Invited Paper:

Index Structures for Distributed Text Databases

Mauricio Marin[*]
Departamento de Computación
Universidad de Magallanes
Casilla 113-D, Punta Arenas, Chile

Abstract

The Web has became an obiquitous resource for distributed computing making it relevant to investigate new ways of providing efficient access to services available at dedicated sites. Efficiency is an ever-increasing demand which can be only satisfied with the development of parallel algorithms which are efficient in practice.

This tutorial paper focuses on the design, analysis and implementation of parallel algorithms and data structures for widely-used text database applications on the Web. In particular we describe parallel algorithms for inverted files and suffix arrays structures that are suitable for implementing search engines. Algorithmic design is effected on top of the BSP model of parallel computing. This model ensures portability across diverse parallel architectures ranging from clusters to super-computers.

1 Introduction

In the last decade, the design of efficient *sequential* data structures and algorithms for text 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. 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, its 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.

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].

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

The bulk-synchronous parallel (BSP) model of computing [29, 35] has been proposed to enable the development of portable and cost-predictable software which achieves scalable performance across diverse parallel architectures. BSP is a distributed memory model with a well-defined structure that enables the prediction of running time. Unlike traditional models of parallel computing, the BSP model ensures portability at the very fundamental level by allowing algorithm design to be effected 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 as they are considered emulators of the more general bulk-synchronous parallel machine.

The practical model of BSP 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 [33] or BSPub [34]. We emphasize 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

[*]Partially supported by Millennium Nucleus CWR, Grant P01-029-F, Mideplan, Chile. Author's e-mail: `Mauricio.Marin@umag.cl`

PVM and MPI libraries. However, a number of studies have shown that BSP algorithms lead to more efficient performance than their message-passing or shared-memory counterparts in many applications [29, 30].

To reduce the cost of searching a full text, specialized indexing structures are adopted. The most popular of these are inverted lists [3]. 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.

On the other hand, *Suffix arrays* or PAT *arrays* [3] are more sophisticated indexing structures than inverted indexes, which also 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) and oriental languages. However, their efficient parallelization is more involved than inverted lists since they exhibit a very poor data locality during query processing.

This tutorial paper focuses on the design, analysis and implementation of parallel algorithms for these two index data structures. It surveys our work on BSP realizations of inverted lists and suffix arrays [16, 17, 18] and cites related work built upon traditional models of parallel computation such asynchronous message passing. In section 2 we present a description of the BSP model of parallel computing. Section 3 presents methods for inverted lists and section 4 presents methods for suffix arrays. Finally section 5 describes research topics.

2 Model of parallel computing

In the BSP model [29, 35], any parallel computer 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.

In most cases, optimal efficiency comes from solutions devised using pure BSP concepts rather than translations from algorithms devised for the traditional models of computing. The cost of a BSP algorithm is given by the sum of the cost of its supersteps where every superstep is costed taking the observed maxima in computation and communication. Proper load balancing is crucial and also algorithms should minimize the total number of supersteps.

More specifically, 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 cost 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 [29] or by determining asymptotic expressions in accordance with the topology of the communication network connecting the BSP processors [20].

As an example of a basic BSP algorithm let us consider a broadcast operation which will be implicitly used in the algorithms described in the following sections. 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 of running time. 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 3, 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.

The well-defined structure of BSP computations allows optimizations such as packing into a single large message a set of small messages sent by a processor to another processor. This amortizes overheads associated with the communication of many small messages addressed to the same processor. Also the requirement of periodically barrier synchronizing the processors can be relaxed in situations in which a given processor knows before hand 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 [34, 35].

3 Inverted Lists

Server-broker relationship

We assume a server operating upon a set of P identical 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 are solved by using an index data structure distributed on the P processors. We assume that the index is implemented using an inverted list which, as described in the next section, is composed of a vocabulary (set of terms) and a set of identifiers representing all the documents that contain at least one of the words that are members of the vocabulary. The inverted list data structure enables the efficient retrieval of all identifiers for which a given term appears in the respective documents.

We assume that under a situation of heavy traffic the server is able to process batches of $Q = q\,P$ queries. Every query is composed of one or more vocabulary terms for which it is necessary to retrieve all document identifiers associated with them. Only the identifiers of the most relevant documents are presented to the user, namely those which more closely match the user information need represented by the query terms. For this, it is necessary to perform a ranking of documents. A widely used strategy for this task is the so-called vector model [3], which provides a measure of how close is a given document to a certain user query. We assume that the reader is familiar with this method and overall terminology [3].

Minimal broker

In order to exploit the available parallelism we try to minimize the amount of sequential work performed by the broker machine. We restrict its functionality to receive user requests, distribute the queries onto the processors (uniformly at random), receive the best ranked documents (K in total) from the server, and pass them back to the user.

The two most basic operations related to providing answers to user queries are left to the parallel sever. That is, the retrieval of document identifiers and its respective ranking. Both operations are effected in parallel where the broker is responsible for scheduling those in a manner that keeps load balance of processors work as close to the optimal $1/P$ as possible. This is achieved by the combination of two strategies.

Firstly, the terms of the vocabulary are distributed uniformly at random onto the processors. This kind of strategy has proven to be a very effective tool for destroying correlation among the input data [35]; query terms in our case, with imbalance coming from the fact that user preferences could cause that many terms be routed to the same processor. We use a hashing function on the term's characters for this purpose. This function is used to distribute the vocabulary's terms at index construction time and during the broker's term distribution process.

Secondly, for every query the broker chooses a server processor in which performing the ranking of documents. Later this processor sends the K best ranked ones back to the broker. As the terms of a given query (or set of queries), are likely to be located in different processors, the broker chooses such processor in a way that tends to maintain a good load balance with all other processors. That is, this processor is chosen in a way that attempts to evenly distribute all ranking tasks onto the P processors. This is effected by maintaining a counter for each processor. These counters keep the number of ranking tasks scheduled in every processor for queries which have not been completed yet. A ranking task for a given query is scheduled on one of the processors that are associated with their respective terms. From those, the processor with the smallest counter value is the selected one.

Thus the query terms are distributed uniformly at random by ways of the hashing function. The targets processors for these terms perform the work required to retrieve their associated lists of document identifiers. Then the lists associated with every query are routed to the processor selected for their final ranking. Lists associated with different queries are expected to be routed to different processors.

A pseudo-code for the steps executed by the broker is the following,

```
while(true)
{
   msg= rcvMessage();
   switch( msg.from() )
   {
     case USER: // new query arrival.
         // select final ranking of docs.
         proc= rankingProcessor(msg.Query());

         foreach term in msg.QueryTerms()
         {
            term.ranker(proc);
            serverProc= hashing(term.str());
            sndMsg(serverProc,term);
         }
        break;
        case SERVER: // server answer arrival.

            updLoadBalanceCounters(msg.Query());
            sndMessage(msg.user,
                       msg.rankedDocumentsList);

           break;
    }
}
```

Bulk-synchronous parallel server

The details of the operations executed by the parallel server are described in the next section as its design is conditioned by the inverted lists strategy being employed. In general, the server executes a never-ending sequence of supersteps in which batches of q terms are processed in each processor along with the ranking of documents. The terms are routed to each processor by the broker as described above. In each superstep, the processing of a new batch is started. Depending on the kind of index strategy employed the processing of a given term plus the associated ranking of documents can take two or more supersteps to complete. Thus at any given superstep we can have several batches being processed, each at a different stage of execution. This is intended to achieve a good amortization of communication and synchronization costs.

Local and global inverted lists

For a collection of documents the inverted lists strategy can be seen as a vocabulary table in which each entry contains a term (relevant word) found in the collection and a pointer to a list of document's identifiers that contains such term. These lists are called *inverted-lists*. Thus, for example, a query composed of the logical AND of terms 1 and 2 can be solved by computing the intersection between the inverted-lists associated with the terms 1 and 2. The resulting list of documents can be then ranked so that the user is presented with the most relevant documents first (the technical literature on this kind of topics is large and diverse, e.g., see [3]). Parallelization of this strategy has been tackled using two approaches [2, 9, 19, 27, 32].

In the local index approach the documents are assumed to be uniformly distributed onto the processors. A local inverted-lists index is constructed in each processor by considering only the documents there stored respectively. We thus have P individual inverted-lists structures so that a query consisting of, say, one term must be solved by simultaneously computing the local sub-lists of document identifiers in each processor, and then producing the final global list from these P local sub-lists.

The BSP realization of the local index approach is as follows. Once the broker machine routes by ways of the hashing function a term w belonging to a query u to processor i, this processor broadcasts the term w to all other processors in the current superstep. Every processor does the same for each term they receive. In the following superstep, all processors scan their local inverted lists to obtain the sub-list of document identifiers for each term they received in the previous broadcast. These sub-lists are then sent to the processors acting as rankers. Thus if processor k happens to be the ranker for query u, the sub-list associated with term w in processor i along with the ones located in all other processors for term w, are routed to processor k. The same is effected for all other terms belonging to the query u so the processor k can perform the final ranking in the following superstep. The size of these sublists is reduced by performing a pre-ranking before sending them to their rankers. Also if two terms of query u happens to be in the same processor a pre-merging is performed (see [2] for this kind of optimizations). The whole process of a query u takes 3 supersteps to complete and send back the final list of document identifiers to the broker.

The second approach is the so-called global index. Here the whole collection of documents is used to produce a single inverted lists index which is identical to the sequential one. Then the T terms that form the global term table are uniformly distributed onto the P processors along with their respective lists of document identifiers. This is done by ways of the same hashing function employed by the broker. Thus, after the mapping, every processor contains about T/P terms per processor. In the local index case, each processor contains the same T terms but the length of document identifier lists are closely a fraction $1/P$ of the global index ones.

The BSP realization of the global index is as follows. Like the previous strategy, each term is routed to one server processor by the broker. For each term w belonging to a query u the inverted lists associated with terms of u are retrieved in their respective processors. Then these lists are sent to the ranker processor defined for the query u to then proceed in the next superstep like the local inverted lists case. The whole process takes 2 supersteps to complete.

An efficiency reduction problem in the global index strategy may arise when the most frequently visited terms tend to be located in the same processors. This produces load imbalance both in computation and communication. There exists some solutions to this problem. For example, in [9] a statistical analysis of terms co-occurrence in every document is effected at initialization time in order to determine which terms should be mapped on different processors. This is an example of static mapping. However, below we provide empirical evidence that when static mapping is done in a randomized manner by using a hash function this imbalance does not arise. On the other hand, a dynamic re-distribution of terms can be applied to move pairs (term, list) to other processors when load imbalance is detected.

Note that real-life textual databases produce index structures with very differing lengths for the document identifier lists (e.g., try to evaluate the Zipf's formula described in [3]). Combining a term with a small-sized list with a term with a large-sized one into the same query can have a catastrophic effect in the global index approach. The local index approach does not have this problem but it is less efficient with small-sized lists because of the relative increase of the cost of broadcasts. Yet small sized lists arise frequently when user put into their queries terms which are very specific to the subject they are looking for. This clearly suggests an approach which combines the two strategies. That is, terms with large identifier document lists are treated using the local index approach whereas terms with small lists are treated using the global one. We call this composite inverted lists.

Simulation study

To motivate the introduction of the composite approach described in the next section, we present simulation results describing the performance of the two the local and global approaches to parallel inverted lists. Note that other researchers have already shown that it is feasible to achieve efficient performance with distributed inverted lists on clusters [2]. Also the fully parallel construction of inverted lists has been investigated in [26].

Table 1 shows the performance of global and local inverted lists for 8 ... 64 processors. The results were obtained for 128 new query arrivals per superstep with simulation runs of 20000 completed queries. Queries were randomly generated by choosing uniformly from 1 to 4 terms, wherein terms for the first and second part of the table were also chosen uniformly at random from a total of 1300 and 6500 terms respectively. The sizes of the lists associated with these terms ranged from L_a (the maximum) to L_b (the minimum) in accordance with the Zipf's law normalized to the FR-TREC collection excluding stopwords as described in [1].

Efficiency columns E_e and E_m indicate the load balance (speed-up divided by the number of processors) for computation and communication respectively. The ratio m/e indicates the total amount of communication over the total amount of computation effected during the simulation.

The results of table 1 confirm the above claims, namely the global and local approaches can outperform each other under different situations. First and given that only 128 new queries arrive in each superstep, for 64 processors we have a modest amount of terms to be processed in each cycle (between 128 and 512, with average 320). Thus as the number of processors increases from 8 to 64 we see a clear reduction of the efficiencies in computation and communication. As expected, the local strategy has better efficiencies in all cases. Its efficiency is decreased only because of imbalance in the process of composing the final answers to queries. However, when the differences between inverted list sizes is small, the efficiencies of the global approach are competitive. In addition, for small sized inverted lists the efficiencies in both cases are fairly the same but the larger values of m/e for the local approach show that broadcasts start to have a significant effect in the communication cost. Note that queries using terms which are more specific are expected to work on small inverted lists. When we increase from 128 to 1024 the number of queries that arrive at every superstep, efficiencies increase to be near optimal. This for terms selected uniformly at random. However, if we chose terms with very large and very small inverted lists sizes to compose the queries, this in an alternate and random fashion, then the efficiencies decrease rather dramatically in the global lists approach. A situation like this can arise when users submit queries that contain very specific terms and less specific ones.

Note that in the proposed BSP server both the selection and ranking of documents is performed in the BSP processors. In particular, the ranking, which is running time demanding, is performed in parallel by attempting to choose a

			global	lists	index
P	L_a	L_b	E_e	E_m	m/e
8	116	76	0.88	0.88	0.68
16	116	76	0.82	0.79	0.73
32	116	76	0.75	0.70	0.73
64	116	76	0.62	0.56	0.75
8	104355	76	0.58	0.72	0.25
16	104355	76	0.47	0.52	0.35
32	104355	76	0.30	0.35	0.41
64	104355	76	0.19	0.18	0.52
			local	lists	index
P	L_a	L_b	E_e	E_m	m/e
8	116	76	0.92	0.93	1.38
16	116	76	0.89	0.89	1.52
32	116	76	0.82	0.85	1.69
64	116	76	0.76	0.79	1.92
8	104355	76	0.97	0.92	0.29
16	104355	76	0.91	0.81	0.45
32	104355	76	0.81	0.72	0.65
64	104355	76	0.58	0.50	0.83

Table 1. Global vs local inverted lists. 128 new queries per superstep

different processor to rank the documents associated with every query. For small rate of query arrivals per superstep, the way we select these processors can have a significant impact on load balance and thereby in computation and communication efficiencies. The load balancing method we described in section 3 is simple and allows the achievement of efficiencies much better than just selecting at random the ranker processor.

Composite Inverted Lists

It is straightforward to combine into a single BSP algorithm the two above described inverted lists strategies. Each processor maintains a hash table to keep information about which terms are kept as in the local index case and which as in the global one. The following pseudo-code shows the superstep executed by a BSP server using a composite inverted list index to solve user queries,

```
while(true)  // Every BSP processor.
{
 * Receive new messages and put them
   in a queue Q.

 * Foreach message msg in Q do
   {
    switch( msg.type )
    {
     case BROKER: // new term from the broker.
      if ( IsLocal(msg.term) == true )
        Broadcast(msg.term);
      else
      { // retrieve and sub-rank doc list.
        List= getInvertedList(msg.term);
        subList= preRanking(List);

        // buffer message for the ranker proc.
        bufferMsg(msg.ranker,RANKING,subList);
      }
     break;

     case BROADCAST:
      List= getInvertedList(msg.term);
      subList= preRanking(List);
      bufferMsg(msg.ranker,RANKING,subList);
     break;

     case RANKING:
      if ( queueSize( msg.queryId ) ==
                         msg.numTermsQry )
      {
        L = dequeueAll(msg.queryId);
        List= CalculateFinalRanking(L );
        bufferMsg( broker, SERVER, List);
      }
      else // queue up to wait for terms
        enqueue(msg.queryId,msg);
    }
   }

 * Send all buffered messages to their
   target processors, and synchronize.
}
```

A term is treated as global or local depending on the size of its associated inverted list. We set the maximum size of a list to be the one which produces the same ratio of computation to communication than the global inverted list approach. List sizes below this maximum are treated as in the global index case whereas sizes above the maximum are treated as in the local index one.

This straightforward combination of the local and global approaches is a strategy which we have found to be practical, efficient and very simple to implement. Its efficiency comes from the fact that most queries containing relevant terms tends to have inverted lists of small sizes. Table 2 shows simulation results for the same conditions to the one above described. It can be seen that efficiencies are similar to those of the local approach whilst the ratio communication/computation (m/e) are similar to that of the global approach.

4 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.

A typical query consists of 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 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. Processing a single query of this kind in a text of size N takes $\log N$ time on the standard sequential suffix array.

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

	comp.	lists	index
P	E_e	E_m	m/e
8	0.88	0.88	0.68
16	0.82	0.79	0.73
32	0.75	0.70	0.73
64	0.62	0.56	0.75
8	0.97	0.90	0.25
16	0.90	0.78	0.37
32	0.75	0.61	0.44
64	0.53	0.43	0.54
	global	lists	index
P	E_e	E_m	m/e
8	0.88	0.88	0.68
16	0.82	0.79	0.73
32	0.75	0.70	0.73
64	0.62	0.56	0.75
8	0.58	0.72	0.25
16	0.47	0.52	0.35
32	0.30	0.35	0.41
64	0.19	0.18	0.52
	local	lists	index
P	E_e	E_m	m/e
8	0.92	0.93	1.38
16	0.89	0.89	1.52
32	0.82	0.85	1.69
64	0.76	0.79	1.92
8	0.97	0.92	0.29
16	0.91	0.81	0.45
32	0.81	0.72	0.65
64	0.58	0.50	0.83

Table 2. Composite, global and local inverted lists. 128 queries per superstep. The first part of the table is for L_a= 116 and L_b= 76, and the second part is for L_a= 104355 and L_b= 76.

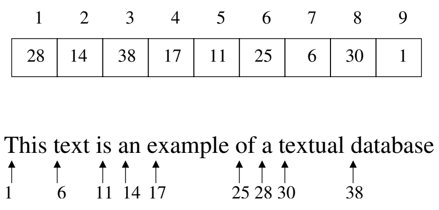

Figure 1. Suffix array.

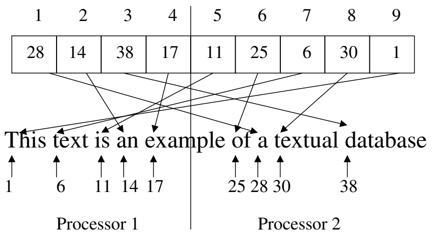

Figure 2. A global index suffix array distributed on two processors.

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.

A search for all text positions associated with a batch of $Q = q\,P$ queries can be performed as follows. The broker routes the queries to their respective target processors. 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 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. Note that, 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 a number of batches at different stages of execution. The net effect is that at the end of every superstep we have the completion of a batch. We call this strategy G0.

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 [14] 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.

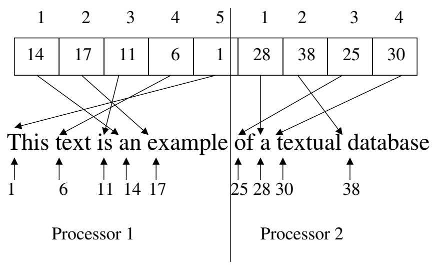

Figure 3. Local index suffix array.

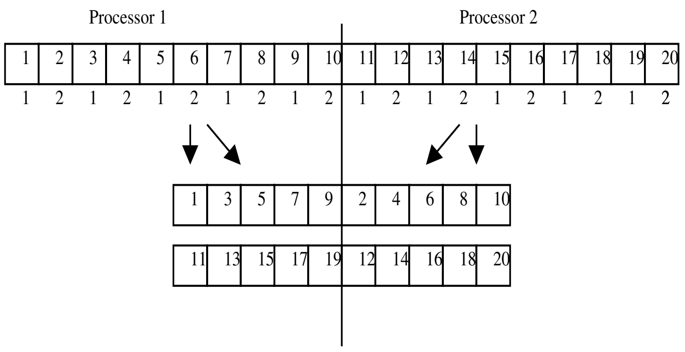

Figure 4. Multiplexing the global index suffix array entries.

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. Unfortunately, the broker must send every query to every processor which can become significant cost for large number of processors.

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.

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.

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. In [17] 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). The practicality of this

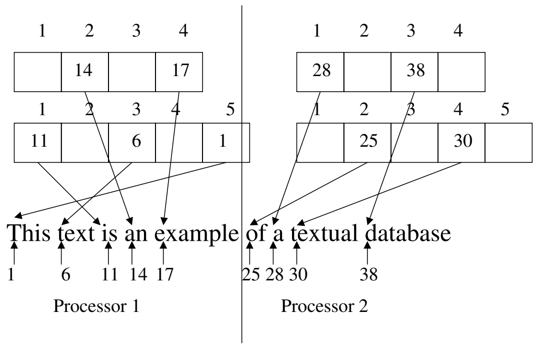

Figure 5. Combining multiplexing with local-only references.

method remains to be further investigated.

Experimental Results

We compared the multiplexed strategy (G2) with the plain global suffix array (G0), and the intermediate strategy (G1). The local index strategy was at least 3 times slower than all others so we do not show empirical results for it. 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).

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. 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.

We define 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 3(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 3(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.

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 3(3) we show running time ratios obtained with our 16 machines cluster. The upper of the table shows re-

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
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

(3) Running times ratios

Table 3. Comparison of search costs.

sults for the biased query terms (queries of type (2)) and the lower part shows results for terms selected uniformly at random (queries of type (1)). The biased workload increased running times by a factor of 1.7 approximately.

The results of Table 3(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). 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.

5 Research topics

Inverted lists. Note that a better approach for large number of processors is to actually look at an intermediate situation between the local and global approaches. The practical realization of this idea is to set buckets of a fixed size whose value is defined by the capacity of disk pages. Every inverted list is divided in a certain number of buckets which are distributed evenly onto the processor at initialization. The key problem to solve in this case is the method employed to properly balancing the processing of queries as we explain in the following.

A distribution of buckets uniformly at random onto the processors is expected to solve in part the load imbalance produced by queries containing natural language text. However it is necessary to consider that the set of documents produced as a result of a given query must be ranked to present them to the user sorted by relevance order. Previous approaches leave this task to the broker machine. We believe that this is not a good idea because ranking demands a significant amount of computation which can transform the broker in a bottleneck. A broker machine should not perform much more operations than just routing arriving queries to the server processors, receiving the respective answers and pass them back to the user.

Document ranking in parallel (which is a parallelization of a strongly sequential algorithm presented in [25]) is effected in [2, 27] using two major steps. First, for each term present in a batch query, a pre-ranking is performed among the respective documents associated with the term. Then a subset of the retrieved documents are selected to perform the final ranking among all the documents associated with the terms contained in the query. In [2] this final ranking is performed in the broker. We believe it is more efficient to effect this in one of the server processors so that good parallelism is achieved when queries from one or more batches reach this stage and the final ranking tasks are scheduled in different processors in a well balanced manner. No scheduling algorithms for this case have been proposed so far.

Note that it is possible, as we did in the case of the composite inverted lists approach [16], to employ heuristics such as “the least loaded processor first”. However, it is also possible to think in terms of whole batches of queries rather than applying heuristics to individual queries. An interesting research topic here is the analysis of existing job scheduling algorithms to see which are more suitable for this case and how they can be modified to deal with dynamically biased user query terms as in natural language applications. In this case dynamic re-allocation of buckets is a feasible option.

Suffix arrays. The efficient construction of suffix arrays is a non-trivial problem which by itself justifies the use of parallel computing because of the large sequential running times involved. In the parallel setting the main problem to cope with is the lack of locality. What one basically looks for is an evenly distributed array of pointers to text positions or documents that usually are located at different processors. The array is lexicographically sorted in the sense that the string pointed to by the position i of the array is less than the one pointed to by position $i + 1$. Previous work on parallel construction of suffix arrays has been done in [14, 24] for the message passing model of parallel computing. No implementations have been effected and the strategies proposed have been based on the adaptation of various sorting algorithms, some of them specialized to text. However, very recently three new sequential algorithms have been proposed for constructing suffix arrays (submitted). It is then relevant to study how to use the concepts behind those algorithms to formulate BSP algorithms for this problem.

Dealing with compressed text. Compression techniques are widely used in natural language text databases [10, 21, 22, 23]. These techniques are particularly relevant to parallel text databases because they reduce the amount of information to be transmitted among server machines themselves and between server and broker machines.

In a distributed memory parallel computation setting like the one supported by the BSP model, the database is evenly distributed onto the machines. In this "sharing-nothing" scheme, text compression can be applied in each machine as they were independent text collections. Alternatively to this local approach, text can be compressed by considering the whole text collection. In both cases the amount and type of operations that is necessary to effect are different.

Very recently a new method for text compression was proposed in [8]. The method is particularly suitable for searching. It is worthwhile to investigate the effectiveness of this method in the parallel context. A key part of the compression process is the construction and maintenance of the coding alphabet table that allows the mapping between actual words and the compressed representation of them. This table enables encoding of new text being entered to the database and the necessary decoding for presentation and other important operations such as searching. It appears interesting to investigate efficient ways of maintaining this table as new text is distributed onto the machines and search queries must react properly to this dynamics.

Concurrency control. Another type of problem that is worthwhile to study is concurrency control upon inverted lists. In this case updates take place simultaneously with queries (e.g., a news service) [11]. In [13] the classic inverted list approach [3] is instrumented with a simple concurrency control mechanism which is based on the use of locks. However, locks tend to significantly reduce the level of parallelism that is possible to exploit from typical workloads for text databases applications. The application of optimistic concurrency control techniques such Time Warp [12, 15] remains to be investigated so far [3, 4, 5, 6, 7, 13, 28, 31, 36].

References

[1] M.D. Araujo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Workshop on String Processing (WSP'97)*, pages 2–20. (Carleton University Press), 1997.

[2] C. Badue, R. Baeza-Yates, B. Ribeiro, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Eighth Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 10–20. (IEEE CS Press), Nov. 2001.

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

[4] R. Baeza-Yates, A. Moffat, and G. Navarro. *Handbook of Massive Data Sets*. Kluwer Academic Publishers, 2002. Chapter 7, pages 195–243.

[5] N. Barghouti and G. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3), September 1991.

[6] B. K. Bhargava. Concurrency control in database systems. *Knowledge and Data Engineering*, 11(1):3–16, 1999.

[7] S. Blott and H. Korth. An almost-serial protocol for transaction execution in main-memory database systems. In *28th International Conference on Very Large Data Bases*, Aug. 2002. Hong Kong, China.

[8] N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In *Proceedings of the 25th European Conference on Information Retrieval Research (ECIR'03)*, LNCS 2633, pages 468–481, 2003.

[9] 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.

[10] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, April 2000.

[11] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[12] D.R. Jefferson. Virtual time. *ACM Trans. Prog. Lang. and Syst.*, 7(3):404–425, July 1985.

[13] M. Kamath and K. Ramamritham. Efficient transaction management and query processing in massive digital databases. Technical Report UM-CS-1995-093, University of Massachusetts, , 1995.

[14] 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.

[15] M. Marín. Time Warp on BSP Computers. In *12th European Simulation Multiconference*, June 1998.

[16] M. Marín. Parallel text query processing using Composite Inverted Lists. In *Second International Conference on Hybrid Intelligent Systems (Web Computing Session)*. IO Press, Feb. 2003.

[17] M. Marín and G. Navarro. Distributed query processing using suffix arrays. In *Int. Conf. on String Processing and Information Retrieval*, Lecture Notes in Computer Science. Springer-Verlag, Sept. 2003. (to appear).

[18] M. Marín and G. Navarro. Suffix arrays in parallel. In *International Conference on Parallel Processing (EuroPar'03)*, Lecture Notes in Computer Science, page (to appear). Springer-Verlag, Aug. 2003.

[19] 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. (IEEE CS Press), 2000.

[20] 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.

[21] A. Moffat. Word-based text compression. *Software – practice and experience*, 19(2):185–198, 1989.

[22] A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. In *Data Compression Conference*, pages 170–179, 1996.

[23] G. Navarro, E. Silva de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information retrieval*, 3(1):49–77, 2000.

[24] 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. (LNCS 1264), 1997.

[25] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[26] B. Ribeiro-Neto, 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. (IEEE CS Press), 1998.

[27] B.A. Ribeiro-Neto and R.A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Third ACM Conference on Digital Libraries*, pages 182–190. (ACM Press), 1998.

[28] S.Dandamudi and J.Jain. Architectures for parallel query processing on networks of workstation. In *1997 International Conference on Parallel and Distributed Computing Systems*, Oct. 1997.

[29] 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.

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

[31] T. Tamura, M. Oguchi, and M. Kitsuregawa. Parallel database processing on a 100 node pc cluster: Cases for decision support query processing and data mining. In *SC'97*, 1997.

[32] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Second International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1993.

[33] URL. BSP and Worldwide Standard, http://www.bsp-worldwide.org/.

[34] URL. BSP PUB Library at Paderborn University, http://www.uni-paderborn.de/bsp.

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

[36] C. Yu and C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, December 1984.