Parallel and Distributed Compressed Indexes ⋆

Luís M. S. Russo[1], Gonzalo Navarro[2], and Arlindo L. Oliveira[3]

[1] CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia, FCT, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal. lsr@di.fct.unl.pt
[2] Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl
[3] INESC-ID, R. Alves Redol 9, 1000 Lisboa, Portugal. aml@algos.inesc-id.pt

Abstract. We study parallel and distributed compressed indexes. Compressed indexes are a new and functional way to index text strings. They exploit the compressibility of the text, so that their size is a function of the compressed text size. Moreover, they support a considerable amount of functions, more than many classical indexes. We make use of this extended functionality to obtain, in a shared-memory parallel machine, near-optimal speedups for solving several stringology problems. We also show how to distribute compressed indexes across several machines.

1 Introduction and Related Work

Suffix trees are extremely important for a large number of string processing problems, in particular in bioinformatics, where large DNA and protein sequences are analyzed. This partnership has produced several important results, but it has also exposed the main shortcoming of suffix trees. Their large space requirements, plus their need to operate in main memory to be useful in practice, renders them inapplicable in the cases where they would be most useful, that is, on large texts.

The space problem is so important that it has originated a plethora of research, ranging from space-engineered suffix tree implementations [1] to novel data structures to simulate them, most notably suffix arrays [2]. Some of those space-reduced variants give away some functionality. For example suffix arrays miss the important suffix link navigational operation. Yet, all these classical approaches require $O(n \log n)$ bits, while the indexed string requires only $n \log \sigma$ bits[4], being n the size of the string and σ the size of the alphabet. For example the human genome can be represented in 700 Megabytes, while even a space-efficient suffix tree on it requires at least 40 Gigabytes [3], and the reduced-functionality suffix array requires more than 10 Gigabytes. This problem is particularly evident in DNA because $\log \sigma = 2$ is much smaller than $\log n$.

These representations are also much larger than the size of the *compressed* string. Recent approaches [4] combining data compression and succinct data

⋆ Funded in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, and Fondecyt grant 1-080019, Chile (second author).

[4] In this paper log stands for $\log_2$.

structures have achieved spectacular results on what we will call generically *compressed suffix arrays (CSAs)*. These require space close to that of the compressed string and support efficient indexed searches. For example the most compact member of the so-called *FM-Index family* [5], which we will simply call *FMI*, is a CSA that requires $nH_k + o(n \log \sigma)$ bits of space and counts the number of occurrences of a pattern of length m in time $O(m(1+\frac{\log \sigma}{\log \log n}))$. Here nH_k denotes the k-th order empirical entropy of the string [6], a lower bound on the space achieved by any compressor using k-th order modeling. Within that space the FMI represents the text as well, which can thus be dropped.

It turns out that it is possible to add a few extra structures to CSAs and support all the operations provided by suffix trees. Sadakane was the first to present such a *compressed suffix tree (CST)* [3], adding $6n$ bits to the size of the CSA. This $\Theta(n)$ extra-bits space barrier was recently broken by the so-called *fully-compressed suffix tree (FCST)* [7] and by another entropy-bounded CST [8]. The former is particularly interesting as it achieves $nH_k + o(n \log \sigma)$ bits of space, asymptotically the same as the FMI, its underlying CSA.

Distributing CSAs have been studied, yet focusing only on pattern matching. For example, Mäkinen et al. [9] achieved optimal speedup in the amortized sense, that is, when many queries arrive in batch.

In this paper we study parallel and distributed algorithms for several stringology problems (with well-known applications to bioinformatics) based on compressed suffix arrays and trees. This is not just applying known parallel algorithms to compressed representations, as the latter have usually richer functionality than classical ones, and thus offer unique opportunities for parallelization and distributed representations. In Section 4 we present parallel shared-memory algorithms to solve problems like pattern matching, computing matching statistics, longest common substrings, and maximal repeats. We obtain near-optimal speedups, by using sophisticated operations supported by these compressed indexes, such as generalized branching. Optimal speedups for some of those problems (and for others, like all-pairs suffix-prefix matching) have been obtained on classical suffix trees as well [10]. Here we show that one can obtain similar results on those problems and others, over a compressed representation that handles much larger texts in main memory. In Section 5 we further mitigate the space problem by introducing distributed compressed indexes. We show how CSAs and CSTs can be split across q machines, at some price in extra space and reasonable slowdown. The practical effect is that a much larger main memory is available, and compression helps reducing the number of machines across which the index needs to be distributed.

2 Basic Concepts

Fig. 1 illustrates the concepts in this section. We denote by T a **string**; by Σ the **alphabet** of size σ; by $T[i]$ the symbol at position (i mod n) (so the first symbol of T is $T[0]$); by $T.T'$ the **concatenation**; by $T = T[..i-1].T[i..j].T[j+1..]$ respectively a **prefix**, a **subtring**, and a **suffix** of T.

The **path-label** of a node v, in a tree with edges labeled with strings over Σ, is the concatenation of the edge-labels from the root down to v. We refer indifferently to nodes and to their path-labels, also denoted by v. A **point** in $\mathcal{T}$ corresponds to any substring of T; this can be a node in $\mathcal{T}$ or a position within an edge label. The i-th letter of the path-label is denoted as $\textsc{Letter}(v, i) = v[i]$. The **string-depth** of a node v, denoted $\textsc{SDep}(v)$, is the length of its path-label, whereas the tree depth in number of edges is denoted $\textsc{TDep}(v)$. $\textsc{SLAQ}(v, d)$ is the highest ancestor of node v with $\textsc{SDep} \geq d$, and $\textsc{TLAQ}(v, d)$ is its ancestor of tree depth d. $\textsc{Parent}(v)$ is the parent node of v, whereas $\textsc{Child}(v, X)$ is the node that results of descending from v by the edge whose label starts with symbol X, if it exists. $\textsc{FChild}(v)$ is the first child of v, and $\textsc{NSib}(v)$ the next child of the same parent. $\textsc{Ancestor}(v, v')$ tells whether v is an ancestor of v', and $\textsc{LCA}(v, v')$ is the **lowest common ancestor** of v and v'.

The **suffix tree** of T is the deterministic compact labeled tree for which the path-labels of the leaves are the suffixes of $T\$$, where \$ is a terminator symbol not belonging to Σ. We will assume n is the length of $T\$$. The **generalized suffix tree** of T and T' is the suffix tree of $T\$T'\#$ where # is a new terminator symbol. For a detailed explanation see Gusfield's book [11]. The **suffix-link** of a node $v \neq \textsc{Root}$ of a suffix tree, denoted $\textsc{SLink}(v)$, is a pointer to node $v[1..]$. Note that $\textsc{SDep}(v)$ of a leaf v identifies the suffix of $T\$$ starting at position $n - \textsc{SDep}(v) = \textsc{Locate}(v)$. For example $T[\textsc{Locate}(ab\$)..] = T[7-3..] = T[4..] = ab\$$. The **suffix array** $A[0, n-1]$ stores the $\textsc{Locate}$ values of the leaves in lexicographical order. The *suffix tree nodes can be identified with suffix array intervals*: each node corresponds to the *range* of leaves that descend from v. The node b corresponds to the interval $[3, 6]$. Hence the node v will be represented by the interval $[v_l, v_r]$. Leaves are also represented by their left-to-right index (starting at 0). For example by $v_l - 1$ we refer to the leaf immediately before v_l, *i.e.* $[v_l - 1, v_l - 1]$. With this representation we can $\textsc{Count}$ in constant time the number of leaves that descend from v. The number of leaves below b is $4 = 6 - 3 + 1$. This is precisely the number of times that the string b occurs in the indexed string T. We can also compute $\textsc{Ancestor}$ in $O(1)$ time: $\textsc{Ancestor}(v, v') \ \Leftrightarrow \ v_l \leq v'_l \leq v'_r \leq v_r$. Operation $\textsc{Rank}_a(T, i)$ over a string T counts the number of times that the letter a occurs in T up to position i. Likewise, $\textsc{Select}_a(T, i)$ gives the position of the i-th occurence of a in T.

2.1 Parallel Computation Models

The parallel algorithms in this paper are studied under the Parallel Random Access Model (PRAM), which considers a set of independent sequential RAM processors, which have a private (or local) memory and a global shared memory. We assume that the RAMs can execute arithmetic and bitwise operations in constant time. We use the CREW model, where several processors can read the same shared memory cell simultaneously, but not write it. We call p the number of processors, and study the time used by the slowest processor.

For uniformity, distributed processing is studied by dividing the processors into q sites, which have a (fast) local memory and communicate using a (slow)

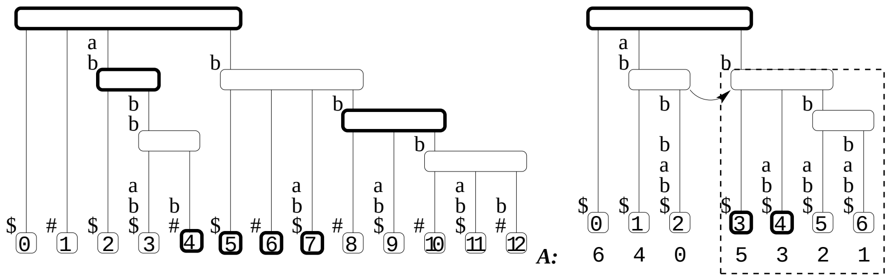

Fig. 1. Suffix tree $\mathcal{T}$ of string *abbbab* (right), with the leaves numbered. The arrow shows the SLINK between node *ab* and *b*. Below it we show the suffix array. The portion of the tree corresponding to node *b* and respective leaves interval is highlighted with a dashed rectangle. The sampled nodes have bold outlines. We also show the generalized suffix tree for *abbbab* and *abbbb* (left), using the \$ and # terminators respectively.

```
        ((0)(1)((2)((3)(4)))((5)(6)(7)((8)(9)(10)(11)(12))))

        ( 0  1 ( 2 ( 3 (4))) (5)(6)(7)( 8  9  10  11  12 ) )
    B : 1 0  0 1 0 1 0 10111 1011011011 0  0   0   0   0 1 1
    B0: 1 0    1 0 1 0 1 111 1011 11011    0       0     1 1
    B1: 1    0 1   1   10111 1 11011 11 0      0       0 1 1
```

Fig. 2. A parentheses representation of the generalized suffix tree in Fig. 1 (left), followed by the parentheses representation of the respective sampled tree $S_{\mathcal{C}}$. We also show B bitmap for the LSA operation of the sequential FCST and the B_i bitmaps for the LSA$_i$ operations of the distributed FCST.

shared memory. The number of accesses to this slower memory are accounted for separately, and they can be identified with the amount of communication carried out on shared-nothing distributed models. We measure the local and total memory space required by the algorithms.

3 Overview of Sequential Fully-Compressed Suffix Trees

In this section we briefly explain the local FCST [7]. It consists of a compressed suffix array, a sampled tree S, and mappings between these structures.

Compressed suffix arrays (CSAs) are compact and functional representations of suffix arrays [4]. Apart from the basic functionality of retrieving $A[i] = \text{LOCATE}(i)$ (within a time complexity that we will call Φ), state-of-the-art CSAs support operation SLINK(v) *for leaves v*. This is called $\psi(v)$ in the literature: $A[\psi(v)] = A[v]+1$, and thus SLINK$(v) = \psi(v)$, let its time complexity be Ψ. The iterated version of ψ, denoted ψ^i, can usually be computed faster than $O(i\Psi)$ with CSAs, since $\psi^i(v) = A^{-1}[A[v]+i]$. We assume the CSA also computes A^{-1} within $O(\Phi)$ time. CSAs might also support operation WEINERLINK(v, X) [12], which for a node v gives the suffix tree node with path-label $X.v[0..]$. This is called the LF mapping in CSAs, and is a kind of inverse of ψ. Let its time complexity be $O(\tau)$. We extend LF to strings, $\mathrm{LF}(X.Y, v) = \mathrm{LF}(X, \mathrm{LF}(Y, v))$. For example, consider the interval $[3, 6]$ that represents the leaves whose path-labels start by b. In this case we have that $\mathrm{LF}(a, [3, 6]) = [1, 2]$, *i.e.* by using the LF mapping with a we obtain the interval of leaves whose path-labels start by ab.

CSAs also implement $\textsc{Letter}(v, i)$ for leaves v. $\textsc{Letter}(v, 0) = T[A[v]]$ is $v[0]$, the first letter of the path-label of leaf v. CSAs implement $v[0]$ in $O(1)$ time, and $\textsc{Letter}(v, i) = \textsc{Letter}(\textsc{SLink}^i(v), 0)$ in $O(\Phi)$ time. CSAs are usually self-indexes, meaning that they replace the text: they can extract any substring $T[i..i+\ell-1]$ in time $O(\Phi+\ell\Psi)$ time, since $T[i..i+\ell-1] = \textsc{Letter}(A^{-1}[i], 0..\ell-1)$.

We will use a CSA called the FMI [5], which requires $nH_k + o(n \log \sigma)$ bits, for any $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$. It achieves $\Psi = \tau = O(1 + \frac{\log \sigma}{\log \log n})$ and $\Phi = O(\log n \log \log n)$.[5] The instantiation in Table 1 refers to the FMI.

The δ-sampled tree exploits the property that suffix trees are self-similar, $\textsc{SLink}(\mathrm{LCA}(v, v')) = \mathrm{LCA}(\textsc{SLink}(v), \textsc{SLink}(v'))$. A *$\delta$-sampled tree S*, from a suffix tree $\mathcal{T}$ of $\Theta(n)$ nodes, chooses $O(n/\delta)$ nodes such that, for each node v, node $\textsc{SLink}^i(v)$ is sampled for some $i < \delta$. Such a sampling can be obtained by choosing nodes with $\textsc{SDep}(v) = 0 \pmod{\delta/2}$ such that there is another node v' for which $v = \textsc{SLink}^{\delta/2}(v')$. Then the following equation holds, where $\mathrm{LCSA}(v, v')$ is the *lowest common sampled ancestor* of v and v':

$$\textsc{SDep}(\mathrm{LCA}(v, v')) = \max_{0 \leq i < \delta} \{i + \textsc{SDep}(\mathrm{LCSA}(\textsc{SLink}^i(v), \textsc{SLink}^i(v')))\} \qquad (1)$$

From this relation the kernel operations are computed as follows. The i in LCA is the one that maximizes the computation in Eq. (1).
•$\textsc{SDep}(v) = \textsc{SDep}(\mathrm{LCA}(v, v)) = \max_{0 \leq i < d}\{i + \textsc{SDep}(\mathrm{LCSA}(\psi^i(v_l), \psi^i(v_r)))\}$,
•$\mathrm{LCA}(v, v') = \mathrm{LF}(v[0..i-1], \mathrm{LCSA}(\psi^i(\min\{v_l, v'_l\}), \psi^i(\max\{v_r, v'_r\})))$,
•$\textsc{SLink}(v) = \mathrm{LCA}(\psi(v_l), \psi(v_r))$.

These operations plus $\textsc{Parent}(v)$, which is easily computed on top of LCA, take time $O((\Psi + t)\delta)$. The exception is $\textsc{SDep}$, which takes $O(\Psi\delta)$.

Note that we have to solve LCSA. This requires mapping nodes to their lowest sampled ancestors in S, an operation called LSA we explain next. In addition, each sampled node v must store its $[v_l, v_r]$ interval, its $\textsc{Parent}_S$ and $\textsc{TDep}_{\mathcal{T}}$, and also compute LCA_S queries in constant time. All this takes $O((n/\delta) \log n)$ bits. The rest is handled by the CSA.

Computing lowest sampled ancestors. Given a CSA interval $[v_l, v_r]$ representing node v of $\mathcal{T}$, the *lowest sampled ancestor* $\mathrm{LSA}(v)$ gives the lowest sampled tree node containing v. With LSA we can compute $\mathrm{LCSA}(v, v') = \mathrm{LCA}_S(\mathrm{LSA}(v), \mathrm{LSA}(v'))$.

[5] $\psi(i)$ can be computed as $select_{T[A[i]]}(T^{bwt}, T[A[i]])$ using the wavelet tree [13]. The cost for Φ assumes a sampling step of $\log n \log \log n$, which adds $o(n)$ extra bits.

Table 1. Comparing local and distributed FCST representations. The operations are defined in Section 2. Time complexities, but not space, are big-O expressions. The dominant terms in the distributed times count slow-memory accesses. We give the generalized performance and an instantiation using $\delta = \log n \log\log n$, assuming $\sigma = O(\text{polylog}(n))$, and using the FMI [5] as the CSA.

	Local	Distributed
Space in bits	$\lvert CSA\rvert + O((n/\delta)\log n)$ $= nH_k + o(n\log\sigma)$	$\lvert CSA\rvert + O((n/\delta)\log n + n\log(1+q/\delta))$ $= nH_k + o(n\log\sigma) + O(n\log q)$
SDep	$\Psi\delta$ $= \log n\log\log n$	$(\log q + \Psi + \tau)\delta$ $= \log q \log n \log\log n$
Count	1 $= 1$	$\log q$ $= \log q$
Ancestor	1 $= 1$	1 $= 1$
Parent/ FChild/ NSib/ SLink/ LCA	$(\Psi + \tau)\delta$ $= \log n\log\log n$	$(\log q + \Psi + \tau)\delta$ $= \log q \log n \log\log n$
SLink^i	$\Phi + (\Psi + \tau)\delta$ $= \log n\log\log n$	$\Phi + (\log q + \Psi + \tau)\delta$ $= \log q \log n \log\log n$
Letter(v, i)	Φ $= \log n\log\log n$	Φ $= \log n\log\log n$
Child	$\Phi\log\delta + (\Psi+\tau)\delta + \log(n/\delta)$ $= \log n(\log\log n)^2$	$\Phi\log\delta + (\Psi + \tau)\delta + \log(n/\delta)$ $= \log n(\log\log n)^2$
TDep	$(\Psi + \tau)\delta^2$ $= (\log n\log\log n)^2$	$(\log q + \Psi + \tau)\delta^2$ $= \log q\,(\log n\log\log n)^2$
TLAQ	$\log n + (\Psi + \tau)\delta^2$ $= (\log n\log\log n)^2$	$\log n + (\log q + \Psi + \tau)\delta^2$ $= \log q\,(\log n\log\log n)^2$
SLAQ	$\log n + (\Psi + \tau)\delta$ $= \log n\log\log n$	$\log n + (\log q + \Psi + \tau)\delta$ $= \log q \log n \log\log n$
WeinerLink	τ $= 1$	τ $= 1$

The key component for these operations is a bitmap B that is obtained by writing a 1 whenever we find a '(' or a ')' in the parentheses representation of the sampled tree S and 0 whenever we find a leaf of $\mathcal{T}$, see Fig. 2. Then the LSA is computed via Rank/Select on B. As B contains $m = O(n/\delta)$ ones and n zeros, it can be stored in $m\log(n/m) + O(m + n\log\log n/\log n) = O((n/\delta)\log\delta) + o(n)$ bits [14]. We now present a summary of the FCST representation.

Theorem 1. *Using a compressed suffix array (CSA) that supports ψ, ψ^i, $T[A[v]]$ and* LF *in times $O(\Psi)$, $O(\Phi)$, $O(1)$, and $O(\tau)$, respectively, it is possible to represent a suffix tree with the properties given in Table 1 (column "local").*

4 Parallel Compressed Indexes

In this section we study the situation where the index resides in main memory and we want to speed up its main search operations. We start by studying exact matching, matching statistics and longest common substrings over CSAs and FCSTs, and finish with maximal repeats (only over FCSTs).

The CSA's basic operations, such as ψ, LF, $A[i]$ and $A^{-1}[i]$, seem to be intrinsically sequential. However using *generalized branching* we can speed up several algorithms. The generalized branching $\text{Child}(v_1, v_2)$, for suffix tree points v_1 and v_2, is the point with path label $v_1.v_2$ if it exists. This operation was first considered by Huynh et al. [15], who achieved $O(\Phi \log n)$ time over a CSA. Russo et al. [16] achieved $O((\Psi + \tau)\delta + \Phi \log \delta + \log(n/\delta))$ time on the FCST. The procedure binary searches the interval of v_1 for the subinterval where $\psi^{\text{SDep}(v_1)} \in v_2$. With the information stored at sampled nodes, only an interval of size δ is binary searched this way; the other $O(\log(n/\delta))$ steps require constant-time accesses.

These binary searches can be accelerated using p processors. Instead of dividing the interval into two pieces we divide it into p and assign each comparison to a different processor. The time complexity becomes $\Pi(p) = O((\Psi+\tau)\delta+\Phi \log_p \delta + \log_p(n/\delta))$. The CSA-based algorithm also improves to $\Pi(p) = O(\Phi \log_p n)$.

Pattern matching. Assume we want to search for a pattern P of size m. We divide P into p parts and assign one to each processor. Each piece is searched for, like in the FMI, with the LF operation. This requires $O(m\tau/p)$ time. We then join the respective intervals with a binary tree of generalized Child operations. Assume for simplicity that m/p is a power of 2. We show a flow-graph of this procedure in Fig. 3. We first concatenate the $p/2$ pairs of leaves, using 2 processors for the generalized branching operation, using time $\Pi(2)$. We then concatenate the $p/4$ pairs of nodes of height 1, using 4 processors for the generalized branching, using time $\Pi(4)$. We continue until merging the two halves of P using p processors. The overall time of the hierarchical concatenation process is $O((\Psi + \tau)\delta \log p + (\Phi \log \delta + \log(n/\delta)) \log \log p)$ on the FCST and $O(\Phi \log n \log \log p)$ on the bare FMI. Using the same instantiation as in Table 1 this result becomes $O(m/p + \log n \log \log n(\log p + \log \log n \log \log p))$ time on the FCST and $O(m/p + \log^2 n \log \log n \log \log p)$ on the FMI, both in $nH_k + o(n \log \sigma)$ bits of index space. The speedup is linear in p, except for the polylogarithmic additive term. On the other hand, there is no point in using more than m processors; the optimum is achieved using less than m.

Matching statistics. The matching statistics $m(i)$ indicate the size of the longest prefix of $P[i..]$ that is a substring of T. Consider for example the string $P = abbbbabb$, using the running example suffix tree $\mathcal{T}$, right of Fig. 1. The corresponding matching statistics are $4, 3, 5, 4, 3, 3, 2, 1$. To compute these values we will again resort to the generalized Child operation. As before the idea is to first compute a generalized branch tree, which is the tree in the flow-graph of Fig. 3. This tree contains the intervals over CSA that correspond to strings $P[2^j k..2^j(k+1)-1]$, where j will indicate the level in the tree and k the position in the level. The levels and the positions start at 0. Notice that constructing this tree can be done exactly as for pattern matching, except that we do not stop the tree at pieces of length m/p but continue up to length 1. Since the subtrees for pieces of size m/p must be handled sequentially by one processor, they require additional time $O(m\tau/p + (m/p)\Pi(2))$. Note each node of this branching tree

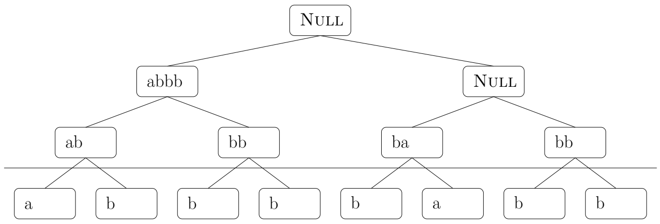

Fig. 3. The flow-graph for parallel exact matching and matching statistics is a tree of generalized CHILD operations for pattern $P = abbbbabb$ and $p = 4$ processors. Matching statistics use all the operations in the tree, whereas exact matching performs only the operations above the line and the search below the line is computed with LF operations.

stores the suffix tree point (or suffix array interval plus length) that corresponds to its substring, if it exists, and NULL otherwise.

After building the tree we traverse it m times, once for each $P[i..]$. For each such i we find $m(i)$ by traversing the tree path that covers $P[i..i + m(i) - 1]$ with the maximal nodes. This describes a path that ascends and then descends, touching $O(\log m)$ nodes of the branching tree. We start at the ith leaf x of the branching tree, which corresponds to the letter $P[i]$, with $v = P[i]$ the current point of $\mathcal{T}$, and move up to the parent z of x. If x is the right child of z, we do nothing more than $x \leftarrow z$. If x is the left child of z, we do a generalized CHILD(v, u) operation, where u is the point of $\mathcal{T}$ that is stored in y, the right child of z. If the resulting point is not NULL we set v to this new point, $v \leftarrow$ CHILD(v, u), and continue moving up, $x \leftarrow z$. Otherwise we start moving down on y, $x \leftarrow y$. While we are moving down we compute the generalized CHILD operation between v and the point u in the left child y of x. If the resulting point is still NULL we move to the left child of x, $x \leftarrow y$. Otherwise we set v to this new point, $v \leftarrow$ CHILD(v, u), and move to the right child z of x, $x \leftarrow z$. The value $m(i)$ is obtained by initializing it at zero and adding 2^j to it each time we update v at level j of the branching tree.

For example, assume we want to compute $m(2)$, *i.e.* we want to determine the longest prefix of *bbbabb* that is a substring of T. We start on the third leaf of the tree in Fig. 3 and set $v = b$. Then move up. Since we are moving from a left child we compute CHILD(b, b) and obtain $v = bb$. Again we move up but this time we are moving from a right child so we do nothing else. We move up again. Since the interval on the right sibling is NULL the CHILD operation also returns NULL. Therefore we start descending in the right sibling. We now consider the left child of that sibling, and compute CHILD(v, ba) = CHILD(bb, ba) = $bbba$. Since this node is not NULL we set v to it and move to the node labeled bb. Considering its left child we compute CHILD$(bbba, b) = bbbab$. Since it is not NULL we set v to it and move to rightmost leaf. Finally we check whether CHILD$(bbbab, b) \neq$ NULL since this is that case we know that we should consider the rightmost leaf as part of the common substring. This means that $m(2) = 5 = 2^0 + 2^0 + 2^1 + 2^0 = 7 - 2$.

Traversing the tree takes $O(\Pi(2)\log m)$ time per traversal, thus with p processors the time is $O((m/p)\Pi(2)\log m)$. By considering that only $p \leq m$ processors are useful, we have that the traversal time dominates the branching tree construction time. Using the instantiation in Table 1 this result becomes $O((m/p)\log m \log n(\log\log n)^2)$ on the FCST and $O((m/p)\log m \log^2 n \log\log n)$ on the FMI. The total space is $O(m\log m) + nH_k + o(n\log\sigma)$ bits. This time the linear speedup is multiplied by a polylogarithmic factor, since the sequential algorithm can be made $O(m)$ time.

Longest common substring. We can compute the longest common substring between P and T by taking the maximum matching statistic $m(i)$ in additional negligible $O(m/p + \log p)$ time.

Maximal repeats. For this problem we need a FCST, and cannot simulate it with a CSA as before. A maximal repeat in T is a substring S, of size ℓ, that occurs in at least two positions i and i', *i.e.* $S = T[i..i+\ell-1] = T[i'..i'+\ell-1]$, and cannot be extended either way, *i.e.* $T[i-1] \neq T[i'-1]$ and $T[i+\ell] \neq T[i'+\ell]$. The solution for this problem consists in identifying the deepest internal nodes v of $\mathcal{T}$ that are left-diverse, *i.e.* the nodes for which there exist letters $X \neq Y$ such that $X.v$ and $Y.v$ are substrings of T [11]. Assume $v = [v_l, v_r]$. Then FMIs allow one to access $\text{Letter}(v_l, -1) = T[v_l - 1]$ in $O(\tau)$ time[6]. Hence node v is left-diverse iff $\text{Count}(\text{LF}(\text{Letter}(v_l, -1), v)) \neq \text{Count}(v)$. This can be verified in $O(\tau)$ time, moreover this verification can be performed independently for every internal node of $\mathcal{T}$. At each step the algorithm chooses p nodes from $\mathcal{T}$ and performs this verification. A simple way to choose all internal nodes (albeit with repetitions, which does not affect the asymptotic time of this algorithm) is to compute $\text{LCA}([i,i],[i+1,i+1])$ for all $0 \leq i < n-1$. Hence this procedure requires $O((n/p)(\Psi + \tau)\delta)$ time, plus negligible $O(n/p + \log p)$ time to find the longest candidates. Using the same instantiation as in Table 1 the result becomes $O((n/p)\log n \log\log n)$ time within optimal $nH_k + o(n\log\sigma)$ overall bits. This is an optimal speedup, if we consider the polylogarithmic penalty of using a FCST.

The speedups in this section are similar to the results obtained for "classical" uncompressed suffix trees by Clifford [10], which do not speed up exact matching because they do not use a generalized Child operation. Clifford speeds up the longest common substring problem and the maximal repeats, among others.

5 Distributed Compressed Indexes

In this section we study distributed CSAs and FCSTs, mainly to obtain support for large string databases. In this case we assume we have a collection $\mathcal{C}$ of q texts of total length n, distributed across q machines. Hence distributed FCSTs are always generalized suffix trees, and likewise for CSAs. In fact, the local text of each machine could also be a collection of smaller texts, and the whole database

[6] This is an access to the Burrows-Wheeler transform.

could be a single string arbitrarily partitioned into q segments: CSAs and CSTs treat both cases similarly. The only difference is whether the SLink of the last symbol of a text sends one to the next text or it stays within that text, but either variant can be handled with minimal changes. We choose the latter option.

Various data layouts have been considered for distributing classical suffix trees and arrays [17, 9, 10]. One can distribute the texts and leave each machine index its own text, or distribute a single global index into lexicographical intervals, or opt for other combinations. In this paper we consider reducing the time of a single query, in contrast to previous work [17] where the focus is on speeding up batches of queries by distributing them across machines. Our approach is essentially that of indexing each text piece in its own machine, yet we end up distributing some global information across machines, similarly to the idea storing local indexes with global identifiers [17].

First we study the case where the local CSAs are used to simulate a global CSA, which can then be used directly in the FCST representation. This solution turns out to require extra space due to the need of storing some redundant information. Then we introduce a new technique to combine FCSTs that removes some of those redundant storage requirements.

Distributed Compressed Suffix Arrays. Assume we have a collection $\mathcal{C} = \{T_j\}_0^{q-1}$, and the respective local CSAs. We denote their operations with a subscript j, *i.e.* as A_j, A_j^{-1}, ψ_j and LF_j. The generalized CSA that results from this collection is denoted $A_{\mathcal{C}}$. Assume we store the accumulated text sizes, $AccT[i] = \sum_{j=0}^{i-1} |T_j|$, which need just $O(q \log n)$ bits.

We define the sequence $Id_{\mathcal{C}}$ of suffix indexes of $\mathcal{C}$, where $Id_{\mathcal{C}}[i] = j$ if the suffix in $A_{\mathcal{C}}[i]$ belongs to text T_j. Consider T as T_0 and T' as T_1 in our running example. The respective generalized suffix tree is shown in the left of Fig. 1. The Id sequence for this example is obtained by reading the leaves, and replacing \$ by 0 and # by 1. The resulting sequence is $Id = 0100101010101$.

If we process Id for Rank and Select queries we can obtain the operations of $A_{\mathcal{C}}$ from the operations of the A_j's. To compute Locate we use the equation $A_{\mathcal{C}}[v] = A_{Id[v]}[\textsc{Rank}_{Id[v]}(v-1)] + AccT[Id[v]]$. For example for $A_{\mathcal{C}}[4]$ we have that $A_1[\textsc{Rank}_1(4-1)] + AccT[1] = A_1[1] + 7 = 7$. To compute $A_{\mathcal{C}}^{-1}$ we use a similar relation, $A_{\mathcal{C}}^{-1}[i] = \textsc{Select}_j(A_j^{-1}[i - AccT[j]] + 1)$, where j is such that $AccT[j] \leq i < AccT[j+1]$. Likewise $\psi_{\mathcal{C}}$ is computed as $\psi_{\mathcal{C}}[v] = \textsc{Select}_{Id[v]}(\psi_{Id[v]}[\textsc{Rank}_{Id[v]}(v-1)] + 1)$. Computing $\mathrm{LF}_{\mathcal{C}}(X, [v_l, v_r])$ is more complicated: we compute LF in all the CSAs, *i.e.* $\mathrm{LF}_j(X, [\textsc{Rank}_j(v_l - 1), \textsc{Rank}_j(v_r) - 1])$ for every $0 \leq j < q$. If $[xv_{j,l}, xv_{j,r}]$ are the resulting intervals then $\mathrm{LF}_{\mathcal{C}}(X, [v_l, v_r]) = [\min_{j=0}^{q-1}\{\textsc{Select}_j(xv_{j,l} + 1)\}, \max_{j=0}^{q-1}\{\textsc{Select}_j(xv_{j,r} + 1)\}]$. Consider for example, how to compute $\mathrm{LF}_{\mathcal{C}}(a, [5, 12])$. We compute $\mathrm{LF}_0(a, [3, 6]) = [1, 2]$ and $\mathrm{LF}_1(a, [2, 5]) = [1, 1]$ and use the results to obtain that $\mathrm{LF}_{\mathcal{C}}(a, [5, 12]) = [\min\{\textsc{Select}_0(1+1), \textsc{Select}_1(1+1)\}, \max\{\textsc{Select}_0(2+1), \textsc{Select}_1(1+1)\}] = [\min\{2, 4\}, \max\{3, 4\}] = [2, 4]$. This requires $O(\log q)$ accesses to slow memory to compute minima and maxima in parallel.

A problem with this approach is the space necessary to store sequence Id and support Rank and Select. An efficient approach is to unfold Id into q

bitmaps, $BId_j[i] = 1$ iff $Id[i] = j$, and process each one for constant-time binary RANK and SELECT queries while storing them in compressed form [14]. Then since BId_j contains about n/q 1s, it requires $(n/q)\log q + O(n/q) + o(n)$ bits of space. We store each BId_j in the local memory of processor j, which requires space $|CSA_j| + (n/q)\log q + O(n/q) + o(n)$ local bits. The total space usage is $|CSA_{\mathcal{C}}| + n\log q + O(n) + o(qn)$ bits (if the partitions are not equal it is even less; $n\log q$ bits is the worst case). This essentially lets each machine map its own local CSA positions to the global suffix array, as done in previous work for classical suffix arrays (where the global identifiers can be directly stored) [17].

In this setup, most of the accesses are to local memory. One model is that queries are sent to all processors and the one able of handling it takes the lead. For $A_{\mathcal{C}}[v]$, each processor j looks if $BId_j[v] = 1$, in which case $j = Id[v]$ and this is the processor solving the query locally, in $O(\Phi)$ accesses to fast memory (processor j also stores values $AccT[j]$ and $AccT[j+1]$ locally). For $A_{\mathcal{C}}^{-1}[i]$, each processor j checks if $AccT[j] \leq i < AccT[j+1]$ and the one answering positively takes the lead, answering again in $O(\Phi)$ local accesses. $\psi_{\mathcal{C}}$ proceeds similarly to $A_{\mathcal{C}}$, in $O(\Psi)$ local accesses. $\mathrm{LF}_{\mathcal{C}}$ is more complex since all the processors must be involved, each spending $O(\tau)$ local accesses, and then computing global minima and maxima in $O(\log q)$ accesses to slow memory. Compare to the alternative of storing $CSA_{\mathcal{C}}$ explicitly and splitting it lexicographically: all the local accesses in the time complexities become global.

The $o(qn)$ extra memory scales badly with q (as more processors are available, each needs more local memory). A way to get rid of it is to use bitmap representations that require $n\log q + o(n\log q) + O(n\log\log q) = O(n\log q)$ bits and solve RANK and SELECT queries within $o((\log\log n)^2)$ time [18]. We will now present a new technique that directly represents global FCSTs using tuples of ranges instead of a single suffix array range.

Distributed Fully-Compressed Suffix Trees. Consider the generalized suffix tree $\mathcal{T}_{\mathcal{C}}$ of a collection of texts $\mathcal{C} = \{T_j\}_0^{q-1}$ and the respective individual suffix trees $\mathcal{T}_i$. Assume, also, that we are storing the $\mathcal{T}_i$ trees with the FCST representation and want to obtain a representation for $\mathcal{T}_{\mathcal{C}}$. A node of $\mathcal{T}_{\mathcal{C}}$ can be represented all the time as a q-tuple of intervals $\langle v_0, \ldots, v_{q-1}\rangle = \langle [v_{0,l}, v_{0,r}], \ldots, [v_{q-1,l}, v_{q-1,r}]\rangle$ over the corresponding CSAs. For example the node *abbb* can be represented as $\langle [2,2],[1,1]\rangle$. In fact we have just explained, in the distributed LF operation, how to obtain from these intervals the $[v_l, v_r]$ representation of node v of $\mathcal{T}_{\mathcal{C}}$ (via SELECT on $Id_{\mathcal{C}}$ and distributed minima and maxima). Thus these intervals are enough to represent v.

To avoid storing the Id sequence we map every interval $[v_{i,l}, v_{i,r}]$ directly to the sampled tree of $FCST_{\mathcal{C}}$, instead of mapping it to an interval v over $CSA_{\mathcal{C}}$ and then reducing it to the sampled tree of $FCST_{\mathcal{C}}$ with $\mathrm{LSA}_{\mathcal{C}}(v)$. We use the same bitmap-based technique for $\mathrm{LSA}_{\mathcal{C}}$, but store q local bitmaps instead of just a global one. The bitmaps B_j are obtained from the bitmap B of the $FCST_{\mathcal{C}}$ by removing the zeros that do not correspond to leaves of T_j, see Fig 2. This means that, in B_j, we are representing the $O(n/\delta)$ nodes of the global sampled tree and the n/q leaves of T_j. As each B_j has n/q 0s and $O(n/\delta)$ 1s, the compressed representation [14] supporting constant-time Rank and Select requires $(n/q)\log(1+q/\delta)+O(n/q)+o(n/\delta+n/q)$ bits. This is slightly better than the extra space of CSAs, totalling $O(n\log(1+q/\delta))+o(nq/\delta)$ bits. As before, the $o(\ldots)$ term can be removed by using the representation by Gupta et al. [18] at the price of $o((\log\log n)^2)$ accesses to fast local memory. Now the same computation for LSA carried out on B_j gives a global interval.

We then compute $\mathrm{LSA}_{\mathcal{C}}(v)=\mathrm{LCA}_{S_{\mathcal{C}}}(\mathrm{LSA}_0(v_0),\ldots,\mathrm{LSA}_{q-1}(v_{q-1}))$, where $\mathrm{LCA}_{S_{\mathcal{C}}}$ is the LCA operation over the sampled tree of $\mathcal{T}_{\mathcal{C}}$ and $\mathrm{LSA}_j(v_j)$ is the global LSA value obtained by processor j. This operation is computed in parallel in $O(\log q)$ accesses to slow memory (which replaces the global minima/maxima of the CSA). The sampled tree S and its extra data (e.g., to compute LCA in constant time) is stored in the shared memory. Hence accesses to S are always slow, which does not change the stated complexities. This mechanism supports the usual representation of the global FCST.

Consider, for example, that we want to compute the SDep of node *abbb*. Note that the SDep of $[2,2]$ in $\mathcal{T}_0$ is 7 and that the SDep of $[1,1]$ in $\mathcal{T}_1$ is 6. However the SDep of *abbb* in $\mathcal{T}_{\mathcal{C}}$ is 4. In this example we do not have to use ψ to obtain the result, altough in general it is necessary. By reducing the $[2,2]$ and $[1,1]$ intervals to the sampled tree of $FCST_{\mathcal{C}}$ we obtain the node *abbb* and the leaf *abbbb#*, see Fig. 1. The node we want is the LCA of these nodes, *i.e. abbb*.

Theorem 2. *Given a collection of q texts $\mathcal{C}=\{T_j\}_0^{q-1}$ represented by compressed suffix arrays (CSA_j) that support ψ, ψ^i, $T[A[v]]$ and* LF *in times $O(\Psi)$, $O(\Phi)$, $O(1)$, and $O(\tau)$, respectively, it is possible to represent a distributed suffix tree with the properties given in Table 1 (column "distributed").*

Moreover this technique has the added benefit that we can simulate the generalized suffix tree from any subcollection of the q texts, by using only the intervals of the texts T_j that we want to consider. However in this case we lose the TDep, TLAQ and SLAQ operations.

6 Conclusions and Future Work

Compressed indexes are a new and functional way to index text strings using little space, and their parallelization has not been studied yet. We have focused on parallel (shared RAM) and distributed suffix trees and arrays, which are the most pervasive compressed text indexes. We obtained almost linear speedups for the basic pattern search problem, and also for more complex ones such as computing matching statistics, longest common substrings, and maximal matches. The sequential algorithms for these problems are linear-time and easy to carry over compressed indexes, but hard to parallelize. Thanks to the stronger functionality of compressed indexes, namely the support of generalized branching, we achieve parallel versions for all of these. Some of our solutions can do with a compressed suffix array; others require a compressed suffix tree. We plan to apply this idea to other problems with applications in bioinformatics [11], such as all-pairs prefix-suffix queries.

Distributing the index across q machines further alleviates the space problem, allowing it to run on a larger virtual memory. Our distributed suffix arrays require $O(n \log q) + o(n)$ extra bits, whereas our suffix trees require $o(nq/\delta)$ extra bits. Both simulate a global index with $O(\log q)$ slowdown (measured in communication cost), so they achieve $O(q/\log q)$ speedup on each query.

A challenge for future work is to reduce this extra space, as $O(n \log q)$ can be larger than the compressed suffix array itself. We also plan to consider other models models such as BSP and batched queries [17]. An exciting direction is to convert the distributed index into an efficient external-memory representation for compressed text indexes, which suffer from poor locality of reference.

References

1. Giegerich, R., Kurtz, S., Stoye, J.: Efficient implementation of lazy suffix trees. Softw., Pract. Exper. **33**(11) (2003) 1035–1049
2. Manber, U., Myers, E.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. **22**(5) (1993) 935–948
3. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. **41**(4) (2007) 589–607
4. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. **39**(1) (2007) article 2
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algor. **3**(2) (2007) article 20
6. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48**(3) (2001) 407–430
7. Russo, L., Navarro, G., Oliveira, A.: Fully-Compressed Suffix Trees. In: Proc. 8th LATIN. LNCS 4957 (2008) 362–373
8. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comp. Sci. **410**(51) (2009) 5354–5364
9. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In: Proc. 15th ISAAC. LNCS 3341 (2004) 681–692
10. Clifford, R.: Distributed suffix trees. J. Discrete Algorithms **3**(2-4) (2005) 176–197
11. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press (1997)
12. Weiner, P.: Linear pattern matching algorithms. In: IEEE Symp. on Switching and Automata Theory. (1973) 1–11
13. Lee, S., Park, K.: Dynamic rank-select structures with applications to run-length encoded texts. In: Proc. 18th CPM. LNCS 4580 (2007) 95–106
14. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc 13th SODA. (2002) 233–242
15. Huynh, T.N.D., Hon, W.K., Lam, T.W., Sung, W.K.: Approximate string matching using compressed suffix arrays. Theor. Comput. Sci. **352**(1-3) (2006) 240–249
16. Russo, L., Navarro, G., Oliveira, A.: Dynamic Fully-Compressed Suffix Trees. In: Proc. 19th CPM. LNCS 5029 (2008) 191–203
17. Marín, M., Navarro, G.: Distributed query processing using suffix arrays. In: Proc. 10th SPIRE. LNCS 2857 (2003) 311–325
18. Gupta, A., Hon, W.K., Shah, R., Vitter, J.: Compressed data structures: dictionaries and data-aware measures. In: In Proc. 5th WEA. (2006) 158–169