On bulk-synchronous distributed-memory parallel processing of relational-database transactions

Mauricio Marín (contact author)
Departamento de Computación, Universidad de Magallanes
Casilla 113-D, Punta Arenas, CHILE
E-mail: mmarin@ona.fi.umag.cl

Sandra Casas
División de Informática
Universidad Nacional de la Patágonia Austral
Río Gallegos, Argentina

Abstract

This paper describes two parallel algorithms for the efficient processing of relational database transactions and presents a performance analysis of them. These algorithms are built upon the bulk-synchronous parallel model of computation. The well-defined structure of this model enabled us to evaluate their performance by using an implementation independent and yet empirical approach which includes the effects of synchronization, communication and computation. The analysis reveals that the algorithm which borrows ideas from optimistic parallel discrete event simulation achieves better performance than the classical approach for synchronizing concurrent transactions on a distributed memory system.

1 Introduction

Transactions in relational database systems are sets of related read/write (R/W) operations which are treated as an atomic entity in the sense that they are either entirely executed on the database, or "rolled-back" and then re-executed when a causality error is detected. Most obvious application of this concept is in computational systems that are prone to failure. However, a need for transactions arises also in cases in which it is necessary to synchronize database accesses coming from a number of concurrent transactions trying to execute their operations on it over the same period of time.

This paper is concerned with the synchronization of relational database transaction operations running on a parallel distributed-memory environment which supports an architecture independent and cost predictable model of computation. The database is assumed to be distributed on P processors, and transactions are allowed to read/write records from/on any processor. Each record is maintained in a unique processor (i.e., sharing-nothing database model) and there exists a mapping function to determine the associated target processor. The model of computation is the so-called *bulk-synchronous parallel* (BSP) model [7].

Our underlying conjecture is that by processing parallel transactions on a model like BSP, it is possible to achieve scaleable performance in heavy-load scenarios wherein thousands of transactions are serviced by a distributed memory/disk server. This conjecture is not a product of chance since it has already been shown that a related problem can be solved more efficiently than traditional models by BSP-like styles of computation. We refer to the synchronization of event occurrences in parallel discrete-event simulations [2]. In our case, read and writes are equivalent to simulation events that take place concurrently during a given period of time which must satisfy causality restrictions like serialization.

This paper compares an event-synchronization protocol used in both application domains with the two-phases lock protocol which is commonly used in relational database management systems. Our results show the better comparative performance of the former and its clear potential for achieving scaleable performance on architectures ranging from expensive super-computers to clusters of PCs. Another contribution of this paper is the specific BSP realization of the synchronization protocols we use in our experiments. To the best of our knowledge, this particular topic, namely transaction processing under the BSP model of computation, has received no attention in the literature and thereby our discussion on previous work and reference list is necessarily short. Related work on BSP and databases systems can be found in [3, 1, 6] and in the references there mentioned.

The remaining of this paper is organized as follows. In section 2 we present an efficient BSP realization of two synchronization algorithms for concurrent transactions. In section 3 we present a comparative analysis of the algorithms. Section 4 presents conclusions.

2 Models and protocols

In the BSP model of parallel computation both computation and communication take place in bulk before the next point of global synchronisation of processors. A BSP program is composed of a sequence of *supersteps*. During each superstep, the processors may only perform computations on data held in their local memories and/or send messages to other processors. These messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors. The model can be easily implemented in todays parallel architectures as facilities for performing message sending and barrier synchronisation are commonly available in them. Moreover, instead of general purpose communication libraries such as PVM or MPI, a special purpose BSP library can be employed to write efficient C++ parallel programs [5].

In the following we describe two synchronization protocols for concurrent transactions on BSP computers. In our setting, every processor of the BSP machine must execute R/W operations of a large number of transactions. We assume that each processor maintains a piece of the database records either in its main memory or in its local disk. Every transaction is created in a given processor so that all processors maintain about the same number of them. Targets work-loads are those which produce large number of transactions per processor during long periods of time. When a transaction must perform a read or write operation on a record located in another processor, it effects such task by sending messages to the target processor. Recall that messages take one superstep to reach their destinations.

In the **two-phases protocol**, transactions first request locks on the subset of records upon which they need to perform their R/W operations. After the locks on that records have been granted and all R/W operations performed on them, the acquired locks are released. Deadlocks are avoided by establishing a global order for the records and requesting them following the same order. Once a given lock has been granted, the transaction requests the following one in the global order, and so on. A direct realization of this protocol on a BSP machine is to use one superstep for sending a message to request a single lock, then wait one superstep for the remote processor to make all arrangements for granting the lock, and finally at the next superstep receive the grant notification, then proceed with the same procedure for the next lock, and so on. If the 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 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.

A more aggressive strategy would be to request all the required locks at the same superstep, and then wait during one or more supersteps to receive all the pending lock authorisations. However, this introduces deadlock problems since the above mentioned global order can be broken. A solution is to define priorities for the transactions, and use such values to determine which transaction should get the lock on a certain record at a given superstep. Priorities must be unique across all transactions which leads to the problem of getting unique and increasing integer values in a distributed environment. In addition, read operations have to be treated as exclusive ones, like writes, since at all times only the transactions with the best global priorities should get the locks they request for. [At the time of writing this paper we have not been able to figure out a realization of this aggressive strategy which be significantly more efficient than the straightforward one described above. So we use the former one in our experiments. Similar strategy can be used in the protocol described below, but we exclude this possibility in order to compare both approaches under the same context.]

The **Time Warp protocol** is a popular event synchronization strategy for parallel discrete event simulation [2]. This asynchronous and distributed memory algorithm 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 record 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.

Our realization of this strategy for the BSP setting is as follows. Transactions are created in any superstep and each of them is composed of a set of R/W operations. 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. When an operation of a given transaction is undone, it is also necessary to undone all following 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 [4] we propose an efficient BSP algorithm for Time Warp on BSP Computers which can be easily adapted to support this strategy. This can be done as follows. Every processor keeps a priority queue where priorities values are the operations timestamps. No operations from different transactions can have the same priority value and operations belonging to the same transaction can be given a second priority value in accordance with their relative ordering within the transaction. To reduce the number of roll-backs we use the local priority queues in a way that emulates a global priority queue. Note that if all operations were taken from this global priority queue, no roll-back would ever occur. In each superstep, each processor executes the n operations with the highest priorities (least numerical values). Every time an error is detected, all the operations are returned to the local priority queues. Errors occur because messages containing read/write operations take one superstep to reach their destinations, and this causes that in a given superstep the processors do not contain the correct first n priorities. Some of them are due to arrive in the following supersteps. However, already-executed and new operations are treated identically for the purpose of determining when the upper limit n is reached. This is precisely what tends to emulate the global queue since a processor working at a high error rate will be kept restrain from advancing too far in the processing of operations with low priorities. The value of n can be calculated automatically during execution as we propose in [4].

In the next section we empirically compare the two strategies above described by using a synthetic but very demanding workload. Our performance metrics are independent of the particular implementation of the protocols since we measure the amount of computation, communication and synchronization required to complete a given instance of the work-load.

The **work-load model** is as follows. On a P processors BSP machine we initially create T transactions per processor. Every time a transaction finishes the execution of all its operations, a new one is created so that the total of $T \cdot P$ transactions is kept constant throughout the whole experiment. In each work-load instance, namely different values for P and T and other parameters described below, the experiment ends after a very large number of superteps is executed. All performance metrics are normalized to the amount of something done per superstep. When a new transaction is created, it is given a random number N of operations to be executed. Read and write operations have certain probability of being chosen and the records upon which they are to be executed are selected uniformly at random among all records of the database. In particular, these records are evenly distributed across all the processors. Each work-load instance is assumed to contain R records per processor. Each instance of a experiment is executed several times, each with a different random number seed.

Our performance metrics are the following: (**i**) Average number of transactions processed per superstep S considering the effect of all the processors; (**ii**) Average load balance in computation

C measured as the average maximum across supersteps of the amount of computation registered in each processor divided by the observed average. We count as 1 the execution of a single R/W operation and also the computations associated with the housekeeping of records (i.e., we assume a database distributed in the processor's main memory); (**iii**) Average load balance in communication M which is similar to C but counting messages sent at the end of each superstep. In other words, we measure performance in terms of the costs of synchronization, communication and computation. In this case, an algorithm is more efficient than another if it has less activity in synchronization (supersteps), less communication, and computation is well balanced across processors (the same is valid for communication).

3 Performance analysis

The best performance is achieved in a situation in which no R/W conflicts ever take place so that all transactions are able to execute their operations as fast as possible with no waiting supersteps. For a remote read operation it is necessary to wait one superstep for the data to come over. Remote write operations require no waiting superstep. In addition, in this best case scenario, transactions can send all their read requests at the very first superstep in which they are created and start their execution. Thus each transaction requires at least three supersteps to complete its execution, where in the last one all the operations are executed in bulk and write messages are sent to remote processors (this assumes that at least one read is in every transaction). Then the average number of transactions processed per superstep S is given by $T \cdot P/3$. Load balance in computation and communication tends to the optimum $C = 1$ and $M = 1$ as T increases since this is equivalent to the average occupancy of the "balls thrown into a set of baskets" problem.

Note that synchronization strategies like the optimistic one have the potential for achieving the above optimum performance as the number of records per processor R is scaled up whilst the number of transactions per processor T is kept fixed and large, since this reduces the probability of two given transactions willing to access the same record during the same superstep. The aggressive two-phases protocol with priorities can also present the same behaviour. However, taking a more realistic approach we are going to assume that it is not possible to send all read messages in the same superstep in which the transaction is created. That is, we assume that reads must be executed one after the another so that, for example, if 3 consecutive reads are required, each in a different processor (our case with high probability $1 - 1/P$), the machine will use at least 6 supersteps to complete the reads. Of course, we are not interested in high performance at the individual transaction level but at the overall level since our goal is to efficiently process a large number of transactions during a long period of time wherein transactions are created at any point (but uniformly distributed) of this period. That is, transactions are created in any superstep of the computation. Note that our workload achieves this because it maintains a constant number of transactions per processor and the creation of new transactions is distributed along the time because the number of operations per transaction is a random variable. A new transaction is created as soon as an old one finishes the execution of all its operations.

The standard realization of the two-phases lock protocol will in the best case require a sequence of $2N$ supersteps to complete all lock requests and then proceed to execute all associated operations in bulk. Thus, on the average, the total number of transactions per superstep in this protocol is

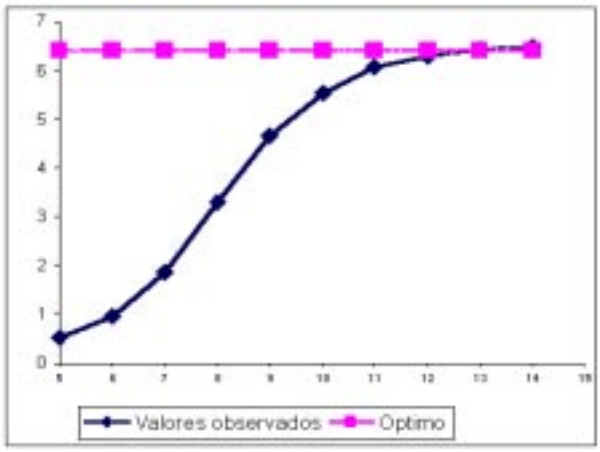

Figure 1: Number of processed transactions per superstep. The x-axis is in log_2 scale.

$T \cdot P/(2\,\mathrm{E}[N])$. Since N is a random number, the bulk of operations will tend to ocurr in different supersteps which reduces the total amount operations per superstep that are executed by a factor of $1/(2\,\mathrm{E}[N])$. Less operations per superstep, leads to poorer load balance in computation and communication.

To observe how the lock protocol achieves the above optima we performed experiments with the work-load using $E[N] = 10$, $P = 16$, $T = 8$ and $R = 32, 64, 128, 256, ..., 16384$. The results are shown in figure 1. This figure shows that it is necessary a large number R of database records per processors to achieve the optimal number of transactions per superstep. For smaller values of R contention for the same records becomes intense. Load balance in computation and communication is not good since typical C and M values were in the range 4 to 6. However, this problem has an easy solution. As we know what the optimal S should be, we can estimate the average number of operations executed in each superstep. (These values can be measured empirically for any system during execution as proposed in [4]). A factor of this estimate can be used as an upper limit to the number of operations executed per processor in each superstep. If in a given supersteps this maximum is reached and there are more operations to be executed, these are processed in the following superstep and so on. Using a factor 1/2 we observed that load balance improved significantly at the cost of increasing about three times the total number of supersteps. This reduces three times the values shown in figure 1.

Though the strategy used by the Time Warp algorithm is quite different to the one used by the lock protocol, the optimal value for the expected number of transactions per superstep S is the same value $T \cdot P/(2\,\mathrm{E}[N])$. In Time Warp a transaction sends messages containing R/W operations where reads take, in the best case, two supersteps to complete. If no roll-back takes place, then the optimal S is achieved. We performed similar experiments to that of figure 1. The results for S are very similar to those of figure 1 for large R. However, in the region of small R the Time Warp protocol achieves quicker near optimal S values. But much more important is the fact that load balance is near optimal (on average $C = 1.2$ and $M = 1.6$). Similar load balance was achieved with the lock protocol but S had to be decreased three times.

4 Conclusions

We have presented and analysed two bulk-synchronous parallel algorithms for synchronizing concurrent relational database transactions running on a distributed memory environment. Overall, we have observed that the BSP Time Warp algorithm we propose in this paper outperforms an efficient BSP realization of the traditional two-phases lock protocol by a wide margin. BSP Time Warp requires a near optimal number of supersteps to process a large number of transactions whilst at the same time it also achieves near optimal load balance.

Note that the amount of computation performed by both algorithms is similar since we have observed that the number of roll-backs in Time Warp is not significant whereas the overheads associated with locks administration are avoided completely. Thus what matters in the comparison is balance in computation and communication and the amount of synchronization of processors.

We are at present exploring ways of optimizing the proposed algorithms and working on the efficient BSP realization of other well-known protocols for synchronizing concurrent transactions in relational database systems.

References

[1] M. Arriagada, J. Canuman, D. Laguia, and M. Marin. "Bases de datos relacionales paralelas sobre BSPlib". In *2000 Workshop Chileno en Sistemas Distribuidos y Paralelismo*, Nov. 2000. Santiago, Chile.

[2] R.M. Fujimoto. "Parallel discrete event simulation". *Comm. ACM*, 33(10):30–53, Oct. 1990.

[3] J.M.D. Hill, S. Jarvis, C. Siniolakis, and V.P. Vasilev. "Analysing an SQL application with a BSPlib call-graph profiling tool". In *Euro-Par'98*, 1998. Lecture Notes in Computer Science.

[4] M. Marín. "Time Warp On BSP Computers". In *12th SCS European Simulation Multiconference*, June 1998.

[5] BSP Worldwide Standard. http://www.bsp-worldwide.org/.

[6] K.R. Sujithan. "Towards a scalable parallel object database — The bulk-synchronous parallel approach". Technical Report PRG-TR-17-96, Computing Laboratory, Oxford University, 1996.

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