Parallel External Memory Suffix Sorting

Juha Kärkkäinen, Dominik Kempa(✉), and Simon J. Puglisi

Department of Computer Science, Helsinki Institute for Information Technology HIIT, University of Helsinki, Helsinki, Finland
{juha.karkkainen,dominik.kempa,simon.puglisi}@cs.helsinki.fi

Abstract. Suffix sorting (or suffix array construction) is one of the most important tasks in string processing, with dozens of applications, particularly in text indexing and data compression. Some of these applications require the suffix array to be built for large inputs that greatly exceed the size of RAM and so external memory must be used. However, existing approaches for external memory suffix sorting either use debilitatingly large amounts of disk space, or become too slow when the size of the input data is more than a few times bigger than the size of RAM. In this paper we address the latter problem via a non-trivial parallelization of computation. In our experiments, the resulting algorithm is much faster than the best prior external memory algorithms while using very little disk space in addition to what is needed for the input and output. On the way to this result we provide the current fastest (parallel) internal memory algorithm for suffix sorting, which is usually around twice as fast as previous methods, while using around one quarter of the working space.

1 Introduction

Suffix sorting (or suffix array construction) is one of the most important tasks in string processing. It is fundamental to building index data structures such as suffix trees [10,32], (compressed) suffix arrays [14,24], and FM-indexes [11], which in turn have dozens of applications in bioinformatics, including pattern matching (i.e. read alignment [21,22]), genome assembly [30], and discovery of repetitive structures [1]. Suffix sorting is also key to several major lossless compression transforms, such as the Burrows-Wheeler transform, Lempel-Ziv (LZ77) parsing [16,17,34], and several grammar compressors (e.g. [4,26]). Many of these applications deal with massive data and often suffix sorting is the computationally most demanding task.

Suffix sorting is also one of the most studied tasks in string processing [29], but the majority of the work has focused on sequential, internal memory algorithms, which do not really scale for massive data and do not fully utilize the resources on modern computers. There has been some research on speeding up suffix sorting by parallel computation and on external memory suffix sorting algorithms that escape the limits of RAM, but no really effective combination of

The work is partially supported by the Academy of Finland through grant 258308.

F. Cicalese et al. (Eds.): CPM 2015, LNCS 9133, pp. 329–342, 2015.
DOI: 10.1007/978-3-319-19929-0_28

the two approaches. This is not very surprising since external memory computation is often I/O-bound and would not benefit greatly from (internal memory) parallelism. Nevertheless, in this paper we show that the two computational paradigms can be fruitfully combined in a suffix sorting algorithm.

Our contribution. Our starting point is the recent external memory suffix sorting algorithm SAscan [15], the basic idea of which is to divide the text into blocks, construct suffix arrays for the blocks and then merge these partial suffix arrays. In this paper, we describe a parallelization of the central procedure that merges two partial suffix arrays. Using this procedure, we first design an internal memory suffix sorting algorithm that constructs several partial suffix arrays in parallel (using any sequential suffix sorter) and then merges them together. The result is the fastest internal memory algorithm that we are aware of. This internal memory suffix sorter and the parallel merging procedure are then used in designing a parallelized version of SAscan which we call pSAscan. On a machine with 12 physical cores (24 with hyper-threading), pSAscan is over four times faster than SAscan and much faster than any other external memory algorithm in all of our experiments.

The algorithms are not theoretically optimal. The internal memory algorithm needs $\Omega(n \log p)$ work on p processors, and the external memory pSAscan needs $\widetilde{\Omega}(n^2/M)$ work, where M is the size of the RAM. However, low constant factors and, crucially, space efficiency make them more scalable in practice than their competitors. The internal memory algorithm needs less than $10n$ bytes of RAM, and pSAscan needs just $7.5n$ bytes of disk space, which is nearly optimal. The best competitors use about four times as much RAM/disk space, which is likely to be a more serious limitation to their scalability than the time complexity is to our algorithms. To demonstrate the scalability, we have constructed the suffix array of a 1 TiB text in a little over 8 days.

Related work. The idea of external memory suffix sorting by merging separately constructed partial suffix arrays goes back over 20 years [13], and there has been several improvements over the years [7,12] (see also [31]). The recent incarnation SAscan [15] is one of the fastest external memory suffix sorters in practice. A different approach to merging suffix arrays in [23] is limited to merging separate files rather than blocks of the same file. The main competitor of SAscan is the eSAIS algorithm by Bingmann, Fischer and Osipov [5]. eSAIS is theoretically optimal but suffers from a large disk space usage (roughly $28n$ bytes, for an input of n symbols). SAscan needs just $7.5n$ bytes of disk space but because of its $\widetilde{\mathrm{O}}(n^2/M)$ time complexity, it is competitive with eSAIS only when the input is less than about five times the size of RAM. The new pSAscan extends the advantage over eSAIS to much bigger inputs. Another recent external memory suffix sorter EM-SA-DS [27] appears to be slightly worse than eSAIS in practice, although a direct comparison is missing.

In contrast to the large number of algorithms for serial suffix sorting [29], results on parallel algorithms for suffix sorting are reasonably sparse. Earlier research focused on suffix tree construction (see, e.g., [3]) and was mostly of theoretical interest. More recently, research into practical algorithms has focused on either distributed [20] or GPU platforms [9,28]. Most relevant to this paper is a parallel version of DC3, a work optimal EREW-PRAM algorithm due to Kärkkäinen, Sanders and Burkhardt [19] that has been subsequently implemented by Blelloch and Shun [6]. We use their implementation as a baseline in experiments with our internal memory algorithm.

2 Preliminaries

Let $\mathsf{X} = \mathsf{X}[0..m)$ be a string over an integer alphabet $[0..\sigma)$. Here and elsewhere we use $[i..j)$ as a shorthand for $[i..j-1]$. For $i \in [0..m)$ we write $\mathsf{X}[i..m)$ to denote the *suffix* of X of length $m - i$, that is $\mathsf{X}[i..m) = \mathsf{X}[i]\mathsf{X}[i+1] \dots \mathsf{X}[m-1]$. Similarly, we write $\mathsf{X}[0..i)$ to denote the *prefix* of X of length i and $\mathsf{X}[i..j)$ to denote the *substring* $\mathsf{X}[i]\mathsf{X}[i+1] \dots \mathsf{X}[j-1]$ of length $j-i$. If $i = j$, the substring $\mathsf{X}[i..j)$ is the empty string, also denoted by ε.

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

The Burrows-Wheeler transform $\mathsf{BWT}_\mathsf{X}[0..m)$ of a string X contains the characters preceding each suffix in lexicographical order: $\mathsf{BWT}_\mathsf{X}[i] = \mathsf{X}[\mathsf{SA}_\mathsf{X}[i]-1]$ if $\mathsf{SA}_\mathsf{X}[i] > 0$ and otherwise \$, a special symbol that does not appear in the text.

Partial suffix arrays. The partial suffix array $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ is the lexicographical ordering of the suffixes of XY with a starting position in X, i.e., it is an array $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[0..m)$ that contains a permutation of the integers $[0..m)$ such that $\mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[0]..m)\mathsf{Y} < \mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[1]..m)\mathsf{Y} < \cdots < \mathsf{X}[\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}[m-1]..m)\mathsf{Y}$. Note that $\mathsf{SA}_{\mathsf{X}:\varepsilon} = \mathsf{SA}_\mathsf{X}$ and that $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ is usually similar but not identical to SA_X. Also note that $\mathsf{SA}_{\mathsf{X}:\mathsf{Y}}$ can be obtained from $\mathsf{SA}_{\mathsf{XY}}$ by removing all entries that are larger or equal to m. The definition of the Burrows–Wheeler transform extends naturally to the partial version $\mathsf{BWT}_{\mathsf{X}:\mathsf{Y}}[0..m)$.

When comparing two suffixes of XY starting in X, in most cases we only need to access characters in X, but sometimes the comparison continues beyond the end of X and may, in an extreme case, continue all the way to the end of Y. To avoid such long comparisons, we store additional information about the order of the suffixes in the form of bitvectors $\mathsf{gt}^{\mathsf{S}}_{\mathsf{X}:\mathsf{Y}}[0..m)$ defined as follows:

$$\mathsf{gt}^{\mathsf{S}}_{\mathsf{X}:\mathsf{Y}}[i] = \begin{cases} 1 & \text{if } \mathsf{X}[i..m)\mathsf{Y} > \mathsf{S} \\ 0 & \text{if } \mathsf{X}[i..m)\mathsf{Y} \le \mathsf{S} \end{cases} .$$

For example, for $0 \le i < j < m$, the following are equivalent:

1. $\mathsf{X}[i..m)\mathsf{Y} < \mathsf{X}[j..m)\mathsf{Y}$
2. $\mathsf{X}[i..m) < \mathsf{X}[j..m)\mathsf{Y}[0..j-i)$ or $\mathsf{X}[i..m) = \mathsf{X}[j..m)\mathsf{Y}[0..j-i)$ and $\mathsf{gt}^{\mathsf{Y}}_{\mathsf{Y}:\varepsilon}[j-i] = 1$
3. $\mathsf{X}[i..m-j+i) < \mathsf{X}[j..m)$ or $\mathsf{X}[i..m-j+i) = \mathsf{X}[j..m)$ and $\mathsf{gt}^{\mathsf{Y}}_{\mathsf{X}:\mathsf{Y}}[m-j+i] = 0$.

3 Merging of Partial SAs

The basic building block of pSAscan is a procedure for merging two adjacent partial suffix arrays. In this section, we describe a sequential algorithm for performing the merging and then, in the next section, show how to parallelize it.

Given the partial suffix arrays $\mathsf{SA}_{\mathsf{X:YZ}}$ and $\mathsf{SA}_{\mathsf{Y:Z}}$, for some strings X, Y and Z, the task is to construct the partial suffix array $\mathsf{SA}_{\mathsf{XY:Z}}$. The suffixes in each input array stay in the same relative order in the output, and thus we just need to know how to interleave the input arrays. For this purpose, we compute the *gap array* $\mathsf{gap}_{\mathsf{X:Y:Z}}[0..|\mathsf{X}|]$, where $\mathsf{gap}_{\mathsf{X:Y:Z}}[i]$ is the number of suffixes in $\mathsf{SA}_{\mathsf{Y:Z}}$ that are lexicographically between the suffixes $\mathsf{SA}_{\mathsf{X:YZ}}[i-1]$ and $\mathsf{SA}_{\mathsf{X:YZ}}[i]$. Formally, denoting $m = |\mathsf{X}|$ and $n = |\mathsf{Y}|$,

$$\mathsf{gap}_{\mathsf{X:Y:Z}}[0] = \left|\{j \in [0..n) : \mathsf{Y}[j..n)\mathsf{Z} < \mathsf{X}[\mathsf{SA}_{\mathsf{X:YZ}}[0]..m)\mathsf{YZ}\}\right|$$
$$\mathsf{gap}_{\mathsf{X:Y:Z}}[m] = \left|\{j \in [0..n) : \mathsf{X}[\mathsf{SA}_{\mathsf{X:YZ}}[m-1]..m)\mathsf{YZ} < \mathsf{Y}[j..n)\mathsf{Z}\}\right|$$

and, for $i \in [1..m)$,

$$\mathsf{gap}_{\mathsf{X:Y:Z}}[i] = \big|\{j \in [0..n) : \mathsf{X}[\mathsf{SA}_{\mathsf{X:YZ}}[i-1]..m)\mathsf{YZ} < \mathsf{Y}[j..n)\mathsf{Z} < \mathsf{X}[\mathsf{SA}_{\mathsf{X:YZ}}[i]..m)\mathsf{YZ}\}\big|.$$

Given the gap array, the actual merging is easy; the difficult part is computing the gap array.

For a string S, let $\mathsf{sufrank}_{\mathsf{X:YZ}}(\mathsf{S})$ be the number of suffixes in $\mathsf{SA}_{\mathsf{X:YZ}}$ that are lexicographically smaller than S. In other words, if $\mathsf{sufrank}_{\mathsf{X:YZ}}(\mathsf{S}) = k$ (and $0 < k < m$), then $\mathsf{X}[\mathsf{SA}_{\mathsf{X:YZ}}[k-1]..m)\mathsf{YZ} < \mathsf{S} \leq \mathsf{X}[\mathsf{SA}_{\mathsf{X:YZ}}[k]..m)\mathsf{YZ}$. Thus we can compute the gap array $\mathsf{gap}_{\mathsf{X:Y:Z}}$ by initializing all entries to zeros, and then, for all $j \in [0..n)$, computing $k = \mathsf{sufrank}_{\mathsf{X:YZ}}(\mathsf{Y}[j..n)\mathsf{Z})$ and incrementing $\mathsf{gap}_{\mathsf{X:Y:Z}}[k]$. The values $\mathsf{sufrank}_{\mathsf{X:YZ}}(\mathsf{Y}[j..n)\mathsf{Z})$ are computed starting from the end of Y using a procedure called *backward search* [11].

Backward search is based on rank operations on the Burrows–Wheeler transform $\mathsf{BWT}_{\mathsf{X:YZ}}$. For a character c and an integer $i \in [0..m]$, the answer to the rank query $\mathsf{rank}_{\mathsf{BWT}_{\mathsf{X:YZ}}}(c, i)$ is the number of occurrences of c in $\mathsf{BWT}_{\mathsf{X:YZ}}[0..i)$. We preprocess $\mathsf{BWT}_{\mathsf{X:YZ}}[0..m)$ so that arbitrary rank queries can be answered quickly; see [15] for details. Let $\mathsf{C}[0..\sigma)$ be an array, where $\mathsf{C}[c]$ is the number of positions $i \in [0..m)$ such that $\mathsf{X}[i] < c$. The following lemma shows one step of backward search.

Lemma 1. [11,15]. *Let* $k = \mathsf{sufrank}_{\mathsf{X:YZ}}(\mathsf{S})$ *for a string* S. *For any symbol* c*,*

$$\mathsf{sufrank}_{\mathsf{X:YZ}}(c\mathsf{S}) = \mathsf{C}[c] + \mathsf{rank}_{\mathsf{BWT}_{\mathsf{X:YZ}}}(c, k) + \begin{cases} 1 & \textit{if } \mathsf{X}[m-1] = c \textit{ and } \mathsf{YZ} < \mathsf{S} \\ 0 & \textit{otherwise} \end{cases}.$$

Note that when $\mathsf{S} = \mathsf{Y}[j..n)\mathsf{Z}$, we can replace the comparison $\mathsf{YZ} < \mathsf{S}$ with $\mathsf{gt}^{\mathsf{YZ}}_{\mathsf{Y:Z}}[j] = 1$. Thus, given $\mathsf{sufrank}_{\mathsf{X:YZ}}(\mathsf{Y}[j..n)\mathsf{Z})$, we can easily compute $\mathsf{sufrank}_{\mathsf{X:YZ}}$ $(\mathsf{Y}[j-1..n)\mathsf{Z})$ using the lemma, and we only need to access $\mathsf{Y}[j-1]$ and $\mathsf{gt}^{\mathsf{YZ}}_{\mathsf{Y:Z}}[j]$.

Hence the whole computation of $\mathsf{gap}_{\mathsf{X:Y:Z}}$ can be done with a single sequential pass over Y and $\mathsf{gt}^{\mathsf{YZ}}_{\mathsf{Y:Z}}$.

Improvements to SAscan. The procedure described above is identical to the one in the original SAscan [15], but the rest of this section describes details that differ from (and improve) the original.

First, we need $\mathsf{BWT}_{\mathsf{X:YZ}}$ and $\mathsf{gt}^{\mathsf{YZ}}_{\mathsf{Y:Z}}$ for the gap array computation. In SAscan, these are computed from the strings and the partial suffix arrays as needed. This is easy and takes only linear time but is relatively expensive in practice because of frequent cache misses. We compute them differently based on the assumption that both the BWT and the bitvector are available for every partial suffix array. That is, we assume that we are given $\mathsf{BWT}_{\mathsf{X:YZ}}$, $\mathsf{BWT}_{\mathsf{Y:Z}}$, $\mathsf{gt}^{\mathsf{XYZ}}_{\mathsf{X:YZ}}$ and $\mathsf{gt}^{\mathsf{YZ}}_{\mathsf{Y:Z}}$ as input, and we need to compute $\mathsf{BWT}_{\mathsf{XY:Z}}$ and $\mathsf{gt}^{\mathsf{XYZ}}_{\mathsf{XY:Z}}$ as output. Each BWT is stored interleaved with the corresponding SA so that the merging of the SAs produces the output BWT at almost no additional cost. The output bitvector $\mathsf{gt}^{\mathsf{XYZ}}_{\mathsf{XY:Z}}$ is constructed by concatenating the two bitvectors $\mathsf{gt}^{\mathsf{XYZ}}_{\mathsf{X:YZ}}$ and $\mathsf{gt}^{\mathsf{XYZ}}_{\mathsf{Y:Z}}$. The former was given as an input and the latter is computed (as in SAscan) during the backward search using the fact that $\mathsf{gt}^{\mathsf{XYZ}}_{\mathsf{Y:Z}}[j] = 1$ iff $\mathsf{sufrank}_{\mathsf{X:YZ}}(\mathsf{Y}[j..n]\mathsf{Z}) > i_{\mathsf{XYZ}}$, where i_{XYZ} is the position of XYZ in $\mathsf{SA}_{\mathsf{X:YZ}}$, i.e., $\mathsf{SA}_{\mathsf{X:YZ}}[i_{\mathsf{XYZ}}] = 0$.

Second, we need to know $\mathsf{sufrank}_{\mathsf{X:YZ}}(\mathsf{Z})$ as the starting position of the backward search. We replace the $\mathrm{O}(m + n)$ time string range matching [18] used in SAscan by a binary search over $\mathsf{SA}_{\mathsf{X:YZ}}$ with Z as the query. A plain binary search needs $\mathrm{O}(\ell \log m)$ time, where ℓ is the length of the longest common prefix between Z and any suffix in $\mathsf{SA}_{\mathsf{X:YZ}}$. This is fast enough in most cases as ℓ is typically small and the constant factors are small. However, we employ several techniques to ensure a good performance even in pathological cases. We use a string binary search algorithm with $\mathrm{O}(\ell + \log m)$ average case time (see [24]) and $\mathrm{O}(\ell \log_{\ell} m)$ worst case time (see [2] for an even better complexity); we utilize the gt-bitvectors to resolve comparisons early; and, in the full algorithm with many binary searches, we utilize the fact that all the strings are suffixes of the same text. We omit the details here due to lack of space, and because most of the advanced binary searching techniques are only used in pathological cases and have little effect on the experimental results.

The final difference to SAscan is the actual merging of SAs. In SAscan, the merging is delayed (and the gap array is stored on disk) but here we often need to perform the merging immediately. This is easily done if given a separate array for the output, but we want to do the merging *almost in-place* to reduce space usage. The basic idea, following [19, Appendix B], is to divide the SAs into small blocks, which we call *pages*, and maintain pointers to the pages in an additional array, called the *page index*. Any random access has to go through the page index, which allows us to relocate the pages independently. We assume that both the input SAs and the output SA are stored in this form. As merging proceeds and elements are moved from input to output, input pages that become empty are reused as output pages. This way merging can be performed using only a constant number of extra pages.

4 Parallel Merging of Partial SAs

In this section, we describe a parallelized implementation for the merging procedure. We assume a multicore architecture capable of running p threads simultaneously and a shared memory large enough to hold all the data structures.

The first task during merging is the construction of the rank data structure, which is easily parallelized since the data structure is naturally divided into (almost) independent blocks (see [15]).

The most expensive part of merging is the backward search, mainly because the rank queries are relatively expensive (see [15]). We parallelize it by starting the backward search in p places simultaneously. That is, we divide Y into p blocks of equal size and perform a separate backward search for each block in parallel. Each thread computes its own starting $\mathsf{sufrank}$ value by a binary search, and then the repeated computation of the $\mathsf{sufrank}$ values parallelizes trivially.

For each $\mathsf{sufrank}$ value computed during the backward search, we need to increment the corresponding entry in the gap array, but we cannot allow multiple threads to increment the same entry simultaneously, and guarding the gap array by locks would make the updates too expensive. Instead, each thread collects the $\mathsf{sufrank}$ values into a buffer. When the buffer is full, it is sorted and stored onto a queue. A separate thread takes the full buffers from the queue one at a time, divides the buffer into up to p parts and starts a thread for each part to do the corresponding gap array updates. Since the buffer is sorted, two threads can never try to increment the same gap array entry.

Once the gap array has been constructed, we still need to perform the actual merging. Recall that we assume the paged storage for the SA. We divide the output SA into p blocks, with the block boundaries always at the page boundaries, and assign a thread for each block. Each thread then finds the corresponding ranges in the input SAs using the gap array. The gap array has been preprocessed by computing cumulative sums at p equally spaced positions, so that the input ranges can be determined by scans of length $\mathrm{O}(n/p)$ over the gap array. Next each thread performs the merging using the sequential almost-in-place procedure described in the previous section. The pages containing the beginning and the end of each input range might be shared with another thread and those pages are treated as read-only. Other input pages and all output pages are exclusive to a single thread. Thus each thread needs only four extra pages to do its part of the merging. Once all threads have finished, the extra pages can be relocated to the input boundary pages.

The whole merging procedure can be performed in $\mathrm{O}((m + t_{\mathrm{rank}}n)/p)$ time, where t_{rank} is the time for performing one rank query. The input is overwritten by the output, and significant additional space is needed only for the rank structure, the gap array, the extra $4p$ pages and the page indexes. Using the representations from [15], the first two need about $(4.125 + 1)m$ bytes. If we choose page size $\Theta(\sqrt{n/p})$, the space needed for the latter two is $\Theta(\sqrt{np})$, which is negligible. Assuming one byte characters and five byte SA entries, the input/output itself needs about $7.125(m + n)$ bytes (text, SA, BWT and gt bitvectors). The total is $12.25m + 7.125n$ bytes (plus the $\Theta(\sqrt{np})$ bytes).

5 Parallel SA Construction

In this section, we extend the parallel merging procedure into a full parallel suffix array construction algorithm. As before, we assume a multicore architecture with p threads and a shared memory large enough for all the data structures.

The basic idea is simple: divide the input string of length n into p blocks of size $m = \lceil n/p \rceil$, construct the partial SAs for the blocks in parallel using a sequential suffix array construction algorithm, and then keep merging the partial SAs using the parallel merging procedure until the full SA is obtained.

We construct the block SAs using Yuta Mori's divsufsort [25], possibly the fastest sequential suffix sorting algorithm in practice, but we could use any other algorithm too. Let X be a block, Y the following block, and Z the full suffix starting after Y. To obtain the partial suffix array $\mathsf{SA}_{\mathsf{X}:\mathsf{YZ}}$ instead of the full suffix array SA_{X}, we construct a string $\widehat{\mathsf{X}}$ such that $\mathsf{SA}_{\widehat{\mathsf{X}}} = \mathsf{SA}_{\mathsf{X}:\mathsf{YZ}}$, and for this we need the bitvector $\mathsf{gt}^{\mathsf{YZ}}_{\mathsf{X}:\mathsf{YZ}}$, which we denote by gt_{X} for brevity. For further details of the construction, we refer to [15], but the computation of gt_{X} is different. We first compute $\widetilde{\mathsf{gt}}_{\mathsf{X}} = \mathsf{gt}^{\mathsf{Y}}_{\mathsf{X}:\mathsf{Y}}$ in $\mathrm{O}(m)$ time. During the computation, we identify and mark the positions i, where $\mathsf{X}[i..m)\mathsf{Y}[0..m-i) = \mathsf{Y}$; we call these *undecided positions*. It is easy to see that if $\widetilde{\mathsf{gt}}_{\mathsf{X}}[i] \neq \mathsf{gt}_{\mathsf{X}}[i]$, then i must be an undecided position. Furthermore, in that case $\mathsf{gt}_{\mathsf{X}}[i] = \mathsf{gt}_{\mathsf{Y}}[i]$. Thus, if i is an undecided position in $\widetilde{\mathsf{gt}}_{\mathsf{X}}$, it depends on $\widetilde{\mathsf{gt}}_{\mathsf{Y}}[i]$. If that too is undecided, it depends on the position i in the next block and so on. Thus, given the $\widetilde{\mathsf{gt}}$-bitvectors for all blocks, we can decide all the undecided i-positions in them in $\mathrm{O}(p)$ time. Deciding all undecided positions requires $\mathrm{O}(pm)$ work and $\mathrm{O}(m+p)$ time using p threads.

Let X be a block and Z the suffix starting after the block. Given $\mathsf{SA}_{\mathsf{X}:\mathsf{Z}}$, we can easily compute $\mathsf{BWT}_{\mathsf{X}:\mathsf{Z}}$ and $\mathsf{gt}^{\mathsf{XZ}}_{\mathsf{X}:\mathsf{Z}}$ as well as the page index for $\mathsf{SA}_{\mathsf{X}:\mathsf{Z}}$ in $\mathrm{O}(m)$ time in preparation for the merging phase. Furthermore, we compute $\mathrm{O}(p^2)$ $\mathsf{sufrank}$ values by binary searches (the suffixes starting at the block boundaries against the block SAs); these are used to ensure fast binary searches later during the merging. The worst case complexity of these binary searches is $\mathrm{O}(np)$ work and $\mathrm{O}(n)$ time, i.e., it does not scale with p. We have designed theoretically better ways of computing the $\mathsf{sufrank}$ values, but binary searching is better in practice because of small constant factors and because it is almost always much faster than the worst case. In all our experiments in Sect. 7, the binary searches never took more than 1.5 % of the total time, and even in the very worst case (a unary string) it takes less than 25 % of the total time.

To obtain the final SA from the p initial block SAs, we have to do $p-1$ pairwise merges. If we do the merges in a balanced manner, each element is involved in about $\log p$ merges, and the total time complexity is $\mathrm{O}((t_{\mathrm{rank}} n \log p)/p)$ for a string of length n. Surprisingly, doing the merges in a balanced manner is not necessarily the optimal way. The time for a single merge can be approximated by $a\ell + br$, where ℓ is the size of the left-hand block, r is the size of the right-hand block, and a and b are some constants. Because the merging time is dominated by the backward search phase, b is much larger than a both in theory as well as in practice. We have implemented a dynamic programming algorithm for computing the optimal merging schedule given p and the value b/a. For example, in a balanced merging with $p = 8$, a single element is involved in three merges, 1.5 times on the left-hand side and 1.5 times on the right-hand side on average. However, in an optimal merging schedule for $b/a = 4$, the averages are 2.25 times on the left-hand side and 1.125 times on the right-hand side. The optimal schedule is about 10 % faster than the balanced schedule in this case. The actual value of b/a in our experiments is about 7.

The space requirement of the algorithm is maximized during the last merge when it is about $12.25\ell + 7.125r$ bytes (see Sect. 4). The space usage can be controlled to an extent by skewing the merging to favor larger right-hand block. Thus there is a space-time tradeoff, but only for the largest merges. Smaller merges can be optimized for time only. Our dynamic program can compute a time-optimal merging schedule under a constraint on the maximal space usage.

6 Parallel SA Construction in External Memory

In this section, we combine the parallel SA construction described above and the external memory construction described in [15] to obtain a faster external memory algorithm.

The basic idea of the algorithm in [15] is:

1. Divide the text into blocks of size m that are small enough to handle in internal memory.
2. For each block X (from last to first), construct the partial suffix array $\mathsf{SA}_{\mathsf{X:Z}}$ and the gap array $\mathsf{gap}_{\mathsf{X:Z:}\varepsilon}$, where Z is the suffix starting after X.
3. After constructing all the partial SA and gap arrays, merge the SAs in one multiway merge.

The last step is dominated by I/O and does not benefit much from parallelism, but we will describe how the SA and gap array construction are parallelized.

For constructing $\mathsf{SA}_{\mathsf{X:Z}}$, we can use the algorithm of the previous section with minor changes required because we are constructing a *partial* SA and the tail Z is stored on disk. There are two phases affected by this: the construction of the gt bitvectors in the beginning and the computation of $\mathsf{sufrank}$ values before the merging phase. We assume that the bitvector $\mathsf{gt}^{\mathsf{Z}}_{\mathsf{Z:}\varepsilon}$ is stored on disk too, which allows us to limit the access to a prefix of Z (and $\mathsf{gt}^{\mathsf{Z}}_{\mathsf{Z:}\varepsilon}$) of length at most m.

The construction of $\mathsf{gap}_{\mathsf{X:Z:}\varepsilon}$ is done by backward searching Z over the rank data structure on $\mathsf{BWT}_{\mathsf{X:Z}}$ as described in previous sections. The only difference is that Z and $\mathsf{gt}^{\mathsf{Z}}_{\mathsf{Z:}\varepsilon}$ are now on disk, but this is not a problem as only a sequential access is needed. For large files ($n \gg m$), this is by far the most time consuming part because the total number of backward search steps is $\Theta(n^2/m)$. Even with parallelism, the time is dominated by internal memory computation rather than I/O, because rank queries and gap array updates are expensive and the I/O volume per step is low. Thus the parallelism achieves a great speed-up compared to the sequential version.

Table 1. The memory usage of internal memory parallel suffix-sorting algorithms (in bytes). The merging schedule of pSAscan (see Sect. 5) was configured to use $10n$ bytes of RAM in all experiments.

Algor.	pDC3		divsufsort		pSAscan	
	32-bit	64-bit	32-bit	64-bit	32-bit	40-bit
RAM	$21n$	$41n$	$5n$	$9n$	$10n$	$10n$

Table 2. Dataset statistics

Name	$\lvert\mathsf{X}\rvert$	σ
hg.reads	1024 GiB	6
kernel	200 GiB	229
wiki	2 GiB	210
countries	2 GiB	205
skyline	2 GiB	32
random	2 GiB	255

The block size m is chosen to fit necessary data structures in RAM. However, the gap array construction needs only about $5.2m$ bytes but the SA construction needs nearly $10m$ bytes. Therefore we add one more stage to the computation. We choose m so that $5.2m$ bytes fits in RAM, but each block X of size m is split into two halfblocks X_1 and X_2. We first compute the halfblock suffix arrays $\mathsf{SA}_{\mathsf{X}_1:\mathsf{X}_2\mathsf{Z}}$ and $\mathsf{SA}_{\mathsf{X}_2:\mathsf{Z}}$ separately and write them to disk. Next we compute $\mathsf{gap}_{\mathsf{X}_1:\mathsf{X}_2:\mathsf{Z}}$ and use it to merge $\mathsf{BWT}_{\mathsf{X}_1:\mathsf{X}_2\mathsf{Z}}$ and $\mathsf{BWT}_{\mathsf{X}_2:\mathsf{Z}}$ into $\mathsf{BWT}_{\mathsf{X}:\mathsf{Z}}$, which is then used for computing $\mathsf{gap}_{\mathsf{X}:\mathsf{Z}:\varepsilon}$. This approach minimizes the total number of backward search steps. To reduce I/O, $\mathsf{SA}_{\mathsf{X}_1:\mathsf{X}_2\mathsf{Z}}$ and $\mathsf{SA}_{\mathsf{X}_2:\mathsf{Z}}$ are never merged into $\mathsf{SA}_{\mathsf{X}:\mathsf{Z}}$, but all halfblock SAs are merged simultaneously in the final multiway merging stage. For the final merging, we need $\mathsf{gap}_{\mathsf{X}_1:\mathsf{X}_2\mathsf{Z}:\varepsilon}$ and $\mathsf{gap}_{\mathsf{X}_2:\mathsf{Z}:\varepsilon}$, which can be computed quickly and easily from $\mathsf{gap}_{\mathsf{X}_1:\mathsf{X}_2:\mathsf{Z}}$ and $\mathsf{gap}_{\mathsf{X}:\mathsf{Z}:\varepsilon}$.

The disk usage is less than $7.5n$ bytes consisting of the text (n bytes), SAs ($5n$), gap arrays (about n using vbyte-encoding [33]), and a gt-bitvector (n bits).

7 Experimental Results

Setup. We performed experiments on two different machines referred to as Platform S (small) and Platform L (large). Platform S was equipped with a 4-core 3.40 GHz Intel i7-3770 CPU with 8 MiB L2 cache and 16 GiB of DDR3 RAM. Platform L was equipped with two 6-core 1.9 GHz Intel Xeon E5-2420 CPUs (capable, via hyper-threading, of running 24 threads) with 15 MiB L2 cache and 120 GiB of DDR3 RAM. The machine had 7.2 TiB of disk space striped with RAID0 across four identical local disks (achieving a (combined) transfer rate of about 480 MiB/s), and an additional two-disk RAID0 which was used only for the experiment on 1TiB input. The OS was Linux (Ubuntu 12.04, 64 bit). All programs were compiled using `g++` (Cilk Plus branch) version 4.8.1 with `-O2 -DNDEBUG` options.

Datasets. For the experiments we used the following files varying in the number of repetitions and alphabet size (see Table 2 for some statistics):

- hg.reads: a collection of DNA reads (short fragments produced by a sequencing machine) from 40 human genomes[1] filtered from symbols other than {`A`, `C`, `G`, `T`, `N`} and newline;

[1] http://www.1000genomes.org/.

- wiki: a prefix of English Wikipedia dump[2] (dated 20140707) in the XML format;
- kernel: a concatenation of ∼16.8 million source files from 510 recent versions of Linux kernel[3];
- countries: a concatenation of all versions (edit history) of four Wikipedia articles about countries in the XML format. It contains a large number of 1–5 KiB repetitions;
- skyline: an artificial, highly repetitive sequence (see [5] for details);
- random: a randomly generated sequence of bytes.

Experiments. We implemented the pSAscan algorithm in C++ using STL threads for parallelism[4]. In the first experiment we study the performance of pSAscan as a standalone internal-memory suffix sorting algorithm and compare it with the parallel implementation of DC3 algorithm [6], the fastest parallel suffix-sorter in previous studies, and the parallel version of divsufsort [25]. The latter has a parallel mode that (slightly) improves the runtime, but is mostly known as the fastest sequential suffix array construction algorithm. For each algorithm, we included two versions, one using 32-bit integers and limited to 2 GiB or 4 GiB files, and the other capable of processing larger files. The algorithms and their memory usage are summarized in Table 1. For fair comparison pSAscan produces the suffix array as a plain array (rather than in a paged form). This requires an additional permuting step and slightly slows down our algorithm. The results for Platform L are given in Fig. 1. pSAscan is clearly the fastest algorithm when using full parallelism and at least competitive when using less threads. The exception is the random input with a large alphabet (where DC3 excels due to very shallow recursion) and skyline. The poor performance of pSAscan on the skyline testfile is, however, inherited from divsufsort for which it is the worst case input. The relative performance of pDC3 and pSAscan on Platform S (see Fig. 2 for two sample graphs) is similar to Platform L.

In the second experiment we compare the EM version of pSAscan to the best EM algorithms for suffix array construction: eSAIS [5] (with the STXXL library [8] compiled in parallel mode) and SAscan [15] (sequential), using a moderate amount of RAM (3.5 GiB). Results are given in Fig. 3. For smaller files, pSAscan is several times faster than the competitors. For larger files, eSAIS approaches pSAscan and would probably overtake it somewhere around 250–300 GiB files, which coincidentally is about the size for which eSAIS would run out of disk space on the test machine. Using the full 120 GiB RAM moves the crossover point to several terabytes and allowed us to process the full 1TiB instance of hg.reads (see Table 3).

Finally, Table 4 shows that, particularly for large files, the running time of pSAscan is dominated by the gap array construction, which involves $\Theta(n^2/m)$ steps of backward searching.

[2] http://dumps.wikimedia.org/.

[3] http://www.kernel.org/.

[4] The implementation is available at http://www.cs.helsinki.fi/group/pads/.

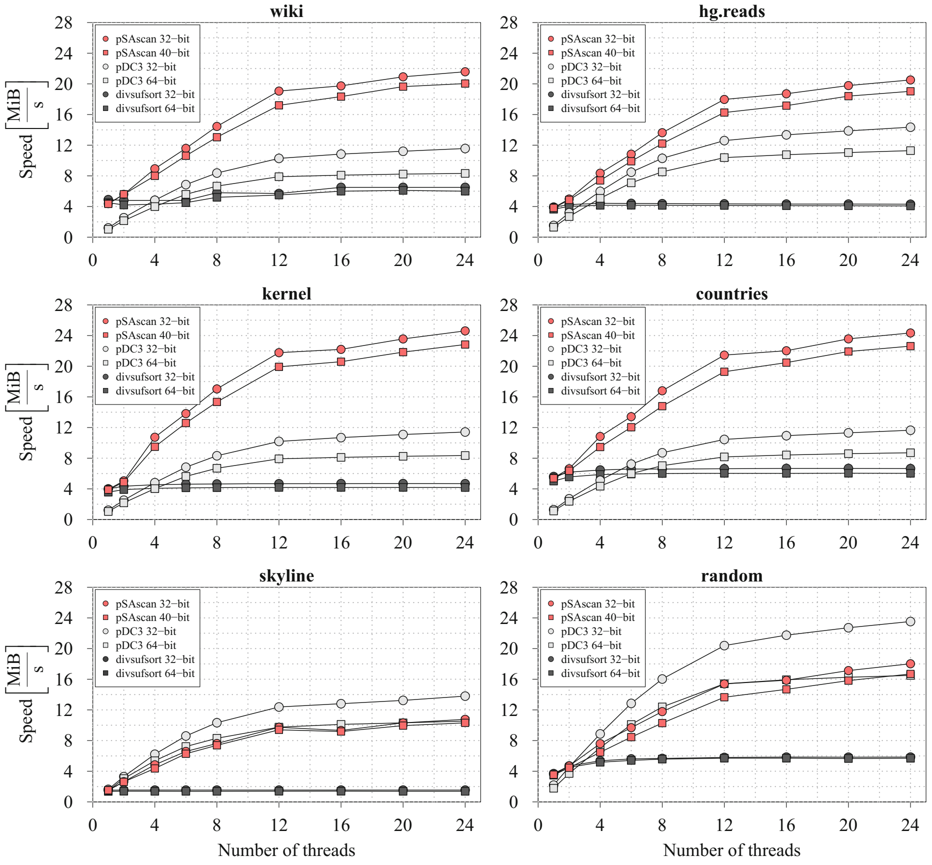

Fig. 1. Internal memory parallel suffix array construction on Platform L. All input files are of size 2 GiB (Color Figure Online).

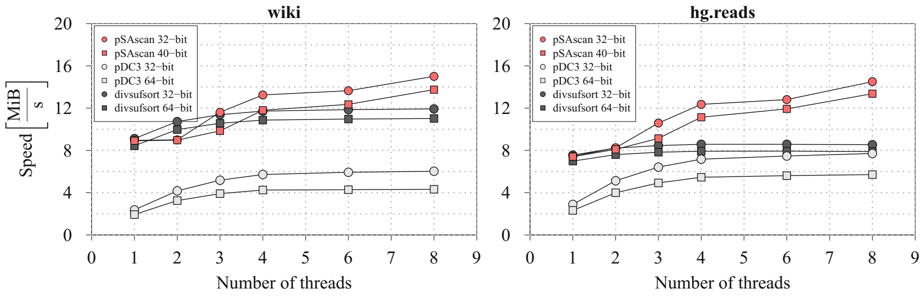

Fig. 2. Internal memory parallel suffix array construction on Platform S. All input files are of size 360 MiB (Color Figure Online).

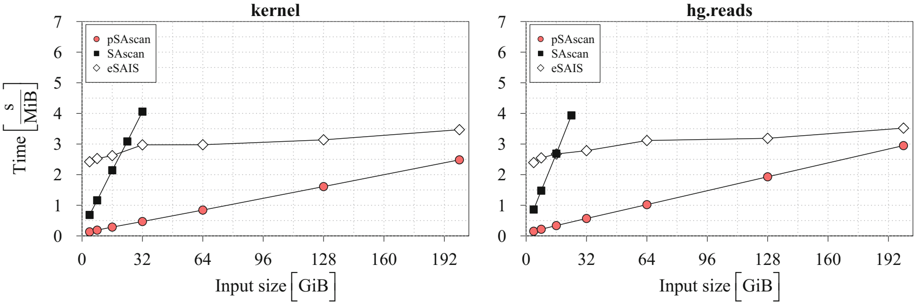

Fig. 3. Scalability of EM version of pSAscan compared to eSAIS and SAscan. All algorithms were allowed to use only 3.5 GiB of RAM for computation. pSAscan and eSAIS were allowed to use the maximum number of threads (24) (Color Figure Online).

Table 3. A performance comparison of eSAIS and pSAscan on prefixes of hg.reads testfile with varying amount of memory available to algorithms. The peak disk space usage includes input and output (which is five times the size of input).

Algorithm	Input size	RAM usage	Runtime	Peak disk usage	I/O volume
eSAIS	200 GiB	3.5 GiB	8.3 days	4.6 TiB	52.0 TiB
	200 GiB	120 GiB	4.1 days	4.6 TiB	36.1 TiB
pSAscan	200 GiB	3.5 GiB	7.0 days	1.4 TiB	43.8 TiB
	200 GiB	120 GiB	0.5 days	1.4 TiB	4.9 TiB
	1024 GiB	120 GiB	8.1 days	7.3 TiB	48.3 TiB

Table 4. A detailed runtime breakdown of external memory pSAscan on the 200GiB instance of hg.reads. The times are given in hours.

RAM usage	Internal memory suffix sort			Gap array construction	Final merge	Other
	I/O	divsufsort	Other			
3.5 GiB	0.4	0.7	2.2	132.4	29.2	1.2
120 GiB	0.6	1.1	2.6	4.3	2.4	0.9

8 Concluding Remarks

When deciding whether an algorithm scales to deal with large inputs, we are principally concerned with three values: RAM, time, and disk usage. The main advantage of pSAscan is that it measures up well on all three of these dimensions. The algorithm is also fairly versatile: for example, it would add little overhead to have it output the BWT in addition to (or instead of) the SA in order to, say, speed up construction of an FM-index.

There are many avenues for future work. Most obviously, one wonders if similar techniques for suffix sorting can be successfully applied to other parallel architectures, such as GPUs and distributed systems. We also believe our merging procedure can find other uses, such as supporting the efficient update of the suffix array when new text is appended to the underlying string.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1), 53–86 (2004)
2. Andersson, A., Hagerup, T., Håstad, J., Petersson, O.: Tight bounds for searching a sorted array of strings. SIAM J. Comput. **30**(5), 1552–1578 (2000)
3. Apostolico, A., Iliopoulos, C.S., Landau, G.M., Schieber, B., Vishkin, U.: Parallel construction of a suffix tree with applications. Algorithmica **3**, 347–365 (1988)
4. Apostolico, A., Lonardi, S.: Off-line compression by greedy textual substitution. Proc. IEEE **88**(11), 1733–1744 (2000)
5. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and LCP arrays in external memory. In: Sanders, P., Zeh, N. (eds.) ALENEX 2013. pp. 88–102. SIAM (2013)
6. Blelloch, G.E., Shun, J.: A simple parallel cartesian tree algorithm and its application to suffix tree construction. In: Müller-Hannemann, M., Werneck, R.F.F. (eds.) ALENEX 2011, pp. 48–58. SIAM (2011)
7. Crauser, A., Ferragina, P.: A theoretical and experimental study on the construction of suffix arrays in external memory. Algorithmica **32**(1), 1–35 (2002)
8. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw. Pract. Exper. **38**(6), 589–637 (2008)
9. Deo, M., Keely, S.: Parallel suffix array and least common prefix for the GPU. In: Nicolau, A., Shen, X., Amarasinghe, S.P., Vuduc, R.W. (eds.) PPoPP 2013, pp. 197–206. ACM (2013)
10. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM **47**(6), 987–1011 (2000)
11. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
12. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica **63**(3), 707–730 (2012)
13. Gonnet, G.H., Baeza-Yates, R.A., Snider, T.: New indices for text: pat trees and pat arrays. In: Frakes, W.B., Baeza-Yates, R. (eds.) Information Retrieval: Data Structures and Algorithms, pp. 66–82. Prentice-Hall, Englewood Cliffs (1992)
14. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005)
15. Kärkkäinen, J., Kempa, D.: Engineering a lightweight external memory suffix array construction algorithm. In: Iliopoulos, C.S., Langiu, A. (eds.) ICABD 2014, CEUR Workshop Proceedings, vol. 1146, pp. 53–60 (2014). CEUR-WS.org
16. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Linear time Lempel-Ziv factorization: simple, fast, small. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 189–200. Springer, Heidelberg (2013)
17. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lempel-Ziv parsing in external memory. In: Bilgin, A., Marcellin, M.W., Serra-Sagristà, J., Storer, J.A. (eds.) DCC 2014, pp. 153–162. IEEE (2014)
18. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: String range matching. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) CPM 2014. LNCS, vol. 8486, pp. 232–241. Springer, Heidelberg (2014)

19. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006)
20. Kulla, F., Sanders, P.: Scalable parallel suffix array construction. Parallel Comput. **33**(9), 605–612 (2007)
21. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biol. **10**(3), R25 (2009)
22. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009)
23. Louza, F.A., Telles, G.P., Ciferri, C.D.D.A.: External memory generalized suffix and LCP arrays construction. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 201–210. Springer, Heidelberg (2013)
24. Manber, U., Myers, G.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
25. Mori, Y.: libdivsufsort, a C library for suffix array construction. http://code.google.com/p/libdivsufsort/
26. Nakamura, R., Inenaga, S., Bannai, H., Funamoto, T., Takeda, M., Shinohara, A.: Linear-time text compression by longest-first substitution. Algorithms **2**(4), 1429–1448 (2009)
27. Nong, G., Chan, W.H., Zhang, S., Guan, X.F.: Suffix array construction in external memory using d-critical substrings. ACM Trans. Inf. Syst. **32**(1), 1 (2014)
28. Osipov, V.: Parallel suffix array construction for shared memory architectures. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 379–384. Springer, Heidelberg (2012)
29. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. **39**(2), 31 (2007). Article 4
30. Simpson, J.T., Durbin, R.: Efficient de novo assembly of large genomes using compressed data structures. Genome Res. **22**(3), 549–556 (2012)
31. Tischler, G.: Faster average case low memory semi-external construction of the Burrows-Wheeler transform. In: Iliopoulos, C.S., Langiu, A. (eds.) ICABD 2014, CEUR Workshop Proceedings, vol. 1146, pp. 61–68 (2014). CEUR-WS.org
32. Weiner, P.: Linear pattern matching algorithms. In: SWAT 1973, pp. 1–11. IEEE (1973)
33. Williams, H.E., Zobel, J.: Compressing integers for fast file access. Comput. J. **42**(3), 193–201 (1999)
34. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theor. **23**(3), 337–343 (1977)