Comparative Study of Parallel Suffix Arrays Algorithms

Mauricio Marín Josselyn Vega Rodrigo Miranda

Departamento de Computación
Universidad de Magallanes
Punta Arenas, Chile
Contact Author: mmarin@ona.fi.umag.cl

Abstract

This paper presents results from experiments devised to evaluate the comparative performance of a number of parallel algorithms we have devised for Suffix Arrays. This is an indexing strategy for very large text databases which allows complex queries to be effected in an efficient way. We have parallelized this strongly sequential strategy and obtained performance results with actual implementations operating upon a cluster of PCs and benchmark work-loads.

1 Introduction

In large text databases documents are represented by either their complete full text or extended abstracts. The user expresses his/her information need via words, phrases or patterns to be matched for and the information system retrieves those documents containing the user specified strings [3].

To reduce the cost of searching a full text, specialized indexing structures are adopted. The most popular of these are *inverted lists* [3, 1, 2]. Inverted lists are useful 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 take space which is 30% to 100% of the text size.

Suffix arrays or PAT *arrays* [3] are more sophisticated indexing structures which take space close to the text size. They are superior to inverted lists for searching phrases or complex queries such as regular expressions [3]. In addition, suffix arrays can be used to index texts other than occidental natural languages, which have clearly separated words that follow some convenient statistical rules [3]. Examples of these applications include computational biology (ADN or protein strings), music retrieval (MIDI or audio files), oriental languages (Chinese, Korean, and others), and other multimedia data files.

Suffix arrays are based on binary searching. Given a text collection, the suffix array contains pointers to the initial positions of all the retrievable strings, for example, all the word beginnings to retrieve words and phrases, or all the text characters to retrieve any substring. These pointers

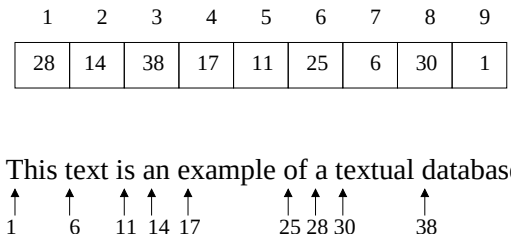

Figure 1: Suffix array.

identify both documents and positions within them. Each such pointer represents a *suffix*, which is the string from that position to the end of the text. The array is sorted in lexicographical order by suffixes as shown in Figure 1. Thus, for example, finding all positions for terms starting with "tex" leads to a binary search to obtain the positions pointed to by the array members 7 and 8 of Figure 1. This search is conducted by direct comparison of the suffixes pointed to by the array elements.

Let us assume that we are interested in determining the text positions in which a given substring x (of length T) is located in. This means that we want all the suffixes starting with x. In the sequential suffix array this can be solved by performing two queries; one with the immediate predecesor and the other with the immediate succesor. This takes $T \log N$ time for a text of N characters. We call this operation an *interval query*.

The efficient parallelization of Suffix Arrays is involved because its elements points to random text positions that cover the whole collection (we assume that document are uniformly distributed among the processors). Perhaps the obvious way to go is producing independent Suffix Arrays in each processor and broadcasting every query to all processors. Our experiments show that this naive approach is too demanding in communication. We have devised other forms of parallelization based on the global information that provides the whole text collection [6, 7]. Some of them seems to be counter intuitive but are devised to cope with the imbalance present in natural language text. Noticeably, they are equally efficient to those suitable for well-behaved text collections.

All the implementations were performed upon the bulk-synchronous model of parallel computing [9]. In particular we used the PUB BSP library and performed experiments upon a set of PCs connected via a communication switch.

In the following we describe the algorithms compared (section 2), present experiments (section 3) and conclusions (section 4).

2 Parallel algorithms

The algorithms employed are based on the BSP model of computing. In BSP, any parallel computer (e.g., PC cluster, shared or distributed memory multiprocessors) is seen as composed of a set of P processor-local-memory components which communicate with each other through messages. 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.

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, $h\,G$ and L, where w is the maximum of the computations performed by each processor, 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 barrier synchronising 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 processors speed s (e.g. mflops) can be empirically determined for each parallel computer by executing benchmark programs at installation [8].

Notice that the requirement of periodically barrier synchronizing the processors can be relaxed in situations in which a given processor knows beforehand the number of messages it should expect from all others. In this case, a given processor just waits until it receives the proper number of messages to further continue its computations on local data. Barrier synchronization of sub-sets of processors is also possible [9, 10].

We assume a server operating upon a set of P machines, each containing its own memory. Clients request service to one or more *broker* machines, which in turn distribute them evenly onto the P machines implementing the server. Requests are queries that must be solved with the data stored on the P machines. We assume that under a situation of heavy traffic the server processes batches of $Q = q\,P$ queries. Processing each batch can be considered as a hyperstep composed of one or more BSP supersteps. The value of q should be large enough to properly amortize the communication and synchronization costs of the particular BSP machine.

Observe that hypersteps can be pipelined so that at any superstep we can have one or more cycles at different stages of execution. For the algorithms presented below we assume that in each superstep a new batch starts execution and its computations are performed together with those associated with the solution to previous batches. Typically processing a batch will require two supersteps, thus on average every superstep deals with queries from two different batches.

The text collection of size N is supposed to be evenly distributed onto the P processors. The parallel algorithms are constructed from various ways to distribute the sequential Suffix Array upon the processors as described in the following,

Seq: The sequential Suffix Array efficiently implemented exclusing all code related to the parallel implemantations.

Local: An independent N/P-sized Suffic Array (SF) is constructed in every processor by considering the local text only.

Lexico: A SF array is constructed by considering the whole text collection. The this *global* array is partitioned in P pieces of size N/P. The array elements from 1 to N/P are stored in the first processor, the following N/P elements in the second one and so on. Thus each processor holds an interval of the lexicographic order given by the global array.

VLexico: Similar to Lexico but $V = 2^k\,P$ processors are considered instead of P. The N/V pieces are distributed circularly among the P actual processors. This is intended to break apart

large intervals localted in the same processors. In this way we avoid imbalance coming from natural langage text. We have found $k = 2$ a good tradeoff among space and speed for our particular hardware platform.

In our realization of Lexico and VLexico we keep in each processor an array of P (V) strings of size L marking the delimiting points of each interval of G0 (G1). The broker machine routes queries uniformly at random to the P processors, and in every processor a $\log P$ ($\log V$) binary search is performed to determine to which processor to send a given query (we do so to avoid the broker becoming a bottleneck).

Multiplex: In this case the N-sized global array is multiplexed among the P processors. We mean, element 1 goes to processor 1, element 2 goes to processor 2, ..., element P goes to processor P, then circularly element $P + 1$ goes to processor 1, $P + 2$ to processor 2, and so on. Thus between elements 1 and 2 of the N/P-sized piece of array stored in any given processor we have $P - 1$ elements of the global array stored in the other $P - 1$ processors. Those elements are stored in lexicographic order so that a binary search among processors can be easily performed. Queries are distributed uniformly at random among the processors. For every query a binary search is conducted in its respective processors to then continue (if necessary) with an inter-processors binary search.

A binary search on the lexico, vlexico and multiplex approaches can lead to a certain number of accesses to remote memory. Array elements can point to any text position located at any processor. Thus with high probability it is necessary to retrieve remote text in every step of the binary search. A very effective way to reduce the average number of remote memory accesses is to associate with every array entry the first t characters of the suffix pointed. This technique is called *pruned suffixes*. The value of t depends on the text and usual queries. In [5] it has been shown that this strategy is able to put below 5% the remote memory references for relatively modest t values.

Multi2: Similar to Multiplex but during the interprocessors search it does not send the full query string but the t first chars of it. This to be compared with the prunned suffixes and only when a match is found the remaining chars are sent to verify if it is a full match.

GLocal: Similar to the local strategy but for every array entry we store the processor and remote array entry where to continue the search if a given query is not found. First this search determines between what array entries is located the string searched for, and then it is continued (if necessary) in other processors as in the multiplex strategy. However, in this case we cannot perform a binary search among processors since only the next lexicographic position is stored in each array entry.

In [7] we propose a $P^{1/r}$ cost strategy to perform this interprocessors search. Here we consider the simplest case $r = 1$ since we employ a small number of processors in our experiments. Note that in this strategy it is not necessary to get remote text during a search in a given processor since all array entries points to local text. However, for the same memory space consumed by the pointers needed to conduct the inter-processor searches the others can maintain a significant number t of characters in the prunned suffixes.

3 Experiments

We use 1GB sample text from the Chilean Web search engine `www.todocl.cl`, treated as a single string of characters. We also employed synthetic text specially devised to be a very demanding benchmark for this application of parallel computing. With such large text database all speed-ups were superlinear due to disk activity in our limited hardware platform. This was not a surprise since for this text size we had to keep large pieces of the suffix array in secondary memory whilst communication among machines was composed of a comparatively much smaller number of small strings.

Thus we better resorted to experiments with a reduced text database. We used several samples of a few MBs per processor. Note that this reduced very significantly the computation costs and thereby it made much more relevant the communication and synchronization costs in the overall running time. That is, this imposed a demanding case for the parallel algorithms with respect to the sequential one.

Queries were formed in three ways: (1) by selecting at random initial word positions within the text and extracting substrings of length 16; (2) similarly but starting at words that start with the four most popular letters of the Spanish language, "c", "m", "a" and "p" ; (3) taken from the query log of `www.todocl.cl`, which registers a few hundred thousand user queries submitted to the web site. In set (1) we expect optimal balance, while in (2) and (3) we expect large imbalance as searches tend to end up in a subset of the global array. We observed that the natural language work-load (3) produced results between (1) and (2), actually quite closer to (2) than (1), thus we only show results for these two extreme cases.

The results were obtained on a PC cluster of 8 machines (PIII 700, 128MB) contected by a 100MB/s communication switch. The whole text was kept on disk so that, for the prunned suffixes strategies, once the first t chars of a query were found to be equal to the t chars kept in the respective array element, a disk access was necessary to verify that the string forming the query was effectively found at that position. This frequently required an access to a disk file located in other processor, in which case the whole query was sent to that processor to be compared with the text retrieved from the remote disk.

The results for 4 processors are shown in figure 2. In 2.1 and 2.2 a first fact to note is that de Local strategy takes even more running time than the sequential one. Also it is shown that GLocal achieves the best overall performance. For the case of queries biased to the 4 most popular letters the strategies based on prunned suffixes depart from GLocal. However, in this case those strategies are operating at a very slow t value (size of the prunned suffixes). The figures 2.3 and 2.4 show that now those strategies outperform GLocal when t is increased to a value similar to that occupied by GLocal. Among them, Multiplex achieves the best performance, specially for the unbalanced case. Note that their performance remains fairly the same between the balanced and unbalanced cases. This is because the t value is large enough to cope with the bias produced by the queries (usually those queries involved many repeated chars at the beginning of the query strings for the biased synthetic work-load).

Figure 3 shows results for 8 processors (note that here we process more queries in total which explains the larger running times observed). Now the conclusions for 4 processors are more evidente since with more processors GLocal must perform a larger interprocessors search. For the small t

Multi2 achieves the best performance. For large t Multiplex performs better. Noticeably, VLexico with $k = 2$ and large t achieves competitive performance. This strategy is actually an intermedia point between Lexico and Multiplex. As expected in all cases Lexico is impacted by biased queries and this is more evident for 8 processors (the same was observed for the natural language workload).

4 Conclusions

We have presented empirical results that show the comparative performance among a number of alternative parallel realizations of Suffix Arrays. Among them, the so-called Multiplex appears to be the best alternative for biased workloads like the ones arising in natural language text dabatases. VLexico is also a good alternative with the advantage of a simpler implementation. The simplest and more intuitive of all is the so-called Local strategy. However, our results show that this is worst in terms of running time. Only in a parallel computer with very efficient communication hardware this alternative might be considered a candidate.

Importantly, our results are consistent with the running time expressions that can be obtained for the different strategies by using the BSP cost. We obtained such expressions in [7] and they are quite predictive of the empirical facts presented in this paper. This show the suitability of the BSP model both as a practical form of parallel computing and as a tool to effect the analysis of algorithms. In this following we present those expressions for the reader to compare with the results of this paper.

The goal is the processing of $Q = q\,P$ queries of T chars each, on a text of N chars using P BSP processors characterized by G (comm.) and L (sync).

$$
\begin{array}{lcl}
\mathit{Seq} & = & q\,P\,T\,\log(N) \\
\mathit{Local} & = & q\,P\,T\,\log(N/P) + q\,P^2\,G + L \\
\mathit{Lexico} & = & q\,T\,\log(N/P) + q\,T\,\log(N/P)\,G + L \\
\mathit{Multiplex} & = & q\,T\,\log(N) + q\,T\,\log(N)\,G + L \\
\mathit{GLocal} & = & q\,T\,(\log(N/P) + P^{1/r}) + q\,T\,P^{1/r}\,G + L
\end{array}
$$

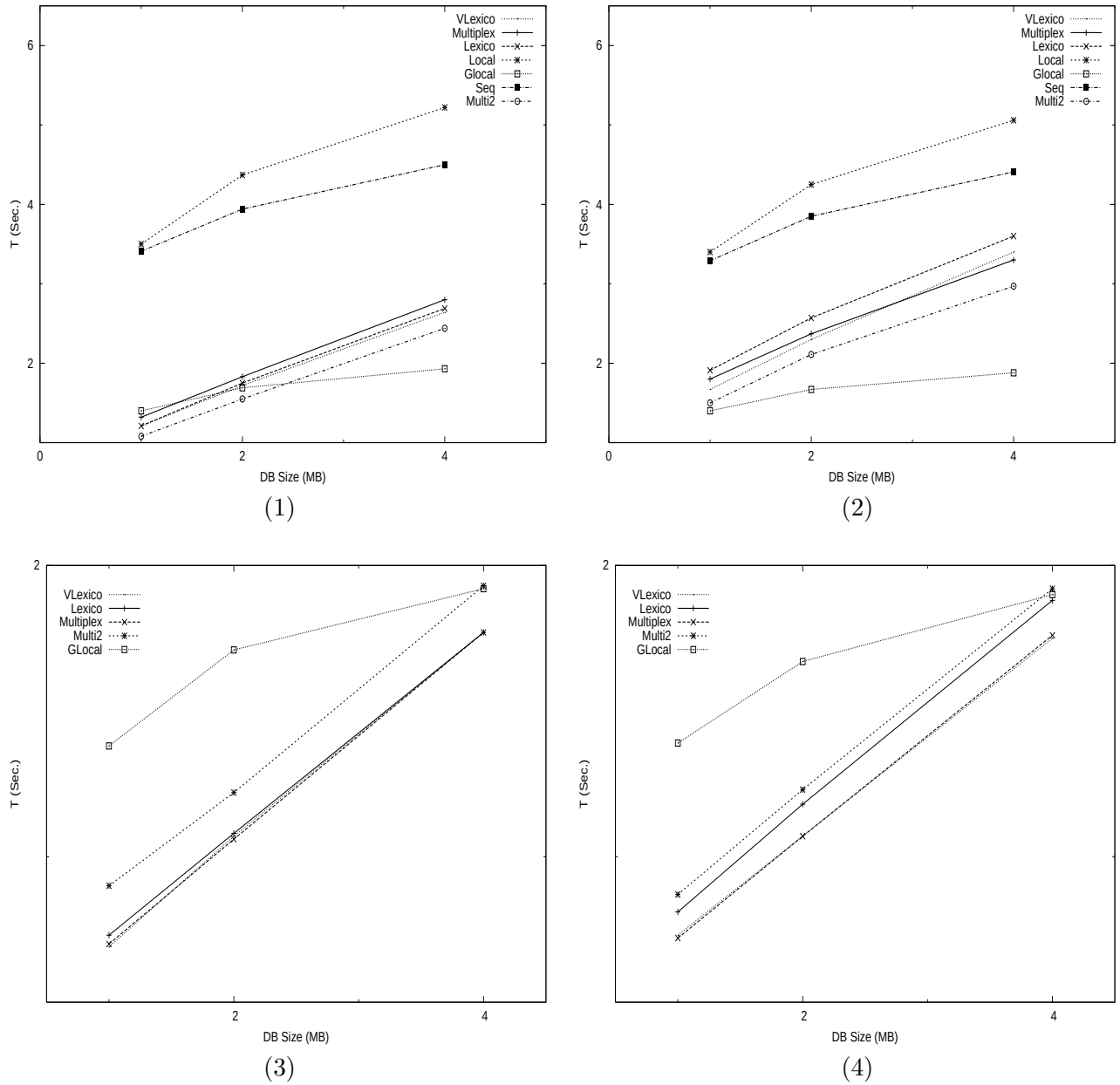

Figure 2: Search times for 4 processors. (1) balanced queries, (2) imbalanced queries. In cases (1) and (2) the prunned suffixes are very small. In cases (3) and (4) the prunned suffixes have been increased to be of the same size of the additional pointers used by GLocal.

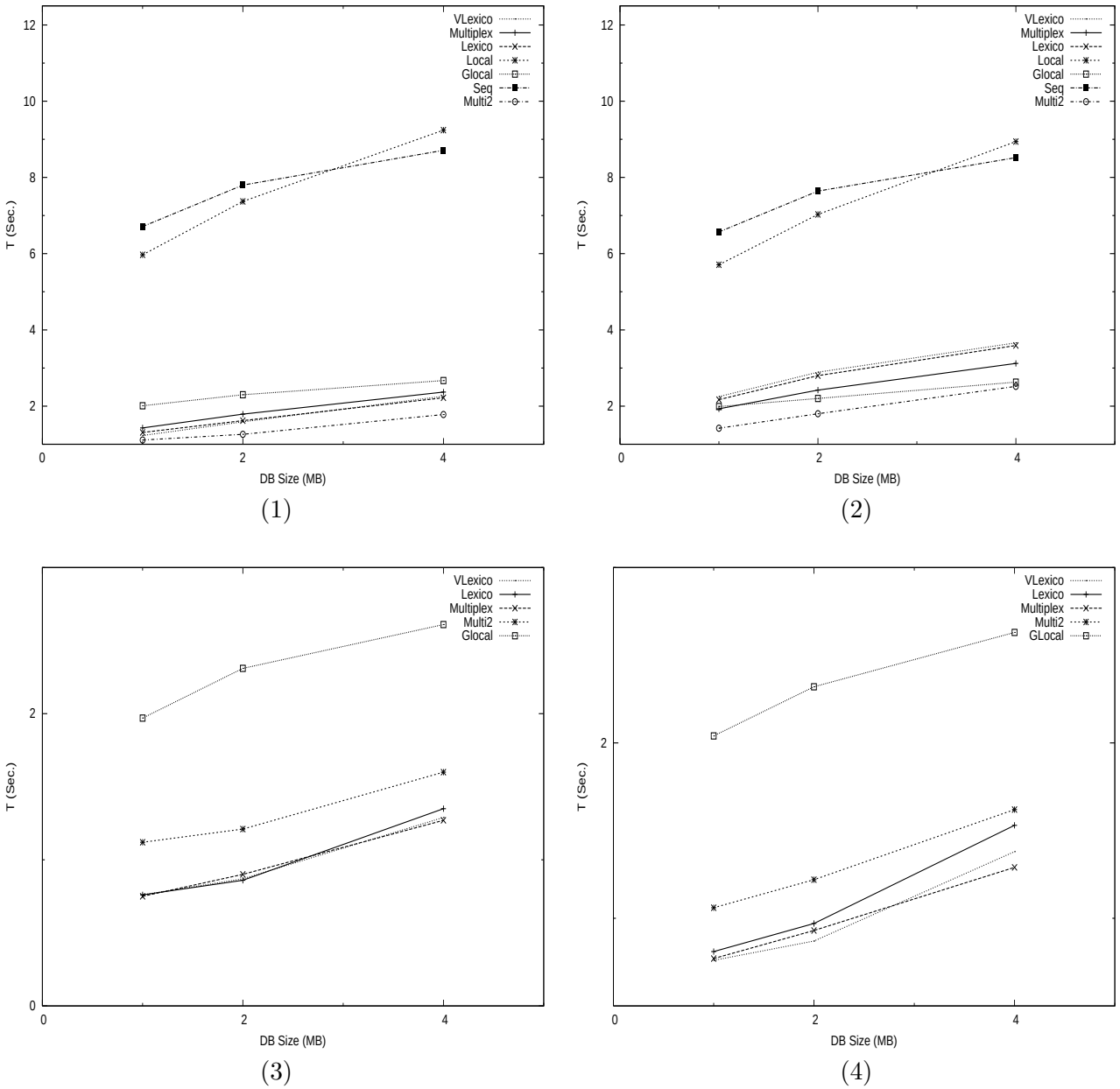

Figure 3: Search times for 8 processors. (1) balanced queries, (2) imbalanced queries. In cases (1) and (2) the prunned suffixes are very small. In cases (3) and (4) the prunned suffixes have been increased to be of the same size of the additional pointers used by GLocal.

References

[1] A. A. MacFarlane, J.A. McCann, and S.E. Robertson. Parallel search using partitioned inverted files. In *7th International Symposium on String Processing and Information Retrieval*, pages 209–220, 2000.

[2] C. Santos Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Concurrent query processing using distributed inverted files. In *8th International Symposium on String Processing and Information Retrieval*, pages 10–20, 2001.

[3] R. Baeza and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley., 1999.

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

[5] J. Kitajima and G. Navarro. A fast distributed suffix array generation algorithm. In *6th International Symposium on String Processing and Information Retrieval*, pages 97–104, 1999.

[6] M. Marin and G. Navarro. Suffix Arrays in Parallel. In *EuroPar 2003*, to appear, LNCS, 2003.

[7] M. Marin and G. Navarro. Distributed Query Processing using Suffix Arrays. In *SPIRE 2003*, to appear, LNCS, 2003.

[8] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. Technical Report PRG-TR-15-96, Computing Laboratory, Oxford University, 1996. Also in *Journal of Scientific Programming*, V.6 N.3, 1997.

[9] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.

[10] BSP PUB Library at Paderborn University, www.uni-paderborn.de/bsp.