Distributed Text Search using Suffix Arrays[☆]

Diego Arroyuelo[a,d,1,*], Carolina Bonacic[b,2], Veronica Gil-Costa[c,d,3], Mauricio Marin[b,d,f,4],
Gonzalo Navarro[f,e,5]

[a]*Dept. of Informatics, Univ. Técnica F. Santa María, Chile*
[b]*Dept. of Informatics, University of Santiago, Chile*
[c]*CONICET, University of San Luis, Argentina*
[d]*Yahoo! Labs Santiago, Chile*
[e]*Dept. of Computer Science, University of Chile*
[f]*Center for Biotechnology and Bioengineering*

Abstract

Text search is a classical problem in Computer Science, with many data-intensive applications. For this problem, *suffix arrays* are among the most widely known and used data structures, enabling fast searches for phrases, terms, substrings and regular expressions in large texts. Potential application domains for these operations include large-scale search services, such as Web search engines, where it is necessary to efficiently process intensive-traffic streams of on-line queries. This paper proposes strategies to enable such services by means of suffix arrays. We introduce techniques for deploying suffix arrays on clusters of distributed-memory processors and then study the processing of multiple queries on the distributed data structure. Even though the cost of individual search operations in sequential (non-distributed) suffix arrays is low in practice, the problem of processing multiple queries on distributed-memory systems, so that hardware resources are used efficiently, is relevant to services aimed at achieving high query throughput at low operational costs. Our theoretical and experimental performance studies show that our proposals are suitable solutions for building efficient and scalable on-line search services based on suffix arrays.

1. Introduction

In the last decade, the design of efficient data structures and algorithms for textual databases and related applications has received a great deal of attention, due to the rapid growth of the amount of text data available from different sources. Typical applications support text searches over big text collections in a client-server fashion, where the user queries are answered by a dedicated server [15]. The server efficiency—in terms of running time—is of paramount importance in cases where the services demanded by clients generate a heavy work load. A feasible way to overcome the limitations of sequential computers is to resort to the use of several computers, or processors, which work together to serve the ever increasing client demands [19].

One such approach to efficient parallelization is to distribute the data onto the processors, in such a way that it becomes feasible to exploit locality via parallel processing of user requests, each on a subset of the data. As opposed to *shared-memory* models, this *distributed-memory* model provides the benefit of better

scalability [44]. However, it introduces new problems related to the communication and synchronization of processors and their load balance.

This paper studies the parallelization of text indexes, in particular *suffix arrays* [39], in distributed memory systems, and describes strategies to reduce the inter-processor communication and to improve the load balance at search time.

1.1. Indexed Text Searching

The advent of powerful processors and cheap storage has enabled alternative models for information retrieval, other than the traditional one of a collection of documents indexed by a fixed set of keywords. One is the *full text* model, in which the user expresses its information need via words, phrases or patterns to be matched for, and the information system retrieves those documents containing the user-specified patterns. While the cost of full-text searching is usually high, the model is powerful, requires no structure in the text, and is conceptually simple [5].

To reduce the cost of searching a text, specialized indexing structures are adopted. The most popular are *inverted indexes* [5, 10, 68]. Inverted indexes are efficient because their search strategy is based on the vocabulary (the set of distinct words in the text), which is usually much smaller than the text, and thus fits in main memory. For each word, the list of all its occurrences (positions) in the text is stored. Those lists are large and may be stored on secondary storage, or in the main memory of the cluster nodes [10]. However, inverted indexes are suitable only for searching natural-language texts (which have clearly separated words that follow some convenient statistical rules [5]). *Suffix arrays* or PAT *arrays* [39, 28], on the other hand, are more sophisticated indexing structures, which are superior to inverted indexes when searching for phrases or complex queries such as regular expressions [5]. In addition, suffix arrays can be used to index texts other than natural language. Examples of these applications include computational biology (DNA or protein sequences), music retrieval (MIDI or audio files), East Asian languages (Chinese, Korean, and others), and other multimedia data files.

Pattern search on suffix arrays is based on binary search [39, 28]; see Section 2.1 for further details. Processing a single query X of length m in a text of length n takes $O(m \log n)$ time on the standard sequential suffix array. One can also achieve $O(m + \log n)$ time, yet by storing an extra array that doubles the space usage. Hence, trying to reduce such query time by using a distributed-memory parallel computer of P processors is not very attractive in practical terms. In real applications, however, many queries arrive at the server per unit of time. Such work load can be served by taking batches of Q queries. Processing batches in parallel is appealing in this context, as one is more interested in improving the throughput of the whole process rather than improving single operations.

To achieve this goal, a pragmatic (though naive) strategy would be to keep a copy of both the whole text database and the search index in each server machine and route the queries uniformly at random among the P machines. For very large databases, however, each machine is forced to keep a copy of a large suffix array, often in secondary memory, which can drastically degrade performance. A more sensible approach is to keep a single copy of the suffix array evenly distributed over the P main memories. Now the challenge is to achieve efficient performance on a cluster of P machines that must communicate and synchronize in order to serve every batch of queries. This is not trivial: on a naive partitioning of the suffix array, most array positions are expected to reference to text stored in a remote memory. We study these problems in this paper in order to achieve efficient text searching.

1.2. Problem Definition and Model of Computation

In its most basic form, the *full-text search* problem is defined as follows: Given a *text* $T[1..n]$, which is a sequence of n symbols from an ordered alphabet $\Sigma = \{1, \ldots, \sigma\}$, and given a *search pattern* $X[1..m]$ (also over Σ), we want to find all the occurrences of X in T. There are different kinds of queries for full-text search, depending on the application:

- locate queries, where one wants to report the starting positions in the text of the pattern occurrences, that is, the set $O = \{i \mid 1 \leq i \leq n - m + 1 \ \wedge \ T[i..i + m - 1] = X[1..m]\}$.

- count queries, where one wants to count the number of pattern occurrences, that is, compute $|O|$.

- exist queries, where one wants to check whether X occurs in T or not, that is, we want to determine whether $O = \emptyset$ or not.

In this paper we will study a variant of this problem suitable for parallel processing when throughput is emphasized: given a set of Q patterns $\{X_1[1..m], X_2[1..m], \ldots, X_Q[1..m]\}$, one wants to find the occurrences of each of these strings in T (we use equal length for all the patterns for simplicity, but our results do not depend on that). We will focus on count queries. In such a case, one wants to obtain the number of occurrences of each of the Q search patterns. The reason is that, on suffix arrays, the algorithmic complexity of the problem lies on the counting part, and when this is solved, locating reduces to merely listing the values in an array range. In addition, as we discuss in the Conclusions, it turns out that the strategies that are best for counting are also the most promising ones for locating. On the other hand, many relevant applications rely solely on count queries. Several examples, like computing matching statistics and longest common substrings, arise in computational biology scenarios [29, 58]. Another application is approximate searching, where k errors are allowed in the search pattern. A successful approach is to split P into $k + 1$ pieces that will be searched for exactly. An algorithm to find the best partition into $k + 1$ pieces carries out $O(m^2 k)$ count queries [50, 59].

We assume a model in which the text is known in advance to queries, and we want to carry out several queries on it. Therefore, as we said above, the text can be preprocessed in order to build a data structure (or *index*) on it, such that queries can be supported efficiently.

We shall study the problem on a distributed computing model. We assume a server operating on a set of P processors $P_0, P_1, \ldots, P_{P-1}$ (the *search nodes*), each containing its own local memory. The communication between processors is carried out through message passing. Users request service to one or more *broker* (or *master*) machines. The broker distributes the queries onto the P search nodes of the server, yet solving the queries may require access to the data stored on various machines. Finally, the broker must gather the results from the search nodes and produce the answer to each query.

We assume that, under a situation of heavy traffic, the server processes batches of $Q = qP$ queries, for some $q \geq 1$. Our main goal is to achieve a high query throughput to answer these batches of queries. We will model the distributed-memory computation using the *bulk-synchronous* model of parallel computing (BSP model) [61, 64]. See Section 2.2 for further details about this model.

1.3. Previous Related Work

1.3.1. Distributed Query Processing and Inverted Indexes

The distributed query processing on text databases has been extensively studied, being the inverted index one of the most used data structures [63, 57, 37, 4, 11, 32, 42, 47, 66]. Those works obtain highly efficient inverted indexes, with a high query throughput. There are two main ways to distribute an inverted index over several processors [63]:

Local index approach The document database is evenly distributed across processors, and then a local inverted index is built for every processor. At query time, the broker must send each query in the batch of Q queries to each of the P processors.

Global index approach A global inverted index is built for the entire collection. Then, the vocabulary (i.e., set of different terms or words in the collection) is evenly distributed across processors, such that each processor stores only the inverted lists for the terms in the local vocabulary. At query time, the broker sends each query in the batch to the processor whose local vocabulary contains the query terms.

A disadvantage of the local index approach is that all the processors have to handle all the queries, and then the results must be merged, whereas in the global index approach just one processor immediately gives the final result. On the other hand, the global index approach may suffer from load imbalance, since queries tend to be skewed towards certain terms (that vary over time) and thus some processors may have to handle many more queries than others. This load imbalance can be avoided by distributing the terms in a more clever way [46], but coping with skewed distributions that vary over time is more challenging.

3

1.3.2. The Importance of Suffix Arrays in Text Processing and Searching

There are many applications that need to search texts where the concept of word is not well defined, as in the case of East Asian languages, or it does not exist at all, such as, biological databases (like DNA and proteins) or music retrieval. We need in these cases more general search tools, able to search for any text substring, not only words or phrases. Suffix trees [65] and suffix arrays [39] are the most well-known and used indexes of this kind.

Suffix trees are probably the most important data structures for processing strings [2, 29]. They are extensively used not only for full-text searching, but also for more complex pattern matching and discovery problems. It has been shown that a suffix array can support the same functionality as a suffix tree, provided that extra data is stored [1], but it needs no extra data to support simple pattern searches. In practice, suffix arrays are in many cases preferred over suffix trees, because of their smaller space requirement and better locality of access. However, suffix arrays (and also suffix trees) were initially designed as sequential data structures, that is, they did not support parallel query processing.

1.3.3. Distributed Suffix Trees and Arrays

The efficient parallel construction of suffix trees [3, 21, 30, 20, 13, 16, 40, 67] and suffix arrays [52, 31, 33, 27, 35, 48, 54] has been thoroughly investigated. The aim was, however, the construction in parallel of a global index for the entire text collection, so that queries upon that index can be carried out using the standard sequential search algorithm. Clifford and Sergot [17] defined a version of suffix trees that supports the distributed query processing on genomic data. Ferragina and Luccio [22] proposed a related algorithm that works on a distributed Patricia tree that is constructed from the suffix array. This algorithm is optimal both in computation and communication time. We have only found one article addressing our specific problem of distributed text searching using suffix arrays: Cheng et al. [14] define two strategies, which are based on the local and global approaches for inverted indexes. These have several drawbacks from a distributed-memory point of view (e.g., low concurrency and load imbalance), as we will thoroughly demonstrate in this paper. Indeed, we take the local and global strategies as baselines and build improved solutions upon them.

1.4. Our Contribution

In this paper we propose several strategies to represent suffix arrays under the distributed memory model. We begin by proposing two basic distribution strategies, which are realizations of the local and global index approaches of inverted indexes already seen in Section 1.3.1. In the case of counting queries on suffix arrays, however, these simple approaches pose serious problems on top of those they present on inverted indexes. The local index approach leads to low concurrency because the search cost on a part of the collection, $O(m \log(n/P))$, is close to that on the whole collection, $O(m \log n)$, and thus low concurrency is achieved. On the other hand, the global index approach requires accessing text stored in remote memories at each step of the search, leading to very high communication complexity.

Henceforth, the main contribution of this paper are approaches that seek to alleviate these problems on suffix arrays. We propose several alternatives to balance the work load, reduce communication costs, and achieve high concurrency.

We focus on improving query throughput in scenarios where batches of queries must be served. Our algorithms are designed and analyzed on the *bulk-synchronous* model of parallel computing (BSP model) [64, 61]. This is a distributed memory model with a well-defined structure that enables predicting the running time of an algorithm. We use this feature to compare different alternatives for index partitioning, considering their effects in communication and synchronization of processors.

Our empirical results show that the proposed algorithms are efficient in practice, outperforming by far the classical local and global choices to distribute the text collection across processors.

The present paper is an extension of a conference version [43], with more in-depth explanations of the proposed methods, improvements to the original algorithms, a tighter algorithm analysis, and a more extensive experimental evaluation.

2. Preliminary Concepts

We study here the preliminary concepts needed to understand our work.

2.1. Suffix Arrays

Suffix arrays [39] (also known as PAT *arrays* [28]) are one of the most used data structures for full-text searching. Given a text $T[1..n]$ of length n over an alphabet Σ, the corresponding suffix array $SA[1..n]$ stores pointers to the initial positions of the text suffixes. The array is sorted in lexicographical order of the suffixes, as shown in Figure 1 for the example text "`this_is_a_sample_text$`". Here, we have replaced spaces by '`_`', which is assumed to be lexicographically smaller than the remaining symbols. Also, symbol $\$ \notin \Sigma$ is a special text terminator, which is assumed to be lexicographically smaller than any other alphabet symbol. Given an interval of the suffix array, notice that all the corresponding suffixes in that interval form a lexicographical subinterval of the text suffixes.

To search for a pattern $X[1..m]$ in the text T we must find all the text suffixes that have X as a prefix, that is, the suffixes starting with X. Since the array is lexicographically sorted, the search for a pattern proceeds by performing two binary searches over the suffix array: one for the immediate predecessor of X, and another for its immediate successor. We obtain in this way an interval in the suffix array that contains the starting positions of the pattern occurrences. Algorithm 1 illustrates how this search is carried out.

Algorithm 1: SASearch(Suffix array SA, search pattern X)

1: $l \leftarrow 1$
2: $r \leftarrow n$
3: **while** $l < r$ **do**
4:     $h \leftarrow \lfloor \frac{l+r}{2} \rfloor$
5:     **if** $X >_{\text{lex}} T[SA[h]...\min\{SA[h] + m - 1, n\}]$ **then**
6:         $l \leftarrow h + 1$
7:     **else**
8:         $r \leftarrow h$
9:     **end if**
10: **end while**

Let us call *interval query* to the operation of using SASearch with both the immediate predecessor and the immediate successor of X. This takes $O(m \log n)$ time.

Example 1. The search for pattern "`s_`" leads to binary searches obtaining the positions pointed to by suffix array members 16 and 17 of Figure 1. This indicates that there are two occurrences of "`s_`" in the text, and these occurrences start at positions $SA[16] = 7$ and $SA[17] = 4$.

As we can see from line 1 of Algorithm 1, the binary searches on the suffix array are conducted by direct comparison of the suffixes pointed to by the array elements. This means that we need to access the text to perform a search. This will be an important issue when designing our distributed realizations of suffix arrays.

2.2. BSP and the Cost Model

In the *bulk-synchronous parallel* (BSP) model of computing [64, 61], any parallel computer (e.g., PC cluster, shared- or distributed-memory multiprocessors, etc.) is seen as a set of P processors composed of a CPU and a local memory, which communicate with each other through messages. Let $\mathtt{P}_0, \dots, \mathtt{P}_{P-1}$ denote these processors. The computation is organized as a sequence of *supersteps*. During a superstep, the processors may perform sequential computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	t	h	i	s	_	i	s	_	a	_	s	a	m	p	l	e	_	t	e	x	t	$

(a) Running-example text $T =$ "this_is_a_sample_text$".

i	text suffix	$SA[i]$
1	$	22
2	_a_sample_text$	8
3	_is_a_sample_text$	5
4	_sample_text$	10
5	_text$	17
6	a_sample_text$	9
7	ample_text$	12
8	e_text$	16
9	ext$	19
10	his_is_a_sample_text$	2
11	is_a_sample_text$	6
12	is_is_a_sample_text$	3
13	le_text$	15
14	mple_text$	13
15	ple_text$	14
16	s_a_sample_text$	7
17	s_is_a_sample_text$	4
18	sample_text$	11
19	t$	21
20	text$	18
21	this_is_a_sample_text$	1
22	xt$	20

(b) Text suffixes (lexicographically sorted) and the corresponding suffix array.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
SA	22	8	5	10	17	9	12	16	19	2	6	3	15	13	14	7	4	11	21	18	1	20

(c) Running-example suffix array.

Figure 1: Our running example.

This model of parallel computation ensures portability at the very fundamental level by allowing algorithm design in a manner that is independent of the architecture of the parallel computer. Shared and distributed memory parallel computers can be programmed in the same way.

The practical model of programming is SPMD, which is realized as P program copies running on the P processors, wherein communication and synchronization among copies is performed via libraries such as BSPlib [8], BSPub [7] or BSPonMPI [9]. Note that BSP is actually a paradigm of parallel programming and not a particular communication library. In practice, it is certainly possible to implement BSP programs using the traditional PVM [55] and MPI [45] libraries. A number of studies have shown that bulk-synchronous parallel algorithms lead to more efficient performance than their message-passing or shared-memory counterparts in many applications [61, 62].

The total running time cost of a BSP program is the cumulative sum of the costs of its supersteps, and the cost of each superstep is the sum of three quantities: w, hG and L, where:

- w is the maximum of the computations performed by each processor, excluding sending/receiving data;

- h is the maximum of the messages sent/received by each processor, with each word costing G units of running time; and

- L is the cost of the barrier synchronization of the processors.

The effect of the computer architecture is included by the parameters G and L, which are increasing functions of P. These values, along with the computing cost of relevant operations for the application at hand can be empirically determined for each parallel computer by executing benchmark programs at installation [61].

Example 2. As an example of a basic BSP algorithm, let us consider a broadcast operation that will be used in this paper. Suppose that a processor needs to send a copy of P chapters of a book, each of size a, to all other P processors (including itself). A naive approach would be to send the P chapters to all the processors in one superstep. That is, in superstep 1, the sending processor sends P chapters to P processors at a cost of $P^2aG + L$ units. Thus, in superstep 2 all P processors have available into their respective incoming message buffers the P chapters of the book. An optimal algorithm for the same problem is as follows. In superstep 1, the sending processor sends just one *different* chapter to each processor at a cost of $PaG + L$ units. In superstep 2, each processor sends its arriving chapter to all others at a cost of $PaG + L$ units. Thus, at superstep 2, all the processors have a copy of the whole book. Hence the broadcast of a large P-piece a-sized message can be carried out at $PaG + L$ cost.

3. Distributed Suffix Arrays

Assume that we are given a text $T[1..n]$, and we want to search for a batch $\{X_1[1..m], X_2[1..m], \ldots, X_Q[1..m]\}$ of $Q = qP$ individual patterns (or queries) of length m each (the assumption of equal-length patterns is made only for simplicity; it is not a restriction for our algorithms). Assume also that we want to support count queries, which from a suffix array point of view means determining the suffix array intervals containing the answers to each query.

We introduce in this section our proposals for the distributed processing of pattern queries using suffix arrays. Subsection 3.1 presents two basic strategies that are analogous to the local and global distribution approaches of inverted indexes. After discussing and highlighting the drawbacks of these solutions, Subsection 3.2 introduces our new proposals, which will lead to more efficient realizations of distributed suffix arrays. As a baseline, recall that the cost of answering such queries using a standard sequential suffix array would be $O(Qm \log n)$.

Note that a simple solution in distributed environments is to replicate the text and its resulting suffix array among the cluster nodes. In such a scheme, the broker sends the query to any node in order to solve it. This results in a perfect load balance. However, the text and index replication leads to using P times the space of the original solution. This makes it suitable only for small texts. Since we focus on moderate-size and big texts (where indexed text search is the most efficient alternative), we will not consider this simple approach in this paper. However, we explore partial replication in a case study where this is feasible, in Section 4.7.

3.1. Global versus Local Suffix Arrays

We first present the basic techniques to distribute suffix arrays, in order to understand the basic ideas and problems we must face. As a result, we obtain the suffix array counterparts for the global and local distributed inverted indexes [63, 14] (already mentioned in Section 1.3.1).

3.1.1. Global Suffix Arrays

In the *global* index approach, a single global suffix array SA is built for the whole text [35] and then mapped evenly on the processors. That is, processor P_0 stores $SA[1..\frac{n}{P}]$, processor P_1 stores $SA[\frac{n}{P} + 1..2\frac{n}{P}]$, and so on. From now on, let SA_i denote the range of the global suffix array SA stored at processor P_i. We call GLOBAL this distribution strategy. The text is also stored in a similar fashion: processor P_1 stores $T[1..\frac{n}{P}]$, processor P_2 stores $T[\frac{n}{P} + 1..2\frac{n}{P}]$, and so on. Let T_i denote the portion of T stored at processor P_i.

Example 3. Figure 2(a) shows a realization of this idea for the example text of Figure 1, using $P = 4$ processors.

Notice that not necessarily a suffix $SA_i[k]$ corresponds to a text position in the local text T_i, but probably to a text position corresponding to T_j for some other processor P_j. This is the case, for instance, of suffix $SA_0[2]$ in Figure 2(a), which is not in T_0 but in T_1. This will have implications when the suffix array search algorithm needs to access the text suffixes.
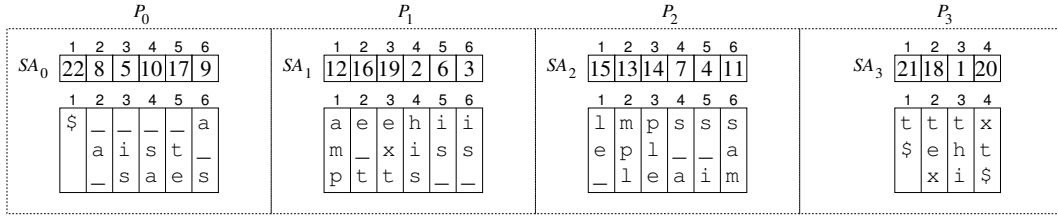
In the GLOBAL approach, each processor stands for a lexicographical interval or range of suffixes. For instance, in Figure 2(a) processor P_0 represents suffixes starting with '$', '_' and 'a_'. The broker machine maintains information on the P suffixes limiting the intervals in each machine[6]. These P suffixes are searched at the broker to determine which machine should handle each query (more precisely, we search for each of the two lexicographical extremes of the query). The CPU cost per pattern can be made $O(m)$ by using, for example, a trie data structure for the P strings [26]. After determining the correct processor for each query, all the queries are routed to their corresponding processors in a single superstep. Load imbalance may occur at this point, if queries tend to be dynamically biased to particular intervals, just as for global distributed inverted indexes [63]. Also, particularly on texts over small alphabets (e.g., DNA), the lexicographic range of a query can span more than one processor. In such a case, the query is routed to all involved processors, which poses some extra overhead.

GLOBAL *Search Algorithm and Cost Analysis.* Assume the ideal scenario, in which the queries are routed uniformly at random onto the processors. A search for $Q = qP$ queries is carried out as follows. The broker takes $Qm + QmG + L$ time to route the Q queries to their respective target processors (the first term is the cost of the trie search that determines which processor must answer each query).
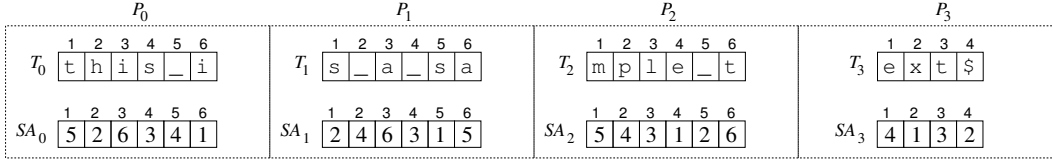
Once the processors get their (on average) q queries, each of them performs q binary searches concurrently, each for a different query. The cost for the q queries in a given processor is $O(qm \log \frac{n}{P})$, since these are performed on local arrays of size $\frac{n}{P}$. Recall, from Section 2.1, that these binary searches are conducted by direct comparison with the corresponding text suffix. As explained before, many of the references in a local suffix array could correspond to text suffixes stored in a remote processor. Indeed, for each query there is a probability of $1 - 1/P$ of needing a piece of text of length m from a remote processor. This takes an additional superstep, plus the cost of communicating m symbols per query. To avoid the case where the piece of remote text we are requesting is split among two consecutive processors, which would trigger extra communication, we store (but do not index) a few kilobytes of the text surrounding each text piece T_i. This gets rid of the problem for any practical query pattern and with a negligible space overhead.

Note that the q binary searches carried out at any processor P_i will start asking for the first m symbols of the suffix pointed from the middle of its suffix array range, SA_i. Thus we need only one remote request to carry out the first step of all the q binary searches. It is not hard to see that this idea can be extended to the first $\log_2 q$ steps, so that we request only q remote suffixes. We can start asking for those q suffixes (or
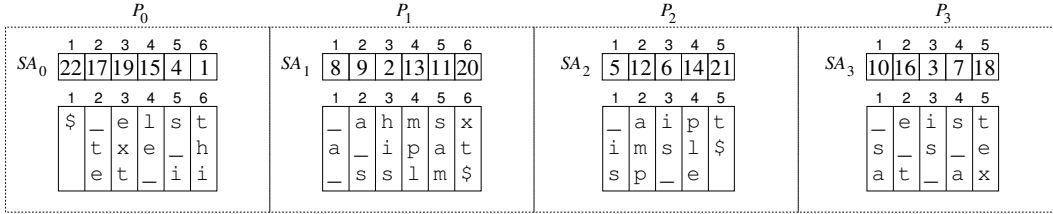
---

[6] Those can be arbitrarily long, but in practice we can limit them to the maximum pattern length m that can be sought.

P_0

SA_0

1	2	3	4	5	6
22	8	5	10	17	9

1	2	3	4	5	6
$	_	_	_	_	a
	a	i	s	t	_
	_	s	a	e	s

P_1

SA_1

1	2	3	4	5	6
12	16	19	2	6	3

1	2	3	4	5	6
a	e	e	h	i	i
m	_	x	i	s	s
p	t	t	s	_	_

P_2

SA_2

1	2	3	4	5	6
15	13	14	7	4	11

1	2	3	4	5	6
l	m	p	s	s	s
e	p	l	_	_	a
_	l	e	a	i	m

P_3

SA_3

1	2	3	4
21	18	1	20

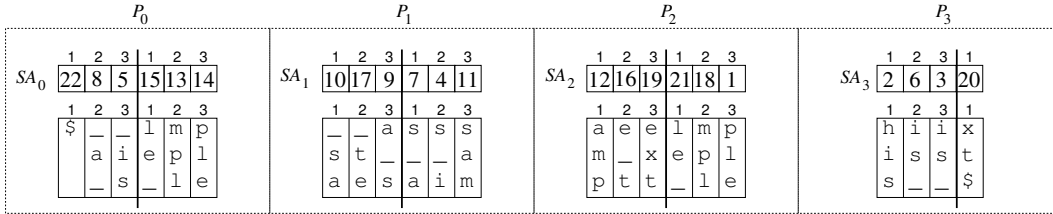| 1	2	3	4
t	t	t	x
$	e	h	t
	x	i	$

(a) The GLOBAL distribution strategy. The pruned suffixes at each processor are shown below each local suffix array, for $t = 3$.
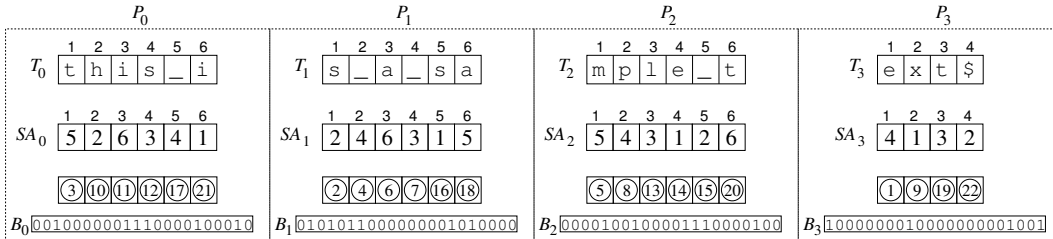
P_0

T_0

1	2	3	4	5	6
t	h	i	s	_	i

SA_0

1	2	3	4	5	6
5	2	6	3	4	1

P_1

T_1

1	2	3	4	5	6
s	_	a	_	s	a

SA_1

1	2	3	4	5	6
2	4	6	3	1	5

P_2

T_2

1	2	3	4	5	6
m	p	l	e	_	t

SA_2

1	2	3	4	5	6
5	4	3	1	2	6

P_3

T_3

1	2	3	4
e	x	t	$

SA_3

1	2	3	4
4	1	3	2

(b) The LOCAL distribution strategy.

P_0

SA_0

1	2	3	4	5	6
22	17	19	15	4	1

1	2	3	4	5	6
$	_	e	l	s	t
	t	x	e	_	h
	e	t	_	i	i

P_1

SA_1

1	2	3	4	5	6
8	9	2	13	11	20

1	2	3	4	5	6
_	a	h	m	s	x
a	_	i	p	a	t
_	s	s	l	m	$

P_2

SA_2

1	2	3	4	5
5	12	6	14	21

1	2	3	4	5
_	a	i	p	t
i	m	s	l	$
s	p	_	e	

P_3

SA_3

1	2	3	4	5
10	16	3	7	18

1	2	3	4	5
_	e	i	s	t
s	_	s	_	e
a	t	_	a	x

(c) The MULTIPLEXED distribution strategy. The pruned suffixes are shown below each local suffix array, for $t = 3$.

P_0

SA_0

1	2	3	1	2	3
22	8	5	15	13	14

1	2	3	1	2	3
$	_	_	l	m	p
	a	i	e	p	l
	_	s	_	l	e

P_1

SA_1

1	2	3	1	2	3
10	17	9	7	4	11

1	2	3	1	2	3
_	_	a	s	s	s
s	t	_	_	_	a
a	e	s	a	i	m

P_2

SA_2

1	2	3	1	2	3
12	16	19	21	18	1

1	2	3	1	2	3
a	e	e	l	m	p
m	_	x	e	p	l
p	t	t	_	l	e

P_3

SA_3

1	2	3	1
2	6	3	20

1	2	3	1
h	i	i	x
i	s	s	t
s	_	_	$

(d) The VMULTIPLEXED distribution strategy for $k = 1$. The pruned suffixes are shown below each local suffix array, for $t = 3$. Note that each processor actually contains $2^k = 2$ independent suffix arrays.

P_0

T_0

1	2	3	4	5	6
t	h	i	s	_	i

SA_0

1	2	3	4	5	6
5	2	6	3	4	1

③ ⑩ ⑪ ⑫ ⑰ ㉑

B_0 0010000001110000100010

P_1

T_1

1	2	3	4	5	6
s	_	a	_	s	a

SA_1

1	2	3	4	5	6
2	4	6	3	1	5

② ④ ⑥ ⑦ ⑯ ⑱

B_1 0101011000000001010000

P_2

T_2

1	2	3	4	5	6
m	p	l	e	_	t

SA_2

1	2	3	4	5	6
5	4	3	1	2	6

⑤ ⑧ ⑬ ⑭ ⑮ ⑳

B_2 0000100100001110000100

P_3

T_3

1	2	3	4
e	x	t	$

SA_3

1	2	3	4
4	1	3	2

① ⑨ ⑲ ㉒

B_3 1000000010000000001001

(e) The GLOCAL distribution strategy. Below each local suffix array, we show an array indicating the corresponding global suffix array position, and its representation as a bit vector.

Figure 2: The different distribution alternatives, for $P = 4$ processors, on our running example.

9

have them stored in the processor). From that point, each query might traverse a distinct path in its binary search. This amounts to a total of $q(\log \frac{n}{P} - \log q) = q \log \frac{n}{Q}$ remote requests for the q queries.

For a global suffix array of size n, the local binary searches and the exchange of the text substrings are performed at cost $qm \log \frac{n}{P} + qm \log \frac{n}{Q} G + L \log \frac{n}{Q}$, again assuming that requests for text pieces arrive uniformly at the processors. Finally, the q answers per processor are received by the broker at cost QG to continue with the following batch. Thus the total (asymptotic) cost per batch is given by

$$\underbrace{(Qm + QmG + L)}_{\text{broker cost}} + \underbrace{\left(qm \log \frac{n}{P} + qm \log \frac{n}{Q} G + \log \frac{n}{Q} L \right)}_{\text{server cost}}, \tag{1}$$

where the first term represents the cost of the operations carried out by the broker machine, and the second term is the cost of processing a Q-sized batch in the server of P processors.

Pruned Suffixes. The idea of storing the top-q suffixes of the binary search can be further exploited, so that we associate with *every* local suffix array entry the first t characters of the corresponding suffix. The value of t depends on the text and usual queries, and involves a space/time tradeoff. In Figure 2(a), the pruned suffixes for $t = 3$ are shown below each suffix array. It has been shown that this strategy is able to reduce the remote-memory references to 5%, even for relatively modest values of t [33]. However, this increases the space usage: the pruned suffixes require $nt \log \sigma$ extra bits overall.

3.1.2. Local Suffix Arrays

In the LOCAL strategy, on the other hand, we first divide the text into chunks of size $\frac{n}{P}$, and distribute these chunks onto the processors. Then, we construct a suffix array in each processor, considering only the portion of text stored in that processor. See Figure 2(b) for an illustration. Unlike the GLOBAL approach, no references to text positions stored in remote processors are made, thus avoiding the cost of sending text substrings of length m per binary-search step.

LOCAL *Search Algorithm and Cost Analysis.* Since the occurrences of a search pattern can be spread over the local texts of all processors, it is necessary to search for every query in all of the processors. Thus, we find the intervals in all the local suffix arrays that, together, form the solution for a given query. Finally, it is necessary to send to the broker QP values (Q per processor) corresponding to the results of the local count queries. The broker must sum up these values to obtain the answer to the original count query.

Special care must be taken with pattern occurrences that are split among two consecutive processors, that is, where for some processor P_i and search pattern $X_j[1..m]$, a suffix of T_i matches $X_j[1..k]$ and a prefix of T_{i+1} matches $X_j[k+1..m]$. The same technique described above, which stores a few kilobytes around each text T_i, avoids this problem. More precisely, P_i will only index the suffixes that start within T_i, and thus only report occurrences that start within T_i, but in order to determine that X_j matches near the end of T_i, it will use the prefix of T_{i+1} that is stored in P_i.

The processing of a batch of Q queries is as follows. The broker must send each query to every processor. This broadcast operation can be carried out as described in Section 2.2. That is, from the qP total queries, the broker sends q different queries to each of the P processors. Then, each individual processor broadcasts them to all other processors, at a total cost of $QmG + L$. In the next superstep, the search process is carried out locally and concurrently: each processor performs Q binary searches and sends the Q answers to the broker. In other words, at a given time all the processors will be dedicated to answer the same set of queries.

The final step, in which each processor sends the Q answers to the broker, makes the latter receive QP messages in one superstep. This cost can be reduced as follows: Each of the P processors will be responsible for summing up a slice of q queries. Therefore, in one superstep each processor will send the results it obtained for each of the Q queries to the processor responsible of handling it. Then each processor will sum up the contributions of each of the q queries it handles over the P processors that have sent it their results on that query. Finally, each processor will send to the broker the sums obtained for the q queries it handles.

10

Thus the cost is only $Q + QG + L$. The total cost of this strategy is given by

$$\underbrace{(Qm + QmG + L)}_{\text{broker cost}} + \underbrace{\left(Qm \log \frac{n}{P} + QG + L\right)}_{\text{server cost}}. \qquad (2)$$

3.1.3. Discussion

The cost analysis of the GLOBAL and LOCAL approaches (see Equations (1) and (2), respectively), lead to some clear conclusions. A first one is that GLOBAL achieves high parallelism, obtaining in its CPU cost an optimal speedup over the sequential search time, $Qm \log n$. It poses, however, a significant communication overhead, comparable to the CPU cost and clearly dominating it in practice (this is alleviated, however, by the use of pruned suffixes). To this we must add that our analysis optimistically assumes that queries are uniformly distributed, which is not the case in practice.

On the other hand, LOCAL has minimal communication costs, which are reduced to the cost of distributing the queries and collecting the answers. However, its CPU cost is very high, almost the same as that of a sequential execution.

In both cases, the results are clearly unsatisfactory and deserve further research. While we believe that the LOCAL strategy is fundamentally doomed by the closeness of the search costs in a global and a local suffix array (i.e., $m \log n$ versus $m \log \frac{n}{P}$), we will show that the challenges posed by the GLOBAL strategy can be addressed in order to achieve efficient solutions.

3.2. Alternative Ways to Distribute the Global Suffix Array

From the previous discussion, the GLOBAL strategy seems to be more promising. In this section we will show how its problems of a high amount of communication and load imbalance can be significantly reduced. Subsection 3.2.1 introduces the MULTIPLEXED strategy, which distributes the suffix array entries onto the processors using a Round-Robin scheme. This breaks load-imbalance but introduces further communication among processors to solve queries. Subsection 3.2.2 provides a smooth transition between GLOBAL and MULTIPLEXED, aiming at obtaining the best from both. Finally, Subsection 3.2.3 introduces the GLOCAL strategy, which can be regarded as an irregular MULTIPLEXED strategy that avoids accessing text from other processors, at the expense of some complications on top of MULTIPLEXED.

Similarly to the GLOBAL approach, in the following three techniques, every query will be initially assigned to just one processor. This processor will use the local suffix array to perform the initial binary search shown in Algorithm 1. Given the way in which the suffix array will be distributed, this initial search will give us just a partial answer. In further supersteps, this partial search will be eventually resumed by a different processor, which will tighten the answer. Notice that in this way the binary search for a particular query is carried out just once by a single processor, thus avoiding the effect of the LOCAL approach, where the binary search for the same pattern is repeated by all the processors at a given time.

3.2.1. Global Multiplexed Suffix Array

One drawback of GLOBAL is the possibility of load imbalance, coming from large and sustained sequences of queries being routed to the same processor. This is because in GLOBAL the suffix array is distributed by lexicographic ranges. The best way to avoid particular preferences for a given processor would be to send queries uniformly at random among the processors, thus achieving a fair distribution of queries onto the processors.

We propose to achieve this effect by *multiplexing* the entries of the original global array, as follows: if the global suffix array element $SA[i]$ is stored at processor P_j, then the global element $SA[i+1]$ is stored at P_{j+1}, $SA[i+2]$ is stored at P_{j+2}, and so on, in a circular fashion. The elements assigned to a processor P_k are stored in the same relative order to form the local suffix array SA_k. We call MULTIPLEXED this strategy.

Example 4. See Figure 2(c) for an illustration of the MULTIPLEXED strategy for our running example.

Notice that consecutive entries in a local suffix array correspond to entries that are separated by P positions in the global suffix array. From another point of view, the suffix array SA_i at processor P_i is formed by a regular sampling of the global suffix array SA, as shown in Figure 3(a). Just as for GLOBAL, we need the pruned suffixes to reduce remote references. The following property states that, under this way of distributing the global suffix array, it is straightforward to know which processor stores any desired suffix array entry.

Property 1. Global suffix array entry $SA[i]$ is stored at processor $P_{i \bmod P}$.

This means that the relative order of the global-array entries is maintained by this distribution strategy.

MULTIPLEXED *Search Algorithm.* Given a search pattern $X[1..m]$, let $[i_1..i_2]$ be the *target interval* in the global suffix array SA containing all the occurrences of X. We do not have the global suffix array, but the local ones. The idea is that the binary search of Algorithm 1 can be carried out at *any* of the local suffix arrays. In this way, we obtain a local interval, containing the local occurrences of X. This is a subset of the whole set of occurrences of X. Assume that the initial search with Algorithm 1 is carried out locally at processor P_j, obtaining the interval $[i'_1..i'_2]$ of SA_j. Since we only use the information of SA_j for this search, we have that entry $SA_j[i'_1]$ does not necessarily correspond to entry $SA[i_1]$ in the global suffix array. The same happens for $SA_j[i'_2]$ and $SA[i_2]$.

Example 5. Consider searching for "s_" in the MULTIPLEXED suffix array of Figure 2(c). Assume that the search starts at processor P_0. Hence, the resulting local interval is $[5..5]$. This corresponds to the global interval $[17..17]$. Notice that the global interval containing the occurrences of "s_" is $[16..17]$. Finally, the interval $[4..6]$ in SA_0 corresponds to the global interval $[13..21]$ in SA, which contains the target interval $[16..17]$ we are looking for.

The initial binary search gives us much information about the target interval for a query. We get a local interval that is a subinterval (in the global sense) of the target interval. More precisely, the target interval starts somewhere in between the global suffix array entries corresponding to $SA_j[i'_1 - 1]$ and $SA_j[i'_1]$, and it ends somewhere between the entries corresponding to $SA_j[i'_2]$ and $SA_j[i'_2 + 1]$.

Our next task is to find the processors that store the endpoints $SA[i_1]$ and $SA[i_2]$ of the target interval containing the answers to the query. Recall that the initial local search has been carried out at processor P_j. Given the way in which the global suffix array entries have been distributed across processors, we can start from processor $j' = P_{(j-1) \bmod P}$, and check whether the entry $SA_{j'}[i'_1 - 1]$ corresponds to $SA[i_1]$ (i.e., we check whether the suffix pointed by $SA_{j'}[i'_1]$ starts with X or not). This process is repeated with the next processor, $(j - 2) \bmod P$, and so on. This stops as soon as we find the first text suffix starting with X or, eventually, find out that X does not occur in T (in the special case where $i'_2 = i'_1 - 1$).

This process corresponds to a sequential search starting from position $i'_1 - 1$, which takes $O(P)$ extra supersteps in the worst case. Instead, the inter-processor search can be carried out with a binary search across the P processors, spending at most $\log P$ additional supersteps (recall from Property 1 how global positions are related to local ones). A similar process is carried out when looking for the processor storing $SA[i_2]$ among processors $(j + 1) \bmod P, (j + 2) \bmod P, \dots, (j + P - 1) \bmod P$.

Given a query that is originally assigned to processor P_i, the corresponding inter-processor search may proceed in two ways. In the first, P_i keeps control of the search and requests texts to other processors, which send back to P_i the requested suffix array values and pruned suffixes. If the pruned suffix is not sufficient then P_i requests more text characters to the processor owning the full suffix. In the second way, P_i *delegates* the problem of tightening this query to the processor that follows in the binary search. This has the advantage that pruned suffixes do not need to be transmitted, but instead the pattern must be sent. We can avoid sending the patterns by making the broker distribute all the patterns to all the machines at the beginning of the batch, at a cost of $QmG + L$, as described in Section 3.1.2.

This idea, in which we start solving a query in a given processor to later delegate the problem to another processor, is similar to the pipelining strategy defined for global distributed inverted indexes [47]. The difference is that, in our case, we need to send a smaller amount of data among processors, versus the partial

(a) The regular sampling pattern of the Multi-plexed distribution strategy.

(b) The non-regular sampling pattern of the GLocal distribution strategy.
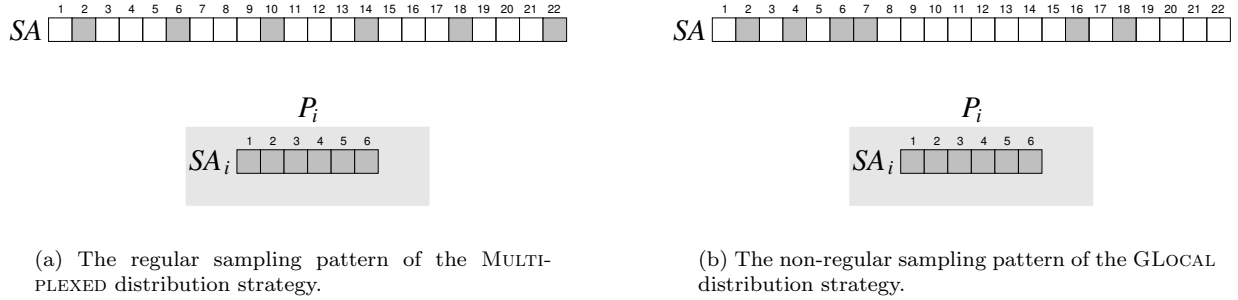
Figure 3: Different ways in which the global suffix array is sampled by the different techniques we propose.

results (in some cases, whole inverted lists) that must be sent in inverted indexes. Also, while load imbalance is a problem in the pipelined strategy [12, 60], Multiplexed should be less sensitive to the query bias. The first stage of Multiplexed query processing is fully balanced because *any* processor can carry out the binary search (indeed, the same query can be started by a different processor each time it is raised). In the second stage, where the target interval is refined, every processor is involved, each performing a small amount of work, thus the load is balanced as well. A small source of imbalance can be the particular number of character comparisons carried out by each processor, however, this should be very low (as we will see in our experiments).

Cost Analysis. The asymptotic cost of Multiplexed is

$$\underbrace{(Qm + QmG + L)}_{\text{broker cost}} + \underbrace{\left(qm \log n + q(m \log \frac{n}{Q} + \log P)\, G + \log \frac{n}{q}\, L\right)}_{\text{server cost}}. \tag{3}$$

The first term in the server cost corresponds to the binary pattern searches, composed of a first internal search cost of $qm \log \frac{n}{P}$, and a second inter-processor search cost of $qm \log P$. The second term corresponds to the total cost of $qm \log \frac{n}{Q} G$ of fetching the suffixes to decide the binary comparisons, plus the $q \log P\, G$ cost of the inter-processor binary searches. The third term corresponds to the $\log \frac{n}{Q}$ supersteps required by the internal searches plus the $\log P$ supersteps required by inter-processor searches.

Overall, the complexity is slightly worse than that of the plain Global strategy. The advantage, however, is that this strategy does not suffer from load imbalance: The queries are distributed uniformly across processors regardless of how biased they are. We also recall that in practice most of the $qm \log \frac{n}{Q} G + \log \frac{n}{Q} L$ cost is avoided thanks to the pruned suffixes.

3.2.2. An Intermediate Distribution Strategy

An intermediate strategy between Global and Multiplexed, called VMultiplexed, can be obtained by applying the Global strategy over $V = 2^k P$ *virtual* processors, for some $k > 0$. Each virtual processor i is mapped on the real processor $i \bmod P$, for $i = 0, \ldots, V - 1$. In this circular mapping, each real processor ends up with $V/P = 2^k$ different intervals of n/V elements of the global array. This tends to break apart the imbalance introduced by biased queries, whereas the inter-processor search to refine the intervals becomes unnecessary (just as in Global, since the intervals of the virtual processors are contiguous in the global suffix array). The only cost is that the broker must also search for the correct (virtual) processor among V delimiting suffixes instead of P, thus k must be small enough to avoid a memory bottleneck in the broker.

Example 6. Figure 2(d) shows how this strategy distributes the suffixes in our running example, for $k = 1$, that is, each processor simulates two virtual processors, and thus it contains two intervals of the global suffix array.

13

The cost of the VMULTIPLEXED strategy is thus

$$\underbrace{(Qm + QmG + L)}_{\text{broker cost}} + \underbrace{\left(qm \log \frac{n}{2^k P} + qm \log \frac{n}{2^k Q}G + \log \frac{n}{2^k Q}L\right)}_{\text{server cost}}, \tag{4}$$

which only improves with k. As explained, the memory of the broker is the practical limit to how large k can be (as it needs to store $2^k P$ delimiter strings).

3.2.3. Global Suffix Array with Local Text

As we have seen, with MULTIPLEXED we are able to uniformly distribute queries among processors, but we must store the pruned suffixes to avoid (or reduce) the references to remote suffixes. Strategy LOCAL, on the other hand, avoids remote references, yet has poor concurrency. We now present a suffix array distribution strategy that solves both the load imbalance and references to remote suffixes. The idea is to redistribute the original global array so that the local array stored in processor P_j contains only pointers to the local text corresponding to processor P_j, as shown in Figure 2(e). In this way, the pruned suffixes are not needed. We call this strategy GLOCAL. As opposed to MULTIPLEXED, the local suffix arrays in GLOCAL are formed by a non-regular sampling of the global suffix array (see Figure 3(b)).

This strategy has the advantages of LOCAL, as it only contains pointers to local text. The difference is that in this case *the local arrays are constructed from the global array*. Hence, we can keep global information—such as, for instance, the order of the global array entries—that will help us avoid the P parallel binary searches (for the same pattern) and broadcasts per query that are carried out by LOCAL.

Just as for MULTIPLEXED, a local search yields a global-suffix-array interval that is usually a subinterval of the one containing the pattern occurrences. Unfortunately, because of the non-regular sampling of the global suffix array, Property 1 is not valid for GLOCAL. Therefore, we are no longer able to carry out the inter-processor binary search as in MULTIPLEXED. To support an efficient search we need, instead, to store extra information about where the remaining array cells are stored. We have illustrated this in Figure 2(e), where we show (in a separate array below each local suffix array) the global position of the corresponding local position. Thus, for instance, we have that $SA_3[1]$ corresponds to the global entry $SA[1]$, as indicated by ①. An important difference with the MULTIPLEXED strategy is that, for the GLOCAL approach, the sampling of the global array is not uniform, as it can be seen in Figure 3(b). Hence, in a pathological case, the global difference between two consecutive local array entries can be up to $O(n)$.

Search Algorithm. We propose an $O(\ell \frac{n}{2^\ell} G)$ cost strategy to perform the inter-processor search, at the cost of storing ℓ values (or "pointers") per suffix array entry (instead, we do not need to store pruned suffixes). The method works for any $\ell \geq 1$, as follows. For $\ell = 1$, each local suffix array entry stores a pointer to the next global array entry. Each pointer consists of the processor that owns the next global entry, plus the local address of that entry within the local suffix array. Hence, every pointer requires $\log P + \log \frac{n}{P} = \log n$ bits of space, for a total of $n \log n$ bits. This can be thought of as a linked list of the global suffix array entries. After we perform the initial local binary search using Algorithm 1, we must follow the linked list in order to find the precise endpoints of the interval containing the occurrences. The cost is one superstep per linked-list node traversed.

Hence, once a processor P_j finds the answer between its local consecutive cells $SA_j[i'_1..i'_2]$, mapped to global cells $SA[i_1, i_2]$, it retrieves from its linked-list information the processor $P_{j'}$ that owns cell $i_2 + 1$, as well as the position of that cell in the local suffix array of processor $P_{j'}$. Then P_j requests $P_{j'}$ to determine whether its text pointed by suffix array cell $i_2 + 1$ is lexicographically larger than the query. If it is so, then $i + d$ is the right answer. If it is not, then $P_{j'}$ is now in charge of finding the right position, by finding the processor $P_{j''}$ that owns cell $i_2 + 2$, and so on. The process is similar for $i_1 - 1$, $i_1 - 2$, etc. (as we do not have "previous" pointers but only "next" in the lists, we must run the search from the local cell $i'_1 - 1$). On average, the distance to traverse in the global suffix array between two cells that are consecutive in some local suffix array is $O(P)$, but in a bad case it can be as large as n, and the cost of the search is one superstep per traversed cell. Note that, as before, all the Q patterns can be broadcast to all the machines in $QmG + L$ time, so they do not need to be transmitted later.
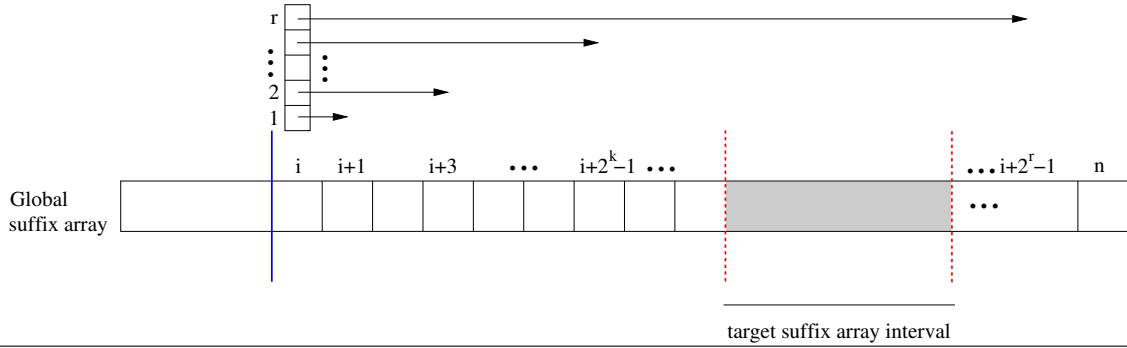
Figure 4: Extra information stored by each suffix array entry $SA[i]$. This information allows us to search the global interval $SA[i..i + 2^r - 1]$ in order to quickly find the processor storing the starting/ending point of the target interval containing the answers to the query.

This can be improved for larger ℓ as follows. Each suffix array entry $SA[i]$ keeps ℓ pointers to the global entries $SA[i + 1]$, $SA[i + 3]$,..., $SA[i + 2^\ell - 1]$, such that $i + 2^\ell - 1 \leq n$. See Figure 4 for an illustration. The extra space is then increased by $\ell n \log n$ bits. Note that the pointer to the next entry $SA[i + 1]$ is essential to ensure the correctness of the algorithm, because this is the only efficient way to know where the next suffix array entry is. That is, we need at least $n \log n$ extra bits on top of the suffix array itself. The remaining $\ell - 1$ pointers are just optional accelerators.

To achieve a performance comparable to that of the inter-processor binary search of the MULTIPLEXED strategy, assume that processor P_j finds that the target interval starts between the global suffix array entries $i - s$ and i, which are consecutive in its local suffix array. Hence, we must find the largest k such that $2^k - 1 < s$, and then find the processor $P_{j'}$ owning the entry $SA[i - 2^k + 1]$. If $P_{j'}$ answers that the query is lexicographically smaller than the suffix corresponding to its suffix array entry, then processor P_j sets $s \leftarrow 2^k - 1$ and goes on with $k - 1$. Otherwise, processor $P_{j'}$ takes care of the search, with $i \leftarrow i - 2^k + 1$, $s \leftarrow s - 2^k + 1$, and $k \leftarrow k - 1$.

In order to carry out the inter-processor search in optimal $O(\log n)$ steps, we must use $\ell = \log n$ pointers per suffix array entry, which implies an extra space of $n \log^2 n$ bits over the suffix array.

Cost Analysis. Using $\ell = \log n$, the interprocessor search can be supported in $O(\log n)$ supersteps under this distribution strategy. Thus the cost of this strategy is given by

$$\underbrace{(Qm + mQG + L)}_{\text{broker cost}} + \underbrace{(qm \log n + q \log n\, G + \log n\, L)}_{\text{server cost}}. \tag{5}$$

Compared with Equations 1 and 3, we can see the effect of using local suffixes (i.e., the communication costs are not multiplied by m), and, in exchange, the cost of the irregular sampling (i.e., the searches require $\log n$ supersteps, not $\log \frac{n}{q}$). We remind that pruned suffixes may make strategies GLOCAL and MULTIPLEXED perform better than what our complexity formulas suggest.

A Space-Efficient Way to Represent the Inter-Processor Pointers. Although strategy GLOCAL can achieve $O(\log n)$ supersteps to find the target interval containing the answers to a query, in practice we may not have enough space to reach this optimum (recall that we need $O(\log n)$ pointers per suffix array entry). We introduce a space-efficient alternative to represent these pointers, based on the use of bit vectors. Given a bit vector $B[1..n]$, we define the operation $\mathsf{rank}_1(B, i)$ as the number of 1s occurring in $B[1..i]$. The operation $\mathsf{select}_1(B, j)$ is defined as the position of the j-th 1 in B. If n' is the total number of 1s in B, then the data structure of Raman et al. [56] requires $n' \log \frac{n}{n'} + O(n') + o(n)$ bits of space, and supports

the above operations in $O(1)$ time. An alternative data structure by Okanohara and Sadakane [53] requires $n' \log \frac{n}{n'} + 1.92n' + o(n')$ bits of space, and supports rank in $O(\log \frac{n}{n'})$ time and select in $O(1)$ time[7]. .

We propose to replace the ℓ pointers per suffix array entry by a single bit vector per processor. This will reduce the space usage significantly. Let $B_j[1..n]$ denote the bit vector stored at processor P_j, such that $B_j[i] = 1$ if and only if processor j stores the global suffix array entry $SA[i]$. In other words, bit vector B_j keeps track of the global array entries stored at processor P_j. Figure 2(e) shows the bit vectors associated to each processor for our running example. Notice how each bit vector marks with a 1 the global positions that are indexed at each processor. For instance, in our example we have that, in processor P_3, $SA_3[2]$ corresponds to the global entry $SA[9]$ (as the 2nd 1 in the bit vector is $B_3[9]$). Conversely, given the global position $SA[9]$, we know that processor P_3 stores it, as $B_3[9] = 1$. Also, $SA[9]$ corresponds to $SA_3[2]$, as there are 2 bits set to 1 in $B_3[1..9]$.

Notice that each bit vector has $\frac{n}{P}$ bits set to 1. We represent B_j using the $\mathtt{sarray}$ data structure of Okanohara and Sadakane [53] for sparse bit vectors, which uses $\frac{n}{P} \log P + 1.92 \frac{n}{P} + o(\frac{n}{P})$ bits of space per processor. This amounts to $n \log P + 1.92n + o(n)$ bits overall for the P processors. This means that we require only about $\lceil \log P + 1.92 \rceil$ *bits* per suffix array entry. Note that, using pointers, even for $\ell = 1$ each suffix array entry needs $\log n$ bits. Moreover, by using the bit vectors we will be able to simulate the performance of any $\ell \geq 1$, thus this reduced amount of information shall be enough to simulate the inter-processor pointers.

Overall, the local data stored by a processor P_j is the local suffix array SA_j, plus the bit vector B_j marking which global entries have been stored in SA_j. Given this representation, if $B_j[i] = 1$, the global entry $SA[i]$ equals the local entry $SA_j[\mathsf{rank}_1(B_j, i)]$. On the other hand, given a local entry $SA_j[i']$, the corresponding global entry is $SA[\mathsf{select}_1(B_j, i')]$. In other words, by using rank and select we can efficiently map between local and global positions (and vice versa). We show next how to use, at search time, the information provided by the bit vectors.

Search Algorithms for the Space-Efficient Version. Let P_j be the processor to which a given search pattern X has been assigned. Let $SA_j[i'_1, i'_2]$ be the local suffix array interval corresponding to X, found using Algorithm 1 on SA_j. Let $SA[i_1..i_2]$ be the corresponding global suffix array interval, computed using $i_1 = \mathsf{select}(B_j, i'_1)$ and $i_2 = \mathsf{select}(B_j, i'_2)$. Further, let $i_1 - s = \mathsf{select}(B_j, i'_1 - 1)$ be the global suffix array cell corresponding to local cell $i'_1 - 1$. Our idea is to binary search the global interval $SA[i_1 - s..i_1]$ in order to find the starting point of the target interval (the process is similar for i_2). Notice the difference with the MULTIPLEXED strategy, where the global interval $SA[i_1 - P + 1..i_1]$ is binary searched instead. Moreover, this time we do not know in advance which processor stores each of the suffix array entries $SA[i_1 - s..i_1]$.

First, processor P_j asks (using a broadcast) for the processor $\mathsf{P}_{j'}$ owning the global entry $SA\left[i - \frac{s}{2}\right]$ (i.e., processor $\mathsf{P}_{j'}$ is such that $B_{j'}\left[i - \frac{s}{2}\right] = 1$), spending one extra superstep. In the next superstep, only processor $\mathsf{P}_{j'}$ answers to P_j Thus, P_j needs two supersteps to reconstruct the pointer to entry $SA\left[i - \frac{s}{2}\right]$. Processor P_j then sends the query to $\mathsf{P}_{j'}$, which goes on with the binary search. At each step, the current processor tries to reduce as much as it can the search interval, by using the 1s in its own bit vector, and then it obtains the corresponding pointer (spending two extra supersteps per pointer). Notice that we can simulate the effect of any value of ℓ, without requiring any extra space. In particular, we choose the optimum $\ell = \log n$. Thus, we add $2 \log n$ extra supersteps in the worst case, and an extra communication cost of $P \log n\, G$ per query (because of the broadcasts at every step in the search). Therefore, the cost is as in Equation (5), except that the communication cost raises to $Q \log n\, G$.

Table 1 shows a comparison of the search performance of our techniques to distribute suffix arrays.

---

[7]If we use a constant-time rank/select data structure (e.g., by Munro [49]) on its internal bitmap H.

Table 1: Comparison of the cost for the different alternatives to distribute suffix arrays. The broker cost is always $Qm+QmG+L$, dominated by the server cost. For strategy GLOBAL we assume a uniform distribution of queries over the processors.

Method	CPU cost	Communication cost (G)	Synchronization cost (L)
Sequential	$Qm \log n$		
GLOBAL	$qm \log \frac{n}{P}$	$qm \log \frac{n}{Q}$	$\log \frac{n}{Q}$
LOCAL	$Qm \log \frac{n}{P}$	Qm	1
MULTIPLEXED	$qm \log n$	$q(m \log \frac{n}{Q} + \log P)$	$\log \frac{n}{q}$
VMULTIPLEXED	$qm \log \frac{n}{2^k P}$	$qm \log \frac{n}{2^k Q}$	$\log \frac{n}{2^k Q}$
GLOCAL	$qm \log n$	$q \log n$	$\log n$

4. Experiments

In this section we experimentally compare the alternatives for distributing suffix arrays, and the respective query processing algorithms, that we have proposed.

4.1. Experimental Setup

Our experiments are carried out on five data collections:

1. A 300-megabyte XML collection from the *Pizza&Chili Corpus* [24]. The resulting global (or sequential) suffix array for this text requires 2.50GB;
2. a 400-megabyte DNA text from the *Pizza&Chili Corpus*. This is a sequence of symbols 'A', 'C', 'G', and 'T', plus a few occurrences of other special characters. The resulting suffix array requires 3.16GB;
3. a 200-megabyte text collection crawled from the Chilean Web (`http://www.todocl.cl`). The resulting suffix array requires 1.50GB. This text allows us to test the proposed algorithms over natural-language applications.
4. a 390-megabyte text with characters generated at random, giving a higher probability to the four most popular Spanish characters ('A', 'C', 'M', and 'P'). The corresponding frequencies are: 7.80% for letter 'A', 12.00% for letter 'C', 8.40% for letter 'M', and 9.30% for letter 'P'. We call *4Letters* this text, and the resulting suffix array requires 3.10GB. This collection is suitable for testing our algorithms in situations of load imbalance; and
5. a 395-megabyte Uniform text collection, where each character has the same probability of occurrence. The size of the suffix array obtained for this collection is 3.10GB. This synthetic text is designed to test the case of unbiased queries.

Overall, these datasets yield suffix arrays containing more than $1,500$ million elements.

We used $q = 50,000$ queries per processor—we tried with several values of $q \leq 900,000$, and obtained basically the same conclusions as the ones presented in this paper. We used patterns of length 5 and 10 extracted from each text at random positions (later in this section we also show results for patterns of length 100). The only exception is for the Chilean Web, where we used queries from a query log of the `todoCl` Web search engine (`http://www.todocl.cl`). These queries were truncated to the corresponding length.

Our results were obtained on the HECToR cluster (`www.hector.ac.uk`) with 1,856 nodes. Each node has two 12-core AMD Opteron 2.1GHz Magny Cours processors, sharing 16GB of memory. The hardware used for communication among cluster nodes is Infiniband, with an MPI point-to-point bandwidth of at least 5 MB/second. The latency between two nodes is around 1.0–1.5 microseconds. We implement our algorithms using the BSPonMPI library [9]. We test with $P = 2, 4, 8, 16, 32, 64$, and 128 processors. We assume that all the data (both text and suffix array) are kept in main memory.

In order to fairly compare all the algorithms, the sequential algorithm also solves $Q = qP$ queries. In all cases we show results normalized in order to (1) better illustrate the comparative performance among them and (2) present a global view of comparative performance across several figures covering all datasets tested [36, 34, 41]. To this end, in each figure, we divide all the experimental times by the observed maximum in

Table 2: Total size of the bit vectors added to GLOCAL to implement the inter-processor pointers. Sizes are shown as a fraction of the space used by the pointer implementation (for $\ell = 1$) and of the text size.

Text	Processors	Size as a fraction of pointers ($\ell = 1$)	Size as a fraction of text size
DNA	$P = 2$	0.15	0.61
	$P = 128$	0.48	1.93
Chilean Web	$P = 2$	0.16	0.63
	$P = 128$	0.51	2.03

the particular experiment. In this way, we can easily estimate the percentage of improvement achieved by all strategies in each experiment with respect to the sequential baseline and across datasets. It is important to note that we measure the whole processing time for queries. This includes the CPU time needed to solve a query, as well as the time used for inter-node communication. In our experiments, the fraction of the total time corresponding to communication cost grows logarithmically with the number of processors, from 5% of the total time with $P = 2$ processors to 35% with $P = 128$.

4.2. Implementing GLOCAL

Recall that in Section 3.2.3 we proposed two ways to implement the inter-processor pointers for GLOCAL: either using raw pointers or using sparse bit vectors. We study now what is the best way to implement the GLOCAL strategy, both in space usage and query time.

Table 2 shows the space usage of the bit-vector implementation of the inter-processor pointers, both as a fraction of the original pointers using $\ell = 1$ (that is, storing just a single pointer to the next suffix array entry, assuming that 32 bits are used for each pointer) and as a fraction of the text size. We show results just for DNA text and the Chilean Web; the other texts yield similar results. Notice that the space occupied by the MULTIPLEXED strategy is similar to the space occupied by GLOCAL with raw pointers and $\ell = 1$.

As it can be seen, the bit vectors use up to half the space of the single pointers for $P = 128$. In addition, the bit vectors support binary searching the processors to find the target suffix array interval. Pointers for $\ell = 1$, on the other hand, use more space and support only sequential search (after the initial internal binary search has been carried out). When compared with the text size, the bit vectors use from 0.6 ($P = 2$) to 2.0 ($P = 128$) times the space of the text. The latter also means that for $P = 128$ the bit vectors for GLOCAL use about the same space as for pruned suffixes with $t = 2$. This fact will be used when comparing GLOCAL with the techniques using pruned suffixes such as the MULTIPLEXED strategy.

Now we compare the query time of both implementations of GLOCAL. Figure 5 shows the running times for some of the tested collections, for patterns of length $m = 5$. We call GLOCAL-LIST the alternative using pointers (for $\ell = 1$), and just GLOCAL the alternative using bit vectors. As it can be seen, GLOCAL improves upon GLOCAL-LIST by about 20%–25% in all the tested collections, except DNA where the improvement is of about 35%. This shows that not only the bit vectors use less space, but also provide better performance at query time. As a result, in the experiments of the following sections we will use GLOCAL as the realization of the alternative using bit vectors.

4.3. Scalability Evaluation

In this section we test the scalability achieved by the distributed suffix array approaches on different data collections. Scalability is evaluated by increasing the workload (i.e., $Q = qP$ queries) as we increase the number of processors P. In this scenario, an algorithm obtaining optimal speedup achieves a stable running time as P grows. Otherwise the time increases with P; the time of an algorithm with no speedup increases linearly with P. This way of measuring is also interesting because it reflects the real-world scenario, where the number of processors will be increased according to the workload.

To provide a glimpse of the actual running times, Table 3 shows total running times (in seconds) required to process $50,000 \times P$ queries for three of the strategies proposed in this paper, with $P = 16$ and 128. The

(a) DNA.

(b) Chilean Web.
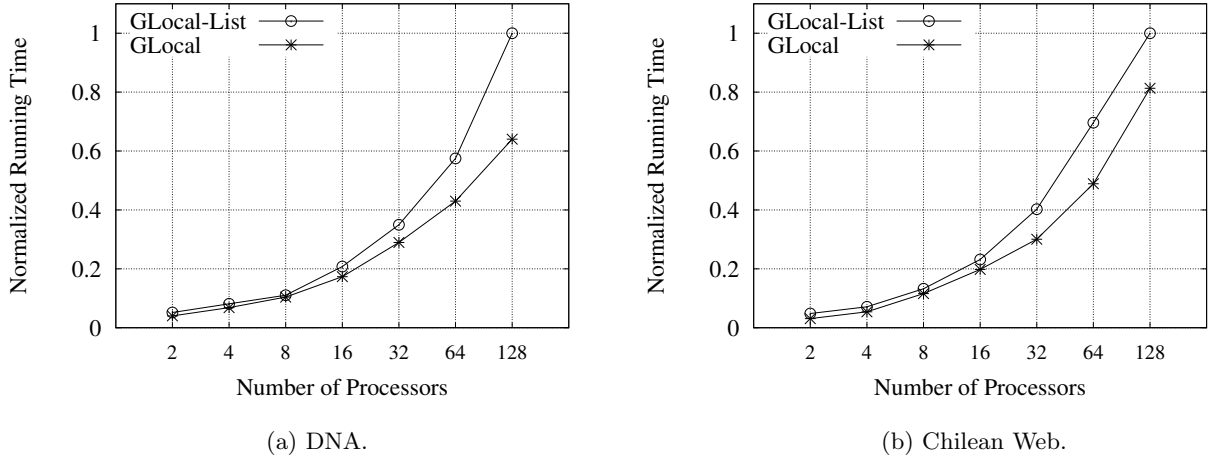
Figure 5: Performance evaluation of the GLOCAL strategy, using pointers to the next suffix array cell ("GLOCAL-LIST") and using bit vectors ("GLOCAL").

Table 3: Total running time (in seconds) required to process $Q = 50,000 \times P$ queries of length $m = 10$. GLOCAL implements the bit-vector strategy of Section 3.2.3.

P	Strategy	Collection				
		DNA	XML	4Letters	Chilean	Uniform
16	SEQUENTIAL	15.2	15.8	15.1	13.6	14.6
	LOCAL	3.8	3.2	2.3	2.9	2.8
	MULTIPLEXED	1.2	1.4	1.2	1.4	1.2
	GLOCAL	2.4	1.7	0.8	1.6	1.1
128	SEQUENTIAL	121.7	126.6	120.8	108.5	117.2
	LOCAL	17.1	22.9	10.8	10.1	9.4
	MULTIPLEXED	3.5	5.3	3.0	4.5	2.8
	GLOCAL	9.3	9.2	4.7	5.9	4.1

speedups obtained over the sequential algorithm is very clear. Translation to the other strategies should be straightforward from the normalized results we present later.

We use VMULTIPLEXED with parameter $k = 5$, and pruned suffix size of 4 characters for both MULTIPLEXED and VMULTIPLEXED.

4.3.1. Results on Uniform Text

We start with the Uniform collection, which displays an idealized scenario free from biases on the text and on the queries (recall that queries are generated from the text, so they follow a uniform distribution as well). Figure 6 shows the normalized running times of all the strategies, with patterns of length $m = 5$ in Figure 6(a) and of length $m = 10$ in Figure 6(b). It can be seen that all the global-partition algorithms (GLOBAL, MULTIPLEXED, VMULTIPLEXED and GLOCAL) show a much better performance than the LOCAL algorithm. This is because, as expected from the analysis, the cost of LOCAL grows as a function of Q rather than of q. Thus its time improvement over the sequential algorithm is explained not because each processor runs fewer queries, but because the queries run binary searches over smaller arrays, with higher locality of reference.

Among the global partitioning techniques, GLOCAL algorithm displays the best performance for $P \leq 16$, but it is outperformed by MULTIPLEXED for $P \geq 32$. One can expect that GLOCAL performs better than the others but degrades faster as P grows. Although it permits perfect load balance and avoids communication at all in the first part of the search, that is run on its local suffix array and local text, the second part of the search, when all the processors have to participate to refine the final interval, becomes heavier in cost with larger P (the interval to refine is of average size P).

This text is the best case for GLOBAL, which suffers from biased queries. Yet, even in this case, the small statistical deviations make some processors more loaded than others, and the total completion time per query is that of the slowest processor. This effect, plus the extra cost incurred by the broker to choose the right processor, makes the GLOBAL slightly slower than MULTIPLEXED, which can perfectly balance the work load. The latter is the fastest with large P, showing that the final binary search on P processors is not significant (this binary search is much lighter than the one performed by GLOCAL, because we know which processor owns each cell). As expected, the performance of VMULTIPLEXED stays between those of GLOBAL and MULTIPLEXED. Note that VMULTIPLEXED must also incur the cost of choosing the processor at the broker.

4.3.2. Results on Biased Synthetic Texts

Figure 6 also shows the results for the 4Letters text, with $m = 5$ in Figure 6(c) and $m = 10$ in Figure 6(d). Recall that this text was especially designed to produce biased queries. These results clearly show how sensitive is the GLOBAL strategy to biased queries. Its performance degrades sharply compared to that in the Uniform distribution. The other global strategies, instead, perform basically the same. This shows that the new strategies we have designed in this paper fulfill the goal of being resistant to biased queries.

Note that the degradation of GLOBAL is sharper on shorter patterns. This owes to a different problem: it is more likely that different processors handle the start and end of the suffix array interval of suffixes that start with a popular 5-letter pattern (especially as P grows and the local arrays are shorter). Thus two processors may have to get involved in a single search, whereas in the other global techniques we can always assign the query to a single processor.

4.3.3. Results on XML Text

Figures 7(a) and 7(b) show the normalized running time on the XML collection. The results are more or less between those of Uniform and 4Letters. On long patterns, the GLOBAL strategy performs as in the 4Letters text, owing to the same biased-queries phenomenon. On short patterns, however, the degradation observed in 4Letters does not occur, because the XML alphabet is much larger and thus it is much less likely that a query spans two processors.

As before, MULTIPLEXED performs best when sufficient processors are used. An interesting difference with the previous collections is that GLOCAL performs slightly worse on the real collections. A problem with GLOCAL that does not show up on our synthetic collections is that real texts tend to have some locality, and thus if we choose a processor to solve a query that rarely appears in its local text, the final interval it will have to refine with the other processors will be much larger.

4.3.4. Results on DNA Text

Figures 7(c) and 7(d) show the running time obtained by the algorithms on the DNA collection. This text should in principle behave close to the Uniform one. However, since DNA has a very small alphabet, the suffix array ranges for suffixes that start with the same character are very long. This impacts, much more than in 4Letters, on the performance of GLOBAL, which will frequently need to involve two processors in solving each query. The effect is also apparent in VMULTIPLEXED, which has the same problem, especially for $m = 10$, where the search cost is higher. As in the other cases, the MULTIPLEXED algorithm is the best choice when sufficiently many processors are involved.
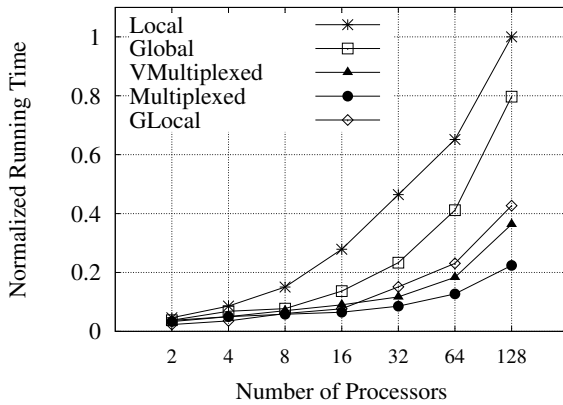
4.3.5. Results on Natural Language Text

Figures 7(e) and 7(f) show the performance on the Chilean Web, whose letter distribution is not very different from 4Letters but it is a real text using a real query log.
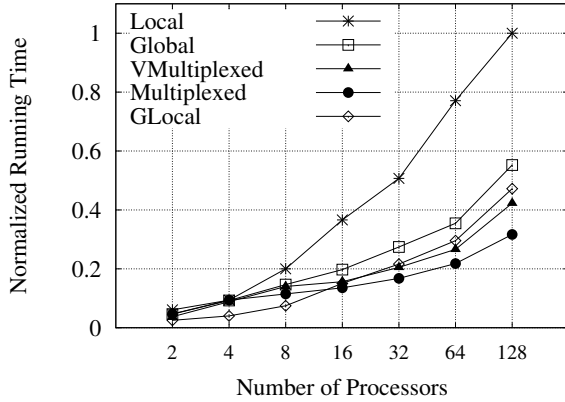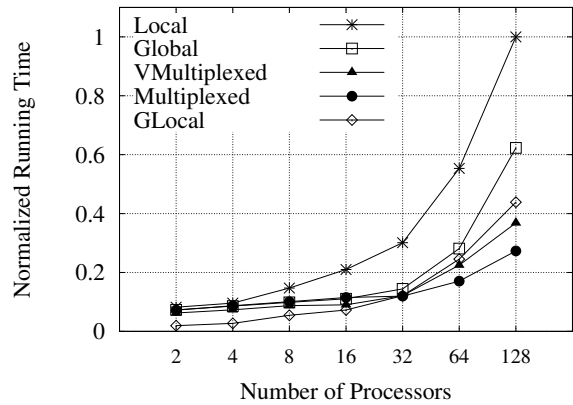
(a) Patterns of length 5 on Uniform.
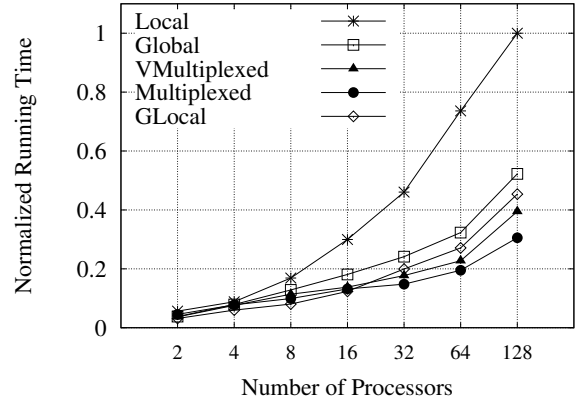
(b) Patterns of length 10 on Uniform.
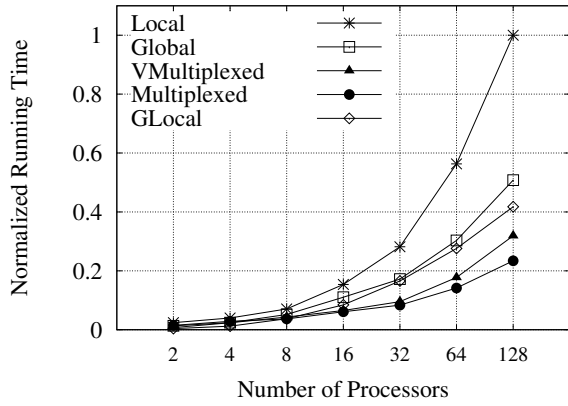
(c) Patterns of length 5 on 4Letters.

(d) Patterns of length 10 on 4Letters.

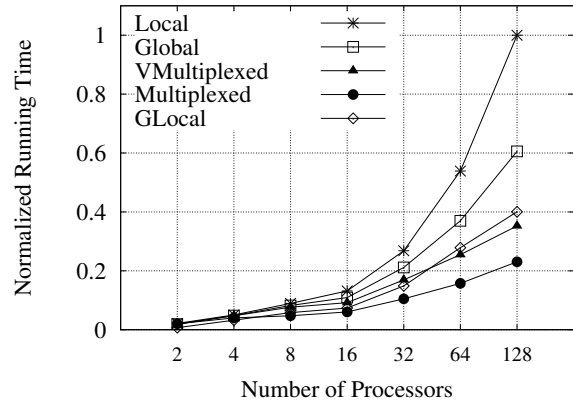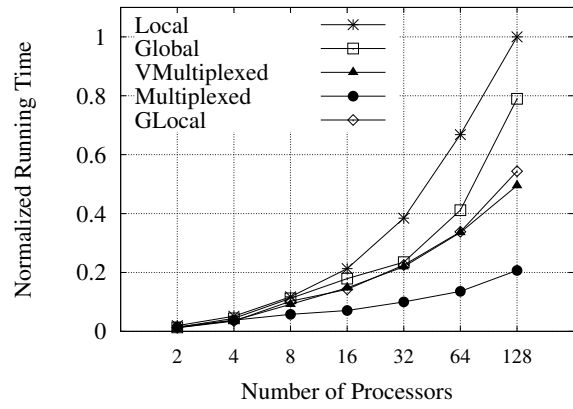Figure 6: Running times obtained for synthetic collections.

Once again, MULTIPLEXED outperforms the other strategies for $P \geq 16$. For $P \leq 8$, the GLOCAL algorithm is the best choice, improving upon MULTIPLEXED by 50% on average. While real-life text collections have locality (that is, some words are more frequent in some parts of the collection), and this impacts negatively on GLOCAL, its competitor, GLOBAL, suffers much more from the load imbalance of real-life queries. This is in contrast to the other texts, where we choose queries randomly from the text. We note that VMULTIPLEXED is effective in improving the performance of GLOBAL upon work imbalance, but still it is never better than both MULTIPLEXED and GLOCAL.

4.4. Load Balance Evaluation

Figure 8 reports results regarding the load imbalance for the LOCAL, MULTIPLEXED and GLOBAL strategies. To this end, let us denote T_s the time taken for a given superstep in the execution of a batch of queries. Let $T_{s,i}$ be the time spent by processor P_i at superstep s. We define the load balance for T_s as the efficiency ratio average$(T_{s,i})/\max(T_{s,i}) \leq 1$, for each processor i and averaged across supersteps. An efficiency value close to 1 means that the workload among processors is well distributed. A value close to the minimum

(a) Patterns of length 5 on XML.

(b) Patterns of length 10 on XML.
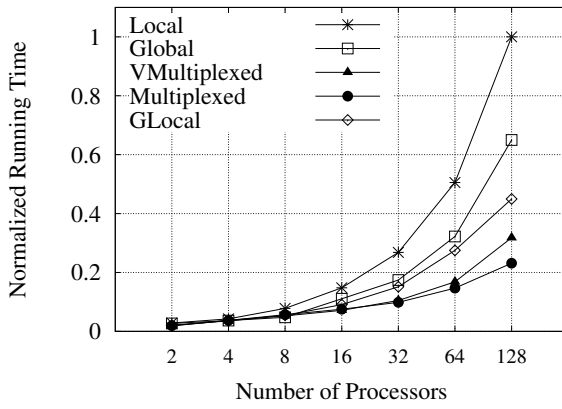
(c) Patterns of length 5 on DNA.

(d) Patterns of length 10 on DNA.
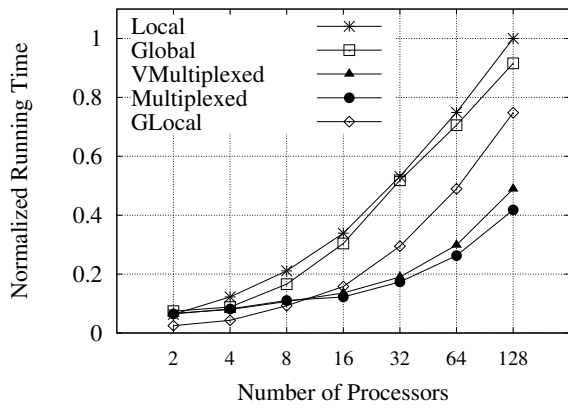
(e) Patterns of length 5 on the Chilean Web.
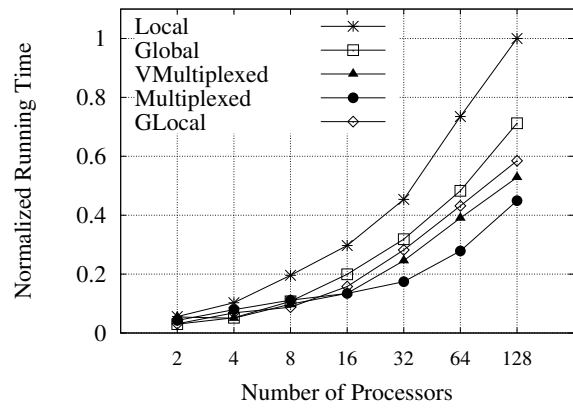
(f) Patterns of length 10 on the Chilean Web.

Figure 7: Running times obtained for the real collections.

$1/P$, on the other hand, indicates a poor load balance. We used patterns of length $m = 10$ executed on the 4Letters and Chilean Web datasets, which are our most skewed scenarios. In this experiment we keep fixed the total number of newly arriving queries per processor q, and vary the total number of processors P. The results show that the MULTIPLEXED strategy achieves competitive balance compared to the LOCAL strategy, with the advantage of requiring less computation and communication cycles per query. As expected, the GLOBAL strategy suffers from high imbalance due to its strong dependency on biased query patterns. Notice

(a) 4Letters dataset.

(b) Chilean Web dataset.
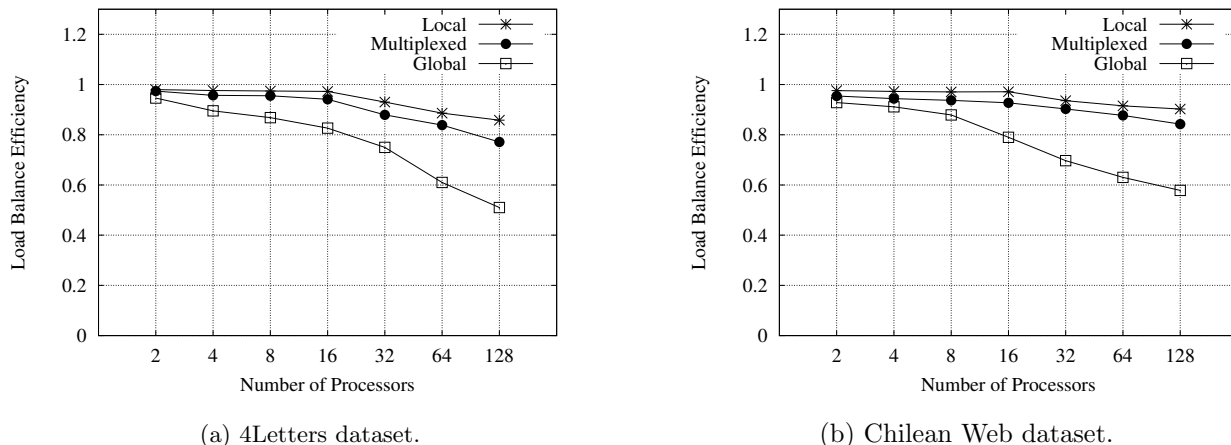
Figure 8: Load balance efficiency.

that even though in LOCAL each processor searches for the same queries, there is a slight imbalance. This is because the local texts and suffix arrays are different in each processor, and hence the number of character comparisons varies among processors. A similar effect can be observed in MULTIPLEXED.

4.5. Searching Big Texts and Long Patterns

We now study how the techniques scale with bigger texts and longer search patterns. We test with DNA (a 3.2GB text corresponding to the Human Genome [8]) and 4Letters (a 1.9GB version of this text). The resulting suffix arrays require 25.40GB for DNA, and 15.70GB for 4Letters.

Figure 9 shows the results on these big texts, for patterns of length 5 and 10. As it can be seen, we can conclude almost exactly as for previous experiments. The only effect of dealing with a larger amount of data is that the most efficient alternatives are slightly slower than for the smaller texts.

Figure 10 shows results for long patterns, $m = 100$. In this case, the performance slightly degrades when compared to the results of Figure 9. A reason for this is that longer patterns must be transmitted, increasing the communication overhead. There is, however, another reason: observe that the difference between MULTIPLEXED and GLOCAL is much smaller than for previous results (this is more noticeably for $P = 128$). This means that MULTIPLEXED degrades faster than GLOCAL. This is because we use pruned suffixes of length 4 for MULTIPLEXED. For long patterns, this increases the amount of pruned suffixes that need to be transmitted. Additional experiments with pruned suffixes of length 20 confirmed this fact.
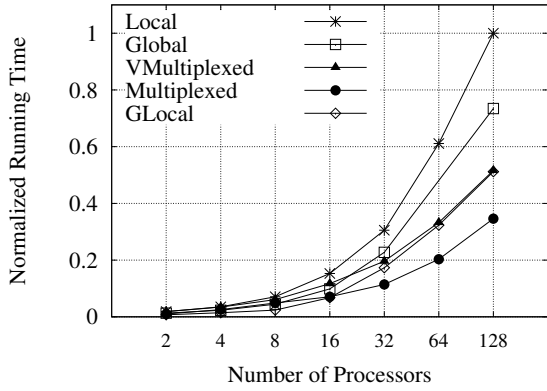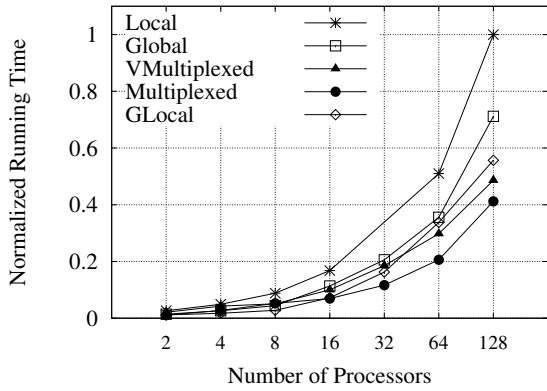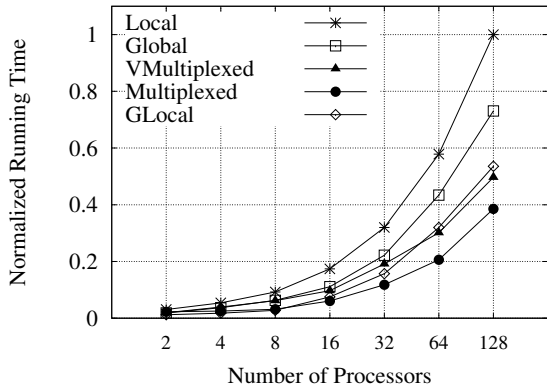

(a) Patterns of length 5 on DNA (3.2GB).
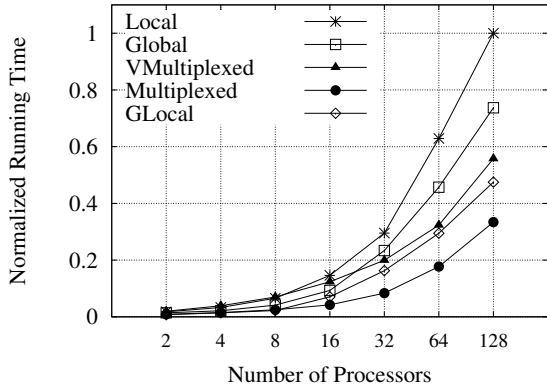
(b) Patterns of length 10 on DNA (1.9GB).

(c) Patterns of length 5 on 4Letters (1.9GB).

(d) Patterns of length 10 on 4Letters (1.9GB).

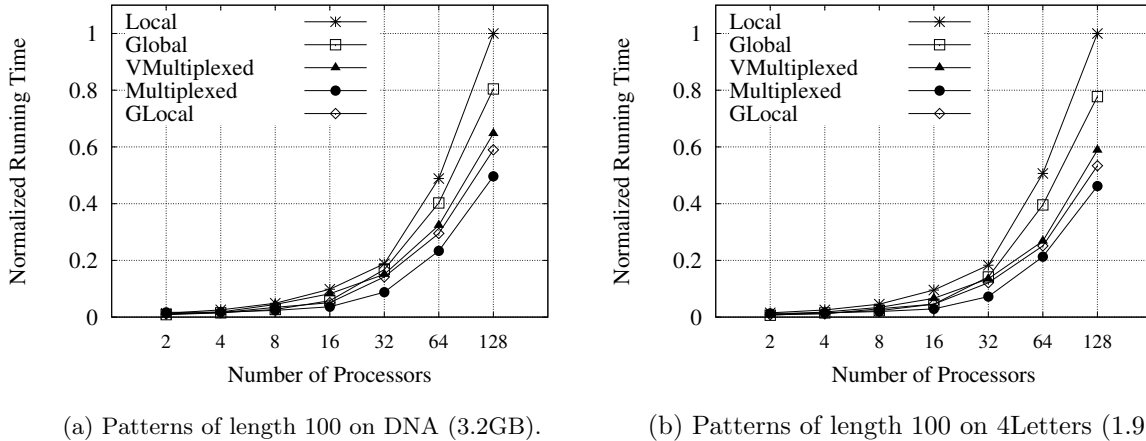Figure 9: Running times obtained for big text collections.

(a) Patterns of length 100 on DNA (3.2GB).   (b) Patterns of length 100 on 4Letters (1.9GB).

Figure 10: Running times obtained for patterns of length $m = 100$.

4.6. Efficiency Evaluation

In this section we compare the local and global indexing strategies in terms of classical parallel efficiency E_f, which is defined as the speedup divided by the total number of processors:

$$E_f = \frac{1}{P} \; \frac{\text{total sequential running time}}{\text{total parallel running time}}.$$

We show results for all the datasets. In these experiments, we no longer increase the total number of queries $Q = qP$ with the number of processors. Instead, we keep the number of queries constant in $Q = 50,000 \times 128$.

Figures 11(a)–(e) show the results for $m = 5$. The GLOCAL strategy has the best efficiency when $P \leq 16$: It is always over 0.8, and in many cases near 0.95. For larger P, as expected from previous experiments, MULTIPLEXED takes over, keeping efficiency over 0.7 even for $P = 128$. In all cases, VMULTIPLEXED stays as a close second best, thus it is interesting if we have to stick to one strategy for every P. We note that the efficiency degrades fastest on LOCAL, as expected, and GLOBAL is the second worst strategy in terms of efficiency. The other strategies degrade much slower.

Note also that, in general, the worse efficiency with few processors is obtained on the Chilean Web dataset, which emphasizes the problems posed by skewness in real-life queries. With more processors, however, the difference with the other collections is not so marked.

Figure 11(f) shows that efficiencies are generally even better for longer patterns, for example improving up to 14% for large P in the 4Letters dataset. A similar effect is observed in the other datasets.

### 4.7. Case Study: A Distributed Text Search Engine

We finish with a complex realistic scenario where the cost of the memory consumption of the different strategies has a direct impact on performance. We consider a distributed suffix array supporting a complementary pattern search service in a Web search engine. More precisely, we use suffix arrays to perform substring searches in the vocabulary table supporting the main search index. This allows, for example, offering autocompletion searches based on substrings and not only prefix matches.The vocabulary is made up of all terms found in the Web sample kept in the search engine. In practice this means searching over tens of millions of strings, and thereby parallelization on distributed memory is a sensible approach. This time, the relatively small size of the collection will allow us maintaining several copies of each data partition. The more memory-efficient approaches will be able to use more copies within the same space, which will increase their throughput.
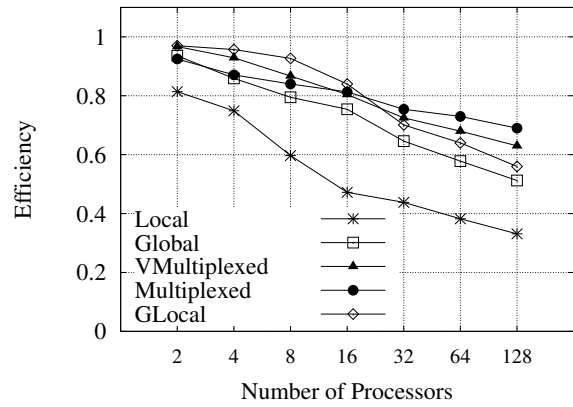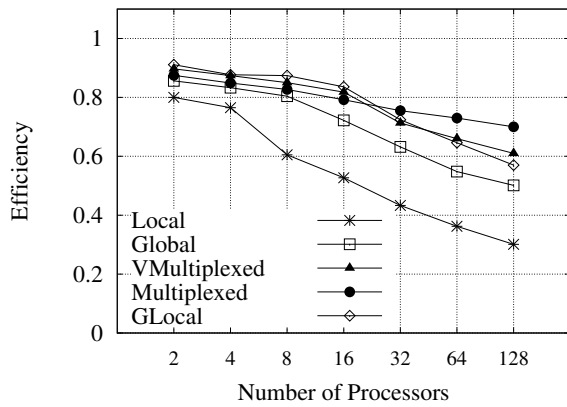
25
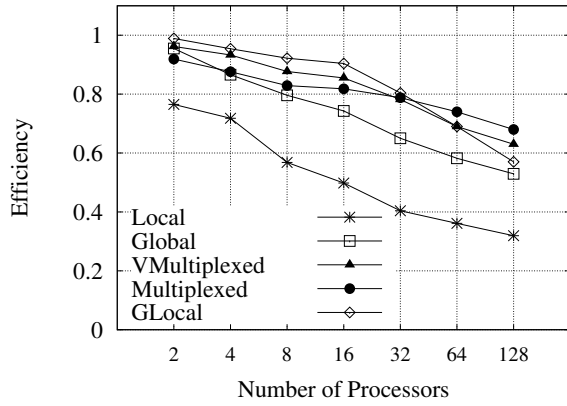
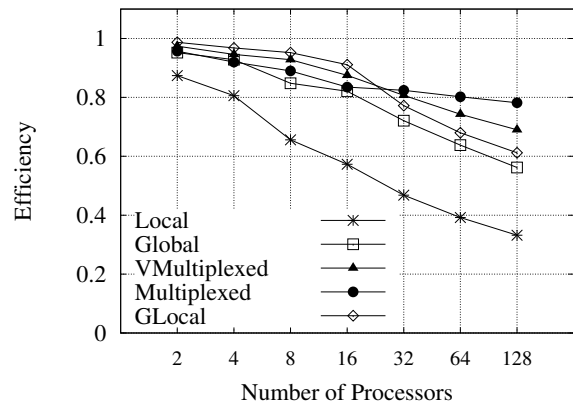(a) Uniform dataset.
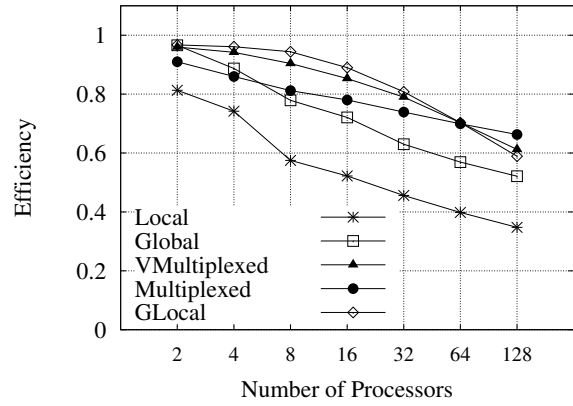
(b) 4Letters dataset.
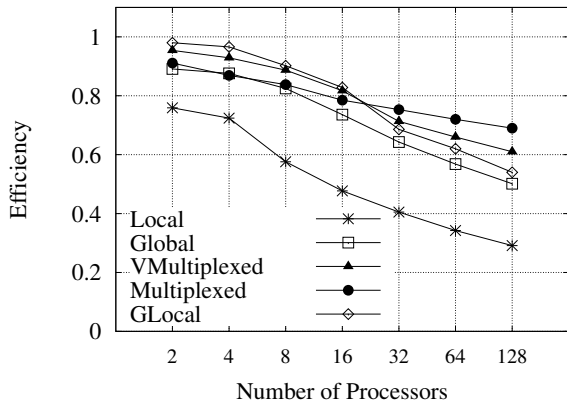
(c) XML dataset.

(d) DNA dataset.

(d) Chilean Web dataset.

(e) 4Letters dataset with $m = 10$.

Figure 11: Parallel efficiency E_f, for $m = 5$.

In this context, it is relevant to be efficient at processing large streams of queries. To this end, typically for this application domain, the P processors supporting the search service are organized as a $P = p \times r$ two-dimensional array, where p is the number of partitions in which the text $T[1..n]$ is divided, and r is the number of copies (replicas) of each partition. That is, the suffix array is partitioned into p parts of size n/p each, and we maintain r replicas of each part. This kind of setting is also useful for fault tolerance in search engines [11]. We compare the traditional LOCAL partitioning strategy with MULTIPLEXED, which has performed best in general in our previous experiments.

Let $T_r(p) = \max\{d_i + s_i(p) + G\}$ be the *individual query response time* of a query, where d_i is a processor delay producing a query waiting time, s_i is the query service time, G is the communication latency, and the maximum is taken over all the processors i involved in solving the query in parallel. The values of p and r of the system are set to meet the following three requirements [18]:

1. Ensure that the individual query response time is bounded above by $T_r(p) \leq B$, where B is a target upper bound. That is, we want to ensure that every query is solved within time B.
2. Ensure that the system query throughput $X_0(p,r)$ is capable of achieving $X_0(p,r) \geq \lambda_0$, for a given target query arrival rate λ_0. That is, we want to ensure that the system can process the queries at the rate they arrive.
3. Minimize the total number of processors $P = p \times r$ provided that no processor utilization gets above a target utilization bound $U \leq 1$. In practice, search engines are over-provisioned so that they are prepared to cope with sudden peaks in query traffic [6, 18]. This means that they maintain processor utilization well below 100% at steady state operation.

For the LOCAL strategy on a $p \times r$ system, the Q queries are broadcast to the p partitions, as usual. In each such partition, every processor gets Q/r queries. For MULTIPLEXED, on the other hand, each partition gets Q/p queries, and each processor in a partition gets $Q/(pr)$ queries. That is, each processor in LOCAL gets p times the number of queries of a processor in MULTIPLEXED. As a result, more processors will be necessary to cope with requirements (1) to (3).

On the other hand, LOCAL uses less memory than MULTIPLEXED, since the former does not store auxiliary data such as pruned suffixes. Let us assume that MULTIPLEXED uses pruned suffixes of length 2^k (assuming 1 byte per symbol, and 4 bytes per integer) with $k \geq 2$. Given an amount of space used by MULTIPLEXED, strategy LOCAL requires fewer partitions (p) to hold the index, and thus can increase the number of replicas (r), improving its performance because each processor recieves fewer queries. To use about the same space in both cases, once we set p and r for for MULTIPLEXED, the number of partitions LOCAL can be decreased to $p^* = p/(1 + 2^{k-2})$, and thus the number of replicas can be increased to $r^* = r(1 + 2^{k-2})$.

The trade-off for requirement (2) can be stated as follows. Let us assume a server where processing a batch of Q queries takes time T_Q in steady-state operation, that is $X_0 = Q/T_Q$. Therefore, for LOCAL we have that T_Q is $O((Q/r)m \log(n/p) + (Q/r)mG + L) = O(qpm \log(n/p) + qpmG + L)$. Notice that this is a generalization of Equation (2), where $r = 1$ is assumed (and hence $p = P$ holds).

For MULTIPLEXED, on the other hand, the total query time T_Q is $O((Q/(pr))m \log n + (Q/pr)(m \log(n/Q) + \log p)G + \log(n/Q)L) = O(qm \log n + q(m \log(n/Q) + \log p)G + \log(n/Q)L)$. Again, notice that this is a generalization of Equation (3).

From the cost analysis above, and for fixed $P = p \times r$, it can be noticed how the cost of LOCAL can be improved if we increase the number of replicas r (i.e., we decrease p in the same proportion). This is not the case for MULTIPLEXED. This effect will be observed experimentally in what follows.

4.7.1. Processor Utilization

We first measure individual query response time $T_r(p)$, as we vary the query traffic intensity, such that the maximum processor utilization is 15%, 25% and 50%, approximately. We use MULTIPLEXED with 4 characters per pruned suffix, so LOCAL needs half the space and thus can use twice the number of replicas of MULTIPLEXED. Figure 12(a) shows the results for the three processor-utilization values tested, for a total of $Q = 50,000 \times P$ queries. The results indicate that only when the processor utilization is low (because we have many replicas) the LOCAL indexing strategy performs faster than MULTIPLEXED.
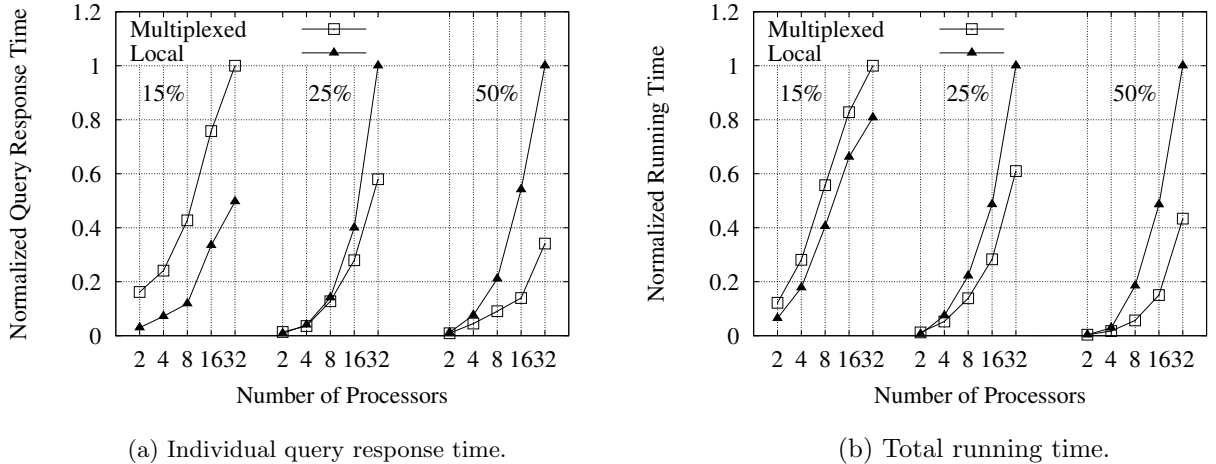
(a) Individual query response time.

(b) Total running time.

Figure 12: Experimental results for different query-traffic conditions, over the XML dataset, queries of length $m = 10$, and a total of $Q = 50{,}000 \times P$ queries. Similar results are obtained for the remaining datasets.

Figure 12(b) shows the effect of the different processor utilization thresholds in the total running time T_Q. The results yield similar conclusions. However, the differences in performance for very low utilization (15%) between LOCAL and MULTIPLEXED are at most 20%, whereas MULTIPLEXED is better suited for higher utilization. Finally, it can be seen that MULTIPLEXED scales up more efficiently with P. In the remaining experiments we use utilization $U = 50\%$.

4.7.2. Evaluation under Similar Memory Usage

We study more in depth the effect of varying the number of replicas. Figure 13 shows the results for $r = 1, 2, 4$ and $P = 2, 4, 8, 16, 32, 64$, where $p = P/r$ was adjusted consequently to achieve the target P indicated in the x-axis. The y-axis shows the normalized total running times required to process a total number of queries given by $Q = 50{,}000 \times P$, with $P = p \times r$. Figure 13(a) shows results for the Uniform dataset, and Figure 13(b) shows results for the 4Letters dataset.

Note that, for example, in LOCAL $r = 1$ we assume that each node holds n/P suffix array cells, whereas for $r = 2$ we assume that each node holds $2n/P$ cells, and $r = P$ assumes that each node holds a copy of the whole suffix array. So they are not comparable because they assume the nodes have different amounts of memory, and results are obviously better as we have more replicas.

What are comparable are the results for LOCAL with value $2r$ and MULTIPLEXED with value r, which assume the same amount of memory, as explained. The results show that, even with this memory handicap, the MULTIPLEXED strategy achieves better performance than the LOCAL strategy for both datasets.

5. Conclusions

In this paper we have presented alternative ways of efficiently performing parallel query processing on distributed suffix arrays. We have studied how to process large streams of on-line queries by using local and global indexing approaches to suffix array distribution on processors.

Local indexing is always appealing to data center engineers as it is simple to implement and deploy on production systems. The text is evenly distributed on processors and a local suffix array is constructed in each processor. Queries are broadcast to all processors and the partial results are then merged to produce the final results for each query. Our experimental study shows that this approach does not scale up efficiently with the number of processors. In the cases where replicas can be used, local indexing is only efficient for

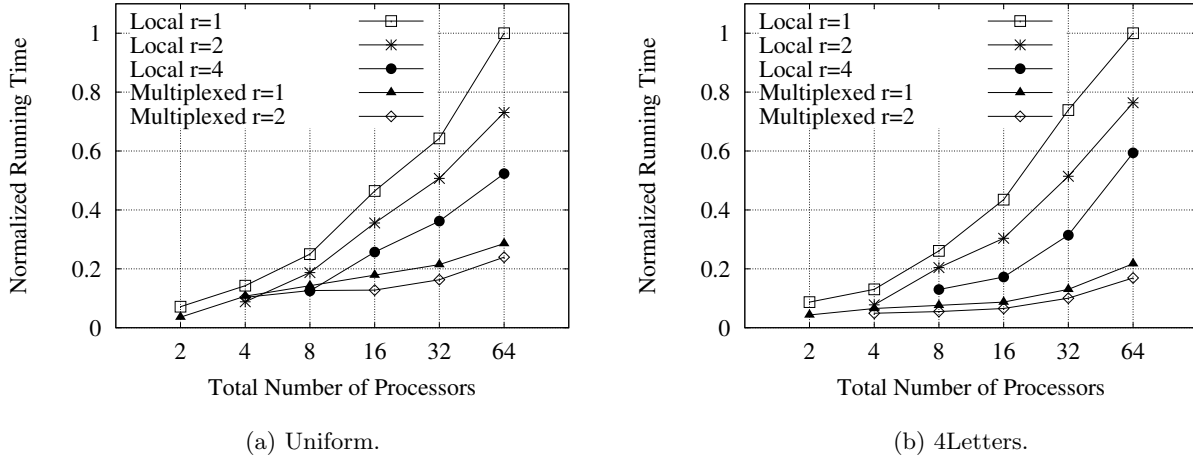|	(a) Uniform.	(b) 4Letters.

Figure 13: Comparison of strategies LOCAL and MULTIPLEXED for different number of replicas.

cases of excessive over-provisioning of hardware resources where processors are kept operating at very low utilization in steady state operation.

On the other hand, global indexing offers the potential of much better scalability than local indexing as queries are not required to reach all the processors. However, the straightforward realization of this strategy, that is, evenly distributing a global suffix array in lexicographic order on processors, is prone to significant load imbalance when query contents are biased. This arises in many realistic cases, most clearly in search engines, where the load imbalance is sharp and varies rapidly over time.

In this paper we have proposed three strategies to distribute global suffix arrays on processors and corresponding parallel query processing algorithms so that load imbalance is prevented. Our experimental study shows that each strategy has its own merits with respect to practical performance. The most efficient strategy in terms of running time under a large number of processors is MULTIPLEXED, where each processor holds a regular sampling of the suffix array and any processor can solve any query. Instead, the GLOCAL strategy achieves competitive performance by using a local-like partitioning so that the result can later be mapped to the global suffix array, and thus any processor can handle the query with its local text. For a small number of processors, the GLOCAL strategy is more efficient than the MULTIPLEXED strategy. A strategy called VMULTIPLEXED, a hybrid between plain global and MULTIPLEXED strategies, is always second-best and becomes a good choice when a single strategy must be chosen for any number of processors. We note that, for various of datasets, the GLOCAL strategy achieves similar performance than the VMULTIPLEXED strategy for a large number of processors, with the advantage of lower memory usage. We also tested a version of the MULTIPLEXED strategy where pruned suffixes were kept compressed to reduce memory usage. However, the cost of pruned suffix decompression required to support binary search was too detrimental to the performance.

The running time results for different datasets show that the MULTIPLEXED and GLOCAL strategies significantly outperform the local indexing strategy and the straightforward realization of the global indexing strategy. The datasets used in experimentation were chosen to expose global indexing to different degrees of imbalance. The new strategies developed in this paper were shown to be resistant to load imbalance, keeping high parallel efficiency even for 128 processors. Among those, MULTIPLEXED was the one showing the least degradation in efficiency. All of our new distribution strategies can be generated as an easy postprocessing on top of the existing distributed suffix array construction algorithms cited in Section 1.3.3 (which essentially build the GLOBAL partitioning).

We have focused on count queries in this paper, as they are the most interesting algorithmically. In

many realistic scenarios, it is also necessary to retrieve some or all of the occurrence positions, and even some text snippet around them. In this regard, it is fortunate that our best-performing strategies are also promising for this task: MULTIPLEXED and VMULTIPLEXED offer perfect balancedness for retrieving the occurrences, whereas GLOCAL is unique in that it can return the text snippets without requesting text from other processors. The amount of extra communication required to send all the results back to the broker is not too significant, however. Figure 14 shows that, even for short patterns of length $m = 5$ on a small alphabet (4Letters), where a fair amount of occurrences are found, the extra communication cost is very low. In this experiment we use the MULTIPLEXED strategy and $P = 2$ to $P = 128$ nodes.
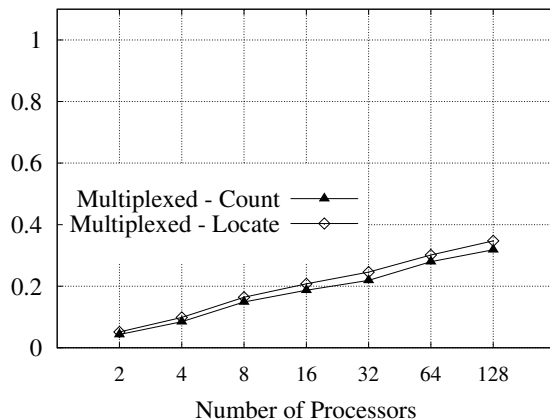

Figure 14: Comparison between the communication cost of count and locate queries. The cost is expressed as percentage of the total query time.

In this work we have only considered classical suffix array representations. There are, however, compressed representations of suffix arrays with the same functionalities as the classical ones, yet using space proportional to that of the compressed text [51]. Previous work [38] proposes a distributed implementation of *compressed suffix arrays* [25] that achieves $qm \log n + qm + qL$ processing time for $Q = qP$ queries. Notice that this is quite interesting compared to the complexities of Table 1, especially for short patterns. Our GLOCAL strategy proposed in the conference version [43] also inspired the development of compressed distributed suffix trees and arrays (in a theoretical stage) aimed at reducing the time of individual queries of more complex types, like computing matching statistics and maximal repeats [58].

Another variant of compressed suffix arrays, called FM-indexes [23], use a search strategy called *backward search*, which has nothing to do with a binary search. The FM-index represents a permutation of the text symbols plus little extra information. It needs access only to this permuted text to compute the suffix array interval that enables counting queries, thus there are no problems regarding accessing remote text positions. Backward search for a pattern of length m requires $2m$ access to arbitrary positions in this permuted text. Distributing this permuted text naturally yields a search algorithm with cost $qm \log \sigma + qmG + mL$. Implementing a distributed search on this compressed suffix array is a promising future work direction.

Acknowledgments

References

[1] Abouelhoda, M., Kurtz, S., Ohlebusch, E.. Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2004;2(1):53–86.

[2] Apostolico, A.. The myriad virtues of subword trees. In: Combinatorial Algorithms on Words. Springer-Verlag; NATO ISI Series; 1985. p. 85–96.

[3] Apostolico, A., Iliopoulos, C., Landau, G., Schieber, B., Vishkin, U.. Parallel construction of a suffix tree with applications. Algorithmica 1988;3:347–365.

[4] Badue, C.S., Baeza-Yates, R., Ribeiro-Neto, B., Ziviani, N.. Concurrent query processing using distributed inverted files. In: Proc. 8th International Symposium on String Processing and Information Retrieval (SPIRE). 2001. p. 10–20.

[5] Baeza-Yates, R.A., Ribeiro-Neto, B.A.. Modern Information Retrieval. 2nd ed. Pearson Education, 2011.

[6] Barroso, L.A., Hölzle, U.. The case for energy-proportional computing. Computer 2007;40(12):33–37.

[7] BSP PUB Library. BSP PUB Library at Paderborn University. `http://www.uni-paderborn.de/bsp`.

[8] BSP Standard. BSP World-wide Standard. `http://www.bsp-worldwide.org`.

[9] BSPonMPI Library. BSPonMPI Software Library. `http://bsponmpi.sourceforge.net`.

[10] Büttcher, S., Clarke, C., Cormack, G.. Information Retrieval: Implementing and Evaluating Search Engines. MIT Press, 2010.

[11] Cambazoglu, B., Catal, A., Aykanat, C.. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In: Proc. 21st International Symposium on Computer and Information Sciences (ISCIS). 2006. p. 717–725.

[12] Cambazoglu, B.B., Kayaaslan, E., Jonassen, S., Aykanat, C.. A term-based inverted index partitioning model for efficient distributed query processing. ACM Transactions on the Web (TWEB) 2013;7(3):15.

[13] Chen, C., Schmidt, B.. Parallel construction of large suffix trees on a PC cluster. In: Proc. 11th International Euro-Par Conference. LNCS 3648; 2005. p. 1227–1236.

[14] Cheng, L.L., Cheung, D.W.L., Yiu, S.M.. Approximate string matching in DNA sequences. In: Proc. of 8th International Conference on Database Systems for Advanced Applications (DASFAA). IEEE Computer Society; 2003. p. 303–310.

[15] Chung, S.H., Kwon, H.C., Ryu, K.R., Jang, H.K., Kim, J.H., Choi, C.A.. Parallel information retrieval on a SCI-based PC-NOW. In: Proc. Workshop on Personal Computers based Networks of Workstations (PC-NOW). 2000. .

[16] Clifford, R.. Distributed suffix trees. Journal of Discrete Algorithms 2005;3(2-4):176–197.

[17] Clifford, R., Sergot, M.. Distributed and paged suffix trees for large genetic databases. In: Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM). 2003. p. 70–82.

[18] Costa, V.G., Lobos, J., Inostrosa-Psijas, A., Marín, M.. Capacity planning for vertical search engines: An approach based on coloured petri nets. In: Petri Nets. 2012. p. 288–307.

[19] Dean, J.. Challenges in building large-scale information retrieval systems: invited talk. In: Proc. 2nd International Conference on Web Search and Web Data Mining (WSDM). 2009. p. 1.

[20] Farach, M., Ferragina, P., Muthukrishnan, S.. Overcoming the memory bottleneck in suffix tree construction. In: Proc. 39th Annual Symposium on Foundations of Computer Science (FOCS). 1998. p. 174–185.

[21] Farach, M., Muthukrishnan, S.. Optimal logarithmic time randomized suffix tree construction. In: Proc. 23rd International Colloquium on Automata, Languages and Programming (ICALP). LNCS 1099; 1996. p. 550–561.

[22] Ferragina, P., Luccio, F.. String search in coarse-grained parallel computers. Algorithmica 1999;24(3–4):177–194.

[23] Ferragina, P., Manzini, G.. Indexing compressed text. Journal of the ACM 2005;52(4):552–581.

[24] Ferragina, P., Navarro, G.. The Pizza&Chili corpus — compressed indexes and their testbeds. 2005. http://pizzachili.dcc.uchile.cl and http://pizzachili.di.unipi.it.

[25] Foschini, L., Grossi, R., Gupta, A., Vitter, J.. When indexing equals compression: Experiments with compressing suffix arrays and applications. ACM Transactions on Algorithms 2006;2(4):611–639.

[26] Fredkin, E.. Trie memory. Communications of the ACM 1960;3(9):490–499.

[27] Futamura, N., Aluru, S., Kurtz, S.. Parallel suffix sorting. In: Proc. of 9th International Conference on Advanced Computing and Communications. Tata McGraw-Hill; 2001. .

[28] Gonnet, G.H., Baeza-Yates, R.A., Snider, T.. New indices for text: Pat trees and pat arrays. In: Information Retrieval: Data Structures & Algorithms. 1992. p. 66–82.

[29] Gusfield, D.. Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press, 1997.

[30] Hariharan, R.. Optimal parallel suffix tree construction. Journal of Computer and Systems Sciences 1997;55(1):44–69.

[31] Iliopoulos, C., Korda, M.. Massively parallel suffix array construction. In: Proc. 25th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM). 1998. p. 371–380.

[32] Jeong, B.S., Omiecinski, E.. Inverted file partitioning schemes in multiple disk systems. IEEE Transactions on Parallel and Distributed Systems 1995;16(2):142–153.

[33] Kitajima, J., Navarro, G.. A fast distributed suffix array generation algorithm. In: Proc. 6th International Symposium on String Processing and Information Retrieval (SPIRE). 1999. p. 97–104.

[34] Kostoulas, M.G., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., Mercaldi, M.. XML screamer: An integrated approach to high performance XML parsing, validation and deserialization. In: Proc. World Wide Web Conference (WWW). 2006. p. 93–102.

[35] Kulla, F., Sanders, P.. Scalable parallel suffix array construction. Parallel Computing 2007;33(9):605–612.

[36] Lim, K.T., Turner, Y., Santos, J.R., AuYoung, A., Chang, J., Ranganathan, P., Wenisch, T.F.. System-level implications of disaggregated memory. In: Proc. 18th International Symposium on High Performance Computer Architecture (HPCA). 2012. p. 189–200.

[37] MacFarlane, A.A., McCann, J.A., Robertson, S.E.. Parallel search using partitioned inverted files. In: Proc. 7th International Symposium on String Processing and Information Retrieval (SPIRE). 2000. p. 209–220.

[38] Mäkinen, V., Navarro, G., Sadakane, K.. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In: Proc. 15th Annual International Symposium on Algorithms and Computation (ISAAC). Springer; LNCS 3341; 2004. p. 681–692.

[39] Manber, U., Myers, E.W.. Suffix Arrays: A new method for on-line string searches. SIAM Journal on Computing 1993;22(5):935–948.

[40] Mansour, E., Allam, A., Skiadopoulos, S., Kalnis, P.. ERA: efficient serial and parallel suffix tree construction for very long strings. Proceedings of the VLDB Endowment 2011;5(1):49–60.

[41] Marín, M., Costa, V.G., Bonacic, C., Baeza-Yates, R.A., Scherson, I.D.. Sync/async parallel search for the efficient design and construction of web search engines. Parallel Computing 2010;36(4):153–168.

[42] Marin, M., Gil-Costa, G.V.. High-performance distributed inverted files. In: Proc. 16th ACM Conference on Information and Knowledge Management (CIKM). 2007. p. 935–938.

[43] Marín, M., Navarro, G.. Distributed query processing using suffix arrays. In: Proc. of 10th International Symposium on String Processing and Information Retrieval (SPIRE). Springer; LNCS 2857; 2003. p. 311–325.

[44] McColl, W.F.. General purpose parallel computing. In: Lectures on Parallel Computation. Cambridge University Press; 1993. p. 337–391.

[45] Message Passing Interface. The Message Passing Interface Standard. http://www.mcs.anl.gov/research/projects/mpi.

[46] Moffat, A., Webber, W., Zobel, J.. Load balancing for term-distributed parallel retrieval. In: Proc. 29th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR). 2006. p. 348–355.

[47] Moffat, A., Webber, W., Zobel, J., Baeza-Yates, R.. A pipelined architecture for distributed text query evaluation. Information Retrieval 2007;10(3):205–231.

[48] Mohamed, H., Abouelhoda, M.. Parallel suffix sorting based on bucket pointer refinement. In: Proc. 5th Cairo International Biomedical Engineering Conference (CIBEC). 2010. p. 98–102.

[49] Munro, I.. Tables. In: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS 1180; 1996. p. 37–42.

[50] Navarro, G., Baeza-Yates, R.. A practical q-gram index for text retrieval allowing errors. CLEI Electronic Journal 1998;1(2). http://www.clei.cl.

[51] Navarro, G., Mäkinen, V.. Compressed full-text indexes. ACM Computing Surveys 2007;39(1):article 2.

[52] Navarro G. Kitajima, J., Ribeiro-Neto, B., Ziviani, N.. Distributed generation of suffix arrays. In: Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS 1264; 1997. p. 102–115.

[53] Okanohara, D., Sadakane, K.. Practical entropy-compressed rank/select dictionary. In: Proc. 8th Workshop on Algorithm Engineering and Experimentation (ALENEX). 2007. p. 60–70.

[54] Pace, M., Tiskin, A.. Parallel suffix array construction by accelerated sampling. CoRR 2013;abs/1302.5851.

[55] Parallel Virtual Machine. Parallel Virtual Machine Software Package. http://www.csm.ornl.gov/pvm.

[56] Raman, R., Raman, V., Rao, S.S.. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 2007;3(4):43.

[57] Ribeiro-Neto, B., Barbosa, R.. Query performance for tightly coupled distributed digital libraries. In: Proc. 3rd ACM International Conference on Digital Libraries (DL). 1998. p. 182–190.

[58] Russo, L., Navarro, G., Oliveira, A.. Parallel and distributed compressed indexes. In: Proc. 21th Annual Symposium on Combinatorial Pattern Matching (CPM). LNCS 6129; 2010. p. 348–360.

[59] Russo, L., Navarro, G., Oliveira, A., Morales, P.. Approximate string matching with compressed indexes. Algorithms 2009;2(3):1105–1136.

[60] Silvestri, F.. Mining query logs: Turning search usage data into knowledge. Foundations and Trends in Information Retrieval 2010;4(1-2):1–174.

[61] Skillicorn, D.B., Hill, J.M.D., McColl, W.F.. Questions and answers about BSP. Journal of Scientific Programming 1997;6(3):249–274.

[62] Skillicorn, D.B., Talia, D.. Models and languages for parallel computation. ACM Computing Surveys 1998;20(2):123–169.

[63] Tomasic, A., Garcia-Molina, H.. Caching and database scaling in distributed shard-nothing information retrieval systems. In: Proc. ACM International Conference on Management of Data (SIGMOD). 1993. p. 129–138.

[64] Valiant, L.G.. A bridging model for parallel computation. Communications of the ACM 1990;33:103–111.

[65] Weiner, P.. Linear pattern matching algorithms. In: Proc. 14th Annual Symposium on Foundations of Computer Science (FOCS). 1973. p. 1–11.

[66] Xi, W., Sornil, O., Luo, M., Fox, E.A.. Hybrid partition inverted files: Experimental validation. In: Proc 6th European Conference on Research and Advanced Technology for Digital Libraries (ECDL). 2002. p. 422–431.

[67] Zhang, B., Huang, Z.. A new parallel suffix tree construction algorithm. In: Proc. 3rd IEEE International Conference on Communication Software and Networks (ICCSN). 2011. p. 143–147.

[68] Zobel, J., Moffat, A.. Inverted files for text search engines. ACM Computing Surveys 2006;38(2):article 6.