Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Engineering External Memory Induced Suffix Sorting*

Juha Kärkkäinen† Dominik Kempa† Simon J. Puglisi† Bella Zhukova†

Abstract

Suffix sorting — determining the lexicographical order of all the suffixes of a string — is one of the most important problems in string processing. The resulting data structure is called the suffix array (SA) and underpins dozens of applications in bioinformatics, data compression, and information retrieval. When the size of the input string or the SA exceeds that of internal memory (RAM), an external memory (EM) suffix sorting algorithm must be used. The most scalable of these EM methods is due to Bingmann et al. (Proc. ALENEX 2013), and is essentially a careful disk-based implementation of the so-called induced sorting technique used by the fastest RAM suffix sorting algorithms.

In this paper we show how to greatly improve the efficiency of induced suffix sorting in external memory via a non-trivial reorganization of the computation involved. Our experiments show this new approach to be twice as fast as state-of-the-art methods, while, just as significantly, using a third of the disk memory. We also demonstrate the efficacy of our implementation for handling strings on large alphabets (with many millions of distinct symbols), which is important, e.g., for applications in natural language processing and information retrieval, but unaddressed by previous EM suffix sorting implementations.

Our implementation uses a (EM) radix heap data structure and, as a side result of independent interest, we introduce a new operation for radix heaps and other monotone priority queues called min-comp, which we believe to be useful for many other applications, including discrete event simulation and sweep line algorithms, even in internal memory.

1 Introduction

Suffix sorting — determining the lexicographical order of all the suffixes of a string — is one of the most important problems in string processing. The resulting data structure is called the suffix array (SA) and underpins dozens of applications in bioinformatics, data compression, and information retrieval. Suffix arrays have become central to modern genomics, where they are used for genome assembly and short read alignment, data intensive tasks at the forefront of modern medical and evolutionary biology [17].

Suffix sorting for the case where the input (string) and output (SA) both fit in internal memory (RAM), is now quite well understood. The many algorithms for this scenario are surveyed in [20], and the fastest of these — both in theory and in practice — use a technique called induced sorting [11, 14, 19], in which the lexicographic order of the majority of suffixes is derived (induced) from the order of a special subset of suffixes.

When the size of the input string or the SA exceeds that of RAM, an external memory (EM) suffix sorting algorithm must be used. Two main approaches to EM suffix sorting define the current state of the art. The most scalable of these is the so-called eSAIS algorithm, due to Bingmann et al. [4], which is essentially a careful disk-based implementation of the above mentioned induced sorting technique used by the fastest RAM-based algorithms. A serious limiting factor of eSAIS is high disk usage: about 24 times the size of the input. Another disk-based implementation of induced sorting DSA-IS [18] is slower than eSAIS and uses even more disk space in our experiments.

The second main approach to EM suffix sorting is a family of scan-based algorithms [9, 12, 13] that reduce disk usage at some cost to runtime, asymptotically at least. The basic idea of these methods is to divide the text into blocks, construct suffix arrays for the blocks and then merge these partial suffix arrays. The fastest member of this family, called pSAscan [13], has disk usage of about eight times the size of the input — about a third that of eSAIS — and is faster than eSAIS when the ratio of input size to RAM size is reasonable. However, pSAscan slows down rapidly as the ratio grows larger, which makes it ultimately less scalable than eSAIS.

1.1 Results.

The main results in this paper are as follows.

1. We show how to significantly improve the efficiency of induced suffix sorting in external memory via a non-trivial reorganization of the computation

*This research is partially supported by Academy of Finland through grant 294143.

†Helsinki Institute for Information Technology (HIIT), Department of Computer Science, University of Helsinki, Finland.

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

involved. Our experiments show this new approach to be twice as fast as eSAIS, the best previous realisation of EM induced sorting, while, just as significantly, using a third of the disk memory. This level of disk usage brings our implementation in line with pSAscan, and is close to optimal.

2. We also demonstrate the efficacy of our implementation for handling strings on large alphabets (with many millions of distinct symbols), which is important, e.g., for applications in natural language processing [21] and information retrieval [10]. Efficiency for large alphabets has been to date unaddressed by previous studies on EM suffix sorting [6, 7, 4, 12, 18, 13], in all of which a byte alphabet is assumed. While eSAIS could probably be modified for large alphabets fairly easily, with pSAscan this would require a major redesign.

3. We introduce a new monotone priority queue operation min-comp, which determines if the the smallest key stored in the priority queue is smaller or equal than the key given as argument. Crucially, this does not necessarily require the computation of the smallest key. Using an external memory radix heap augmented with min-comp is a valuable optimization and simplification in our suffix sorting algorithm. We believe that many other applications of monotone priority queues – such as discrete event simulation and sweep-line algorithms – could benefit from this operation.

In §2 we provide a novel description of induced suffix sorting and make several observations which enable our efficient implementation. §3 and §4 provide the details of our new approach. In particular, §4.2 describes the min-comp operation and its advantages. §5 presents and analyses experimental results. §6 suggests some directions future work might take.

2 Induced Suffix Sorting

We begin with a detailed description of the induced suffix sorting procedure [19], which can be easily implemented in RAM but requires some modifications for external memory implementation as described in the next section. Our description differs from many others in the literature in the following aspects:

- We sometimes use a different notation. In particular, instead of L and S we use $-$ and $+$, which we find more descriptive (see Lemma 2.1 for example).
- We start inducing from $-*$ (L*) positions instead of $+*$ (S*) positions. This makes Lemma 2.2 and some implementation details simpler.
- We do not store the various position subsets into subarrays of the suffix array but keep them in separate queues or stacks, which is closer to the external memory implementation.
- To our knowledge, the technique for computing lexicographical names in §2.5 has never been described in the literature. However, a similar technique is used in the RAM-based induced sorting implementation by Yuta Mori.[1]

2.1 Basic Definitions. Let $\mathsf{X} = \mathsf{X}[0..n)$ be a string over an integer alphabet $[0..\sigma)$. Here and elsewhere we use $[i..j)$ as a shorthand for $[i..j-1]$. For $i \in [0..n]$ we write X_i to denote the *suffix* of X of length $n-i$, that is $\mathsf{X}_i = \mathsf{X}[i..n) = \mathsf{X}[i]\mathsf{X}[i+1]\dots\mathsf{X}[n-1]$. More generally, we write $\mathsf{X}[i..j)$ or $\mathsf{X}[i..j-1]$ to denote the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\dots\mathsf{X}[j-1]$ of length $j-i$.

The *suffix array* SA of a string X contains the starting positions of the non-empty suffixes of X in the lexicographical order, i.e., it is an array $\mathsf{SA}[0..n)$ which contains a permutation of the integers $[0..n)$ such that $\mathsf{X}[\mathsf{SA}[0]..n) < \mathsf{X}[\mathsf{SA}[1]..n) < \cdots < \mathsf{X}[\mathsf{SA}[n-1]..n)$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{X}[i..n)$ is the $(j+1)^{\text{th}}$ suffix of X in ascending lexicographical order.

2.2 Categories of Positions. We divide suffixes and their starting positions into two categories according to whether they are smaller or larger than the suffix starting at the next position. Formally, let

$$C^- = \{i \in [0..n) \mid \mathsf{X}_i > \mathsf{X}_{i+1}\},$$
$$C^+ = \{i \in [0..n) \mid \mathsf{X}_i < \mathsf{X}_{i+1}\}.$$

We are interested in runs of positions of the same type and call the leftmost positions in runs $*$-positions:

$$C^{-*} = \{i \in C^- \mid i-1 \in C^+\},$$
$$C^{+*} = \{i \in C^+ \mid i-1 \in C^-\}.$$

An example showing the above classification is given in Table 1. Notice that 0 is never a $*$-position. We also categorize suffixes/positions by their first character, and consider the intersections of the different categories. For any $c \in [0..\sigma)$ and $\alpha \in \{-,+,-*,+*\}$, let

$$C_c = \{i \in [0..n) \mid \mathsf{X}[i] = c\},$$
$$C_c^\alpha = C_c \cap C^\alpha.$$

The C_c^-- and C_c^+-suffixes are lexicographically separated as shown by the following lemma.

LEMMA 2.1. *For any* $c \in [0..\sigma)$, $C_c^- = \{i \in C_c \mid \mathsf{X}_i < ccc\dots\}$ *and* $C_c^+ = \{i \in C_c \mid \mathsf{X}_i > ccc\dots\}$.

[1] Available at https://sites.google.com/site/yuta256/sais

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

2.3 Sorting Positions. For $i \in [0..n)$, define $\text{next}(i) = \min\{k \in [i+1..n] \mid k \in \{n\} \cup C^{-*}\}$ and

$$\mathsf{X}_i^{-*} = \begin{cases} \mathsf{X}[i..\text{next}(i)] & \text{if next}(i) < n \\ \mathsf{X}[i..n) & \text{if next}(i) = n \end{cases}$$

The strings X_i^{-*} are called $-*$*-ending substrings* and have the following property.

LEMMA 2.2. *For any $i, j \in [0..n)$, $\mathsf{X}_i < \mathsf{X}_j$ if and only if $\mathsf{X}_i^{-*} < \mathsf{X}_j^{-*}$ or $\mathsf{X}_i^{-*} = \mathsf{X}_j^{-*}$ and $\mathsf{X}_{\text{next}(i)} < \mathsf{X}_{\text{next}(j)}$.*

Proof. The result is clearly true except when X_i^{-*} is a proper prefix of X_j^{-*}. But then either $\text{next}(i) = n$ and the result is true, or $i + |\mathsf{X}_i^{-*}| - 1 = \text{next}(i) \in C^-$ and $j + |\mathsf{X}_i^{-*}| - 1 \in C^+$, and the result is true by Lemma 2.1.

For any set $C \subseteq [0..n)$ of text positions, we will use $\dot{C}$ to denote C ordered by the characters at those positions, $\overline{C}$ to denote C ordered by $-*$-ending substrings starting at those positions, and $\overrightarrow{C}$ to denote C ordered by the suffixes starting at those positions.

By Lemma 2.1, we can now express the suffix array as follows: $\mathsf{SA} = \overrightarrow{C_0^-}\overrightarrow{C_0^+}\overrightarrow{C_1^-}\overrightarrow{C_1^+}\cdots\overrightarrow{C_{\sigma-1}^-}\overrightarrow{C_{\sigma-1}^+}$. The construction is done in three main phases:

1. Compute $\overline{C^{-*}}$ from $\dot{C^{-*}}$ by inducing.
2. Compute $\overrightarrow{C^{-*}}$ from $\overline{C^{-*}}$ by recursion.
3. Compute $\overrightarrow{C^+}$ and $\overrightarrow{C^-}$ from $\overrightarrow{C^{-*}}$ by inducing.

2.4 Inducing. The basic idea of inducing is to use information about the order of the suffix X_i (the substring X_i^{-*}) to *induce* the order of the suffix $\mathsf{X}_{i-1} = \mathsf{X}[i-1]\mathsf{X}_i$ (the substring $\mathsf{X}_{i-1}^{-*} = \mathsf{X}[i-1]\mathsf{X}_i^{-*}$).

The inducing is done in two phases:

1. PlusInduce: Given C^{-*} in some order, induce C^+ and, as a subsequence, C^{+*}.
2. MinusInduce: Given C^{+*} in the order of the first phase, induce C^- and C^{-*}.

If the input to the first phase is $\dot{C^{-*}}$, the final output we keep is $\overline{C^{-*}}$. If the input is $\overrightarrow{C^{-*}}$, the final output is $\overrightarrow{C^+}$ and $\overrightarrow{C^-}$. The inducing procedures are given Figure 1 (with a separation by the first character maintained at all stages). Note that the inducing order is ascending lexicographical order in MinusInduce and descending lexicographical order in PlusInduce.

To prove the correctness of the inducing procedures, we first show that they produce the correct *sets* of positions using the following characterization of the desired sets.

i	0	1	2	3	4	5	6	7	8	9	10
$\mathsf{X}[i]$	m	i	s	s	i	s	s	i	p	p	i
type	−	+	−	−	+	−	−	+	−	−	−

Table 1: Illustration of sets C^- and C^+ for the example string. We also have $C^{-*} = \{2, 5, 8\}$, $C^{+*} = \{1, 4, 7\}$.

LEMMA 2.3. *For all $i \in [1..n)$, $i-1 \in C^-$ if and only if either $i \in \{n\} \cup C^{+*}$ or $i \in C^-$ and $\mathsf{X}[i-1] \geq \mathsf{X}[i]$; and $i-1 \in C^+$ if and only if either $i \in C^{-*}$ or $i \in C^+$ and $\mathsf{X}[i-1] \leq \mathsf{X}[i]$.*

COROLLARY 2.1. *If the inputs to PlusInduce and MinusInduce are correct as sets, then the outputs are correct as sets.*

Next we characterize the order of the elements in the output sets.

LEMMA 2.4. *Let $C^{-*} = C_0^{-*} \ldots C_{\sigma-1}^{-*}$ denote the input to PlusInduce. Suppose we run PlusInduce, and then use the produced output as the input to MinusInduce. For any $i, j \in C^+$, i is induced (popped from a queue) before j (in PlusInduce) if and only if $\mathsf{X}_i^{-*} > \mathsf{X}_j^{-*}$ or $\mathsf{X}_i^{-*} = \mathsf{X}_j^{-*}$ and next(i) is after next(j) in C^{-*}. For any $i, j \in C^-$, i is induced before j (in MinusInduce) if and only if $\mathsf{X}_i^{-*} < \mathsf{X}_j^{-*}$ or $\mathsf{X}_i^{-*} = \mathsf{X}_j^{-*}$ and next(i) is before next(j) in $n \cdot C^{-*}$.*

Note that the above lemma is true for *any* order of the sets used as input to PlusInduce, in particular for $\dot{C^{-*}}$ and $\overrightarrow{C^{-*}}$. Thus, the correctness of inducing follows directly from Lemma 2.4 when inducing $\overline{C^{-*}}$ from $\dot{C^{-*}}$, and from the combination of Lemmas 2.4 and 2.2 when inducing $\overrightarrow{C^+}$ and $\overrightarrow{C^-}$ from $\overrightarrow{C^{-*}}$.

2.5 Recursion. Let $n' = |C^{-*}|$ and let $\{p_0, p_1, \ldots, p_{n'-1}\} = C^{-*}$ be the sequence of $-*$-positions with $p_0 < p_1 < \cdots < p_{n'-1}$. We sort the substrings X_i^{-*}, $i \in C^{-*}$, that start and end at $-*$-positions and assign them lexicographical names, i.e., integers $r_j \in [0..n')$, $j \in [0..n')$, such that for all $j, k \in [0..n')$, $r_j \leq r_k$ iff $\mathsf{X}_{p_j}^{-*} \leq \mathsf{X}_{p_k}^{-*}$.

The lexicographical names are computed while inducing $\overline{C^{-*}}$. Since inducing produces $\overline{C^{-*}}$ in the correct lexicographical order, all we need to know to assign names is which consecutive elements represent identical $-*$-substrings. For this, we augment each element with a bit that is zero if the element represents the same $-*$-substring as its predecessor in the queue and one otherwise. In fact, we augment all the C and Q queues (and stacks) used during the substring inducing with such a bit. We will next describe how these bits are

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Procedure PlusInduce($C_0^{-*}, \ldots, C_{\sigma-1}^{-*}$)
1: **for** $c \leftarrow \sigma - 1$ **downto** 0 **do**
2: **while** $Q_c \neq \emptyset$ **do**
3: $i \leftarrow Q_c$.popfront()
4: C_c^+.pushfront(i)
5: **if** $i > 0$ **and** $\mathsf{X}[i-1] \leq c$ **then**
6: $Q_{\mathsf{X}[i-1]}$.pushback($i-1$)
7: **else if** $i > 0$ **then**
8: C_c^{+*}.pushfront(i)
9: **for** $i \in C_c^{-*}$ in reverse order **do**
10: $Q_{\mathsf{X}[i-1]}$.pushback($i-1$)
11: **return** $C_0^+, \ldots, C_{\sigma-1}^+$ and $C_0^{+*}, \ldots, C_{\sigma-1}^{+*}$

Procedure MinusInduce($C_0^{+*}, \ldots, C_{\sigma-1}^{+*}$)
1: $Q_{\mathsf{X}[n-1]}$.pushback($n-1$)
2: **for** $c \leftarrow 0$ **to** $\sigma - 1$ **do**
3: **while** $Q_c \neq \emptyset$ **do**
4: $i \leftarrow Q_c$.popfront()
5: C_c^-.pushback(i)
6: **if** $i > 0$ **and** $\mathsf{X}[i-1] \geq c$ **then**
7: $Q_{\mathsf{X}[i-1]}$.pushback($i-1$)
8: **else if** $i > 0$ **then**
9: C_c^{-*}.pushback(i)
10: **for** $i \in C_c^{+*}$ **do**
11: $Q_{\mathsf{X}[i-1]}$.pushback($i-1$)
12: **return** $C_0^-, \ldots, C_{\sigma-1}^-$ and $C_0^{-*}, \ldots, C_{\sigma-1}^{-*}$

Figure 1: The inducing procedures. The C and Q objects are simple queues (when inserting by pushback) or stacks (when inserting by pushfront).

computed in MinusInduce assuming the input already contains such bits. The computation in PlusInduce is essentially symmetric.

We will assign a rank to each position i popped from Q_c on line 4 or from C_c^{+*} on line 10. If the bit augmenting i in the queue is zero, the rank of i is the same as the rank of its predecessor. Otherwise, the rank of i is the smallest positive integer not yet used as a rank. It is not difficult to see that the rank is a lexicographical name among all $-*$-substrings starting in $C^- \cup C^{+*}$. For each queue C_c^- and C_c^{-*}, we keep the rank of the last position inserted into the queue. When inserting, we compare the rank of the new position to the rank of the previously inserted position to determine the augmenting bit of the new element. Insertions into the queues Q_c are processed similarly, except the rank stored with the queue is the rank of position $i+1$ if the last inserted position is i (because we do not yet know the rank of i). Since $\mathsf{X}[i] = c$ for all positions i inserted into Q_c, the augmenting bits are still correctly computed.

Let $\mathsf{R} = r_0 r_1 \ldots r_{n'-1}$ be the concatenation of the lexicographical names. Then:

LEMMA 2.5. *For any $i, j \in [0..n')$, $\mathsf{X}_{p_i} < \mathsf{X}_{p_j}$ if and only if $\mathsf{R}_i < \mathsf{R}_j$.*

Therefore, we can sort the $-*$-suffixes by sorting all suffixes of R, which can be done recursively by induced suffix sorting. Since $n' \leq n/2$, the length of the string drops exponentially with the depth of the recursion, and thus the recursive calls do not increase the asymptotic time complexity.

2.6 Full Algorithm. The complete algorithm for constructing the suffix array is the following. It can be implemented to run in $\mathrm{O}(n)$ time in internal memory.

1. Construct $\dot{C^{-*}}$ by scanning X once.
2. Construct $\overline{C^{-*}}$ by inducing from $\dot{C^{-*}}$.
3. Construct the string R.
4. Construct the suffix array of R recursively.
5. Construct $\overrightarrow{C^{-*}}$ from the suffix array of R.
6. Construct $\overrightarrow{C^+}$ and $\overrightarrow{C^-}$ by inducing from $\overrightarrow{C^{-*}}$.
7. Merge $\overrightarrow{C^+}$ and $\overrightarrow{C^-}$ into SA.

3 Induced Suffix Sorting in External Memory

The generic induced suffix sorting described above involves a lot of sequential access to data, which is in principle easy to perform efficiently even if the data is in external memory. Some steps — specifically steps 3 and 5 — require external memory sorting/permuting to get elements into an appropriate order. The problematic operations for external memory implementation are the accesses to the preceding characters $\mathsf{X}[i-1]$ when inducing, because those can be essentially random accesses. Another difficulty, which we will consider first, is dealing with too many sequences when the alphabet is large.

3.1 Large Alphabet. During the inducing, the algorithm deals with more than σ sequences simultaneously and needs to keep at least a buffer of some minimum size, say B, in RAM (of size M) for each sequence. When $\sigma > M/B$, this is no more possible. Even if the original string X has a small alphabet, the strings in the recursive calls can have very large alphabets.

For a large alphabet, we will replace the individual sequences C_c^-, C_c^+, C_c^{-*}, and C_c^{+*}, $c \in [0..\sigma)$, with their concatenations C^-, C^+, C^{-*}, and C^{+*}. The

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

accesses to these concatenations are still sequential during the algorithm. We will also maintain the lengths of individual sequences as a separate list so that we can tell where one sequence ends and another begins.

This concatenation approach does not work with the queues Q_c as they are accessed simultaneously. Instead, the queues Q_c, $c \in [0..\sigma)$, are replaced with a single priority queue Q with the symbol c as the priority. Using a minimum priority queue in MinusInduce and a maximum priority queue in PlusInduce leads to the correct order of extraction. Using an I/O-optimal external memory priority queue [2] the I/O-complexity of inducing becomes $\mathrm{O}(\mathrm{sort}(n)) = \mathrm{O}((n/B)\log_{M/B}(n/B))$.

The technique for computing lexicographical names for $-*$-substrings still works with mostly minor changes. In fact, essentially no change is needed for computing the augmenting bits in the concatenated C sequences. With the priority queue Q too, the augmenting bit of each element is exactly the same as it was with separate queues. To compute those bits, however, we will need to keep a separate rank for each symbol $c \in [0..\sigma)$ corresponding to the rank associated with the separate queue Q_c. For extremely large alphabets (say, $\sigma > M/2$), we may not have enough free RAM to store all those ranks. In that case, we store the actual ranks instead of the augmenting bits with each priority queue element.

3.2 Blockwise Preinducing. To deal with the essentially random accesses to the text characters $\mathsf{X}[i-1]$ during inducing we use a technique we call *blockwise preinducing*, where we divide the text into blocks of size m and precompute the sequence of symbol accesses into each block. When we need to know $\mathsf{X}[i-1]$, we determine which block contains the position $i-1$ and then read the next symbol from the precomputed sequence associated with that block. A more cumbersome form of blockwise preinducing was introduced in [18].

Let Y be one of the blocks, and let k be the first C^{-*}-position in X after the end of Y. Let Y' be Y extended to the right up to but not including $\mathsf{X}[k]$. Let $D^{-*} \subseteq C^{-*}$ be the set of C^{-*}-positions in Y plus the position k in the same relative order they are in C^{-*}. We perform inducing on Y' using D^{-*} as the initial input with one modification: line 1 in MinusInduce is omitted because the position k, which is now the beyond-the-end position instead of n, is included in D^{-*} in its appropriate position.

LEMMA 3.1. *If we perform inducing on Y' using D^{-*} as the initial input, the preceding symbol accesses $\mathsf{Y}'[i-1]$ occur in the same relative order as they occur when inducing on X using C^{-*} as the initial input.*

Proof. Follows directly from Lemma 2.4.

If $|\mathsf{Y}'| > 2|\mathsf{Y}|$, we truncate Y' into length $2|\mathsf{Y}|$ and add the position after Y' into D^{-*} as dummy C^{-*}-position with arbitrary ordering. This may change the inducing order of some positions in Y' but not those in Y.

During the inducing of Y', we store the symbol accesses inside Y into a file. Then during the inducing of X, when we need $\mathsf{X}[i-1]$ which is inside Y, we simply read the next symbol from the file. The maximum number of active files, and thus the maximum number of blocks, we can handle is $\mathrm{O}(M/B)$. If the resulting (extended) blocks are too big to fit in RAM, we apply the blockwise preinducing technique recursively. The necessary number of recursive levels is $\mathrm{O}(\log_{M/B}(n/M))$, and thus the I/O complexity of the full algorithm is $\mathrm{O}((n/B)\log^2_{M/B}(n/B))$.

4 Radix Heap

A (minimum) priority queue is a data structure that stores key-value pairs and supports at least the operations $\mathsf{insert}(k, v)$, which adds the pair (k, v) into the queue, and $\mathsf{extract\text{-}min}()$, which removes a pair with the smallest key and returns it. A priority queue is *stable* if $\mathsf{extract\text{-}min}()$ returns key-value pairs with the same key in the order they were inserted into the priority queue. A *monotone* priority queue has the additional restriction that an inserted key may not be smaller than the last extracted key. This restriction ensures that elements are extracted in monotonic order.

A well-known monotone, stable priority queue is the radix heap [1]. As a practical optimization, we use an external memory radix heap as the priority queue in inducing. The implementation is ours but it is similar to the one by Brengel et al. [5] and we omit the details. However, we want to highlight two aspects specific to our implementation and usage.

4.1 Stability and PQ Element Size. The priority queue used during inducing needs to be stable. Any priority queue can be made stable by using a time stamp representing the insertion time as a secondary key, and this is how eSAIS [4] achieves stability, for example. However, the radix heap is naturally stable, and no time stamp is needed. This reduces the size of the elements stored in the priority queue, which reduces I/O and disk space usage substantially.

This saving is particularly significant because the element size in our algorithm is smaller than in eSAIS for other reasons too. In particular, eSAIS does not use preinducing and instead carries several preceding characters with each priority queue element to avoid random accesses to those preceding characters.

As an additional optimization, we do not store text

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

positions with the elements but only the preinducing block numbers. The text position, if needed, is stored with the precomputed symbol sequence associated with the block. We also store that position relative to the beginning of the block, so that the combined space for the block number and the relative position is about the same as the space for the global position. This optimization is particularly effective when inducing $\overline{C^{-*}}$ from $C^{\dot{-}*}$, as we need the exact text positions only for $-*$-positions.

4.2 Monotonicity and Min-Comp Operation. Consider a situation, where we have a monotone priority queue Q and a simple sorted queue S. At each step we want extract the smallest element in either of the two queues, and as a result may insert a new element into Q that is no smaller than the extracted element. This is exactly the situation in inducing, for example in MinusInduce with $S = C^{+*}$. This kind of situation can also arise in other applications including discrete event simulation and sweep line algorithms. In discrete event simulation, a sequence of events is processed in order of time. Some events and their times may be known in advance and can be stored as a sorted sequence S, while other events are generated during the simulation and stored in a priority queue Q. A sweepline algorithm, such as the classical segment intersection algorithm [3], processes points in a Euclidean space in order of one dimension. Again some of the points (the input) are known in advance while others are generated during the computation.

Suppose we extract an element x from Q and compare it to the smallest element y in S. If $y < x$, we process y and might then want to insert an element $z \in [y..x)$ into Q, but the monotonicity of Q prevents this. Instead of extracting x, we might fetch x without deleting it using a min()-operation. However, a min()-operation is not a natural operation for monotone priority queues as the minimum can increase as well as decrease. An efficient implementation of min() requires some non-trivial additional data structures, and many implementations do not support min(). Without a min(), we could (i) use a non-monotone priority queue, (ii) keep a prematurely extracted value x in a separate data structure — possibly a non-monotone priority queue — or (iii) insert the elements of S into Q at the beginning, but all options can increase costs.

We propose an alternative operation to min() called min-comp(k), which returns true if the minimum element in the priority queue is smaller or equal to k. The operation min-comp(k) also sets the insertion lower bound to $\min\{k, m\}$, where m is the current minimum value in the priority queue. In our example scenario, we would call min-comp(y) to determine that y is smaller than (an unknown) x. Since this sets the insertion lower bound to y, an insertion of $z \geq y$ is not a problem. For a monotone priority queue, min-comp is much more natural and simpler to implement than min, and does not require additional data structures. Our radix heap implementation has min-comp (and no min) and it is used during inducing.

5 Experimental Results

5.1 Setup. We performed experiments on a machine equipped with two six-core 1.9 GHz Intel Xeon E5-2420 CPUs (capable, via hyper-threading, of running 24 threads) with 15 MiB L3 cache and 120 GiB of DDR3 RAM. For experiments we limited the RAM in the system to 4 GiB (with the kernel boot flag) and all algorithms were allowed to use 3.5 GiB. The machine had 6.8 TiB of free disk space striped with RAID0 across four identical local disks achieving a (combined) transfer rate of about 480 MiB/s (read/write). All disks used in the experiments were formatted to `ext4` using default settings. All programs except eSAIS (which uses the STXXL library [8] for file I/O) allocate disk space usage on demand, i.e., no `fallocate()` optimizations were performed. The STXXL block size was set in preliminary experiments to 1 MiB. All kernel, disk cache and the I/O scheduler configuration were otherwise unaltered.

The OS was Linux (Ubuntu 12.04, 64bit) running kernel 3.13.0. All programs were compiled using `g++` version 5.2.1 with `-O3 -DNDEBUG -march=native` options. All reported runtimes are wallclock (real) times. The machine had no other significant CPU tasks running and for all algorithms except pSAscan only a single thread of execution was used for computation (we permit a constant number of extra threads as long as they do not perform computation, e.g., threads responsible for scheduling I/O requests are allowed). For pSAscan we used the full parallelism on the machine (24 threads).

5.2 Data Set. For the experiments we used the following files, including both artificial data and inputs from real-world applications (see Table 2 for more detailed statistics).

- **wiki**: recent English, German and French Wikipedia dumps (`http://dumps.wikimedia.org/`) in the XML format concatenated, and truncated to 120 GiB. This file represents natural text.

- **kernel**: a concatenation of ∼5.3 million source files from over 150 versions of Linux kernel. This is an example of highly repetitive file (`http://www.kernel.org/`).

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

- **dna**: a collection of DNA reads (produced by a sequencing machine) from multiple human genomes (http://www.1000genomes.org/) filtered of symbols other than {A, C, G, T, N} and newline.

- **skyline**: an artificial, highly repetitive sequence of length $2^k - 1$ generated by the grammar $\{S \to T_k, T_k \to T_{k-1}kT_{k-1}, T_1 \to 1\}$ for which exactly half of all suffixes at each recursion level are $-*$-suffixes. This is the worst case input for induced suffix sorting algorithms [4].

- **words**: a large collection of natural language text (Turkish) parsed into words and converted into integers (using minimal alphabet) so that all occurrences of the same word in the original text are represented with the same integer, distinct from those assigned to other words; the file is from the collection of training datasets for the machine translation competition (http://www.statmt.org/wmt16/translation-task.html).

Smaller files in experiments are prefixes of full test files. For **skyline**, we generated the input separately for each size. The input symbols are encoded using the minimal number of bytes, i.e., for all files we use byte encoding, except for the **words** file, for which we use 32 bits per symbol. All algorithms encode the output suffix array using 40-bit integers.

5.3 Algorithms. In our experiments we use the following algorithms and implementations:

- **fSAIS**: the new (faster) EM SACA based on induced sorting principle described in this paper. This is the main contribution of this paper.[2]

- **eSAIS** (v0.5.2):[3] the first external-memory implementation of induced suffix sorting by Bingmann et al. [4]. The implementation is fully scalable (i.e., it does not have any serious restriction on the input size) due to the use of the STXXL library [8] which handles all fundamental I/O tasks (scanning, sorting) and implements the scalable EM priority queue. More importantly, however, until now it remained unchallenged in terms of speed, and thus we use it as a baseline in our experiments.[4]

- **DSA-IS** (v11): an external-memory implementation of induced suffix sorting described in Nong et al. [18].[5] It uses a technique similar to our preinducing, but the algorithm and implementation assume that $n = \mathcal{O}(M^2/B)$, and thus the algorithm is less scalable than eSAIS.[6]

- **pSAscan** (v0.1.0): a parallel external-memory suffix array construction algorithm described in [13].[7] pSAscan is currently the fastest way to compute the suffix array if n/M (the ratio between text size and RAM size) is not very large, which is a common situation in practice. The I/O complexity of pSAscan is $\mathcal{O}(\mathrm{sort}(n) + n^2/(MB\log_\sigma n))$, i.e., for sufficiently large value of n/M a SACA with sorting I/O complexity will always be faster.

Input	$n/2^{30}$	$\lvert\Sigma\rvert$
wiki	120.0	213
kernel	192.0	229
dna	192.0	6
skyline	128.0	$\lfloor \log_2 n \rfloor$
words	12.5	97 002 175

Table 2: Statistics of data used in the experiments.

5.4 Comparison of Algorithms Based on Induced Sorting. In the first experiment, we compare the scalability of the new algorithm described in this paper to eSAIS and DSA-IS. For now we restrict our attention to texts over a byte alphabet. We executed all three algorithms on increasing length prefixes of all testfiles and measured the runtime and I/O volume. The results are given in Figure 2.

Due to the limited scalability of DSA-IS we were only able to run the algorithm on small inputs (for larger texts the program terminates abnormally or exits without any error message). However, in the experiments reported in [18], DSA-IS is only ever "marginally faster" than eSAIS, which is also confirmed in our experiments.

Compared to eSAIS, the new algorithm described in this paper is about two times faster on all types of data. The difference in runtime is consistent with the I/O volume. The mean I/O throughput (in MB/s; obtained by dividing the normalized I/O volume by the normalized runtime) of eSAIS is between about 80 MB/s (wiki) and 120 MB/s (skyline). For fSAIS the I/O throughput is

[2] Available at https://www.cs.helsinki.fi/group/pads/

[3] We use version 0.5.2 rather than 0.5.4 (latest), because we found the latest version to get stuck indefinitely on some testfiles.

[4] Code available at https://panthema.net/2012/1119-eSAIS-Inducing-Suffix-and-LCP-Arrays-in-External-Memory/

[5] Available from http://code.google.com/p/ge-nong/

[6] We note that Liu et at. [16] and a subset of the authors of [18] describe yet another EM variant based on induced sorting. In the experiments in [16], the algorithm outperforms eSAIS on a single testfile by about 35%. The runtime, however, is not consistent with the I/O volume and on all remaining testfiles used in experiments, their algorithm is slower than eSAIS by a factor 1.4–2.0.

[7] Available at https://www.cs.helsinki.fi/group/pads/

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

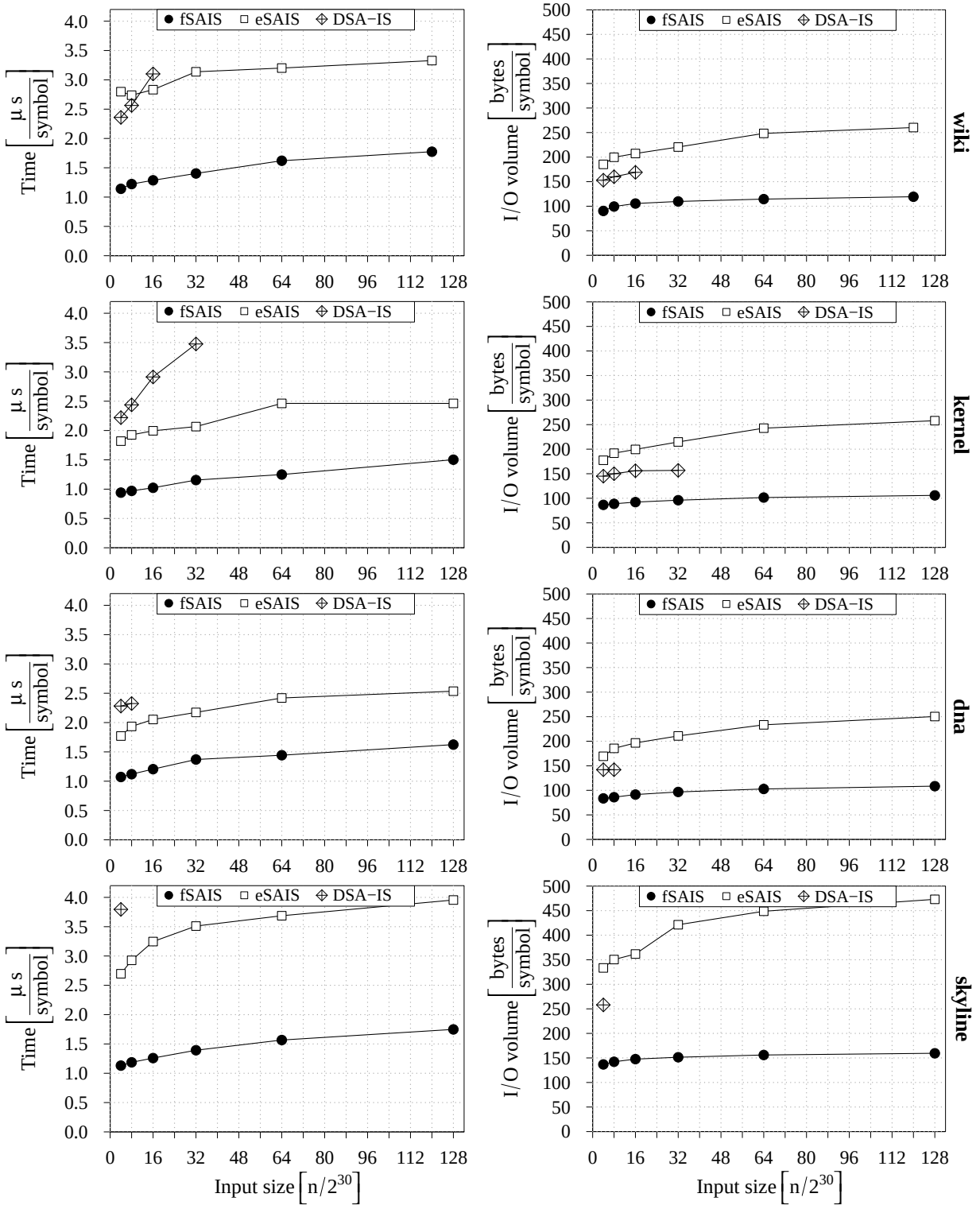

Figure 2: Runtime (left; in μs per input symbol) and I/O volume (right; in bytes per input symbol) of the external-memory suffix array construction algorithm presented in this paper (fSAIS) compared to previously fastest implementation (eSAIS) on increasing length prefixes of testfiles. In the comparison we also include the DSA-IS algorithm. The existing implementation, however, managed to process only small prefixes. All algorithms were allowed to use 3.5 GiB of RAM for all prefixes.

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

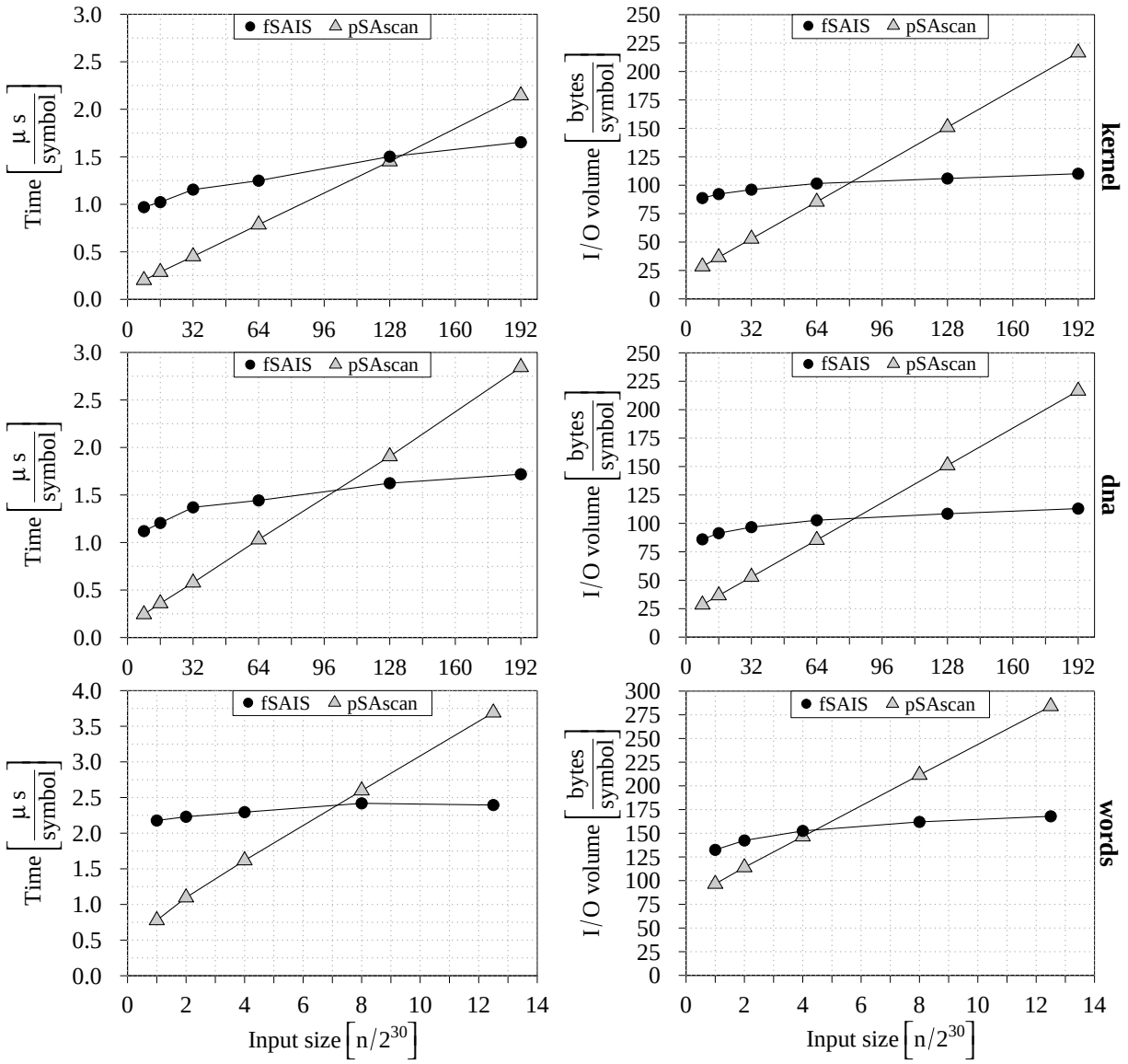

Figure 3: Comparison of the new sequential algorithm described in this paper (fSAIS) to the currently fastest parallel external-memory suffix sorting algorithm (pSAscan) on the small-alphabet (top two rows) and large-alphabet (bottom row) input. Both algorithms use 3.5 GiB of RAM. pSAscan uses the full parallelism on experimental platform (24 threads).

between 65 MB/s (dna) and 90 MB/s (skyline). We note that this means both algorithms achieve less than 25% of the maximum I/O throughput of the experimental platform, indicating they are largely compute-bound rather than I/O-bound.

5.5 Comparison with pSAscan. In the experimental comparison in [13] (using the same experimental platform and setup, e.g., amount of RAM available for algorithms, as here) the text-to-RAM ratio at which eSAIS becomes faster than pSAscan is around 75. We now revise this experiment to determine the crossover point of fSAIS and pSAscan. It should be kept in mind that fSAIS is not parallelized, while pSAscan is.

The results are presented in Figure 3. On the **dna** testfile (which is the same testfile as used in [13]), the point where fSAIS overtakes pSAscan is achieved for prefix of size $n = 104 \times 2^{30}$, i.e., when the text-to-RAM ratio is around 30. On the highly repetitive **kernel** testfile, where pSAscan is usually faster, the crossover is achieved for prefix of size $n = 136 \times 2^{30}$, i.e., when the text-to-RAM ratio is around 38.

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

5.6 Large Alphabet. Even if the input string consists of symbols from a small alphabet, the strings created in recursive calls of the induced suffix-sorting algorithm usually have millions of different symbols. Thus, the algorithms based on the inducing principle natively deal with large alphabets. Unfortunately all implementations we know of do not provide clean interfaces for large alphabets.

An alternative method to compute the suffix array for a string over large alphabet is to split each symbol into a group of symbols over a smaller alphabet. If the splitting is done appropriately (so that next symbol in a group in left-to-right order is created by removing most significant byte from the original symbol), one can apply a byte-based suffix sorter, e.g., pSAscan, and then select the subset of suffixes to obtain the suffix array of the original string.

The drawback of this approach is that the process of reducing the alphabet increases the length of the string. For example, if the symbols of the original string of length n were encoded using 32-bit integers and we wish to obtain a string over byte alphabet, the resulting string has length $4n$.

In this experiment, we compare these two approaches. Namely, we compare the performance of fSAIS to pSAscan. We ran both algorithms on the increasing length prefixes of the **words** testfile using 3.5 GiB of RAM and measured time and I/O volume. fSAIS takes the input without modifications (each symbol encoded using 32-bit integer), and pSAscan splits each symbol into four bytes, computes the suffix array of the modified string and in the final step only outputs the positions i such that $i \bmod 4 = 0$. We scaled the runtime (and I/O volume) of pSAscan with respect to the original string.

The results are given in Figure 3. The point at which fSAIS overtakes pSAscan is significantly smaller than for inputs over byte-alphabet. In particular, fSAIS is the more efficient algorithm already for $n = 7.5 \times 2^{30}$, which means a text-to-RAM ratio of about 8.5.

5.7 Disk Space Usage. Lastly, we have a look at the disk space usage of all algorithms. We measured the peak disk space usage (including input text and the output suffix array) on the 32 GiB prefixes of all testfiles over byte alphabet. The text-to-RAM ratio for such prefix is around 10 hence it is representative for real world usage. All algorithms were allowed to use 3.5 GiB of RAM. The results are given in Table 3.

The peak disk usage of fSAIS is over three times smaller than for eSAIS and over four times smaller than DSA-IS. It is also very close to pSAscan, and since the space for input and output is n (text) + $5n$ (suffix array encoded using 40-bit integers) = $6n$ bytes, the disk usage of fSAIS is close to optimal. The results on the **skyline** input for the algorithms based on inducing principle show the extremal disk allocation, as skyline is the worst-case input for such algorithms.

Algorithm	wiki	kernel	dna	skyline
fSAIS	$7.7n$	$7.7n$	$7.7n$	$8.1n$
eSAIS	$23.6n$	$23.5n$	$23.4n$	$28.0n$
DSA-IS	$34.5n$	$34.2n$	$32.8n$	$40.1n$
pSAscan	$7.5n$	$7.5n$	$7.5n$	$7.5n$

Table 3: Peak disk space usage (in bytes) for all algorithms on the 32 GiB prefixes of byte-alphabet testfiles. The numbers include the input text and the output suffix array. All values are reported by our own measurements (the disk usage of DSA-IS in our measurements was much higher than reported in [18]). For DSA-IS we report the disk usage for largest prefix we were able to process (see Figure 2).

The small disk usage of fSAIS is largely due to the reduced size of priority queue elements as described in § 4.1. Furthermore, whenever possible, the algorithm stores the temporary data using multiple files. In many stages of the computation the output is produced gradually, allowing to simultaneously delete the temporary input files. By carefully choosing the size of a single file, the algorithm suffers only negligible slowdown but has a significantly smaller disk usage. A similar technique is also used in pSAscan. It could in principle be used to reduce the disk space usage of other algorithms but for example in the case of eSAIS this is not as straightforward, as it uses the STXXL library which preallocates all the necessary disk space at the beginning of execution.

Finally, we also measured the peak disk usage for larger prefixes of testfiles. We do not give a full report here, but point out that the above results are representative, i.e., the peak disk usage for larger prefixes does not increase significantly. For example, the peak disk usage of fSAIS on 128 GiB prefix of **skyline** is $8.2n$ bytes. The peak disk usage of pSAscan on 192 GiB prefixes also increases only up to $8.2n$ bytes. The peak disk usage of eSAIS essentially does not change for larger prefixes.

6 Concluding Remarks

We have described a highly scalable external memory implementation of induced suffix sorting that is about two times faster than previous approaches, and simultaneously has close to optimal disk space usage. A promising direction to further speed up the algorithm is parallelism, since our experiments suggests that it is more compute-bound than I/O-bound. Inducing is rather sequential by nature and may be difficult to parallelize, but Labeit

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited

et al. [15] have shown that some speed up is possible. Furthermore, there are easy possibilities for parallelism such as preinducing multiple blocks simultaneously.

Acknowledgements

We thank Matthias Petri for providing us with a large alphabet data set at short notice.

References

[1] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.

[2] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[3] Jon L. Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

[4] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and LCP arrays in external memory. In *Proc. 15th Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 103–112. SIAM, 2013.

[5] Klaus Brengel, Andreas Crauser, Paolo Ferragina, and Ulrich Meyer. An experimental study of priority queues in external memory. *ACM Journal of Experimental Algorithmics*, 5:Article 17, 2000.

[6] Andreas Crauser and Paolo Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[7] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12:Article 3.4, 2008.

[8] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[9] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[10] Simon Gog and Matthias Petri. Compact indexes for flexible top-k retrieval. In *Proc. 26th Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 207–218. Springer, 2015.

[11] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *Proc. 6th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 81–88. IEEE, 1999.

[12] Juha Kärkkäinen and Dominik Kempa. Engineering a lightweight external memory suffix array construction algorithm. In *Proc. 2nd International Conference on Algorithms for Big Data (ICABD)*, pages 53–60. CEUR, 2014.

[13] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In *Proc. 26th Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 329–342. Springer, 2015.

[14] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2–4):143–156, 2005.

[15] Julian Labeit, Julian Shun, and Guy E. Blelloch. Parallel lightweight wavelet tree, suffix array and FM-index construction. In *Proc. Data Compression Conference (DCC)*, pages 33–42. IEEE, 2016.

[16] Weijun Liu, Ge Nong, Wai Hong Chan, and Yi Wu. Induced sorting suffixes in external memory with better design and less space. In *Proc. 22nd Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 9309, pages 83–94. Springer, 2015.

[17] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

[18] Ge Nong, Wai Hong Chan, Sheng Qing Hu, and Yi Wu. Induced sorting suffixes in external memory. *ACM Transactions on Information Systems*, 33(3):12:1–12:15, 2015.

[19] Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

[20] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007.

[21] Ehsan Shareghi, Matthias Petri, Gholamreza Haffari, and Trevor Cohn. Compact, efficient and unlimited capacity: Language modeling with compressed suffix trees. In *Proc. Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2409–2418. The Association for Computational Linguistics, 2015.

Downloaded 01/05/17 to 128.214.9.98. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM
Unauthorized reproduction of this article is prohibited