Distributed Query Processing using Suffix Arrays

Mauricio Marín Gonzalo Navarro

mmarin@ona.fi.umag.cl gnavarro@dcc.uchile.cl
Dept. of Computer Science Dept. of Computer Science
University of Magallanes University of Chile
Center for Web Research (www.cwr.cl) *

Abstract. Suffix arrays are more efficient than inverted files for solving complex queries in a number of applications related to text databases. Examples arise when dealing with biological or musical data or with texts written in oriental languages, and when searching for phrases, approximate patterns and, in general, regular expressions involving separators. In this paper we propose algorithms for processing in parallel batches of queries upon distributed text databases. We present efficient alternatives for speeding up query processing using distributed realizations of suffix arrays. Empirical results obtained from natural language text on a cluster of PCs show that the proposed algorithms are efficient in practice.

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 server efficiency 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 up the data collection and distribute it 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 [7]. However, it 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.

* Funded by Millennium Nucleus CWR, Grant P01-029-F, Mideplan, Chile.

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]. *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.

The suffix array uses a binary search based strategy. Processing a single T-chars-size query in a text of size N takes $O(T \log N)$ time on the standard sequential suffix array. Thus trying to reduce such time by using a P-processors distributed memory parallel computer is not very attractive in practical terms.

In this paper 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. Processing batches in parallel is appealing in this context as one is more interested on improving the throughput of the whole process than single operations. To achieve this goal a pragmatic (though naive) strategy would be to keep a copy of the whole text database and index in each server machine and route the queries uniformly at random among the P machines. This can be acceptable.

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 efficient performance on a P-machines server that must communicate and synchronize in order to service every batch of queries. This is not trivial because most array positions are expected to point to text located in remote memory when naive partitioning is employed.

An important fact to consider in natural language texts is that words are not uniformly distributed, both in the text itself and in 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 the parallel processing of queries.

The efficient index construction using parallel computing techniques has been investigated in [8, 6]. The aim was the construction of a global suffix array for

the entire text collection so that queries upon that index can be performed using the standard sequential binary search algorithm. However, the problem of going further on by properly distributing the suffix array on a set of processors to efficiently support parallel processing of batches of queries has not been investigated so far. Note that in [5] a related parallel algorithm was proposed which works upon a distributed Patricia like tree that is constructed upon the suffix array. No implementation was proposed and tested. We perform parallel searching directly on the distributed suffix array with no additional data structure upon it.

In this paper we focus on such form of query processing. We propose efficient parallel algorithms for (1) processing queries grouped in batches on distributed realizations of suffix arrays, and (2) properly load balancing this process when dealing with biased collections of terms such as in natural language texts. In each case our aim is to reduce the communication and synchronization requirements. We explore alternative ways of solving those problems and our empirical results show that the proposed algorithms are efficient in practice.

A valuable feature of the algorithms we propose is that they are devised upon the bulk-synchronous model of parallel computing (BSP model) [10, 12]. This is a distributed memory model with a well-defined structure that enables the prediction of running time. We use this last feature to compare different alternatives for index partitioning by considering their respective effects in communication and synchronization of processors. The model of 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 are programmed in the same way. They are considered emulators of the more general bulk-synchronous parallel machine.

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 by ways of libraries such as BSPlib [13] or BSPub [14]. 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 and MPI 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 [10, 11].

2 Suffix Arrays

Suffix arrays or PAT *arrays* [3] are data structures for full text retrieval 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 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.

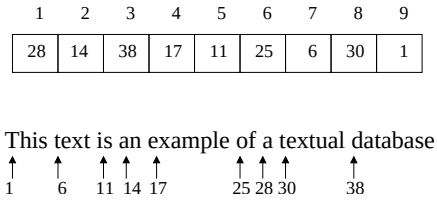

Fig. 1. Suffix array.

3 BSP and The Cost Model

In the bulk-synchronous parallel (BSP) model of computing [12, 10], 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 [10].

As an example of a basic BSP algorithm, let us consider a broadcast operation that will be used in this paper. 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. Hence 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 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.

4 Global versus Local Suffix Arrays

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. 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 a lexicographical 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 dynamically biased to particular intervals.

Let us assume the ideal scenario in which the queries are routed 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 $Q\,T \log P + Q\,T\,G + L$ time to route the queries to their respective target processors (note that this cost can be actually reduced to $Q\,T\,G + L$ by routing uniformly at random the queries as we propose below). Once the processors get their q queries, in parallel each of them performs 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

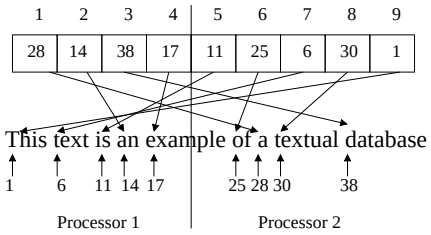

Fig. 2. A global index suffix array distributed on two processors.

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 $q\,T\,\log(N/P) + (q\,T\,G + L)\,\log(N/P)$. 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 since in each superstep we can start the processing of a new batch. This forms a pipelining across supersteps in which at any given superstep we have, on average, $\log(N/P)$ batches at different stages 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) cost per batch is given by

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

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. We call this strategy G0.

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 string, composes and sends to p a message containing it. In superstep $i+2$ the processor p gets the string and performs the comparison that allows continuing the binary search. A cache scheme can be implemented in order to keep in p the most frequently referenced strings from remote memory (i.e., those close to the root of the global binary search virtual tree). 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 [6] 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%.

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.

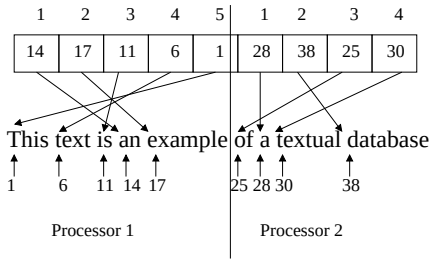

Fig. 3. Local index suffix array.

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 answers for interval queries, it is necessary to send to the broker $Q\,P$ pairs (a, b), Q 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 $Q = q\,P$. Unfortunately, the broker now has to send every query to every processor. This broadcast operation can be effected as described in Section 3. That is, the processors get q queries from the broker and then broadcast them to all other processors at a total cost of $Q + Q\,T\,G + L$. In the next superstep, each processor performs in parallel Q binary searches and sends Q pairs (a, b) to the broker at a total cost of $q\,P\,T\,\log(N/P) + q\,P^2\,G + L$. The broker, in turn, receives $Q\,P$ queries at a cost of $q\,P^2\,G$ units of time. Thus the total cost of this strategy is given by

$$[q\,P + q\,P^2\,G] + [q\,P\,T\,\log(N/P) + q\,P^2\,G + L]\,.$$

Thus we see that the global index approach offers 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 by improving load balance in the P-machines server, and we reduce significantly the amount of communication $(q\,T\,\log(N/P)\,G)$ performed by the server.

5 Global Multiplexed Suffix Array

One 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 array element i is stored in processor p, then elements

$i+1$, $i+2$, ... are stored in processors $p+1$, $p+2$, ... respectively, 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 entry k of its own 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 position 6.

In general, for large P, the inter-processors search can be done in at most $\log P$ additional supersteps by performing a binary search accross processors. This increases computation and communication in an additive $\log P$ factor leaving the pipelined BSP cost of this strategy (broker + server) in

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

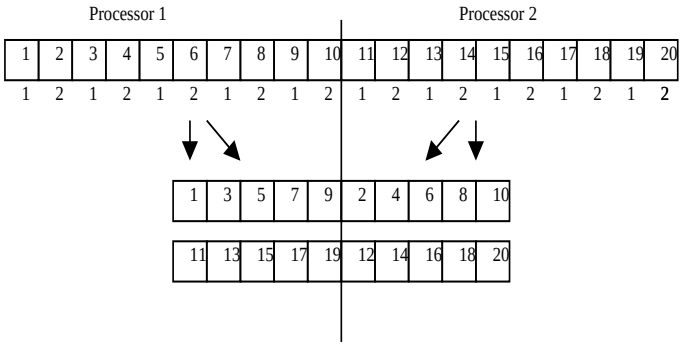

Fig. 4. Multiplexing the global index suffix array entries.

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 (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 and 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 i mod P for $i = 0...V$ with i being the i-th virtual processor. In this case, each real processor ends up with V/P different intervals of N/V elements of the global array. This tends to break apart the imbalance introduced by biased queries. Calculation 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 to 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 approaches well the load balance achieved by G2, as we show in the experiments. The extra space should not be a burden as $N \gg P$ and k is expected to be small.

6 Global Suffix Array with Local Text

Yet another method which solves both load imbalance and remote references is to redistribute the original global array so that every element of local arrays contain only pointers to local text, as shown in Figure 5. This becomes similar to the local index strategy whilst it still keeps global information that avoids the P parallel binary searches and broadcast per query. Unfortunately we now lose the capability of performing the inter-processors $\log P$-cost binary search, since the owners of the next global array positions are unknown. We propose an $O(r\, P^{1/r})$ cost strategy to perform this search when necessary, at the cost of storing r values per suffix array cell (instead of storing a pruned suffix of t chars per cell). We call this strategy G3.

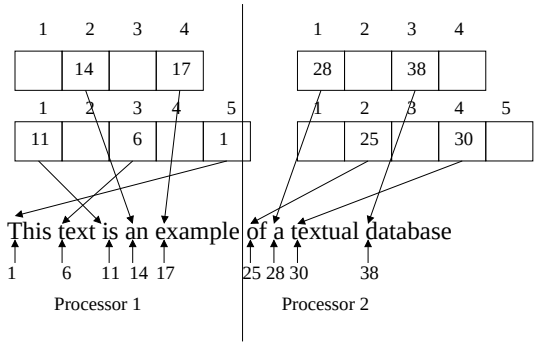

Fig. 5. Combining multiplexing with local-only references.

The method works for any $r \geq 1$, as follows. For $r = 1$, each cell stores the processor that owns the next cell of the global suffix array, plus the local address of that next cell inside the local suffix array of the processor owning it. Hence, given that a processor x finds the answer between its local consecutive cells i and $i + d$ (these are global addresses), it retrieves the processor y that owns cell

$i+1$, as well as the position of that cell in the local suffix array of processor y. Then x requests y to determine whether its text pointed by suffix array cell $i+1$ is lexicographically larger than the query. If it is, then i is the right answer. If it is not, then y is now in charge of finding the right position, by finding the processor z that owns cell $i+2$, and so on. This needs $O(d)$ supersteps because we advance cell by cell. On average $d = O(P)$ is the distance in the global suffix array between two cells that are consecutive in some local suffix array.

This can be improved for larger r as follows. The r values at cell i store the addresses of cells $i + P^{0/r}$, $i + P^{1/r}$, ..., $i + P^{(r-1)/r}$. Note that the first value is, as before, the address of cell $i + 1$. This value is essential to ensure the correctness of the algorithm, the others are just optional accelerators. Now, given that processor x finds that the answer is between cells i and $i + d$, which are consecutive in its local suffix array, it finds the largest j such that $P^{j/r} < d$. Then it finds processor y owning cell $i + P^{j/r}$. If y answers that the query is smaller than its cell, then processor x retains the problem, sets $d \leftarrow P^{j/r}$ and goes on with $j - 1$. Otherwise, processor y gets the problem, with $i \leftarrow i + P^{j/r}$ and $d \leftarrow d - P^{j/r}$. It will keep trying with the same j value.

Let us analyze the above algorithm on average, where $d = P$. We will start with $j = r - 1$. We can transfer the problem forward by $P^{(r-1)/r}$ cells at most $P/P^{(r-1)/r} = P^{1/r}$ times before the interval becomes too short for such a long skip value. At this point we set $j \leftarrow r - 2$ and the interval cannot be larger than $P^{(r-1)/r}$. By jumps of $P^{(r-2)/r}$ cells, we cannot make more than $P^{(r-1)/r}/P^{(r-2)/r} = P^{1/r}$ jumps before the interval becomes too small. This process continues until the interval is of size $P^{1/r}$ and we use the pointers to $i+1$ to finish the search. Overall, we perform $O(r\,P^{1/r})$ steps on average. This complexity is optimized for $r = \ln P$, where the average cost becomes $O(\log P)$, just as with the multiplexed strategy G2. In practice we may not have enough space to reach this optimum. The pipelined BSP cost of this strategy (broker + server) is given by

$$[q\,P + q\,P\,T\,G + L] + [q\,T\,(\log(N/P) + P^{1/r}) + q\,T\,P^{1/r}\,G + L]\,.$$

Strategy G3 is most useful in applications where the t-sized pruned suffixes are unable of significantly reducing the number of accesses to remote memory.

7 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 t characters which are the t-sized prefix of the suffix pointed to by the array element. We found $t = 4$ to be a good value for our text collection.

In G2 the inter-processors binary search is conducted by sending messages with the first t characters of the query. The complete query is sent only when it is necessary to decide the final outcome of the search or when the t characters are not enough to continue the search (this reduces the amount of communication during the inter-processors search).

We use 1GB sample text from the Chilean Web search engine `www.todocl.cl`, treated as a single string of characters. 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.

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 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.

Though we present running time comparisons below, what we considered more relevant for 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 average maxima 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. In all cases the same number of supersteps were performed and 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.

In Table 1(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 1(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 (especially for large t). 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.

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

(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

(2) G2/G1 witk $k = 4$.

P	G2/G0	G2/G1	G2/G3
4	0.68	0.87	0.41
8	0.55	0.66	0.36
16	0.61	0.67	0.31
4	0.78	0.77	0.58
8	0.78	0.73	0.45
16	0.86	0.83	0.46

(3) Running times ratios

Table 1. Comparison of search costs. The upper part of the table (3) shows results for the biased query terms (queries of type (2)) and the lower part for terms selected uniformly at random (queries of type (1)).

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 Table 1(3) we show running time ratios for our 16 machines cluster. The biased workload increased running times by a factor of 1.7 approximately.

The results of Table 1(3) show that the G2 strategy outperformed the other two strategies, though G1 has competitive performance for the imbalanced case (first part of the table). Notice, however, that for the work-load with good load balance (second part of the table) G2 tends to lose 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 and ideal load balance scenario (case in which G0 is expected to achieve its best performance). G3 showed worse performance. However, this and all others were at least 3 times faster than the local index strategy.

In our computational platform we observed that the cost of broadcasts and increased number of binary searches at each processor were significant and too detrimental for the local index strategy.

Let us now further illustrate the comparative performance of G0, G1 and G2 with respect to a sequential implementation of suffix arrays, all using the same workload. This is intended to show the practicality of our algorithms.

For our cluster machines, we explored the points at which page-faults reduced performance dramatically in the sequential strategy. We found $N = 8$MB to be a reasonable maximum. Thus we decided to execute experiments for $N = 1$, 2, 4 and 8 MB in the sequential algorithm. We also performed similar experiments on 4 processors but now keeping $N/P = 1$, 2, 4 and 8 MB in each processor.

This allowed comparing the effect of communication versus the effect of disk activity, since this sequential algorithm only maintains the suffix array in main

memory whereas the text database is kept in disk. For each step of the sequential binary search an access to disk must be performed in order to decide the comparison of suffixes. We retrieved t chars for t large enough so that remote memory accesses in the parallel algorithms were not significant. We tested the three types of queries, (1), (2) and (3). The results are illustrated in Figure 6.

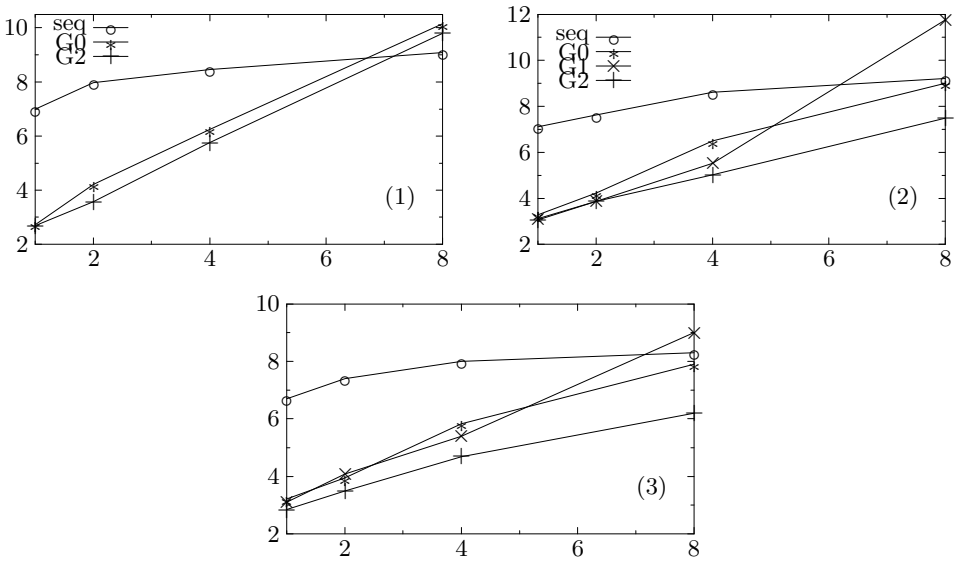

Fig. 6. Search times for (1) random balanced queries, (2) large imbalance, (3) real query log. The y-axis is running time (sec) and x-axis is DB size in MB.

The results show that the proposed algorithms are also useful in cases in which a single machine does not have enough main memory to keep in it both the text database and the index. In that case, it is more efficient to distribute the database and index on a set of machines.

8 Final Comments

We have presented a number of alternative realizations of distributed suffix arrays devised to support parallel processing of batches of queries as encountered in client-server applications. We have analyzed the algorithms by using actual implementations. Experiments were run on natural language texts.

In general, the implementation of the algorithms for G0 and G1 were simpler than that for G2. For texts and queries that are not highly biased we suggest using G1 with $k = 4$ as it is a simple strategy that achieves a reasonable load balance. Certainly the G2 strategy is the best one in cases of query patterns

generating large imbalance. Note that its performance is good enough even in well-behaved (balanced) query patterns.

Strategy G3 is competitive for cases in which the t chars maintained by G0, G1 and G2 in each array cell are not able to reduce significantly the number of references to remote memory. Our results show that G3 is much more efficient than the local index strategy because it avoids completely the parallel local searches across processors while it still keeps references to local text in its array cells.

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. P. Ferragina and F. Luccio. String search in coarse-grained parallel computers. *Algorithmica*, 24:177–194, 1999.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. D.B. Skillicorn and D. Talia, Models and languages for parallel computation, *ACM Computing Surveys* V.20 N.2 1998.
12. L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.
13. BSP World-wide Standard, www.bsp-worldwide.org.
14. BSP PUB Library at Paderborn University, www.uni-paderborn.de/bsp.