Performance Evaluation of Two Multidatabase Transaction Management Algorithms

Y. Breitbart University of Kentucky

A. Silberschatz University of Texas

ABSTRACT: A multidatabase system (MDBS) is an integrated collection of local database systems (DBMSs) that allows users to access and manipulate data distributed among the DBMSs. In this paper, we analyze the performance of two concurrency control algorithms that ensure global serializability. The first algorithm imposes no restrictions on the structure of the concurrency control mechanisms used by local DBMSs, except that transactions' execution order must coincide with their serialization order. The second algorithm requires each local DBMS to use the strict two phase locking protocol. The performance results presented here demonstrate that regardless of the algorithm, the concurrent processing of global transactions always outperforms the serial execution of these transactions. The first algorithm may cause a significant number of global transaction rollbacks, as compared to the second algorithm. We show that for the second algorithm, the number of global transaction rollbacks is quite small for reasonable multiprogramming levels, and that the multidatabase transaction management system performs almost as well as a distributed homogeneous database system.[†]

1. Introduction

A *Multidatabase System* (MDBS) is a software system that allows transactions to access and manipulate data located in a number of autonomous local databases distributed among nodes of a computer network. Each local database is controlled by a local database management system (DBMS).

An MDBS creates the illusion of a single database system and provides a uniform access to preexisting local databases without requiring users to know either the location or the characteristics of different databases and their corresponding DBMSs.

An MDBS is built on top of local DBMSs that manage local data sources. Each local DBMS operates autonomously. The desire to preserve the autonomy of local DBMSs is a major characteristic of a multidatabase system that separates it from conventional distributed database systems. Another important distinction between a multidatabase system and a conventional distributed database system is that in the multidatabase system there are two types of transactions: local and global. The *local transactions* are executed by a local DBMS outside of the MDBS system control, while the *global transactions* are executed under the MDBS system control.

These two characteristics cause significant difficulties in designing concurrency control schemes that ensure global serializability. Multidatabase concurrency control algorithms fall into two basic categories:

[†]This material is based in part upon work sponsored by the Center for Manufacturing and Robotics of the University of Kentucky, the National Science Foundation under grants No. IRI-8904932 and No. RRI-8610671, and a grant from Hewlett-Packard Corporation; and upon work supported by the Texas Advanced Technology Program under Grant No. ATP-024, the National Science Foundation under Grant IRI-9106450, and grants from the IBM and Hewlett-Packard corporations.

1. Algorithms that require the multidatabase system to obtain some local DBMS control information (such as a wait-for-graph, a local schedule, a DBMS log, etc.) ([Pu87], [SKS91]).
2. Algorithms that do not require any local DBMS control information ([AGM87], [BS88], [BLS91], [LT89], [MRBKS92]).

Algorithms of the first type infringe on the local autonomy of a DBMS, and, thus, are not applicable to the MDBS model we are considering here.

The performance characteristics of centralized DBMS concurrency control mechanisms have been studied extensively ([AD83], [Car83], [GST83], [TS84]). Most of these studies concentrate on the performance of either a specific concurrency control algorithm, such as locking (e.g., [RS77], [Lin82]), or a comparison of different concurrency control algorithms in a centralized DBMS (e.g., [AD83], [ACL85]). Performance characteristics of distributed concurrency control algorithms have been studied also ([GM79], [KJ85], [Li87], [CL86]). However, the performance characteristics of a multidatabase environment have not as yet been sufficiently studied. In [BKST84] we reported the results of our performance evaluation of a retrieval-only multidatabase system. We are not aware of any performance studies conducted in a multidatabase environment where updates are also permitted.

One of the difficulties in conducting performance evaluation in a multidatabase environment is designing a simulation model that is capable of simulating interactions between the MDBS and local systems that, in general, can use any type of concurrency control mechanism. To overcome this difficulty, we restrict our attention to the case where all local DBMSs use a locking mechanism to handle the concurrent execution of local transactions. Following the performance study results provided in [ACL85] and [CS84], we claim that such a restriction should not affect significantly our performance evaluation results, since in [ACL85] and [CS84] it was shown that as far as transaction throughput is concerned, basic concurrency control algorithms behave quite similarly.

In this paper we analyze the performance of two multidatabase concurrency control algorithms. The first algorithm [BS88] does not impose any restrictions on concurrency control mechanisms of local

DBMSs or the types of multidatabase transactions that can be used, except that global transactions execution order coincides with their serialization order at local sites. The second algorithm [BLS91] assumes that each local DBMS uses the strict twophase locking protocol [EGLT76]. In the performance study we present here, we address the following question:

How does the multidatabase multiprogramming level (the number of concurrently executed global transactions) and the number of database local sites effect global transaction throughput, the number of global transactions rollbacks, resource utilization, and average response time?

The performance of these two algorithms is compared using extensive simulation studies. Our results demonstrate that regardless of the algorithm, the concurrent processing of global transactions always outperforms the serial execution of these transactions, except in the fully replicated multidatabase case where the first algorithm is used. The first algorithm may cause a significant number of global transaction rollbacks, which makes it a less desirable concurrency control scheme for multidatabases. In the case of the second algorithm, we show that the number of global transaction rollbacks is relatively small for reasonable multiprogramming levels, and that the multidatabase transaction management system performs almost as well as a distributed homogeneous database system.

The remainder of the paper is organized as follows. Section 2 describes the multidatabase model used in this study. Section 3 outlines the two concurrency control algorithms whose performance is studied here. In Section 4 we describe our simulation model and simulation parameters. The simulation results are presented in Section 5. Section 6 concludes the paper.

2. The MDBS Model

A global database is a collection of local databases distributed among different local sites $s_1, s_2, \dots, s_k$ interconnected by a communication network. Transactions considered in our model consist of operations *read* (denoted by r), *write* (denoted by w), *commit* (denoted by c), and *abort* (denoted by a). A transaction results from the execution of a user program written in a high level programming language (e.g., C or PASCAL).

A *commit* operation is used to install permanently in the local databases the results of a global transaction. An *abort* operation, on the other hand, removes all the changes that were caused as a result of the execution of the transaction issuing the abort. A *read* copies a data item into the user address space and a *write* causes a new value of the data item to be written into one or more local databases. We assume that each data item can be read only once by the transaction and if a data item is read and written by the transaction, then a *read* occurs before a *write*. We define the notion of serializable global (local) schedule in the usual manner [BHG87], and use serializability as a correctness criterion for the MDBS and local DBMS concurrency control mechanisms.

We assume that the MDBS software is centrally located. It provides access to different DBMSs that are distributed among various local sites interconnected by a network. The model discussed in this paper is based on the following assumptions:

(1) No changes can be made to the local DBMS software. This means that local DBMSs cannot be modified in a manner that will provide the MDBS with local control information. Consequently, while the MDBS is aware of the fact that local transactions may run at local sites, it is not aware of any specifics of the transactions and what data items they may access. In addition, a local DBMS is not able to distinguish between local and global transactions which are executing at the local site. This assumption ensures local user autonomy. Local and global transactions receive the same treatment at local sites. Therefore, global users cannot claim any advantage over local users.

(2) A local DBMS at one site does not communicate directly with local DBMSs at other sites to synchronize the execution of a global transaction at several sites.

(3) A local DBMS may abort any transaction at any time. The decision to abort a transaction is made based entirely on local control information, without taking into account whether a transaction is local or global. For example, if a local DBMS is in a local deadlock, it selects a victim to abort based on the local DBMS strategy of victim selection.

(4) Each local DBMS ensures local serializability and freedom from local deadlocks.

Thus, only the MDBS is capable of coordinating global transaction execution at different local sites. However, such coordination must be conducted in the absence of any local DBMS control information. Hence, the global transaction manager must make the most pessimistic assumptions about the behavior of the local DBMSs in order to ensure global database consistency and freedom from global deadlocks.

The MDBS system consists of the following three major components:

1. **Global Transaction Manager.** The global transaction manager (GTM) is responsible for users' interactions with the MDBS system. For each operation of a user's transaction, the GTM, using the MDBS directory, prepares all information required to access the data item to which the operation refers. The GTM controls the execution of global transactions. For each global operation to be executed, the GTM selects a local site (or a set of sites) where the operation should be executed. At each such site, the GTM allocates a server, (one per transaction per site) and the operation is sent to the *scheduler* for scheduling and further execution at the selected site. Once the GTM allocates a server to the transaction, it is not released until the transaction either aborts or commits. A server allocated to a transaction at a local site acts as a global transaction agent at that site. All transaction operations that are to be executed at the site are eventually sent to the server. The next operation of the transaction is submitted for scheduling and execution only after the GTM receives a response that the previous operation of the same transaction has completed.
2. **Scheduler.** The scheduler manages the order of execution of the various *read, write, commit,* and *abort* operations of different global transactions. The scheduler receives the next entry from the GTM and then determines whether the operation should be executed, whether the transaction issuing the operation should be aborted, or whether the transaction issuing the operation should wait until it can be executed.

3. **Servers.** A server is a process generated by the transaction manager to act as an agent for the global transaction at the local site. Each server is responsible for translating global operations into the appropriate query language operations of the local DBMS, and submitting these operations for execution to the local DBMS. Each time a global transaction operation is scheduled and is submitted for execution, it is eventually received by the server. Results of the operation execution by a local DBMS are reported to the GTM.

The general structure of the system is depicted in Figure 1.

Each global transaction is submitted from a single central site, where the MDBS system software is located. For each local site at which a global transaction manipulates the local data, the *GTM* gener-

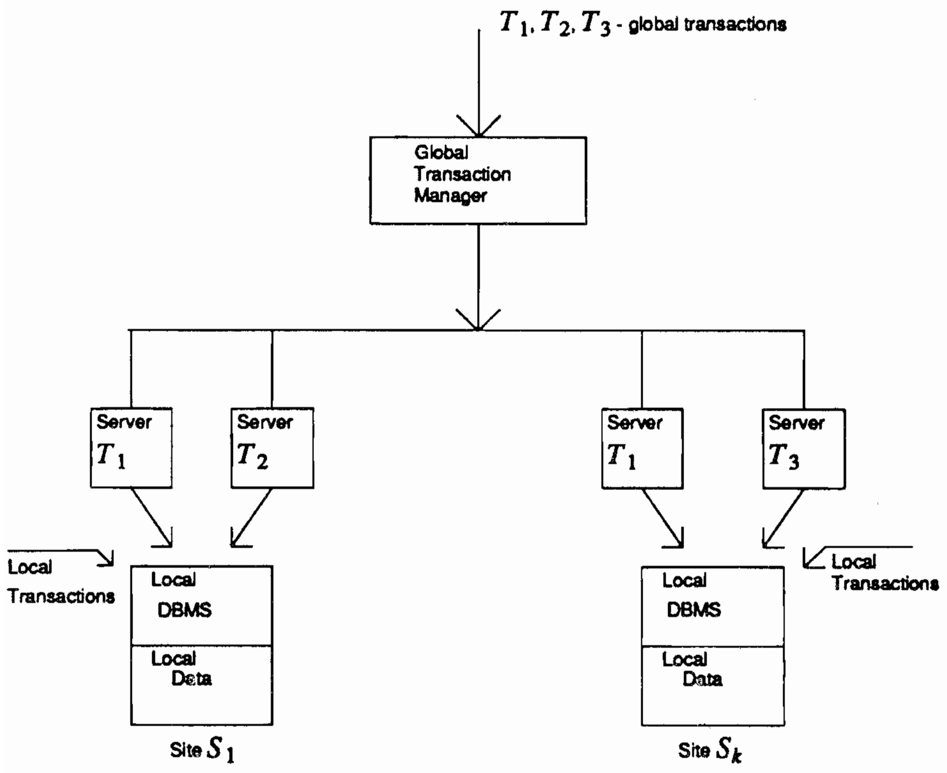

Figure 1: General Structure of the MDBS system

ates a subtransaction that is run by the server allocated to the transaction at the local site. Similar processing of distributed transactions in a homogeneous distributed environment is considered in System R^* [Lin84] and Distributed Ingres [Sto79].

3. Scheduler Algorithms

In this section we present a brief description of the two concurrency control algorithms whose performance characteristics are the subject of this paper.

3.1. The Transaction-Graph Algorithm

The algorithm is described in [BS88]. No assumptions are made concerning the nature of local DBMSs' concurrency control algorithms, except that it is required that if two global transactions are executed serially, then their execution order coincides with their serializable order at each local site where they execute together. The algorithm uses a *transaction graph* that is defined as follows.

A *transaction graph* $TG = (V, E)$ is an undirected bipartite graph whose set of vertices V consists of a set of global transactions (called *transaction* vertices) that are being processed by the MDBS, and a set of local sites (called *site* vertices). Edges in E may connect only transaction vertices with site vertices. An edge $\langle T_i, S_j\rangle$ is in E if and only if the transaction T_i has a server at site S_j and has executed at least one operation at the site.

The algorithm employs the transaction graph to ensure global database consistency. The algorithm works as follows. For each *read/write* operation submitted to the scheduler, the scheduler attempts to find local site(s) to execute the operation such that the addition to the transaction graph of new edges that connect the transaction with these local sites does not create a cycle in the graph. If such site selection is not possible, then the transaction is aborted. The aborted transaction is restarted at some later time. Otherwise, the appropriate edges are inserted into the transaction graph and the transaction operation is scheduled for execution at the site(s).

In the selection of sites to execute a *read* operation, the algorithm always selects a site where the transaction has already accessed some data item. If there are several such sites, the algorithm selects any site where the transaction has executed at least one *write* operation, if such site exists. If no such site exists, then the algorithm selects any site from the set of sites where the transaction has already executed at least one of its operations.

If a transaction has completed its operations at each local site, the transaction node along with all incidental edges are removed from the transaction graph, provided that there is no path between the transaction and any other transaction that is not yet completed its operation at at least one local site.

In [BS88] we proved that the above algorithm ensures global serializability for any set of local and global transactions. This algorithm is nonblocking, since no transaction is waiting at the scheduler level. Therefore, no global deadlock may occur. However, local deadlocks are possible. As a result of a local deadlock, a global transaction can get aborted. Consequently, the scheduler aborts the transaction at all other local sites where the transaction is or was executing and removes the transaction along with all incidental edges from the transaction graph.

3.2 Two-Phase Locking Algorithm

The algorithm is described in [BLS91]. The algorithm assumes that each local DBMS uses the strict two-phase locking protocol [BHG87], and ensures freedom from local deadlocks. The scheduler in such an environment submits all global transaction operations (except *commit*) as they arrive. The *commit* operation for transaction T_i is scheduled only after each local DBMS has completed an execution of T_i at its site. In [BLS91] we observed that such execution guarantees global serializability. Unfortunately, global deadlocks may occur. The algorithm assures freedom from global deadlocks by using a special data structure (called *potential conflict graph*) along with a timeout mechanism. In order to describe the algorithm, let us first introduce the notions of *active* and *waiting* transactions and *potential conflict graph*.

A transaction T_i *is active* at site S_j if it has a server at S_j and the server is either performing the operation of T_i at the site, or has completed the current operation of T_i and is ready to receive the next operation of transaction T_i. A transaction that is not active at site S_j is said to be *waiting* at site S_j, provided that it has a server at the site, and at least one operation of the transaction was submitted to the site. A transaction that is either *active* or *waiting* at a local site is said to be *executing* at the site.

A *potential conflict graph* (*PCG*) is a directed graph with a set of vertices V consisting of all global transactions executing in the system, and a set of edges E such that edge $T_i \rightarrow T_j$ is in E if and only if there is a site at which T_i is *waiting* and T_j is *active*.

The algorithm works as follows. For each *read/write* operation submitted to the GTM, the GTM requests local locks to execute the operation. If locks are granted, the operation is submitted to local sites for the execution and the transaction status at these sites is *active*. Otherwise, the transaction status at these sites is *waiting*. The GTM allows the transaction to wait for a local lock until *timeout* occurs. If during that time the lock is still not obtained, the GTM checks for a cycle in the *potential conflict graph*. If the graph contains a cycle, the transaction is aborted and restarted at some later time. Otherwise, the transaction continues to wait until the next *timeout* occurs.

We proved in [BLS91] that if a global deadlock exists, then there is a loop in the potential conflict graph. However, not every loop in the potential conflict graph implies a deadlock. Since the local DBMSs do not report to the MDBS the transactions waiting queues, it is possible that the global transaction manager aborts the transaction, assuming that the deadlock occurred, where, in fact, no deadlock exists. Such situations are unavoidable in a multidatabase environment where local autonomy must be preserved. For this reason, we coupled the potential conflict graph with a timeout technique that, in some cases, allows us to minimize the number of *false* deadlocks in the system.

4. Global Simulation Model

The simulation model for studying the performance of multidatabase concurrency control algorithms consists of a single global component and a set of local components. A global component consists of a model of a global database, a set of randomly generated global transactions that are executed under the control of the MDBS system, and a model of the global scheduler. Each local component consists of a model of a local database, a local DBMS, and a set of randomly generated local transactions that are submitted to the local DBMS outside of the MDBS control.

Local components communicate with the global component via messages. A local DBMS places information about a transaction operation execution on a *response queue* that is available to the MDBS system. The MDBS submits an operation for execution by placing one or more messages on a message queue that is available to any local DBMS. In our implementation we used the same data structure to simulate both a response and a message queue (shown in Fig. 2 as response queue).

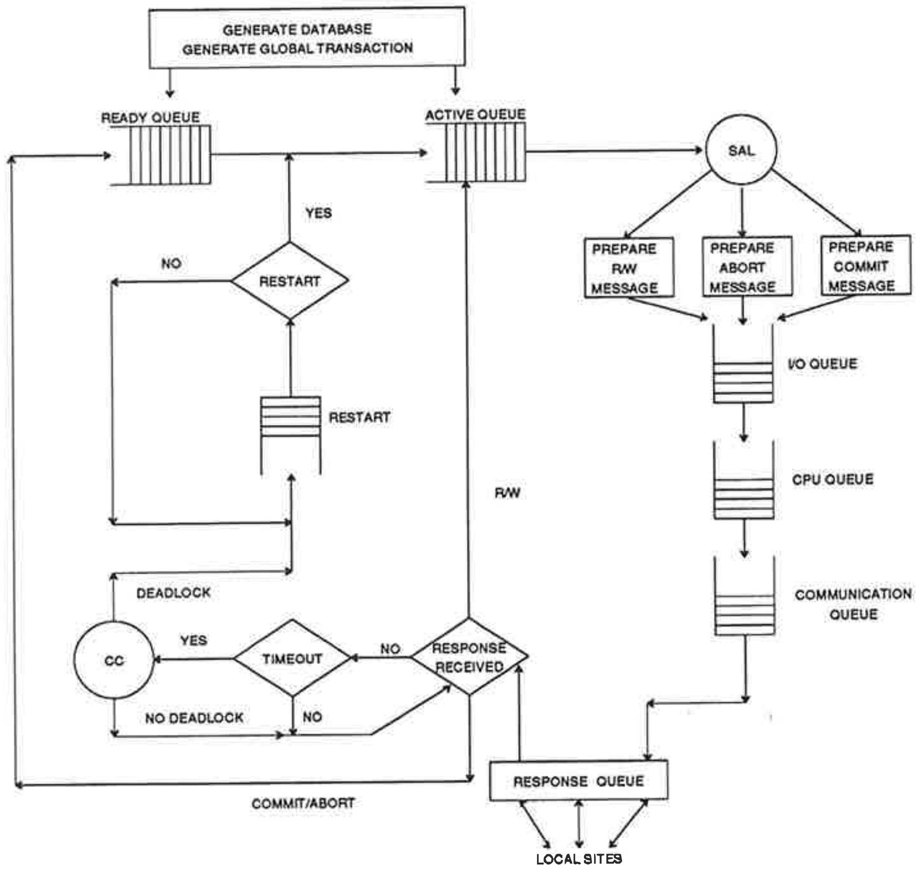

Figure 2: Global Transaction Simulation Diagram

4.1. Global Database

A global database is modeled by a collection of global data items uniformly distributed among local sites. Each global data item could be thought of as a relation, possibly replicated among different local sites, managed by different DBMSs. Figure 3 summarizes global database input simulation parameters that include the number of global data items, the number of different local sites, the number of replicated data items, and the number of data items in each local database. We define the *replication level* of a global database as a percentage of replicated data items. To simulate different replication levels we used two parameters—upper and lower bounds on the number of local sites at which a data item can be located. Replicated data items in the model were uniformly distributed among local sites. The size of the global database has a direct effect on performance results. In the performance study we evaluated multidatabases of different sizes with different levels of replication. Results reported here, however, are re-

ITEMCNT	a number of global data items. We used *ITEMCNT* = 1000 to simulate a medium size global database
SITECNT	a number of local sites. We conducted experiments for 10, 20, and 30 sites multidatabase
ITEMREP	a number of replicated data items; for fully replicated multidatabases this parameter is equal to *ITEMCNT* and for non replicated multidatabases it is equal to 0
MAXSITE	a maximum number of local sites at which each data item can be replicated *MAXSITE* = 1, for non-replicated multidatabases; *MAXSITE* = *SITECNT*, for fully replicated multidatabases
MINSITE	minimum number of local sites at which each data item can be replicated *MINSITE* = 1, for non-replicated multidatabases; *MINSITE* = *SITECNT*, for fully replicated multidatabases
$LDBSIZE_i$	a number of data items at the site S_i Each local database consists of two parts: nonreplicated and replicated data items.

Figure 3: Global Database Input Simulation Parameters

stricted to a nonreplicated multidatabase with 1,000 data items that reflects a medium-size multidatabase.

4.2 Global Component Workload Parameters

The performance of any multidatabase concurrency control system depends essentially on both the workload of the global component of the model, and the workload of each local component. In this subsection we describe the global component workload parameters that significantly effect MDBS performance.

A fixed set of global transactions generated at the start of the simulation process were circulating continuously through the simulation model as shown in Figure 2. The set consists of 400 different transactions. The size of the set ensures that during the simulation process each transaction circulated through the simulation process approximately 8 to 9 times. The model assumes that the system is never idle and that as soon as a global transaction is completed, another transaction is always available and waiting to start processing.

We assume that any global transaction in the model cannot write a data item unless it reads it first. We also assume that each global transaction accesses no more than 5% of the global database, and on the average each global transaction accesses 3% of the global database. Each transaction in the system writes no more than half of the read items. Thus, each global transaction in the system contains no more than 7 operations with no more than two writes among them.

The performance of a set of global transactions depends on essentially the number of global transaction restarts caused by:

- the global concurrency control algorithm
- the average number of transaction operations per each transaction
- the replication level among global data items
- the maximum number of global transactions that are allowed to be executed concurrently at the global level along with the maximum number of transactions to be executed concurrently at each local site.

The final number may be different at each local site. However, to simplify our model we assume that each local DBMS executes the same number of transactions concurrently at its site.

Figure 4 summarizes the basic global transaction parameters. They include the maximum number of global transactions that can be concurrently processed by the system (*multiprogramming level* of the system), the CPU and I/O times spent by the concurrency control algorithm of the MDBS system per each transaction operation, a restart delay—the minimal time that a global transaction should wait after the

GMAXACT	multiprogramming level. We conducted experiments for *GMAXACT* = 1, 10, 15, 25, 50, 75 and 100 for the transaction graph algorithm and for *GMAXACT* = 5, 10, 25, 50, 75 for the 2-PL algorithm. This parameter determines a number of active global transactions that are executed by the MDBS system.
CPUCC	CPU time spent by the concurrency control algorithm per data item. *CPUCC* = .007
IOCC	I/O time spent by the concurrency control algorithm per data item. *IOCC* = .015
GRESTRT	restart delay. We assumed that the rolled back transaction cannot be restarted for at least 120 simulations units.
MAXGR	maximum number of reads in each global transaction
MAXGW	maximum number of writes in each global transaction
MINGR	minimum number of reads in each global transaction
MINGW	minimum number of writes in each global transaction
GIOM	I/O time to prepare a message from MDBS to a local site. *GIOM* = .035
GCPUM	CPU time to prepare a message from MDBS to a local site. *GCPUM* = .015
GCOMTM	time to send a message from MDBS to a local site. *GCOMTM* = .1
GMDM	number of messages per a data item to be sent from MDBS to a local site. *GMDM* = 2
OPDELAY	delay required by a transaction to process data received from a local site before submitting the next read/write request from the same transaction.

Figure 4: Global Component Workload Input Simulation Parameters

rollback before it can be restarted by the system, and the upper and lower bounds on a number of global *read* and *write* operations for each global transaction in the system.

After the execution of each read/write operation of a global transaction by the local DBMSs, the MDBS may need some time to perform additional operations on the data obtained (for example, these operations may include a join of data from two relations, in the case that the local DBMS has no capacity to perform a join). In addition, the transaction needs some time to process the obtained information before the next operation of the same transaction is submitted to the MDBS. We lump all these times into a parameter *OPDELAY* (operational delay), i.e., the time that a global transaction should wait before submitting the next read/write operation after the previous operation has been completed. We performed experiments with different values of *OPDELAY* to understand its impact on a number of global transaction restarts and on global transaction throughput. We found, however, that *OPDELAY* does not significantly affect either of these data. Thus, we report the result *OPDELAY* = 0.

Each *read/write* and *abort/commit* request that is issued by a global transaction is translated into one or more messages that are sent to local sites as determined by the MDBS. Thus, global transaction parameters also include required I/O, CPU, communication times to send a data message from MDBS to a local site, and the number of messages required per one data item in case of *read/write* operations. Since *abort/commit* messages are relatively short, we assumed that only one message per site for an *abort* operation and one message per site for each stage of the *commit* operation was required.

Initial values for these parameters were selected to approximate a realistic computing environment. The restart delay value was chosen in a way that ensures the completion of at least one of the executing transactions in the model. In this case, we can reduce a number of transactions rollbacks, since the completion of a transaction in the system creates new conditions for the restarted transaction that may eliminate the conditions that caused it to abort the first time.

4.3. Local Component Workload Parameters

In a multidatabase environment, the response time of a global transaction depends significantly on the amount of local processing. A local

DBMS processes transactions of two types: local transactions generated by local users outside of the MDBS system control, and global transactions whose operations are sent by the MDBS for execution to the local site. Each local simulation model may, *a priori,* have its own local simulation parameters. To simplify the multidatabase simulation model, we assume that input simulation parameters at each site are independent of a local site.

We assume that at each local site there is a constant ratio of local and global transactions. We simulate local transactions at local sites by generating a system of local transactions and circulating them through the local site simulation model. The number of generated local transactions can be changed dynamically during the simulation to maintain a constant ratio of local and global transactions. It is reasonable to assume that in the multidatabase environment most transactions at a local site are local and few are submitted by the MDBS system.

Our local simulation model is closely related to the model described by [ACL85]. Figure 5 shows the local workload input simulation parameters that include a local multiprogramming level, upper and lower bounds for a number of local *read/write* operations, time delay before a local transaction restarts after it has been rolled back, and local CPU and I/O times to perform one transaction operation.

Generally, a local DBMS does not know whether a transaction is either local or has been submitted by the MDBS. The result of a global transaction is communicated by the server to the MDBS system. Thus, local component workload parameters should include the I/O, CPU, and communication times to send data messages from a local system to the MDBS, as well as the number of required messages per one data item in case of *read/write* operations.

When a local or global subtransaction enters the ready queue at site s_i, the entrance time is recorded as the start time of the transaction processing. After the transaction successfully completes at the site, transaction completion time is also recorded. The performance of multidatabase schedulers was studied using a global transaction simulation process shown in Figure 2. Initially, a fixed set of global transactions is generated, and placed on a *READY* queue. A subset of *GMAXACT* transactions is placed on the *ACTIVE* queue. The number of transactions concurrently executed in the system is limited. A transaction is considered to be executing in the system if it is either receiving or waiting for service at a local site. Thus, an executing

MAXACT	maximum number of active transactions at a local site (local multiprogramming level). In our experiments we assumed that *MAXACT* = 15 at all local sites.
MAXLR	maximum number of reads in a local transaction at a local site. In our experiments we assumed that *MAXLR* = 5. Global subtransactions may have more than *MAXLR* read operations.
MAXLW	maximum number of writes in a local transaction at a local site. In our experiments we assumed that *MAXLW* = 2. Global subtransactions may have more than *MAXLW* read operations.
MINLR	minimum number of reads in a local transaction at a local site. In our experiments we assumed that *MINLR* = 2. Global subtransactions may have less than *MINLR* read operations.
MINLW	minimum number of writes in a local transaction at a local site. In our experiments we assumed that *MINLW* = 1. Global subtransactions may have less than *MINLW* read operations.
LIO	local I/O time to perform one transaction operation at a local site. *LIO* = .015
LCPU	local CPU time to perform one transaction operation. *LCPU* = .007
LRESTRT	restart delay at the site. *LRESTRT* = 30 simulation units.
LIOM	local system I/O time to prepare a message from a local site to the MDBS. *LIOM* = .035
LCPUM	local system CPU time to prepare a message from a local site to the MDBS. *LCPUM* = .015
LCOMTM	time to send a message from a local site to the MDBS. *LCOMTM* = .1
LMDM	number of messages per a data item to be sent from a local system to the MDBS. *LMDM* = 2.
LOPDELAY	a delay required to process a local data item by a local transaction before the next transaction operation is submitted.

Figure 5: Local Workload Input Simulation Parameters

global transaction in the model is either on the *ACTIVE* or the local *BLOCKED* queue (see Fig. 6). At any time during the simulation, the system maintains *GMAXACT* concurrently executing global transactions in the system.

The first transaction on the *ACTIVE* queue makes its request to the MDBS. The transaction operation is analyzed by the GTM to determine the local sites that the information should be either sent to or requested from. Following this, the transaction operation is submitted to the scheduler.

If the transaction operation is scheduled, the transaction proceeds consecutively to the *I/O, CPU,* and *COMMUNICATION* queues to perform I/O, CPU, and communication operations, respectively, to prepare and send messages to local sites where the transaction operation should be executed.

Sending messages to a local site is simulated either by updating the global subtransaction that is already on the *ACTIVE* queue of the local model with a new operation that needs to be executed, or by placing a global subtransaction at the local site on the *READY* queue of the local model, if the transaction has no operations yet executed at the local site. The local simulation model (depicted in Figure 6) is used to conduct a local system simulation (local transaction processing is described in the next subsection). The transaction then is placed on the *RESPONSE* queue (see Figure 2) to wait for the response from the local site that should execute the transaction operation. A response received from a local site is either a requested data item (in case of the *read* operation) or a confirmation that the requested operation has completed or failed. If a response is received before the transaction timed out and it is either a set of data or a confirmation that the *write* operation has successfully completed, the transaction is placed back on the *ACTIVE* queue and it is ready to place its next operation request. If a response is a confirmation of a *commit* completion, the transaction is placed on the back of the *READY* queue as if it were a new transaction that has arrived for execution.

An *abort* completion is more complex. There are two types of an *abort* operation: an *abort* as a part of the transaction generated by the user, and an *abort* generated by the system due to some system conditions, such as, global or local deadlock. The first type of *abort* is considered a normal user operation. After the *abort* is executed, the transaction is placed at the back of the *READY* queue (considered to be a new transaction without any connection whatsoever to the aborted transaction). The second type of *abort* is considered as a transaction failure; the transaction is placed at the back of the *RESTART* queue. If the MDBS decides to abort the transaction, then the *abort* operation is included as a part of the transaction, and the transaction is put at the front of the *ACTIVE* queue. The transaction will be actually aborted in the next simulation unit as a result of the *abort* operation.

If a global deadlock is declared, then the scheduler selects the transaction to be aborted in order to break a deadlock. The aborted transaction is placed at the back of the *RESTART* queue, and it can be restarted after *GRESTRT* restart delay by placing the transaction at the front of the *ACTIVE* queue. After the transaction successfully completes all its *read/write* requests, the scheduler sends the *commit* operation to all sites at which the transaction has performed at least one operation. Upon transaction completion (that is, after the transaction commits or aborts), the system inspects first the *RESTART* queue and then the *READY* queue to select a transaction to place on the back of the *ACTIVE* queue.

In the simulation process, the global transaction response time is one of the parameters of interest. This time consists of the global CPU and I/O times spent by the transaction for all its global data items, the maximum of local response times, and, finally, the overhead of possible restarts or other delays caused by conflicts discovered either at the global or local level. During the simulation, each global transaction will accumulate the time spent for data item processing (I/O, CPU, and communication) and the time spent at local sites. If transaction processing should be stopped and a transaction should be backed out, the cumulative CPU and I/O time required to perform a back out at each site is recorded and the transaction is put on the restart queue.

4.4. Local Transaction Processing Model

Our local transaction processing model is a slightly extended version of the model proposed by Ries and Stonebraker [RS77], and extended by Agrawal [ACL85]. For the purposes of our simulation, we assume that each local site uses the strict two-phase locking (2PL) protocol, specifically, the blocking algorithm described in [Gra79], to simulate local concurrency control in a multidatabase environment. The transaction-graph algorithm, however, does not take this fact into consideration, although the 2PL algorithm does. The global transactions that are to be executed at a local site, along with generated local transactions, are placed on the local *READY* queue at the start of the simulation. During the simulation, there is a limit *MAXACT* on the number of local and global transactions that can be active at the local site. A transaction at the local site is active if it is on the local *ACTIVE, I/O, COMMUNICATION,* or *BLOCKED* queues.

At any time during the local simulation, a global transaction may send an operation to be executed at the local site. If the global subtransaction is already active at this site, then a requested operation is added to the subtransaction. Otherwise a global subtransaction is created at the local site and it is placed on the local *READY* queue.

The simulation process starts with the first transaction at the local *ACTIVE* queue submitting a request to the concurrency control mechanism of the local DBMS. Each *read/write* operation of a transaction requires a lock on a local data item. If the concurrency control module can grant the lock, then the transaction operation is executed. If, however, the lock cannot be granted, and the local deadlock detection algorithm has determined that no local deadlock can occur from the transaction wait for the lock, then the transaction is placed on the *BLOCKED* queue. If the transaction's wait for the lock causes a deadlock, then the transaction is aborted at the local site and is placed on the local *RESTART* queue. Transactions from the local *RESTART* queue can be restarted only after *LRESTRT* delay.

If the transaction has received a lock to execute the operation, it proceeds consecutively through local *I/O* and *CPU* queues to perform the I/O and CPU operations required to access the local data item. The results of the operation, in the form of messages, are placed on the local *COMMMUNICATION* queue to be sent to the global component site.

The global and local components exchange information about a global transaction execution through a common area known as the global *RESPONSE* queue (shown in Fig. 2). Messages prepared by the local concurrency control mechanism are placed on the global *RESPONSE* queue to simulate message sending from the local site to the MDBS site.

If a transaction request at a local site was either a *commit* or an *abort* operation, and it was successfully executed, then the transaction is purged from the system and a new transaction is generated and placed at the back of the local *READY* queue. For the purposes of this study, we simplified this process by distinguishing between local and global transactions. Global transactions were purged from the local system, while local transactions were placed at the back of the local *READY* queue. In either case, after a transaction has completed, the local system checks first the local *RESTART* queue and then the *READY* queue to determine whether a new transaction can become active. In addition, the *BLOCKED* queue is checked to determine whether any transaction from the queue can be unblocked by granting locks to a transaction that was released by the committed or aborted transaction.

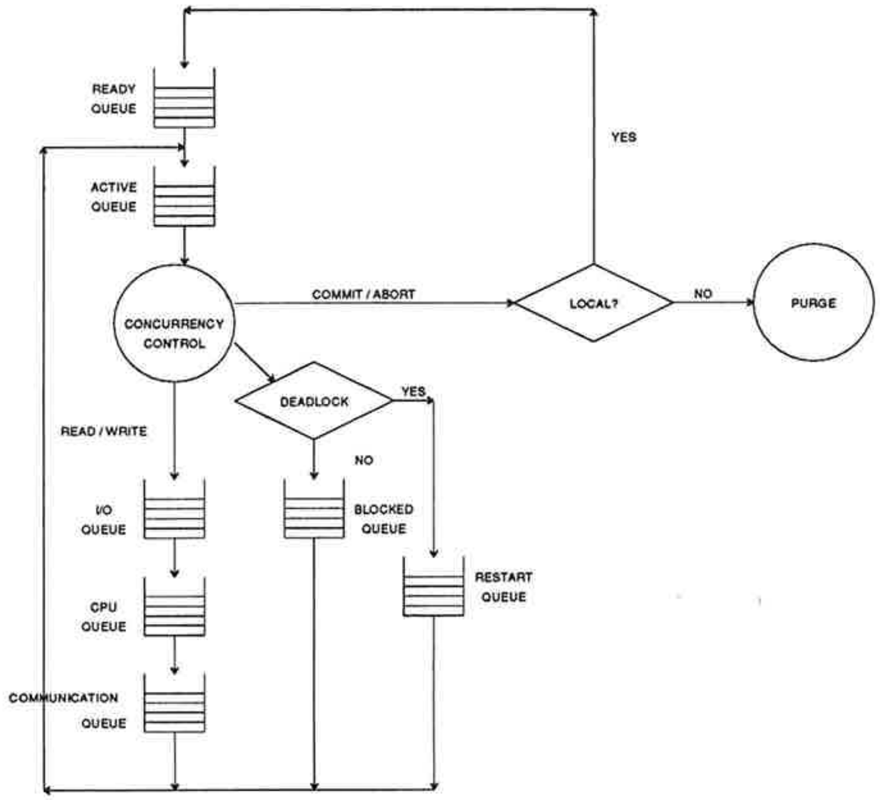

Figure 6: Local Transaction Simulation Diagram

4.5. Physical Queuing Model

Associated with each transaction operation is a set of messages to be prepared and sent from/to the MDBS to/from a local site. Preparing the messages requires physical resources at both the MDBS site and each local site. Both logical models are characterized by three physical resources: CPU, I/O, and communication.

Whenever a global or a local transaction requests some services described by their logical models, it will use one of these resources. The amounts of I/O, CPU, and communication used by each transaction operation are specified as input simulation parameters (global or local). The physical queuing model is depicted in Figure 7. The physical model is a collection of global and local I/O devices, CPU devices and communication devices.

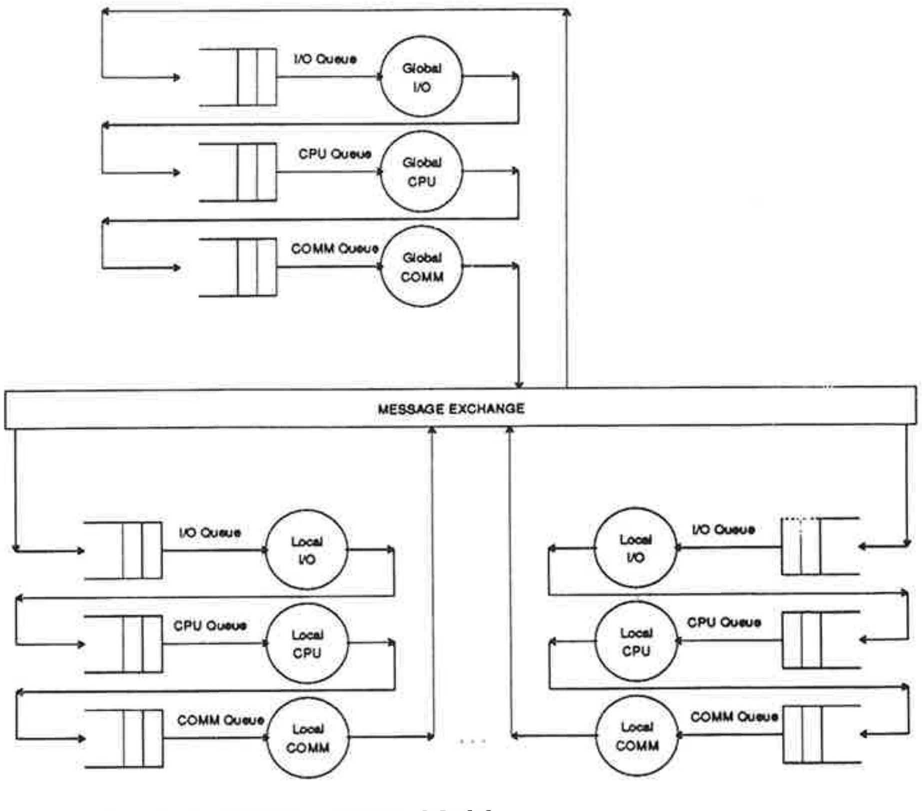

Figure 7: Physical Queuing Model

In this study, we assume that a global, and each local, component contains a single CPU, a single I/O disk and a single communication port. The first two resources are used to prepare messages that are exchanged between global and local sites; the last resource is used to simulate message sending through a network of global and local sites.

Each request submitted to the system is entered first on an I/O queue of the global model, and after being served, is entered at the end of the CPU queue. If any messages have been created, they were placed on the bottom of the communication queue. For *read/commit/abort* operations, the MDBS creates a single message to be sent to a local site. For a *write* operation, the MDBS creates *GMDM* messages to send to a local site, new values of a data item that is to be updated. For *write/commit/abort* operations a local site creates a single message to be passed to the MDBS. For a *read* operation, a local site creates *LMDM* messages containing the data requested by the MDBS.

Physical resource requests queues generally will be served on a firstcome-first-serve basis. We do not exclude, however, that a local resource queue will assign higher or lower priority to global subtransactions in experiments to clarify the impact of global subtransactions on local transaction processing.

5. *Performance Results*

In this section we describe the results of our performance simulation experiments. The results presented here assume a nonreplicated multidatabase, although our complete results include data pertaining to multidatabases with different replication levels.

In our experiments, we measured global transaction throughput; CPU, I/O, and communication times, the average global transaction response time, the restart ratio; and the average number of global restarts per a restarted transaction as a function of a global multiprogramming level and a number of local sites. The margin of error in our experiments is within 15%.

To measure global transaction throughput, the parameter *TOTSUB*—the total number of submitted transactions—was kept by the system. The parameter *TOTCOMP*—the total number of completed transactions—was also kept in the system. Each time a transaction moved from the *READY* to the *ACTIVE* queue, *TOTSUB* was increased. Each time a transaction committed and was placed at the back of the *READY* queue as a completely new transaction, *TOTCOMP* was increased. We also measured the percentage of completed transactions out of transactions that were submitted for execution during the simulation process.

For each completed global transaction, we measured the transaction response time. This time consists of the global CPU, I/O and communication times spent for all global data items, the maximum local response time for each local site where the transaction was active, and the overhead caused by all transaction restarts due to a global deadlock. During the simulation, each global transaction accumulated times spent at local sites and times that a transactions waited on any of the global model queues. If a transaction was aborted and restarted, cumulative CPU, I/O and communication times that the transaction spent before the abort were retained and further updated after the transaction were restarted.

The results of our tests indicate that the number of completed transactions during the simulation period is much larger for a concurrent transaction execution than for serial execution for both algorithms that were simulated.

For completed global transactions, we also measured the percentage of global transactions that were completed and restarted at least once along with the average number of restarts per a completed global transaction that was restarted at least once. A global transaction is restarted for only one reason: The GTM aborts the transaction due to the global scheduler algorithm requirements (for example, a loop in the transaction graph is discovered). If a global transaction is aborted at a local site (by a local concurrency control mechanism) then it is placed on a local restart queue. After local restart delay, the transaction is restarted invisible to the global scheduler. Such an assumption is valid in our model, since we did not consider the case of failure during transaction processing.

Finally, we measured I/O, CPU and network communication utilization. These values were computed as a ratio of I/O, CPU, and network communication times used by completed transactions to total available I/O, CPU, and network communication time, respectively.

Experiments were conducted for multiprogramming levels with 10, 20, and 30 sites and multidatabases for 3000 simulation units. The total number of transactions that completed during the simulation process as well as a percentage of completed transactions for the transaction-graph and 2PL algorithms are shown in Figures 8 and 9, and 8A and 9A, respectively. In both cases, the number of completed transactions increases with the increase of the multiprogramming level. In the case of the transaction-graph algorithm, the maximum is reached at multiprogramming levels equal to 25 or 50 (depending on the number of local sites). In the case of the 2PL algorithm, the maximum is reached at a multiprogramming level equal to 50. In the case of the transaction-graph algorithm, the total number of completed transactions oscillates around its maximum value for larger multiprogramming levels. In the case of the 2PL algorithm, however, the number of transactions completed after reaching the maximum level starts to decrease. In both cases, the number of local sites had little impact on the number of completed transactions.

A simple explanation exists for these facts: In the case of the transaction-graph algorithm, a significant number of global transactions aborts caused by cycles in the transaction graph overshadows the effect of local deadlocks that may also cause global transaction aborts. On the other hand, in the case of the 2PL algorithm, there is no possibility of global transaction aborts, except for a deadlock (global or local). The large multiprogramming levels of global transactions increase significantly the possibility of such deadlocks. It is interesting to observe that with the increase of multiprogramming levels, the majority of global deadlocks are false deadlocks caused by increased response time from local sites that exceeds the predetermined value of the timeout. These results indicate that in the multidatabase environment, with each local DBMS using the 2PL protocol, the number of concurrently executed global transactions should be limited in order to achieve larger throughput.

Figures 8 and 9 also indicate that concurrent processing of global transactions provides better throughput than serial processing. In the case of a fully replicated multidatabase, however, the transaction-graph algorithm simulation indicated that the serial execution of global transactions provides better throughput than the corresponding concurrent execution. However, the case of fully replicated databases in a multidatabase environment is highly unlikely. If data were to be fully replicated then no need exists to integrate the data under the MDBS, since the addition of different data sources does not provide users with new information.

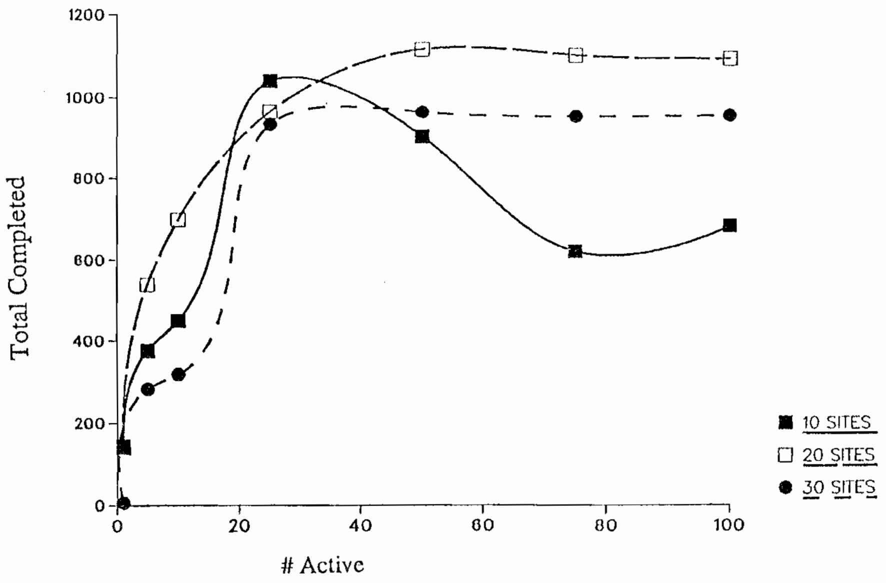

Figure 8

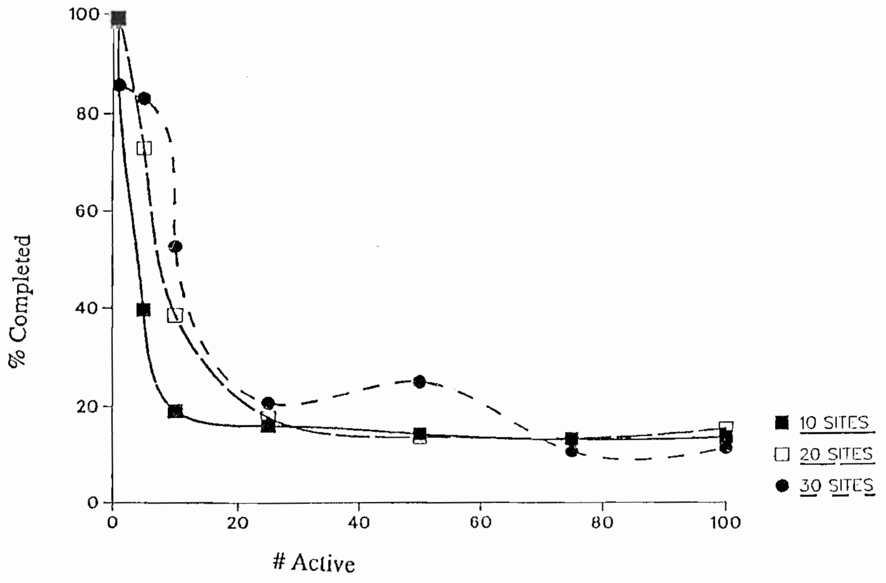

Figure 8A

2 PL Case

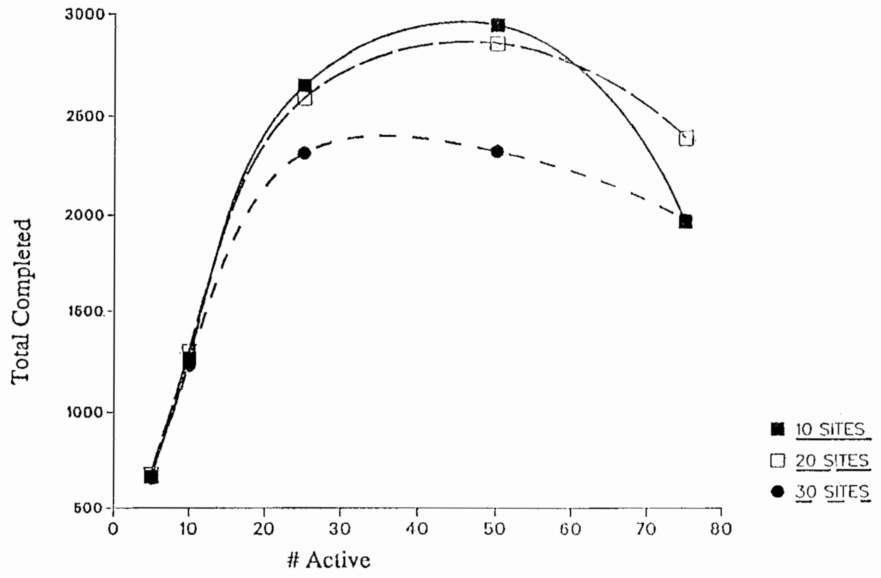

Figure 9

2PL Case

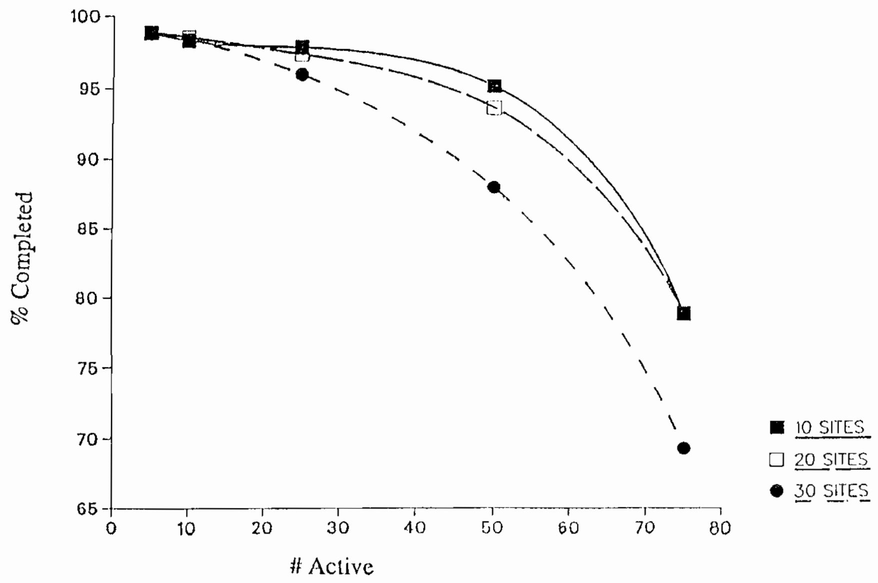

Figure 9A

Figures 8A and 9A illustrate, respectively, the percentage of completed transactions for the transaction-graph and for the 2PL algorithm. In the case of the transaction-graph algorithm, a percentage of completed transactions rapidly decreases with larger multiprogramming levels. For a multiprogramming level equal to 25, only about 20% of the transactions complete. Transaction completion remains near this number regardless of the increase in the multiprogramming level. We explain this fact by the following: With larger multiprogramming levels, and under the conditions of our model, the transaction graph generates a significant number of cycles that cause a large number of transactions aborts. Therefore, the number of completed transactions cannot be as affected by local database aborts as it is affected by the cycles in the transaction graph. Apparently, higher multiprogramming levels are insignificant in the total number of aborted transactions.

In the case of the 2PL algorithm, the percentage of completed transactions monotonically decreases with an increase in the multiprogramming level. This result reflects the fact that with large multiprogramming levels, the 2PL algorithm generates a significant number of false global deadlocks that causes an increase in global transactions aborts.

Figures 10 and 11 show the average response time for the transaction-graph and the 2PL algorithms, respectively. The response time figures are not surprising. In both cases, the average response time increases with an increase in multiprogramming levels. On the other hand, in the case of the 2PL algorithm, the average response time is approximately one quarter of the average response time for the transaction-graph algorithm.

In order to measure the impact of a multiprogramming level and the number of local sites at which a multidatabase is distributed on a total number of global transactions rollbacks, the total number of restarts was increased each time a global transaction (that eventually was completed) was aborted due to the global deadlock problem. A ratio of the total number of global transactions restarts to the total number of global transactions completed during the simulation period indicates a relative frequency of transaction aborts caused by the algorithm. Figures 12 and 13 illustrate the behavior of this ratio as a function of the multiprogramming level. In the case of the transaction-graph algorithm the per cent of restarts increases very fast with an in-

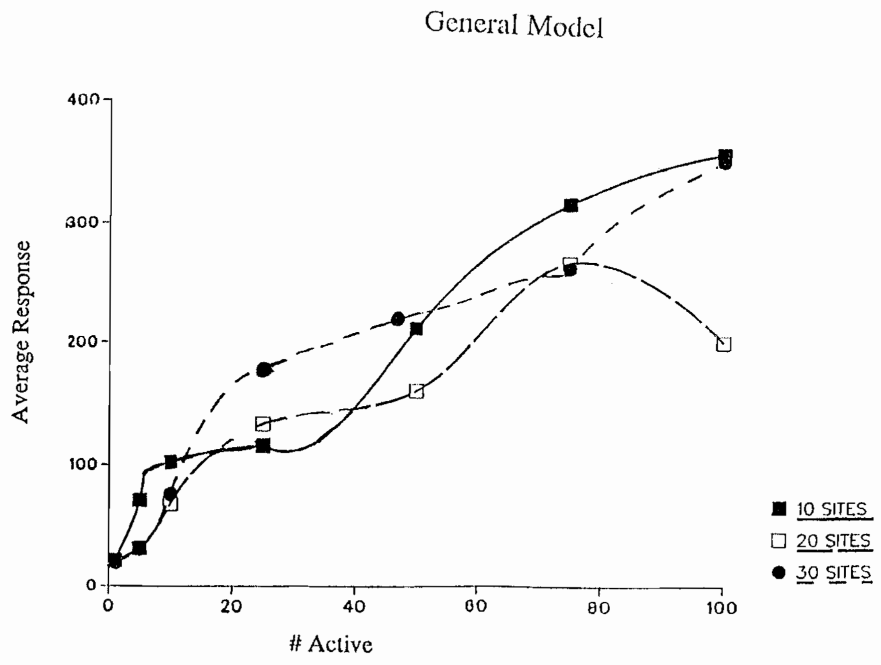

Figure 10

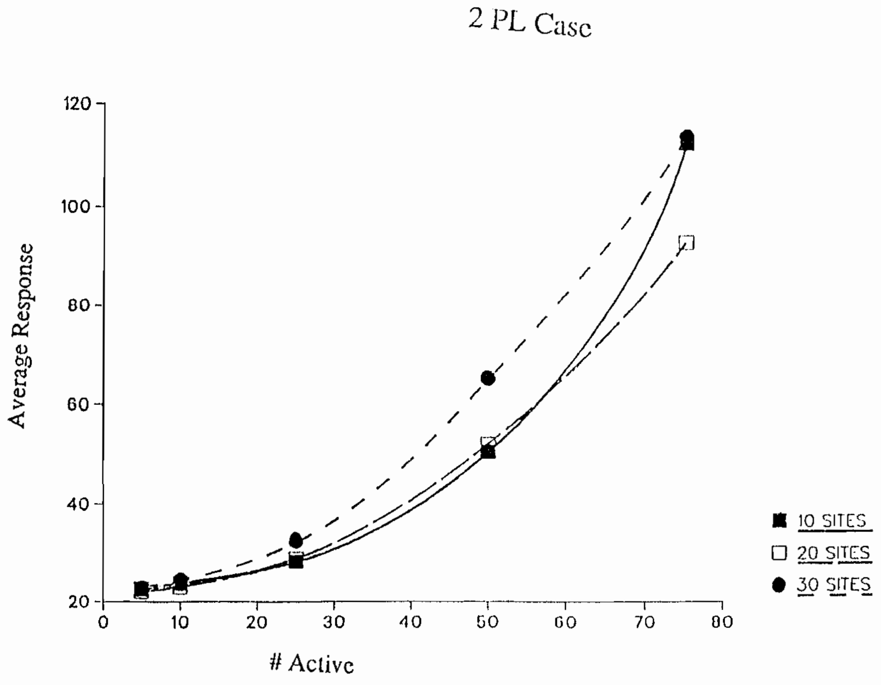

Figure 11

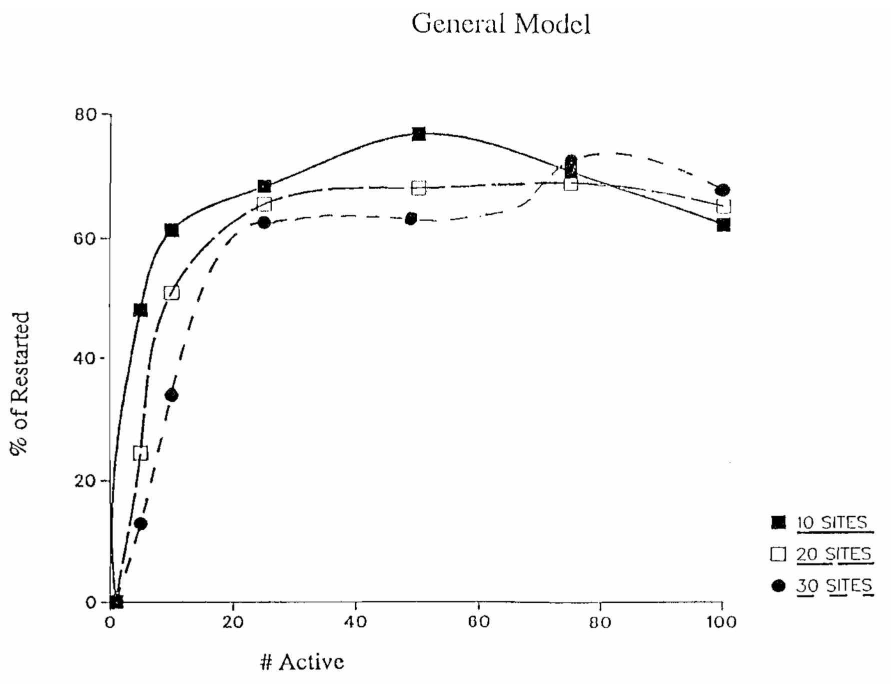

Figure 12

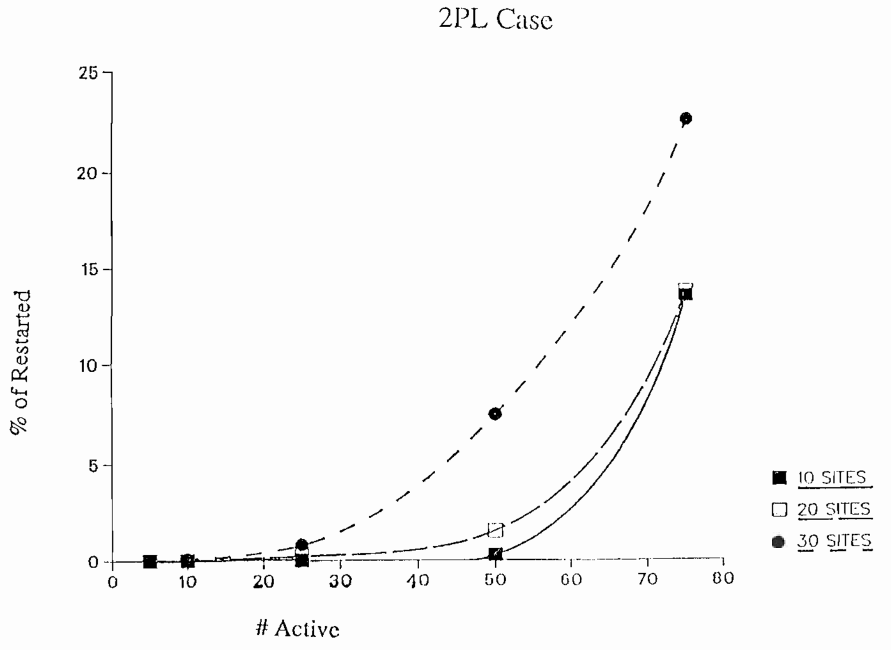

Figure 13

crease of the multiprogramming level. Figure 12 illustrates that approximately 80% of completed transactions are aborted at least once. The significant number of global transaction aborts does not seem to depend significantly on the number of local sites in a multidatabase. The 2PL algorithm, however, provides a more optimistic picture (see Figure 13). In situations where the transaction-graph algorithm causes 80% of global transactions to abort, the 2PL algorithm causes no more than 4% of transactions to abort. This number of rollbacks is quite acceptable for any practical multidatabase system.

Our next results show the average number of global transactions restarts computed as the total number of restarts divided by the number of restarted transactions. In the case of the transaction-graph algorithm (see Figure 14), each transaction can be restarted up to 5 to 6 times depending on the multiprogramming level and the number of local sites. This high number of aborts, in all likelihood, can not be tolerated in a practical multidatabase system. In the case of the 2PL algorithm (see Figure 15), few transactions are restarted more than once.

Our last results pertain to resource utilization of the computer system model used in the simulation model. These results are shown in Figures 16, 18, and 20 for the transaction-graph algorithm and in Figures 17, 19, and 21 for the 2PL algorithms. In both cases, the resource utilization is very similar: About 40–50% I/O utilization, about 20–25% CPU utilization, and about 90–97% network communication utilization. The third result indicates that in the multidatabase environment, for reasonably large multiprogramming levels, the majority of time is spent on message exchange between local sites and the MDBS system site. It appears that this fact is independent of the type of algorithm being used for multidatabase concurrency control. These results are in accord with [BKST84], who obtained similar results for retrieval-only multidatabase systems.

Our results clearly demonstrate the advantage and practicality of the 2PL algorithm versus that of the transaction-graph algorithm. On the other hand, it is likely the case that in an environment where no information is available about local concurrency control algorithms, no algorithm will perform much better than our transaction-graph algorithm. This leads us to the following two conclusions. The good news is that, in practically important situations, multidatabase concurrency control that ensures global database consistency and freedom

General Model

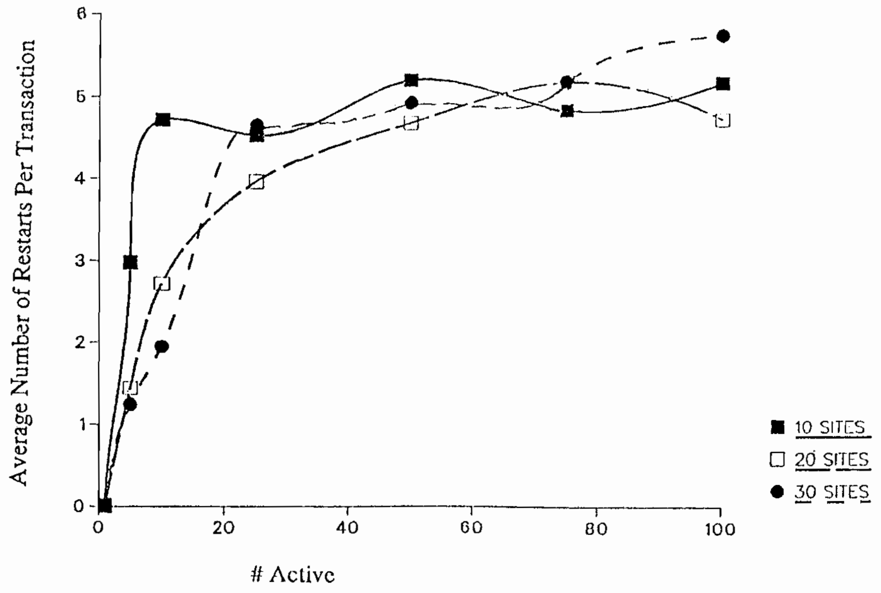

Figure 14

2PL Case

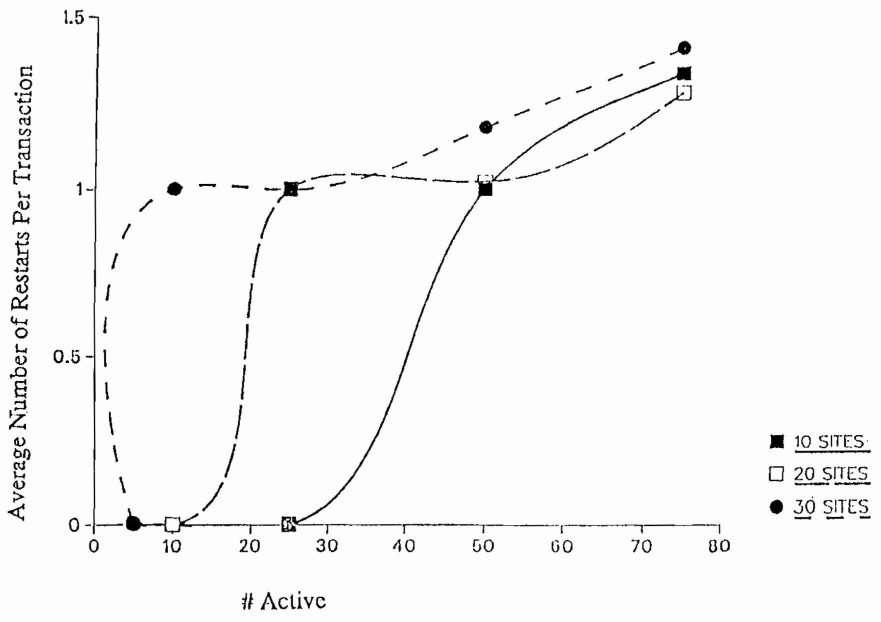

Figure 15

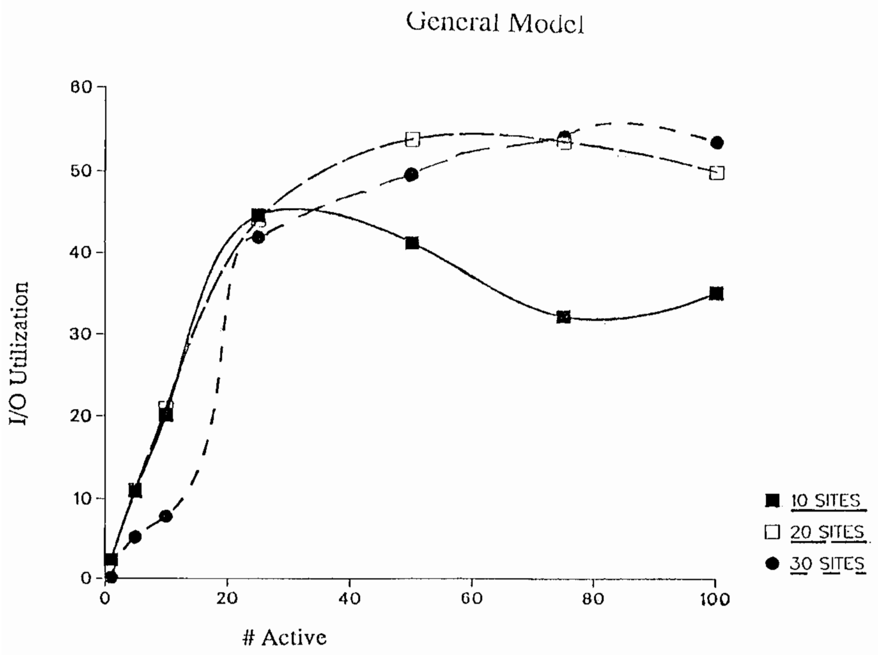

Figure 16

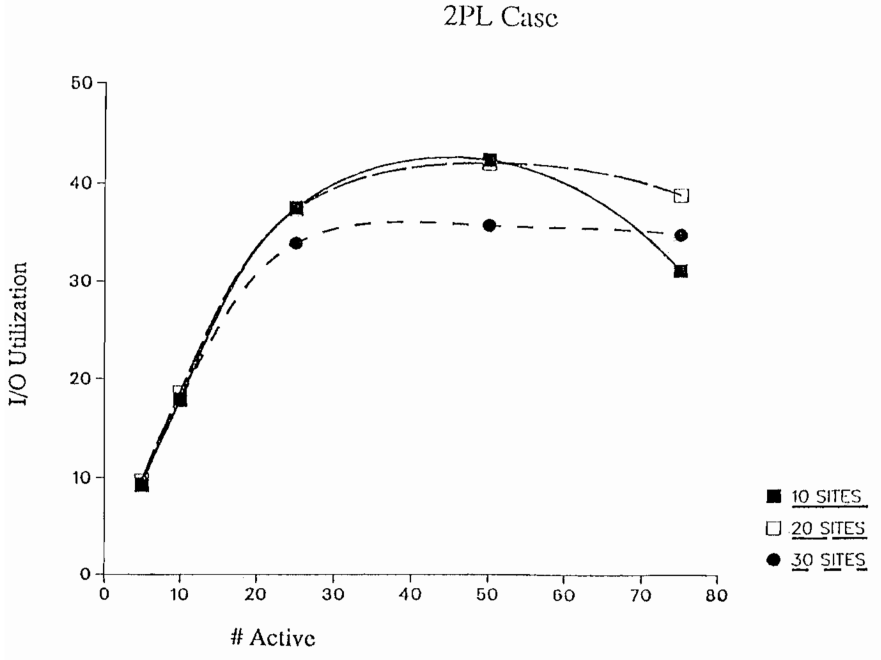

Figure 17

General Model

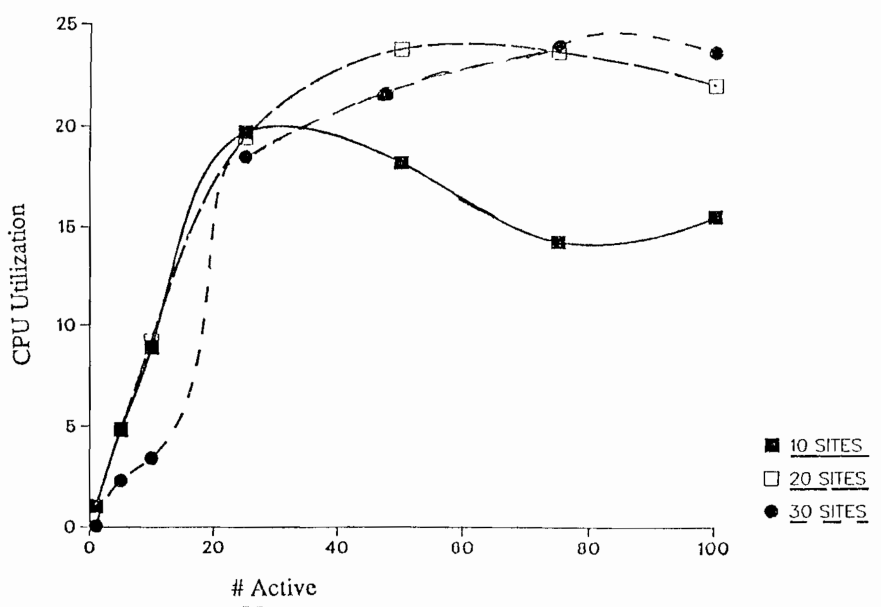

Figure 18

2PL Case

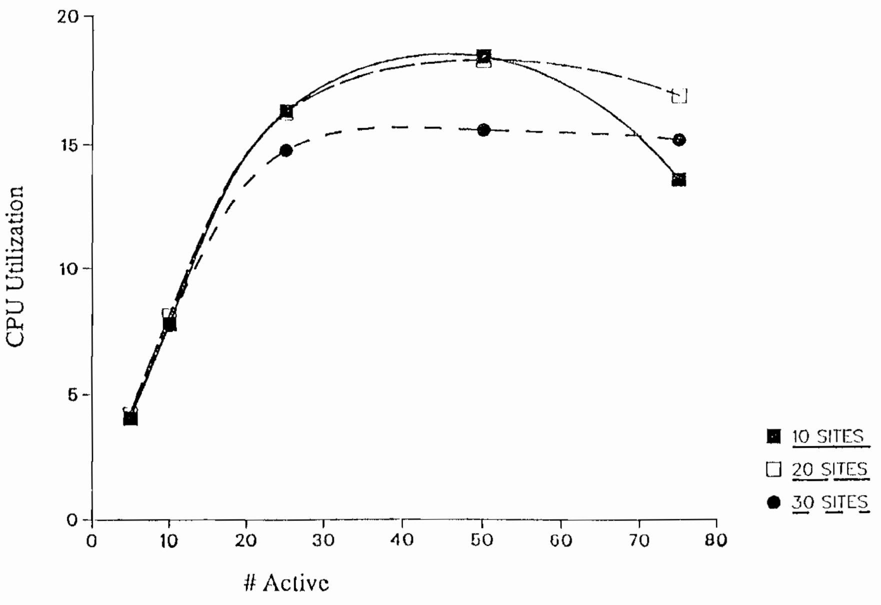

Figure 19

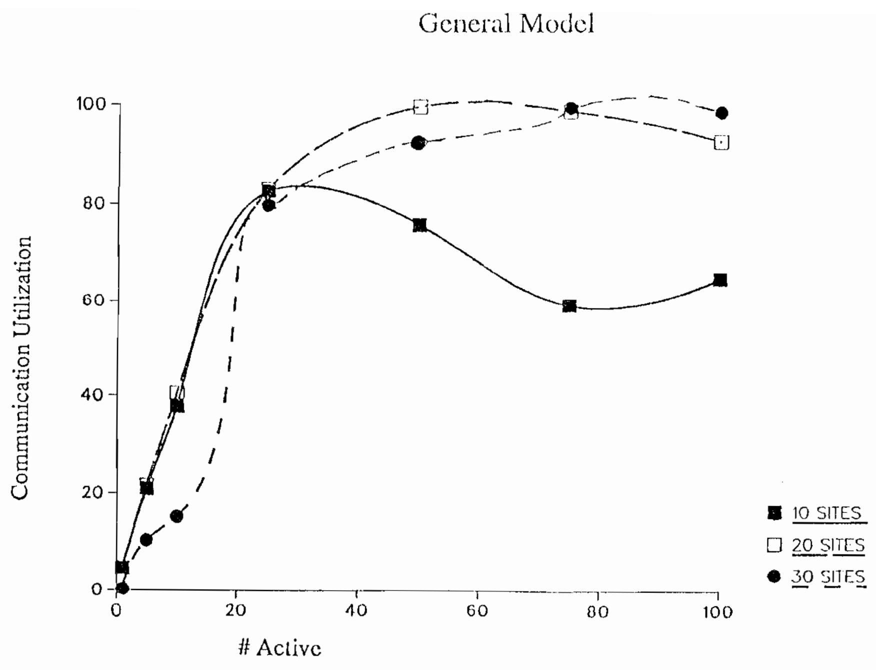

Figure 20

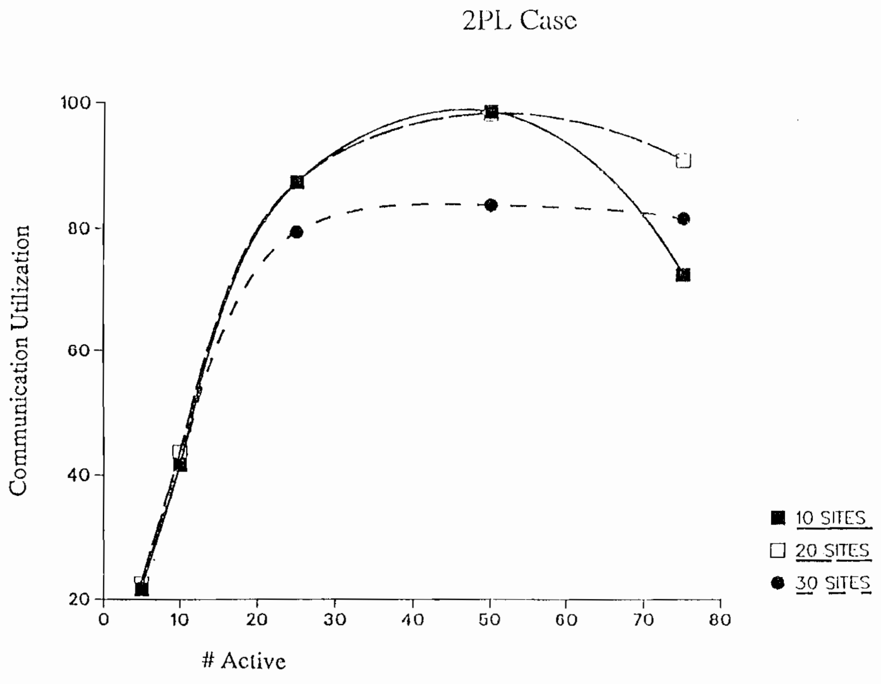

Figure 21

from global deadlocks is a viable option. In fact, the 2PL algorithm is implemented in a pilot version of the ADDS system [BOT86]. The bad news is that without any information about local concurrency control mechanisms, there is little hope a practical multidatabase concurrency control mechanism exists.

6. Conclusions

In this paper we have studied the performance characteristics of multidatabase concurrency control mechanisms using a general performance evaluation simulation model to compare the performance of two concurrency control algorithms. We studied the impact of the global multiprogramming level and the number of local sites on global transaction throughput, the number of global transactions rollbacks, and resource utilization for both algorithms. In terms of performance, we determined that the 2PL algorithm outperforms the transaction-graph algorithm in terms of both global transaction throughput, and the number of global transactions restarts. We also determined that the number of local sites has little impact on algorithm performance. Both algorithms exhibit similar resource utilization. Our results clearly indicate a general trend: The more information that is available to the multidatabase concurrency control, the better the algorithm performance.

References

[AD83] Agrawal,, R. Dewitt, D. "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," Technical Report No. 487, Computer Science Department, University of Wisconsin-Madison, 1983.

[ACL85] Agrawal, R., Carey, M., Livny, M., "Models for Studying Concurrency Control Performance: Alternatives and Implications," Proceedings of the ACM SIGMOD Int'l. Conf. on Management of Data, 1985.

[AGM87] Alonso, R., H. Garcia-Molina, K. Salem "Concurrency Control and Recovery for Global Procedures in Federated Database Systems," IEEE Data Engineering, 1987.

[BHG87] Bernstein, P., V. Hadzilacos, N. Goodman "Concurrency Control and Recovery in Database Systems" Addison-Wesley, 1987.

[BKST84] Breitbart, Y., L. Kemp, A. Silberschatz, G. Thompson "Performance Evaluation of a Simulation Model in Heterogeneous Database Environment," Proceedings of Trends and Applications Conference, NBS, 1984.

[BS88] Breitbart, Y., A. Silberschatz "Multidatabase Update Issues," Proceedings of ACM SIGMOD Conf., 1988.

[BLS91] Breitbart, Y., W. Litwin, A. Silberschatz "Deadlock Problems in a Multidatabase Environment," COMPCON91, Digest of Papers, Spring 91, 1991.

[BOT86] Breitbart, Y., P. L. Olson, G. R. Thompson "Database Integration Issues in A Distributed Heterogeneous Database System", Proceedings of the 2nd Database Engineering Conference, 1986.

[Car83] Carey, M. "An Abstract Model of Database Concurrency Control Algorithms" Proceedings of the ACM SIGMOD Int'l. Conf. on Management of Data, San Jose, 1983.

[CL86] Carey, M., H. Lu "Load Balancing in a Locally Distributed Database System," Proceedings SIGMOD, 1986.

[CS84] Carey, M., M. Stonebraker "The performance of Concurrency Control Algorithms for Database Management Systems", Proceedings of the Tenth International Conference on Very Large Data Bases, 1984.

[EGLT76] Eswaran, K., J. Gray, R. Lorie, I. Traiger "The notion of Consistency and Predicate Locks in a Database System," CACM, 19:11, 1976.

[GM79] Garcia-Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed Database," PhD Thesis, Computer Science Department, Stanford University, June, 1979.

[GRS91] D. Georgakopolous, M. Rusinkiewicz, and A. Sheth "On the serializability of multidatabase transactions through forced local conflicts," In *Proceedings of the Seventh International Conference on Data Engineering, Kobe, Japan,* 1991.

[GST83] Goodman, N., Suri, R., Tay, Y., "A simple Analytic Model for Performance of Exclusive Locking in Database Systems," Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Georgia, 1983.

[Gra79] Gray, J. "Notes on Database Operating Systems," Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[KJ85] Kohler, W., B. Jeng, "Performance Evaluation of Integrated Concurrency Control and Recovery Algorithm using a Distributed Transaction Processing Testbed," Technical Report # CS-85-133, Department of Electrical & Computing Engineering, University of Massachusetts, Amherst, 1985.

[Li87] Li, V., "Performance Model of Timestamp-Ordering Concurrency Control Algorithms in Distributed Database," IEEE Transaction on Computers, C-36, 9, 1987.

[LN82] Lin, W., Nolte, J. "Distributed Database Control and Allocation: Semiannual Report," Technical Report, Computer Corporation of America, Cambridge, Mass, January 1982.

[Lin84] Lindsay, B., et al. "Computation and Communication in R*," ACM Transaction on Computing Systems, 2, 1, 1984.

[LT89] Litwin W., H. Tirri "Flexible Concurrency Control using Value Date," in "Integration of Information Systems: Bridging Heterogeneous Databases," ed. A. Gupta, IEEE Press, 1989.

[MRBKS92] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. "The Concurrency Control Problem in Multidatabases: Characteristics and Solutions" In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. San Diego, Calif., 1992.

[Pu87] Pu, C. "Superdatabases: Transactions across Database Boundaries" IEEE Data Engineering, 1987.

[RS77] Ries, D., Stonebraker, M. "Effects of Locking Granularity on Database Management Systems Performance," ACM Transactions on Database Systems, 2(3), September 1977.

[SKS91] N. R. Soparkar, H. F. Korth, and A. Silberschatz. Failure-resilient transaction management in multidatabases. *IEEE Computer,* December 1991.

[Sto79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," IEEE Transactions on Software Engineering, SE-5, 3, 1979.

[TS84] Tay, Y., Suri, R. "Choice and Performance and Locking for Databases," Proceedings of the Tenth Int'l. Conf. on Very Large Databases, Singapore, 1984.