPS on these two data sets.

Skewed Data

On data generated by DNGenerator the differences between string- and character-based sampling cannot be evaluated as both deliver the same results. Therefore, we executed the same experiments as above on data sets generated by SkewedDNGenerator. Figure 7.4 shows the performance of the algorithms on the data sets $\mathcal{SDN}(5p \cdot 10^5, 500, r, 0.2, 4)$ for $r \in \{0, 0.25, 0.5, 0.75, 1.0\}$, i.e. we generated the same set as above but the smallest 20 percent of the generated strings were padded such that they were four times as long as the other strings. Recall that the distinguishing prefix length of the strings is not influenced by this padding. The effect is that – if string-based sampling is used – PEs with a small index obtain up to four times as much data as the other PEs. For character-based sampling the increase in characters per PE should be well balanced. Table A.2 contains the exact running times and numbers of bytes sent per string. Table B.6 shows the speed-up against the fastest of our two competitors.

As above `fkss` performs worse than our algorithms on all data sets. Only `msLCPS` is slower than `fkss` for $r = 0$ and $p \leq 320$. For these runs the overhead of LCP compression seems to be too high. The hypercube algorithm `hQuick` could only be executed on $p \leq 80$. On more PEs it crashes due to memory limitations. This can be explained by its hypercube design as some compute nodes obtain initially twice as much data as in the other algorithms (recall that `hQuick` only uses $2^{\lfloor \log(p) \rfloor}$ PEs).

The positive effect of LCP-merging on the running time of the string-based sampling variants is still existent. The variant `msNoLCPS` is with $r = 1$ about a factor of 1.2 faster on average than `msSimpleS`. However, LCP compression does no longer seem to pay off. The variant `msLCPS` is slower than `msNoLCPS` on all data sets except for $r = 1$. Even here the LCP compression variant has only a speed-up of 1.05 compared to `msNoLCPS`. This can be explained by the skewness of the data. Since we measure the length of the critical path, i.e. the "slowest" PE, one slow processor alone can deteriorate the overall performance. As PEs with a small index obtain 4 times as much data as the other PEs and at least 75% of this data cannot be LCP-compressed (depending on r), the poor performance of `msLCPS` seems reasonable.

Let us now consider the character-based sampling methods. The plot reveals that these perform much better than their string-based sampling counterparts. For $r = 0$ `msNoLCPC` is faster than `msSimpleS` by a factor of 1.3 – the fastest string-based sampling variant on this data set. For $r = 1$ we observe the performance of `msLCPC` to be better than that of `msLCPS`, which is the fastest string-based variant on this set, by a factor of 1.4 on average over all PEs. This shows that character-based sampling is a useful alternative on data sets with skewed data. Additionally, LCP compression also pays off for $r \geq 0.5$ although the gain is only 5% to 10% even for $r = 1$.

As above, the dPDSS variants perform best on all data sets. This time even for $r = 0.75$ and $r = 1$ dPDSS is clearly the best algorithm. Recall that in the experiment with regular data, the `msLCPS`-variant nearly shows the same performance as dPDSS. For $r = 0$ `pdNoGolombS` shows on average a speed-up of 5.3 compared to the quickest dMSS-variant over all PEs. The reason why the performance of dPDSS is barely influenced by the skewedness of the data lies in its robustness to varying string lengths as long as the distinguishing prefixes are not affected. This is the case for data generated with SkewedDNGenerator. Thus, it comes at no surprise that the running times of the dPDSS variants is nearly the same in this and the last experiment. We did not include the running time of `pdNoGolombC` and `pdGolombC` in this plot since these are nearly identical to their string-based sampling counterparts (see Table A.2). This is as expected as dPDSS operates on the distinguishing prefixes only and these are all of equal length.

The number of bytes sent per string (lower part of Figure 7.4) shows the same trends as in the experiment on regular data with only difference that the values are shifted for the dMSS-variants. Again, dPDSS is quite robust against the skewness of the data. Since the longer strings are expected to be cut off at a length of 512 characters, basically the same amount of data is sent as in the experiment on regular data.

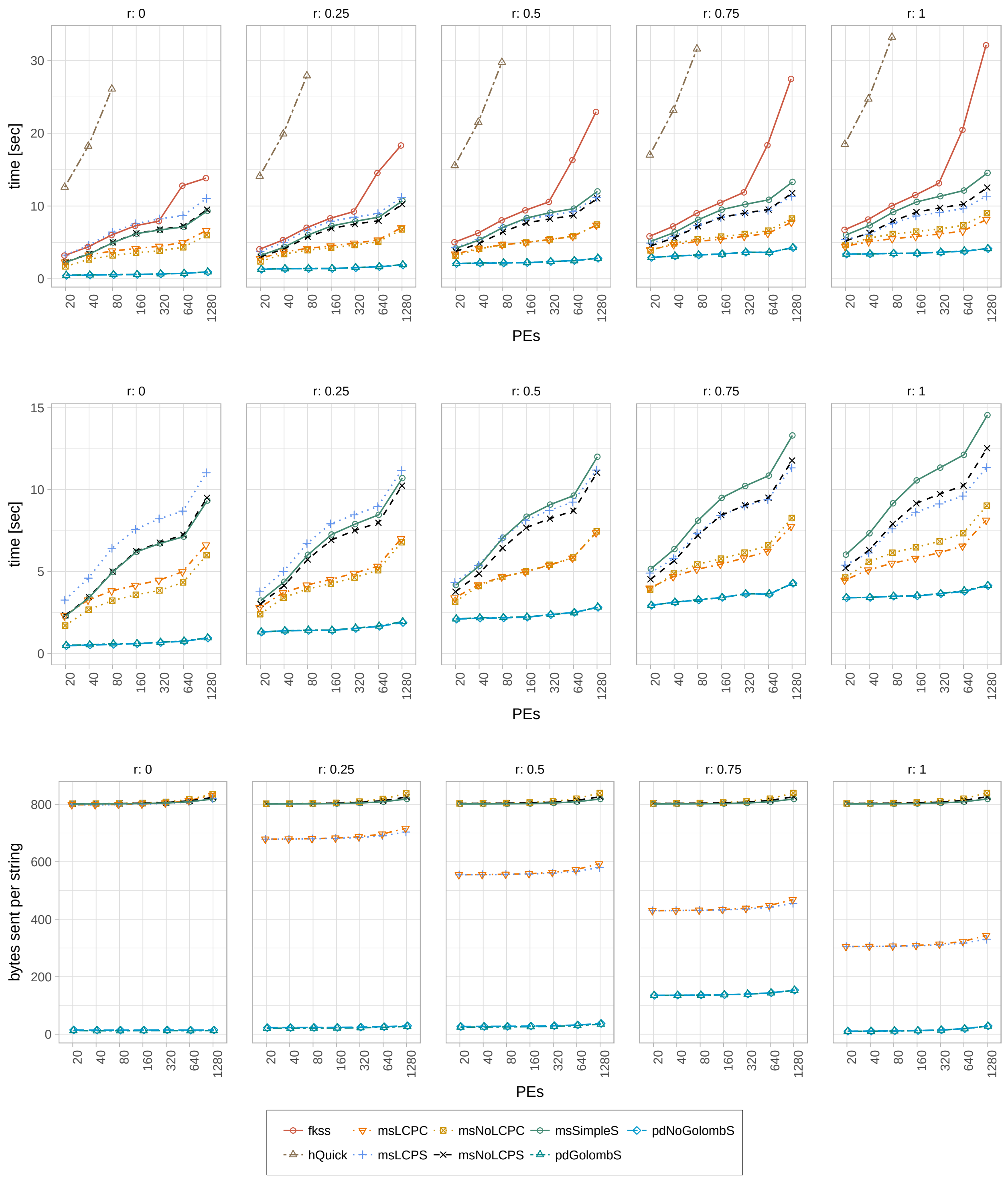


Figure 7.4: Running times and number of bytes sent per string in the skewed DToN weak-scaling experiment.

7.3.2 DN-Data Strong Scaling

We also performed strong scaling experiments on data generated by DNGenerator and SkewedDNGenerator.

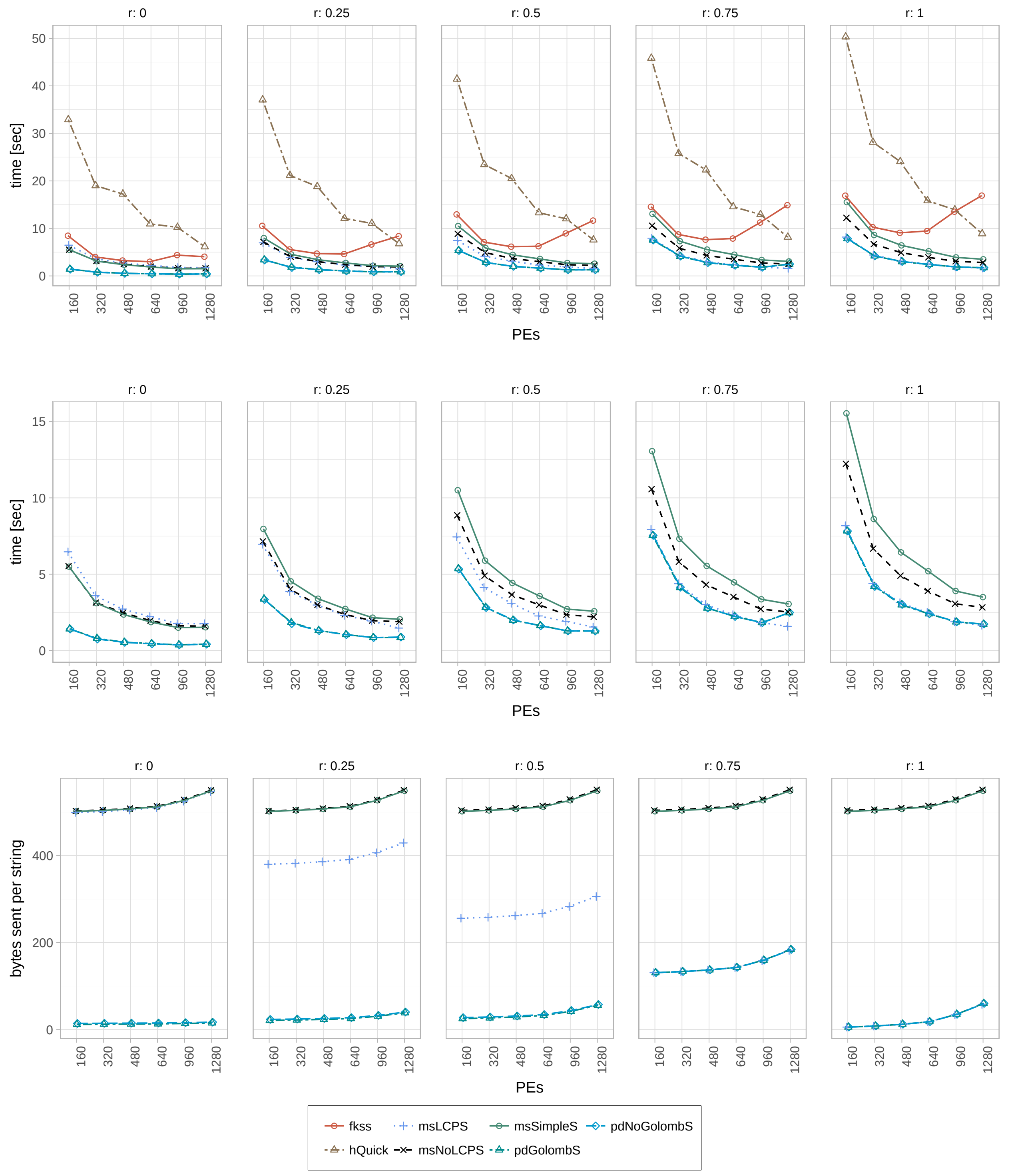


Figure 7.5: Running times and number of bytes sent per string in the strong-scaling experiment on DToN data.

For the former we created five data sets $\mathcal{DN}(2 \cdot 10^8, 500, r)$ with $r \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$. The five data sets created with SkewedDNGenerator have the parameters $\mathcal{SDN}(2 \cdot 10^8, 500, r, 0.2, 4)$ and

$r \in \{0.0, 0.25, 0.5, 0.75, 1.0\}$, i.e. the smallest twenty percent of the strings are padded such that they have four times the length of the other strings.

Regular Data

Figure 7.5 shows the running time for the data sets generated with DNGenerator. The top row shows the running times of all algorithms, the middle displays only the dMSS- and dPDSS-variants. The bottom row in the figure shows the number of bytes sent per string for the variants of our two algorithms. Detailed numbers can be found in the appendix in Table A.3 and Table B.7. As in the weak-scaling experiments, all variants of both of our algorithms – dMSS and dPDSS – perform better than their competitors. Although `hQuick` scales quite well for increasing p, `fkss` even gets slower for $p > 640$. For $p = 1280$ (and $r \geq 0.25$) it performs even worse than `hQuick`. The characteristic plot of `hQuick`, which resembles a staircase, can be explained by its hypercube design. For sorting `hQuick` uses $p' := 2^{\lfloor \log(p) \rfloor}$ PEs only. PEs with an index greater than p' send their input to PEs within the hypercube of dimension $\log(p')$. For $p = 320$ and $p = 480$ the value of p' is identical. The same holds for $p = 640$ and $p = 960$. This explains the similar performances on these pairs of PEs. Apart from this particularity of `hQuick` the experiment does not reveal any new insights. The overall effects of LCP-merging and compression are similar to those in the weak-scaling experiment. We do not display the results for the character-based sampling variants of our experiments in the plots as all strings are of equal length and they therefore show the same performance.

Skewed Data

Figure 7.6 shows the running time and communication volume of the algorithms on the data sets generated by SkewedDNGenerator. Again, exact numbers and the speed-up can be found in the appendix in the Tables A.4 and B.7.

The algorithm `hQuick` could only be executed on $p \geq 640$. The missing data points are caused by the memory limitations of the compute nodes since the PEs with small indices obtain the same number of strings as the other PEs but these strings are significantly longer. But also `fkss` and the string-based variants of dMSS demand too much memory for $p = 160$. The prefix-doubling algorithm and the character-based sampling variants do not suffer from this problem. The prefix-doubling algorithm is robust against skewed input because the distinguishing prefix lengths are not influenced by the additional padding as mentioned above. The character-based sampling variants of dMSS can handle the skewed input by adjusting the generated buckets in the sampling phase to the varying string lengths.

All of our algorithms are clearly faster than their competitors; `fkss` confirms the results of the weak-scaling experiment that it does not scale, especially not for increasing values for r. The fastest algorithms on all data sets are the prefix doubling variants. They have a speed-up of about 6.4 compared to the best dMSS-variant for $r = 0$ and speed-up of 1.8 over the fastest dMSS-variant (`msLCPC`) for $r = 1$. Similar to the weak-scaling experiment on skewed data the plots show the character-based sampling variants to be on average about a factor of 1.6 ($r = 0$) and 1.4 ($r = 1$) faster than their string-sampling counterparts. For dPDSS there is no difference between string- and character-based sampling as the distinguishing prefix of the strings is not changed by the additional padding at the end of the 20 percent smallest strings (see Table A.4). Therefore, they are not shown in the plot. The effects of LCP-related optimizations are the same as in the weak-scaling experiment.

7.3.3 CommonCrawl

We now give the results of the runs of our algorithms on real world data. The left-hand plot in Figure 7.7 shows the running time on CommonCrawl; likewise the right-hand plot the running time on CommonCrawlR. The plots in the second row of the figure show the running time of our algorithms – dMSS and dPDSS – in more detail. The third row displays the number of bytes sent per

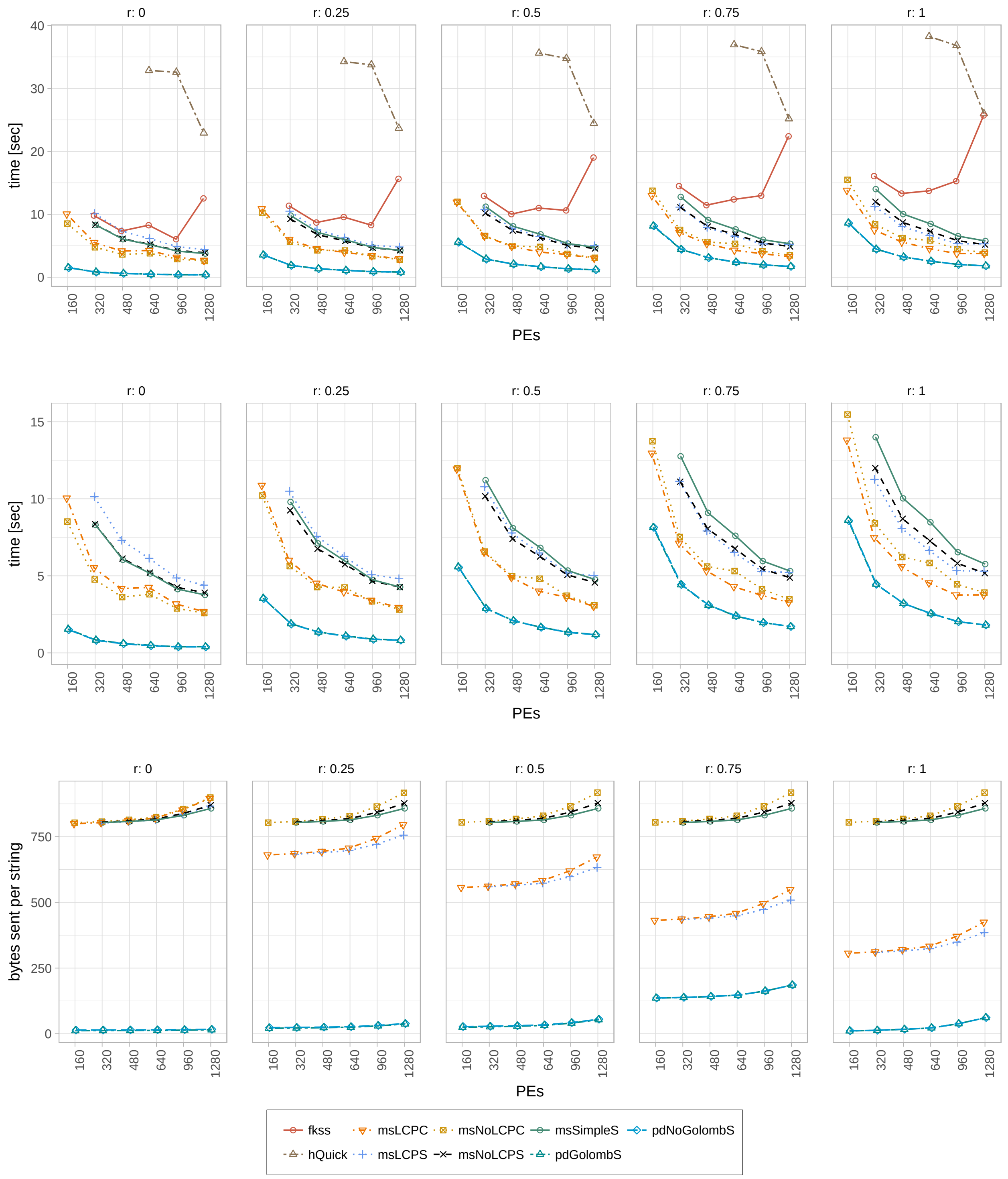


Figure 7.6: Running times and number of bytes sent per string in the strong-scaling experiment on skewed DToN data.

string on both data sets. The exact values can be found in the appendix in Table A.5. The speed-up of our algorithms compared to `hQuick` is given in Table B.8. This experiment is strong-scaling, i.e. we applied the same data set to the different numbers of PEs.

We cannot provide the running time of `fkss` as it crashes on this data set. We believe this happens because of the large number of duplicates in both data sets. The algorithm `fkss` does not address this problem. Our algorithms can distinguish duplicates within `partition` by adding a unique identifier to each string.

Again all of our algorithms perform better than their only remaining competitor `hQuick`. The string-based variants of dMSS show an average speed-up of about 4 over `hQuick`. The dPDSS algorithm is faster than its competitor by a factor of 5.4 on average. The effect of LCP merging seems to pay off as `msSimpleS` performs notably worse than `msLCPS` and `msNoLCPS` (by a factor of 2).

The positive effect of LCP compression is still existent but rather small. For $p > 160$ dPDSS shows the best performance. This is a bit surprising as the (D/N)-ratio of this data set is relatively high. The bad performance of the dMSS-variants using character-based sampling is particularly interesting. Their running time is about a factor of two worse than the one of their string-based sampling counterparts. A closer look reveals that they spend much more time in the merging step than the string-based sampling variants. The data set CommonCrawl contains many very short duplicates. In character-based sampling this implies that the number of strings sent to one PE whose bucket contains many short strings is considerably higher than the number of strings in a bucket with longer strings. This seems to negatively affect the running time of the merging step. To verify our assumption, we created the data set CommonCrawlR by deleting all strings from CommonCrawl with a length less than 10. On CommonCrawlR `msNoLCPC` and `msLCPC` perform even better than their string-based sampling counterparts. Hence, we conclude that it is not just very long strings which can deteriorate the performance of character-based sampling (as we can no longer give guarantees on the number of characters in a bucket) but also very short duplicates.

The performance of the character-based sampling variants of dPDSS is even worse than the one of the corresponding dMSS-variants on both CommonCrawl and CommonCrawlR. A closer look on the detailed running times of the single step revealed that this behaviour has two causes. The first is the same as for dMSS: There are many short duplicates resulting in a few PEs containing significantly more strings than the other PEs. This slows down the merging step. The computation of the ordered partitions and especially the sorting sub-step of this method accounts for the second reason: in character-based sampling longer strings are more likely to be chosen to form the elements of the local sample set as we sample the character array equidistantly. Therefore, more data needs to be sent and longer strings need to be inspected in string comparisons to sort the local sample sets globally. This can be seen in the increase of bytes sent per string in the lower part of Figure 7.7. All this also holds for the character-based sampling variants of dMSS. The difference is that in dMSS the chosen strings are cut off after $\overline{d} := 100(\sum_{i=1}^{p} \mathcal{L}(\mathcal{S}_i) + 5)/|\mathcal{S}|$ characters due to the heuristic presented in Section 4.1.2. In dPDSS the whole distinguishing prefixes are sent. A closer look at the input to the partitioning algorithm revealed that the distinguishing prefixes sent in dPDSS within `partitionCB` are much longer than the strings cut off after $\overline{d}$ characters in the same subroutine for dMSS on this data set. This can be explained as follows: CommonCrawl also contains very long duplicates. For these there is no difference between distinguishing prefix and the string itself. Therefore, the strings drawn from the input array during character-based tend to have long distinguishing prefixes. On the other hand $\overline{d}$ is computed over the (local) LCP array entries of *all* strings and is (although multiplied by a factor of 100) therefore much smaller than the average distinguishing prefixes length of the longer (potentially duplicate) strings of the local character-based sample sets for this data set. As this is not affected by deleting small strings, the running times suffers from the same problem in CommonCrawlR, too.

Let us now consider the plot showing the number of bytes sent per string. As in the other experiments, there is a slight increase in this number for increasing p for all algorithms but `pdNoGolombC`/`pdGolombC`. The increase for the character-based sampling methods of dMSS is a bit stronger than for their string-based sampling counterparts since longer strings a more likely to be chosen for the local sample set in these strategies. This causes more communication in the sorting of the sample sets. The strong increase for the character-based sampling methods of dPDSS has

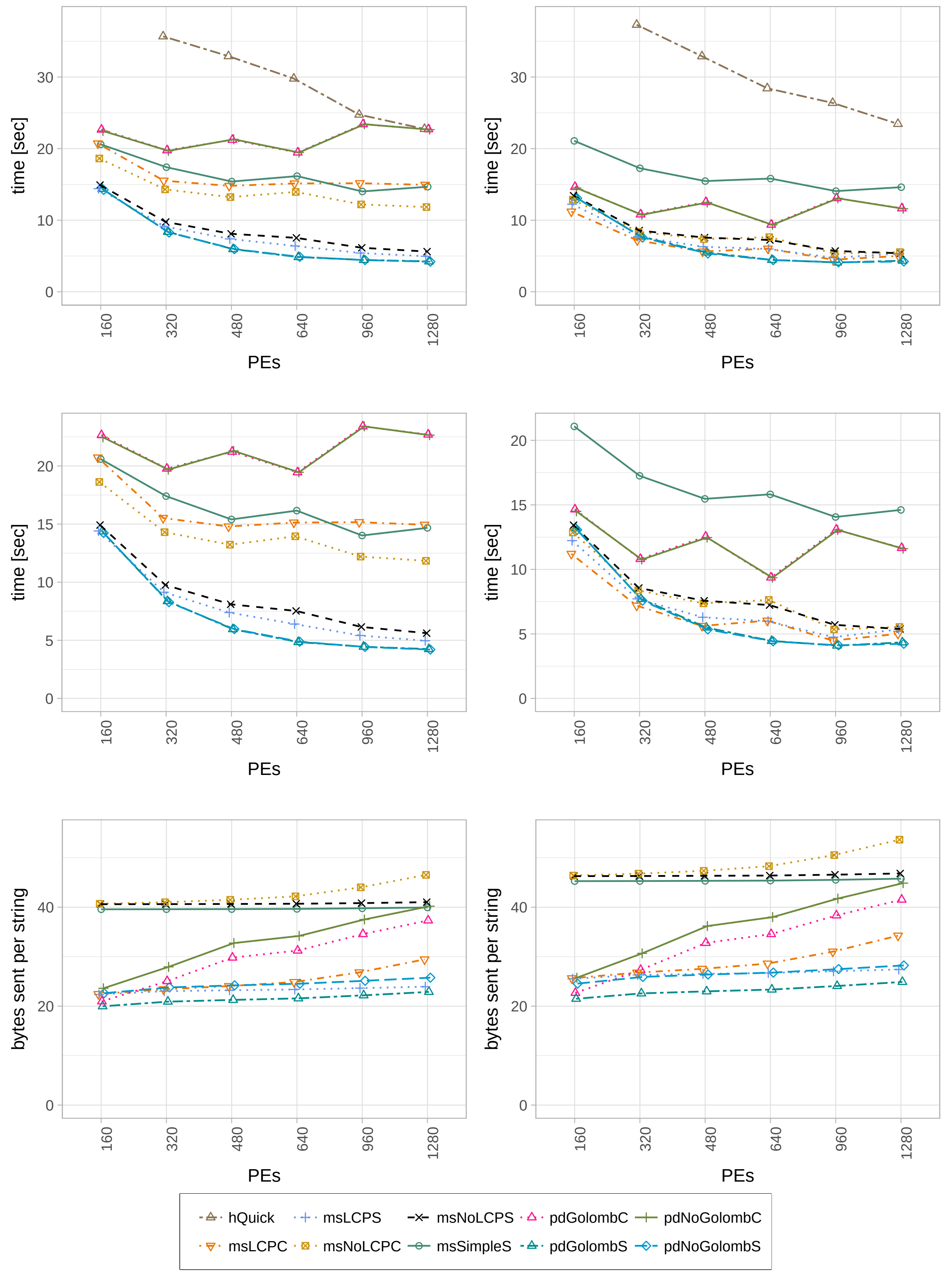


Figure 7.7: Running times and number of bytes sent per string in the strong-scaling experiment on CommonCrawl (left-hand side) and CommonCrawlR (right-hand side).

already been explained above. As before, we can observe that `msSimpleS` needs one byte less than `msNoLCPS`. This is due to the additional communication of the LCP values in the latter variant. Due to 7-bit compression the LCP values can be sent in one byte on average. Without 7-bit compression this would cause additional 7 bytes per string. The average string length of CommonCrawl is 39.5 characters, for CommonCrawlR we find the average length to be 45.2 characters. This corresponds very well to the actual communication per string for the non-compressing variants `msSimpleS`, `msNoLCPS` and `msNoLCPC`.

Although the (D/N)-ratio of CommonCrawl is 0.68 (0.67 for CommonCrawlR), only about half the bytes can be saved with LCP compression. This is explained by Lemma 2.1, which states that the sum of all distinguishing prefix lengths can be more than twice as large as sum of the entries in the LCP array. The dPDSS-variants communicate only slightly less than the dMSS variants with LCP compression enabled. Again, this is due to the high (D/N)-ratio; there are simply too few “unnecessary” characters that could be cut off.

7.3.4 Wikipedia

Figure 7.8 shows the running time on Wiki (upper left-hand plot) and on WikiReduced (upper right-hand plot). The statistics on the communication volume per string can be found in the two lower plots of the figure. The two plots in the centre show the running time of our two algorithms in more detail.

As before, `fkss` could not be run on this instance. Wiki contains many short duplicate strings, which we believe to be the reason for the crashing of the algorithm.

All of our algorithms are faster than their competitor `hQuick`. The step-form of `hQuick` is due to its hypercube design. As for CommonCrawl, we observe that the character-based sampling variants of dMSS are slow and do not scale for $p > 480$ anymore. This changes for WikiReduced. Here `msNoLCPC` and `msLCPC` outperform their counterparts. The character-based sampling variants (both show the same performance) are on average a factor of 1.3 faster than the best string-based sampling variant (`msNoLCPS`) over all PEs. The comparison of `msSimpleS` and `msNoLCPS` shows that LCP merging is beneficial even for the relatively low (D/N)-ratios of 0.43 for Wiki and 0.42 in WikiReduced. The variant `msNoLCPS` is about a factor of 1.3 faster than `msSimpleS` on both data sets. LCP compression, however, does not further improve the running time. In contrast to CommonCrawl it even slightly deteriorates the performance. Here the (D/N)-ratio seems to be too low to compensate for the additional overhead of compression and decompression in this variant. This time, the character-based sampling variants of dPDSS show the expected behaviour. They are slower than their string-based counterparts on Wiki due to the high number of short duplicates and the slow-down of the merging step caused by this. Since the distinguishing prefixes are shorter for Wiki, the `partition` algorithm does not seem to cause a problem. This results in a nearly equal running time of `pdNoGolombS`/`pdGolombS` and `pdNoGolombC`/`pdGolombC` on WikiReduced.

The plots containing the average number of bytes sent per string show that the communication per string without compression corresponds very well with the average string length of 65.2 characters for Wiki and 80.5 characters for WikiReduced. Again, character-based sampling needs more communication, as longer strings are more likely to be chosen during the computation of the ordered partition. dPDSS can reduce about half of the communication compared to sole LCP compression. The (D/N)-ratio is lower than the one in CommonCrawl. Therefore, our algorithm can cut off more characters that are not required to sort the strings. It can be seen that Golomb encoding can reduce the number of bytes by 3 compared to `pdNoGolombS`.

7.3.5 Suffixes

In this experiment we evaluated the algorithms on the data set Suffixes – suffixes created from text interpreted as a single string. These suffixes have the interesting property that they are quite long on average (about 200,000 characters) and that the (D/N)-ratio is very small. The upper part of Figure 7.9 shows the running time of the algorithms on this data set. The lower part states the number of bytes sent per string. The row in the middle displays a more detailed view on our algorithms dMSS and dPDSS.

As with the other real world data sets we could not run `fkss`. This time the long strings and the sequential sorting of the sample sets seems to be the problem.

All of our algorithms are clearly faster than `hQuick`. The most interesting observation in Figure 7.9 is that dPDSS is very fast compared to dMSS. For $p = 160$ the dPDSS-variants have a speed-up of 29 compared to `msLCPC` – the fastest variant of dMSS. For $p = 1280$, `pdNoGolombS` is still faster than the best variant of dMSS (`msSimpleS`) by about a factor of 7 for this number of p. The main reason for this can be seen in the lower part of Figure 7.9. For $p = 160$ only 0.0005% of the number of bytes sent by the dMSS-variants is required. For $p = 1280$ dPDSS needs 0.007% of the communication of the dMSS-variants. There is no noticeable difference between the character- and string-based sampling methods of dPDSS.

However, `hQuick` and the dMSS-variants scale quite well, whereas dPDSS becomes slightly slower for $p \geq 960$. Nevertheless, this data set shows that in practice the approximate computation of the distinguishing prefix lengths works quite well in dPDSS.

7.4 Summary

Our experiments show that our two algorithms clearly outperform `fkss` and `hQuick`. Furthermore, LCP-related optimizations improve the performance of distributed string sorting on most data sets. It is only for small (D/N)-ratio that they do not accelerate the sorting process. Moreover, the evaluation clearly shows that for merge sort-based algorithms the partitioning algorithm has to be parallelized as the sequential sorting in this step accounts for most of the deficits of `fkss`. Our new character-based approach to computing an ordered partition of the input has been proven to be beneficial for skewed inputs. However, data sets with small duplicates can be a problem for this variant. Also, `partitionCB` itself can be more expensive than `partitionSB` as the chosen sample sets usually contain longer strings. Depending on the used heuristic to cut off the strings in the sample set (or the distinguishing prefixes itself for dPDSS) this effect can become significant. The algorithm dPDSS also shows very good performance, it is the fastest algorithm on almost all data sets/number of PEs. Especially on data sets with a small (D/N)-ratio it is considerably faster than the other algorithms. However, one should keep in mind that this algorithm only computes the sorted distinguishing prefixes/the permutation defining the sorted order of the input and does not output the sorted strings themselves.

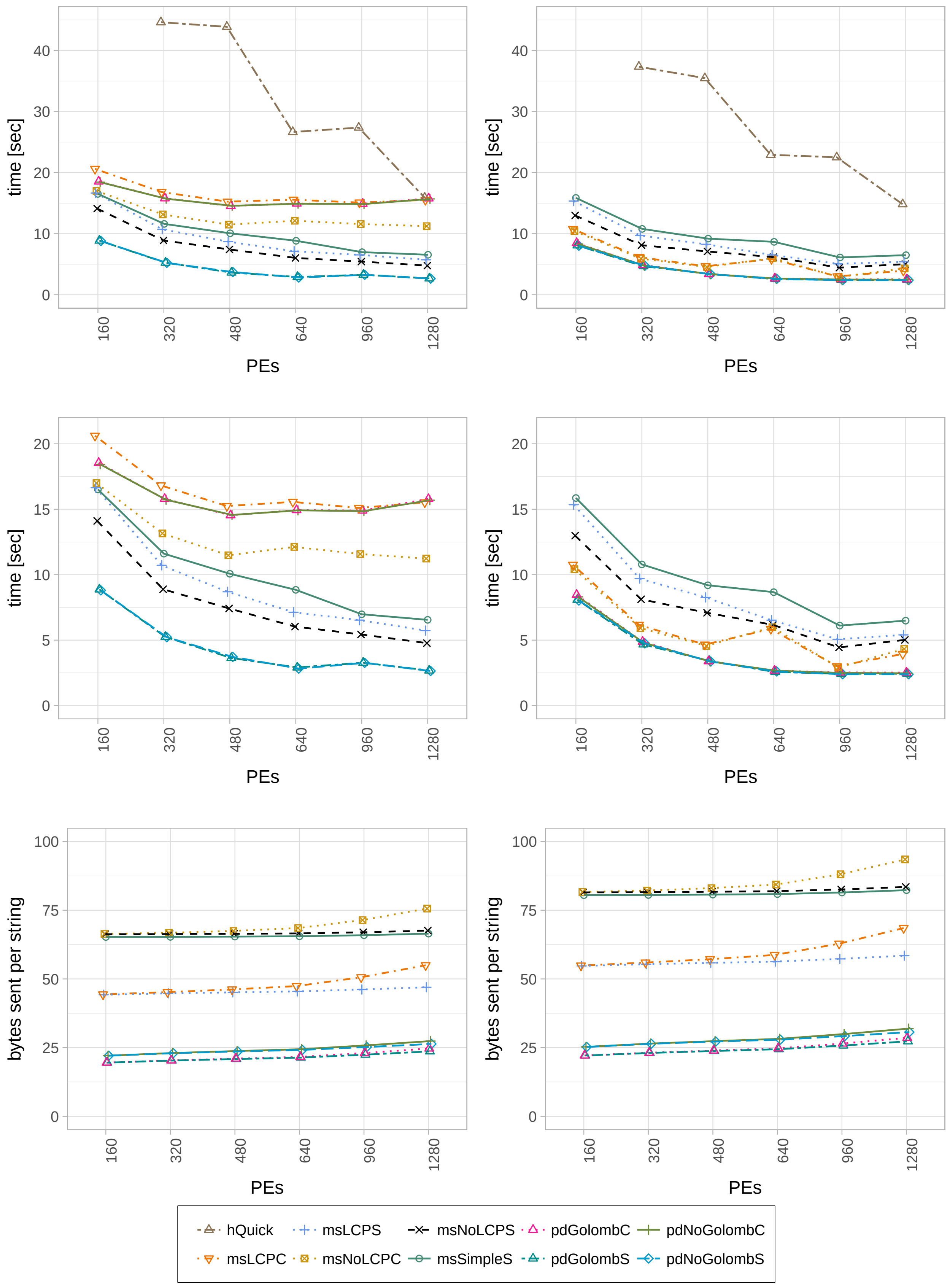


Figure 7.8: Running times and number of bytes sent per string in the strong-scaling experiment on WIKI (left-hand side) and WIKIREDUCED (right-hand side).

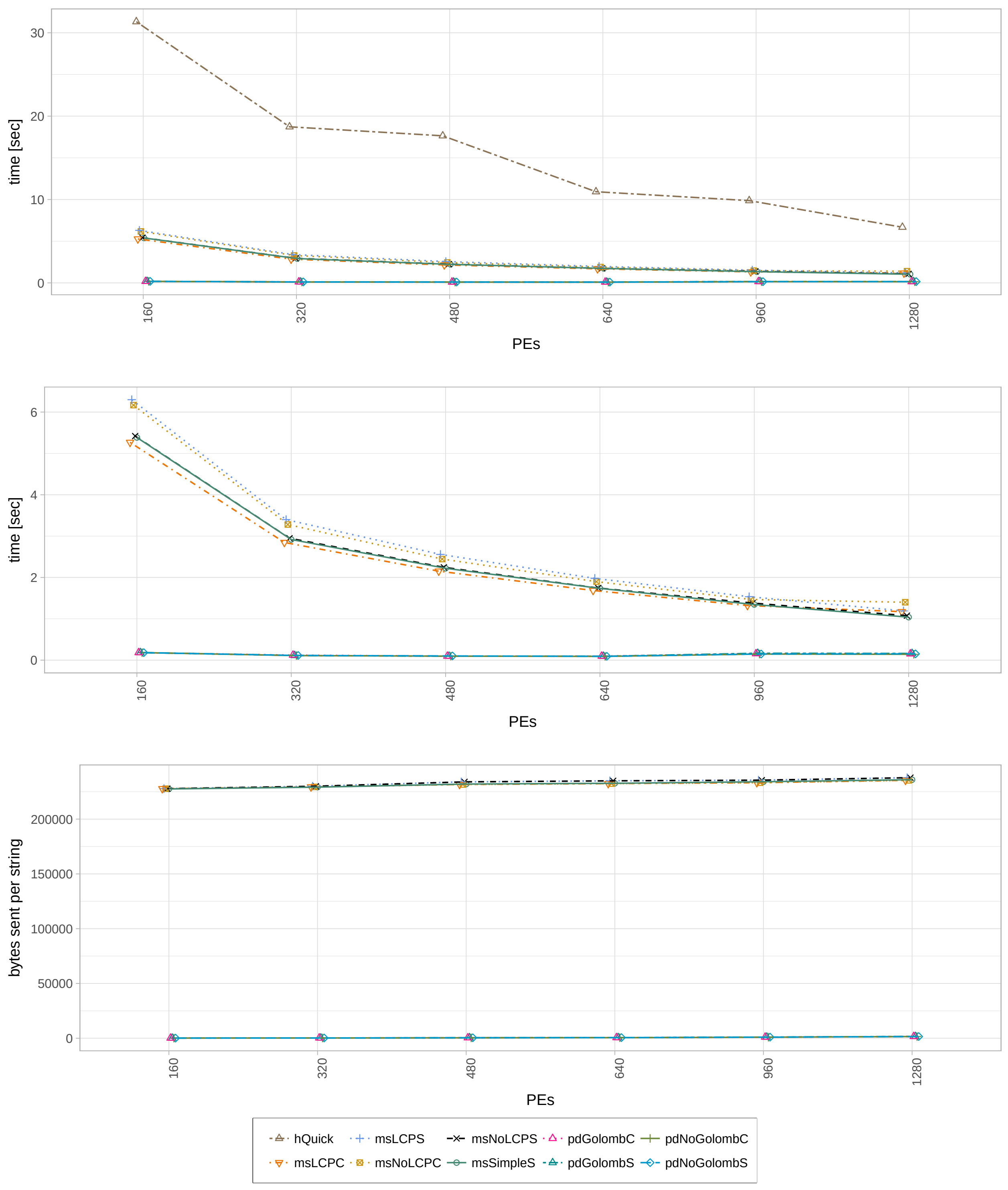


Figure 7.9: Running times and number of bytes sent per string in the strong-scaling experiment on SUFFIXES.

8. Conclusion

In this thesis we have presented two new distributed string sorting algorithms. The first algorithm (dMSS) is an extension of the – to our knowledge – only distributed string sorting algorithm introduced by Fischer and Kurpicz [11]. We optimized this algorithm by applying LCP-based techniques to compress and merge string sequences. To allow the algorithm to scale for large numbers of processors, we improved the original algorithm's partitioning step by using a distributed sorting approach.

Additionally, we have presented character-based sampling as another sampling method for the ordered partitioning algorithm. Using character-based sampling, partitions with provable upper bounds on the number of characters per set can be achieved. This can help improve the load-balancing of the entire algorithm for input data with a high variance in string length.

In the second part of this thesis we have investigated another approach to distributed string sorting resulting in our prefix-doubling algorithm dPDSS. This algorithm computes an approximate value of each string's distinguishing prefix length without communicating the entire string. Since the distinguishing prefixes are sufficient to sort a string set, only these are communicated in the algorithm.

To facilitate evaluation, we developed DNGenerator, a generator which is able to create string arrays with the (D/N)-ratio – the ratio between the sum of all distinguishing prefix lengths and the total number of characters – being an input parameter. Using this generator, we evaluated our algorithms against the algorithm by Fischer and Kurpicz and the hypercube quicksort algorithm proposed by Axtmann et al. [2], which we adapted to string sorting. Additionally, we evaluated different variants of our algorithms to be able to trace back performance gains to specific strategies and optimizations within them. The experiments were carried out on multiple data sets with differing (D/N)-ratios, including real-world data sets, and on up to 1280 processors.

Our evaluation shows that dMSS and dPDSS outperform their competitors on every data set and all tested numbers of PEs. The concrete speed-up values differ across the different evaluated data sets. However, the following general trends could be observed:

1. The dMSS algorithm with all LCP-related optimizations enabled performs best on data sets with a high (D/N)-ratio. On data sets with a very small (D/N)-ratio, however, the dMSS-variants without LCP-related optimizations or with LCP-enhanced merging only have proven to be faster.

2. The strength of dPDSS lies in sorting data sets with a low (D/N)-ratio. On data sets with a high (D/N)-ratio its performance is comparable to that of dMSS with all LCP-related optimizations turned on.

3. On skewed inputs character-based sampling helps improve load-balancing and, thus, the running time of our algorithms. However, on data with too large a number of very short duplicates it can slow down the merging step. Additionally, character-based sampling tends to draw longer strings from the input. Although with most data sets this is not a problem, this property did deteriorate the running time on one data set which contains very long duplicates.

One can conclude that exploiting the internal structure of string data sets can considerably accelerate the task of sorting them. However, we could not find *the* best algorithmic variant to do so. We feel that this is due to the multidimensionality of string arrays, which is not easy to capture with one particular strategy alone.

8.1 Future Work

Shared-Memory Parallelization: In our implementation, we solely rely on parallelization via MPI, i.e. we do not use shared-memory parallelism on the compute nodes. We believe the latter could further improve the performance of our algorithms. Especially the computation of the ordered partitions within our algorithms should benefit from this as it would reduce the required size of the local sample sets by the number of cores per compute node while keeping the same theoretical worst-case bounds on the number of strings/characters per bucket.

Further Sampling Methods: Our evaluation revealed that computing partitions such that the number of strings or characters is nearly equal across all partition-buckets is not sufficient on all data sets as this only balances the string exchange step but does not consider load-balancing for the subsequent (LCP-)merging step. Further investigation on sampling methods taking into account both aspects might be beneficial.

Methods Bounding the Splitter Length: In dMSS we presented a rather simple heuristic to bound the length of strings in the local sample sets so as to reduce the communication volume in the distributed partitioning algorithm. However, this can deteriorate the bound on the number of strings/characters per bucket arbitrarily. Applying the distinguishing-prefix-computation subroutine of dPDSS to the strings in the local sample sets in order to determine their distinguishing prefixes and then only communicating the latter could be a solution to this problem.

Bibliography

[1] Michael Axtmann and Peter Sanders. Robust massively parallel sorting. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 83–97. SIAM, 2017.

[2] Michael Axtmann, Timo Bingmann, Peter Sanders, and Christian Schulz. Practical massively parallel sorting. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 13–23. ACM, 2015.

[3] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place parallel super scalar samplesort (ipsssso). In *25th Annual European Symposium on Algorithms (ESA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[4] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369. Society for Industrial and Applied Mathematics, 1997.

[5] Timo Bingmann. *Scalable string and suffix sorting: Algorithms, techniques, and tools.* PhD thesis, Karlsruhe Institute of Technology, 2018.

[6] Timo Bingmann, Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. *Algorithmica*, 77(1):235–286, Jan 2017. ISSN 1432-0541.

[7] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[8] Guy E Blelloch, Charles E Leiserson, Bruce M Maggs, C Greg Plaxton, Stephen J Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. *Commun. ACM*, 39(12es):273–297, 1996.

[9] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems*, 8(11):1143–1156, 1997.

[11] Johannes Fischer and Florian Kurpicz. Lightweight distributed suffix array construction. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 27–38. SIAM, 2019.

[12] Message Passing Interface Forum. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015.* High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[13] Pierre Fraigniaud and Emmanuel Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics*, 53(1-3):79–133, 1994.

[14] Juha Kärkkäinen and Tommi Rantala. Engineering radix sort for strings. In *International Symposium on String Processing and Information Retrieval*, pages 3–14. Springer, 2008.

[15] Donald Knuth. Sorting and searching. *The art of computer programming*, 3:513, 1998.

[16] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, 1993.

[17] Alistair Moffat and Andrew Turpin. *Compression and coding algorithms*. Springer Science & Business Media, 2002.

[18] Waihong Ng and Katsuhiko Kakehi. Cache efficient radix sort for string sorting. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 90(2): 457–466, 2007.

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

[20] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[21] Sanjay Ranka, Ravi V Shankar, and Khaled A Alsabti. Many-to-many personalized communication with bounded traffic. In *Proceedings Frontiers' 95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 20–27. IEEE, 1995.

[22] Peter Sanders and Jesper Larsson Träff. The hierarchical factor algorithm for all-to-all communication. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, pages 799–803, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45706-0.

[23] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[24] Peter Sanders, Sebastian Schlag, and Ingo Müller. Communication efficient algorithms for fundamental big data problems. In *2013 IEEE International Conference on Big Data*, pages 15–23. IEEE, 2013.

[25] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of parallel and distributed computing*, 14(4):361–372, 1992.

[26] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

Appendix

A Running Times and Communication

B Speed-up

Table A.1: Running time in [sec] (first column) and number of bytes sent per string (second column) for all p in the DNGenerator-weak-scaling experiment.

PEs	20		40		80		160		320		640		1280	
	r = 0													
fkss	1.68	502	2.29	504	2.67	514	2.91	552	3.27	706	4.25	1321	7.51	3782
hQuick	5.95	1102	7.55	1353	10.4	1603	13.1	1854	15.1	2104	17.2	2355	20.5	2605
msLCPC	1.48	499	2.10	498	2.43	499	2.70	500	2.95	501	3.24	504	4.45	513
msLCPS	1.48	499	2.10	498	2.44	499	2.70	500	2.94	501	3.27	504	4.47	513
msNoLCPC	1.12	502	1.75	502	2.11	503	2.32	503	2.57	505	2.84	509	4.07	517
msNoLCPS	1.12	502	1.74	502	2.08	503	2.33	503	2.58	505	2.86	509	4.07	517
msSimpleS	1.09	501	1.72	501	2.07	502	2.32	502	2.56	504	2.86	508	3.98	516
pdGolombC	0.45	12	0.52	**12**	0.53	**12**	0.56	**12**	0.64	12	0.74	**12**	0.88	**12**
pdGolombS	0.46	**12**	0.52	12	0.56	12	0.58	12	0.67	**12**	0.75	12	0.90	12
pdNoGolombC	**0.43**	15	**0.50**	14	**0.51**	14	**0.54**	14	**0.62**	15	**0.70**	14	**0.86**	15
pdNoGolombS	0.44	15	0.50	14	0.54	14	0.58	14	0.62	15	0.73	14	0.88	15
	r = 0.25													
fkss	2.51	502	3.17	504	3.63	514	3.95	552	4.63	706	6.33	1321	12.0	3782
hQuick	7.24	1102	8.99	1353	12.0	1603	14.8	1854	16.8	2104	19.0	2355	22.3	2605
msLCPC	1.95	379	2.43	379	2.71	380	2.94	381	3.21	383	3.52	387	4.62	395
msLCPS	1.94	379	2.44	379	2.70	380	2.93	381	3.18	383	3.54	387	4.61	395
msNoLCPC	1.81	502	2.42	502	2.77	503	3.01	503	3.26	505	3.69	509	4.82	517
msNoLCPS	1.80	502	2.42	502	2.76	503	3.01	503	3.29	505	3.69	509	4.80	517
msSimpleS	1.97	501	2.65	501	3.05	502	3.36	502	3.73	504	4.18	508	5.32	516
pdGolombC	1.26	20	1.33	21	1.36	**21**	1.39	22	**1.47**	22	1.61	**24**	1.84	**27**
pdGolombS	1.26	**20**	1.33	**21**	1.35	21	1.40	**22**	1.48	**22**	1.62	24	1.87	27
pdNoGolombC	**1.24**	23	1.32	23	1.33	23	**1.36**	24	1.48	25	**1.58**	26	**1.82**	29
pdNoGolombS	1.26	23	**1.31**	23	**1.33**	23	1.40	24	1.48	25	1.61	26	1.86	29
	r = 0.5													
fkss	3.36	502	4.07	504	4.58	514	5.01	552	6.02	706	8.45	1321	16.5	3782
hQuick	8.62	1102	10.6	1353	13.7	1603	16.6	1854	18.7	2104	20.9	2355	24.2	2605
msLCPC	2.43	255	2.76	255	2.98	256	3.13	257	3.38	259	3.70	263	4.31	271
msLCPS	2.43	255	2.77	255	2.98	256	3.14	257	3.41	259	3.61	263	4.34	271
msNoLCPC	2.52	503	3.17	503	3.48	504	3.74	504	4.01	506	4.30	510	5.55	518
msNoLCPS	2.52	503	3.17	503	3.49	504	3.73	504	4.04	506	4.44	510	5.55	518
msSimpleS	2.90	501	3.61	501	4.06	502	4.41	502	4.86	504	5.40	508	6.66	516
pdGolombC	2.06	**25**	2.09	**25**	2.12	**25**	2.18	**26**	2.28	27	2.43	**30**	2.73	**35**
pdGolombS	2.05	25	2.10	25	2.13	25	2.20	26	2.31	**27**	2.46	30	2.75	35
pdNoGolombC	2.03	27	**2.08**	27	**2.10**	27	**2.16**	28	**2.28**	29	**2.40**	32	**2.70**	37
pdNoGolombS	**2.03**	27	2.09	27	2.13	27	2.18	28	2.30	29	2.45	32	2.74	37
	r = 0.75													
fkss	4.22	502	4.97	504	5.54	514	6.04	552	7.27	706	10.5	1321	20.8	3782
hQuick	10.0	1102	12.1	1353	15.4	1603	18.4	1854	20.6	2104	22.9	2355	26.4	2605
msLCPC	2.94	**130**	3.13	130	3.27	131	3.45	**132**	3.59	134	3.58	**138**	4.04	**147**
msLCPS	2.95	130	3.12	**130**	3.26	**131**	3.39	132	3.58	**134**	3.58	138	4.01	147
msNoLCPC	3.26	503	3.89	503	4.25	504	4.52	504	4.79	506	5.29	510	6.31	518
msNoLCPS	3.24	503	3.89	503	4.23	504	4.53	504	4.78	506	5.12	510	6.32	518
msSimpleS	3.82	501	4.58	501	5.09	502	5.52	502	6.00	504	6.64	508	7.92	516
pdGolombC	2.78	130	2.97	130	**3.08**	131	**3.25**	132	**3.36**	134	3.37	138	3.88	147
pdGolombS	2.79	130	2.97	130	3.10	131	3.30	132	3.43	134	3.41	138	3.91	147
pdNoGolombC	**2.76**	130	**2.96**	130	3.10	131	3.26	132	3.40	134	**3.36**	138	**3.85**	147
pdNoGolombS	2.78	130	2.97	130	3.10	131	3.28	132	3.43	134	3.42	138	3.89	147
	r = 1													
fkss	5.09	502	5.88	504	6.50	514	7.14	552	8.56	706	12.6	1321	25.2	3782
hQuick	11.4	1102	13.7	1353	17.0	1603	20.2	1854	22.5	2104	24.8	2355	28.2	2605
msLCPC	3.38	5	3.40	**5**	3.42	**6**	3.48	**7**	3.58	9	3.70	**13**	3.94	**22**
msLCPS	3.38	**5**	3.39	5	3.43	6	3.48	7	3.57	**9**	3.68	13	3.91	22
msNoLCPC	3.95	503	4.58	503	4.92	504	5.18	504	5.47	506	5.78	510	7.05	518
msNoLCPS	3.95	503	4.59	503	4.93	504	5.18	504	5.45	506	5.91	510	7.03	518
msSimpleS	4.72	501	5.53	501	6.07	502	6.54	502	7.13	504	7.66	508	9.12	516
pdGolombC	3.22	5	3.24	5	**3.26**	6	**3.32**	7	3.46	9	**3.55**	13	**3.80**	23
pdGolombS	3.22	5	**3.22**	5	3.27	6	3.33	7	**3.43**	9	3.59	13	3.84	23
pdNoGolombC	**3.21**	5	3.23	5	3.26	6	3.34	7	3.45	9	3.55	13	3.83	23
pdNoGolombS	3.22	5	3.24	5	3.29	6	3.32	7	3.46	9	3.60	13	3.83	23

Table A.2: Running time in [sec] (first column) and number of bytes sent per string (second column) for all p in the SkewedDNGenerator-weak-scaling experiment.

PEs	20		40		80		160		320		640		1280	
	r = 0													
fkss	3.19	802	4.39	806	6.00	821	7.28	882	7.92	1128	12.8	2111	13.8	6047
hQuick	12.6	1703	18.2	2103	26.1	2503								
msLCPC	2.29	799	3.26	798	3.81	799	4.15	801	4.46	805	4.98	813	6.62	831
msLCPS	3.25	799	4.59	798	6.42	799	7.57	800	8.22	803	8.69	808	11.0	820
msNoLCPC	1.70	802	2.67	803	3.22	803	3.58	805	3.84	808	4.33	817	6.00	835
msNoLCPS	2.32	802	3.43	802	5.00	803	6.24	804	6.76	806	7.24	812	9.50	824
msSimpleS	2.29	801	3.42	801	4.98	802	6.20	803	6.72	805	7.13	809	9.33	819
pdGolombC	0.50	**12**	0.53	12	0.58	**12**	0.61	12	0.66	12	0.77	**12**	0.94	**12**
pdGolombS	0.49	12	0.53	**12**	0.59	12	0.60	**12**	0.68	**12**	0.75	12	0.95	12
pdNoGolombC	0.46	15	0.53	14	**0.54**	14	**0.58**	14	**0.65**	15	0.75	14	**0.92**	15
pdNoGolombS	**0.46**	15	**0.51**	14	0.55	14	0.59	14	0.66	15	**0.75**	14	0.93	15
	r = 0.25													
fkss	4.06	802	5.32	806	7.01	821	8.31	882	9.24	1128	14.5	2111	18.3	6047
hQuick	14.1	1703	19.9	2103	27.9	2503								
msLCPC	2.79	679	3.69	679	4.17	680	4.51	683	4.89	687	5.32	696	7.00	716
msLCPS	3.76	679	5.00	679	6.71	680	7.91	681	8.46	684	8.96	690	11.2	704
msNoLCPC	2.39	802	3.41	803	3.92	803	4.26	805	4.63	810	5.08	819	6.79	838
msNoLCPS	3.02	802	4.14	802	5.74	803	6.92	804	7.51	807	7.97	813	10.2	825
msSimpleS	3.22	801	4.38	801	6.02	802	7.26	803	7.90	805	8.47	809	10.7	819
pdGolombC	1.32	20	1.38	21	1.39	21	1.45	**22**	1.55	22	**1.64**	**24**	1.92	**27**
pdGolombS	1.31	**20**	1.38	**21**	1.42	**21**	1.42	22	1.54	**22**	1.67	24	1.94	27
pdNoGolombC	**1.29**	23	**1.36**	23	**1.38**	23	1.42	24	**1.51**	25	1.64	26	1.89	29
pdNoGolombS	1.30	23	1.38	23	1.40	23	**1.40**	24	1.52	25	1.65	26	**1.88**	29
	r = 0.5													
fkss	5.02	802	6.27	806	8.03	821	9.40	882	10.6	1128	16.3	2111	22.9	6047
hQuick	15.5	1703	21.5	2103	29.7	2503								
msLCPC	3.38	555	4.18	555	4.67	557	4.98	559	5.38	563	5.81	573	7.36	593
msLCPS	4.33	555	5.37	555	7.02	556	8.13	557	8.73	560	9.24	567	11.2	580
msNoLCPC	3.15	803	4.12	804	4.66	804	5.00	806	5.41	811	5.85	820	7.45	839
msNoLCPS	3.77	803	4.87	803	6.43	804	7.68	805	8.23	808	8.71	814	11.0	826
msSimpleS	4.18	801	5.35	801	7.07	802	8.35	803	9.09	805	9.64	809	12.0	819
pdGolombC	2.11	25	2.16	25	2.17	25	2.26	**26**	**2.32**	27	2.53	**30**	2.79	**35**
pdGolombS	2.11	**25**	2.17	**25**	2.19	**25**	2.22	26	2.37	**27**	2.48	30	2.83	35
pdNoGolombC	**2.08**	27	2.15	27	**2.16**	27	2.23	28	2.32	29	**2.45**	32	**2.77**	37
pdNoGolombS	2.09	27	**2.15**	27	2.17	27	**2.21**	28	2.37	29	2.51	32	2.81	37
	r = 0.75													
fkss	5.83	802	7.19	806	9.01	821	10.4	882	11.9	1128	18.4	2111	27.5	6047
hQuick	17.0	1703	23.1	2103	31.6	2503								
msLCPC	3.98	430	4.68	430	5.13	432	5.44	434	5.82	438	6.23	448	7.75	468
msLCPS	4.90	430	5.79	430	7.33	431	8.42	432	8.99	435	9.39	442	11.3	455
msNoLCPC	3.90	803	4.87	804	5.43	804	5.78	806	6.15	811	6.61	820	8.27	839
msNoLCPS	4.52	803	5.65	803	7.21	804	8.44	805	9.05	808	9.51	814	11.8	826
msSimpleS	5.16	801	6.37	801	8.11	802	9.50	803	10.2	805	10.9	809	13.3	819
pdGolombC	2.92	135	3.11	135	3.27	136	3.37	137	**3.57**	139	**3.57**	144	**4.22**	**153**
pdGolombS	2.94	**135**	3.13	**135**	3.29	**136**	3.40	**137**	3.64	**139**	3.64	**144**	4.28	153
pdNoGolombC	**2.90**	135	**3.11**	136	**3.24**	136	**3.37**	137	3.58	140	3.57	144	4.22	153
pdNoGolombS	2.94	135	3.13	136	3.26	136	3.42	137	3.64	140	3.62	144	4.28	153
	r = 1													
fkss	6.73	802	8.14	806	10.0	821	11.5	882	13.1	1128	20.5	2111	32.1	6047
hQuick	18.5	1703	24.7	2103	33.2	2503								
msLCPC	4.46	305	5.09	305	5.50	307	5.80	309	6.14	313	6.55	323	8.13	343
msLCPS	5.39	305	6.13	305	7.60	306	8.62	307	9.13	311	9.61	317	11.3	331
msNoLCPC	4.63	803	5.59	804	6.15	804	6.48	806	6.84	811	7.34	820	9.02	839
msNoLCPS	5.22	803	6.31	803	7.91	804	9.16	805	9.74	808	10.2	814	12.5	826
msSimpleS	6.03	801	7.33	801	9.17	802	10.6	803	11.3	805	12.1	809	14.6	819
pdGolombC	3.40	10	3.43	10	3.50	11	3.54	12	**3.63**	14	3.78	19	**4.12**	**28**
pdGolombS	3.40	**10**	3.42	**10**	3.49	**11**	3.52	**12**	3.66	**14**	3.83	**19**	4.15	28
pdNoGolombC	**3.38**	10	**3.41**	11	3.52	11	3.54	12	3.66	15	**3.77**	19	4.13	28
pdNoGolombS	3.39	10	3.41	11	**3.49**	11	**3.51**	12	3.64	15	3.79	19	4.13	29

Table A.3: Running time in [sec] (first column) and number of bytes sent per string (second column) for all p in the DNGenerator-strong-scaling experiment.

PEs	160		320		480		640		960		1280	
	r = 0											
fkss	8.47	521	4.00	665	3.24	1054	2.99	1812	4.38	4929	4.04	11000
hQuick	32.9	1854	19.0	2104	17.2	2238	11.0	2355	10.2	2488	6.14	2605
msLCPC	6.46	499	3.61	501	2.73	504	2.22	510	1.76	525	1.73	546
msLCPS	6.47	499	3.60	501	2.72	504	2.23	510	1.77	525	1.76	547
msNoLCPC	5.55	503	3.13	504	2.43	508	1.97	513	1.63	528	1.59	550
msNoLCPS	5.53	503	3.14	504	2.48	508	1.97	513	1.62	528	1.60	550
msSimpleS	5.50	502	3.11	503	2.36	507	1.87	512	1.50	527	1.53	549
pdGolombC	1.47	**12**	0.81	12	0.55	**13**	0.43	**13**	0.38	**14**	0.38	**15**
pdGolombS	1.42	12	0.80	**12**	0.55	13	0.46	13	0.38	14	0.43	15
pdNoGolombC	**1.39**	14	0.77	15	**0.54**	15	**0.42**	15	**0.37**	16	**0.38**	17
pdNoGolombS	1.41	14	**0.76**	15	0.54	15	0.44	15	0.37	16	0.42	17
	r = 0.25											
fkss	10.5	521	5.57	665	4.70	1054	4.61	1812	6.62	4929	8.38	11000
hQuick	37.1	1854	21.2	2104	18.8	2238	12.1	2355	11.1	2488	6.78	2605
msLCPC	6.96	380	3.89	382	2.93	386	2.31	391	1.91	406	1.45	428
msLCPS	6.96	380	3.87	382	2.98	386	2.30	391	1.94	406	1.48	429
msNoLCPC	7.16	503	4.01	504	3.01	508	2.48	513	1.99	528	1.88	550
msNoLCPS	7.15	503	4.02	504	3.00	508	2.37	513	1.96	528	1.90	550
msSimpleS	7.97	502	4.54	503	3.40	507	2.73	512	2.16	527	2.05	549
pdGolombC	3.39	21	1.83	22	1.31	**23**	1.06	**25**	0.86	**31**	0.79	**38**
pdGolombS	3.38	**21**	1.84	**22**	1.32	23	1.05	25	0.86	31	0.86	39
pdNoGolombC	3.35	23	1.79	24	**1.29**	26	**1.03**	27	0.85	33	**0.79**	40
pdNoGolombS	**3.35**	23	**1.77**	24	1.30	26	1.04	27	**0.85**	33	0.89	40
	r = 0.5											
fkss	12.9	521	7.13	665	6.16	1054	6.25	1812	8.94	4929	11.6	11000
hQuick	41.4	1854	23.4	2104	20.5	2238	13.3	2355	12.0	2488	7.58	2605
msLCPC	7.45	256	4.21	258	3.06	262	2.24	267	1.93	283	1.54	305
msLCPS	7.44	256	4.13	258	3.10	262	2.27	267	1.92	283	1.55	306
msNoLCPC	8.85	504	4.89	505	3.69	509	2.99	514	2.31	529	2.21	551
msNoLCPS	8.86	504	4.90	505	3.66	509	2.99	514	2.35	529	2.21	551
msSimpleS	10.5	502	5.89	503	4.43	507	3.57	512	2.72	527	2.58	549
pdGolombC	5.40	25	2.84	27	2.00	**29**	1.65	**32**	1.29	**42**	1.15	**56**
pdGolombS	5.36	**25**	2.83	**27**	1.99	29	1.64	32	1.29	42	1.29	56
pdNoGolombC	**5.33**	28	**2.81**	29	**1.98**	31	1.64	35	1.31	44	**1.14**	57
pdNoGolombS	5.33	28	2.81	29	2.01	31	**1.62**	35	**1.29**	44	1.28	58
	r = 0.75											
fkss	14.6	521	8.73	665	7.63	1054	7.87	1812	11.2	4929	14.9	11000
hQuick	45.8	1854	25.8	2104	22.3	2238	14.6	2355	12.9	2488	8.16	2605
msLCPC	7.96	131	4.39	133	3.03	**137**	2.36	**142**	1.84	**158**	**1.58**	**181**
msLCPS	7.93	**131**	4.38	**133**	3.02	137	2.35	142	1.84	158	1.59	182
msNoLCPC	10.6	504	5.83	505	4.31	509	3.43	514	2.76	529	2.50	551
msNoLCPS	10.6	504	5.81	505	4.32	509	3.52	514	2.72	529	2.53	551
msSimpleS	13.1	502	7.33	503	5.55	507	4.47	512	3.36	527	3.05	549
pdGolombC	7.60	131	4.16	133	2.82	137	**2.22**	143	**1.80**	159	1.58	184
pdGolombS	**7.54**	131	**4.13**	133	**2.78**	137	2.24	143	1.83	160	2.48	184
pdNoGolombC	7.55	131	4.13	133	2.86	137	2.24	143	1.81	159	1.59	184
pdNoGolombS	7.54	131	4.15	133	2.79	137	2.23	143	1.83	160	2.49	184
	r = 1											
fkss	16.9	521	10.3	665	9.08	1054	9.45	1812	13.5	4929	16.9	11000
hQuick	50.3	1854	28.2	2104	24.1	2238	15.8	2355	13.9	2488	8.88	2605
msLCPC	8.16	6	4.37	**8**	3.14	**12**	2.51	**18**	1.90	**34**	1.67	**57**
msLCPS	8.18	**6**	4.40	8	3.14	12	2.50	18	1.89	34	1.66	58
msNoLCPC	12.2	504	6.67	505	4.92	509	3.92	514	3.09	529	2.79	551
msNoLCPS	12.2	504	6.68	505	4.91	509	3.91	514	3.07	529	2.83	551
msSimpleS	15.5	502	8.62	503	6.43	507	5.20	512	3.90	527	3.50	549
pdGolombC	7.86	6	4.20	8	2.99	12	2.40	18	1.90	35	**1.64**	60
pdGolombS	7.84	6	4.21	8	3.00	12	2.41	18	**1.88**	35	1.73	60
pdNoGolombC	7.85	6	4.21	8	**2.98**	12	**2.40**	18	1.90	35	1.65	60
pdNoGolombS	**7.83**	6	**4.20**	8	3.02	12	2.41	18	1.88	35	1.74	60

Table A.4: Running time in [sec] (first column) and number of bytes sent per string (second column) for all p in the SkewedDNGenerator-strong-scaling experiment.

PEs	160		320		480		640		960		1280	
	r = 0											
fkss			9.80	1062	7.34	1684	8.27	2897	6.04	7883	12.5	17604
hQuick							32.8	3705	32.6	3956	22.9	4105
msLCPC	10.0	799	5.52	803	4.17	811	4.25	821	3.17	852	2.68	896
msLCPS			10.1	802	7.30	807	6.14	814	4.86	836	4.39	867
msNoLCPC	8.52	803	4.76	807	3.62	814	3.80	824	2.88	855	2.58	899
msNoLCPS			8.35	805	6.12	810	5.21	817	4.25	839	3.90	870
msSimpleS			8.30	804	6.04	808	5.15	814	4.13	832	3.76	858
pdGolombC	1.55	12	0.83	12	0.60	**13**	0.47	**13**	0.40	**14**	0.39	**15**
pdGolombS	1.54	**12**	0.83	**12**	0.61	13	0.49	13	0.40	14	0.40	15
pdNoGolombC	**1.48**	14	0.81	15	0.62	15	**0.46**	15	0.39	16	0.40	17
pdNoGolombS	1.50	14	**0.81**	15	**0.59**	15	0.46	15	**0.39**	16	**0.39**	17
	r = 0.25											
fkss			11.4	1062	8.69	1684	9.56	2897	8.28	7883	15.6	17604
hQuick							34.3	3705	33.7	3956	23.7	4105
msLCPC	10.9	681	6.01	685	4.52	694	3.95	706	3.42	743	2.94	795
msLCPS			10.5	683	7.56	689	6.26	697	5.07	721	4.81	756
msNoLCPC	10.2	803	5.63	808	4.27	816	4.24	829	3.35	865	2.81	917
msNoLCPS			9.25	806	6.75	811	5.75	819	4.67	843	4.27	877
msSimpleS			9.79	804	7.11	808	5.94	814	4.73	832	4.27	858
pdGolombC	3.54	21	1.90	22	1.36	**23**	1.09	**25**	0.89	**30**	0.82	**37**
pdGolombS	3.55	**21**	1.89	**22**	1.35	23	1.10	25	0.89	30	0.82	37
pdNoGolombC	**3.51**	23	**1.86**	24	1.36	25	**1.06**	27	0.90	32	**0.81**	39
pdNoGolombS	3.52	23	1.87	24	**1.34**	25	1.08	27	**0.87**	32	0.82	39
	r = 0.5											
fkss			12.9	1062	10.0	1684	11.0	2897	10.6	7883	19.0	17604
hQuick							35.6	3705	34.8	3956	24.4	4105
msLCPC	11.9	557	6.54	562	4.89	570	4.00	583	3.61	620	3.02	673
msLCPS			10.8	559	7.78	565	6.48	573	5.19	598	5.01	633
msNoLCPC	12.0	804	6.57	809	4.97	817	4.81	830	3.71	866	3.09	918
msNoLCPS			10.2	807	7.40	812	6.25	820	5.06	844	4.57	878
msSimpleS			11.2	804	8.09	808	6.82	814	5.34	832	4.76	858
pdGolombC	5.61	25	2.90	27	2.07	**29**	1.66	**32**	1.31	**41**	1.19	**53**
pdGolombS	5.59	**25**	2.89	**27**	2.09	29	1.66	32	1.35	41	1.19	54
pdNoGolombC	5.56	27	2.88	29	**2.05**	31	**1.65**	34	**1.30**	43	**1.17**	55
pdNoGolombS	**5.54**	27	**2.86**	29	2.07	31	1.65	34	1.32	43	1.18	55
	r = 0.75											
fkss			14.5	1062	11.4	1684	12.4	2897	13.0	7883	22.4	17604
hQuick							36.9	3705	35.8	3956	25.2	4105
msLCPC	13.0	432	7.11	437	5.33	445	4.29	458	3.74	495	3.28	549
msLCPS			11.1	434	7.91	440	6.52	448	5.29	473	5.22	509
msNoLCPC	13.7	804	7.51	809	5.60	817	5.31	830	4.13	866	3.47	918
msNoLCPS			11.1	807	8.05	812	6.77	820	5.44	844	4.88	878
msSimpleS			12.8	804	9.09	808	7.60	814	5.96	832	5.31	858
pdGolombC	8.12	136	4.43	138	**3.06**	142	2.39	**147**	1.94	**163**	**1.70**	**185**
pdGolombS	8.14	**136**	**4.42**	**138**	3.10	**142**	**2.39**	147	1.96	163	1.71	186
pdNoGolombC	**8.10**	136	4.42	139	3.06	142	2.42	148	**1.94**	163	1.72	185
pdNoGolombS	8.14	136	4.43	139	3.10	142	2.40	148	1.96	163	1.71	186
	r = 1											
fkss			16.1	1062	13.3	1684	13.7	2897	15.3	7883	25.7	17604
hQuick							38.2	3705	36.8	3956	26.0	4105
msLCPC	13.8	307	7.49	312	5.60	320	4.54	333	3.76	371	3.76	425
msLCPS			11.3	310	8.07	315	6.64	324	5.33	349	5.30	385
msNoLCPC	15.5	804	8.40	809	6.22	817	5.84	830	4.45	866	3.91	918
msNoLCPS			12.0	807	8.70	812	7.26	820	5.81	844	5.17	878
msSimpleS			14.0	804	10.0	808	8.48	814	6.53	832	5.76	858
pdGolombC	8.58	11	4.46	13	3.19	17	**2.53**	**22**	2.03	**38**	1.80	**61**
pdGolombS	8.61	**11**	4.46	**13**	3.21	**17**	2.54	22	2.03	38	1.80	62
pdNoGolombC	8.61	11	4.47	14	**3.18**	17	2.57	23	2.02	38	**1.77**	62
pdNoGolombS	**8.57**	11	**4.45**	14	3.20	17	2.55	23	**2.02**	39	1.81	62

Table A.5: Running time in [sec] (first column) and number of bytes sent per string (second column) for all p on the real-world instances.

PEs	160		320		480		640		960		1280	
	CommonCrawl											
hQuick			35.7	167	32.9	177	29.8	187	24.8	197	22.7	206
msLCPC	20.7	22	15.5	23	14.8	24	15.1	25	15.2	27	14.9	29
msLCPS	14.4	22	9.14	23	7.39	23	6.40	23	5.42	24	4.96	24
msNoLCPC	18.6	41	14.3	41	13.2	41	14.0	42	12.2	44	11.8	47
msNoLCPS	14.9	41	9.76	41	8.10	41	7.53	41	6.15	41	5.61	41
msSimpleS	20.6	40	17.4	40	15.4	40	16.2	40	14.0	40	14.7	40
pdGolombC	22.7	21	19.8	25	21.2	30	19.5	31	23.4	35	22.7	37
pdGolombS	14.4	**20**	8.37	**21**	**5.96**	**21**	**4.85**	**22**	4.45	**22**	4.24	**23**
pdNoGolombC	22.5	24	19.7	28	21.3	33	19.5	34	23.4	38	22.7	40
pdNoGolombS	**14.2**	23	**8.28**	24	5.96	24	4.88	25	**4.43**	25	**4.20**	26
	CommonCrawlReduced											
hQuick			37.3	190	32.9	205	28.4	212	26.4	225	23.4	235
msLCPC	**11.2**	26	**7.19**	27	5.62	28	6.03	29	4.48	31	5.00	34
msLCPS	12.2	26	7.71	26	6.29	27	5.98	27	4.75	27	5.35	27
msNoLCPC	12.9	46	8.37	47	7.37	47	7.63	48	5.35	51	5.54	54
msNoLCPS	13.4	46	8.56	46	7.57	46	7.23	46	5.70	47	5.38	47
msSimpleS	21.1	45	17.3	45	15.5	45	15.8	45	14.1	46	14.6	46
pdGolombC	14.6	23	10.8	27	12.5	33	9.37	35	13.1	38	11.7	42
pdGolombS	13.2	**21**	7.71	**23**	5.49	**23**	4.46	**23**	**4.09**	**24**	4.34	**25**
pdNoGolombC	14.5	26	10.8	31	12.5	36	9.35	38	13.1	42	11.6	45
pdNoGolombS	13.1	25	7.58	26	**5.35**	26	**4.42**	27	4.12	27	**4.23**	28
	Wikipedia											
hQuick			44.6	275	43.9	294	26.7	307	27.4	326	15.9	340
msLCPC	20.6	44	16.8	45	15.2	46	15.6	47	15.1	51	15.5	55
msLCPS	16.7	44	10.7	45	8.69	45	7.13	45	6.53	46	5.73	47
msNoLCPC	17.0	66	13.2	67	11.5	67	12.1	69	11.6	71	11.2	76
msNoLCPS	14.1	66	8.89	66	7.42	66	6.03	67	5.43	67	4.77	68
msSimpleS	16.5	65	11.6	65	10.1	65	8.84	66	6.98	66	6.55	66
pdGolombC	18.5	20	15.8	20	14.5	21	14.9	22	14.9	23	15.8	25
pdGolombS	8.87	**20**	5.25	**20**	**3.63**	**21**	2.91	**21**	3.27	**22**	2.68	**24**
pdNoGolombC	18.4	22	15.7	23	14.6	24	14.9	24	14.9	26	15.7	27
pdNoGolombS	**8.80**	22	**5.22**	23	3.71	24	**2.83**	24	**3.26**	25	**2.64**	26
	WikipediaReduced											
hQuick			37.3	338	35.5	361	22.9	379	22.5	401	14.8	419
msLCPC	10.8	55	6.16	56	4.66	57	5.85	59	3.00	63	3.94	69
msLCPS	15.3	55	9.70	55	8.27	56	6.52	56	5.07	57	5.40	58
msNoLCPC	10.4	82	5.92	82	4.56	83	5.99	84	2.91	88	4.32	94
msNoLCPS	13.0	81	8.12	82	7.09	82	6.16	82	4.44	83	5.02	83
msSimpleS	15.9	80	10.8	81	9.19	81	8.67	81	6.11	81	6.49	82
pdGolombC	8.46	22	4.88	23	3.39	24	2.67	25	2.53	26	2.52	29
pdGolombS	8.08	**22**	**4.67**	**23**	3.41	**24**	**2.56**	**24**	2.43	**26**	2.43	**27**
pdNoGolombC	8.31	25	4.78	26	3.37	27	2.66	28	2.49	30	2.47	32
pdNoGolombS	**8.03**	25	4.75	26	**3.36**	27	2.60	28	**2.39**	29	**2.39**	31
	Suffixes											
hQuick	31.3	865694	18.7	983494	17.6	1045929	10.9	1103499	9.87	1165552	6.68	1228273
msLCPC	5.27	234105	2.85	235782	2.15	238034	1.68	238859	1.32	239711	1.17	241929
msLCPS	6.30	234143	3.40	236407	2.56	240481	1.98	241477	1.53	242095	1.19	244415
msNoLCPC	6.17	234101	3.28	235780	2.45	238032	1.90	238858	1.47	239710	1.40	241928
msNoLCPS	5.42	234147	2.95	236410	2.24	240482	1.74	241478	1.38	242095	1.07	244415
msSimpleS	5.39	233856	2.92	235545	2.22	238460	1.74	239169	1.34	240625	1.04	242540
pdGolombC	0.18	130	0.12	262	0.10	414	0.10	575	0.16	1020	0.15	1638
pdGolombS	0.18	100	0.11	238	**0.10**	458	0.09	620	0.17	1048	0.16	1636
pdNoGolombC	0.18	128	**0.11**	245	0.10	**387**	**0.09**	**541**	**0.15**	**979**	**0.14**	1595
pdNoGolombS	**0.18**	**97**	0.11	**221**	0.10	430	0.09	586	0.15	1007	0.15	**1593**

Table B.6: Speed-up for all p in the DNGenerator/SkewedDNGenerator-weak-scaling experiment.

	DToN								**DToNSkewed**							
	∅	20	40	80	160	320	640	1280	∅	20	40	80	160	320	640	1280
r = 0																
fkss	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
hQuick	0.3	0.3	0.3	0.3	0.2	0.2	0.2	0.4	0.2	0.3	0.2	0.2				
msLCPC	1.2	1.1	1.1	1.1	1.1	1.1	1.3	1.7	1.8	1.4	1.3	1.6	1.8	1.8	2.6	2.1
msLCPS	1.2	1.1	1.1	1.1	1.1	1.1	1.3	1.7	1.1	1.0	1.0	0.9	1.0	1.0	1.5	1.3
msNoLCPC	1.4	1.5	1.3	1.3	1.3	1.3	1.5	1.8	2.1	1.9	1.6	1.9	2.0	2.1	3.0	2.3
msNoLCPS	1.4	1.5	1.3	1.3	1.3	1.3	1.5	1.8	1.3	1.4	1.3	1.2	1.2	1.2	1.8	1.5
msSimpleS	1.4	1.5	1.3	1.3	1.3	1.3	1.5	1.9	1.4	1.4	1.3	1.2	1.2	1.2	1.8	1.5
pdGolombC	5.4	3.7	4.4	5.0	5.2	5.2	5.8	8.5	11.5	6.4	8.3	10.4	12.0	12.0	16.6	14.7
pdGolombS	5.2	3.6	4.4	4.7	5.0	4.9	5.7	8.4	11.5	6.5	8.3	10.2	12.2	11.7	17.0	14.5
pdNoGolombC	5.6	3.9	4.6	5.2	5.4	5.3	6.0	8.7	11.9	6.9	8.3	11.2	12.5	12.2	17.0	15.0
pdNoGolombS	5.4	3.9	4.6	4.9	5.0	5.3	5.8	8.6	11.8	6.9	8.6	10.9	12.3	12.0	17.2	14.9
r = 0.25																
fkss	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
hQuick	0.3	0.3	0.4	0.3	0.3	0.3	0.3	0.5	0.3	0.3	0.3	0.3				
msLCPC	1.6	1.3	1.3	1.3	1.3	1.4	1.8	2.6	2.0	1.5	1.4	1.7	1.8	1.9	2.7	2.6
msLCPS	1.6	1.3	1.3	1.3	1.3	1.5	1.8	2.6	1.2	1.1	1.1	1.0	1.0	1.1	1.6	1.6
msNoLCPC	1.6	1.4	1.3	1.3	1.3	1.4	1.7	2.5	2.1	1.7	1.6	1.8	2.0	2.0	2.9	2.7
msNoLCPS	1.6	1.4	1.3	1.3	1.3	1.4	1.7	2.5	1.4	1.3	1.3	1.2	1.2	1.2	1.8	1.8
msSimpleS	1.4	1.3	1.2	1.2	1.2	1.2	1.5	2.2	1.3	1.3	1.2	1.2	1.1	1.2	1.7	1.7
pdGolombC	3.4	2.0	2.4	2.7	2.8	3.1	3.9	6.5	6.0	3.1	3.9	5.0	5.7	6.0	8.9	9.6
pdGolombS	3.3	2.0	2.4	2.7	2.8	3.1	3.9	6.4	6.0	3.1	3.9	4.9	5.9	6.0	8.7	9.4
pdNoGolombC	3.4	2.0	2.4	2.7	2.9	3.1	4.0	6.6	6.1	3.1	3.9	5.1	5.9	6.1	8.9	9.7
pdNoGolombS	3.4	2.0	2.4	2.7	2.8	3.1	3.9	6.4	6.1	3.1	3.9	5.0	5.9	6.1	8.8	9.7
r = 0.5																
fkss	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
hQuick	0.4	0.4	0.4	0.3	0.3	0.3	0.4	0.7	0.3	0.3	0.3	0.3				
msLCPC	2.0	1.4	1.5	1.5	1.6	1.8	2.3	3.8	2.1	1.5	1.5	1.7	1.9	2.0	2.8	3.1
msLCPS	2.0	1.4	1.5	1.5	1.6	1.8	2.3	3.8	1.4	1.2	1.2	1.1	1.2	1.2	1.8	2.0
msNoLCPC	1.7	1.3	1.3	1.3	1.3	1.5	2.0	3.0	2.1	1.6	1.5	1.7	1.9	2.0	2.8	3.1
msNoLCPS	1.7	1.3	1.3	1.3	1.3	1.5	1.9	3.0	1.5	1.3	1.3	1.2	1.2	1.3	1.9	2.1
msSimpleS	1.4	1.2	1.1	1.1	1.1	1.2	1.6	2.5	1.3	1.2	1.2	1.1	1.1	1.2	1.7	1.9
pdGolombC	2.9	1.6	1.9	2.2	2.3	2.6	3.5	6.0	4.6	2.4	2.9	3.7	4.2	4.6	6.4	8.2
pdGolombS	2.9	1.6	1.9	2.2	2.3	2.6	3.4	6.0	4.6	2.4	2.9	3.7	4.2	4.5	6.6	8.1
pdNoGolombC	2.9	1.7	2.0	2.2	2.3	2.6	3.5	6.1	4.7	2.4	2.9	3.7	4.2	4.6	6.7	8.3
pdNoGolombS	2.9	1.7	1.9	2.1	2.3	2.6	3.5	6.0	4.6	2.4	2.9	3.7	4.3	4.5	6.5	8.2
r = 0.75																
fkss	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
hQuick	0.4	0.4	0.4	0.4	0.3	0.4	0.5	0.8	0.3	0.3	0.3	0.3				
msLCPC	2.4	1.4	1.6	1.7	1.8	2.0	2.9	5.2	2.2	1.5	1.5	1.8	1.9	2.0	2.9	3.5
msLCPS	2.4	1.4	1.6	1.7	1.8	2.0	2.9	5.2	1.5	1.2	1.2	1.2	1.2	1.3	2.0	2.4
msNoLCPC	1.7	1.3	1.3	1.3	1.3	1.5	2.0	3.3	2.1	1.5	1.5	1.7	1.8	1.9	2.8	3.3
msNoLCPS	1.7	1.3	1.3	1.3	1.3	1.5	2.1	3.3	1.5	1.3	1.3	1.2	1.2	1.3	1.9	2.3
msSimpleS	1.4	1.1	1.1	1.1	1.1	1.2	1.6	2.6	1.3	1.1	1.1	1.1	1.1	1.2	1.7	2.1
pdGolombC	2.5	1.5	1.7	1.8	1.9	2.2	3.1	5.4	3.6	2.0	2.3	2.8	3.1	3.3	5.1	6.5
pdGolombS	2.5	1.5	1.7	1.8	1.8	2.1	3.1	5.3	3.5	2.0	2.3	2.7	3.1	3.3	5.0	6.4
pdNoGolombC	2.5	1.5	1.7	1.8	1.9	2.1	3.1	5.4	3.6	2.0	2.3	2.8	3.1	3.3	5.1	6.5
pdNoGolombS	2.5	1.5	1.7	1.8	1.8	2.1	3.1	5.4	3.6	2.0	2.3	2.8	3.1	3.3	5.1	6.4
r = 1																
fkss	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
hQuick	0.5	0.4	0.4	0.4	0.4	0.4	0.5	0.9	0.3	0.4	0.3	0.3				
msLCPC	2.8	1.5	1.7	1.9	2.1	2.4	3.4	6.4	2.3	1.5	1.6	1.8	2.0	2.1	3.1	3.9
msLCPS	2.8	1.5	1.7	1.9	2.0	2.4	3.4	6.4	1.7	1.2	1.3	1.3	1.3	1.4	2.1	2.8
msNoLCPC	1.8	1.3	1.3	1.3	1.4	1.6	2.2	3.6	2.1	1.5	1.5	1.6	1.8	1.9	2.8	3.6
msNoLCPS	1.8	1.3	1.3	1.3	1.4	1.6	2.1	3.6	1.6	1.3	1.3	1.3	1.3	1.3	2.0	2.6
msSimpleS	1.4	1.1	1.1	1.1	1.1	1.2	1.6	2.8	1.4	1.1	1.1	1.1	1.1	1.2	1.7	2.2
pdGolombC	2.9	1.6	1.8	2.0	2.2	2.5	3.5	6.6	3.9	2.0	2.4	2.9	3.2	3.6	5.4	7.8
pdGolombS	2.9	1.6	1.8	2.0	2.1	2.5	3.5	6.6	3.9	2.0	2.4	2.9	3.3	3.6	5.3	7.7
pdNoGolombC	2.9	1.6	1.8	2.0	2.1	2.5	3.5	6.6	3.9	2.0	2.4	2.8	3.2	3.6	5.4	7.8
pdNoGolombS	2.9	1.6	1.8	2.0	2.1	2.5	3.5	6.6	3.9	2.0	2.4	2.9	3.3	3.6	5.4	7.8

Table B.7: Speed-up for all p in the DNGenerator/SkewedDNGenerator-strong-scaling experiment.

	DToN							**DToNSkewed**						
	∅	160	320	480	640	960	1280	∅	160	320	480	640	960	1280
r = 0														
fkss	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0		1.0	1.0	1.0	1.0	1.0
hQuick	0.3	0.3	0.2	0.2	0.3	0.4	0.7	0.3				0.3	0.2	0.5
msLCPC	1.6	1.3	1.1	1.2	1.3	2.5	2.3	2.4	-	1.8	1.8	1.9	1.9	4.7
msLCPS	1.6	1.3	1.1	1.2	1.3	2.5	2.3	1.5		1.0	1.0	1.3	1.2	2.9
msNoLCPC	1.8	1.5	1.3	1.3	1.5	2.7	2.5	2.6	-	2.1	2.0	2.2	2.1	4.8
msNoLCPS	1.8	1.5	1.3	1.3	1.5	2.7	2.5	1.7		1.2	1.2	1.6	1.4	3.2
msSimpleS	1.9	1.5	1.3	1.4	1.6	2.9	2.6	1.8		1.2	1.2	1.6	1.5	3.3
pdGolombC	7.6	5.8	4.9	5.9	6.9	11.4	10.7	17.4	-	11.8	12.1	17.3	15.0	30.9
pdGolombS	7.4	6.0	5.0	5.9	6.5	11.6	9.5	17.4	-	11.8	12.0	17.0	15.2	31.2
pdNoGolombC	7.8	6.1	5.2	6.1	7.0	11.9	10.7	18.0	-	12.3	12.7	17.2	15.2	32.7
pdNoGolombS	7.6	6.0	5.3	6.0	6.7	11.7	9.7	18.1	-	12.2	12.4	17.9	15.6	32.3
r = 0.25														
fkss	1.0	1.0	1.0	1.0	1.0	1.0	0.8	1.0		1.0	1.0	1.0	1.0	1.0
hQuick	0.5	0.3	0.3	0.2	0.4	0.6	1.0	0.4				0.3	0.2	0.7
msLCPC	2.4	1.5	1.4	1.6	2.0	3.5	4.7	2.8	-	1.9	1.9	2.4	2.4	5.3
msLCPS	2.4	1.5	1.4	1.6	2.0	3.4	4.6	1.7		1.1	1.1	1.5	1.6	3.2
msNoLCPC	2.2	1.5	1.4	1.6	1.9	3.3	3.6	2.9	-	2.0	2.0	2.3	2.5	5.6
msNoLCPS	2.2	1.5	1.4	1.6	1.9	3.4	3.6	1.9		1.2	1.3	1.7	1.8	3.7
msSimpleS	2.0	1.3	1.2	1.4	1.7	3.1	3.3	1.9		1.2	1.2	1.6	1.7	3.7
pdGolombC	5.1	3.1	3.0	3.6	4.4	7.7	8.5	9.9	-	6.1	6.3	8.9	9.4	19.1
pdGolombS	4.9	3.1	3.0	3.6	4.4	7.7	7.9	9.9	-	6.0	6.4	8.7	9.3	19.1
pdNoGolombC	5.1	3.1	3.1	3.6	4.5	7.8	8.6	10.0	-	6.1	6.5	8.9	9.5	18.9
pdNoGolombS	5.0	3.2	3.2	3.6	4.4	7.8	7.6	10.0	-	6.1	6.5	8.9	9.5	19.0
r = 0.5														
fkss	0.9	1.0	1.0	1.0	1.0	1.0	0.7	1.0		1.0	1.0	1.0	1.0	1.0
hQuick	0.5	0.3	0.3	0.3	0.5	0.7	1.0	0.5				0.3	0.3	0.8
msLCPC	3.0	1.7	1.7	2.0	2.8	4.6	4.9	3.2	-	2.0	2.0	2.7	2.9	6.3
msLCPS	3.0	1.7	1.7	2.0	2.8	4.7	4.9	2.0		1.2	1.3	1.7	2.0	3.8
msNoLCPC	2.3	1.5	1.5	1.7	2.1	3.9	3.4	3.1	-	2.0	2.0	2.3	2.9	6.2
msNoLCPS	2.3	1.5	1.5	1.7	2.1	3.8	3.4	2.1		1.3	1.4	1.8	2.1	4.2
msSimpleS	2.0	1.2	1.2	1.4	1.7	3.3	2.9	2.0		1.2	1.2	1.6	2.0	4.0
pdGolombC	4.2	2.4	2.5	3.1	3.8	6.9	6.6	8.0	-	4.5	4.8	6.6	7.9	16.1
pdGolombS	4.1	2.4	2.5	3.1	3.8	6.9	5.9	8.0	-	4.5	4.8	6.6	7.9	16.0
pdNoGolombC	4.2	2.4	2.5	3.1	3.8	6.8	6.7	8.1	-	4.5	4.9	6.7	8.1	16.3
pdNoGolombS	4.1	2.4	2.5	3.1	3.9	7.0	5.9	8.0	-	4.5	4.8	6.6	8.0	16.2
r = 0.75														
fkss	0.9	1.0	1.0	1.0	1.0	1.0	0.5	1.0		1.0	1.0	1.0	1.0	1.0
hQuick	0.6	0.3	0.3	0.3	0.5	0.9	1.0	0.5				0.3	0.4	0.9
msLCPC	3.5	1.8	2.0	2.5	3.3	6.1	5.2	3.5	-	2.0	2.1	2.9	3.5	6.8
msLCPS	3.5	1.8	2.0	2.5	3.3	6.1	5.1	2.3		1.3	1.4	1.9	2.4	4.3
msNoLCPC	2.4	1.4	1.5	1.8	2.3	4.1	3.3	3.2	-	1.9	2.0	2.3	3.1	6.5
msNoLCPS	2.4	1.4	1.5	1.8	2.2	4.1	3.2	2.3		1.3	1.4	1.8	2.4	4.6
msSimpleS	1.9	1.1	1.2	1.4	1.8	3.3	2.7	2.1		1.1	1.3	1.6	2.2	4.2
pdGolombC	3.6	1.9	2.1	2.7	3.5	6.2	5.2	6.4	-	3.3	3.8	5.1	6.7	13.2
pdGolombS	3.3	1.9	2.1	2.7	3.5	6.1	3.3	6.4	-	3.3	3.7	5.2	6.6	13.1
pdNoGolombC	3.6	1.9	2.1	2.7	3.5	6.2	5.1	6.4	-	3.3	3.8	5.1	6.7	13.1
pdNoGolombS	3.3	1.9	2.1	2.7	3.5	6.1	3.3	6.4	-	3.3	3.7	5.1	6.6	13.1
r = 1														
fkss	0.9	1.0	1.0	1.0	1.0	1.0	0.5	1.0		1.0	1.0	1.0	1.0	1.0
hQuick	0.6	0.3	0.4	0.4	0.6	1.0	1.0	0.6				0.4	0.4	1.0
msLCPC	3.9	2.1	2.3	2.9	3.8	7.1	5.3	3.7	-	2.1	2.4	3.0	4.1	6.8
msLCPS	3.9	2.1	2.3	2.9	3.8	7.2	5.4	2.6		1.4	1.6	2.1	2.9	4.9
msNoLCPC	2.5	1.4	1.5	1.8	2.4	4.4	3.2	3.3	-	1.9	2.1	2.4	3.4	6.6
msNoLCPS	2.5	1.4	1.5	1.8	2.4	4.4	3.1	2.5		1.3	1.5	1.9	2.6	5.0
msSimpleS	1.9	1.1	1.2	1.4	1.8	3.5	2.5	2.2		1.1	1.3	1.6	2.3	4.5
pdGolombC	4.0	2.1	2.4	3.0	3.9	7.1	5.4	7.0	-	3.6	4.2	5.5	7.5	14.3
pdGolombS	4.0	2.2	2.4	3.0	3.9	7.2	5.1	7.0	-	3.6	4.1	5.4	7.5	14.3
pdNoGolombC	4.0	2.2	2.4	3.0	3.9	7.1	5.4	7.1	-	3.6	4.2	5.4	7.6	14.4
pdNoGolombS	4.0	2.2	2.4	3.0	3.9	7.2	5.1	7.0	-	3.6	4.2	5.4	7.6	14.2

Table B.8: Speed-up for all *p* in the real-world-strong-scaling experiments.

PEs	∅	160	320	480	640	960	1280
CommonCrawl							
hQuick	1.0		1.0	1.0	1.0	1.0	1.0
msLCPC	1.9	–	2.3	2.2	2.0	1.6	1.5
msLCPS	4.4	–	3.9	4.5	4.7	4.6	4.6
msNoLCPC	2.2	–	2.5	2.5	2.1	2.0	1.9
msNoLCPS	4.0	–	3.7	4.1	4.0	4.0	4.1
msSimpleS	1.9	–	2.1	2.1	1.8	1.8	1.6
pdGolombC	1.4	–	1.8	1.6	1.5	1.1	1.0
pdGolombS	5.4	–	4.3	5.5	6.1	5.6	5.4
pdNoGolombC	1.4	–	1.8	1.5	1.5	1.1	1.0
pdNoGolombS	5.4	–	4.3	5.5	6.1	5.6	5.4
CommonCrawlReduced							
hQuick	1.0		1.0	1.0	1.0	1.0	1.0
msLCPC	5.3	–	5.2	5.9	4.7	5.9	4.7
msLCPS	5.0	–	4.8	5.2	4.8	5.6	4.4
msNoLCPC	4.4	–	4.5	4.5	3.7	4.9	4.2
msNoLCPS	4.3	–	4.4	4.3	3.9	4.6	4.4
msSimpleS	1.9	–	2.2	2.1	1.8	1.9	1.6
pdGolombC	2.6	–	3.5	2.6	3.0	2.0	2.0
pdGolombS	5.8	–	4.8	6.0	6.4	6.4	5.4
pdNoGolombC	2.6	–	3.5	2.6	3.0	2.0	2.0
pdNoGolombS	5.9	–	4.9	6.2	6.4	6.4	5.5
Wikipedia							
hQuick	1.0		1.0	1.0	1.0	1.0	1.0
msLCPC	2.0	–	2.7	2.9	1.7	1.8	1.0
msLCPS	4.0	–	4.2	5.0	3.7	4.2	2.8
msNoLCPC	2.6	–	3.4	3.8	2.2	2.4	1.4
msNoLCPS	4.7	–	5.0	5.9	4.4	5.0	3.3
msSimpleS	3.5	–	3.8	4.4	3.0	3.9	2.4
pdGolombC	2.1	–	2.8	3.0	1.8	1.8	1.0
pdGolombS	8.8	–	8.5	12.1	9.2	8.4	5.9
pdNoGolombC	2.1	–	2.8	3.0	1.8	1.8	1.0
pdNoGolombS	8.8	–	8.5	11.8	9.4	8.4	6.0
WikipediaReduced							
hQuick	1.0		1.0	1.0	1.0	1.0	1.0
msLCPC	5.8	–	6.1	7.6	3.9	7.5	3.8
msLCPS	3.8	–	3.8	4.3	3.5	4.4	2.7
msNoLCPC	5.8	–	6.3	7.8	3.8	7.7	3.4
msNoLCPS	4.3	–	4.6	5.0	3.7	5.1	3.0
msSimpleS	3.2	–	3.5	3.9	2.6	3.7	2.3
pdGolombC	8.3	–	7.7	10.5	8.6	8.9	5.9
pdGolombS	8.5	–	8.0	10.4	8.9	9.3	6.1
pdNoGolombC	8.4	–	7.8	10.5	8.6	9.0	6.0
pdNoGolombS	8.6	–	7.9	10.5	8.8	9.4	6.2
Suffixes							
hQuick	1.0	1.0	1.0	1.0	1.0	1.0	1.0
msLCPC	6.7	5.9	6.6	8.2	6.5	7.5	5.7
msLCPS	5.8	5.0	5.5	6.9	5.5	6.4	5.6
msNoLCPC	5.9	5.1	5.7	7.2	5.8	6.7	4.8
msNoLCPS	6.6	5.8	6.4	7.9	6.3	7.1	6.2
msSimpleS	6.7	5.8	6.4	7.9	6.3	7.3	6.4
pdGolombC	122.2	172.8	155.5	183.9	113.5	63.3	44.0
pdGolombS	123.8	172.5	167.9	184.8	116.7	59.5	41.5
pdNoGolombC	126.8	171.9	169.9	183.9	120.8	67.0	47.3
pdNoGolombS	124.0	173.1	166.4	174.6	119.2	66.7	44.1