Parallelism and Concurrency Control Performance in Distributed Database Machines

Michael J. Carey
Miron Livny

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT — While several distributed (or 'shared nothing') database machines exist in the form of prototypes or commercial products, and a number of distributed concurrency control algorithms are available, the effect of parallelism on concurrency control performance has received little attention. This paper examines the interplay between parallelism and transaction performance in a distributed database machine context. Four alternative concurrency control algorithms are considered, including two-phase locking, wound-wait, basic timestamp ordering, and optimistic concurrency control. Issues addressed include how performance scales as a function of machine size and the degree to which partitioning the database for intra-transaction parallelism improves performance for the different algorithms. We examine performance from several perspectives, including response time, throughput, and speedup, and we do so over a fairly wide range of system loads. We also examine the performance impact of certain important overhead factors (e.g., communication and process initiation costs) on the four alternative concurrency control algorithms.

1. INTRODUCTION

During the past five years, it has become clear that multiprocessor database machines represent a viable solution to the problem of providing many users with access to large volumes of data. Distributed (or 'shared nothing') database machines appear especially attractive from the standpoint of scalability and reliability [Ston86, Bora88]. Commercially available systems of this type include Non-Stop SQL from Tandem [Borr88] and the DBC/1012 database machine from Teradata [Tera85]; working, high-performance research prototypes include Gamma at the University of Wisconsin [DeWi86] and Bubba at MCC [Alex88]. These systems differ in the extent to which parallelism is exploited to improve performance. Non-Stop SQL employs inter-transaction parallelism, allowing many transactions to execute simultaneously, but it does not employ intra-transaction parallelism (except in its parallel sort utility [Borr88]). In contrast, the Teradata, Gamma, and Bubba machines all employ intra-transaction parallelism (in addition to inter-transaction parallelism) in order to provide response time improvements for individual transactions. A recent performance evaluation of Non-Stop SQL for a standard multi-user Debit Credit workload showed linear throughput gains as the system size was increased from 2 to 32 processors [Tand88], demonstrating that inter-transaction parallelism can be very effective in a distributed database machine architecture. Recent measurements of Gamma showed that intra-transaction parallelism can be highly effective as well, as impressive single-user speedups were obtained for a variety of queries [DeWi88].

An issue that has only recently received attention is that of concurrency control algorithms and their performance in distributed database machines. Quite a large number of concurrency control algorithms have been proposed for use in distributed database systems, and they are all applicable for use in a database machine environment. The algorithms generally fall into one of three basic classes: *locking* algorithms [Mena78, Rose78, Gray79, Ston79, Trai82], *timestamp* algorithms, [Thom79, Bern80, Reed83], and *optimistic* (or certification) algorithms [Bada79, Schl81, Ceri82, Sinh85].[1] Concurrency control in the Non-Stop SQL, Teradata, Gamma, and Bubba systems is based on two-phase locking, but others have used or proposed optimistic algorithms in a database machine context (e.g., [Khos88, Lai88]). In the performance area, there have been a number of studies of distributed concurrency control algorithm performance. Relevant studies include [Garc79, Ries79, Balt82, Bhar82, Gall82, Kohl85, Oszu85, Lin83, Kohl85, Li87, Noe87, Care88]; we briefly review and critique them in [Care88]. These studies, however, did not focus on parallel transactions, so interesting questions remain open regarding the interplay between parallelism and concurrency control performance. For example:

(1) How does performance scale as a function of machine size and system load when intra-transaction parallelism is employed and data contention exists?

(2) How do the different concurrency control algorithm classes react to data partitioning and intra-transaction parallelism?

(3) How do key system overheads, such as communications and process startup costs, affect the different algorithm classes?

This research was partially supported by the National Science Foundation under grant IRI-8657323 and by grants from the Digital Equipment Corporation and the Microelectronics and Computer Technology Consortium (MCC).

[1] Bernstein and Goodman provide an excellent survey of many of the algorithms in [Bern81].

We are aware of only two studies that have addressed issues related to these questions. The first such study is [Bhid88], where the objective was to compare the performance of distributed versus shared memory based approaches to multiprocessor database machines in the context of transaction processing; the performance of these two alternative architectures was compared assuming the use of two-phase locking. This was a nice first step towards a better understanding of how parallelism affects locking under different communications costs and system loads. However, it did not address scaling issues or other concurrency control algorithms. The second relevant study is [Jenq89], which looked at how the performance of locking might scale in a Bubba-like database machine. This study focused on a specific workload (based on analyzing an actual order entry application) and looked at a wide range of machine sizes; for this workload they found that locking was a crucial factor in determining the performance of such a system. Their results also indicated that a larger machine can lead to better performance, which is expected since they increased the database size (thus reducing data contention) along with the machine size. Both of these studies used timeouts to avoid deadlocks, which can cause performance problems if data contention is significant [Agra87b].[2]

In this paper, we report on a study aimed at addressing the questions raised above. The study employs a performance model based on the distributed DBMS simulator developed for [Care88]. The model is reasonably detailed, capturing the main elements of a distributed database machine, including physical resources (disks, CPUs, and the interconnection network), parallel transaction execution, and the placement of data (i.e., data partitioning). The design of the model was guided by previous results on the importance of realistic concurrency control modeling assumptions, particularly regarding system resources [Agra87a]. Using this model, we investigate the performance impact of changes in the size of the database machine, the degree of data partitioning, the system load, and system-related overheads. Four concurrency control algorithms are examined in this study, including two variants of two-phase locking, a timestamp-based algorithm, and an optimistic algorithm; the algorithms span a wide range of characteristics in terms of how conflicts are detected and resolved.

The remainder of the paper is organized as follows: Section 2 describes our choice of concurrency control algorithms. The structure and characteristics of our performance model and its implementation are reviewed in Section 3. Section 4 presents our performance experiments and the associated results. Finally, Section 5 summarizes the main conclusions of this study.

2. CONCURRENCY CONTROL ALGORITHMS

In this study we examine four algorithms that span the basic design space of available concurrency control mechanisms. We briefly review the main features of each of the algorithms in this section. Before doing so, however, we need to describe the transaction structure assumed in the study. Since replicated data is not considered in this paper, we ignore replication-related considerations throughout this section.

[2] In fact, [Jenq89] noted that the timeout interval was a critical and sensitive performance factor in their experiments.

2.1. The Structure of a Parallel Transaction

Each transaction has a master or *coordinator* process that runs at its node of origination. The coordinator process, in turn, sets up a collection of *cohort* processes that are responsible for performing the actual processing involved in running the transaction. In particular, there is one such cohort at each node where data is accessed by the transaction.[3] Transaction completion is controlled through a centralized two-phase commit protocol [Gray79], with the coordinator process running the protocol. This same protocol is used for all of the concurrency control algorithms studied. Finally, whether the cohorts of a transaction execute sequentially or in parallel depends on the query execution model of interest. We will discuss cohort placement and query execution further in Section 3 when we describe the details of our workload model.

2.2. Distributed Two-Phase Locking (2PL)

The first algorithm is the distributed two-phase locking algorithm described in [Gray79]. Cohorts set read locks on items that they read, converting their read locks to write locks on items that need to be updated. As usual, read locks can be shared but write locks cannot. Locks are set dynamically, as the cohort executes, with the cohort blocking until the next item to be accessed has been successfully locked. Locks are held until the transaction has successfully committed or aborted. Of course, deadlocks are possible. Local deadlock detection occurs whenever a cohort blocks. Global deadlocks are handled through a "Snoop" scheme similar to that of Distributed INGRES [Ston79]: A "Snoop" process periodically gathers up waits-for information from all nodes, checking for global deadlocks. Each node takes a turn being the "Snoop" node, and then passes the job on to the next node, so the "Snoop" responsibility rotates among the nodes in a round-robin fashion. Deadlocks are resolved by aborting the transaction with the most recent initial startup time among those involved in the deadlock.

2.3. Wound-Wait (WW)

The second algorithm is the distributed wound-wait locking algorithm of [Rose78]. It differs from 2PL in its handling of the deadlock problem, as deadlocks are prevented through the use of timestamps. Each transaction is numbered according to its initial startup time, and younger transactions are prevented from making older ones wait. If a cohort of an older transaction requests a lock, and if the request would lead to the cohort waiting for a cohort of a younger transaction, then the younger transaction is "wounded" — it is aborted unless it is already in the second phase of its commit protocol (in which case the "wound" is not fatal, and is simply ignored). Younger transactions are permitted to wait for older transactions, however.

2.4. Basic Timestamp Ordering (BTO)

The third algorithm is the basic timestamp ordering algorithm of [Bern80b, Bern81]. Like wound-wait, it employs transaction startup timestamps, but it uses them differently. BTO associates

[3] The full model that we support is somewhat more general than this [Care88], but one cohort per transaction per node will suffice here.

read and write timestamps with all recently accessed data items and requires that conflicting data accesses be performed in timestamp order. Transactions that attempt out-of-order accesses are aborted (except in the case of write-write conflicts, where the Thomas write rule applies [Bern81]). The BTO algorithm interacts with the two-phase commit protocol as follows [Bern81]: Writers keep their updates in a private workspace until commit time. Granted writes for a given data item are queued in timestamp order without blocking the writers, and they are processed as the writers commit. Accepted read requests for pending writes must also be queued, blocking the readers, in order to ensure that readers do not read uncommitted data. Effectively, a write request locks out subsequent reads with later timestamps until the write actually becomes visible at commit time.

2.5. Distributed Certification (OPT)

The fourth algorithm is the distributed, timestamp-based, optimistic concurrency control algorithm of [Sinh85], which operates by exchanging certification information during the commit protocol.[4] For each data item, a read timestamp and a write timestamp are maintained. Cohorts may read and update data items freely, storing any updates into a local workspace until commit time. For each read, the cohort must remember the version identifier (i.e., write timestamp) associated with the item when it was read. Then, when all of the cohorts of the transaction have completed their work, and have reported back to the coordinator, the transaction is given a globally unique timestamp. This timestamp is sent to each cohort in the "prepare to commit" message, and it is used to locally certify all of its reads and writes as follows: A read request is certified if (i) the version that was read is still the current version of the item, and (ii) no write with a newer timestamp has already been locally certified. A write request is certified if (i) no later reads have been certified and subsequently committed, and (ii) no later reads have been locally certified already. The term "later" refers to timestamp time here, so these conditions are checked using the timestamp given to the transaction when it started the commit protocol. These local certification computations are performed in a critical section.

2.6. Some Observations

The four algorithms that we have selected span the three major algorithm classes, and they represent a fairly wide range of conflict detection and resolution methods and times. 2PL prevents conflicts as they occur using locking, resolving global deadlocks via a centralized deadlock detection scheme. WW is similar to 2PL, except that it uses timestamps and aborts to prevent deadlocks. BTO uses timestamps to order transactions a priori, aborting transactions when conflicting, out-of-order accesses occur; read requests must occasionally block when they request data from pending, uncommitted updates. OPT only checks for conflicts when a transaction is ready to commit, and it uses aborts to resolve them.

[4] Actually, two such algorithms are proposed in [Sinh85]. We use their first algorithm here, as it is the simpler of the two.

3. MODELING A DISTRIBUTED DBMS

As mentioned in Section 1, we developed a distributed DBMS model for studying concurrency control algorithms and performance tradeoffs in [Care88]. The model is equally applicable to a distributed database machine, so we use it here as well. While a complete description of the model is given in [Care88], we summarize it here (with a database machine orientation) for completeness and to aid the reader in interpreting the results. Figure 1 shows the general structure of the model. There are two types of nodes, *host nodes* and *processing nodes*. Each host node has four components: a *source*, which generates transactions and also maintains transaction-level performance information for the node, a *transaction manager*, which models the execution behavior of transactions, a *concurrency control manager*, which implements the details of a particular concurrency control algorithm; and a *resource manager*, which models the CPU and I/O resources of the node. The processing nodes are similar, but they lack the source component; thus, they do not contribute transactions to the workload. In addition to these four per-node components, the model also has a *network manager*, which models the behavior of the communications network. The interfaces between components were designed to support modularity, making it easy to replace one component (e.g., the concurrency control manager) without affecting the others. We describe each component in turn in this section, preceded by a discussion of how the database itself is modeled. We omit details related to how replicated data is modeled [Care88], as they are not relevant to this study.

3.1. The Database Model

We model a distributed database machine as containing a collection of *files*. A file can be used to represent an entire relation, or it can represent a partition of a relation when relations are horizontally partitioned [Ries78] as is common in distributed database machines. Table 1 summarizes the parameters of the database model, which include the number of host and processing nodes, the number of files in the database, and the size of each file. As indicated in the table, files are modeled at the page level. The mapping of files to nodes is specified via the parameter

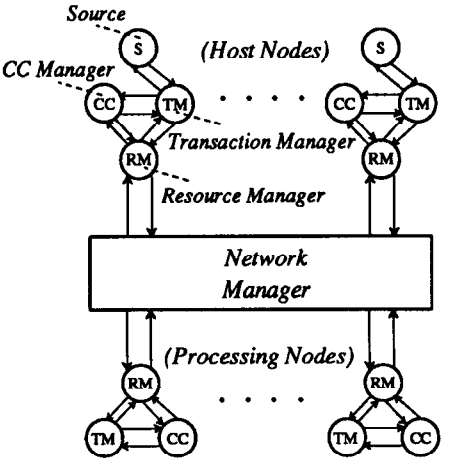


Figure 1: Distributed DB Machine Model.

FileLocations, a boolean array in which $FileLocations_{ij}$ is true if file i resides at node j. In general, files can reside at either host or processing nodes, although we place them only at processing nodes throughout this study.

3.2. The Source

The source is the component responsible for generating the workload for a host node. The workload model used by the source characterizes transactions in terms of the files that they access and the number of pages that they access and update in each file. Table 2 summarizes the key parameters of the workload model for a node; each host node has its own set of values for these parameters. The *NumTerminals* parameter specifies the number of terminals attached to the node, and the *ThinkTime* parameter is the mean of an exponentially distributed think time between the completion of one transaction and the submission of the next one at a terminal. *NumClasses* gives the number of transaction classes for the node.

The *ClassFrac* parameter specifies the fraction of the node's terminals that generate transactions of a given class. The remaining per-class parameters characterize transactions of the class as follows: *ExecPattern* specifies the execution pattern, either sequential or parallel, for transactions. (More will be said about this shortly.) *FileCount* is the number of files accessed, and $FileProb_i$ gives the probability distribution for choosing the actual files that the transaction will access. The next two parameters determine the file-dependent access characteristics for transactions of the class, including the average number of pages read and the probability that an accessed page will be updated. The last parameter specifies the mean number of instructions required for transactions of the class to process a page of data when reading or writing it. The actual number of pages accessed ranges uniformly between half and twice the average, and the per-page instruction count is exponentially distributed.

Parameter	Meaning
NumHostNodes	Number of host nodes
NumProcNodes	Number of processing nodes
NumFiles	Number of files in the database
$FileSize_i$	Number of pages in file i
$FileLocations_{ij}$	Placement of files at nodes

Table 1: Database Model Parameters.

Parameter	Meaning
Per-Host-Node Parameters	
NumTerminals	Number of terminals attached to node
ThinkTime	Think time for the terminals
NumClasses	Number of transaction classes
Per-Class Parameters	
ClassFrac	Fraction of terminals of this class
ExecPattern	Sequential or parallel cohort execution
FileCount	Number of files accessed
$FileProb_i$	Access probability for file i
$NumPages_i$	Average number of file i pages read
$WriteProb_i$	Write probability for file i pages
InstPerPage	Instruction count for page processing

Table 2: Workload Model Parameters for a Host Node.

3.3. The Transaction Manager

Each transaction in the workload has the general structure described in Section 2.1. The coordinator resides at the host node where the transaction originated. A transaction has one cohort at each node where it needs to access data, so each cohort makes a sequence of read and write requests to one or more files that are stored locally. A transaction can execute in either a sequential or parallel fashion, depending on the execution pattern of the transaction class. Cohorts in a sequential transaction execute one after another, whereas cohorts in a parallel transaction are started together and execute independently until commit time. A sequential transaction can be thought of as representing a series of remote procedure calls, which is how Non-Stop SQL executes most multi-node transactions [Borr88], while a parallel transaction can be thought of as modeling the kind of parallel query execution employed in the Gamma [DeWi86], Bubba [Alex88], and Teradata [Tera85] database machines.

To understand how transaction execution is modeled, let us follow a typical parallel transaction from beginning to end. When a transaction is initiated, the set of files and data items that it will access are chosen by the source. The coordinator is then loaded at the originating host node, and it sends "load cohort" messages to initiate cohorts at the appropriate processing nodes. Each cohort makes a series of read and write accesses. A read access involves a concurrency control request to get access permission, followed by a disk I/O to read the page, followed by a period of CPU usage for processing the page. Write requests are the same except for the disk I/O; the I/O activity for writes takes places asynchronously after the transaction has committed.[5] A concurrency control request for a read or write access is always granted in the case of the OPT algorithm, but this is not the case for the other algorithms. When a concurrency control request cannot be granted immediately due to a conflict, the cohort will wait until the request is granted by the concurrency control manager. If the cohort must be aborted, the concurrency control manager notifies the transaction manager, which then invokes the abort protocol. Once the transaction manager has finished aborting the transaction, it delays the coordinator for a period of time before letting it attempt to rerun the transaction; as in [Agra87a], we use one average transaction response time (as observed at the coordinator node) for the length of this period.

3.4. The Resource Manager

The resource manager can be viewed as a model of the node's operating system. It manages the physical resources of the node, including its CPU and its disks.[6] The resource manager provides CPU and I/O service to the transaction manager and concurrency

[5] We assume sufficient buffer space to allow the retention of updates until commit time, and we also assume the use of a log-based recovery scheme where only log pages must be forced prior to commit. We do not model logging, as we assume it is not the bottleneck.

[6] Note: The resource manager employed in this study does not capture details related to managing the buffer resources of a node. While modeling buffering in detail would certainly lead to different absolute results, we do not expect that doing so would significantly affect the general conclusions of this study. (If anything, we suspect that the conclusions would be even stronger; we plan to verify this conjecture in the future.)

control manager, and it also provides message-sending services (which involve using the CPU resource). The transaction manager uses CPU and I/O resources for reading and writing disk pages, and it also sends messages. The concurrency control manager uses the CPU resource for processing requests, and it too sends messages.

The parameters of the resource manager are summarized in Table 3. Each node has one CPU, which executes instructions at the rate given by *CPURate*, plus *NumDisks* disks. The CPU service discipline is first-come, first-served (FIFO) for message service and processor sharing for all other services, with message processing being higher priority. Each of the disks has its own queue, which it serves in a FIFO manner; the resource manager assigns a disk to serve a new request randomly, with all disks being equally probable, so our I/O model assumes that the files stored at a node are evenly balanced across its disks. Disk access times for the disks are uniform over the range [*MinDiskTime*, *MaxDiskTime*]. Disk writes are given priority over disk reads (to ensure that the system keeps up with the demand for asynchronously writing updated pages back to disk after the transaction has committed). The parameter *InstPerUpdate* models the CPU overhead associated with initiating a disk write for an updated page. The parameter *InstPerStartup* captures the CPU overhead associated with starting up a new process (e.g., a new cohort to operate on behalf of some transaction). Finally, *InstPerMsg* captures the CPU cost of protocol processing for sending or receiving a message.

3.5. The Network Manager

The network manager encapsulates the model of the interconnection network. Our network model is currently quite simplistic, acting just as a switch for routing messages from node to node. This is because our experiments assume a fast network, where the actual wire time for messages is neglible, although we do take the CPU overhead for message processing into account at both the sending and receiving nodes. This cost assumption has become common in the analysis of locally distributed systems, as it has been found to provide reasonably accurate performance results despite its simplicity [Lazo86].

3.6. The Concurrency Control Manager

The concurrency control manager captures the semantics of a given concurrency control algorithm, and it is the only module that must change from algorithm to algorithm. It handles concurrency control requests made by the transaction manager, including read and write access requests, requests to get permission to commit a transaction, and certain management requests (e.g., to initialize and terminate coordinator and cohort processes). This study employs four different concurrency control managers, one for each of the concurrency control algorithms described in Section 2.

Parameter	Meaning
CPURate	CPU instruction rate for node
NumDisks	Number of disks attached to node
MinDiskTime	Minimum disk access time
MaxDiskTime	Maximum disk access time
InstPerUpdate	Instruction count to start a disk write
InstPerStartup	Instruction count to start a process
InstPerMsg	Instruction count to send/receive a message

Table 3: Resource Manager Parameters.

The concurrency control manager has a variable number of parameters. One parameter, *InstPerCCReq*, specifies the CPU overhead for processing a read or write access request; this parameter is present in all of our algorithms. Each algorithm then has zero or more additional parameters. Of the algorithms studied in this paper, only 2PL uses another parameter. Its second parameter is *DetectionInterval*, which determines the amount of time that a node should wait, after becoming the next "Snoop" node, before gathering global waits-for information and performing global deadlock detection.

4. EXPERIMENTS AND RESULTS

In this section, we present performance results for the four concurrency control algorithms of Section 2 under various assumptions about the size of the system, the partitioning of the database, the system load, and the CPU costs for sending and receiving messages and initiating processes. The simulator used to obtain the results was written in the Modula-2-based DeNet simulation language [Livn88]. We describe the performance experiments and results after a brief discussion of the performance metrics and parameter settings used.

4.1. Metrics and Parameter Settings

We consider four main performance metrics in this paper. The first metric is the transaction response time, measured from the time when a transaction originates at a node until it finally completes successfully. The second metric used is the throughput (or transaction completion rate) of the system. The third and fourth metrics of interest are the response time and throughput speedups obtained by adding resources and/or parallelism to the system. The speedup metrics will be more carefully explained when we present the results. We also employ several additional metrics to aid in interpreting the results. CPU and disk utilizations will be presented in some places. Other metrics of interest include the average blocking time (for 2PL) and a metric that we call the *abort ratio*. This last metric gives the average number of times that a transaction has to abort per commit, and is computed by dividing the number of transaction aborts by the number of transaction commits.

Table 4 gives the values of the key simulation parameters in our experiments. We describe them briefly here; those that vary are mentioned again in the appropriate experiment descriptions. We consider machine configurations with one host node and 1, 2, 4, and 8 processing nodes in this study, using the 8-node configuration in those experiments where the machine size is held constant.[7] The database itself consists of 8 relations with 8 partitions apiece, modeled by 64 files, and all partitions are the same size. The partition size is set at 300 or 1200 pages in our

[7] In addition to an 8-node configuration, we also ran several experiments with 16-node and 32-node configurations (with larger update transactions). Since the trends there were similar, and we have limited space here, we present only the 8-node results.

experiments, so the database size is either 19,200 or 76,800 pages of data depending on the experiment in question. The mapping of relations' partitions to nodes is varied in order to control the amount of parallelism assumed; we will describe this mapping shortly. There are 128 terminals, which are attached to the host node in order to keep the structure of the system's workload uniform over the various machine sizes. The mean terminal think time is varied over the range from 0 to 120 seconds in order to vary the load on the system.

In terms of the workload, a transaction accesses data from each partition of one of the 8 relations. Each transaction reads an average of 8 pages from each of the relation's partitions, and each page is updated with a probability of 1/4. Transactions thus involve an average of 64 reads, and they do an average of 8 writes.[8] This transaction size was chosen as being small enough to retain somewhat of a "transaction processing flavor" without being so small as to make parallel execution seem ridiculous.[9] The corresponding file sizes were selected so as to provide interesting levels of data contention. Finally, it takes a transaction an average of 8K CPU instructions to process each page that it reads or writes. In all experiments, the choice of the particular file to be accessed by a transaction is determined by its terminal of origin. The 128 terminals attached to the host node are divided into groups of 16, with terminals in each group generating transactions that access a common relation.

Continuing through the parameters in Table 4, the host CPU is 10 MIPS, which is ten times as fast as the 1 MIPS processing nodes so that the host won't limit system performance. Each node has two disks, and each disk has an average access time of 20 milliseconds. Initiating a disk write for an updated page takes 2K instructions. The mean number of instructions for starting up a new process is set to 0, 2K, or 20K, so we consider the effects of both lightweight and heavyweight processes. The mean instruction path length for message protocol processing on each end is set to 0, 1K, or 4K instructions. The concurrency control CPU overhead is assumed to be negligible, for all algorithms, compared to the 8K instruction count for page processing.[10] Finally, the global deadlock detection interval for 2PL and O2PL is 1 second.

Parameter	Value(s)
NumHostNodes	1 host
NumProcNodes	1, 2, 4, 8 nodes (8 when fixed)
NumFiles	64 files (8 relations, 8 partitions each)
$FileSize_i$	300, 1200 pages/file
NumTerminals	128 terminals (attached to host)
ThinkTime	0-120 seconds
FileCount	8 files
$FileProb_i$	1.0
$NumPages_i$	8 pages
$WriteProb_i$	1/4
InstPerPage	8K instructions
CPURate	host — 10 MIPS, nodes — 1 MIPS
NumDisks	2 disks/node
MinDiskTime	10 ms
MaxDiskTime	30 ms
InstPerUpdate	2K instructions
InstPerStartup	0, 2K, 20K instructions (2K when fixed)
InstPerMsg	0, 1K, 4K instructions (1K when fixed)
InstPerCCReq	negligible (0)
DetectionInterval	1 second

Table 4: Simulation Parameter Settings.

The I/O and CPU cost parameter values for the experiments reported here were chosen so that, messages aside, the processing nodes operate in an I/O-bound region. In particular, when the disks are fully utilized, 80-90% of the CPU capacity of the processing nodes is utilized; this makes the system slightly I/O-bound.

In the remainder of this section, we report on the results of our experiments. The presentation of the results is divided into three main sections. In the first section, we examine the potential gains offered by the combination of increasing the number of nodes in the system together with partitioning the data for parallelism. Here, we will see the extent to which the various concurrency control algorithms are able to take advantage of the opportunities available in a distributed database machine architecture. In the second section, we fix the size of the system at eight nodes, and we look more closely at how partitioning (i.e., parallelism) alone impacts performance under different system loads. In the third section, we look at how certain system overheads affect the gains due to partitioning and parallelism.

4.2. The Impact of Machine Size and Parallelism

In this section, we vary the size of the database machine while keeping the workload constant. We do so in order to study the performance gains provided by scaling up a database machine and the way that concurrency control influences these gains. As we vary the number of nodes in the machine, we also partition the data across the nodes in order to enable intra-transaction parallelism to be employed. Specifically, three different system configurations are considered here:

1-Node System: In this case, there is a single database machine node, and consequently no partitioning is used. Each of the relations R_i, $1 \le i \le 8$, has all of its partitions F_{ij}, $1 \le j \le 8$, stored at the single node S_1, and all transactions execute on this node.

4-Node System: In this case, the machine has four nodes. Each relation R_i has partitions F_{i1} and F_{i2} stored at node S_1, F_{i3} and F_{i4} at S_2, F_{i5} and F_{i6} at S_3, and F_{i7} and F_{i8} at S_4. Here, transactions consist of four cohorts running in parallel on all four nodes.

8-Node System: In this case, the machine has eight nodes. Each relation R_i has partition F_{ij} stored at node S_j in this case, so transactions consist of eight cohorts running in parallel on all eight nodes.

[8] Note that with this workload model, the nature of transaction access streams is independent of data placement and machine size, unlike [Bhid88, Jenq89].

[9] We also ran experiments with other transaction sizes (e.g., 32 reads). The basic trends were similar, so we present only the 64-read results in order to limit the number of graphs needed.

[10] Alternatively, the concurrency control overhead can simply be viewed as part of the 8K per-page instruction count.

We examine the performance gains due to scaling by comparing the 4-node and 8-node throughput and response time results with those obtained in the 1-node case. In order to push the system into a region of operation where data contention is significant, we set the $FileSize_i$ parameter to 300 pages per file. As a result, the overall size of the database is 19,200 pages (or 153 MB, assuming a page size of 8K bytes), with the average transaction reading 64 of them. As described earlier, we assume 128 terminals with their think time varying from 0 to 120 seconds. Finally, for this experiment we assume a value of 2K instructions for *InstPerStartup*, the CPU overhead for starting up a cohort; 1K instructions is assumed for *InstPerMsg*, the CPU overhead on each end for sending and receiving a message.

Figures 2 and 3 present the performance results for the 1-node and 8-node cases. Figure 2 shows the throughputs obtained with the four different concurrency control algorithms in these cases, and Figure 3 shows the associated response time results. In addition to the results for the four algorithms, we also show results that we refer to as the NO_DC (no data contention) results. The NO_DC results, which can be viewed as results for 2PL with an infinitely large database, show the performance that would be obtained if data contention were not a factor. Examining the relative ordering of the algorithms, we see that 2PL outperforms BTO, which outperforms WW, which in turn outperforms OPT. These differences are evident only in the lower think time range, where the system load is sufficient to generate a significant amount of data contention. The NO_DC curve displays the best performance, of course, with the distance between this curve and the various algorithm curves showing the extent to which each algorithm suffers due to data contention. In particular, we see that all four of the algorithms thrash due to data contention under high loads. (This is consistent with the results of our previous centralized and distributed concurrency control performance studies [Agra87a, Care88].)

Figures 4 and 5 present the 8-node speedups resulting from the throughput and response time data of Figures 2 and 3. These speedups were computed as ratios of the 8-node and 1-node performance metrics for each algorithm. In Figure 4, we see the throughput speedups for NO_DC behaving as one would expect. When the think time is small, the system is heavily loaded, so increasing the number of nodes in the system by a factor of eight enables the system to deliver very close to eight times the throughput. As the think time is increased, reducing the load, the throughput speedup decreases. This happens when there are not enough transactions running simultaneously to keep all of the nodes fully utilized; the speedup drops towards one since the think time will eventually reach a point where there is usually just one transaction running in the system. The throughput speedups for the various concurrency control algorithms follow the same general trend, but two differences should be noted. First, the various algorithms actually have somewhat better throughput speedups than NO_DC; they even exceed eight in some cases. This is because going from sequential execution in a 1-node system to parallel execution in an 8-node system also leads to a reduction in data contention, reducing the number of aborts and increasing the degree to which the system resources can be utilized usefully. Second, 2PL has the smallest additional speedup, and OPT generally has the greatest additional speedup, with the other algorithms falling between these two. The reason for this is evident from the results in Figure 2: The worse an algorithm performs due to data contention in the 1-node case, the more room it has for improvement due to reduced contention in the 8-node case.

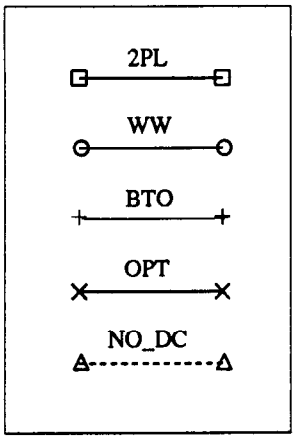
Looking at Figure 5, we see how the response time speedup varies with think time for the various concurrency control algorithms and for NO_DC. The values at the lowest and highest think times are quite intuitive. When the think time is low, and the system is heavily loaded, the 8-node system provides a response time reduction of about 7.5. Note that this reduction is not due to parallelism, as intra-transaction parallelism does not improve response time under high loads.[11] Rather, it is due to the fact that the 8-node system is eight times as powerful as the 1-node system. When the think time is large, the improvement in response time ends up in a similar range. It will actually end up dropping to about 5.3 due to variations in the size of the cohorts; since all cohorts must complete before a transaction can commit, its response time is determined by the length of its longest cohort.[12] The speedup at this end of the think time spectrum is indeed due to parallel execution. Under light loads, a transaction experiences little competition for the disks or the CPU at any of the nodes, so splitting it into parallel cohorts leads to a significant speedup.

Turning our attention to the intermediate think time range in Figure 5, we find that the response time speedups for the four algorithms, and also for NO_DC, are quite remarkable. The NO_DC speedup exceeds 100 (i.e., a factor of 100 improvement) in this range, and the speedups for the various concurrency control algorithms are even better. While this surprised us at first, there is in fact a sensible explanation: At intermediate loads, the system gains both from having multiple nodes (like at high loads) and from parallellism (like at low loads). In this range, the system's behavior becomes similar to that of a moderately to heavily loaded open queuing system — with its familiar, non-linear relationship between utilization and response time. Figures 6 and 7 present the disk and CPU utilizations underlying these results. Note how rapidly the utilizations drop in the 8-node case as compared to the 1-node case, leading to the large response time speedups seen here. Finally, the way that the 1-node and 8-node response time curves decrease relative to one another in Figure 3 also indicates why the speedups are so large in this range.

We repeated this experiment with the 4-node configuration, again comparing the results to the 1-node case via speedups. While space limitations prevent us from including the figures, the results can be summarized as follows: Qualitatively, the throughput and response time speedup curves for the 4-node case were very similar to the 8-node curves of Figures 4 and 5. Quantitatively, the maximum throughput speedup for the various concurrency control algorithms was slightly greater than four, and

[11] Splitting a transaction into eight pieces indeed makes each piece one-eighth as long, but under high loads each node becomes eight times as loaded (thus nullifying any gain due to parallelism).

[12] Recall from Section 3 that actual cohort sizes vary around their mean size; they actually access between 4 and 12 pages per partition. With 8 cohorts per transaction, there is a good chance of a transaction having a cohort of size 12, limiting the expected speedup to 64/12 (rather than 64/8), or 5.33.



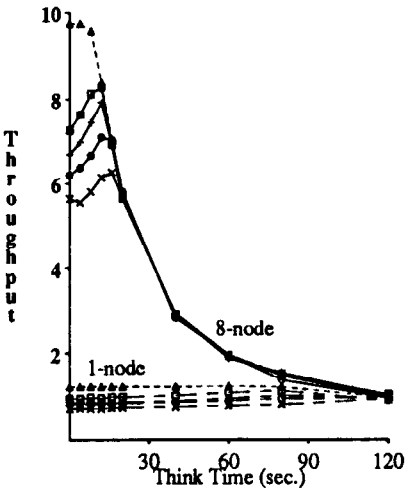


Figure 2:Throughput (1-node & 8-node).
(FileSize = 300)

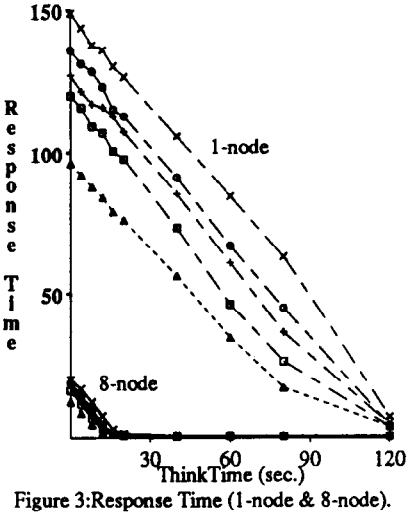


Figure 3:Response Time (1-node & 8-node).
(FileSize = 300)

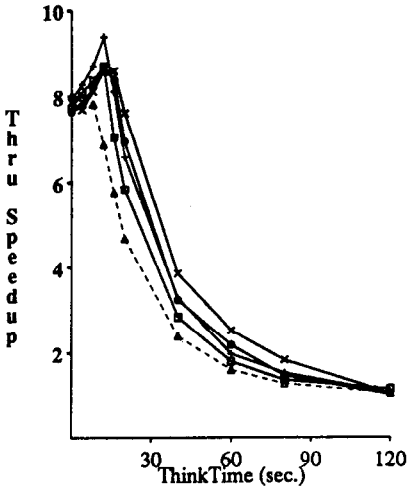


Figure 4:Throughput Speedup (8-node/1-node).
(FileSize = 300)

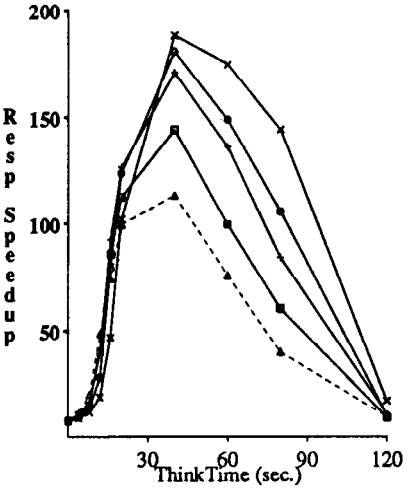


Figure 5:Response Speedup (8-node/1-node).
(FileSize = 300)

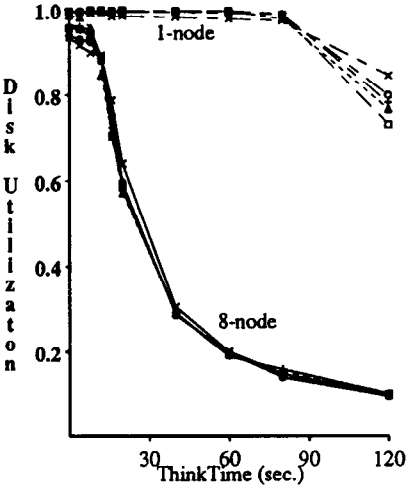


Figure 6:Disk Utilization (1-node & 8-node).
(FileZise = 300)

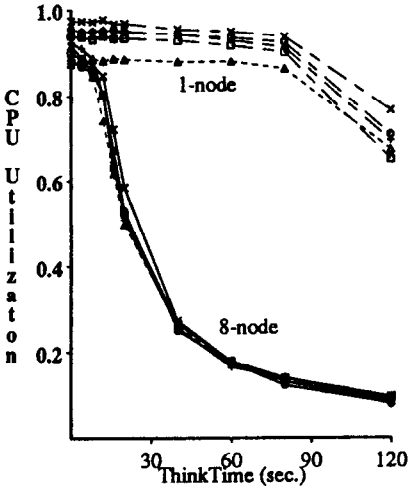


Figure 7:Cpu Utilization (1-node & 8-node).
(FileSize = 300)

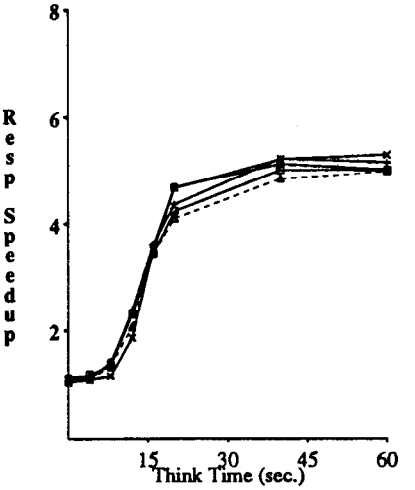


Figure 8:Response Speedup (8-way/1-way).
(FileSize = 1200)

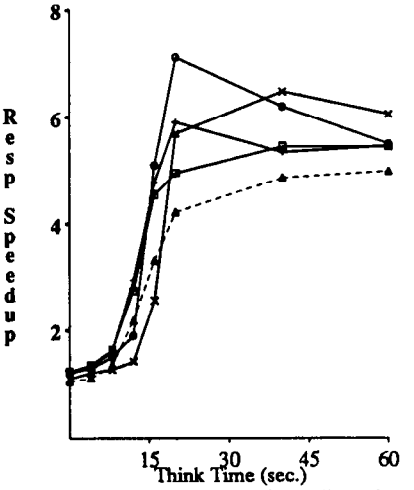


Figure 9:Response Speedup (8-way/1-way).
(FileSize = 300)

the response time speedup was also limited at the extremes by 4-way rather than 8-way parallelism. Under medium loads, the response time speedup for NO_DC in the 4-way case reached almost 60, with the various concurrency control algorithms enjoying even larger speedups in this region.

4.3. The Impact of Parallelism

In this section, we examine the performance of the system as a function of data partitioning and intra-transaction parallelism while holding the size of the system constant. Since a transaction accesses data from every partition of one relation, we control the level of parallelism here by varying the placement of partitions. We consider two alternatives for placing the partitions F_{ij}, $1 \le j \le 8$, of the eight relations R_i, $1 \le i \le 8$, on nodes S_i, $1 \le i \le 8$:

1-Way Partitioning: In this case, no partitioning is used. All eight partitions F_{ij} of relation R_i reside at node S_i, so a transaction that accesses R_i runs sequentially at node S_i.

8-Way Partitioning: In this alternative, each relation R_i is split across all eight nodes, so transactions that access R_i have eight cohorts that run in parallel. Here, R_i's partitions F_{ij}, $1 \le j \le 8$, are stored at nodes S_i through $S_{(i \bmod 8)+7}$.

Most of the other parameter values used in this section are the same as those in the previous experiment; the only difference is that here we use both the smaller and larger database sizes.

Figures 8 and 9 show the response time improvements obtained with 8-way partitioning (and thus 8-way parallelism) as a function of the mean terminal think time. Figure 8 presents the results for the larger database size, and Figure 9 is for the smaller database. At low think times, there is no improvement in either figure due to the high load; parallelism simply doesn't help here, as would be expected. Once the load is lessened by increasing the think time, we indeed see a significant response time improvement for all of the algorithms. For large think times, we see speedups due to parallelism of about five (as discussed in the previous experiment). Again, we also see that all of the concurrency control algorithms actually have better speedups than in the NO_DC case due to the fact that moving to parallelism relieves data contention. 2PL has the largest speedup among the algorithms at low think times (under heavy loads), whereas OPT has the largest speedup at the highest think times. These effects are subtle in the case of the larger database, but they are clear in the smaller case, where data contention is more significant.

Figures 10 and 11 show the percentage response time degradations for the various algorithms (computed relative to the NO_DC response times) for the smaller database size. These figures illustrate the extent to which data contention leads to a response time loss for each algorithm. Figure 10 shows the results for 8-way partitioning, while Figure 11 shows the results without partitioning. In both cases, the ordering of the algorithms is the same as in the first experiment, with 2PL providing the best performance (i.e., the smallest loss relative to NO_DC), followed by BTO, and then WW, with OPT providing the worst performance (i.e., the biggest relative loss). Figures 12 and 13 show the associated abort ratios. It is clear from these figures that the relative performance of the algorithms is consistent with their abort ratios (modulo the fact that WW aborts are cheaper than OPT aborts because they occur earlier). The corresponding results that we obtained using the larger database size displayed similar trends, with approximately the same relative algorithm differences but smaller absolute degradation values.

Comparing Figures 10 and 11, we see that the differences in degradation among the algorithms are more pronounced in the 8-way case. A comparison of the 8-way and 1-way results reveals that 2PL actually benefits from 8-way parallelism under fairly high loads, i.e., the gap between 2PL and NO_DC is less in the 8-way case than in the 1-way case. This is due to the fact that locks are held for shorter periods, reducing data contention there. For example, at a think time of 12 seconds, the average blocking time for 2PL in the 1-way case turns out to be 60% higher than that of the 8-way case. In contrast, OPT cannot take full advantage of parallelism because of its reliance on aborts as the sole conflict resolution mechanism. In fact, unlike 2PL, the gap between OPT and NO_DC is larger in the 8-way case than in the 1-way case, particularly for the smaller database size. This is because, by letting all conflicting transactions execute simultaneously rather than blocking some of them, OPT cannot benefit from the shorter blocking times that the 8-way case implies for 2PL; in addition, aborts are quite expensive in the 8-way case. The degradation behavior of the other algorithms relative to NO_DC is in between that of 2PL and OPT. The degradation of BTO shrinks somewhat in moving to the 8-way case, like 2PL (but less so), whereas the degradation for WW moves in the opposite direction, like OPT (but less so). This behavior is in agreement with the extent to which BTO and WW rely on aborts to handle conflicts: since WW relies more heavily on aborts, it is the more OPT-like of the two.

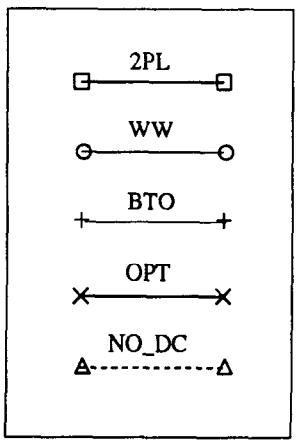
4.4. The Effect of System Overheads

This section studies how varying the CPU costs for sending messages and for starting cohort processes impacts the performance gains seen in the second experiment; we thus use the same parameters here, but we restrict our attention to the smaller database size. Performance is examined as a function of the degree of partitioning and parallelism here, so we consider two additional partitioning alternatives (together with the 1-way and 8-way cases from the previous experiment):

2-Way Partitioning: In this case, each relation R_i is split across two nodes, and transactions that access R_i consist of two parallel cohorts, one at each of the nodes. More specifically, R_i is partitioned by storing F_{ij}, $1 \le j \le 4$, at node S_i, while F_{ij}, $5 \le j \le 8$, are stored at node $S_{(i \bmod 8)+1}$.

4-Way Partitioning: In this case, each relation R_i is split across four nodes, so transactions that access R_i consist of four parallel cohorts. Partitions F_{i1} and F_{i2} reside at node S_i, F_{i3} and F_{i4} reside at node $S_{(i \bmod 8)+1}$, F_{i5} and F_{i6} reside at node $S_{(i \bmod 8)+2}$, and F_{i7} and F_{i8} reside at node $S_{(i \bmod 8)+3}$.

Figure 14 presents the response time speedups for the different algorithms in the no overhead case, with both *InstPerStartup* and *InstPerMsg* set to zero, for a think time of zero (where the load is the heaviest). Since the load is heavy, NO_DC gains almost nothing due to data partitioning here. However, we see that the various concurrency control algorithms do show some degree of response time speedup. 2PL speeds up the most under these extremely heavy loading conditions, and OPT speeds up the least. 2PL has the advantage that, by reducing the number of active transactions via blocking, it allows transactions that are not blocked to benefit more from parallel execution (since they



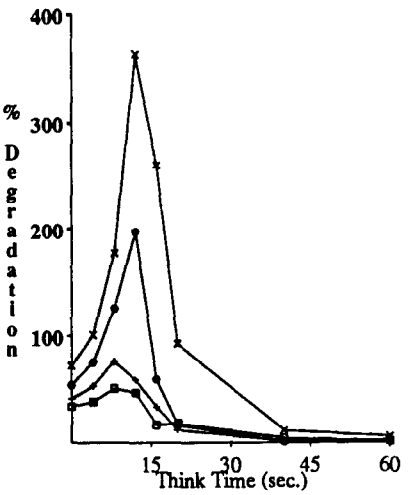


Figure 10:% Degradation, 8-way system.
(FileSize = 300)

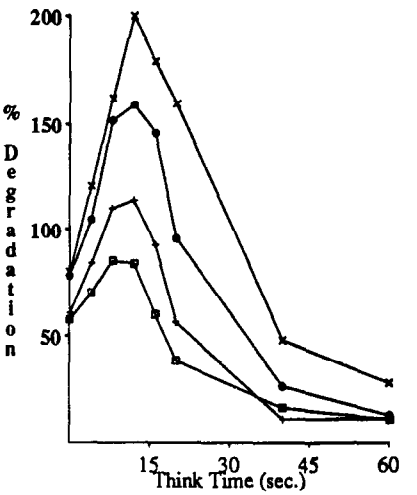


Figure 11:% Degradation, 1-way system.
(FileSize = 300)

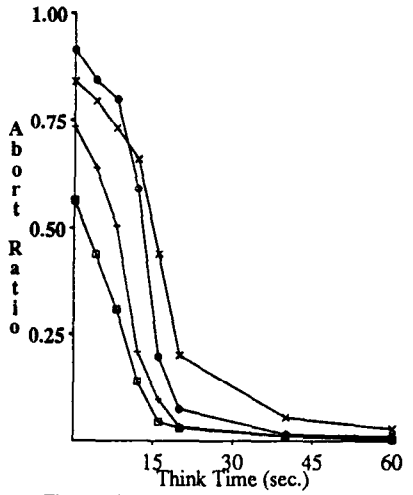


Figure 12:Abort Ratio, 8-way system.
(FileSize = 300)

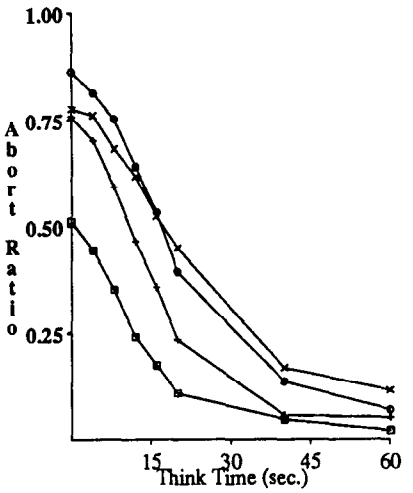


Figure 13:Abort Ratio, 1-way system.
(FileSize = 300)

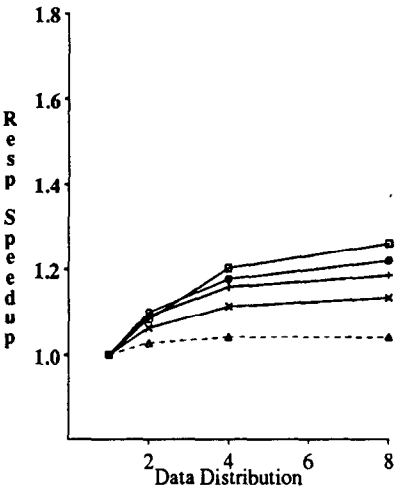


Figure 14:Response Speedup, ThinkTime = 0 sec.
(InstPerStartup = 0, InstPerMsg = 0)

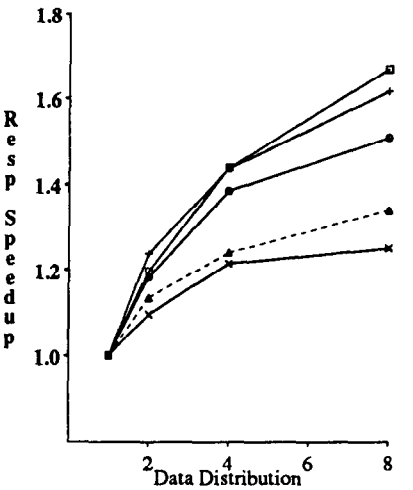


Figure 15:Response Speedup, ThinkTime = 8 sec.
(InstPerStartup = 0, InstPerMsg = 0)

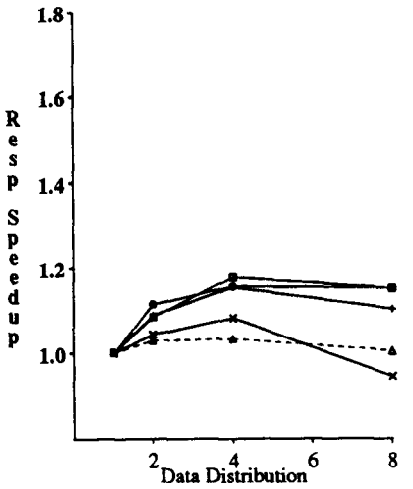


Figure 16:Response Speedup, ThinkTime = 0 sec.
(InstPerStartup = 0, InstPerMsg = 4K)

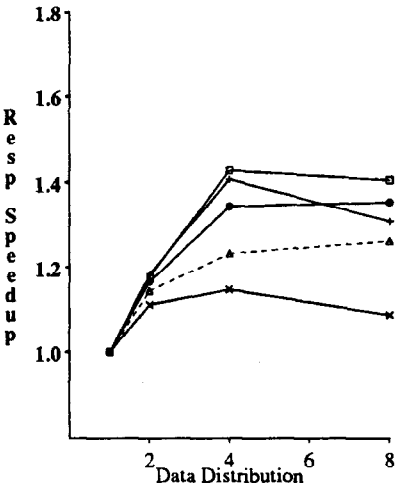


Figure 17:Response Speedup, ThinkTime = 8 sec.
(InstPerStartup = 0, InstPerMsg = 4K)

compete with fewer other transactions); this in turn benefits the blocked transactions since the locks that they are waiting for will be released earlier. Figure 15 presents the corresponding results for a think time of 8 seconds, where the load is not quite as heavy, and we see that the algorithms benefit more here (since there is at least some chance that parallel cohorts will run faster). Again, 2PL benefits the most, and OPT benefits the least. (This order of speedups was actually the same at very high loads in the earlier experiments; we just didn't focus on the high load regions there.) We also obtained results very similar to those of Figures 14 and 15 with settings of 2K instructions for *InstPerStartup* and 1K instructions for *InstPerMsg*, which are the values that we used in the earlier experiments in the paper.

Figures 16 and 17 present the zero and 8 second think time results once again, with *InstPerStartup* still being zero but *InstPerMsg* increased to 4K instructions. At both think times we observe a drop in speedup relative to Figures 14 and 15. In fact, several of the concurrency control algorithms (especially OPT) actually do worse with 8-way parallelism than with 4-way parallelism in this case. This is because aborts are very expensive in the 8-way case; the four-fold message cost increase makes it expensive to start and restart distributed transactions. Even the NO_DC curve shows little gain in moving from 4-way to 8-way parallelism here due to the message overhead involved in multisite transaction coordination. We also repeated this experiment with *InstPerMsg* set to zero but with *InstPerStartup* set to 20K instructions. The results in that case were very close to those of Figures 16 and 17, but the factor limiting speedup with these parameter settings was the process initiation cost involved in setting up a transaction.

5. CONCLUSIONS

In this paper, we have examined the interaction between parallelism as found in distributed (or 'shared nothing') database machines and data contention. Four alternative concurrency control algorithms were considered in this study, including two-phase locking, wound-wait, basic timestamp ordering, and optimistic concurrency control. We looked at how performance scales as a function of machine size, and at how the degree to which partitioning the database for intra-transaction parallelism changes performance under the different concurrency control algorithms. We examined performance from several perspectives, including response time, throughput, and speedups, over a range of system loads. We also examined the performance impact of the CPU overhead associated with communications and process initiation.

In an earlier study, we examined the performance of the same set of algorithms in a setting with replicated data and non-distributed transactions [Care88]. The work described here is complementary, the objective being to understand how distributed transactions and intra-transaction parallelism change things (and by how much). In terms of absolute performance, the results obtained here are basically consistent with those in [Care88]: Two-phase locking provided the best performance, followed by basic timestamp ordering, followed by wound-wait, followed by the optimistic algorithm.[13] The ordering of the algorithms was due to the degree to which they tend to rely on aborts for resolving conflicts. The differences in absolute performance were quite pronounced under 8-way parallelism due to the high cost of aborts in this case and the fact that blocking a subset of the conflicting transactions can enable the other transactions to complete more quickly, thus reducing waiting times. Transaction distribution and parallelism therefore do not alter the fact that two-phase locking appears to have superior performance characteristics, contrary to what one might (mistakenly) interpret the many-resource results of [Agra87a] to mean.

In terms of relative performance, we found that concurrency control effects due to data contention can lead to somewhat different speedup behavior than that obtained in the absence of concurrency control. Thus, even when the load is high, where parallelism would not usually be expected to help, our results indicate that parallelism can be beneficial. This was especially true in the case of two-phase locking, which was seen to benefit the most from parallelism under high loads.

REFERENCES

[Agra87a] Agrawal, R., Carey, M., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. on Database Sys.* 12, 4, Dec. 1987.

[Agra87b] Agrawal, R., Carey, M., and McVoy, L., "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *IEEE Trans. on Software Eng.* SE-13, 12, Dec. 1987.

[Alex88] Alexander, W., and Copeland, G., "Process and Dataflow Control in Distributed Data-Intensive Systems ," *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[Bada79] Badal, D., "Correctness of Concurrency Control and Implications in Distributed Databases," *Proc. COMPSAC '79 Conf.*, Chicago, IL, Nov. 1979.

[Balt82] Balter, R., Berard, P., and Decitre, P., "Why Control of the Concurrency Level in Distributed Systems is More Fundamental than Deadlock Management," *Proc. 1st ACM SIGACT-SIGOPS Symp. on Principles of Dist. Comp.*, Aug. 1982.

[Bern80] Bernstein, P., and Goodman, N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," *Proc. 6th VLDB Conf.*, Mexico City, Mexico, Oct. 1980.

[Bern81] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Comp. Surveys* 13, 2, June 1981.

[Bhar82] Bhargava, B., "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking," *Proc. 3rd Int'l. Conf. on Dist. Comp. Sys.*, Miami, FL, October 1982.

[13] Note: The optimistic algorithm actually outperformed two-phase locking in [Care88] when several copies of each data item needed updating and messages were expensive. However, more recently we showed that two-phase locking can dominate the optimistic algorithm, even with expensive messages and replicated data, by simply deferring requests for write locks on remote copies until the first phase of the commit protocol [Care89].

[Bhid88] Bhide, A., and Stonebraker, B., "A Performance Comparison of Architectures for Fast Transaction Processing," *Proc. 4th Int'l. Conf. on Data Eng.*, Los Angeles, CA, Feb. 1988.

[Bora88] Boral, H., "Parallelism and Data Management," *Proc. 3rd Int'l Conf. on Data and Knowledge Bases*, Jerusalem, Israel, June 1988.

[Borr88] Borr, A., and Putzolu, F., "High Performance SQL Through Low-Level System Integration," *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[Care88] Carey, M., and Livny, M., "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proc. 14th VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Care89] Carey, M., and Livny, M., *Conflict Detection Tradeoffs for Replicated Data*, Tech. Rep. No. 826, Comp. Sci. Dept., Univ. of Wisconsin, Madison, Feb. 1988.

[Ceri82] Ceri, S., and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," *Proc. 6th Berkeley Workshop on Dist. Data Mgmt. and Comp. Networks*, Feb. 1982.

[DeWi86] DeWitt, D., et al, "GAMMA — A High Performance Backend Database Machine," *Proc. 12th VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[DeWi88] DeWitt, D., Ghandeharizadeh, S., and Schneider, D., "A Performance Analysis of the Gamma Database Machine," *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[Gall82] Galler, B., *Concurrency Control Performance Issues*, Ph.D. Thesis, Comp. Sci. Dept., Univ. of Toronto, Sept. 1982.

[Garc79] Garcia-Molina, H., *Performance of Update Algorithms for Replicated Data in a Distributed Database*, Ph.D. Thesis, Comp. Sci. Dept., Stanford Univ., June 1979.

[Gray79] Gray, J., "Notes On Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Jenq89] Jenq, B., Twichell, B., and Keller, T., "Locking Performance in a Shared Nothing Parallel Database Machine," *Proc. 5th Int'l. Conf. on Data Eng.*, Los Angeles, CA, Feb. 1989.

[Khos88] Khoshafian, S., and Valduriez, P., "Parallel Execution Strategies for Declustered Databases," in *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka, eds., Kluwer Academic Press, 1988.

[Kohl85] Kohler, W., and Jenq, B., *Performance Evaluation of Integrated Concurrency Control and Recovery Algorithms Using a Distributed Transaction Processing Testbed*, Tech. Rep. No. CS-85-133, Dept. of Elec. and Comp. Eng., Univ. of Massachusetts, Amherst, 1985.

[Lai88] Lai, M., Wilkinson, W., and Lanin, V., "Distributing the Optimistic Multiversioning Page Manager in the JASMIN Database Machine," in *Database Machines and Knowledge Base Machines*, M. Kitsuregawa and H. Tanaka, eds., Kluwer Academic Press, 1988.

[Lazo86] Lazowska, E., et al, "File Access Performance of Diskless Workstations," *ACM Trans. on Comp. Sys.* 4, 3, Aug. 1986.

[Li87] Li, V., "Performance Model of Timestamp-Ordering Concurrency Control Algorithms in Distributed Databases," *IEEE Trans. on Comp.* C-36, 9, Sept. 1987.

[Lin83] Lin, W., and Nolte, J., "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking," *Proc. 9th VLDB Conf.*, Florence, Italy, Nov. 1983.

[Livn88] Livny, M., *DeNet User's Guide*, Version 1.0, Comp. Sci. Dept., Univ. of Wisconsin, Madison, 1988.

[Mena78] Menasce, D., and Muntz, R., "Locking and Deadlock Detection in Distributed Databases," *Proc. 3rd Berkeley Workshop on Dist. Data Mgmt. and Comp. Networks*, Aug. 1978.

[Noe87] Noe, J., and Wagner, D., "Measured Performance of Time Interval Concurrency Control Techniques," *Proc. 13th VLDB Conf.*, Brighton, England, Sept. 1987.

[Oszu85] Oszu, M., "Modeling and Analysis of Distributed Database Concurrency Control Algorithms Using an Extended Petri Net Formalism," *IEEE Trans. on Softw. Eng.* SE-11, 10, Oct. 1985.

[Reed83] Reed, D., "Implementing Atomic Actions on Decentralized Data," *ACM Trans. on Comp. Sys.* 1, 1, Feb. 1983.

[Ries78] Ries, D., and Epstein, R., *Evaluation of Distribution Criteria for Distributed Database Systems*, ERL Memo. No. UCB/ERL M78/22, Univ. of California, Berkeley, May 1978.

[Ries79] Ries, D., "The Effects of Concurrency Control on the Performance of a Distributed Data Management System," *Proc. 4th Berkeley Workshop on Dist. Data Mgmt. and Comp. Networks*, Aug. 1979.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. on Database Sys.* 3, 2, June 1978.

[Schl81] Schlageter, G., "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proc. 7th VLDB Conf.*, Cannes, France, Sept. 1981.

[Sinh85] Sinha, M., et al, "Timestamp Based Certification Schemes for Transactions in Distributed Database Systems," *Proc. ACM SIGMOD Conf.*, Austin, TX, May 1985.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. on Softw. Eng.* SE-5, 3, May 1979.

[Ston86] Stonebraker, M., "The Case for Shared Nothing," *Database Eng.* 9, 1, March 1986.

[Tand88] The Tandem Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction," *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988.

[Tera85] *Teradata DBC/1012 Data Base Computer Systems Manual, Release 1.3*, Teradata Corp. Document No. C10-0001-00, Feb. 1985.

[Thom79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. on Database Sys.* 4, 2, June 1979.

[Trai82] Traiger, I., et al, "Transactions and Consistency in Distributed Database Systems," *ACM Trans. on Database Sys.* 7, 3, Sept. 1982.