Parallel Suffix Array Construction by Accelerated Sampling*

Matthew Felice Pace and Alexander Tiskin

DIMAP and Department of Computer Science
University of Warwick, Coventry, CV4 7AL, UK
matthewfp,tiskin@dcs.warwick.ac.uk

Abstract. A deterministic BSP algorithm for constructing the suffix array of a given string is presented, based on a technique that we call *accelerated sampling*. It runs in optimal $O(\frac{n}{p})$ local computation and communication, and requires a near optimal $O(\log \log p)$ supersteps. The algorithm provides an improvement over the synchronisation costs of existing algorithms, and reinforces the importance of the sampling technique.

Keywords: BSP, suffix array, accelerated sampling

1 Introduction

Suffix arrays are a fundamental data structure in the string processing field and have been researched extensively since their introduction by Manber and Myers [11,14].

Definition 1. *Given a string $x = x[0] \cdots x[n-1]$ of length $n \geq 1$, defined over an alphabet Σ, the suffix array problem is that of constructing the suffix array $SA_x = SA_x[0] \cdots SA_x[n-1]$ of x, which holds the ordering of all the suffixes $s_i = x[i] \cdots x[n-1]$ of x in ascending lexicographical order; i.e. $SA_x[j] = i$ iff s_i is the j^{th} suffix of x in ascending lexicographical order.*

1.1 Notation, Assumptions and Restrictions

We assume zero-based indexing throughout the paper, and that the set of natural numbers $\mathbb{N}$ includes zero. For any $i, j \in \mathbb{N}$, we use the notation $[i : j]$ to denote the set $\{a \in \mathbb{N} \mid i \leq a \leq j\}$, and $[i : j)$ to denote $\{a \in \mathbb{N} \mid i \leq a < j\}$. This notation is extended to substrings by denoting the substrings of string x, of size n, by $x\,[i : j) = x[i] \cdots x[j-1]$, for $i \in [0 : n)$ and $j > i$.

The input to the algorithms to be presented in this paper is restricted to strings defined over the alphabet $\Sigma = [0 : n)$, where n is the size of the input string. This allows us to use counting sort [3] throughout when sorting characters, in order to keep the running time linear in the size of the input. We also use counting sort in conjunction with the radix sorting technique [3].

The end of any string is assumed to be marked by an end sentinel, typically denoted by '$', that precedes all the characters in the alphabet order. Therefore, to mark the end of the string and to ensure that any substring $x\,[i : j)$ is well defined, we adopt the padding convention $x[k] = -1$, for $k \geq n$.

* Research supported by the Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, EPSRC award EP/D063191/1

Matthew Felice Pace, Alexander Tiskin: Parallel Suffix Array Construction by Accelerated Sampling, pp. 142–156.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0

Note that the suffix array construction algorithms to be presented in Sections 3 and 5 can also be applied to any string X, of size n, over an indexed alphabet Σ' [14,16], which is defined as follows:

- Σ' is a totally ordered set.
- an array A can be defined, such that, $\forall \sigma \in \Sigma'$, $A[\sigma]$ can be accessed in constant time.
- $|\Sigma'| \leq n$.

Commonly used indexed alphabets include the ASCII alphabet and the DNA bases. It should also be noted that any string X, of size n, over a totally ordered alphabet can be encoded as a string X', of size n, over integers. This is achieved by sorting the characters of X, removing any duplicates, and assigning a rank to each character. The string X' is then constructed, such that it is identical to X except that each character of X is replaced by its rank in the sorted array of characters. However, sorting the characters of X could require $O(n \log n)$ time, depending on the nature of the alphabet over which X is defined.

Let $x_1 \odot x_2$ denote the concatenation of strings x_1 and x_2. Then, for any set of integers A, $\bigodot_{i \in A} x_i$ is the concatenation of the strings indexed by the elements of A, in ascending index order. Throughout the paper we use $|b|$ to denote the size of an array or string b. To omit $\lceil \cdot \rceil$ operations, we assume that real numbers are rounded up to the nearest integer.

1.2 Problem Overview

As previously stated, the suffix array problem is that of constructing the suffix array of a given string. The example in Table 1 shows the suffix array of string X, of size 12, over an indexed alphabet of a subset of the ASCII characters, written as string X' over $\Sigma = [0 : 12)$.

	0	1	2	3	4	5	6	7	8	9	10	11	12
$X =$	a	c	b	a	a	c	e	d	b	b	e	a	\$
$X' =$	0	2	1	0	0	2	4	3	1	1	4	0	-1
$SA_X =$	11	3	0	4	2	8	9	1	5	7	10	6	

Table 1. Suffix array of a string X over an indexed alphabet, written as string X' over $\Sigma = [0 : 12)$

The problem is, by definition, directly related to the sorting problem. In fact, if all the characters of the input string are distinct, then the suffix array is obtained by sorting the strings' characters and returning the indices of the characters in their sorted order. In general, if the characters of the string are not distinct, the naive solution is to radix sort all the suffixes, which runs in $O(n^2)$ time if counting sort is used to sort the characters at each level of the radix sort. However, numerous algorithms exist that improve on this. The first such algorithm was presented by Manber and Myers [11] and required $O(n \log n)$ time. The running time was reduced to $O(n)$ through three separate algorithms by Kärkkäinen and Sanders [4], Kim et al. [7], and Ko and Aluru [8]. A number of other algorithms exist with a higher theoretical worst case running time but faster running time in practice, as discussed in [14]. However, the study of these is beyond the scope of this work.

The idea behind the algorithms having linear theoretical worst case running time is to use recursion as follows:

1. Divide the indices of the input string x into two nonempty disjoint sets. Form strings x' and y' from the characters indexed by the elements of each set. Recursively construct $SA_{x'}$.
2. Use $SA_{x'}$ to construct $SA_{y'}$.
3. Merge $SA_{x'}$ and $SA_{y'}$ to obtain SA_x.

The aim of this paper is to investigate the suffix array problem in the Bulk Synchronous Parallel (BSP) model, on a p processor distributed memory system. As in the sequential setting, the naive solution to the problem is to radix sort all the suffixes of the string. Shi and Shaeffer [15] provide a comparison based parallel sorting algorithm, using a technique known as regular sampling that is then adapted by Chan and Dehne [1] for integer sorting. However, using such a technique to sort the suffixes of a given string of size n leads to a parallel algorithm with $O(\frac{n^2}{p})$ local computation cost, which is clearly inefficient.

Kärkkäinen et al. [5] give a brief overview of a BSP suffix array construction algorithm having optimal $O(\frac{n}{p})$ local computation and communication costs and requiring $O(\log^2 p)$ supersteps. They also present similar algorithms for various computation models including the PRAM model. Kulla and Sanders [9] show that the BSP algorithm presented in [5] requires $O(\log p)$ supersteps and discuss their experimental evaluation of the algorithm.

In this paper we reduce the number of supersteps required to a near optimal $O(\log \log p)$, while keeping the local computation and communication costs optimal. The algorithm is based on a technique that we call *accelerated sampling*. This technique was introduced (without a name) by Tiskin [18] for the parallel selection problem. An accelerated sampling algorithm is a recursive algorithm that samples the data at each level of recursion, changing the sampling frequency at a carefully chosen rate as the algorithm progresses.

1.3 Paper Structure

The rest of the paper is structured as follows. The next section provides an overview of the concept of difference covers. A description of the sequential suffix array construction algorithm presented in [5] is given in Section 3. This is a generalised version of the algorithm of [4], which is known as the DC3 algorithm. An overview of the BSP model is provided in Section 4. In Section 5 we present our parallel suffix array construction algorithm, based on the accelerated sampling technique, building on top of the detailed algorithm description given in Section 3. A detailed description is given since, as opposed to the parallel DC3 algorithms of [5,9], we do not assume a fixed input parameter $v = 3$ in all the levels of recursion, so a more general version of the algorithm is required. A detailed analysis of our proposed parallel suffix array construction algorithm in the BSP model is then presented. The last section offers some concluding views and discusses possible future work.

2 Difference Covers

The suffix array construction algorithms to be presented in this paper make use of the concept of difference covers [2,6,13]. Given a positive integer v, let $\mathbb{Z}_v$ denote the set of integers $[0 : v)$. A set $D \subseteq \mathbb{Z}_v$ can be defined such that for any $z \in \mathbb{Z}_v$, there exist $a, b \in D$ such that $z \equiv a - b \pmod{v}$. Such a set D is known as a *difference*

v	$5 \dots 13$	$14 \dots 73$	$74 \dots 181$	$182 \dots 337$	$338 \dots 541$	$\dots$	1024	$\dots$	2048	$\dots$
$\lvert D_v \rvert$	4	10	16	22	28		40		58	

Table 2. Size of the difference cover obtained using the algorithm in [2] for various values of v

cover of $\mathbb{Z}_v$, or *difference cover modulo* v. For example, $\{0,1,2\}$, $\{0,1,3\}$, $\{0,2,3\}$ and $\{1,2,3\}$ are valid difference covers of $\mathbb{Z}_4$ while no other proper subset of $\mathbb{Z}_4$ is.

Colbourn and Ling [2] present a method for obtaining, for any v, a difference cover D of $\mathbb{Z}_v$ in time $O(\sqrt{v})$, where $|D| = 6r+4$, $r = \frac{-36+\sqrt{48+96v}}{48}$. Hence, $|D| \leq \sqrt{1.5v}+6$. Note that, in general, for any v and any difference cover D of $\mathbb{Z}_v$, $|D| \geq \frac{1+\sqrt{4v-3}}{2}$, since we must have $|D|(|D|-1)+1 \geq v$. Therefore, the size of the difference cover obtained by using the algorithm in [2] is optimal up to a multiplicative constant.

For technical reasons, discussed in Section 3, the algorithms to be presented in this paper require that $0 \notin D$. This does not represent a restriction since, for any v and difference cover D of $\mathbb{Z}_v$, for all $z \in \mathbb{Z}_v$ the set $D' = \{(d-z) \bmod v \mid d \in D\}$ is also a difference cover of $\mathbb{Z}_v$ (see e.g. [13]).

Furthermore, we require that $|D| < v$. Since the minimum size of a difference cover constructed using the method of [2] is 4, we only use this method for $v \geq 5$. For $v = 3$ and $v = 4$ we use the difference covers $\{1,2\}$ and $\{1,2,3\}$ respectively. Table 2 shows the size of the difference cover obtained using the algorithm of [2] for various values of v.

The following simple lemma is also required to ensure the correctness of the algorithms to be presented.

Lemma 2. **[5]** *If D is a difference cover of $\mathbb{Z}_v$, and i and j are integers, then there exists $l \in [0:v)$ such that $(i+l) \bmod v$ and $(j+l) \bmod v$ are both in D.*

For any difference cover D of $\mathbb{Z}_v$ and integer $n \geq v$, a *difference cover sample* is defined as $C = \{i \in [0:n) \mid i \bmod v \in D\}$. The index set C is a v-periodic sample of $[0:n)$, as defined in [5]. The fact that difference cover samples are periodic allows them to be used for efficient suffix sorting on a given string.

3 Sequential Algorithm

Kärkkäinen and Sanders [4] present a sequential recursive algorithm that constructs the suffix array of a given string x, of size n, using the difference cover $\{1,2\}$ of $\mathbb{Z}_3$, in time $O(n)$. This algorithm is generally known as the DC3 algorithm. Kärkkäinen et al. [5] then generalise the DC3 algorithm such that the suffix array of x can be constructed using a difference cover D of $\mathbb{Z}_v$, for any arbitrary choice of $v \in [3:n]$, in time $O(vn)$. Clearly, setting $v = 3$ results in a running time of $O(n)$, with a small multiplicative constant. As v approaches n the running time approaches $O(n^2)$, and when $v = n$ the algorithm is simply a complex version of the naive suffix array construction algorithm. However, by initially letting $v = 3$ and increasing the value of v at a carefully chosen rate in every subsequent level of recursion, we can reduce the total number of recursion levels required for the algorithm to terminate, while still keeping the total running time linear in the size of the input string. This technique can be used to decrease the number of supersteps required by the parallel suffix array construction algorithm in the BSP model. This is discussed further in Section 5. The detailed sequential algorithm presented in [5] proceeds as follows:

Algorithm 1. *Sequential Suffix Array Construction*
Parameters: integer n; integer $v \in [3 : n]$.
Input: string $x = x[0] \cdots x[n-1]$ over alphabet $\Sigma = [0 : n)$.
Output: suffix array $SA_x = SA_x[0] \cdots SA_x[n-1]$.
Description:

Recursion base

We sort x using counting sort, in time $O(n)$. If all the characters of x are distinct we return, for each character, in the sorted order, the index of the character in x, i.e. SA_x. Otherwise, the following stages are performed:

Stage 0 - Sample construction and initialisation

Construct the difference cover D of $\mathbb{Z}_v$ as discussed in Section 2. Then, for each $k \in [0 : v)$, define the set $B_k = \{i \in [0 : n) \mid i \bmod v = k\}$. This partitions the set of indices of x into v sets of size about $\frac{n}{v}$. The difference cover sample $C = \bigcup_{k \in D} B_k$ is then constructed. For $i \in C$, we call the characters $x[i]$ *sample characters* and the suffixes s_i *sample suffixes*. We denote by S_k, $k \in [0 : v)$, the set of suffixes s_i, $i \in B_k$.

Furthermore, an array $rank$ of size $n + v$ is declared and initialised by $rank[0] = \ldots = rank[n + v - 1] = -1$. This array will be used to store the rank of the sample suffixes of x in the suffix array returned by the recursive call made later in Stage 1. While only $|C|$ elements of $rank$ will be used, and in fact a smaller array could be used to hold these values. However, we use a larger array to avoid complex indexing schemes relating elements in $rank$ to characters in x.

Stage 1 - Sort the sample suffixes

Let $\overline{\Sigma}$ be an alphabet of super-characters, which are defined to be in 1-1 correspondence with the distinct substrings of x of length v, i.e. super-character $\overline{x\,[i : i + v)}$ corresponds to the substring $x\,[i : i + v)$, for all $i \in C$. Therefore, $\overline{\Sigma} \subseteq (\Sigma \cup \{-1\})^v$. Recall from Section 1 that, due to the padding convention, any substring $x\,[i : j)$ is well-defined, for $i \in [0 : n)$ and $j > i$, and, therefore, any super-character $\overline{x\,[i : j)}$ is also well-defined.

For each $k \in D$, we now define a string of super-characters X_k over $\overline{\Sigma}$, where $X_k = \bigodot_{i \in B_k} \overline{x\,[i : i + v)}$ and $|X_k| = \frac{n}{v}$. Then, we construct the string of super-characters $X = \bigodot_{k \in D} X_k$, with $|X| = |D|\frac{n}{v}$. Note that for each k, the suffixes of X_k correspond to the set of suffixes S_k. The last super-character of X_k ends with one or more -1 sentinel elements, since 0 is not allowed to be in the difference cover. Therefore, each suffix of X corresponds to a different sample suffix of x, followed by one or more -1 sentinel characters followed by other characters that do not affect the lexicographic order of the suffixes of X. Note that, if 0 was allowed in the difference cover and n was a multiple of v, then the last super-character of X_k would not end with -1.

Recall from Section 1 that since the input to the algorithm is a string over integers, the string of super-characters X can be encoded as string X' over $\Sigma' = [0 : |X|)$ using radix sorting, in time $O(v|X|)$, where $|X'| = |X| = |D|\frac{n}{v}$. The order of the suffixes of X, i.e. the suffix array of X, can then be found by recursively calling the algorithm on the string X' over Σ', with parameters $|X'|$ and v', where v' can be chosen arbitrarily from the range $\left[3 : \min\left((1 - \epsilon)\,\frac{v^2}{|D|}, |X'|\right)\right]$, for some fixed $\epsilon > 0$. Thus, v' becomes the value of v in the subsequent recursion level. The bound $v' \leq (1 - \epsilon)\,\frac{v^2}{|D|}$ follows from the fact that we want the work done in the current level to be greater than the work done in the subsequent recursion level.

Recall from Section 2 that we require $|D| < v$. This ensures that $|X| < n$, so the algorithm is guaranteed to terminate, since each recursive call is always made on a shorter string.

When the recursive call returns with $SA_{X'}$, this holds the ordering of all the suffixes of X', i.e. the rank of the sample suffixes of x within the ordered set of sample suffixes. Then, for $i \in C$, the rank of s_i in $SA_{X'}$ is recorded in $rank[i]$. The order of the sample suffixes within each set S_k, $k \in D$, is also found from $SA_{X'}$.

The total cost of this stage is dominated by the radix sorting procedure required to encode string X into X' over $\Sigma' = [0 : |X|)$, which runs in time $O(|D|n)$.

Note that we can now compare any pair of suffixes by the result of Lemma 2. However, this is not sufficient to sort the suffixes of x in linear time, since each non-sample suffix of x would have to be compared to, possibly, all the other suffixes of x using different values of l. Instead, we perform the following.

Stage 2 - Find the order of the non-sample suffixes within each set S_k***,*** $k \in \mathbb{Z}_v \setminus D$

For each $k \in \mathbb{Z}_v \setminus D$, consider any $l_k \in [1 : v)$ such that $(k + l_k) \bmod v \in D$. For every character $x[i]$, $i \in [0 : n) \setminus C$, define the tuple $t_i = (x[i], x[i+1], \ldots, x[i+l_k-1], rank[i+l_k])$, where $k = i \mod v$. Note that $rank[i+l_k]$ is defined for each i, since $rank[a]$, for all $a \in C$, has been found in the previous stage and $rank[a] = -1$ for all $a \geq n$.

Then, for each set B_k, $k \in \mathbb{Z}_v \setminus D$, construct the sequence of tuples $(t_i)_{i \in B_k}$. Each of the $v - |D|$ constructed sequences has about $\frac{n}{v}$ tuples, with each tuple having less than v elements. The order of the suffixes within S_k is then obtained by independently sorting every sequence of tuples $(t_i)_{i \in B_k}$, using radix sorting.

The total computation cost of this stage is dominated by the cost of radix sorting all the sequences, i.e. $O((v - |D|)\, n) = O(vn)$.

Stage 3 - Sort all suffixes by first v ***characters***

Note that in the previous stages the order of every suffix within each set S_k, $k \in [0 : v)$, has been found. Now, let S^α be the set of suffixes starting with α, for $\alpha \in (\Sigma \cup \{-1\})^v$. Then, every set S^α is composed of ordered subsets S^α_k, where $S^\alpha_k = S^\alpha \bigcap S_k$.

All the suffixes s_i, $i \in [0 : n)$, are partitioned into the sets S^α by representing each suffix by the substring $x\,[i : i + v)$, and sorting these substrings using radix sort in time $O(vn)$.

Stage 4 - Merge and complete the suffix ordering

For all $\alpha \in \Sigma^v$, the total order within set S^α can be obtained by merging the subsets S^α_k, $k \in \mathbb{Z}_v$. This comparison-based v-way merging stage uses the fact that all the suffixes in x^α start with the same substring α, in conjunction with Lemma 1. Due to this lemma, a value $l \in [0 : v)$ exists such that for any i, j the comparison of suffixes s_i, s_j only requires the comparison of $rank[i + l]$ and $rank[j + l]$. Having already partitioned the suffixes into sets S^α and found the order of the suffixes within each set S_k, $k \in [0, v)$, the suffix array can be fully constructed through this merging process in time $O(n \log v) = O(vn)$. □

All the stages of the algorithm can be completed in time $O(vn)$, and the recursive call is made on a string of size at most $\frac{4}{5}n$, which corresponds to $|D| = 4$, $v = 5$. Note that a smaller difference cover of $\mathbb{Z}_5$ exists, but as discussed in section 2 we use the algorithm presented in [2] to construct the difference cover of $\mathbb{Z}_v$ for $v \geq 5$. This leads to an overall running time of $O(vn)$.

4 BSP model

The *bulk-synchronous parallel* (BSP) computation model [19,12] was introduced by Valiant in 1990, and has been widely studied ever since. The model was introduced with the aim of bridging the gap between the hardware development of parallel systems and the design of algorithms on such systems, by separating the system processors from the communication network. Crucially, it treats the underlying communication medium as a fully abstract communication network providing point-to-point communication in a strictly synchronous fashion. This allows the model to be architecture independent, promoting the design of scalable and portable parallel algorithms, while also allowing for simplified algorithm cost analysis based on a limited number of parameters.

A BSP machine consists of p processors, each with its local primary and secondary memory, connected together through a communication network that allows for point-to-point communication and is equipped with an efficient barrier synchronisation mechanism. It is assumed that the processors are homogeneous and can perform an elementary operation per unit time. The communication network is able to send and receive a word of data to and from every processor in g time units, i.e. g is the inverse bandwidth of the network. Finally, the machine allows the processors to be synchronised every l time units. The machine is fully specified using only parameters p, g, l, and is denoted by $BSP(p, g, l)$.

An algorithm in the BSP model consists of a series of *supersteps*, or *synchronisation steps*. In a single superstep, each processor performs a number of, possibly overlapping, computation and communication steps in an asynchronous fashion. However, a processor is only allowed to perform operations on data that was available to it at the start of the superstep. Therefore, in a single superstep, a processor can send and receive any amount of data, however, any received data can only be operated on in the following superstep. At the end of a superstep, barrier synchronisation is used to ensure that each processor is finished with all of its computation and data transfer.

The cost of a BSP superstep on a $BSP(p, g, l)$ machine can be computed as follows. Let $work_i$ be the number of elementary operations performed by processor $\pi \in [0 : p)$, in this superstep. Then, the *local computation cost* w of this superstep is given by $w = \max_{\pi \in [0:p)}(work_\pi)$. Let h_π^{out} and h_π^{in} be the maximum number of data units sent and received, respectively, by processor $\pi \in [0 : p)$, in this superstep. Then, the *communication cost* h of this superstep is defined as $h = \max_{\pi \in [0:p)}(h_\pi^{out}) + \max_{\pi \in [0:p)}(h_\pi^{in})$. Therefore, the total cost of the superstep is $w + h \cdot g + l$. The total cost of a BSP algorithm with S supersteps, with local computation costs w_s and communication costs h_s, $s \in [0 : S)$, is $W + H \cdot g + S \cdot l$, where $W = \sum_{s=0}^{S-1} w_s$ is the total local computation cost and $H = \sum_{s=0}^{S-1} h_s$ is the total communication cost.

The main principle of efficient BSP algorithm design is the minimisation of the algorithm's parameters W, H, and S. These values typically depend on the number of processors p and the problem size.

5 BSP Algorithm

Along with the sequential suffix array construction algorithm, described in Section 3, Kärkkäinen et al. [5] discuss the design of the algorithm on various computation models, including the BSP model. They give a brief overview of a parallel suffix array construction algorithm, running on a $BSP(p, g, l)$ machine, with optimal $O(\frac{n}{p})$

local computation and communication costs and requiring $O(\log^2 p)$ supersteps. The algorithm is based on the sequential algorithm described in Section 3 with parameter $v = 3$ used in every level of recursion. A number of existing parallel sorting and merging algorithms are used to achieve this result. Kulla and Sanders [9] show that this parallel algorithm actually requires $O(\log p)$ supersteps, implement it on a 64 dual-core processor machine and discuss the obtained results.

The algorithm described in Section 3 initially solves the suffix array problem on a sample of the suffixes of the input string, in order to gain information that is then used to efficiently sort all the suffixes. Sampling techniques are widely used in various fields ranging from statistics to engineering to computer science. A number of parallel algorithms exist that use sampling to efficiently solve problems, such as the sorting [15,1] and convex hull [17] algorithms. In [18], Tiskin presents a BSP algorithm for the selection problem, in which, not only is the data sampled, but, the sampling rate is increased at a carefully chosen rate in successive levels of recursion. This reduces the number of supersteps required by the parallel selection algorithm from the previous upper bound of $O(\log p)$ to a near optimal $O(\log \log p)$, while keeping the local computation and communication costs optimal.

In this section we make use of this technique, which we call *accelerated sampling*, to achieve the same synchronisation costs for our parallel suffix array construction algorithm, while, again, keeping the local computation and communication costs optimal. In contrast with [18], in our algorithm the sampling frequency has to be decreased, rather than increased, in successive levels of recursion. This is achieved by increasing the parameter v in successive levels of recursion. Since, as opposed to the previous work in [5,9], the presented algorithm does not assume a fixed parameter $v = 3$ the algorithm is described in great detail in order to cater for this generality, building on the description given in Section 3.

The algorithms to be presented in this section are designed to run on a $BSP(p, g, l)$ machine. We denote the sub-array of an array a assigned to processor $\pi \in [0 : p)$ by a_π and extend this notation to sets, i.e. we denote by A_π the subset of a set A assigned to processor π.

In the suffix array construction algorithm to be presented, we make extensive use of the parallel integer stable sorting algorithm introduced in [1]. This algorithm is based on the parallel sorting by regular sampling algorithm [15], but uses radix sorting to locally sort the input, removing the extra cost associated with comparison sorting. Given an array y having m distinct integers, such that each integer is represented by at most κ digits, the algorithm returns all the elements of y sorted in ascending order. Since the presented suffix array construction algorithm runs on strings over $\Sigma = \mathbb{N} \cup \{-1\}$, then we can use the same algorithm, which we refer to as the parallel string sorting algorithm, to sort an array of m strings or tuples, each of fixed length κ. In this case, the algorithm has $O(\kappa \frac{m}{p})$ local computation and communication costs and requires $O(1)$ supersteps.

Algorithm 2. *Parallel String Sorting*
Parameters: integer $m \geq p^3$; integer κ.
Input: array of strings $y = y[0] \cdots y[m-1]$, with each string of size κ over $\Sigma = \mathbb{N} \cup \{-1\}$.
Output: array y ordered in ascending lexicographical order.

Description:

The input array y is assumed to be equally distributed among the p processors, with every processor $\pi \in [0 : p-2]$, assigned the elements $y\left[\frac{m}{p}\pi : \frac{m}{p}(\pi+1)\right)$, and processor $p-1$ assigned elements $y\left[\frac{m}{p}(p-1) : m\right)$. Note that each processor holds $\frac{m}{p}$ elements, except the last processor $p-1$, which may hold fewer elements. We call this type of distribution of elements among the p processors a *block distribution*.

Each processor π first locally sorts sub-array y_π, using radix sorting, and then chooses $p+1$ equally spaced samples from the sorted sub-array, including the minimum and maximum values of y_π. These samples, which we call *primary samples*, are sent to processor 0. Having received $(p+1)p$ primary samples, each of which is a string of length κ, processor 0 locally sorts these samples, using radix sorting, and chooses $p+1$ sub-samples, including the minimum and maximum values of the primary samples. These chosen sub-samples, which we call *secondary samples*, partition the elements of y into p blocks $Y_0, \ldots, Y_{p-1}$. The secondary samples are broadcast to every processor, and each processor π then uses the secondary samples to partition its sub-array y_π into the p sub-blocks $Y_{0,\pi}, \ldots, Y_{p-1,\pi}$. Each processor π collects the sub-blocks $Y_{\pi,\chi}$ from processors $\chi \in [0 : p)$, i.e. all the elements of Y_π, and locally sorts these elements using radix sorting. The array y is now sorted in ascending lexicographic order, however, it might not be equally distributed among the processors, so an extra step is performed to ensure that each processor has $\frac{m}{p}$ elements of the sorted array. Note that each primary and secondary sample also has the index of the sample in y attached to it, so that any ties can be broken. Also note that the size of each block is bounded by $O(p)$ so the partitioning of the elements among the processors is balanced.□

The parallel suffix array construction algorithm presented below requires that the input string x of size n be equally distributed among the p processors, using a block distribution, prior to the algorithm being called. We denote by I_π the subset of the index set $[0 : n)$ that indexes x_π, $\pi \in [0 : p)$, i.e. $I_\pi = \left[\frac{n}{p}\pi : \frac{n}{p}(\pi+1)\right)$, for $\pi \in [0 : p-2]$, and $I_{p-1} = \left[\frac{n}{p}(p-1) : n\right)$. Finally, we use the same indexing for a and a_π, i.e. $a[i] = a_\pi[i]$. The algorithm is initially called on string x of length n, with parameters n and $v = 3$.

Algorithm 3. *Parallel Suffix Array Construction*
Parameters: integer $n \geq p^4$; integer $v \in [3 : n]$.
Input: string $x = x[0] \cdots x[n-1]$ over alphabet $\Sigma = [0 : n)$.
Output: suffix array $SA_x = SA_x[0] \cdots SA_x[n-1]$.
Description:

Recursion base

Recall that if all the characters of x are distinct, then SA_x can be obtained by sorting the characters of x in ascending order. Therefore, we call Algorithm 2 on string x with parameters $m = n$ and $\kappa = 1$. When the algorithm returns with the sorted array of characters, which we call x', each processor π holds the sub-array x'_π, of size $\frac{n}{p}$, and checks for character uniqueness in its sub-array. If all the characters in each sub-array are distinct, then, each processor $\pi \in [0 : p-2]$, checks with its neighbour $\pi+1$ to ensure that $x'[\frac{n}{p}(\pi+1)-1] \neq x'[\frac{n}{p}(\pi+1)]$. If every character is distinct then each character in the sorted array x' is replaced by its index in x and x' is returned. However, if at any point in this process identical characters are found, then the following stages are performed:

Stage 0 - Sample construction and initialisation

Every processor π, constructs the difference cover D of $\mathbb{Z}_v$ as discussed in Section 2. Then, each processor π, for each $k \in [0:v)$, defines the subset $B_{k\pi} = \{i \in I_\pi \mid i \bmod v = k\}$. This partitions each set of indices B_k into p subsets of size about $\frac{n}{pv}$. The subset C_π of the difference cover sample C is then constructed by every processor π, such that $C_\pi = \cup_{k \in D} B_{k\pi}$. We denote by $S_{k\pi}$, $k \in [0:v)$ and $\pi \in [0:p)$, the set of suffixes s_i, $i \in B_{k\pi}$.

To ensure that each processor is able to locally construct its subset of super-characters in the next stage, we require that every processor $\pi \in [1:p)$ sends the first $v-1$ characters of x_π to processor $\pi - 1$.

Finally, every processor π also declares the array $rank_\pi$, of size $\frac{n}{p} + v$ for $\pi \in [0:p-2]$, and size $n - \frac{n}{p}(p-1) + v$ for $\pi = p-1$. Each element of $rank_\pi$ is initialised by -1. Note that the size of each $rank_\pi$, $\pi \in [0:p-2]$, allows each processor to store a copy of the first v elements of $rank_{\pi+1}$ in order to be able to locally construct the tuples associated with all the non-sample characters in x_π.

Stage 1 - Sort the sample suffixes

For every processor π, we define, for each $k \in D$, the substring of super-characters $X_{k\pi} = \bigodot_{i \in B_{k\pi}} \overline{x\,[i:i+v)}$, such that the size of $X_{k\pi}$ is about $\frac{n}{pv}$. Note that every substring $x\,[i:i+v)$ is locally available for all $i \in C_\pi$, due to the padding convention and the distribution of x among the processors. Then, construct the string of super-characters X, as discussed in Section 3. This string is distributed among the p processors using a block distribution, with each processor having around $|D|\frac{n}{pv}$ super-characters. Note that it is not necessary to actually construct X, since the position of each $X_{k\pi}$, and, therefore, the index of each super-character $\overline{x\,[i:i+v)}$, $i \in C$, in X can be calculated by every processor π. However, this is done for simplicity. Algorithm 2 is then called on string X with parameters $m = |D|\frac{n}{v}$ and $\kappa = v$. After sorting, a rank is assigned to each super-character in its sorted order, with any identical super-characters given the same rank, and the string X' is constructed as discussed in Section 3. Note that X' is already equally distributed among the processors.

The algorithm is then called recursively on the string X' with parameters $n = |X'|$ and $v' = v^{5/4}$, where v' is the value of v in the subsequent recursion level. If $|X'| \leq \frac{n}{p}$, then X' is sent to processor 0, which calls the sequential suffix array algorithm on X' with parameters $n = |X'|$ and $v = 3$. A detailed discussion on the assignment $v' = v^{5/4}$ and its impact on the synchronisation costs of the algorithm is given later in this section.

When the recursive call returns with $SA_{X'}$, the rank of each s_i in $SA_{X'}$, $i \in C_{k\pi}$, $\pi \in [0:p)$, is recorded in $rank_\pi$. Also, a copy of the first v elements of $rank_\pi$, for $\pi \in [1:p)$, is kept in $rank_{\pi-1}$. The order of each suffix s_i within each set S_k, $k \in D$, is stored by each processor π, for $i \in I_\pi$.

Stage 2 - Find the order of the non-sample suffixes within each set S_k, $k \in \mathbb{Z}_v \setminus D$

For each $k \in \mathbb{Z}_v \setminus D$, consider any $l_k \in [1:v)$ such that $(k + l_k) \bmod v \in D$. We define the tuple $t_i = (x[i], x[i+1], \ldots, x[i+l_k-1], rank[i+l_k])$, for each character $x[i]$, $i \in I_\pi \setminus C_\pi$, $\pi \in [0:p)$ and $k = i \mod v$. Note that every character in the tuple t_i, $i \in I_\pi$, is locally available on processor π.

Then, every processor $\pi \in [0:p)$ constructs the subsequence of tuples $(t_i)_{i \in B_{k\pi}}$, for each subset $B_{k\pi}$, $k \in \mathbb{Z}_v \setminus D$. Therefore, each sequence $(t_i)_{i \in B_k}$ is the concatenation of the subsequences $(t_i)_{i \in B_{k\pi}}$ in ascending order of π. Recall from Section 3, that the

number of sequences $(t_i)_{i \in B_k}$ to be sorted is $v - |D|$, and that each sequence contains $\frac{n}{v}$ tuples, of length at most v; i.e. each processor holds about $\frac{n}{vp}$ tuples of each sequence.

Each sequence is then sorted independently using Algorithm 2 with parameters $m = \frac{n}{v}$ and κ being the length of the tuples in the sequence, which is at most v. After each sequence is sorted, the order of each non-sample suffix s_i, $i \in I_\pi$, within each set S_k, $k \in \mathbb{Z}_v \setminus D$, is stored by each processor π.

Stage 3 - Sort all suffixes by first v characters

Let each suffix s_i, $i \in [0:n)$, of x be represented by the substrings $x[i:i+v)$. These substrings are stably sorted using Algorithm 2 with parameters $m = n$ and $\kappa = v$. After sorting, the suffixes of x will have been partitioned into the sets S^α, $\alpha \in (\Sigma \cup \{-1\})^v$, as discussed in Section 3.

Stage 4 - Merge and complete the suffix ordering

Recall from Section 3 that each set S^α, $\alpha \in (\Sigma \cup \{-1\})^v$, is partitioned into at most v subsets S^α_k, $k \in [0:v)$, and that the order of the suffixes within each such subset has been found in the previous stages. Ordering a set S^α is achieved through a v-way merging procedure based on Lemma 2. For every two subsets $S^\alpha_{k'}$ and $S^\alpha_{k''}$, $k', k'' \in [0:v)$, we choose any $l \in [0:v)$ such that $(k'+l) \bmod v$ and $(k''+l) \bmod v$ are both in D. Then, comparing two suffixes $s_i \in S_{k'}$ and $s_j \in S_{k''}$ only requires the comparison of $rank[i+l]$ and $rank[j+l]$.

Therefore, in order to sort S^α we require, for each element of S^α, the rank of the element within the sorted subset S_k it belongs to and at most $|D|$ values from the array $rank$. Hence, at most $(|D|+1)\frac{n}{p}$ values need to be received by each processor. Note that the rank of each suffix s_i, $i \in [0:n)$, within the set S_k, $i \bmod v = k$, is stored on processor π, $i \in I_\pi$, as is $rank[i+l]$, for all $l \in [0:v)$.

After the sorting procedure in the previous stage, the suffixes of a set S^α, $\alpha \in \Sigma^v$, are contiguous and can be either contained within a single processor, or span two or more processors. If S^α is contained within one processor, then all the subsets of S^α are locally merged. If the set spans two processors $\pi', \pi'' \in [0:p)$, then, for each of the suffixes $s_i \in S^\alpha$, $i \in [0:n)$, on processor π'', the values required to merge the suffixes into the ordered set S^α are sent to processor π', which then constructs the order set. Otherwise, if S^α spans more than two processors, the following procedure, based on the parallel sorting by regular sampling technique, is performed.

The set S^α is block distributed among the p processors. Again, note that the actual suffixes $s_i \in S^\alpha$, $i \in [0:n)$, are not communicated, but only the values required by the merging process are, i.e. at most $|D|+1$ values for each suffix in S^α. Each processor locally sorts its assigned elements of S^α, using the v-way merging procedure, and chooses $p+1$ equally spaced primary samples from the sorted elements, including the minimum and maximum elements. Every primary sample is sent to one of the p processors that is chosen as the designated processor. Therefore, this designated processor receives $(p+1)p$ primary samples, which it sorts locally using the v-way merging procedure. It then chooses $p+1$ equally spaced secondary samples from the merged primary samples, including the minimum and maximum primary samples, that partition S^α into p blocks. These secondary samples are broadcast to the p processors such that each processor can partition its assigned elements into p sub-blocks. Every processor then collects all the sub-blocks that make up a unique block and locally merges the received elements. Finally, send the ordered set S^α back to the processors it originally spanned. Note that the size of each block is bounded by $O(p)$ so the partitioning of the elements among the processors is balanced.

Round i	v_i	$\lvert D_i\rvert$	n_i	Total Work = $O(v_i \cdot n_i)$
0	v	$O\left(v^{\frac{1}{2}}\right)$	n	$O\left(v \cdot n\right)$
1	$v^{\frac{5}{4}}$	$O\left(\left(v^{\frac{5}{4}}\right)^{\frac{1}{2}}\right)$	$O\left(v^{\frac{1}{2}} \cdot v^{-1} \cdot n\right) = O\left(v^{-\frac{1}{2}} \cdot n\right)$	$O\left(v^{\frac{3}{4}} \cdot n\right)$
2	$v^{\left(\frac{5}{4}\right)^2}$	$O\left((v^{\left(\frac{5}{4}\right)^2})^{\frac{1}{2}}\right)$	$O\left(\left(v^{\left(\frac{5}{4}\right)}\right)^{\frac{1}{2}} \cdot v^{-\frac{5}{4}} \cdot v^{-\frac{1}{2}} \cdot n\right) = O\left(v^{-\frac{9}{8}} \cdot n\right)$	$O\left(v^{\frac{7}{16}} \cdot n\right)$
i	$v^{\left(\frac{5}{4}\right)^i}$	$O\left(v^{\left(\frac{5}{4}\right)^i\left(\frac{1}{2}\right)}\right)$	$O\left(v^{-2\left(\frac{5}{4}\right)^i+2} \cdot n\right)$	$O\left(v^{-\left(\frac{5}{4}\right)^i+2} \cdot n\right)$
$\log_{\frac{5}{4}}\left(\log_v p^{\frac{1}{2}}+1\right)$	$v \cdot p^{\frac{1}{2}}$	$O\left(v^{\frac{1}{2}} \cdot p^{\frac{1}{4}}\right)$	$O\left(p^{-1} \cdot n\right)$	$O\left(v \cdot p^{-\frac{1}{2}} \cdot n\right)$

Table 3. Algorithm analysis

After all the sets S^α have been sorted, all the suffixes of x have been ordered and the suffix array is returned. □

5.1 Algorithmic Analysis

The presented suffix array construction algorithms are recursive, and the number of levels of recursion required for the algorithms to terminate depends on the factor by which the size of the input string is reduced in successive recursive calls. While the number of levels of recursion does not influence the running time of the sequential algorithm, in BSP this determines the synchronisation costs of the algorithm, and, therefore, we want to reduce it to a minimum. Before detailing the costs of each stage of the algorithm we explain how changing the sample size at each subsequent level of recursion results in $O(\log\log p)$ levels of recursion.

We refer to the i^{th} level of recursion of the algorithm as round i, $i \geq 0$. Then, we denote by n_i, v_i and D_i the size of the input string, the parameter v and the difference cover D of $\mathbb{Z}_{v_i}$, respectively, in round i.

Recall from Section 2 that the maximum size of a difference cover D of $\mathbb{Z}_v$, for any positive integer v, that can be found in time $O(\sqrt{v})$ is $\sqrt{1.5v} + 6$, i.e. $|D| = O(v^{1/2})$. Therefore, for the sake of simplicity, in our cost analysis we assume that $|D_i| = O\left({v_i}^{1/2}\right)$.

Changing the parameter v in successive recursive calls affects the sampling rate and the size of the input string. Let v_0 and n_0 be the parameters v and n given in the initial call to the algorithm, while $|D_0| = O\left({v_0}^{1/2}\right)$. Then, in round $i \geq 1$, $v_i = {v_{i-1}}^{5/4}$, $|D_i| = O\left(|D_i|^{5/4}\right)$ and $n_i = n_{i-1}v^{(-1/2)(5/4)^{i-1}}$. Note that $n_i = nv^{\sum_{k=1}^{i} -1/2(5/4)^{k-1}}$, i.e. the exponent of the term v is a geometric series with $a = -\frac{1}{2}$ and $r = \frac{5}{4}$. The analysis given in Table 3 illustrates how these values change in successive recursion levels. Recall from Section 3 that, the cost of each level of recursion in the sequential algorithm is $O(v_i n_i)$. Therefore, the table also shows that the order of work done decreases in subsequent recursive levels.

The results in Table 3 clearly show that if the algorithm is initially called on a string of size n, with parameter $v = 3$, on a $BSP(p, g, l)$ machine, then the size of the input converges towards $\frac{n}{p}$ super-exponentially. In fact, after $\log_{5/4}(\log_3 p^{1/2} + 1) = O(\log\log p)$ levels of recursion, the size of the input string is $O(\frac{n}{p})$, and in the subsequent level of recursion the suffix array is computed sequentially on processor 0. Note that the value $\frac{5}{4}$ as a power of v is not the only one possible. In fact, any value $1 < a < \frac{3}{2}$ can be used, but $a = \frac{5}{4}$ is used for simplicity.

Finally, recall that in Section 3 we require $v_i \leq n_i$ for the sequential algorithm. However, in our parallel algorithm, since we require that $n_0 > p^4$ and we set $v_0 = 3$, this will always be the case in the first $O(\log \log p)$ levels of recursion, at which point the algorithm is called sequentially on a single processor. Therefore, this bound is not required in our parallel algorithm.

Having determined the number of recursive calls required by the algorithm, the cost of each stage is now analysed. In the recursion base, the costs are dominated by those of Algorithm 2, i.e. $O(\frac{n_i}{p})$ local computation and communication cost.

In stage 0, constructing the difference cover D_i has local computation costs $O(\sqrt{v_i})$, while constructing the subsets C_π, independently for each processor $\pi \in [0 : p)$, has $O(|D_i|\frac{n_i}{pv_i})$ local computation cost. Passing the first $v-1$ characters of x_π, $\pi \in [1 : p)$, from processor π to $\pi - 1$ has $O(v)$ local computation and communication costs. Finally, initialising $rank_\pi$ requires $O(\frac{n_i}{p} + v_i)$ work. Only $O(1)$ supersteps are required.

In stage 1, the costs are dominated by the construction of the string of supercharacters X and the call to Algorithm 2, leading to $O(|D_i|\frac{n_i}{p})$ local computation and communication costs and requires $O(1)$ supersteps.

In stage 2, the costs are again dominated by the call to Algorithm 2 for each sequence of tuples. The number of sequences to be sorted is $v_i - |D_i|$, which is always less than p. Therefore, we can use a different designated processor for each call to Algorithm 2. The size of each sequence is $\frac{n_i}{v_i}$, and the size of each tuple is at most v_i. Therefore, each processor has $O(\frac{n_i}{v_i p})$ tuples from each sequence, i.e. $O(\frac{n_i}{p})$ tuples in total. Therefore, sorting all these tuples independently for each sequence has $O(v_i\frac{n_i}{p})$ local computation and communication costs and requires $O(1)$ supersteps.

The cost of stage 3 is simply the cost of Algorithm 2 on a string of size n_i with $\kappa = v_i$, i.e. $O(v_i\frac{n_i}{p})$ local computation and communication costs and $O(1)$ supersteps.

In stage 4, obtaining, for each suffix of x, the information required to sort each set S^α using a v-way merging procedure has $O(|D_i|\frac{n_i}{p})$ local computation and communication costs. Then, sorting a set S^α that is contained on a single processor has $O(|S^\alpha|v_i)$ local computation costs, and no communication is required. Note that in this case $|S^\alpha| < \frac{n_i}{p}$. If S^α spans two processors, then we send all the elements of the set to one of the two processors. Therefore, since each processor has $\frac{n_i}{p}$ suffixes, then, $2 \leq |S^\alpha| \leq 2\frac{n_i}{p}$, and the costs of sorting this set are $O(v_i\frac{n_i}{p})$ local computation, $O(|D_i|\frac{n_i}{p})$ communication and $O(1)$ supersteps.

Finally, if a set S^α spans more than 2 processors, then $|S^\alpha| > \frac{n_i}{p}$. Therefore, the number of such sets is less than p. In this case a technique based on parallel sorting by regular sampling on p processors is performed, choosing a different designated processor for each such set S^α. In fact, the only difference between the two techniques is that v-way merging is used, instead of radix sorting, to locally sort the suffixes. Since the v-way merging procedure on n elements has the same asymptotic costs as the radix sorting procedure on an array of n strings each of size v, over an alphabet $\Sigma = \mathbb{N} \cup \{-1\}$, then the local computation cost for this procedure is $O(v_i\frac{n_i}{p})$ and the communication cost is $O(|D_i|\frac{n_i}{p})$. Since each such set can be merged independently in parallel, then a constant number supersteps is required. Note that we could merge any set spanning $p' > 2$ processors on the p' processors instead of distributing the set across all the p processors, however we choose not to do this for the sake of simplicity.

In the i^{th} level of recursion, each stage has $O(v_i\frac{n_i}{p})$ local computation and communication costs and requires $O(1)$ supersteps. The presented parallel suffix array

construction algorithm is initially called on a string of size n with parameter $v = 3$, so the local computation and communication costs are $O(\frac{n}{p})$ in round 0. In round $\log_{5/4}(\log_3 p^{1/2} + 1)$ these costs are $O(\frac{n}{p^{3/2}})$. Note that after this round the algorithm is called sequentially on a string of length less than $\frac{n}{p}$. Also note that, as shown in table 3, $O(v_i n_i)$ decreases super-exponentially in each successive level of recursion, and, therefore, the order of work done in each such level also decreases super-exponentially. Therefore, the algorithm has $O(\frac{n}{p})$ local computation and communication costs and requires $O(\log \log p)$ supersteps.

Finally, recall that Algorithm 2 requires slackness, $m \geq p^3$. Since in the critical round Algorithm 2 is called in stage 2 on $O(p^{\frac{1}{2}})$ sequences of size $\frac{n}{p^{3/2}}$, we require that $n \geq p^4$. Note that this slackness can be reduced by sorting the sequences locally if each sequence fits on a separate processor, however, such detail is beyond the scope of this paper and will be given in a full version of this paper.

6 Conclusion

In this paper we have presented a deterministic BSP algorithm for the construction of the suffix array of a given string. The algorithm runs in optimal $O(\frac{n}{p})$ local computation and communication, and requires a near optimal $O(\log \log p)$ supersteps.

The method of regular sampling has been used to solve the sorting [15,1], and 2D and 3D convex hulls [17] problems. Random sampling has been used to solve the maximal matching problem and provide an approximation to the minimum cut problem [10] in a parallel context. An extension of the regular sampling technique, which we call *accelerated sampling*, was introduced by Tiskin [18] to improve the synchronisation upper bound of the BSP algorithm for the selection problem. The same technique was used here to improve the synchronisation upper bounds of the suffix array problem. Accelerated sampling is a theoretically interesting technique, allowing, in specific cases, for an exponential factor improvement in the number of supersteps required over existing algorithms.

It is still an open question whether the synchronisation cost of the suffix array problem and the selection problem can be reduced to the optimal $O(1)$ while still having optimal local computation and communication costs. Another open question is whether further applications of the sampling technique, whether regular, random or accelerated, are possible.

References

1. A. Chan and F. Dehne: *A note on coarse grained parallel integer sorting.* Parallel Processing Letters, 9(4) 1999, pp. 533–538.
2. C. J. Colbourn and A. C. H. Ling: *Quorums from difference covers.* Information Processing Letters, 75(1-2) July 2000, pp. 9–12.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms*, MIT Press, 3rd ed., 2009.
4. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proceedings of the 30th International Conference on Automata, Languages and Programming, 2003, pp. 943–955.
5. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction.* Journal of the ACM, 53(6) Nov. 2006, pp. 918–936.

6. J. Kilian, S. Kipnis, and C. E. Leiserson: *The organization of permutation architectures with bused interconnections*. IEEE Transactions on Computers, 39(11) November 1990, pp. 1346–1358.
7. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Linear-time construction of suffix arrays*, in Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching, 2003, pp. 186–199.
8. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching, 2003, pp. 200–210.
9. F. Kulla and P. Sanders: *Scalable parallel suffix array construction*. Parallel Computing, 33(9) 2007, pp. 605–612.
10. S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii: *Filtering: A method for solving graph problems in MapReduce*, in Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, 2011, pp. 85–94.
11. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches*, in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 319–327.
12. W. McColl: *Scalable computing*, in Computer Science Today, J. van Leeuwen, ed., vol. 1000 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1995, pp. 46–61.
13. C. Mereghetti and B. Palano: *The complexity of minimum difference cover*. Journal of Discrete Algorithms, 4(2) June 2006, pp. 239–254.
14. S. J. Puglisi, W. F. Smyth, and A. H. Turpin: *A taxonomy of suffix array construction algorithms*. ACM Computing Surveys, 39(2) July 2007.
15. H. Shi and J. Schaeffer: *Parallel sorting by regular sampling*. Journal of Parallel Distributed Computing, 14(4) Apr. 1992, pp. 361–372.
16. W. F. Smyth: *Computing Patterns in Strings*, Addison-Wesley, April 2003.
17. A. Tiskin: *Parallel convex hull computation by generalised regular sampling*, in Proceedings of the 8th International Euro-Par Conference on Parallel Processing, 2002, pp. 392–399.
18. A. Tiskin: *Parallel selection by regular sampling*, in Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II, 2010, pp. 393–399.
19. L. G. Valiant: *A bridging model for parallel computation*. Communications of the ACM, 33 August 1990, pp. 103–111.