LCP-Aware Parallel String Sorting

Jonas Ellert$^{1[0000-0003-3305-6185]}$, Johannes Fischer^{1}, and Nodari Sitchinava$^{\star 2}$

^{1} Technical University of Dortmund, Dortmund, Germany
jonas.ellert@tu-dortmund.de, johannes.fischer@cs.tu-dortmund.de
^{2} University of Hawaii at Manoa, Honolulu, Hawaii, USA
nodari@hawaii.edu

Abstract. When lexicographically sorting strings, it is not always necessary to inspect all symbols. For example, the lexicographical rank of `europar` amongst the strings `eureka`, `eurasia`, and `excells` only depends on its so called *relevant prefix* `euro`. The *distinguishing prefix size* D of a set of strings is the number of symbols that actually need to be inspected to establish the lexicographical ordering of all strings. Efficient string sorters should be D-aware, i.e. their complexity should depend on D rather than on the total number N of all symbols in all strings. While there are many D-aware sorters in the sequential setting, there appear to be no such results in the PRAM model. We propose a framework yielding a D-aware modification of any existing PRAM string sorter. The derived algorithms are work-optimal with respect to their original counterpart: If the original algorithm requires $\mathcal{O}(w(N))$ work, the derived one requires $\mathcal{O}(w(D))$ work. The execution time increases only by a small factor that is logarithmic in the length of the longest relevant prefix. Our framework universally works for deterministic and randomized algorithms in all variations of the PRAM model, such that future improvements in (D-unaware) parallel string sorting will directly result in improvements in D-aware parallel string sorting.

Keywords: String sorting · lexicographical sorting · parallel · PRAM · distinguishing prefix · longest common prefix · LCP · Karp-Rabin fingerprints

1 Introduction

The problem of string sorting is defined as follows: Given k strings $s_1, \ldots, s_k$ of total length $N = \sum |s_i|$ stored in RAM, and an array S of k pointers to the strings ($S[i]$ points to the memory location of s_i), compute a permutation S' of S such that S' lists the strings in lexicographical order ($S'[i]$ points to the lexicographically i-th smallest string). It is commonly known that establishing the lexicographical order on the strings does not necessarily require inspecting all N symbols. In fact, the rank of a string s_i only depends on its shortest prefix $s_i[1..\ell_i]$ that is not a prefix of any another string. The *distinguishing prefix*

* Supported by the National Science Foundation under Grants CCF-1533823 and CCF-1911245

size of the k strings is defined as $D = \sum_{i=1}^{k} \ell_i$. In simple words, an algorithm that sorts the strings only needs to *inspect* the D symbols that are part of the distinguishing prefix, while all other symbols are irrelevant for the lexicographical ordering. In this paper, we present parallel D-aware string sorting solutions. That is, the time and work complexity of the algorithms depends only on k, D, and possibly σ, but not on N. We present algorithms in the PRAM model and consider the following variations of the model (ordered from the weakest to the strongest): EREW, CREW, Common-CRCW, and Arbitrary-CRCW. Observe that algorithms designed for the weaker models can run on the stronger models within the same complexity measures.

1.1 Related Work

There is a variety of algorithms that aim to efficiently solve the problem of string sorting, most of which belong to one of two classes: The ones that are based on comparison sorting and generally allow arbitrary alphabets, and the ones that use (ideas from) integer sorting and are usually limited to alphabets of polynomial size $\sigma = N^{\mathcal{O}(1)}$.

If comparison sorting is the underlying technique, the well-known information-theoretical lower bound of $\Omega(k \lg k)$ comparisons applies, such that the fastest possible sequential algorithm cannot take fewer than $\Omega(k \lg k + D)$ operations. Ternary quicksort [2] runs in $\mathcal{O}(k \lg k + D)$ time, and thus matches this lower bound. In the Common-CRCW model, JáJá et al. [14] achieve $\mathcal{O}(k \lg k + N)$ work and $\mathcal{O}(\lg^2 k / \lg \lg k)$ time, and also provide a randomized algorithm that requires the same amount of work and $\mathcal{O}(\lg k)$ time with high probability. However, a D-aware modification of the algorithm cannot easily be derived.

In terms of alphabet-dependent sequential algorithms, we can use radix-sort-like approaches to achieve either $\mathcal{O}(N + \sigma)$ time [1, Alg. 3.2], or even $\mathcal{O}(D + \sigma)$ time [16], where σ is the number of different characters. Hagerup [11] presents an Arbitrary-CRCW algorithm that achieves $\mathcal{O}(N \lg \lg N)$ work and $\mathcal{O}(\lg N / \lg \lg N)$ time, assuming that the alphabet is polynomial in N. Alternatively, it can be implemented to run in $\mathcal{O}(N \sqrt{\lg N})$ work and $\mathcal{O}(\lg^{3/2} N \sqrt{\lg \lg N})$ time in the CREW model, or $\mathcal{O}(N \sqrt{\lg N \lg \lg N})$ work and the same time in the EREW model. Note that Hagerup's algorithm is based on an algorithm by Vaidyanathan et al. [17] that reduces each string to a single integer by repeatedly merging adjacent symbols. Due to the nature of the reduction technique, it always inspects all N symbols, and a D-aware modification cannot easily be derived.

There are practical parallel algorithms that exploit the distinguishing prefix and are fast in practice [4–6]; however, we are not aware of any algorithms with D-aware complexity bounds in the PRAM model.

1.2 Our Contributions

We present a theoretical framework that yields a D-aware version of any existing string sorting algorithm. Particularly, we derive D-aware versions of the algorithms by JáJá et al. and Hagerup that are work optimal with respect to their original counterparts: If the original algorithm requires $w(k, N, \sigma)$ work, then our modification requires $\mathcal{O}(w(k, D, \sigma))$ work. Additionally, in case of Hagerup's algorithm, we are no longer limited to polynomial alphabets. Generally, the new algorithms are only by a $(\lg d)$-factor slower than the original ones, where $d = \max\{\ell_i \mid 1 \leq i \leq k\}$ denotes the length of the longest relevant prefix.

Our framework is based on the idea of approximating the distinguishing prefix. It yields a 2-approximation of the relevant prefix lengths: For each string s_i, we determine a value $L[i] \in [\ell_i, 2\ell_i)$. In the Arbitrary-CRCW model, this takes expected optimal $\mathcal{O}(D)$ work and $\mathcal{O}(\lg d \cdot (\lg d + \lg k))$ time with high probability. In the weaker EREW model, we achieve $\mathcal{O}(k\sqrt{\lg k} \lg\lg k + D)$ work and $\mathcal{O}(\lg d \cdot (\lg d + \lg k) + \lg^{3/2} k \cdot \lg\lg k)$ time with high probability. An overview of our results is provided in Table 1.

The rest of the paper is structured as follows: In Section 2 we introduce the basic notation and definitions regarding the PRAM model and string processing. In Section 3 we explain our approximation scheme for the distinguishing prefix, which we use in Section 4 to derive deterministic D-aware string sorters. By using Karp-Rabin fingerprinting, we can also derive randomized string sorters, and achieve better complexity bounds for our approximation scheme (Section 5). We summarize our results in Section 6.

2 Preliminaries

Throughout this paper, we write $\lg x$ to denote the binary logarithm $\log_2 x$, and $[x, y]$ to denote the discrete interval $\{x, x + 1, \ldots, y\}$. Our research is situated in the PRAM model of computation, where multiple processors work on a shared memory. In each processing cycle, each processor may read from a memory cell, write to a memory cell, or perform a simple local operation (logical shifts, basic arithmetic operations etc). We consider the following variations of the PRAM model: EREW (each memory location can be read and written by at most one processor in each time step), CREW (each memory location can be read by multiple processors in each time step, and written by a single processor in each time step), and CRCW (each memory location can be read and written by multiple processors in each time step). For the CRCW model, we consider two variants: In the Common-CRCW model, multiple processors are allowed to write to the same memory location in the same time step only if all of them write the same value. In the Arbitrary-CRCW model, multiple processors are allowed to write different values to the same memory location in the same time step, and an arbitrary processor succeeds. However, the designer of an algorithm for this model may not make any assumptions as to which one it is. The time required by a PRAM algorithm is the total number of processing cycles. The *work* of a PRAM algorithm is defined as the total number of primitive operations that are performed by all processors, or (equivalently) as the running time of the algorithm when using only a single processor. One of the most fundamental operations in the PRAM model is the *all-prefix-operation*, and its specialization, the *all-prefix-sums-operation*:

Table 1. New results on D-aware parallel string sorting. The original (D-unaware) results are written in gray. Whenever the model is annotated with *w.h.p.*, the respective algorithms are successful with high probability $1 - \mathcal{O}(k^{-c})$ for an arbitrarily large constant c. We write $\hat{\mathcal{O}}(x)$ to denote *expected* complexity bounds.

a.) Results based on the sorter by Hagerup [11]:

Model	Work	Time	Theorem
Arbitrary CRCW	$\mathcal{O}(D \lg\lg \max(D,\sigma)))$ $\mathcal{O}(N \lg\lg N)$	$\lg d \cdot \mathcal{O}(\lg D/\lg\lg D + \lg\lg\sigma)$ $\mathcal{O}(\lg N/\lg\lg N)$	Theorem 2 [11] Theorem 4.4
CREW	$\mathcal{O}(D\sqrt{\lg D})$ $\mathcal{O}(N\sqrt{\lg N})$	$\lg d \cdot \mathcal{O}(\lg^{3/2} D\sqrt{\lg\lg D})$ $\mathcal{O}(\lg^{3/2} N\sqrt{\lg\lg N})$	Theorem 2 [11] Theorem 4.5
EREW	$\mathcal{O}(D\sqrt{\lg D \lg\lg D})$ $\mathcal{O}(N\sqrt{\lg N \lg\lg N})$	$\lg d \cdot \mathcal{O}(\lg^{3/2} D\sqrt{\lg\lg D})$ $\mathcal{O}(\lg^{3/2} N\sqrt{\lg\lg N})$	Theorem 2 [11] Theorem 4.5

b.) Results based on the sorter by JáJá et al. [14]:

Model	Work	Time	Theorem
Common CRCW	$\mathcal{O}(k \lg k + D)$ $\mathcal{O}(k \lg k + N)$	$\lg d \cdot \mathcal{O}(\lg^2 k/\lg\lg k)$ $\mathcal{O}(\lg^2 k/\lg\lg k)$	Theorem 3 [14] Theorem 3.1
Common CRCW w.h.p.	$\mathcal{O}(k \lg k + D)$ $\mathcal{O}(k \lg k + N)$	$\lg d \cdot \mathcal{O}(\lg k + \lg d)$ $\mathcal{O}(\lg k)$	Theorem 4 [14] Theorem 5.1

c.) General results that hold for any parallel string sorter:

Model	Work	Time	Lemma
Arbitrary CRCW w.h.p.	$\hat{\mathcal{O}}(D) + w(k, 2D, \sigma)$ $w(k, N, \sigma)$	$\lg d \cdot \mathcal{O}(\lg k + \lg d) + t(k, 2D, \sigma)$ $t(k, N, \sigma)$	Lemma 6 –
EREW w.h.p.	$\mathcal{O}(k\sqrt{\lg k}\lg\lg k + D)$ $+ w(k, 2D, \sigma)$ $w(k, N, \sigma)$	$\lg d \cdot \mathcal{O}(\lg k + \lg d) + \mathcal{O}(\lg^{3/2} k \cdot \lg\lg k)$ $+ t(k, 2D, \sigma)$ $t(k, N, \sigma)$	Lemma 7 –

Lemma 1 (All-Prefix-Operation, e.g. [7]). *Let $a_1, \ldots, a_n$ be n integers, and let $\oplus$ be a binary associative operator that can be evaluated in constant time. The sequence $a_1, (a_1 \oplus a_2), (a_1 \oplus a_2 \oplus a_3), \ldots, (a_1 \oplus \cdots \oplus a_n)$ can be computed in the EREW model in $\mathcal{O}(n)$ work, $\mathcal{O}(n)$ space and $\mathcal{O}(\lg n)$ time.*

Lemma 2 (All-Prefix-Sums, [9]). *The all-prefix-operation with addition as associative operator can be computed in the Common-CRCW model in $\mathcal{O}(n)$ work, $\mathcal{O}(n)$ space and $\mathcal{O}(\lg n/\lg\lg n)$ time.*

Next, we introduce basic string processing notations. A *string* over the *alphabet* Σ is a finite sequence of *symbols* from the set $\Sigma = \{1, \ldots, \sigma\}$. We write $|s|$ to denote the length of a string s. The x-th symbol of a string is $s[x]$, while the *substring* from the x-th to the y-th symbol is denoted as $s[x..y] = s[x]s[x+1]\ldots s[y]$. The substring $s[1..y]$ is called length-y prefix of s.

Given k strings $s_1, \ldots, s_k$, the length of the *longest common prefix* of two strings s_i, s_j is defined as $\mathit{lcp}(s_i, s_j) = \max\{\, \ell \mid s_i[1..\ell] = s_j[1..\ell] \,\}$. Let $\ell = \mathit{lcp}(s_i, s_j)$. We say that s_i is *lexicographically not larger* than s_j and write $s_i \preceq s_j$, iff either $\ell = |s_i|$, or $\ell < \min(|s_i|, |s_j|)$ and $s_i[\ell+1] < s_j[\ell+1]$. The strings are in *lexicographical order* iff we have $s_1 \preceq s_2 \preceq \ldots \preceq s_k$. The *relevant prefix length* of s_i is $\ell_i = \min(|s_i|, 1 + \max\{\, \mathit{lcp}(s_i, s_j) \mid 1 \le j \le k \wedge j \ne i \,\})$. The maximum number of characters that need to be inspected for a single string-to-string comparison is $d = \max\{\, \ell_i \mid 1 \le i \le k \,\}$. Finally, the *distinguishing prefix size* of the strings is defined as $D = \sum_{i=1}^{k} \ell_i$, which is the minimum number of characters that need to be inspected in order to lexicographically sort the strings.

Given k strings of total length N over the alphabet $[1, \sigma]$, let $f(k, N, \sigma)$ be a function indicating the resources (e.g. the time or space) needed by an algorithm to perform some task on the strings. We say that f is *resilient in* N iff multiplying N by a constant factor increases f by at most a constant factor, i.e.,

$$\forall c_1 : \exists c_2 : \forall k, N, \sigma : f(k, c_1 \cdot N, \sigma) \le c_2 \cdot f(k, N, \sigma) \tag{1}$$

(where all variables are from $\mathbb{N}^+$). This property will be useful when determining the worst-case complexity bounds of our algorithms. Note that the equation holds in the practical case where f is composed of a constant number of polynomial and polylogarithmic terms.

3 Approximating the Distinguishing Prefix

In this section, we introduce our framework for D-aware parallel string sorting. The general approach is to approximate the distinguishing prefix, resulting in an array L of size k with $L[i] \in [\ell_i, 2\ell_i)$, i.e. we obtain a 2-approximation of the relevant prefix lengths. Afterwards, we can safely prune each string s_i to its prefix $s'_i = s_i[1..L[i]]$. Clearly, the total length of the strings $s'_1, \ldots s'_k$ is less than $2D$, and for any two strings we have $s_i \prec s_j \Leftrightarrow s'_i \prec s'_j$. Therefore, we can then use any (not D-aware) string sorting algorithm to sort the strings in time and work depending solely on k, D, and σ.

Broadly speaking, the approximation scheme performs $\lceil \lg d \rceil + 1$ rounds, where in round r we identify and discard the strings s_i with $\ell_i \in (2^{r-1}, 2^r]$ (starting with round $r = 0$). More precisely, amongst all not yet discarded strings, we determine the ones whose length-2^r prefix is unique. Since any such string has not been discarded in the previous rounds, we have $\ell_i > 2^{r-1}$, while the uniqueness of the length-2^r prefix guarantees $\ell_i \le 2^r$. By assigning $L[i] \leftarrow \min(|s_i|, 2^r)$, we obtain the desired 2-approximation of ℓ_i. The algorithm terminates as soon as all strings have been discarded (and thus all relevant prefix approximations have been found).

Let us look at a single round in technical detail. Let I_r be the set of strings (or more precisely their indices) that survived until round r, and whose length

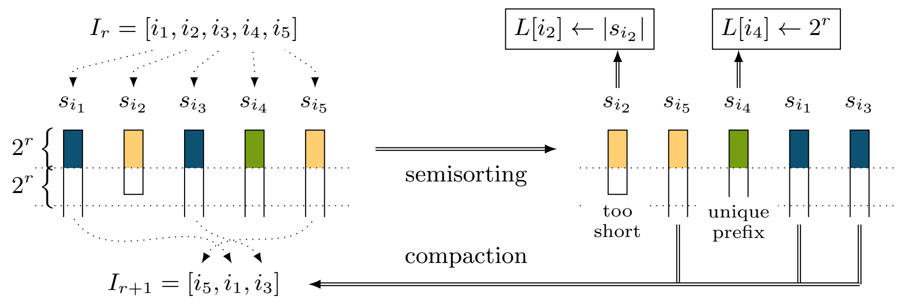

$I_r = [i_1, i_2, i_3, i_4, i_5]$
s_{i_1} s_{i_2} s_{i_3} s_{i_4} s_{i_5}
2^r
2^r
semisorting
compaction
$I_{r+1} = [i_5, i_1, i_3]$
$L[i_2] \leftarrow |s_{i_2}|$
$L[i_4] \leftarrow 2^r$
s_{i_2} s_{i_5} s_{i_4} s_{i_1} s_{i_3}
too short
unique prefix

Fig. 1. Round r of our approximation scheme. Equal colors identify equal prefixes (best viewed in color).

is at least 2^r, i.e. $I_r = \{i \in [1,k] \mid \ell_i > 2^{r-1} \wedge |s_i| \geq 2^r\}$. Initially, before round $r = 0$, we have $I_0 = \{1, \ldots, k\}$. From now on, let $k_r = |I_r|$ denote the number of strings that survived until round r. Before starting the round, we assume that I_r is given as a compact array of k_r words. Each round consists of two phases, which we explain in the following. The description is supported by Fig. 1.

Semisorting Phase. We semisort I_r using the length-2^r prefixes of the corresponding strings as keys (i.e. entry $I_r[j] = i$ is represented by the key $s_i[1..2^r]$). Semisorting is a relaxation of sorting that reorders the entries such that equal keys are contiguous, but different keys do not necessarily have to appear in correct order. In the upcoming sections, we propose different approaches for this phase.

Compaction Phase. Let I_r be semisorted as described above, and let $i = I_r[j]$ be any entry. Furthermore, let $i^- = I_r[j-1]$ and $i^+ = I_r[j+1]$ be the neighboring entries of $I_r[j]$. Due to the semisorting, the length-2^r prefix of s_i is unique iff $s_{i^-}[1..2^r] \neq s_i[1..2^r] \neq s_{i^+}[1..2^r]$. We trivially check this condition for all entries simultaneously in $\mathcal{O}(k_r \cdot 2^r)$ work and $\mathcal{O}(1)$ time in the Common-CRCW model, or in the same work and $\mathcal{O}(\lg 2^r) = \mathcal{O}(r)$ time in the EREW model (which can be easily achieved using Lemma 1). If the prefix of s_i is unique, we assign $L[i] \leftarrow 2^r$ and $I_r[j] \leftarrow 0$ (where $I_r[j] = 0$ indicates that we no longer need to consider s_i in upcoming rounds). Otherwise, we check if s_i is too short to be considered in the next round: If $|s_i| \leq 2^{r+1}$ holds, we assign $L[i] \leftarrow |s_i|$ and $I_r[j] \leftarrow 0$. Finally, we obtain I_{r+1} by moving the non-zero entries of I_r to the front of the array. This requires a single all-prefix-sums-operation [18, Section 3.1], and thus $\mathcal{O}(k_r)$ work and $\mathcal{O}(\lg k_r)$ time in the EREW model, or the same work and $\mathcal{O}(\lg k_r / \lg\lg k_r)$ time in the Common-CRCW model (Lemmas 1 and 2).

Complexity. Before discussing different approaches for the semisorting phase, we already give general bounds for the work and time complexity of our approximation scheme. For this purpose we only consider the compaction phase, which takes $\mathcal{O}(k_r \cdot 2^r)$ work in round r (regardless of the PRAM model) and thus

$\mathcal{O}(\sum_{r=0}^{\infty} k_r \cdot 2^r)$ work in total. This is asymptotically optimal:

$$\sum_{r=0}^{\lceil \lg d \rceil} k_r \cdot 2^r \;=\; \sum_{r=0}^{\lceil \lg d \rceil} \sum_{i \in I_r} 2^r \;\leq\; \sum_{i=1}^{k} \sum_{r=0}^{\lceil \lg \ell_i \rceil} 2^r \;<\; \sum_{i=1}^{k} 2^{\lceil \lg \ell_i \rceil + 1} \;\leq\; \sum_{i=1}^{k} 4\ell_i \;=\; 4D \quad (2)$$

Next, we focus on the execution time in the EREW model. The compaction phase of round r takes $\mathcal{O}(r + \lg k_r) \subseteq \mathcal{O}(\lg d + \lg k)$ time, resulting in $\mathcal{O}(\lg d \cdot (\lg d + \lg k))$ time for all rounds. In the Common-CRCW model, we have $\mathcal{O}(\lg k_r / \lg\lg k_r) \subseteq \mathcal{O}(\lg k / \lg\lg k)$ time for round r, and thus $\mathcal{O}(\lg d \cdot \lg k / \lg\lg k)$ time in total.

4 Deriving Deterministic *D*-aware String Sorters

The perhaps easiest solution for the semisorting phase is to use an existing string sorter as a subroutine, e.g. one of the algorithms that we discussed in Section 1.1. Then, after finishing the last round of our approximation scheme, we reduce the strings to their length-$L[i]$ prefixes and sort them with the same algorithm that we already used during the semisorting phase. This naturally results in a new D-aware string sorter, as visualized in Fig. 2.

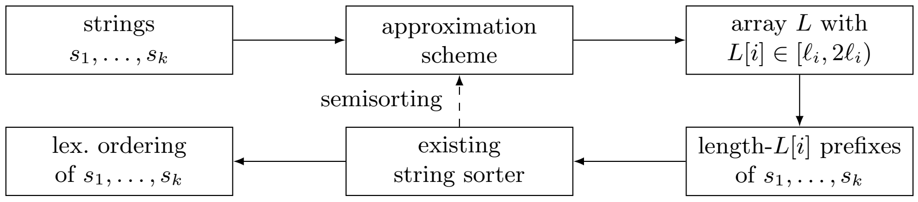
We obtain a general result for an important class of sorters: The ones that do not rely on comparison sorting and typically require $N \cdot w(k, N, \sigma)$ work and $t(k, N, \sigma)$ time for some functions w and t that are resilient in N and non-decreasing in k and N (e.g. Hagerup's algorithm [11]). Using such an algorithm, the semisorting phase of round r takes $(k_r \cdot 2^r) \cdot w(k_r, k_r \cdot 2^r, \sigma)$ work. Summing up all rounds, the total work for semisorting is $\mathcal{O}(D \cdot w(k, D, \sigma))$:

$$\sum_{r=0}^{\lceil \lg d \rceil} (k_r \cdot 2^r) \cdot w(k_r, k_r \cdot 2^r, \sigma) \;\leq\; \sum_{r=0}^{\lceil \lg d \rceil} (k_r \cdot 2^r) \cdot w(k, 2D, \sigma) \;<\; 4D \cdot w(k, 2D, \sigma) \quad (3)$$

The first inequality holds because w is non-decreasing in k and N, while the second one holds due to Eq. (2). We have $w(k, 2D, \sigma) = \mathcal{O}(w(k, D, \sigma))$ because w is resilient in N. For the same reason, the time for the semisorting phase of round r is $t(k_r, k_r \cdot 2^r, \sigma) \leq t(k, 2D, \sigma) = \mathcal{O}(t(k, D, \sigma))$. Combined with the bounds from Section 3 we have:

Theorem 1. *Let $N \cdot w(k, N, \sigma)$ and $t(k, N, \sigma)$ be the work and time needed by some algorithm to sort k strings of total length N over the alphabet $[1, \sigma]$ (for arbitrarily large σ), with w and t resilient in N and non-decreasing in k and N. Let D be the distinguishing prefix size. Then we can sort the strings in $\mathcal{O}(D \cdot w(k, D, \sigma))$ work and $\mathcal{O}(\lg d \cdot (\lg d + \lg k + t(k, D, \sigma)))$ time. The PRAM model matches the one of the string sorter. If the model is at least as strong as the Common-CRCW model, the time decreases to $\mathcal{O}(\lg d \cdot (\lg k / \lg\lg k + t(k, D, \sigma)))$.*

Note that the theorem requires a string sorter that allows *arbitrary* alphabets. This is due to the fact that (even after the first round) the number k_r of remaining strings can become arbitrarily small. Consequently, the alphabet size might become arbitrarily large compared to the total length $k_r \cdot 2^r$ of the strings that we have to semisort in round r.



strings $s_1, \ldots, s_k$
approximation scheme
array L with $L[i] \in [\ell_i, 2\ell_i)$
semisorting
lex. ordering of $s_1, \ldots, s_k$
existing string sorter
length-$L[i]$ prefixes of $s_1, \ldots, s_k$

Fig. 2. Deriving D-aware string sorters from existing D-unaware solutions.

Dealing With Large Alphabets. In theory, Theorem 1 directly implies new D-aware string sorters. However, while the theorem applies to sorters for arbitrary alphabets, many of the existing string sorting algorithms are restricted to polynomial alphabets (i.e. $\sigma = N^{\mathcal{O}(1)}$). In the remainder of this section, we show that even such alphabet restricted sorters work with Theorem 1, if we equip them with an additional preprocessing routine. We demonstrate the technique using Hagerup's algorithm [11] as an example. It will be easy to see that it would just as well work with any other string sorter. Recall Hagerup's original result:

Lemma 3 (Hagerup [11], Theorems 4.4 and 4.5). *A set of strings of total length N over the alphabet $[1, N^{\mathcal{O}(1)}]$ can be sorted in $\mathcal{O}(\lg N / \lg\lg N)$ time and $\mathcal{O}(N \lg\lg N)$ work in the CRCW model, or in $\mathcal{O}(N\sqrt{\lg N})$ work and $\mathcal{O}(\lg^{3/2} N \sqrt{\lg\lg N})$ time in the CREW model, or in $\mathcal{O}(N\sqrt{\lg N \lg\lg N})$ work and $\mathcal{O}(\lg^{3/2} N \sqrt{\lg\lg N})$ time in the EREW model.*

Remark: Hagerup does not explicitly state which variant of the CRCW model is used. However, the algorithm relies on a padded integer sorting subroutine that requires the Arbitrary-CRCW model [12]. It appears that all other operations performed by the algorithm require at most the Arbitrary-CRCW model as well.

In order to generalize Lemma 3 to arbitrary alphabets $[1, \sigma]$ with $\sigma \notin N^{\mathcal{O}(1)}$, we perform a preprocessing that reduces the alphabet to $[1, N]$ in an order preserving manner. The general idea is to use an integer sorter to sort the symbols that actually occur in any of the strings. Then, we can simply replace each symbol with its rank amongst the sorted symbols. A similar reduction technique has previously been used by Hagerup [11, p. 389] (but for a different purpose). For now, we only consider the Arbitrary-CRCW model.

First, we create N tuples of the form $\langle i, j, c \rangle$, where c is the j-th symbol of s_i. Initially, the tuples are ordered by their first and second component, i.e. $\langle 1, 1, \cdot \rangle \ldots \langle 1, |s_1|, \cdot \rangle \langle 2, 1, \cdot \rangle \ldots \langle 2, |s_2|, \cdot \rangle \ldots \langle k, 1, \cdot \rangle \ldots \langle k, |s_k|, \cdot \rangle$. In order to store this sequence in a consecutive memory area, we have to determine the position of each tuple within the sequence. Using the all-prefix-sums-operation, we can trivially realize this step in $\mathcal{O}(N)$ work and $\mathcal{O}(\lg N / \lg\lg N)$ time due to Lemma 2. Then, we use the integer sorting algorithm by Bhatt et al. [3] to sort the tuples by their third component, which takes $\mathcal{O}(N \lg\lg \sigma)$ work and $\mathcal{O}(\lg N / \lg\lg N + \lg\lg \sigma)$ time. Let $\langle i_1, j_1, c_1 \rangle \ldots \langle i_N, j_N, c_N \rangle$ be the sorted sequence of tuples. In an array $A \in \{0, 1\}^N$, we mark the (in terms of the sequence) leftmost occurrence of each character, i.e. $\forall h \in [2, N] : c_{h-1} \neq c_h \Leftrightarrow A[h] = 1$. Next, we replace A with its prefix-sums, once again taking $\mathcal{O}(N)$ work and $\mathcal{O}(\lg N / \lg \lg N)$ time due to Lemma 2. Now each entry $A[h]$ contains exactly the rank of the symbol c_h amongst all symbols. Finally, for each $h \in [1, N]$, we replace the j_h-th symbol of the i_h-th string with $A[h] + 1$. Since this reduces the alphabet to (a subset of) $[1, N]$ in an order preserving manner, we can sort the strings using Lemma 3.

In the weaker CREW and EREW models we use the same technique, but replace the algorithm by Bhatt et al. with Han and Shen's integer sorter in the EREW model [13, Theorem 4.1], which sorts the N tuples in $\mathcal{O}(N\sqrt{\lg N})$ work and $\mathcal{O}(\lg^{3/2} N)$ time. We have shown:

Corollary 1. *A set of strings of total length N over the alphabet $[1, \sigma]$ can be sorted in $\mathcal{O}(\lg N / \lg \lg N + \lg \lg \sigma)$ time and $\mathcal{O}(N \lg \lg N + N \lg \lg \sigma)$ work in the Arbitrary-CRCW model, or in $\mathcal{O}(N\sqrt{\lg N})$ work and $\mathcal{O}(\lg^{3/2} N \sqrt{\lg \lg N})$ time in the CREW model, or in $\mathcal{O}(N\sqrt{\lg N \lg \lg N})$ work and $\mathcal{O}(\lg^{3/2} N \sqrt{\lg \lg N})$ time in the EREW model.*

Theorem 2. *A set of k strings over the alphabet $[1, \sigma]$ with distinguishing prefix size D and longest relevant prefix of length d can be sorted in the work and time stated in Table 1(a).*

The theorem follows from Corollary 1 and Theorem 1. Note that the work and time in the Arbitrary-CRCW model are $\mathcal{O}(D \lg \lg D)$ and $\mathcal{O}(\lg d \cdot \lg D / \lg \lg D)$, respectively, if the alphabet is quasipolynomial in the distinguishing prefix size, i.e. $\sigma = D^{(\lg^{\mathcal{O}(1)} D)}$.

4.1 Deriving Comparison-Based Sorters

As mentioned earlier, any comparison-based string sorter requires $\Omega(k \lg k + D)$ work. In this section, we take the $\mathcal{O}(k \lg k + N)$ work algorithm by JáJá et al. [14], and derive an $\mathcal{O}(k \lg k + D)$ work modification, thus matching the lower bound. Assuming that we use the $\mathcal{O}(k \lg k + N)$ work algorithm to realize the semisorting phase of our approximation scheme, the work for semisorting in round r becomes $\mathcal{O}(k_r \lg k_r + k_r \cdot 2^r)$. After the $\lceil \lg \lg k \rceil$-th round, the $k_r \cdot 2^r$ term dominates the $k_r \lg k_r$ term. Therefore, the total work for semisorting is:

$$\mathcal{O}\left(\sum_{r=0}^{\lceil \lg d \rceil} k_r \lg k_r + k_r \cdot 2^r\right) = \mathcal{O}\left(\sum_{r=0}^{\lceil \lg \lg k \rceil - 1} k_r \lg k_r + \sum_{r=0}^{\lceil \lg d \rceil} k_r \cdot 2^r\right) \tag{4}$$

Following Eq. (2), the second sum on the right-hand side of the equation is bounded by $\mathcal{O}(D)$. Unfortunately, there appears to be no such upper bound for the first sum. Therefore, we relax our approximation scheme by simply skipping the initial $\lceil \lg \lg k \rceil$ rounds. This way, the first round that we actually perform is round $r = \lceil \lg \lg k \rceil$, during which we consider prefixes of length $2^{\lceil \lg \lg k \rceil} < 2 \lg k$. Note that consequently we may overestimate the length of relevant prefixes by

$2\lg k$ additional symbols, i.e. we obtain $L'[i] \in [\ell_i, 2 \cdot \max(\lg k, \ell_i))$. Thus, when truncating each string to its prefix $s_i[1..L'[i]]$, the total length of the strings is

$$D' := \sum_{i=1}^{k} L'[i] < 2\sum_{i=1}^{k}(\lg k + \ell_i) = 2k\lg k + 2D. \tag{5}$$

Therefore, after computing L', we can use the algorithm by JáJá et al. once more to sort the truncated strings in optimal $\mathcal{O}(k\lg k + D') \subseteq \mathcal{O}(k\lg k + D)$ work. The semisorting in round r takes $\mathcal{O}(\lg^2 k_r/\lg\lg k_r) \subseteq \mathcal{O}(\lg^2 k/\lg\lg k)$ time, and there are $\lceil\lg d\rceil - \lceil\lg\lg k\rceil = \mathcal{O}(\lg d)$ rounds. Together with the bounds from Section 3 we have:

Theorem 3. *A set of k strings with distinguishing prefix size D and longest relevant prefix of length d can be sorted in the Common-CRCW model in $\mathcal{O}(k\lg k + D)$ work and $\mathcal{O}(\lg d \cdot \lg^2 k/\lg\lg k)$ time.*

Note that we cannot trivially use our approximation scheme to derive a D-aware modification of the randomized string sorter by JáJá et al. [14], which sorts k strings of total length N in $\mathcal{O}(k\lg k + N)$ work and $\mathcal{O}(\lg k)$ time with high probability, i.e. with probability $1-(1/k)^c$ for any constant $c > 0$. If we were using this algorithm for the semisorting phase, then the probability of successfully sorting the remaining strings in round r would be $1-(1/k_r)^c$. However, even after the first round, k_r can become arbitrarily small, resulting in a low probability of success. The randomized string semisorters from the next section will allow us to circumvent this problem.

5 Randomized String Semisorting

In this section, we equip our approximation scheme with randomized string semisorters that are based on Karp-Rabin fingerprints [15]. The goal of these fingerprints is to hash substrings to small integers, which allows fast equality testing. Consider the semisorting phase of round r, during which we have to semisort k_r string prefixes of length 2^r each. Instead of directly semisorting the prefixes, we first compute a fingerprint as a representative for each prefix, and then semisort the fingerprints. This way, we can use less complex integer sorting algorithms as a subroutine. Before going into detail, we show how to efficiently compute fingerprints in the EREW model.

In order to define Karp-Rabin fingerprints, we use a prime number $q = \Theta(N^c)$ for some constant $c > 1$, and a value $b \in [q, 2q)$ chosen uniformly at random. The Karp-Rabin fingerprint $\phi_i(x, y)$ of the substring $s_i[x..y]$ is defined as follows:

$$\phi_i(x,y) = \sum_{z=x}^{y} s_i[z] \cdot b^{y-z} \mod q \tag{6}$$

Observe that equal substrings have equal fingerprints, i.e. for every integer $n \geq 0$ it holds $s_i[x..x+n] = s_j[y..y+n] \implies \phi_i(x, x+n) = \phi_j(y, y+n)$.

On the other hand, if two substrings are not equal, their fingerprints will be different with high probability. In particular, if $s_i[x..x+n] \neq s_j[y..y+n]$ then $\mathsf{Prob}[\phi_i(x, x+n) = \phi_j(y, y+n)] \leq \frac{n+1}{q} = \mathcal{O}(N^{1-c})$. Thus, by choosing a large enough constant $c > 1$, we can control the probability of false positives when comparing fingerprints instead of substrings. Using the all-prefix-operation, Karp-Rabin fingerprints can be computed efficiently in parallel:

Lemma 4. *For every ℓ-character substring $s_i[x..x+\ell-1]$, the Karp-Rabin fingerprint $\phi_i(x, x+\ell-1)$ can be computed in $\mathcal{O}(\ell)$ work, $O(\ell)$ space, and $\mathcal{O}(\lg \ell)$ time in the EREW model.*

Proof. First, we compute the sequence of exponents $b^0, b^1, \ldots, b^{\ell-1} \pmod q$ using the all-prefix-operation with multiplication over $\mathbb{Z}_q$ as the associative operator. Then, we simultaneously compute all values $f_0, \ldots, f_{\ell-1}$ with $f_j = s_i[x+j] \cdot b^{\ell-j-1} \pmod q$ in constant time. Finally, the Karp-Rabin fingerprint $\phi_i(x, x+\ell-1)$ is the sum of all the f_j over $\mathbb{Z}_q$, which can be computed via another all-prefix-operation. The stated complexity bounds follow from Lemma 1. □

During round r of our approximation scheme, we can simultaneously compute the fingerprints of all length-2^r prefixes, which takes $\mathcal{O}(k_r \cdot 2^r)$ work and $\mathcal{O}(r) \subseteq \mathcal{O}(\lg d)$ time. It remains to be shown how to semisort the fingerprints. For now, similarly to Section 4.1, we skip the first $\lceil \lg \lg k \rceil$ rounds. In the remaining rounds, we use Cole's parallel merge sort [8], which sorts the k_r fingerprints in round r in $\mathcal{O}(k_r \lg k_r) \subseteq \mathcal{O}(k_r \cdot 2^r)$ work and $\mathcal{O}(\lg k_r)$ time. This results in the following complexity bounds:

Lemma 5. *For any constant $c > 0$, the array L' with $L'[i] \in [\ell_i, 2 \cdot \max(\lg k, \ell_i))$ can be computed in the EREW model in $\mathcal{O}(D)$ work and $\mathcal{O}(\lg d \cdot (\lg d + \lg k))$ time w.h.p. $1 - (1/N)^c$.*

Now we can already derive a D-aware modification of the randomized string sorter by JáJá et al [14]. Just as in Section 4.1, we simply compute L' (using Lemma 5), and then run the original string sorter. It follows:

Theorem 4. *For any constant $c > 0$, a set of k strings with distinguishing prefix size D and longest relevant prefix of length d can be sorted in the Common-CRCW model in $\mathcal{O}(k \lg k + D)$ work and $\mathcal{O}(\lg d \cdot (\lg d + \lg k))$ time w.h.p. $1 - (1/k)^c$.*

5.1 Handling the Initial $\lceil \lg \lg k \rceil$ Rounds

Finally, we show how to (semi-)sort the fingerprints in the first $\lceil \lg \lg k \rceil$ rounds. Ideally, we would like to use the randomized semisorter by Gu et al. [10], which sorts k_r fingerprints in the Arbitrary-CRCW model in expected optimal $\mathcal{O}(k_r)$ work and $\mathcal{O}(\lg k_r)$ time with high probability $1 - (1/k_r)^c$. However, as in the previous section, k_r and thus the probability of success can become arbitrarily small. Therefore, we only use the semisorter by Gu et al. in rounds when $k_r > k/\lg^2 k$ (resulting in $\mathcal{O}(k_r)$ work), and Cole's mergesort, otherwise (resulting in

$\mathcal{O}(k/\lg k)$ work). This way, in every round the expected work for semisorting fingerprints is $\mathcal{O}(k_r + k/\lg k)$, the time is $\mathcal{O}(\lg k)$, and the probability of success is at least $1 - (\lg^2 k/k)^c > 1 - (1/k)^{(c/2)}$. Summing up the expected work for semisorting during the first $\lceil \lg \lg k \rceil$ rounds, we have:

$$\sum_{r=1}^{\lceil \lg \lg k \rceil} k_r + \sum_{r=1}^{\lceil \lg \lg k \rceil} k/\lg k = \sum_{r=1}^{\lceil \lg \lg k \rceil} k_r + o(k) = \mathcal{O}(D).$$

Together with the bounds for computing fingerprints (see Section 5) and for the compaction phase (see Section 3), we get:

Lemma 6. *For any constant $c > 0$, the array L with $L[i] \in [\ell_i, 2\ell_i)$ can be computed in the Arbitrary-CRCW model in expected optimal $\mathcal{O}(D)$ work and $\mathcal{O}(\lg d \cdot (\lg d + \lg k))$ time w.h.p. $1 - (1/k)^c$.*

In the weaker EREW model, we can replace the semisorter by Gu et al. with the deterministic integer sorter by Han and Shen [13] that we already used in the proof of Corollary 1. This results in the following bounds:

Lemma 7. *For any constant $c > 0$, the array L with $L[i] \in [\ell_i, 2\ell_i)$ can be computed in the EREW model in $\mathcal{O}(k\sqrt{\lg k} \lg \lg k + D)$ work and $\mathcal{O}(\lg d \cdot (\lg d + \lg k) + \lg^{3/2} k \cdot \lg \lg k)$ time w.h.p. $1 - (1/N)^c$.*

Note that the probability of success is $1 - (1/N)^c$ (rather than $1 - (1/k)^c$ as in Lemma 6) because we no longer use a probabilistic semisorter, and errors can only occur due to fingerprint collisions.

Lemmas 6 and 7 directly imply the results stated in Table 1(c).

6 Conclusion and Open Questions

We presented a theoretical framework that approximates the distinguishing prefix, resulting in the first D-aware string sorters in the PRAM model. It remains an open question, if the $\lg d$ time factor can be avoided without increasing the work. Generally, it is unknown if a constant approximation of the distinguishing prefix can be computed *deterministically* in optimal $\mathcal{O}(D)$ work and reasonable time.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, 1 edn. (1974)
2. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: Proceedings of the 8th Annual Symposium on Discrete Algorithms. pp. 360–369. SODA 1997, New Orleans, LA, USA (Jan 1997)
3. Bhatt, P., Diks, K., Hagerup, T., Prasad, V., Radzik, T., Saxena, S.: Improved deterministic parallel integer sorting. Information and Computation **94**(1), 29–47 (1991). https://doi.org/10.1016/0890-5401(91)90031-V

4. Bingmann, T., Eberle, A., Sanders, P.: Engineering parallel string sorting. Algorithmica **77**(1), 235–286 (2017). https://doi.org/10.1007/s00453-015-0071-1
5. Bingmann, T., Sanders, P.: Parallel string sample sort. In: Proceedings of the 21st Annual European Symposium on Algorithms. pp. 169–180. ESA 2013, Sophia Antipolis, France (Sep 2013). https://doi.org/10.1007/978-3-642-40450-4_15
6. Bingmann, T., Sanders, P., Schimek, M.: Communication-efficient string sorting. In: Proceedings of the 34th International Parallel and Distributed Processing Symposium (IPDPS 2020). pp. 137–147. New Orleans, LA, USA (May 2020). https://doi.org/10.1109/IPDPS47924.2020.00024
7. Blelloch, G.E.: Prefix sums and their applications. In: Reif, J.H. (ed.) Synthesis of Parallel Algorithms, chap. 1, pp. 35–60. Morgan Kaufmann Publishers Inc., 1st edn. (1993)
8. Cole, R.: Parallel merge sort. SIAM Journal on Computing **17**(4), 770–785 (1988). https://doi.org/10.1137/0217049
9. Cole, R., Vishkin, U.: Faster optimal parallel prefix sums and list ranking. Information and Computation **81**(3), 334–352 (1989). https://doi.org/10.1016/0890-5401(89)90036-9
10. Gu, Y., Shun, J., Sun, Y., Blelloch, G.E.: A top-down parallel semisort. In: Proceedings of the 27th Symposium on Parallelism in Algorithms and Architectures. pp. 24–34. SPAA 2015, Portland, OR, USA (Jun 2015). https://doi.org/10.1145/2755573.2755597
11. Hagerup, T.: Optimal parallel string algorithms: sorting, merging and computing the minimum. In: Proceedings of the 26th Annual Symposium on Theory of Computing. pp. 382–391. STOC 1994, Montréal, Canada (May 1994). https://doi.org/10.1145/195058.195202
12. Hagerup, T., Raman, R.: Fast deterministic approximate and exact parallel sorting. In: Proceedings of the 5th Annual Symposium on Parallel Algorithms and Architectures. pp. 346–355. SPAA 1993, Velen, Germany (Jun 1993). https://doi.org/10.1145/165231.157380
13. Han, Y., Shen, X.: Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs. SIAM Journal on Computing **31**(6), 1852–1878 (2002). https://doi.org/10.1137/S0097539799352449
14. JáJá, J., Ryu, K.W., Vishkin, U.: Sorting strings and constructing digital search trees in parallel. Theoretical Computer Science **154**(2), 225–245 (1996). https://doi.org/10.1016/0304-3975(94)00263-0
15. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development **31**(2), 249–260 (1987). https://doi.org/10.1147/rd.312.0249
16. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing **16**(6), 973–989 (1987). https://doi.org/10.1137/0216062
17. Vaidyanathan, R., Hartmann, C.R., Varshney, P.: Optimal parallel lexicographic sorting using a fine-grained decomposition. Tech. Rep. 127, Electrical Engineering and Computer Science, Syracuse University, NY, USA (1991), https://surface.syr.edu/eecs_techreports/127
18. Vishkin, U.: Thinking in parallel: Some basic data-parallel algorithms and techniques. Lecture notes at the University of Maryland Institute for Advanced Computer Studies (UMIACS) (Oct 2010), http://users.umiacs.umd.edu/~vishkin/PUBLICATIONS/classnotes.pdf