Suffix Arrays in Parallel

Mauricio Marín Gonzalo Navarro
Center for Web Research
University of Chile
(www.cwr.cl)
mmarin@ona.fi.umag.cl

Abstract

Suffix arrays are powerful data structures for text indexing. Unlike the popular inverted files, suffix arrays do not depend on the existence of words in the text and are able of solving much more complex search problems. This makes them appealing in many scenarios, such as dealing with oriental languages and genomic and protein databases.

In this paper we present parallel algorithms devised to increase throughput of suffix arrays on a multiple-query setting. Design and cost evaluation is effected on top of the bulk-synchronous model of parallel computing, and thereby they are independent of programming details and architecture of the parallel machine. Experimental results show that efficient performance is indeed feasible in this strongly sequential and very poor locality data structure.

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 Web [3]. Typical applications are those known as client-server in which users take advantage of specialized services available at dedicated sites [4]. For the cases in which the number and type of services demanded by clients is such that it generates a very heavy work-load on the server, the efficiency of it in terms of running time is of paramount importance. As such it is not difficult to see that the only feasible way to overcome limitations of sequential computers is to resort to the use of several computers or processors working together to service the ever increasing demands of clients.

An approach to efficient parallelization is to split the data collection up and distribute them onto the processors in such a way that it becomes feasible to exploit locality by effecting parallel processing of user requests, each upon a subset of the data. As opposed to shared memory models,

this distributed memory model provides the benefit of better scalability [6]. However, this introduces new problems related to the communication and synchronization of processors and their load balance. This paper describes strategies to overcome these problems in the context of the parallelization of suffix arrays [3]. We propose strategies for reduction of inter-processors communication and load balancing.

The advent of powerful processors and cheap storage has allowed the consideration of alternative models for information retrieval other than the traditional one of a collection of documents indexed by keywords. One such a model which is gaining popularity is the *full text* model. In this model 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. While the cost of searching the full text is usually high, the model is powerful, requires no structure in the text, and is conceptually simple [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 also take space close to the text size. Their main drawback is their costly construction and maintenance procedures (i.e., creating and updating a suffix array). However, suffix arrays are superior to inverted lists for searching phrases or complex queries such as regular expressions [3].

The suffix array is a binary search based strategy. The array contains pointers to the document terms, where pointers identify both documents and positions of terms within them. The array is sorted in lexicographical order by terms 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 terms pointed to by the array elements. The meaning of "term" in this strategy can be different to that of the inverted lists [3]. Though it depends on the application, a "term" could describe a large portions of text such as a regular expression. A typical query is finding all text positions where a given substring appears in. For the purpose of the description of the algorithms presented in this paper we assume that this last one is the query of interest and that it is solved by performing two searches which locate the delimiting positions of the array for a given substring.

We assume a server site at which lots of queries are arriving per unit of time. Such work-load can be serviced by taking batches of Q queries each. On a 1-machine server the processing of each batch takes $Q \log N$ running time. Our aim is to improve this cost to the optimal $(Q \log N)/P$

1	2	3	4	5	6	7	8	9
28	14	38	17	11	25	6	30	1

This text is an example of a textual database

1 6 11 14 17 25 28 30 38

Figure 1: Suffix array.

by using a P-machines server. To achieve this goal a pragmatic (though naive) strategy would be to keep a copy of the whole database and index in each server's machine and route the queries uniformly at random among the P machines. That's fine as we are not expected to perform update operations on suffix arrays.

For very large databases, however, the non-cooperating machines are forced to keep large pieces of their identical suffix arrays in secondary memory which can degrade performance dramatically. A more sensible approach is then to keep a single copy of the suffix array distributed evenly onto the P main memories. Now the challenge is to achieve the optimal $(Q \log N)/P$ on a P-machines server which must perform communication and synchronization operations in order to service every batch of queries.

An important fact to consider in natural language texts is that words are not uniformly distributed both in the text itself and the queries provided by the users of the system. For example, in the Chilean web (www.todocl.cl) words starting with letters such as "c", "m", "a" and "p" are the most frequent ones. This fact can lead to significant imbalance in terms of parallel processing of queries.

The efficient parallel index construction has been investigated in [7, 5]. In this paper we focus on query processing. We propose efficient parallel algorithms for (1) processing queries grouped in batches, and (2) load balancing properly this process when dealing with biased collections of terms such as in natural language.

2 Preliminaries

In the bulk-synchronous parallel (BSP) model of computing [10, 9], 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 organised 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 synchronisation 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 time [9].

As an example of a basic BSP algorithm let us consider a broadcast operation which will be used in the algorithms described in the following subsection. Suppose a processor wants to send a copy of P chapters of a book, each of size a, to all other P processors (itself included). A naive approach would be to send the P chapters to all processors in one superstep. That is, in superstep 1, the sending processor sends P chapters to P processors at a cost of $O(P^2\,(a+a\,G)+L)$ 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 $O(P\,(a+a\,G)+L)$ units. In superstep 2, each processor sends its arriving chapter to all others at a cost of $O(P\,(a+a\,G)+L)$ units. Thus at superstep 2, all processors have a copy of the whole book. That is, the broadcast of a large P-pieces a-sized message can be effected at $O(P\,(a+a\,G)+L)$ cost.

We assume a server operating upon a set of P machines, each containing its own main and secondary memory. We treat secondary memory like the communication network. That is, we include an additional parameter D to represent the average cost of accessing the secondary memory. Its value can be easily determined by benchmark programs available on Unix systems. The textual database is evenly distributed over the P machines. If the whole database index is expected to fit on the P sized main memory, we just assume $D = 1$.

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. It is not difficult to see that the this cycle of three supersteps can actually be reduced to a cycle of one superstep by dealing with three separate batches, each in a different stage of execution.

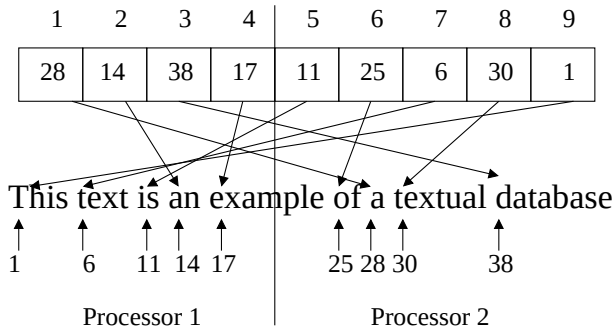

Figure 2: A global index suffix array distributed on two processors.

2.1 Global vs local suffix arrays

Let us assume that we are interested in determining the text positions in which a given substring is located in. In the sequential suffix array this can be solved by performing two queries; one with the inmediate predecesor and the other with the inmediate succesor. Let us call this operation *interval query*.

A suffix array can be distributed onto the processors using a global index approach in which a single array is built from the whole text collection and mapped evenly on the processors. A realization of this idea for the example in figure 1 is shown in figure 2 for 2 processors. Notice that in this global index approach each processor stands for an interval or range of suffixes (for example, in figure 2 processor 1 represents suffixes with first letters from "a" to "e"). The broker machine mantains information of the values limiting the intervals in each machine and route queries to the processors accordingly. This fact can be the source of load imbalance in the processors when queries tend to be dinamically biased to particular intervals (in the next section we propose a solution to this problem).

Let us assume the ideal scenario in which the queries are ruoted uniformly at random onto the processors. A search for all text positions associated with a batch of $Q = q\,P$ queries can be performed as follows. The broker takes $Q \log P + Q\,G + L$ time to route the queries to their respective target processors. Once the processors get their q queries each of them performs in parallel q binary searches. Note that for each query, with high probability $1 - 1/P$, it is necessary to get from a remote processor a T-sized piece of text in order to decide the result of the comparison and go to the next step in the search. This reading takes one additional superstep plus the involved cost of communicating T bytes per query. For a global array of size N, the binary search and the respective sending of the array positions are performed at cost $q\,\log(N/P) + q\,G + L + (q\,T\,G + L)\,\log(N/P)$. Then the q array positions per processor are received by the broker at cost $Q\,G$ to continue with the following batch and so on. However, it is not necessary to wait for a given batch to finish

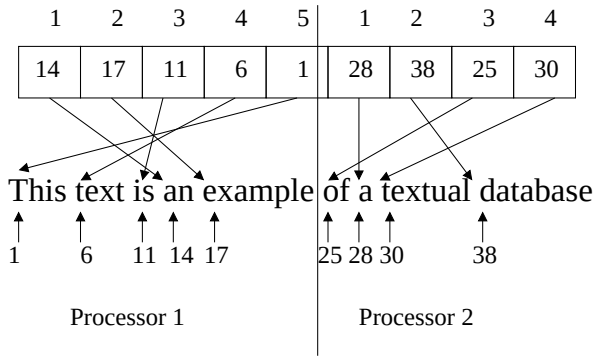

Figure 3: Local index suffix array.

since in each superstep we can start the processing of a new batch. This form a pipelining across supersteps in which at any given superstep we have, on average, $\log(N/P)$ batches at different stage of execution. The net effect is that at the end of every superstep we have the completion of a different batch. Thus the total (asymptotic) average cost per batch is given by

$$[q\,P\,\log P + q\,P\,G] + [q\,\log(N/P) + q\,T\,G + L]\,.$$

where the first term represents the cost of the operations effected by the broker machine whereas the second term is the (pipelined) cost of processing a Q-sized batch in the P-machines server.

In the local index strategy, on the other hand, a suffix array is constructed in each processor by considering only the subset of text stored in its respective processor. See figure 3. No references to text postitions stored in other processors are made. Thus it is not necessary to pay for the cost of sending T-sized pieces of text per each binary search step.

However, for every query it is necessary to search in all of the processors in order to find the pieces of local arrays that form the solution for a given interval query. As an answer, it suffices to send to the broker P pairs (a, b), one per processor, where a/b are the start/end positions respectively of the local arrays.

The processing of a batch of Q queries is as follows. Let us charge 1 unit to the handling of each query by the broker so it first does a work proportional to $q\,P$. Unfortunately, the broker now has to send every query to every processor. This broadcast operation can be effected as described in section 2. That is, the processors get q queries from the broker and then broadcast them to all other processors at a total cost of $q\,P + q\,P\,G + 2L$. In the next superstep, each processor performs in parallel $q\,P$ binary searches and sends q pairs (a, b) to the broker at a total cost of $q\,P\,\log(N/P) + q\,G + L$. The broker, in turn, receives $q\,P$ queries at a cost of $q\,P\,G$ units of time.

Thus the total cost of this strategy is given by

$$[q\,P + q\,P\,G] + [q\,P\,\log(N/P) + q\,G + L]\,.$$

Thus we see that the global index approach offer the potential of better performance in asymptotic terms. It is worthwhile then to focus on how to improve some performance drawbacks of the global index strategy. In the proposal we describe below we get rid of the $\log P$ factor in the broker machine which comes as a product of improving load balance in the P-machines server, and reduce significantly the amount of communication ($T\,G$) performed by the server.

3 Global Suffix Arrays in parallel

The first drawback of the global index approach is related to the possibility of load imbalance coming from large and sustained sequences of queries being routed to the same processor. The best way to avoid particular preferences for a given processor is to send queries uniformly at random among the processors. We propose to achieve this effect by multiplexing each interval defined by the original global array so that if the array element i is stored in processor p, then the elements $i+1$, $i+2$, ... are stored stored in processors $p+1$, $p+2$, ... respectively and in a circular manner as shown in figure 4. We call this strategy G2.

In this case, any binary search can start at any processor. Once a search has determined that the given term must be located between two consecutive entries k and $k+1$ of the array in a processor, the search is continued in the next processor and so on, where at each processor it is only necessary to look at the entry k of the array. For example, in figure 4 a term located in the first interval, may be located either in processor 1 or 2. If it happens that a search for a term located at position 6 of the array starts in processor 1, then once it determines that the term is between positions 5 and 7, the search is continued in processor 2 by directly examining the position 6. In general, for large P, the inter-processors search can be done in at most $\log P$ additional supersteps. This by performing a binary search accross processors. Indeed, the calculations involved in the determination of the next processor and array possition (circularly) during every search step are trivial. This simplifies the algorithm implementation.

As shown in figure 2 a binary search on the global index approach can lead to a certain number of accesses to remote memory. In BSP, one of these accesses must be done using an additional superstep; in superstep i a processor p sends a message to another processor, it receives the message in superstep $i+1$, reads the term, composes and sends to p a message containing it. In superstep $i+2$ the processor p gets the term and performs the comparison which allows it to continue the binary search. A cache scheme can be implemented in order to keep in p the most frequently referenced terms from remote memory. We have realized that a very effective way to reduce the average

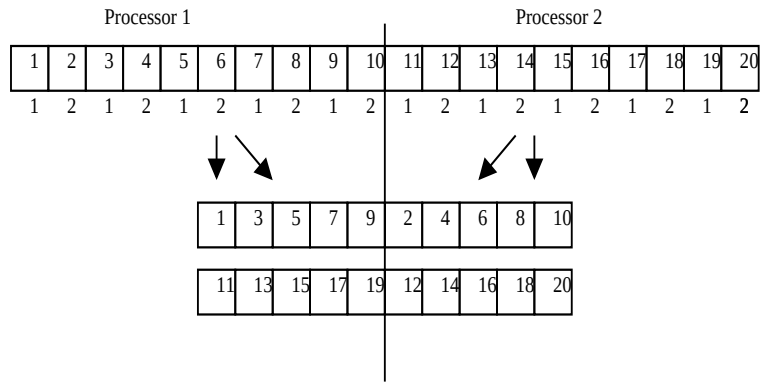

Figure 4: Multiplexing the global index suffix array entries.

number of remote memory accesses is to associate with every array entry the first t characters of the terms respectively. This value t depends on the average length of terms. In [5] it has been shown that this strategy is able to put below 5% the remote memory references for relatively modest t values. Our experiments show rates below 1%.

Note that the multiplexed strategy (G2) can be seen as the opposite extreme of the global index distributed lexicographically starting from processor 0 to $P-1$, wherein each processor holds a certain interval of the suffixes pointed to by the N/P array elements. Let us call this last one G0. The delimiting points of each interval of the G0 strategy can be kept in an array of size $P-1$ so that a binary search conducted on it can determine to which processor to route a given query.

An intermediate strategy (G1) between G0 y G2 can be obtained by considering the global array as distributed on $V = 2^k P$ virtual processors with $k > 0$ and that each of the V virtual processors is mapped circularly on the P real processors using the usual $i \bmod P$ for $i = 0...V$ with i being the i-est virtual processor. In this case, each real processor ends up with V/P different intervals of N/V elements of the global array which tends to break apart the imbalance introduced by biased queries. Calculations of the array positions are trivial.

In our realization of G0 and G1 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 real processors, and in every processor a $\log P$ ($\log V$) binary search is performed to determine to which processor send a given query (we do so to avoid the broker becoming a bottleneck). Once a query has been sent to its target processor it cannot migrate to other processors as in the case of G2. That is, this strategy avoids the inter-processors $\log P$ binary search. In particular, G1 avoids this search for a modest k whilst it well approaches the load balance achieved by G2 as we show in the next section. The extra-space should not be a burden as $N \gg P$ and k is expected to be small.

4 Experimental results

We compared the multiplexed strategy (G2) with the plain global suffix array (G0), and the intermediate strategy (G1). For each element of the array we kept C characters which are the C-sized prefix of the suffix pointed to by the array element. We found $C = 4$ to be a good value for our text collection. In G2 the inter-processors binary search is conducted by sending messages with the first C characters of the query.

The text collection is formed by a 1GB sample of the Chilean Web retrieved by the search engine www.todocl.cl. We treated it as a single string of characters, and queries were formed by selecting positions at random within this string. For each position a substring of size 16 is used as a query. In the first set of experiments these positions were selected uniformly at random. Thus load balance is expected to be near optimal. In the second set of experiments we selected at random only the positions whose starting word character were one of the four most popular letters of the Spanish language. This produces large imbalance as searches tend to end up in a subset of the global array.

The results were obtained on a PC cluster of 16 machines (PIII 700, 128MB) contected by a 100MB/s communication switch. Experiments with more than 16 processors were performed by simulating virtual processors. In this small cluster most speed-ups obtained against a sequential realization of suffix arrays were super-linear. This was not a surprise since due to hardware limitations we had to keep large pieces of the suffix array in secondary memory whilst communication among machines was composed by a comparatively small number of small strings. The whole text was kept on disk so that once the first C chars of a query were found to be equal to the C 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. With high probability this required an access to a disk file located in other processor, case in which the whole query is sent to that processor to be compared with the text retrieved from the remote disk.

Though we present running time comparisons below, what we considered more relevant to the scope of this paper is an implementation and hardware independent comparison among G0, G1 and G2. This came in the form of two performance metrics devised to evaluate load balance in computation and communication. They are the average maximum across supersteps. During the processing of a query each strategy performs the same kind of operations, so for the case of computation the number of these ones executed in each processor per superstep suffices as an indicator of load balance for computation. For communication we measured the amount of data sent to and received from at each processor in every superstep. We also measured balance of disk accesses. All runs were to the same number of supersteps and in all cases a very similar number of queries were completed. In each case 5 runs with different seeds were performed and averaged. At each superstep we introduced $1024/P$ new queries in each processor.

P	comp.	comm.	disk
2	0.95	0.90	0.89
4	0.49	0.61	0.69
8	0.43	0.45	0.53
16	0.39	0.35	0.36
32	0.38	0.29	0.24
64	0.35	0.27	0.17

Table 1: Ratio G2/G0.

P	comp.	comm.	disk
2	1.10	0.90	0.89
4	0.92	0.82	0.69
8	0.86	0.65	0.53
16	0.80	0.55	0.36
32	0.78	0.45	0.24
64	0.75	0.43	0.17

Table 2: Ratio G2/G1 witk $k = 4$.

In table 1 we show results for queries biased to the 4 popular letters. Columns 2, 3, and 4 show the ratio G2/G0 for each of the above defined performance metrics (average maximum for computation, communication and disk access). The results for G2/G1 are shown in table 2. These results confirm intuition, that is G0 can degenerate into a very poor performance strategy whereas G2 and G1 are a much better alternative. Noticeably G1 can achieve similar performance to G2 at a small $k = 4$. This value depends on the application, in particular on the type of queries generated by the users. G2 is independent of the application but, though well-balanced, it tends to generate more message traffic due to the inter-processors binary searches (specially for large C). The differences among G2, G1, G0 are not significant for the case of queries selected uniformly at random. G2 tends to have a slightly better load balance.

As speed-ups were superlinear due to disk activity, we performed experiments with a reduced text database. We used a sample of 1MB per processor which reduces very significantly the computation costs and thereby it makes much more relevant the communication and synchronization costs in the overall running time. We observed an average efficiency (speed-up divided by the number of processors) of 0.65.

In the table 3 we show running time ratios obtained with our 16 machines cluster. The first part of the table shows results for the biased query terms (large imbalance) and the second one

P	G2/G0	G2/G1
4	0.68	0.87
8	0.55	0.66
16	0.61	0.67
4	0.78	0.77
8	0.78	0.73
16	0.86	0.83

Table 3: Running times ratios

shows results for terms selected uniformly at random ("well-behaved" work-load). In all cases, the difference between running times of the two work-loads, for the same strategy, were almost twice. The biased workload increased running times by a factor of 1.7 approximately.

The results of the table 3 show that the G2 strategy outperformed the other two strategies, though G1 has competitive performance. Notice, however, that G2 losses efficiency as the number of processors increases. This is because, as P grows up, the effect of performing inter-processors binary searches becomes more significant in this very low-cost computation scenario.

5 Final comments

We have compared different realizations of the Suffix Array we have devised to perform query processing in parallel. In general, the implementation of the algorithms for G0 and G1 were simpler than the ones for G2. For texts and queries which are not highly biased we suggest using G1 with $k = 4$ as it is a simple strategy which achieves a reasonable load balance.

Yet another method which solves both load imbalance and remote references is to re-order the original global array so that every element of it contains only pointers to local text (or most of them). This is shown in figure 5. This becomes similar to the local index strategy whilst it still keep global information which avoids the P parallel binary searches and broadcast per query. Unfortunaly we now lose the capability of performing the inter-processors $\log P$-cost binary search. We are currently investigating ways of performing this search efficiently but at reduced extra-space requirements. For example, a reference to a single array element consumes 4 bytes, that is the space of 4 chars. To reduce the number of steps of the inter-processors search we would need to store one or more remote references for each array element.

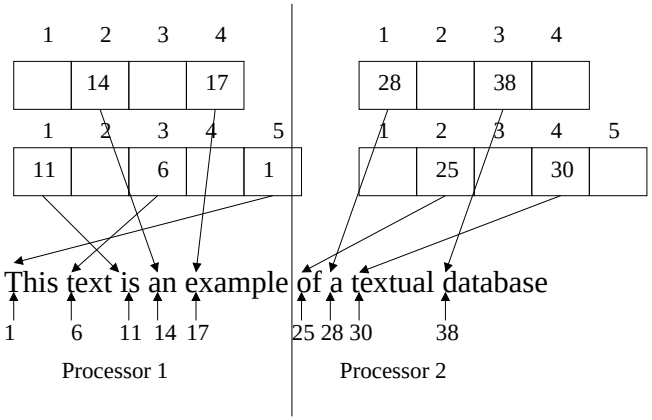

Figure 5: Combining multiplexing with local only references.

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] W.F. McColl. General purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, pages 337–391. Cambridge University Press, 1993.

[7] G. Navarro, J. Kitajima, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays. In *8th Annual Symposium on Combinatorial Pattern Matching*, pages 102–115, 1997. LNCS 1264.

[8] B. Ribeiro, J. Kitajima, G. Navarro, C. Santana, and N. Ziviani. Parallel generation of inverted lists for distributed text collections. In *XVIII Conference of the Chilean Computer Science Society*, pages 149–157, 1998.

[9] 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.

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