Fast Concurrency Control for Distributed Inverted Files

Mauricio Marín*

Computing Department, University of Magallanes
Casilla 113-D, Punta Arenas, CHILE
mmarin@ona.fi.umag.cl

Abstract. A new method for controlling concurrent read/write operations upon inverted files is proposed and evaluated. Communication and synchronization among processors is effected by ways of the bulk-synchronous parallel model of computing. Thanks to the global synchronization property of this model, a simple but very efficient mechanism for synchronizing read/write operations is feasible at very low overheads in running time. Experimental results using a large text collection show that our method is more efficient than traditional approaches to the synchronization problem.

1 Introduction

The inverted file [2] is a popular data structure that is frequently used as an index for text databases. Its purpose is to speed-up query operations over large text collections. A typical application is in Web search engines in which case the server site must be able to cope efficiently with thousands of query operations per unit time coming from Internet users. This has lead to the consideration of parallel realizations of inverted files [1, 5, 9, 7, 11].

Query operations over parallel search engines are usually read-only requests upon the distributed inverted file. This means that one is not concerned with multiple users attempting to get information from the same text collection. All of them are serviced with no regards for consistency problems since no concurrent updates are performed over the data structure. However, it is becoming relevant to consider mixes of read and write operations. For example, for a large news service we want users to get very fresh texts as answers to their queries. Certainly we cannot stop the server every time we add and index a few news into the text collection. It is more convenient to let writes and reads take place concurrently. Solutions to this problem using traditional approaches from relational databases developments have been proposed for inverted files in [8].

Concurrency control is perfomed by algorithms that are in charge of properly synchronizing simultaneous accesses to the underlying data structure. From the database and parallel discrete-event simulation literature we learn of a number

* Partially supported by projects FONDECYT 1030454 and UMAG PRF101IC04.

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3514, pp. 411–418, 2005.
© Springer-Verlag Berlin Heidelberg 2005

of synchronization algorithms [3]. They can be divided into conservative and optimistic ones. The two-phases locks and time warp protocols are good examples of conservative and optimistic approaches respectively [4]. In the first case, write operations are performed when it is certain that no reads are to take place whereas in the second one no such restriction is imposed and errors are detected and corrected when necessary.

In this paper we propose a conservative algorithm that departs from previous approaches as it organizes computations in a bulk-synchronous manner as understood in the BSP model of parallel computing [12]. This model is known to be efficient, portable and scalable for a wide range of applications but it has not been widely employed to support distributed indexes for text databases. In BSP, processors are globally synchronized after performing computations on local data and communication actions. Messages are available at their target processors only after the global synchronization.

We take advantage of this fact to synchronize read/write operations in a straightforward manner: queries are timestamped and organized in batches delimited by processor sychronizations. Rules for message availability and processor synchronization ensure that consistency is maintained by just processing queries in timestamp order. We apply the proposed method to a particular realization of Distributed Inverted Files though it can actually applied to any other.

2 The BSP Model and Server Configuration

The bulk-synchronous parallel (BSP) model of computing [12] is a distributed memory model with a well-defined structure that enables the prediction of running time. 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 or BSPub. In practice, it is certainly possible to implement BSP programs using the traditional PVM and MPI libraries.

In BSP, the 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 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.

We assume a server operating upon a set of P identical machines, each containing its own main and secondary memory (e.g., a cluster of PCs). The text database (documents) is evenly distributed over the P machines.

Clients request service to one broker machine, 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 file which, as described in the next section, is composed of a vocabulary (set of terms) and a set of

identifiers (inverted list) representing all the documents that contain at least one of the words that are members of the vocabulary. The inverted file 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 K 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 [2], which provides a measure of how close is a given document to a certain user query.

In order to better exploit the available parallelism we try to minimize the amount of work performed by the *broker* machine. We restrict its functionality to (**a**) receive user requests, (**b**) distribute the queries onto the processors (uniformly at random by means of a hashing function on the terms, i.e., vocabulary words), (**c**) receive the best ranked documents (K in total) from the server, and (**d**) 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.

For a collection of documents the inverted file 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 (inverted list) 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 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 [2]). Parallelization of this strategy has been tackled using two approaches.

The global index approach is as follows. The whole collection of documents is used to produce a single inverted file index which is identical to the sequential one. Then the T terms that form the global term table (vocabulary) 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. This is the strategy used by most popular search engines such as `Google` though there has been some discussions in the literature [1, 5, 9, 7, 11] about which approach is better (including variations and combinations). This discus-

sion is out of the scope of this paper and we present our results in the context of the global index approach.

The BSP realization of the global index is as follows. Every 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.

The following pseudo-code shows the major tasks performed by every processor of the server (which is a set of machines supporting the BSP model of computing) for read-only queries,

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

 Foreach message msg in Q do
 {
  switch( msg.type )
  {
   case BROKER:// term from the broker.
   // retrieve and rank doc. list
    List=FirstK-ItemsOfList(msg.term);
    subList= preRanking(List);

  // buffer message to be sent to the
  // ranker processor.
    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);
   break;
  } // switch
 } // foreach

 Send all buffered messages to their
 target processors, and globally
 synchronize the processors.
} // while
```

3 Conservative Synchronization Algorithm

Every processor of the BSP machine must execute R/W operations of a large number of queries. They are evenly distributed so that during a superstep all processors maintain about the same number of them.

For each new document to be included in the text collection, a sub-set of the vocabulary terms get their respective inverted-lists modified. Those modifications come in the form of write operations on the inverted file. The parsing process and other calculations on the new document to be included is assumed to be performed by a secondary machine which in turn sends the write operations to the broker machine. Thus the broker send the write operations as they were normal queries. That is, they are routed to the server processors using the hashing function on the vocabulary terms.

A key fact here is that the broker can assign a unique timestamp to each query it sends to the BSP server. The vocabulary terms are those which are assigned timestamps and terms belonging to the same query get identical timestamps. Now, no R/W consistency conflicts can ever take place if server processors process terms with associated R/W operations in increasing timestamps. This is true because at the end of every superstep the processors are barrier synchronized and new messages arriving from other processors are only available by the following superstep. This introduces a global order because batches of queries are send to the server and every processor executes sequentially, one by one, the queries it receives at the begining of each superstep.

Thus the R/W version of the global inverted file algorithm as implemented in the BSP model is as follows,

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

 Sort Q by increasing timestamps
 (ranking messages are not considered).

 Foreach message msg in Q do
 {
  switch( msg.type )
  {
   case BROKER_READ:// R query
    List=FirstK-ItemsOfList(msg.term);
    subList= preRanking(List);
    bufferMsg( msg.ranker,RANKING,
               subList);
   break;
   case BROKER_WRITE: // W query
    UpdateList( msg.term,
       msg.documentId, msg.info);
   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);
  }// switch
 }// foreach

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

As shown in this pseudo-code, the protocol is very simple which is in contrast with previous appraches to synchronization of query operations in inverted files [8], see next section.

4 Experimental Evaluation

We performed experiments using a 2GB sample of the Chilean Web and a query log from www.todocl.cl. This gave us a realistic setting both on the set of terms that compose the text collection and the type of terms that typically are part of user queries. Transactions were generated at random by taking terms from the

query log. We started with 60% of the text collection and increased it by including the remaining 40% divided in documents as part of the write transactions generated at random. At the start of this almost real-life system every processor "knows" its set of queries (i.e., we exclude the effect of query traffic and broker operations). We performed our experiments on a high-performance cluster with 16 processors (Pentium IV, 1GB main memory).

We worked with rather large query batches (64 ... 1024) to simulate high query traffic scenarios. For the final ranking of answers to every query, we considered only a small fraction of the involved inverted list since we considered only the top 100 of those answers (Persin's strategy was applied to organize the inverted lists and filtering [10] and parallel priority queue technique proposed in [1]).

We compared our method against the two-phases and time warp protocols. In the **two-phases protocol**, transactions first request locks on the subset of index-terms that are part of a read-only query or the relevant terms found in the document being added to the collection. After all of the locks have been granted, the associated operations are allowed to take place and the locks are released. Deadlocks are avoided by asking locks in lexicographic term order. A direct realization of this protocol on a BSP machine is to request all the required locks in the same superstep, and then wait during one or more supersteps to receive all the pending lock authorisations. If a required lock is being held by another transaction, it is necessary to wait until this transaction releases the lock. Read locks are answered with the data itself to be read. Write locks are requested by sending into the same message the new data to be written. That is, no additional message traffic is necessary for effecting the R/W operations. All messages releasing the granted locks can be sent in the same superstep.

The **Time Warp protocol** is based on the optimistic assumption that no events will probably get into conflict with each other, and if that situation happens to occur a correction procedure is executed by moving backwards the computation, correcting the error, and then moving it forward again but this time taking into consideration the cause of the trouble. The same strategy can be applied to the parallel processing of transactions. That is, they are allowed to perform their R/W operations at will, but each time a data item is read or written a consistency check is executed to detect if it necessary to do a roll-back of all causally related transactions or let them continue forward. A timestamp is assigned to each transaction. This is an increasing integer number. All operations of a given transaction receive the transaction timestamp and the protocol is in charge of ensuring that all operations on records are done in increasing timestamp order. Whenever an operation breaks this rule, all already-executed operations on the involved record that have timestamps greater than the new one are undone and re-executed on the record to obtain the right sequence. Only read operations are allowed to be done in different timestamp order as long as no write operation should have been executed in between.

Every time an operation of a given transaction is undone, it is also necessary to undone all subsequent operations of the same transaction which have already been executed on other records. Note that these records can be located in other

processors. Then these operations must be re-executed again since each one in the sequence can depend on the previous one. All this process is call a *roll-back*. Efficiency depends heavily on the amount of roll-backs performed during the computation. Transactions are committed when all their operations become ones with timestamps less than the smallest timestamp of any operation waiting to be executed (this considering all processors). In [6] we propose an efficient BSP algorithm for Time Warp on BSP Computers which can be easily adapted to support this strategy.

In figure 1.a we present speed-ups values for different number of processors obtained in a 16-processors PC cluster running the BSPpub library. The speed-up values where obtained taking the running time achieved by a efficient sequential realization of inverted files, and dividing it by the running time achieved with the method proposed in this paper (CON), the two-phases lock protocol (LOCKS) and the time warp (TW) protocol. The server was assumed to receive batches of 1024 queries. It is observed that the proposed concurrency control method achieves better performance than the other alternative algorithms.

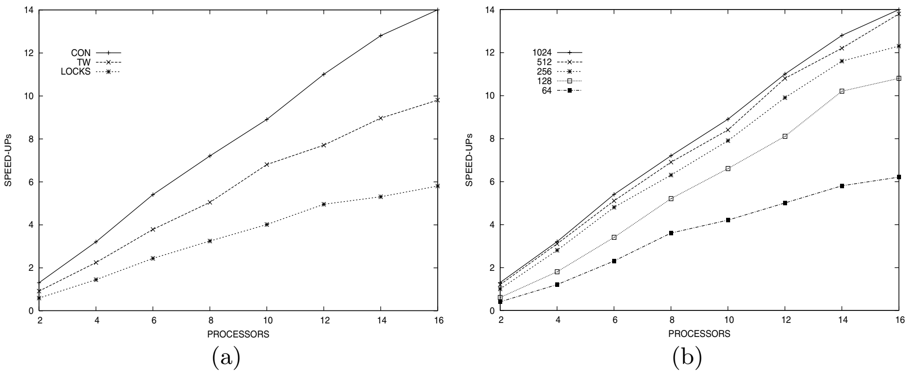

Fig. 1. (**a**) Speed-ups for different number of processors (**b**) Speed-ups for different number of queries per batch

To see the effect of batch size on the speed-up we obtained speed-up values for different number of queries per batch. The results are shown in figure 1.b where it can be seen that the proposed method is quite robust to small sized batched. In practice we should expect large size batches as parallelism is justified in situations of heavy query traffic on the server.

5 Conclusions

We have presented a very efficient method for synchronizing concurrent accesses to distributed text databases which are indexed by a parallel realization of inverted files. The method outperforms previous solutions to this problem which are based on the conservative two-phases locks and the optimistic Time Warp

approaches. Both have involved implementations and are less efficient than the method proposed in this paper.

The implementation is extremely simple because it only requieres processing the new arriving R/W queries by increasing timestamps. Correctness comes from the following points: (a) The broker send for processing batches of R/W queries, (b) All R/W operations of a given batch are processed during the same superstep (this is possible because the global index approach is employed to implement the inverted file), and (c) At the end of every supersteps are barrier synchronized and new queries only arrive by the begining of the next superstep. Some people could argue that the cost of globally synchronizing the processors after every batch can be too high. We claim that this is not the case because fairly small-sized batches can easily amortize the cost of barrier synchronization of processors. Our empirical results confirm this claim.

References

1. 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.
2. R. Baeza and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley., 1999.
3. 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.
4. D.R. Jefferson. Virtual time. *ACM Trans. Prog. Lang. and Syst.*, 7(3):404–425, July 1985.
5. B.S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.
6. M. Marín. Time Warp on BSP Computers. In *12th European Simulation Multiconference*, June 1998.
7. 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.
8. M. Marín. Optimistic Concurrency Control for Inverted Files in Text Databases. In *IASTED International Conference on Databases and Applications (DBA2004)*. Acta Press, Feb. 2004.
9. 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.
10. 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.
11. 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.
12. L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.