Analysis of two indexing structures for textual databases

Mauricio Marín, Carolina Bonacic
Departamento de Computación
Universidad de Magallanes
Punta Arenas, Chile
{mmarin,cbonacic}@ona.fi.umag.cl

Sandra Casas
División de Informática
Universidad N. Patagonia Austral
Rio Gallegos, Argentina
sicasas@spse.com.ar

Abstract

This article describes strategies devised to improve the efficiency of two classical index data structures for parallel textual databases. The design and cost evaluation is effected on top of the bulk-synchronous model of parallel computing. This allows us to compare different alternatives under the same framework in a way which is independent of programming details and architecture of the parallel machine. Our interest is on query processing upon cluster of PCs, and thereby we focus on communication and synchronization optimization.

1 Introduction

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

One approach is to just duplicate the same server into a collection of machines doing the same work upon the same collection of data, wherein clients are distributed uniformly at random on them. Though this scheme is simple, it has the problem of dealing efficiently with data consistency. But even for services of the type read-only, this approach is not convenient when the data collection is huge and there exists opportunities to exploit the locality of access patterns; a common situation in many applications of textual databases (e.g., document retrieval in search engines [3]).

In our view, a more promising approach is to split up the data collection and distribute them onto the processors in such a way that it becomes feasible to exploit locality by effecting parallel processing of user requests, each upon a subset of the data. As opposed to shared memory models, this distributed memory model provides the benefit of better scalability [6]. However, this introduces new problems related to the communication and synchronization of processors.

In this paper we investigate the feasibility of using the bulk-synchronous parallel (BSP) model of computing [10, 12] for solving these problems. Like previous work on traditional models of parallel computing (mostly message passing) [2, 4, 9, 1, 11], we look at the efficient BSP parallelization of two classical sequential index data structures, namely the *inverted list* and *suffix array* approaches [3]. However, unlike traditional models, as BSP is a much more structured style of parallel computation, we are capable of studying the comparative performance of different alternatives by using an implementation and architecture independent approach. This is effected by ways of the BSP cost model, which in simple terms means that our conclusions are not so restricted to benchmarks on particular data sets like previous performance evaluation studies. Mainly we use the BSP cost model to compare alternative ways of parallelizing the two sequential index structures (some of them resulting equivalent to previous approaches). This enabled us to identify situations in which it is possible to achieve the best performance, though at this stage of our research we have not come to realize yet a practical implementation of all of them.

2 Model of computing and server design

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 organised as a sequence of *supersteps*. During a superstep, the processors may perform sequential computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronisation of the processors.

The total running time cost of a BSP program is the cumulative sum of the costs of its supersteps, and the cost of each superstep is the sum of three quantities: w, $h\,G$ and L, where w is the maximum of the computations performed by each processor, h is the maximum of the messages sent/received by each processor with each word costing G units of running time, and L is the cost of barrier synchronising the processors. The effect of the computer architecture is included by the parameters G and L, which are increasing functions of P. These values along with the processors' speed s (e.g. mflops) can be empirically determined for each parallel computer by executing benchmark programs at installation time [10].

As an example of a basic BSP algorithm let us consider a broadcast operation which will be used in the algorithms described in the following 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. Thus in superstep 2 all P processors have available into their respective incoming message buffers the P chapters of the book. An optimal algorithm for the same problem is as follows. In superstep 1, the sending processor sends just one *different* chapter to each processor at a cost of $O(P\,(a+a\,G)+L)$ units. In superstep 2, each processor sends its arriving chapter to all others at a cost of $O(P\,(a+a\,G)+L)$ units. Thus at superstep 2, all processors have a copy of the whole book.

That is, the broadcast of a large P-pieces a-sized message can be effected at $O(P\,(a + a\,G) + L)$ cost.

The practical model of programming is SPMD, which is realized as C and C++ 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 parallel programming paradigm and not a particular communication library. In practice, it is certainly possible to implement BSP programs using the traditional PVM and MPI libraries.

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

Clients request service to one or more front-end 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.

A pseudo-code for the most basic steps executed in each processor is as follows,

```
// All of the BSP processors execute the same code.
while(true)
{
// Superstep 1
  * Receive new messages.
  * Queue new queries arrived from front-end machines.
  * Form a batch of queries by taking new queries from the queue.
  * Initiate parallel processing of queries by (possibly)
    buffering messages with tasks to do in other processors.
  * Send buffered messages to their destinations.

  Synchronize processors

// Superstep 2
  * Receive new messages.
  * Process its own batch of queries along with queries
    received from other processors.
  * Send buffered messages to their destinations.

  Synchronize processors
```

```
// Superstep 3
  * Receive new messages.
  * Integrate partial results coming from the other processors
    to produce the answers to the current batch of queries, and
    buffer them as messages to be sent to their respective
    front-end machines.
  * Send buffered messages to their destinations.

  Synchronize processors
}
```

It is not difficult to see that the above cycle of three supersteps can actually be reduced to a cycle of one superstep by dealing with three separate batches, each in a different stage of execution. Also note that variations to this basic code are introduced in accordance with the particular requirements of the index structures being used to speed-up the solution to client queries.

3 Inverted lists

For a collection of documents the inverted lists strategy can be seen as a 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. Thus, for example, a query composed of the logical AND of terms 1 and 2 can be solved by computing the intersection between the 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 more 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.

In the local index approach the documents are assumed to be uniformly distributed onto the processors. A local inverted list is constructed in each processor by considering only the documents there stored respectively. We thus have P individual inverted lists 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 lists.

The BSP realization of the local index approach is as follows. Once the front-end machine routes an one-term query to processor i, this processor broadcasts the query to all other processors in the current superstep. In the following superstep, all processors scan their local inverted lists to obtain the sub-list of document identifiers. This sub-list is then sent to the requesting processor i. Thus in the next superstep the processor i produces the final list of document identifiers, performs a ranking on those documents, and sends the ranked final list back to the requesting front-end machine.

The second approach is the so-called global index. Here the whole collection of documents is used to produce a single inverted list 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. Two terms are put on a different processor if they are

likely to appear in the same query. In addition, 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 document identifier lists are closely a fraction $1/P$ of the global index ones.

The BSP realization of the global index is as follows. As one-term queries are fully solved in a single processor, either the front-end machine or the receiving BSP processor must know the mapping rule employed during the distribution process of terms onto the processors. A more-than-one-term query can be treated as a group of one-term queries and then composed to form the final answer (we should expect that most typical user queries from the Web have a few terms).

The BSP cost of the two strategies is the following. Suppose that the average length of the document identifier lists is W. For large textual databases we should expect $W \gg P$. We also should expect that the term tables (usually called the vocabulary) fit fully on their respective main memories. Document identifiers lists are supposed to be stored on secondary memory. We consider the cost from the front-end machine to the instant at which this machine receives the results from the server. We cost the communication between the two using the BSP cost model. Actually front-end machines can well be processes located in the BSP processors themselves.

In the local index approach it is necessary to let each processor work on the same set of $Q = q\,P$ queries. It is thus necessary to perform a broadcast of Q queries so that all processors get a copy. Assuming just one front-end machine and using the method described in section 2, this operation can be done at cost $P\,(q + q\,G) + L$. However, in the proposed implementation of the BSP server what happens is that every processor retrieves q queries from its incoming queries queue, and broadcasts them to all other processors at $q + q\,P\,G + L$ cost. After this broadcast, the processors work on the determination of the document identifier lists for the $Q = q\,P$ queries, and send every list of average size W/P to their source processors so that they can compose the final list of average size W. The cost of this superstep is $Q\,(W/P)\,(D + G) + L = q\,W D + q\,W\,G + L$. Finally, in the last superstep, the final lists of size W are composed and sent to the, say, P front-end machines or processes, which is done at $Q\,(W/P)\,(1 + G) + L = q\,W + q\,W\,G + L$ cost. Thus the asymptotic cost of the local index strategy is

$$q\,W\,D \;+\; q\,(\,P + W\,)\,G \;+\; L\,.$$

For the case of one front-end machine, the above cost must be incremented in $P\,(q\,W\,G)$ units. Clearly, it is more efficient to have one front-end (machine or process) per BSP processor, and let them route queries at random among the BSP processors. Randomness is convenient since it allows the achievement of good load balance in cases in which front-ends are not expected to receive similar number of queries from the Web.

The global index strategy is cost as follows. The distribution of $q\,P$ queries from a single front-end machine takes similar time to the broadcast of the previous strategy, namely $P\,(q + q\,G) + L$. In the proposed BSP server, once every processor gets its q queries, they build in parallel the identifiers lists for each query at a cost of $q\,W\,D$ units, and then send the results at a cost of $q\,W\,G$ units if P front-ends are participating. Thus the total asymptotic cost is given by

$$q\,W\,D \;+\; q\,W\,G \;+\; L\,.$$

For real-life situations where $W \gg P$ (cases which justify the use of parallelism) the costs of the two approaches is essentially the same. Similar to the previous case, a term $P\,(q\,W\,G)$ must be included in the above expression when the results are sent to a single front-end machine. Also an additional superstep is needed in situations in which each term of a, say, two-terms query is processed in a different processor. In this case it is necessary to send one of the W-sized lists to the other processor at a cost of $W\,G + L$ units. All this makes the global index strategy more similar to the local index strategy.

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 [4] 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 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. This is a topic we plan to investigate in the near future. To the best of our knowledge, no dynamic load balancing strategies has so far been investigated in this context (at least none in BSP).

The cost of the sequential inverted list strategy is simply $q\,P\,W\,D$ or it may well be $q\,P\,W\,D + q\,P\,(1+W)\,G + L$ when either one or more front-end machines are considered, and they are actually separated from the sequential server (we assume the former case for comparison purposes). Thus the comparative performance of parallel inverted lists can be severely limited by its communication costs when processing requests from just one front-end machine, as its performance is bounded by $P\,(q\,W\,G)$.

Note that for systems with small W, we can expect a modest performance improvement of the global index over the local index approach. This is because the cost of broadcasts in the local index approach tends to be more significant for small W. However, 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. 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 are currently investigating an efficient way of implementing this composite inverted lists approach.

For the global index approach, we have realized that the initial mapping of the inverted lists structure onto the BSP processors can be made in a very simple and efficient manner. Once the whole sequential structure has been completed by using the usual procedure (the fully parallel construction of inverted lists has been investigated in [8]), we can pick up one by one the pairs (term, list), where "list" is the list of document identifiers where the "term" occurs in, and distribute them evenly onto the processors by selecting them uniformly at random. For this we can construct

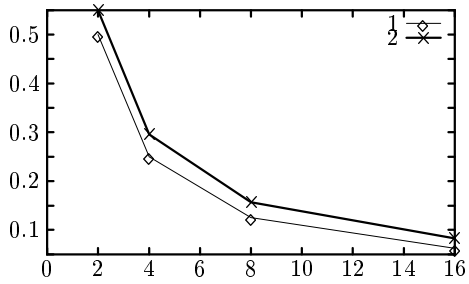

Figure 1: Load balance of global inverted lists.

a hashing function on the term characters. The same function is used by front-end machines during the routing of queries. A query with two or more terms is treated as a set of two or more one-term queries, and one of the BSP processors (uniformly at random as well) is given the task of composing the final document list for these many-terms queries.

In figure 1 we show normalized load balance measures for random queries of 1 to 4 terms for 2, 4, 8 and 16 processors. The inverted list was constructed from a Chilean newspaper web site with about 50MB of text. What the curve labelled 1 shows is the optimal load balance given by $1/P$ decrease. The curve labelled 2 shows the actual data obtained from the experiments. It shows that near optimal balance is achieved in all cases. The values were computed taking the average of the observed maxima across supersteps. These results are intuitive if we consider that pairs (term, lists) are randomly distributed, large batches of $Q = q\,P$ queries are processed, multiple terms queries are treated as a set of individual queries, and their composition is performed in a randomly selected processor. This also means that in a given superstep we can have the P processors composing results from different queries in parallel.

4 Suffix arrays

The suffix array is a binary search based strategy [3]. The array contains pointers to the terms, where pointers identify both documents and positions of terms within them. The array is sorted in lexicographical order by terms as shown in figure 2. 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 2. This search is conducted by direct comparison of the terms pointed to by the array elements. The meaning of "term" in this strategy can be different to that of the inverted lists. Though it depends on the application, a "term" could be the entire text, or a large portion delimited from a certain point to the end. The suffix array can be employed to support search by regular expressions.

Similar to inverted lists, the suffix array can be distributed onto the processors using the global index approach. This is shown in figure 3. This index is built considering the whole collection of documents. The efficient parallel index construction has been investigated in [7, 5]. Notice that in this global index each processor stands for an interval or range of terms (for example, in figure 3

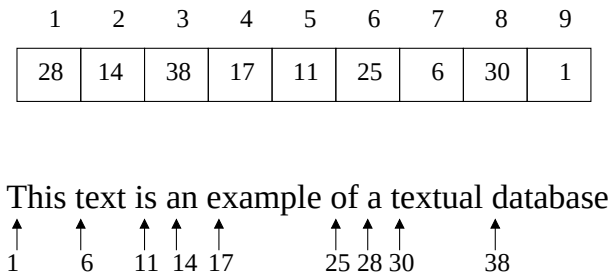

Figure 2: Suffix array.

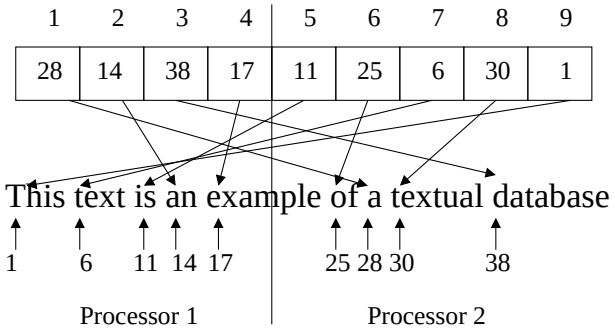

Figure 3: A global index suffix array distributed on two processors.

processor 1 represents words with first letters from "a" to "e").

A search for all documents associated with $Q = q\,P$ queries can be performed as follows in the global index approach. The processors get their q queries and each performs in parallel q binary searches. For a global array of size N, this can be performed at cost $q + q\,\log(N/P) + q\,W\,D$ with W being the average number of documents in which a given term is in. Then the document identifiers are sent to P front-end machines at a cost of $q\,W\,G + L$ units. For one front-end, this cost increases to $q\,P\,W\,G + L$. In this approach it is necessary for front-ends to determine to which processor (interval) to route a given query. To this end, front-end machines can keep $P - 1$ pivots delimiting the respective intervals. The routing can be done at cost $q\,P\,G + L$ by one front-end. Thus the cost for P front-ends is given by

$$q\,(\,1 + \log(N/P) + W\,D\,) + q\,W\,G + L\,.$$

In the local index strategy, a suffix array is constructed in each processor by considering only the subset of documents stored in each processor. See figure 4. No references to documents stored in other processors are made. However, it is necessary to search in every processor in order to form the average list of W document identifiers for a given term.

In this case, a search for all documents associated with $Q = q\,P$ queries can be performed as follows. The processors get their q queries, and broadcast them to all other processors at a cost of $q + q\,P\,G + L$. Then, in the next superstep, each processor performs in parallel $q\,P$ binary searches

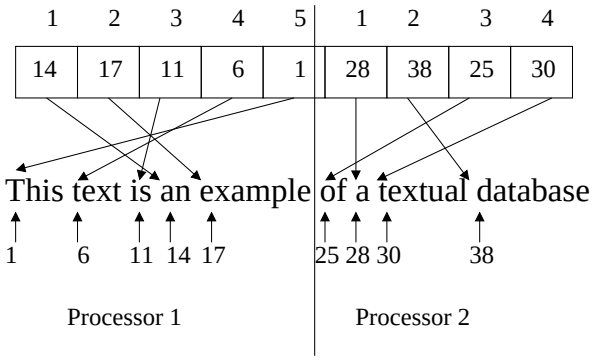

Figure 4: Local index suffix array.

to get the partial lists of documents associated with each term. For a global array of size N, this can be performed at cost $q\,P\,\log(N/P) + q\,W\,D$ so that each term gets on average W/P document identifiers. Then these identifiers are sent to the processors that originated every query at a cost of $q\,W\,G + L$ units. Next the final lists are formed for every query at a total of $q\,W$ units, and sent to P front-end machines at $q\,W\,G + L$ units of running time. For one front-end, this cost increases to $q\,P\,W\,G + L$. Thus the cost for P front-ends is given by

$$q\,(\,1 +\,P\,\log(N/P) + W\,D\,) + q\,(P + W)\,G + L\,.$$

The sequential cost is given by

$$q\,P\,\log(N) + q\,P\,W\,D\,,$$

where the additional term $q\,P\,(1 + W)\,G + L$ should be included whenever front-end machine(s) is (are) separated from the server. The same arguments to inverted lists apply regarding $q\,P\,W\,D$ versus $q\,(P + W)\,G + L$ or even $q\,P\,W\,G + L$. However, in this last case it is clear that the global index approach offers the best opportunity for efficient performance. It is worthwhile then to focus on how to improve some performance drawbacks of the global index strategy.

The first one 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 as shown in figure 5. In this case, any binary search can start in any processor. Once a search has determined that the given term must be located between two consecutive entries of the array in a processor, the search is continued in the next processor and so on. This last inter-processors search can be done in at most $\log P$ additional supersteps. For example, in figure 5 a term located in the first interval, may be located either in processor 1 or 2. If it happens that a search for a term located at position 6 of the array starts in processor 1, then once it determines that the term is between positions 5 and 7, the search is continued in processor 2 by directly examining the position 6.

As shown in figure 3 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

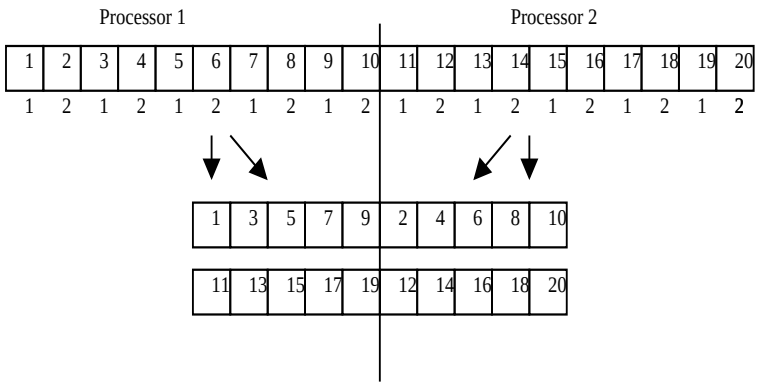

Figure 5: Multiplexing the global index suffix array entries.

in superstep $i+1$, reads the term, composes and sends to p a message containing it. In superstep $i+2$ the processor p gets the term to perform the comparison which allows it to continue the binary search. Some cache scheme can be implemented in order to keep in p the most frequently referenced terms from remote memory. This reduces the average, with the worst case requiring $\log(N/P)$ additional supersteps, namely the cost of the global index considering the remote accesses is given by

$$q\,(\,1+\log(N/P)+WD\,)+q\,W\,G+\log(N/P)\,L\,.$$

However, this worst case cost can be amortized by employing similar strategy to that described for the pseudo-code of section 2. That is, the processing of $\log(N/P)$ simultaneous batches, all at different stages of execution. Also queries of a given batch not necessarily ends at the same superstep, so we can replace the finished ones with new ones as soon as they end up. This also can contribute to amortize the cost of multiplexing which only requires at most $\log P$ supersteps. Another way to reduce the average number of remote memory accesses is to associate with every array entry the first t characters of the terms respectively. This value t depends on the average length of terms. In [5] it has been shown that this strategy is able to put below 5% the remote memory references for relatively modest t values.

Yet another method which would solve both load imbalance and remote references is to re-order the original global array so that every piece located on the processors, contains only pointers to local text. This is shown in figure 6. How to reduce the memory requirements of such structure is a topic we are presently investigating. We are certain that this is an alternative (or a variant of it) which is worthwhile to study since it solves the two major drawbacks of the global index approach for suffix arrays.

5 Conclusions

We have presented BSP algorithms for implementing global and local indexes devised to support efficient query processing in textual databases. Global refers to indexes built by considering the

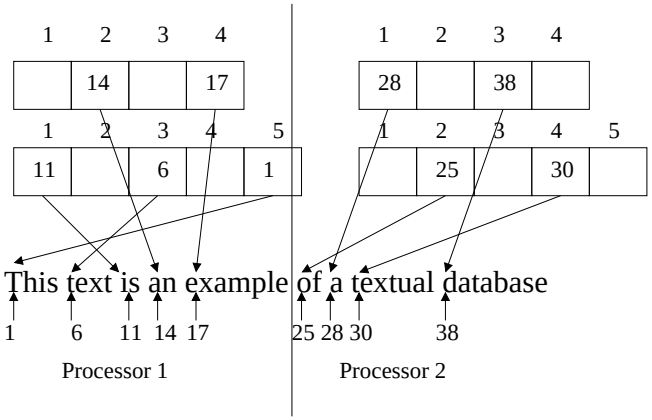

Figure 6: Combining multiplexing with local only references.

whole collection of documents, whereas local refers to indexes built by considering only a fraction $(1/P)$ of the whole collection.

In the global one, the answer to a given query is constructed by only one processor, and in the local one the answer is constructed by all P processors in parallel. This has led to the claim that local is more parallel whereas global supports more concurrency [2]. None of these claims make sense when one is posed with the problem of processing a large number of queries per unit of real time. We believe that this is precisely the case that justifies the adoption of parallel computing, namely situations of very high traffic of queries continuously flowing into a server. In this context what one wants is to optimize the throughput of the overall system rather than particular queries. We propose a general BSP algorithm for implementing such a server.

For solving the problem of speeding up the solution to queries, we have studied the BSP realization of inverted lists and suffix arrays. Their running time costs were expressed in terms of the BSP cost model in order to compare the different approaches upon the same framework. This provided us with running time expressions which are not tailored either to particular implementations of the algorithms or particular architectures of the parallel computer. This also allowed us to identify opportunities for significant optimizations of the total running time, mainly in the form of pipelining across supersteps.

Overall, our main conclusions are that (**i**) for inverted lists both the local and global index approach have essentially the same running time cost, and (**ii**) for suffix arrays, the global index approach is clearly more efficient than the local index approach, though it tends to demand the execution of more supersteps due to remote memory accesses. For the last case we have proposed alternative ways of reducing the cost of the additional supersteps.

Nevertheless, the performance of both approaches, namely local and global, are highly dependent on the cost of communication between front-end machine(s) and BSP processors. Front-end machines should be located into the same BSP processors, for example, as separate tcp/ip processes which run independently of BSP supersteps (this is perfectly feasible in current realizations

of BSP communication libraries). If this not were the case, we would need to include an extra term $q\,P\,(1+W)\,G+L$ to the sequential cost in order to provide a fair answer to the "sequentially or in parallel" question.

Acknowledgements

This work has been partially funded by research projects (a) Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile, (b) Fondecyt 1020803, and (c) UMAG-UNPA research collaboration.

References

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

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

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

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

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

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

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

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

[9] 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, 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] 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.

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

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

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