International Journal of Parallel Programming, Vol. 17, No. 1, 1988

Parallel Evaluation of the Transitive Closure of a Database Relation[1]

Patrick Valduriez and Setrag Khoshafian

Received February 1988; Revised July 1988

Parallelism is a promising approach to high performance data management. In a highly parallel data server with declustered data placement, an important issue is to exploit parallelism in processing complex queries such as recursive queries. In this paper, we consider the transitive closure of a database relation as a paradigm to study parallel recursive query processing. And we propose two new parallel algorithms for evaluating the transitive closure of a relation in a parallel data server. Performance comparisons based on an analytical model indicate the superior response time of the parallel algorithms over their centralized version. With one hundred nodes, performance gain is between one and two orders of magnitude. One parallel algorithm provides superior response time while the other exhibits better response time/total time trade-off.

KEY WORDS: Database relation; parallelism; transitive closure; algorithms; performance.

1. INTRODUCTION

One promising approach to high performance data processing is to group the database functions into a dedicated computer, called *data server*. With its single focus on database management, a data server can provide a better price/performance ratio than a database system implemented on a general purpose computer. The performance of database management is essentially hurt by the I/O bottleneck[1] which stems from high disk access time compared to low main memory access time. The data server approach makes the use of architectural solutions to the I/O bottleneck possible. The main

[1] This work had been done within the Advanced Computer Architecture Program, Microelectronics and Computer Technology Corporation, Austin, Texas. The current affiliation of Setrag Khoshafian is Ashton-Tate, Walnut Creek, California.

solution is to increase the I/O bandwidth through parallelism.[2,3] Instead of having the entire database residing on a few high capacity disk units, many smaller disks should be employed so that disk accesses can be done in parallel. Thus, disk access time can be divided by the number of disk units.

Parallelizing the I/O bandwidth can be achieved using the recent multiprocessor architectures based on the *shared nothing* paradigm.[4] In a shared nothing architecture, each processor has exclusive (nonshared) access to one or more memory modules and one or more disk units. Shared nothing architectures are more scalable than shared memory architectures since the number of nodes need not be limited. In particular, highly parallel shared nothing architectures are now viable.[5,6]

Two important and related issues that face the design of a highly parallel shared nothing data server are data placement and query processing. A good solution to data placement is *declustering*[7] which consists of horizontally fragmenting the relations across many nodes to favor the parallel execution of database operations. Query processing must exploit the potential parallelism available with declustered data placement in order to achieve good response time/throughput trade-off. Parallel algorithms for optimizing relational algebra operations, particularly join,[8–10] and relational queries[11] are now well understood. However, little attention has been paid to parallel algorithms for recursive query processing.[12]

In this paper, following recent proposals,[13–17] we consider the transitive closure as a paradigm to study parallel recursive query processing. We observe that virtually no attempt has been made to implement the transitive closure of a database relation in a shared nothing parallel architecture. The few parallel transitive closure algorithms proposed as a method of finding the connected components of an undirected graph typically assume a shared memory architecture model.[18] A notable exception is some preliminary work[19] which attempted to implement our logarithmic algorithm[13] in GAMMA.[6] In this paper, we present two new parallel algorithms for evaluating the transitive closure of a declustered relation in a shared nothing data server. One of them has been thoroughly described and analyzed in another paper,[20] and will be only briefly mentioned in this paper for the sake of comparison. Performance comparisons based on an analytical model indicate the much better response time of the parallel algorithms over their centralized version. In particular, one parallel algorithm provides superor response time while the other exhibits better response time/total time trade-off.

The paper is organized as follows. Section 2 describes the assumptions regarding the parallel operational model. Section 3 presents generic algorithms for computing the transitive closure of a centralized relation. They are used subsequently in a parallel version. A uniprocessor algorithm and two parallel algorithms for computing the transitive closure of a declustered relation are presented in Section 4 and analyzed in Section 5. Section 6 provides the performance comparisons.

2. PARALLEL OPERATIONAL MODEL

Many parameters regarding the processing environment and the database will affect the performance of various transitive closure algorithms. In order to concentrate on the critical aspects of the algorithms (parallelism) and on their comparison, we make a number of assumptions. The implications of relaxing these assumptions will be studied in the future. The assumptions concern the data server, the operand relation, the algorithms and communication.

2.1. The Parallel Data Server

A generic shared nothing parallel architecture is illustrated in Figure 1. Each node includes one or more processors, a local main memory (RAM) and a disk unit on which resides a local database. Diskless nodes could also be used to interface the data server with other machines or to process intermediate relations in parallel. The term "shared nothing" refers to the fact that there is no sharing of main memory by the nodes. The only shared resource is the network, with which the nodes can exchange messages. Underlying the data server is a distributed operating system which, among other things, provides low-level support for task management and communication. Examples of shared nothing architectures are the Teradata DBC/1012[5] and GAMMA.[6]

The database consists of relations which are *declustered* across DBM nodes. Declustering[7] is a placement strategy which horizontally partitions

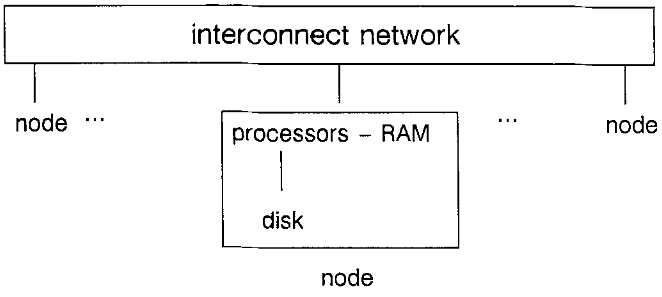

Fig. 1. Shared nothing parallel data server.

and distributes each relation across a number of nodes. This number of nodes is a function of the size and access frequency of the relation.(21) The number of repositories over which a relation is distributed is called the *degree of declustering*. There are several ways to distribute tuples across multiple nodes. The simplest approach is to place tuples in a round robin fashion among multiple nodes. Although more complex approaches could provide opportunities for improving the performance of database operations such as transitive closure, we will assume round robin placement for simplicity. The main feature of declustered data placement, that we are interested in, is that accessing all tuples of a relation is inherently *parallel*, i.e., all nodes storing a subset of the relation can be accessed "simultaneously".(11)

2.2. Operand Relation

We denote by R the relation to which transitive closure is applied. Relation R is horizontally partitioned (declustered) across d nodes in a round robin fashion. For the sake of simplicity and generality, we assume that each subset of relation R stored at one node has no particular access method other than sequential scan. Note that the performance of transitive closure would be improved by the addition of some particular data structures such as join indices.(22)

2.3. Parallel Join and Union Algorithms

Our implementation of transitive closure will require the use of join and union operations. We will use hash-based algorithms for performing both joins and unions efficiently.(23) Hash-based algorithms have been primarily designed to speed up the join operation.(24) The basic idea is to partition each of the relations being joined, say R and S, into mutually exclusive sets $R_0, R_1, \ldots, R_n$ and $S_0, S_1, \ldots, S_n$ such that

$$R \bowtie S = \bigcup_{i=0}^{n} R_i \bowtie S_i$$

The partitioning is based on a hash function applied to the join attribute. The individual joins $R_i \bowtie S_i$ can be done simply with a nested loop procedure where for each tuple in R_i, S_i is probed. If there is a match, then a result tuple is produced. This algorithm can be easily extended to operate in a multiprocessor environment where each join $R_i \bowtie S_i$ is done in parallel by a separate processor.(10)

The union operation can also be implemented using hashing. We use the following algorithm for performing the union of relations R and S. The partitioning phase is the same as above when the hash function is applied to the key attribute(s). The individual unions $R_i \cup S_i$ are also done with a nested loop where for each tuple r of R_i, S_i is probed. If r is not in S_i, then it is inserted in the relation S_i which will therefore contain $R_i \cup S_i$.

2.4. Communication

The parallel execution of database operations requires data to be transferred between nodes. We assume two basic communication primitives for transferring data: send and receive.

send (message, node(s))

transfers the message to the destination nodes. To avoid implementation details, we assume that the kernel of the node is intelligent enough to examine the message and give it to the receiving task. When the message contains a relation, the destination nodes may be specified by a hash function applied to some attributes. In this case, the tuples are first inserted into different buckets based on the result of the hash function and each bucket is sent to a different node.

$R :=$ receive

gets the content of the message in R for the receiving task.

3. TRANSITIVE CLOSURE OF A CENTRALIZED DATABASE RELATION

This section provides a number of common definitions, and recalls two basic transitive closure algorithms.

3.1. Definitions

We assume that transitive closure operates on a binary relation R having attributes A and B defined on the same domain $\mathfrak{D}$. Relation R can be viewed as a set of edges in a directed graph, wherein a node is an element of $\mathfrak{D}$ and an edge from node a to node b indicates the tuple (a, b) of R. For simplicity, we assume that R is acyclic. We call the depth of R, noted p, the length as measured by the number of edges, of the longest path in the graph. p is an important parameter affecting the cost of the transitive closure. The transitive closure of R is equivalent to the transitive closure of the corresponding graph, i.e., the tuple (x, y) is in R^+ iff there is a path of length >0 from x to y. Let $\bullet$ denote the composition of two binary relations (or binary composition) where all attributes belong to the same domain:

$$R \bullet S = \{(a, c) \mid \exists b \ (a, b) \in R \text{ and } (b, c) \in S\}$$

and let R^i be the ith power of relation R, i.e., $R^1 = R$ and $R^i = R^{i-1} \bullet R$. Then, R^+ is:

$$R^+ = \bigcup_{i>0} R^i$$

$R \bullet S$ can be implemented by a join with projection as:

$$R \bullet S = \pi 1, 4\ (R.2 \bowtie S.1)$$

Note that the binary composition is not commutative, i.e., $R \bullet S \neq S \bullet R$.

3.2. Iterative Transitive Closure

Several uniprocessor algorithms which compute the transitive closure of a database relation have been proposed.[13-15] In Ref. 13, we presented two basic transitive closure algorithms; an iterative algorithm whose complexity is $O(p)$ and a logarithmic algorithm whose complexity is $O(\log p)$. The superiority of the logarithmic algorithm has been analyzed in Ref. 13 and then confirmed in Refs. 14 and 15. Both iterative and logarithmic algorithms can operate on cyclic relations.

In this paper, we will investigate parallel versions of the iterative algorithm for three reasons. First, it is the simplest algorithm. Second, we are mostly interested in analyzing the performance gained with parallelism. Third, we believe that the extension to the iterative algorithm for operating in a parallel environment can be applied to the logarithmic algorithm and other algorithms as well. The iterative algorithm, noted ITC, working on an acyclic relation can be expressed using relational algebra extended with assignment and iteration, as shown in Fig. 2. The correctness of this algorithm is given in Ref. 14. As demonstrated in Ref. 13, this algorithm easily applies to cyclic relations by adding a difference operation before the union.

```
ITC (R: operand, T: result)
  T:= R;
  D:= R;          (* D will contain new tuples *)
  repeat
    $D := D \bullet R$ ;
    $T := T \cup D$ ;
  until $D = \phi$ ;
```

Fig. 2. Iterative transitive closure (ITC).

ITC computes the transitive closure of relation R in T using a differential relation D. We illustrate algorithm ITC by applying it to the following example relation R.

R	A	B
	1	8
	8	24
	24	30
	24	7
	30	36

We indicate the version number of a differential relation by superscript. Relation D^1 and T are initialized to R. At iteration i, the differential relation D^{i+1} is produced and unioned with T. Iterations 1, 2 and 3 produce the differential relations D^2, D^3 and D^4 respectively.

D^2	A	B
	1	24
	8	30
	8	7
	24	36

D^3	A	B
	1	30
	1	7
	8	36

D^4	A	B
	1	36

Iteration 4 produces no more tuples and the operation terminates with the result

$$T = R \cup D^2 \cup D^3 \cup D^4$$

3.3. Transitive Closure of Transitively Closed Relations

Transitive closure of transitively closed relations will be the basis for a parallel algorithm. Assume that a relation R is partitioned into R_1 and R_2. A simple way to compute R^+ in parallel is to first compute R_1^+ and R_2^+ and to complete the transitive closure of $R_1^+ \cup R_2^+$. The problem is to avoid redundant work when performing the transitive closure of two transitively closed relations. A naive way that consists of performing ITC $(R_1^+ \cup R_2^+, T)$ would also recalculate R_1^+ and R_2^+ which is useless. The problem with this naive approach is that the differential relation D is initialized with $R_1^+ \cup R_2^+$.

The following algorithm has been proposed in Ref. 20 to perform the transitive closure of two transitively closed relations without redundant work. The resulting algorithm, called TCCR, is shown in Fig. 3. The algorithm and its correctness are described in Ref. 20.

```
TCCR ($R_1$ , $R_2$ : operand, T: result)
flip := true
$D_1$ := $R_1 \bullet R_2$ ;
$D_2$ := $R_2 \bullet R_1$ ;
T := $R_1 \cup R_2 \cup D_1 \cup D_2$ ;
repeat
    if flip then $D_1 := D_1 \bullet R_1$ else $D_1 := D_1 \bullet R_2$ ;
    if flip then $D_2 := D_2 \bullet R_2$ else $D_2 := D_2 \bullet R_1$ ;
    T := $T \cup D_1 \cup D_2$ ;
    flip:= not flip ;
until $D_1 = \phi$ and $D_2 = \phi$
```

Fig. 3. Transitive closure of two transitively closed relations (TCCR).

Let an *alternating composition sequence* of R_1 and R_2 be a sequence of binary compositions of R_1 and R_2 such that $R_1 \bullet R_2$ or $R_2 \bullet R_1$ *never* occurs in the sequence. The algorithm TCCR computes the transitive closure of $R_1 \cup R_2$ by producing only alternating composition sequences. Since alternating composition sequences never contain redundant compositions such as $R_1 \bullet R_1$ or $R_2 \bullet R_2$, the algorithm TCCR does not perform redundant work.

4. TRANSITIVE CLOSURE OF A DECLUSTERED DATABASE RELATION

In this section we present three versions of the iterative algorithm to deal with a declustered relation. The first algorithm, called transitive closure with unique processor, applies the iterative algorithm at one node where the operand relation has been centralized. The second algorithm, called transitive closure with parallel operatons, iteratively applies the join and union operations in parallel. The third algorithm, called transitive closure with parallel programs, applies the transitive closure programs in parallel and iteratively integrates the transitively closed intermediate relations.

4.1. Uniprocessor Transitive Closure Algorithm

Given a relation R distributed across d nodes, the simplest way to perform its transitive closure is to resort to a centralized algorithm. Although this method is not really parallel, it will be useful for comparison with

```
TCUP (R:operand, T:result)
  (1) at each node i do
        send ($R_i$ , nodeP);        (* $R_1$ , $R_2$ , ..., $R_d$ are sent to nodeP *)
  (2) at nodeP do
        begin
        R := receive;
        ITC (R, T)                   (* computes T as transitive closure of R *)
        end
```

Fig. 4. Transitive closure with unique processor (TCUP).

parallel transitive closure algorithms. The transitive closure algorithm with unique processor (TCUP) is detailed in Fig. 4. It is assumed to be controlled by some coordinating node. It consists of two phases. In the first phase, each node 1, 2,..., d storing a subset R_i of R sends R_i to a pre-determined node P. The sends can be done in parallel, as indicated by the "at each" statement. The second phase is done by a single node P which receives all subsets of R and performs the transitive closure of R using the iterative algorithm described in Section 3.2.

4.2. Transitive Closure with Parallel Operations

The transitive closure algorithm with parallel operations (TCPO) is assumed to be controlled by some coordinator node chosen among the n nodes allocated to the operation. The basic idea is to execute the iterative algorithm where each join operation (necessary for the binary composition) is performed in parallel by a hash-based algorithm.[10] Parallel join operations are achieved by partitioning the operands between n disjoint sets based on a hash function on some attribute. Partitioning is done with the following procedure:

```
partition ($R$, $h(A)$);
  at each node storing $R_i$ of $R$ do
    send ($R_i$, node $h(A)$);
```

where R_i is first hashed into, say, n buckets and each bucket is sent to a different node. After the two operand relations have been partitioned, the operation is achieved as n partial operations.

The algorithm TCPO (see Fig. 5) consists of two phases. First, relation R is partitioned on one attribute, say B, between n nodes (we assumed R is not partitioned on B). T is initialized to R and thus partitioned on B. Second, the transitive closure is applied to R as a loop of the following operations. D is partitioned on the join attribute (A) between n nodes and each D_i is joined with R_i on the predicate $D_i \cdot A = R_i \cdot B$. The result of the join (after removal of useless attributes) gives D_i which must

```
TCPO  (R:operand, T:result)
  (1) partition (R, h(B));                    (* generates and sends $R_1$, $R_2$, ..., $R_n$ *)
      at each node i (i=1, ..., n) do         (* initialize each node i *)
          begin
          $T_i$ := $R_i$ := receive;
          $D_i$ := $R_i$ ;
          end
  (2) repeat
      (2.1) partition (D, h(A));              (* generates and sends $D_1$, $D_2$, ..., $D_n$ *)
      (2.2) at each node i (i=1, ..., n) do     (* compute local D and T *)
              begin
              $D_i$ := receive;
              $D_i$ := $R_i$ • $D_i$ ;
              $T_i$ := $T_i$ U $D_i$ ;
              end
      until AND$_{i=1}^{n}$ ($D_i$ = $\phi$)
```

Fig. 5. Transitive closure with parallel operations (TCPO).

be unioned with T. However, since T is partitioned on B (because it has been initialized with R), D_i should first be partitioned on B before being unioned with T. However, partitioning may incur a substantial communication cost. One alternative is to replace the global union by local unions, in which case result tuples at different nodes may be duplicated. The choice between local versus global union involves estimation of the cost/benefit of duplicate elimination. Although it is an important issue, it is not addressed here. For simplicity, the algorithm in Fig. 5 applies local unions. The loop terminates when no new tuples are generated. The coordinator receives the Boolean value $(D_i \neq \phi)$ from all nodes and determines the end of the loop when all Boolean values are true. The result of the operation is distributed across n nodes.

To illustrate TCPO, we apply it to the example relation R introduced in Section 2.2 assuming $n=2$ and the hash function is $h(\mathrm{A})=1$ if $A<20$ and $h(\mathrm{A})=2$ if $A \geqslant 20$. We describe the algorithm by giving for each step (2.1) and (2.2) of phase 2 and each version, the value of the relation D (the value of T can be easily deduced). Superscripts indicate version numbers and subscripts indicate node numbers. For instance D_2^1 is relation D produced at iteration 1 at node 2.

Phase 1	R_1	A	B	R_2	A	B
		1	8		8	24
		24	7		24	30
					30	36

	$D_1^1 := R_1$			$D_2^1 := R_2$			
Step 2.1	D_1^2	A	B	D_2^2	A	B	iteration 1
		1	8		24	30	
		8	24		24	7	
					30	36	
Step 2.2	D_1^2	A	B	D_2^2	A	B	
		1	24		8	30	
					8	7	
					24	36	
Step 2.1	D_1^3	A	B	D_2^3	A	B	iteration 2
		8	7		24	36	
		1	24				
		8	30				
Step 2.2	D_1^3	A	B	D_2^3	A	B	
		1	7		8	36	
		1	30				
Step 2.1	D_1^4	A	B	$D_2^4 = \phi$			iteration 3
		1	7				
		1	30				
		8	36				
Step 2.2	D_1^4	A	B	$D_2^4 = \phi$			
		1	36				

4.3. Transitive Closure with Parallel Programs

The transitive closure algorithm with parallel programs (TCPP)[20] computes the operation as much as possible where the data is. The algorithm proceeds in several passes. The algorithm ITC is used in the first pass and the algorithm TCCR is used in subsequent passes. Recall that the operand relation is distributed over d nodes. For simplicity, we assume that d is a power of number 2. The first pass computes in parallel d partial transitive closures, each using the algorithm ITC. The partial results obtained at the end of the first pass are transitively closed relations. In order to complete the transitive closure using the algorithm TCCR, we divide it into several passes using a two-way merge type operation. Similar to the parallel binary sort-merge algorithm,[8] the d nodes are arranged as a binary tree. The second pass of the algorithm TCPP proceeds as follows. Every other node that participated in the first pass, say nodes 1, 3, 5,..., $d-1$, sends its result to its neighbor immediately to the right, i.e., nodes 2, 4, 6,..., d. Every receiving node (there are $d/2$ such nodes) applies the algorithm TCCR on the relation received and the relation it produced, and therefore generates another transitively closed relation. At pass i, $d/2^{i-1}$ nodes that produced a transitively closed relation at pass $(i-1)$ send their result to their neighbors immediately to the right which apply the algorithm TCCR to their input. The algorithm terminates when a single node, that received the relation computed by its unique neighbor, performs the last execution of TCCR. The number of passes where TCCR is applied in parallel is $[\log_2 d]$ where $[a]$ denotes the smallest integer greater or equal to a. Therefore the total number of passes, including the first pass in which ITC is applied by d nodes, is $[\log_2 d]+1$. The algorithm is described in Fig. 6.

5. PERFORMANCE ANALYSIS

In this section, we analyze the performance of the transitive closure algorithms presented in Sections 4 and 5. We first define the performance measure in terms of response time and total time, and the analysis parameters that we assume. Then, we analyze the three transitive closure

```
TCPP   (R:operand, T:result)
 (1)  at each node i (i=1, ..., d) do                  (* first pass *)
        begin
        ITC (R_i , T_i );
        if i mod 2 ≠ 0 then  send (T_i , node (i+1));
        end;

 (2)  for j := 1 to [log_2 d]  do                      (* other passes *)
        at each node i (i=2^(j-1), 2*2^(j-1), 3*2^(j-1), ..., d)  do
          begin
          S_i := receive;                              (* receive from predecessor *)
          TCCR (R_i , S_i , T_i );
          if i mod 2^j ≠ 0 then  send (T_i , node (i+2^(j-1) ));
          end;
```

Fig. 6. Transitive closure with parallel programs (TCPO).

algorithms with uniprocessor (TCUP), parallel operations (TCPO) and parallel programs (TCPP).

5.1. Performance Measure

There are several ways of measuring the performance of an algorithm in a parallel environment. In this paper, we will consider the *total time* and the *response time* of each algorithm. The total time is the sum of all component times (IO, CPU and communication time) and therefore gives a fair estimation of the use of machine resources. The response time is the time elapsed from the initiation to the completion of the algorithm. Parallel algorithms are more able to minimize response time but generally at the expense of total time. Therefore, both measures are complementary to understand the performance trade-offs of parallel algorithms. Total time, denoted by TT, and response time, denoted by RT, will be expressed in terms of local processing time and communication time.

Local processing time is incurred by reading and comparing the operand tuples and by producing new tuples using join and union. Both join and union operations may be efficiently implemented through hashing with a complexity almost linear in the size of the operands.(10,23) To simplify the analysis and especially to concentrate on the effects of parallelism, we assume that the time to produce a new tuple is constant. This time typically incorporates a fraction of disk access time and CPU time (to hash, compare and move tuples). Therefore, details about join and union algorithms need not be given. If a large number of new tuples is generated by transitive closure, then this assumption is quite good. It will be the case in our performance comparisons. In experimenting a more complex analytical model in which IO and CPU times were detailed, we found results very similar to these of Section 7.

5.2. Analysis Parameters

The following notation will be used to evaluate the algorithms:

$\|R\|$	number of tuples in relation R
p	depth of the graph corresponding to relation R
$\|DT\|$	number of new tuples produced by transitive closure
d	degree of declustering of relation R
n	number of nodes allocated for TCPO
newtup	time to produce a new tuple (includes fraction of IO time and CPU time)
K	number of tuples per packet
msg	time to send a message (includes send and receive time)
trf	time to transfer a packet

The parameter newtup is measured in number of CPU instructions. *trf* is the constant time required to transfer a data packet from one node to another. Messages are typically of variable size, i.e., of multiple packets. The communication time incurred in sending and routing an *m* packet message from one node to another is $(msg + m * trf)$. We assume that there is always enough available buffer space for holding *m* packets. The communication time necessary to move *t* tuples to a given node is therefore

$$TRF(t) = msg + \frac{t}{K} * trf$$

Traditional evaluation and comparison of parallel algorithms for database operations assume uniform distribution of work among the participating nodes.(8,9) The uniformity assumption is optimistic for evaluating response time and favors the parallel algorithms against their centralized version. However, the assumption of nonuniform distribution of work in evaluating response times would make the analysis too complex and intractable. Analyzing transitive closure in a centralized context is already complex enough.(13) Therefore, we will assume that work is equally distributed among the nodes participating in the execution of the parallel transitive closure.

5.3. Analysis of Algorithm TCUP

Algorithm TCUP has two phases. The first phase sends *R*, distributed across *d* nodes, to a single node (one of the *d* nodes). The second phase performs the transitive closure locally. The total time to produce new tuples is the same for the three algorithms

$$|DT| * \text{newtup}$$

The communication time of TCUP is the time to send $(d-1)$ pieces of relation *R* to one of the *d* nodes on which *R* resides. This time is simply

$$(d-1) * \text{TRF}\left(\frac{|R|}{d}\right)$$

The total time of TCUP is therefore

$$\text{TT(TCUP)} = (d-1) * \text{TRF}\left(\frac{|R|}{d}\right) + |DT| * \text{newtup}$$

The response time of TCUP is simply the sum of the time to transfer *R* to the result node and the time to compute the transitive closure. Since there is a single receiver node for all pieces of R that are sent, the transfer of R is essentially sequential. Thus, we can approximate the response time of TCUP as

$$RT\,(\mathrm{TCUP}) = TT\,(\mathrm{TCUP})$$

5.4. Analysis of Algorithm TCPO

Algorithm TCPO has two phases. The first phase partitions R, distributed across d nodes, onto n nodes. Each of the d nodes holds $(|R|/d)$ tuples of R. Since it must be hashed both on attribute A and attribute B, R must be partitioned twice. The second phase performs the transitive closure in parallel by iteratively performing a local composition and a local union, and partitioning relation D (that contains the new tuples) onto n nodes. We assume that each of the p passes of TCPO uniformly produces the same number of tuples. Therefore, we have at each pass i

$$|D^i| = \left\lceil \frac{|DT|}{p} \right\rceil$$

Parallel execution is obtained mainly by distributing operand tuples across n nodes based on a hash function. Let relation R be partitioned across n nodes and let $M(R, n, m)$ denote the number of messages necessary to send pieces of a sub-relation R_k (with $k = 1, n$) to m nodes. We make the pessimistic assumption that this number is the maximum number of potential receiver nodes, i.e.,

$$M(R, n, m) = \min\left(\frac{|R|}{n}, m\right)$$

To fairly compare with the two other algorithms which produce the result at a single node, we include the time to transfer the final result distributed across n nodes to a single node. The total time of TCPO is therefore

TT (TCPO) =
(* *each of the d nodes sends twice all the data it holds to n nodes except itself* *)

$$2 * d * (M(R, d, n) - 1) * TRF\left(\frac{|R|}{d * M(R, d, n)}\right)$$

(* *iteratively partition D during* $(p-1)$ *passes* *)

$$+\sum_{i=2}^{p}\sum_{k=1}^{n}(M(D^i,n,n)-1)*TRF\left(\frac{|D^i|}{n*M(D^i,n,n)}\right)$$

$$+(n-1)*\mathrm{TRF}\left(\frac{|R|+|DT|}{n}\right)$$

(* *transfer final result to one node* *)

$$+|DT|*\mathrm{newtup}$$ (* *local processing time* *)

The response time of TCPO is

$RT(\mathrm{TCPO})=$

(* *one node sends twice all the data it holds to M nodes* *)

$$2*(M(R,d,n)-1)*TRF\left(\frac{|R|}{d*M(R,d,n)}\right)$$

(* *iteratively partition D* *)

$$+\sum_{i=2}^{p}(M(D^i,n,n)-1)*TRF\left(\frac{|D^i|}{n*M(D^i,n,n)}\right)$$

$$+TRF\left(\frac{|R|+|DT|}{n}\right)$$ (* *transfer final result* *)

$$+\frac{|DT|}{n}*\mathrm{newtup}$$ (* *local processing time* *)

5.5. Analysis of Algorithm TCPP

The analysis of TCPP(20) provides the following formulas:

$$\mathrm{TT}\ (\mathrm{TCPP})=\sum_{i=1}^{\log_2 d}\frac{d}{2^i}*\mathrm{TRF}\left(\frac{|R|}{d}+\frac{2^{i-1}}{d}*|DL|\right)$$
$$+|\mathrm{DT}|*\mathrm{newtup}$$

$$\mathrm{RT}\ (\mathrm{TCPP})=\sum_{i=1}^{\log_2 d}\mathrm{TRF}\left(\frac{|R|}{d}+\frac{2^{i-1}}{d}*|\mathrm{DL}|\right)$$
$$+\sum_{i=1}^{\log_2 d}\frac{2^{i-1}}{d}*|\mathrm{DL}|*\mathrm{newtup}$$

6. PERFORMANCE COMPARISONS

This section presents performance comparisons of the proposed transitive closure algorithms using the previous cost formulas. The most sensitive parameters (d, $|DT|$ and MIPS/node) have been varied. The other analysis parameters are set as follows:

$\|R\|$	number of tuples is 1,000,000
p	depth of R is 32
K	number of tuples per packet is 20
msg	time to process a message is 5000 instructions
trf	time to transfer a packet is 100 microseconds (assuming a network speed of 10 MegaBytes per second)
newtup	time to produce a new tuple is 1000 instructions

Experiments with different parameters settings produced results similar to those described below. In particular, varying p did not affect the results. Note that, in all the graphs discussed below, the y-axis scale is logarithmic. To compare fairly TCPO and TCPP, we also assume $d = n$.

Figures 7–10 illustrate the performance of the algorithms versus number of processing nodes. The performance of parallel algorithms is strongly influenced by communication cost, which is a function of the number of new tuples produced and the time to transfer a packet. In order to push the limits of the parallel algorithms, parameters are set so as to produce a large

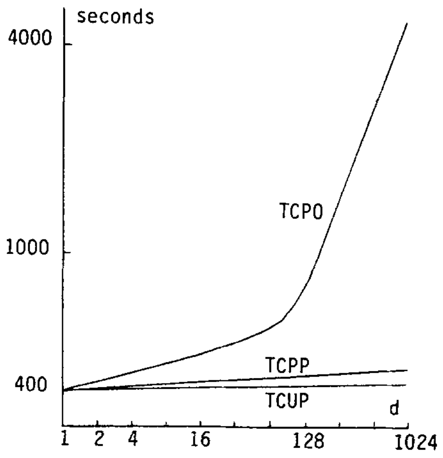

Fig. 7. Total time versus number of nodes ($|DT| = 2{,}000{,}000$).

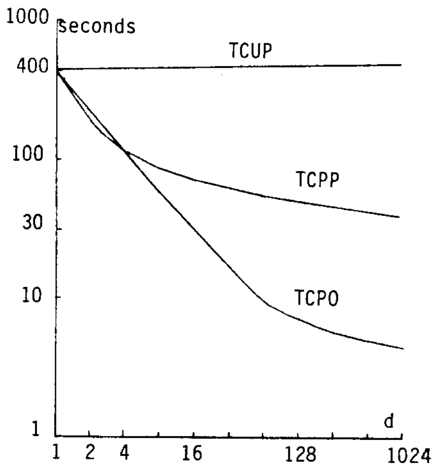

Fig. 8. Response time versus number of nodes (|DT| = 2,000,000).

number of new tuples. Assuming that a node is realized with a 5 MIPS microprocessor, the time to process a message is 1 millisecond.

Figures 7 and 8 describe the variation of total time and response time, respectively, to produce 2,000,000 new tuples. In Fig. 7, TCUP obviously provides the best total time. The total time of TCPP is slightly superior to this of TCUP. The difference is essentially the additional cost of transfer-

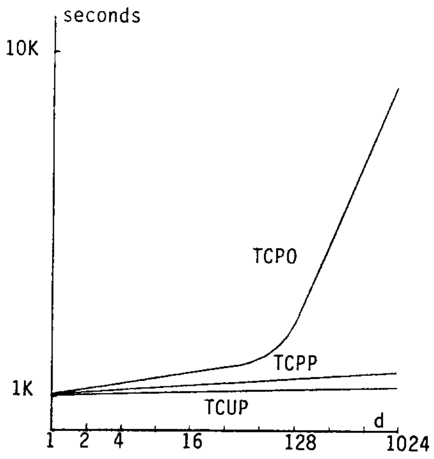

Fig. 9. Total time versus number of nodes (|DT| = 5,000,000).

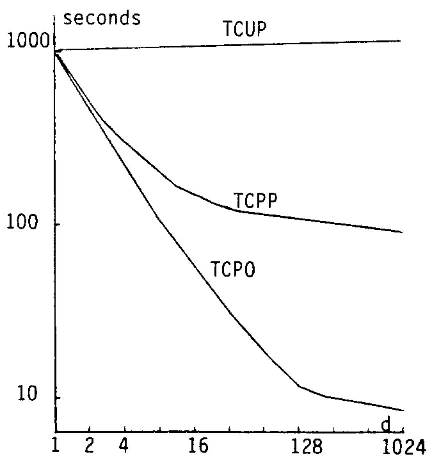

Fig. 10. Response time versus number of nodes (|DT| = 5,000,000).

ring transitively closed relations for TCPP. Since the number of messages is not high in TCPP (inter-node communication is 1 – 1), the difference is low. This results in excellent total time of TCPP compared to TCPO. The total time of TCPO is always the worst and degrades dramatically as the number of nodes is greater than 64. This behavior is due to the cost of partitioning the new tuples, which increases significantly with the number of nodes.

In Fig. 8, TCUP obviously provides the worst response time. The response times of both TCPP and TCPO improve constantly as the number of nodes increases. Performance of TCPP is slightly better than this of TCPO with a few nodes. Above four nodes, TCPO is the best and the performance difference increases with the number of nodes. With 1024 nodes, the improvement factor of TCPO is about two orders of magnitude over TCUP and one order of magnitude over TCPP. This good performance of TCPO is due to its constant degree of parallelism.

Figures 9 and 10 depict total time and response time, respectively, when producing a larger relation having 5,000,000 new tuples. The performance curves relative to one another are similar to those observed in Figs. 7 and 8. However, total times and response times are higher because of the larger result. In Fig. 10, the performance of the parallel algorithms with respect to the centralized one is slightly better than in Fig. 8.

Figures 11–14 illustrate the performance of the algorithms for a fixed number of nodes ($d = 32$). Two important performance parameters have been varied: the number |DT| of new tuples generated by the transitive closure (Figs. 11 and 12) and the processor speed per node (Figs. 13

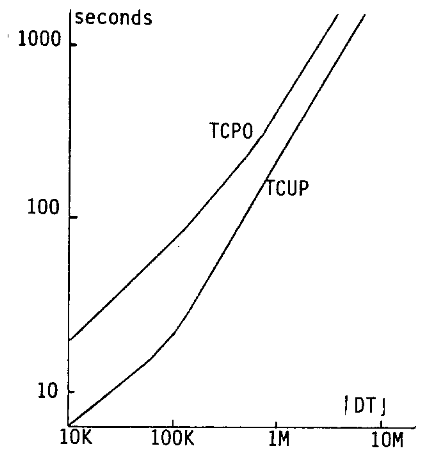

Fig. 11. Total time versus number of new tuples (32 nodes).

and 14). The total time of TCPP is not shown in Figs. 11 and 13 since it is always a little higher than that of TCUP.

In Fig. 11, the increase of the total time of both algorithms is almost linearly proportional to the increase of the size of relation |DT|. Again, TCPO incurs the worst total time. TCPO generates large messages when the result is large and smaller messages when the result is small. However,

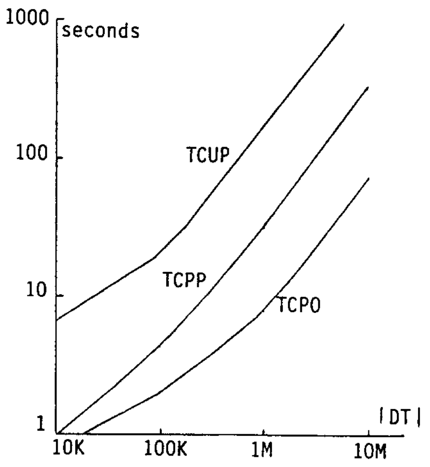

Fig. 12. Response time versus number of new tuples (32 nodes).

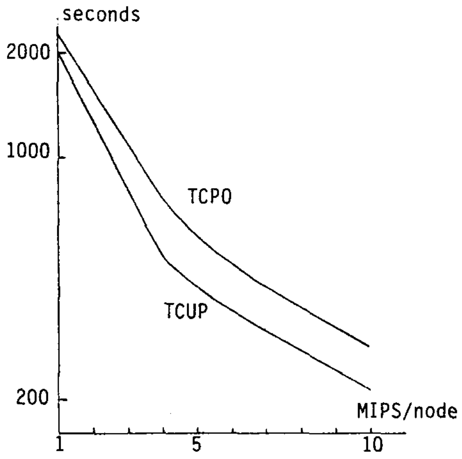

Fig. 13. Total time versus processor speed (32 nodes).

the number of messages remains high even for a small result and the fixed cost per message is the dominant factor. Therefore, the performance difference between TCPP and TCPO is higher when |DT| is small. As the result becomes larger than 1M tuples, the performance difference remains approximately constant.

In Fig. 12, the increase of the response time of all algorithms is almost

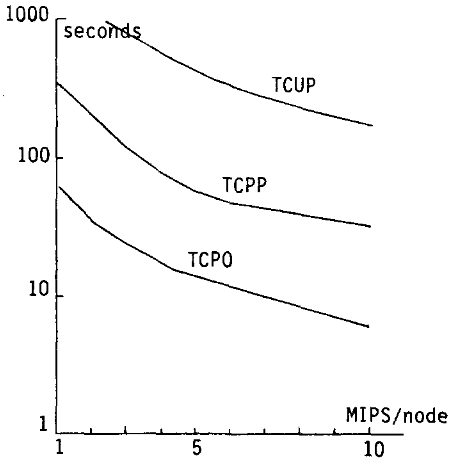

Fig. 14. Response time versus processor speed (32 nodes).

linearly proportional to the increase of the size of relation |DT|. Again, TCPO incurs the best response time.

In Figs. 13 and 14, the total time and response time of all algorithms linearly decreases as the processor speed of each node increases. Going from one MIPS to 10 MIPS processors yields exactly one order of magnitude improvement for all algorithms. The reason is that, in our model, all of processing time (parameter newtup) and the most important part of communication time (parameter *msg*) are given in number of CPU instructions.

In conclusion, parallel algorithms for the transitive closure can provide significant performance improvement over the centralized algorithm. The improvement factor is best (between one and two orders of magnitude) with a high number of nodes and a large amount of work. The centralized algorithm always involves the best total time. TCPO always incurs the worst total time which becomes prohibitive above 64 nodes. However, TCPO is almost always better than TCPP. Finally, TCPP provides a better compromise between response time and total time than TCPO.

7. CONCLUSION

We have proposed and analyzed two parallel algorithms to compute the transitive closure of a database relation in a shared nothing parallel data server. These algorithms are parallel versions of the iterative transitive closure algorithm. Compared to the centralized algorithm, the parallel algorithms may significantly improve response time when the number of nodes is high (about 100) and the transitive closure produces a large number of new tuples. The response time of the algorithm TCPO (with parallel operations) is generally superior to the algorithm TCPP (with parallel programs). The best response time improvement over the centralized algorithm is one order of magnitude for TCPP and two orders of magnitude for TCPO. However, TCPP provides a better compromise between response time and total time than TCPO.

In this paper, we were mostly interested in studying the value of parallelism for recursive query processing with respect to a centralized algorithm. Therefore, we chose a simple transitive closure algorithm, the iterative algorithm, that is easily amenable to parallel execution. However, there are better centralized algorithms to compute the transitive closure.(13,17) The parallel algorithms introduced in this paper were based on two principles: (1) executing the individual operations of the transitive closure in parallel (TCPO) or (2) executing the transitive closure program in parallel (TCPP). We believe the same principles could be applied to parallelize more efficient transitive closure algorithms, which looks a promising research area.

The performance results were obtained using a simple analytical model which ignored many practical considerations like network contention and nonuniform distribution of work among the nodes. It is our objective to do real experiments when the prototype of a shared nothing parallel data server being developed at MCC[3] is completed.

ACKNOWLEDGMENTS

The authors thank Haran Boral for his careful review and helpful comments. Thanks also to Marc Smith for useful discussions on communication costs. The anonymous referees also provided valuable suggestions.

REFERENCES

1. H. Boral and D. J. DeWitt, Database Machines: an Idea Whose Time has Passed? a Critique of the Future of Database Machines, *Int. Workshop on Database Machines*, Munich, (September 1983).
2. H. C. Du, Distributing a Database for Parallel Processing is NP-hard, *ACM SIGMOD Record*, Vol. 14, No. 1, (March 1984).
3. H. Boral, Parallelism and Data Management, *Int. Conf. on Databases*, Jerusalem, (June 1988).
4. M. Stonebraker, The Case for Shared Nothing, *Database Engineering*, Vol. 9, No. 1 (March 1986).
5. P. M. Neches, The Anatomy of a Database Computer System, *COMPCON Int. Conf.*, San Francisco, (February 1985).
6. D. J. DeWitt *et al.* GAMMA – a High Performance Dataflow Database Machine, *Int. Conf. on VLDB*, Kyoto, (August 1986).
7. M. Livny, S. Khoshafian, and H. Boral, Multi-Disk Management Algorithms, *ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Banff, Alberta, (May 1987).
8. D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson, Parallel Algorithms for the Execution of Relational Database Operations, *ACM TODS*, Vol. 8, No. 3, (September 1983).
9. P. Valduriez and G. Gardarin, Join and Semijoin Algorithms for a Multiprocessor Database Machine, *ACM TODS*, Vol. 9, No. 1, (March 1984).
10. D. J. DeWitt and Gerber, Multiprocessor Hash-based Join Algorithms, *Int. Conf. on VLDB*, Stockholm, (August 1985).
11. S. Khoshafian and P. Valduriez, Parallel Execution Strategies for Declustered Databases, *5th Int. Workshop on Database Machines*, Karuizawa, Japan, (October 1987).
12. F. Bancilhon and R. Ramakrishan, An Amateur's Introduction to Recursive Query Processing Strategies, *ACM-SIGMOD Int. Conf.*, Washington, D.C., (May 1986).
13. P. Valduriez and H. Boral, Evaluation of Recursive Queries Using Join Indices, *1st Int. Conf. on Expert Database Systems*, Charleston, South Carolina, (April 1986).

14. Y. E. Ioannidis, On the Computation of the Transitive Closure of Relational Operators, *Int. Conf. on VLDB*, Kyoto, (August 1986).
15. H. Lu, K. Mikkilineni, and J. P. Richardson, Design and Analysis of Algorithms to Compute the Transitive Closure of a Database Relation, *IEEE Int. Conf. on Data Engineering*, Los Angeles, (February 1987).
16. H. V. Jagadish, R. Agrawal, and L. Ness, A Study of Transitive Closure as a Recursion Mechanism, *ACM-SIGMOD Int. Conf.*, San Francisco, (May 1987).
17. R. Agrawal and H. V. Jagadish, Direct Algorithms for Computing the Transitive Closure of Database Relations, *Int. Conf. on VLDB*, Brighton, England, (September 1987).
18. M. J. Quinn and N. Deo, Parallel Graph Algorithms, *Computing Surveys*, Vol. 16, No. 3, (September 1984).
19. D. A. Schneider and M. J. Skarpelos, Design and Implementation of a Distributed Transitive Closure Algorithm, Unpublished Manuscript, U. of Wisconsin, Madison, (May 1986).
20. P. Valduriez, S. Khoshafian, Transitive Closure of Transitively Closed Relations, *2nd Int. Conf. on Expert Database Systems*, Tysons Corner, Virginia, (April 1988).
21. G. Copeland, W. Alexander, E. Boughter, and T. Keller, Data Placement in Bubba, *ACM SIGMOD Int. Conf.*, Chicago, Illinois, (May 1988).
22. P. Valduriez, Join Indices, *ACM TODS*, Vol. 12, No. 2, (June 1987).
23. K. Bratbergsengen, Hashing Methods and Relational Algebra Operations, *Int. Conf. on VLDB*, Singapore, (August 1984).
24. M. Kitsuregawa *et al.*, Application of Hash to Data Base Machine and Its Architecture, New Generation Computing, Vol. 1, (1983).