A Performance Study of Locking Granularity in Shared-Nothing Parallel Database Systems

S. Dandamudi, S. L. Au, and C. Y. Chow

School of Computer Science, Carleton University
Ottawa, Ontario K1S 5B6, Canada

TECHNICAL REPORT TR-97-4

ABSTRACT - Locking granularity refers to the size of a lockable data unit, called a "granule", in a database system. Fine granularity improves system performance by increasing concurrency level but it also increases lock management overhead. Coarse granularity, on the other hand, sacrifices system performance but lowers the lock management cost. This paper explores the impact of granule size on performance of shared-nothing parallel database systems. We also report the interaction between transaction and sub-transaction scheduling policies and the locking granularity.

Index Terms – Database systems, parallel systems, performance evaluation, locking, locking granularity, scheduling.

1. INTRODUCTION

In recent years, the demand for high transaction processing rates has continued to increase substantially. At the same time, the amount of data involved in processing these transactions is increasing. One approach to meet this demand for increased transaction processing power is to parallelize transaction processing by utilizing multiple processors. In such parallel database systems, transactions are divided into several sub-transactions, all of which can be run in parallel.

A parallel database system can be built by connecting together many processors, memory units, and disk devices through a global interconnection network. Depending on how these components are interconnected there can be a variety of possible architectures. These include the shared-nothing, shared-disk, and shared-everything [DeWi92]. In this paper we consider the shared-nothing architecture in which each processor has its own (private) disk and memory units. Shared-nothing database systems are attractive from the standpoint of scalability and reliability [Ston86]. Several commercial and experimental systems such as the Non-Stop SQL from Tandem [Tand88], the DBC/1012 from Teradata [Tera85], the Gamma machine at the University of Wisconsin [DeWi88], and Bubba at MCC [Jenq89] belong to this type of architecture.

Locking is one of the common techniques used to achieve concurrency control (see [Bern81] for an excellent survey of concurrency control algorithms). Several commercial and experimental systems such as the Non-Stop SQL from Tandem [Tand88], the DBC/1012 from Teradata [Tera85], the Gamma database machine at the University of Wisconsin [DeWi88], and the Bubba at MCC [Jenq89] use lock-based concurrency control mechanisms.

These systems, which use physical locks (as opposed to predicate locks [Eswa76]), support the notion of granule. A granule is a lockable unit in the entire database and typically all granules in the database have the same size. The granule can be a column of a relation, a single record, a disk block, a relation or the entire database. Locking granularity refers to the size of lockable granule. Fine granularity improves system performance by increasing concurrency level but it also increases lock

management cost because the number of locks increases as the granularity gets finer. Furthermore, more lock and unlock operations will be issued, causing a higher overhead. In addition, more storage space is required for the locks. For example, if the granule size is a record, the system must maintain a lock table that has the same number of entries as the number of records in the database. Such large lock tables are necessarily stored on secondary storage device such as disk. Coarse granularity, on the other hand, sacrifices system performance but lowers the lock management cost. For example, if the granule size is the entire database, then transactions are forced to run in a serial order. However, the system handles only one lock, which can be kept in main memory.

Performance analysis of centralized database systems with locking has been extensively studied by using analytical and simulation models [Fran85, Tay85, Thom85, Thom89, Ryu90]. Effects of locking granularity in centralized database systems have been studied by Ries and Stonebraker [Ries77, Ries79]. Performance issues in parallel database systems have been studied in [Care88, Care89, Jenq89, Yu90]. In this paper, we study the impact of locking granularity on the performance of parallel database systems. As in previous studies, we use simulation as the principal means of performance evaluation.

The remainder of the paper is organized as follows. Section 2 describes the simulation model used in the study. The results of the simulation experiments are discussed in Section 3. Section 4 discusses the impact of both transaction and sub-transaction scheduling policies on locking granularity. Section 5 concludes the paper by summarizing the results.

2. THE SIMULATION MODEL

This section describes the simulation model used in our study, which is an extension of the model used by Ries and Stonebraker [Ries77, Ries79]. This is a closed model with a fixed number of transactions in the system at any instance. As shown in Figure 1, the transactions cycle continuously through the system. In our simulation model, every time a transaction completes execution, a new transaction with a new set of parameter values is created so that the system always sees a fixed number of transactions.

As stated in Section 1, we assume the shared-nothing architecture in our study. In this architecture the database is partitioned across the system resources such that parallel processing is facilitated. In our model, all relations are assumed to be horizontally partitioned across all disk units in the system (i.e., full declustering is used) using a round robin strategy. Thus any given relation is equally partitioned among all the disk drives (the relations are assumed to be large enough to allow this). Full declustering is employed in the Teradata and the Gamma parallel database systems. In addition, we also model the random partitioning method in which all relations are randomly partitioned into some items and these items are randomly distributed to a *subset* of disk drives in the system.

We assume a *fork and join* type of transaction structure. In this model, a transaction is assumed to be split into several sub-transactions, each working on fragments of the relations involved. For example, if the operation is the *selection* operation, then each processor will be working on a smaller relation to retrieve partial response to the selection operation. These partial responses are then combined to derive the final response. Thus, a transaction is not considered complete unless all the sub-transactions are finished. It is also assumed that each sub-transaction consists of two phases: an I/O phase and a CPU phase. This model of transaction assumes that there is no communication between the sub-transactions. We, however, believe that it is useful in obtaining insights into the influence of locking granularity on performance of parallel database systems. We intend to consider the impact of this communication in the near future.

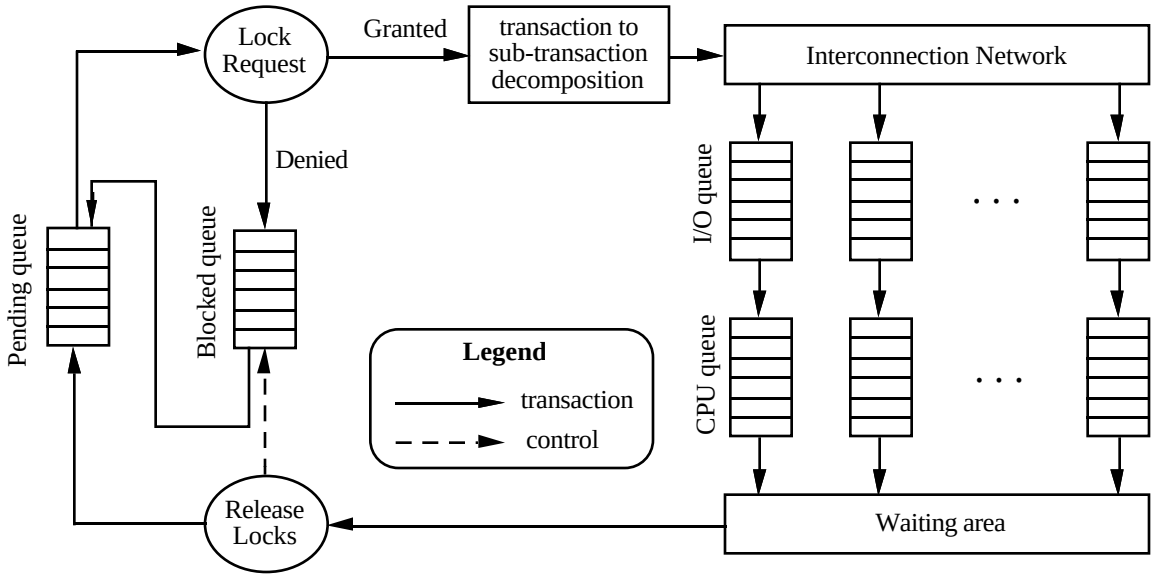

Figure 1 Simulation model

We now describe the simulation model in detail. The system is initialized to contain *ntrans* transactions. Each transaction goes through the following stages.

1. The transaction at the head of the pending queue is removed and the required locks are requested. It is assumed that all accessed data items are updated. Therefore, all lock requests are for exclusive locks. If the locks are denied, the transaction is placed at the end of the blocked queue. System will record the blocked transaction. This record is used to help release blocked transactions after blocking transactions finish their processing and release their locks. If the transaction receives all its locks, it is split into sub-transactions. For horizontal partitioning, it is split into *npros* (the total number of processors in the system) sub-transactions; for random partitioning, it is split into i sub-transactions where $1 \leq i \leq npros$. Then each sub-transaction is placed at the end of the I/O queue of its assigned processor. Notice that no two sub-transactions are assigned to the same processor.
2. After completing the required I/O and CPU service, the sub-transaction waits in the waiting area for the other sub-transactions of its parent transaction to complete.
3. A transaction is considered complete only when all its sub-transactions are completed. A completed transaction releases all its locks (and those transactions blocked by it). The releasing process of blocked transactions is described at the end of this section.

Locking schemes can be classified as static or dynamic schemes [Poti80]. In static locking, locks required by a transaction are predeclared. A transaction is not activated until all needed locks are acquired (i.e., the transaction does not conflict with any running transaction). Once a transaction is activated, the transaction can be executed to completion without blocking. While this locking scheme avoids deadlocks, it does not preclude starvation. Explicit locking in DB2 is an example of static locking [Date89]. In dynamic locking, locks are requested on demand (i.e., on a "claim as needed" basis). In this case, the transactions may be blocked after activation since all needed locks are not acquired before the activation. Deadlocks are also possible with dynamic locking.

We use static locking in this study. Thus, transactions request all needed locks before using the I/O

and CPU resources. It has been shown in a previous study [Ries79] that changing this assumption to "claim as needed" did not affect the conclusions of the study. Therefore, we do not model this strategy in our simulation model.

It is assumed that the cost to request and set locks includes the cost of releasing those locks. The cost is assumed to be the same even if the locks are denied. Moreover, the locking mechanism has preemptive priority over running transactions for I/O and CPU resources. In addition, we assume that processors share the work for the locking mechanism. This is justified because relations are equally distributed among the system resources.

The database is considered to be a collection of *dbsize* data entities or granules. These data granules are grouped into *ltot* lockable granules. Each lockable granule has a lock associated with it. The number of locks *ltot* can be varied to vary locking granule size. If *ltot* is 1 (coarse granularity), the granule size is the entire database and no concurrency is permitted in transaction execution. On the other hand, finest locking granularity can be achieved by setting *ltot = dbsize*. The number of transactions in the system is represented by *ntrans*. The size of the transactions is uniformly distributed over the range 1 and *maxtransize*. Thus the average transaction size is approximately equal to *maxtransize*/2. *cputime* (*iotime*) represents the CPU (I/O) time required by a transaction to process one data granule. *lcputime* (*liotime*) represents the CPU (I/O) time required to request and set one lock. The number of processors in the system is represented by *npros*.

The number of data granules required by *i*th transaction is uniformly distributed over the range 1 and *maxtransize* The number of locks required by *i*th transaction is given by **ceil**(*NUi * ltot / dbsize*). This corresponds to the best placement of granules requiring number of locks proportional to the portion of the database accessed by a transaction. The impact of alternative assumptions is discussed in Section 3.5.

The computation of lock conflicts

We use a probabilistic lock contention model in which the probability of lock contention is given as the ratio of the number of locks held by active transactions to the total number of locks. Similar lock contention models have been used in the literature [Ries77, Tay85, Yu90]. The validation results in [Yu90] show that this model accurately predicts lock contention. This model is extended to include information on the transaction that is causing the lock conflict.

Let T_1, T_2, ..., T_k denote the k active transactions currently using the I/O and CPU resources. Suppose that each transaction T_i has L_i locks. Then divide the interval (0,1] into k+1 partitions.

$$P_1 = (0, L_1/ltot]$$

$$P_2 = (L_1/ltot, (L_1+L_2)/ltot]$$

$$\ldots$$

$$P_k = ((L_1+L_2+...+L_{k-1})/ltot, ((L_1+L_2+...+L_k)/ltot]$$

$$P_{k+1} = ((L_1+L_2+...+L_k)/ltot, 1]$$

To determine if a given transaction must be blocked, choose a random number p, uniformly distributed on (0, 1). If p is in P_j, for some $j \le k$, then the transaction is blocked by T_j. Otherwise, the transaction may proceed. Hence, if the transaction is blocked, T_j will be the one to release it. Note, however, that the released transaction is not guaranteed to receive all its locks. Instead, the released transaction is placed in the pending queue and then goes through the lock request process. If locks are denied, it will be placed in the blocked queue again. This reflects the situation that there may be

other transactions that are still holding a subset of the locks required by the released transaction. Thus the lock conflict computation models the fact that an arriving transaction in general requires locks currently held by more than one transaction. In addition, static locking may cause lock starvation. The lock conflict computation used here models starvation in which a transaction may be repeatedly denied its predeclared locks and, therefore, not activated.

3. RESULTS AND DISCUSSION

This section presents the results obtained from the simulation experiments. The default input parameter are given in Table 1. All the results were obtained with the horizontal partitioning scheme, except for the results reported in Section 3.4, which discusses the impact of random partitioning on locking granularity. Also, the transaction size is uniformly distributed[1] as described in Section 2. Furthermore, the results in Sections 3.1 through 3.4 use the best placement strategy discussed in Section 2. For all the results reported in this section, we assume first-in-first-out (FIFO) scheduling policy for both transaction scheduling and sub-transaction scheduling. The impact of other scheduling policies is considered in Section 4.

Transaction throughput, which is defined as the number of completed transactions per unit time, is used to measure the performance of the system. In addition, we use the following performance indicators.

RT = average response time (i.e., from the time the transaction enters the pending queue till the time it completes all processing and releases locks).

BT = average blocking time (blocking time is the time spent by a transaction in the blocked queue awaiting release of the needed locks).

ST = average synchronization time (synchronization time is defined as the time between the completion of the first sub-transaction to the completion of the last sub-transaction of the same transaction).

Table 1 Input parameters used in the simulation experiments

parameter	value	parameter	value
dbsize	5000	*iotime*	0.20
ntrans	10	*lcputime*	0.01
cputime	0.05	*liotime*	0.20
maxtransize	500		

3.1. Impact of the Number of Processors

Figure 2 shows the impact of the number of processors and the number of locks on system throughput. For a fixed number of locks, the throughput increases with increasing number of processors. The convex nature of the curves arises from the trade-off between the locking overhead and the allowed degree of concurrency. That is, increasing the number of locks initially increases the throughput by allowing concurrent execution of more transactions until a certain point after which the overhead for locking predominates and the throughput decreases. The curves become very steep

[1]The use of other distributions results in similar conclusions; therefore, these results are omitted.

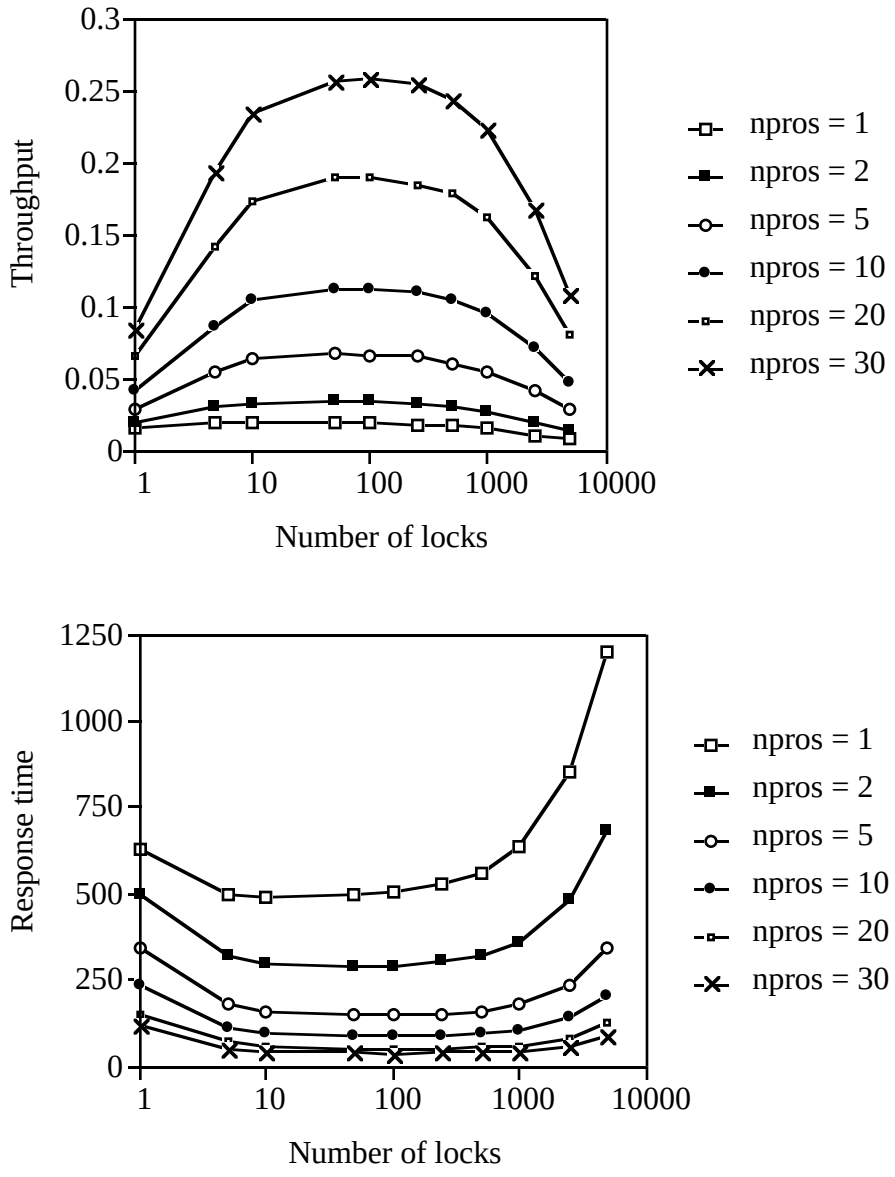

Figure 2 Effects of number of locks and number of processors on throughput and response time

with large number of processors because more locks allow higher degree of parallelism which offers greater benefit to a system with larger number of processors. However, the optimal point, the highest point on the curve, has a number of locks that is less than or equal to 100 even with 30 processors. These results show that coarse granularity is sufficient.

These observations agree with the conclusions reported in [Ries77]. However, it is important to note that the penalty (in terms of reduced throughput) associated with not maintaining the optimum number of locks increases with increasing number of processors. This penalty becomes more severe for large systems consisting of hundreds of processors.

Figure 2 also shows the impact on response time. The concave nature of the curves is due to the trade-off discussed before. For a fixed number of locks, the average response time decreases with increasing number of processors. There are several factors contributing to this reduction in response

time. First, the work for requesting, setting, and releasing locks is shared by all processors so the overhead per processor is reduced with increasing number of processors (and lock processing can be done by all processors concurrently). Second, a transaction is split into more sub-transactions with increasing number of processors. This in turn leads to smaller sub-transactions queueing up for service. Hence the queueing time as well as the service time of sub-transactions becomes smaller and the synchronization time reduces. Decreases in service time and lock overhead push the response time toward the minimum so varying the number of locks does not significantly affect the response time in systems with large numbers of processors. This explains why the curves become flatter when the number of processors is large.

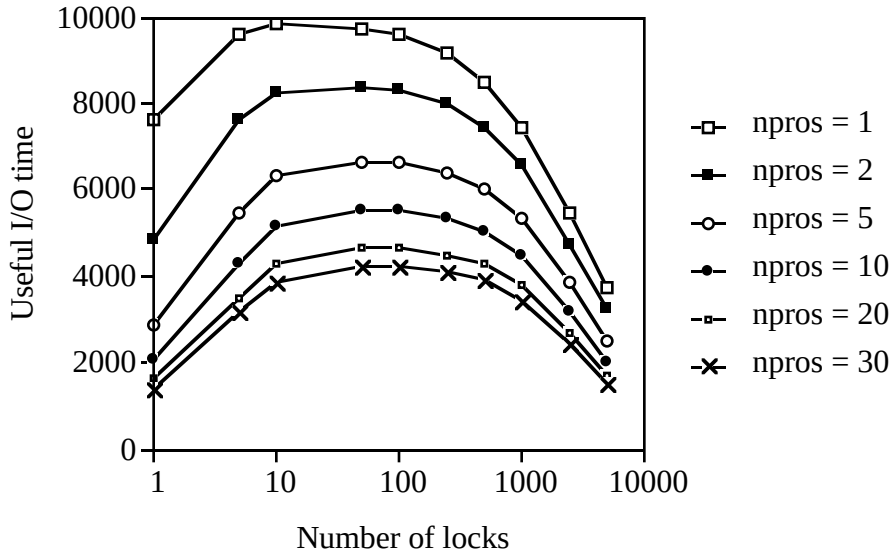

Figure 3 Effects of number of locks and number of processors on useful I/O time

The useful I/O time is shown in Figure 3 as a function of number of processors and number of locks. (The useful CPU time also exhibits similar behaviour, except for the scaling difference.) Again, the convex nature of these curves is due to the trade-off discussed previously. Note that as the number of locks is increased beyond the optimum value (between 10 and 100 locks) the difference in useful I/O time (and useful CPU time) associated with the different number of processors decreases. This is because systems with smaller number of processors tend to spend more time on lock operations as shown in Figure 4, which gives the amount of time spent on lock operations. It may be noted from this figure that lock overhead increases substantially when the number of locks is greater than about 200.

The concave nature of the curves (more pronounced in Figure 4b) can be explained as follows. With small number of locks, the high lock request failure rate causes increased I/O and CPU lock overhead. For example, when the number of locks is one, only one transaction can access the database. Lock requests from all the other transactions are denied. This lock overhead reduces as the number of locks is increased from one because of the associated reduction in lock request failure rate. Eventually, after the optimal point, increasing the number of locks results in increased lock overhead simply because each transaction requests large number of locks. As can be seen from Figure 4, this lock overhead increases with the transaction size and the number of locks.

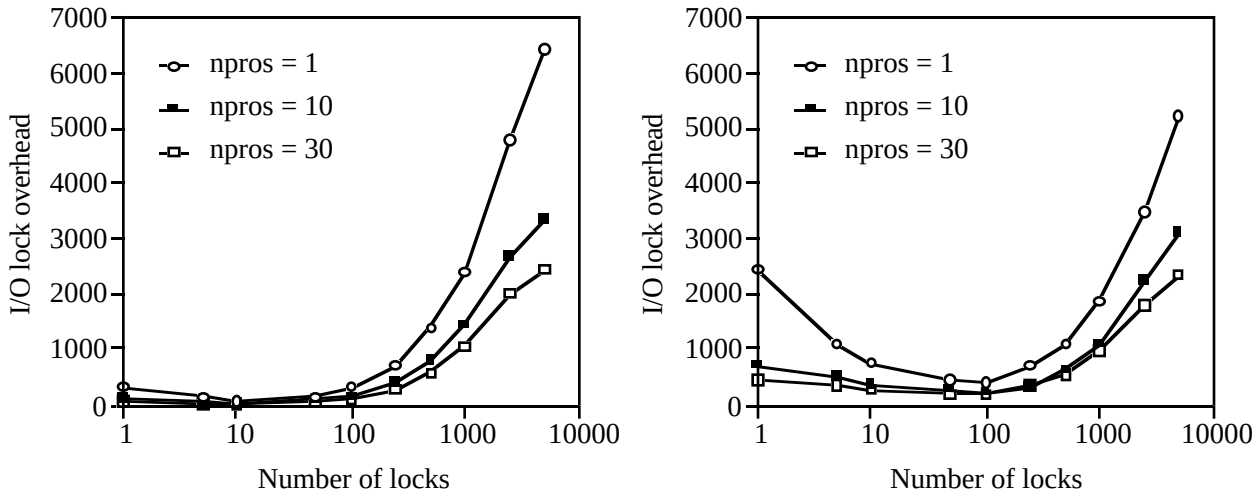

Figure 4 Effect of number of processors and number of locks on lock overhead with **(a)** large transactions **(b)** small transactions

The concave nature is less pronounced with a large number of processors because the lock overhead cost is shared by all processors. Also note that the initial part of these curves (from 1 to ≈100 locks) show more overhead associated with small transactions. This is due to the fact that small transactions result in increased system throughput, which in turn results in increased lock request rate leading to increased lock overhead.

3.2. Effects of Transaction Size

In this section the impact of the average transaction size is presented. All the input parameters are fixed as in Table 1 except for *maxtransize* which is varied from 50 to 5000 (50, 100, 500, 2500 and 5000). This corresponds to an average transaction size of 0.5%, 1%, 5%, 25% and 50% of the database, respectively. For these simulation runs the number of processors (*npros*) = 10 was chosen after running several experiments with *npros* = 1, 2, 5, 10, 20, and 30 and we did not observe any significant differences except for scaling.

Figure 5 shows the impact of transaction size on system throughput and response time. The throughput for smaller transactions exhibits more sensitivity to the number of locks. This is because, with smaller transactions, the probability of lock conflict decreases. Thus the curves tend to be steeper with decreasing transaction size. This directly translates into flatter response time behaviour with smaller transaction sizes, as shown in Figure 5.

The optimal point shifts toward the right (i.e., larger number of locks) and the throughput curves become steeper with decreasing transaction size. The reason is that, for systems with smaller transaction sizes, increasing the number of locks from one reduces unnecessary lock request failures. For systems with larger transaction sizes, fine granularity does not improve the degree of concurrency because more database entities are required by each transaction. However, all these optimal points have a number of locks that is ≤100. This suggests that systems should have better than coarse granularity for processing smaller transactions. However, to process large transactions, coarse granularity is sufficient.

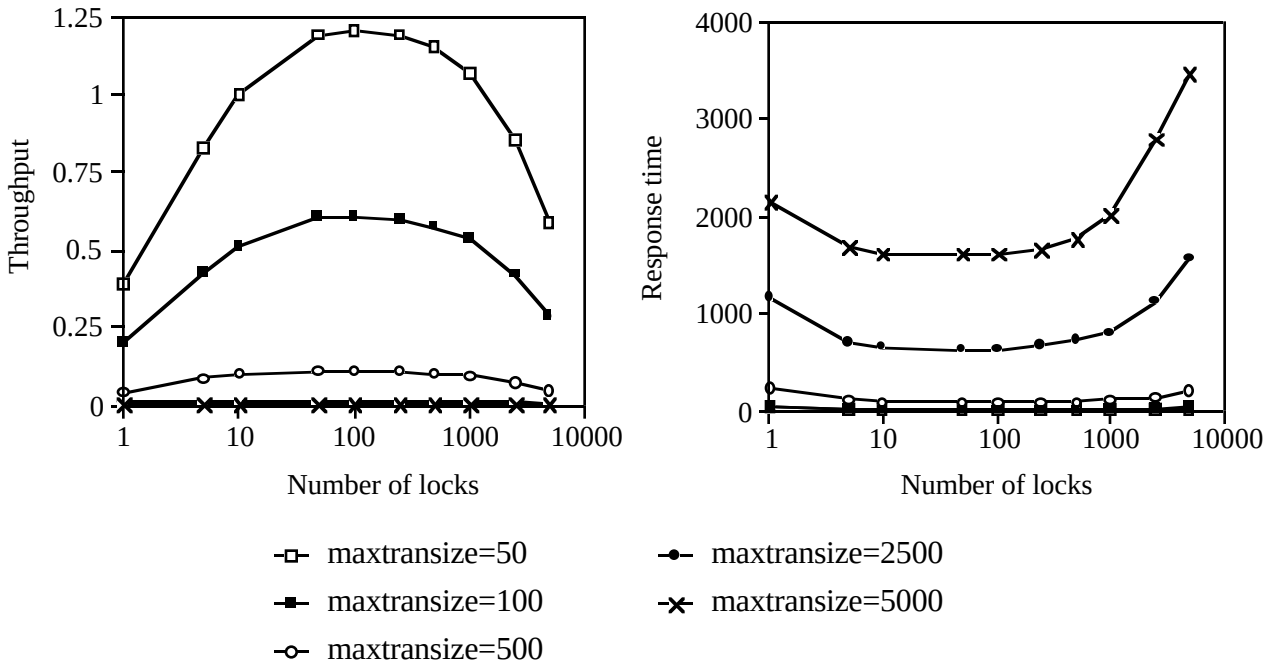

Figure 5 Effects of number of locks and transaction size on throughput and response time

3.3. Impact of Lock I/O Time

In previous experiments, the lock I/O time (*liotime*) was set equal to the transaction I/O time (*iotime*). In the next set of experiments the lock I/O time and the number of locks are varied. Fixing the other parameters as in Table 1 and the number of processors at 10, the simulation experiments were run with *liotime* equal to 0.2, 0.1, and 0. Note that *liotime* = 0 models the situation in which the concurrency control mechanism maintains the lock table in main memory.

The throughput results are shown in Figure 6. This figure shows that as the lock I/O time decreases, larger number of locks could be afforded until the benefits of increased level of concurrency were outweighed by the lock processing overhead. Even with no lock I/O overhead, throughput is flat between 100 and 5000 locks because the throughput has already reached its maximum at 100 locks due to lock contention.

In conclusion it appears that even when the lock table is kept in main memory coarse granularity is sufficient. The only difference is that, when the lock table is kept in main memory, finer granularity does not hurt performance as is the case with finite lock I/O time.

3.4. Impact of Random Partitioning

This section investigates the impact of random partitioning on locking granularity. The simulation experiments were run with the same input parameters as in Section 3.1 (given in Table 1). The results are shown in Figure 7. This graph shows that the impact of number of processors does not depend on the partitioning method used.

By comparing Figures 2 and 7 it can be observed that horizontal partitioning leads to better system performance without actually affecting the locking granularity. For example, for the same number of processors, a curve in Figure 2 has a higher throughput than the corresponding curve in Figure 7.

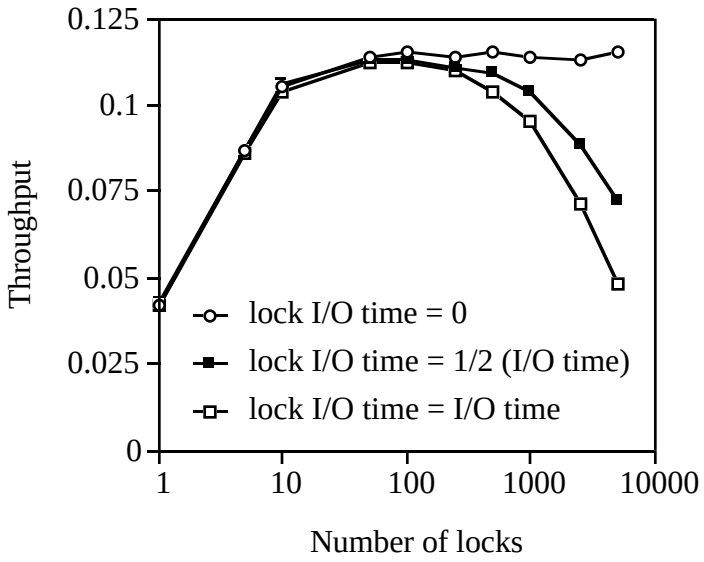

Figure 6 Effects of number of locks and lock I/O time on throughput (*npros* = 10)

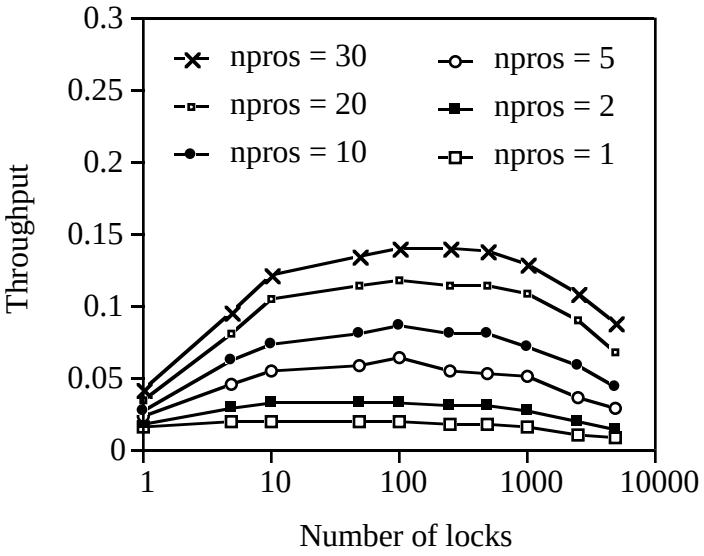

Figure 7 Effects of number of locks and number of processors on throughput (random partitioning)

Note that horizontal partitioning maximizes the number of sub-transactions into which a transaction is divided, thus leading to smaller size sub-transactions than those with random partitioning. Thus the queueing and service times are smaller with horizontal partitioning. This further leads to reduced synchronization times which improves the overall system performance.

In an earlier study, [Livn87] compared the performance of full declustering with that of no declustering on a multi-disk file system. They concluded that full declustering is a better approach under normal system loads. These observations are consistent with the conclusions of this section. Note that full declustering is used in the Teradata and the Gamma parallel database systems.

3.5. Impact of Granule Placement

So far in our discussion, we have assumed that the number of locks required by a given transaction is proportional to the percentage of the database entities accessed by the transaction. For example, a transaction accessing 10% of the database requires 10% of the total locks (*ltot*). This corresponds to a situation where the required entities are packed in as few granules as possible (call this the *best placement*). This is a reasonable assumption for transactions that access the database sequentially. For example, in answering a range query, sequential access is often used to retrieve all the records satisfying the given range query. Empirical data suggests that sequential processing is fairly substantial in practice [Rodr76]. However, actual transactions also require a combination sequential and random accesses to the database. This requires alternative assumptions of granule placement, the impact of which is discussed in this section.

In order to study the impact of granule placement on locking granularity, we employ the two alternative granule placement strategies used in [Ries79]: *worst placement* and *random placement.*

Worst Placement: In this placement strategy, if a transaction requires a number of data granules *NUi* that is greater than the number of locks (*ltot*), then in the worst case, all of the locks may be required by the transaction. If *NUi* is less than *ltot*, then the transaction may require *NUi* locks to be set before accessing the database. Therefore, in this placement strategy, the number of required locks is given by **min** (*NUi*, *ltot*). This represents a transaction whose behaviour is at the other extreme to that of the best placement strategy used so far in our discussion.

Random Placement: This strategy provides a compromise between the two extremes considered so far. For each transaction, a mean value formula is used to estimate the number of locks required. The number of locks *LUi* required by a transaction accessing *NUi* data granules is given by [Ries79, Yao77]

$$\text{number of locks, } LUi = ltot\left[1 - \frac{\binom{dbsize - \frac{dbsize}{ltot}}{NUi}}{\binom{dbsize}{NUi}}\right]$$

Which model is more accurate depends on the nature of the transaction in a given application. Ries and Stonebraker observe that typical transactions fall somewhere between best placement and random placement depending on the amount of sequential processing done by the transactions.

Figure 8 shows the impact granule placement on system throughput for large transactions (*maxtransize* = 500). There are two sets of curves corresponding to the number of processors *npros* = 1 and 30. As shown in Figure 8, both the worst placement and random placement strategies exhibit substantially different behaviour from that with the best placement strategy. However, both these placement strategies exhibit similar behaviour.

With random or worst placement strategies, the system throughput decreases as the number of locks (*ltot*) is increased from one to the average number of entities (with *maxtransize* = 500 this average is 250 entities). This decrease is due to the fact that each transaction requires more and more locks thus increasing the locking overhead. However, there is no increase in the concurrency level because each transaction locked the entire database. Similar behaviour can be seen with smaller transaction size, as shown in Figure 9 (corresponding to an average transaction size of 25 entities with *maxtransize* = 50).

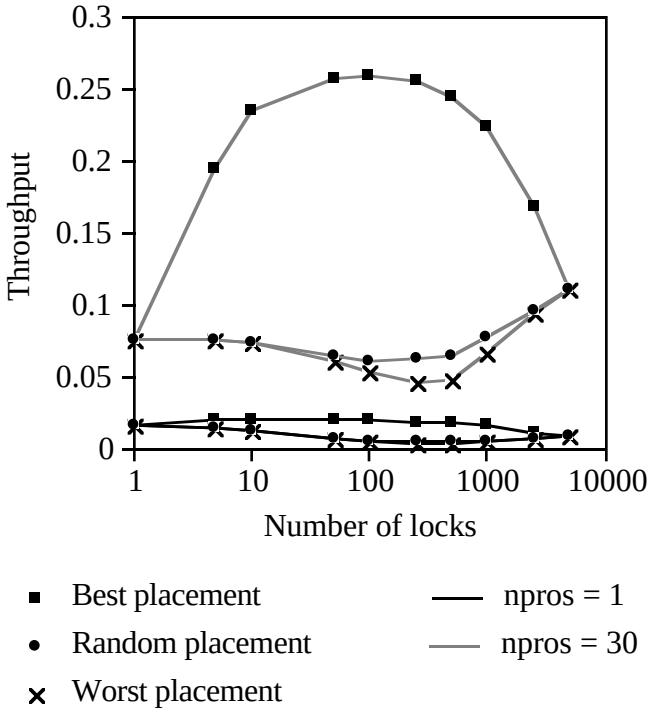

Figure 8 Effects of number of locks and granule placement on throughput with large transactions (*maxtransize* = 500)

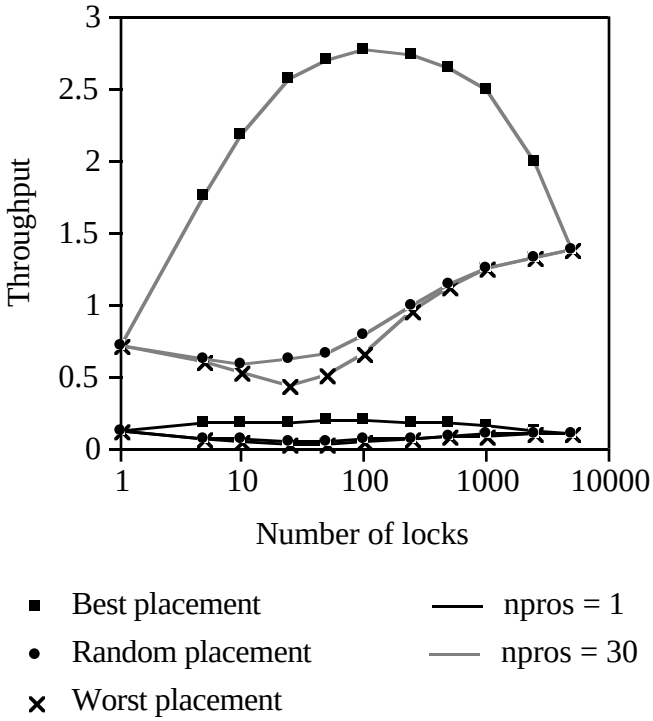

Figure 9 Effects of number of locks and granule placement on throughput with small transactions (*maxtransize* = 50)

The throughput increases as the number of locks is increased from the average number of entities accessed by the transactions to the total number of entities in the database. This increase is due to the fact that the locking overhead remains constant (because each transaction locks on average *maxtransize*/2 locks) while the allowed degree of concurrency increases. Similar behaviour has been observed in centralized databases with static as well as dynamic locking [Ries79, Ryu90].

3.6. Impact of Mixed Sized Transactions

A typical database may process transactions of different sizes. The impact of this is shown in Figure 10. These results were obtained by assuming that 80% of the transactions are small in size (average transaction size of 25) and the remaining 20% are large transactions (average transaction size of 250).

Figures 8 and 9 show the effect on system throughput when all the transactions are large and when they are all small, respectively. The results obtained with a 80/20 mix fall in between these two extremes. However, it should be noted that even the presence of 20% large transactions substantially affects systems throughput. For example, when the number of locks is equal to the number of entities in the database and when the number of processors is 30, small transactions yield a throughput of 1.373 and large transactions yield 0.1107 while a 80/20 mix of these two transaction groups yields a throughput of only 0.3419.

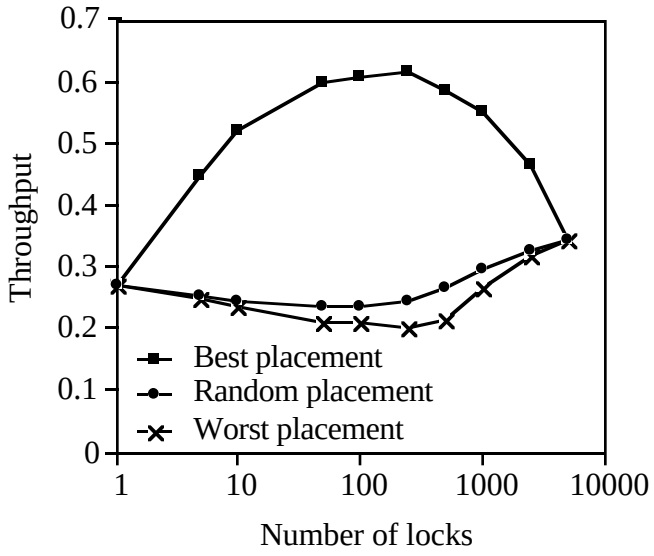

Figure 10 Effects of number of locks and granule placement on throughput with mixed transactions: 80% small transactions and 20% large transactions (*npros* = 30)

3.7. Sensitivity to Number of Transactions

The results of the previous sections assumed 10 transactions in the system. This section presents the effect of this parameter on locking granularity. Figure 11 shows this impact when the number of transactions is increased to 200 and the number of processors is fixed at 20. The average transaction size is maintained at 250 as in Figure 8. (Similar results were obtained with other transaction sizes as well.)

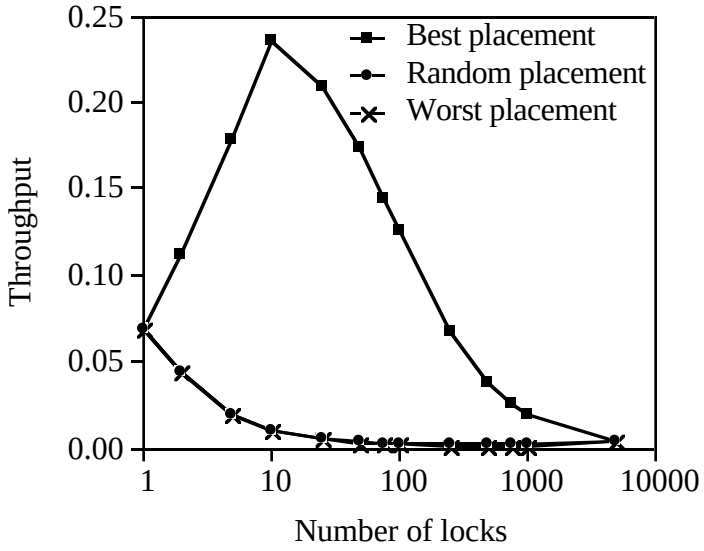

Figure 11 Effects of number of locks and granule placement on throughput with large number of transactions (*ntrans* = 200)

The effect of increased number of transactions can be seen by comparing Figures 8 and 11. A key observation is that, when the number of transactions is large, maintaining finer granularity (e.g., number of locks = number of data granules) leads to decreased throughput when compared to that obtained with coarse granularity. The reason for this is that the lock processing overhead increases in direct proportion to the number of transactions and the number of locks. Furthermore, most of these increased lock requests are denied and thus the concurrency level is not increased. This essentially results in decreased throughput. The seriousness of this overhead increases with transaction size as well (for the sake of brevity these results are omitted). Transaction-level scheduling can be effectively used to handle this problem (see Section 4.2).

4. THE INFLUENCE OF SCHEDULING POLICY ON LOCKING GRANULARITY

The performance of parallel database systems is affected by several factors. Scheduling in such systems is an important factor affecting the overall system performance. In this section, we study the impact of scheduling on locking granularity in parallel database systems. We consider scheduling policies at two levels: sub-transaction level policies and transaction level policies. Section 4.1 considers the effect of sub-transaction level scheduling policies and Section 4.2 discusses the impact of transaction level scheduling policies on locking granularity.

4.1. Influence of Sub-Transaction Scheduling Policies

Sub-transaction scheduling policy refers to how the waiting sub-transactions (either at a CPU or I/O resource) are scheduled for execution. Section 4.1.1 describes the sub-transaction scheduling policies and Section 4.1.2 presents the impact of these policies on locking granularity.

4.1.1. Scheduling Policies

There are several scheduling policies that are applicable to sub-transaction scheduling. These policies can be broadly divided into two classes: policies that are independent of the sub-transaction characteristics and policies that need certain information on sub-transactions. We consider two policies belonging to each category.

Sub-transaction independent policies

- *First-In-First-Out (FIFO):* This is the simplest scheduling policy that does not require any knowledge about sub-transaction characteristics. The sub-transactions are placed in the order of their arrival to the queue. When a resource (either CPU or I/O) becomes available the sub-transaction at the head of the waiting queue is assigned this resource.

- *Round Robin (RR)*: This is a preemptive policy in which each sub-transaction is given a fixed quantum of service and if the sub-transaction is not completed within this quantum it is preempted and returned to the tail of the associated queue. Note that this scheduling policy is useful only for processor scheduling, and not for I/O scheduling.

Sub-transaction dependent policies

- *Shortest Job First (SJF)*: This policy requires service time requirements of sub-transactions. If this knowledge is known *a priori*, this policy schedules the sub-transaction which has the smallest service time requirement among the sub-transactions that are waiting to be scheduled. In the context of databases, an estimate of this information can be obtained. For example, if the transaction involves a join operation, then selectivity characteristics of the relations involved can be used to obtain a good estimate of the complexity involved in processing the transaction.

- *Highest Priority First (HPF):* A potential disadvantage with the above strategy is that it is possible (depending on transaction characteristics) that some sub-transactions of a given transaction receive quick service (because their service requirements are smaller) while other sub-transactions of the same transaction, which have larger service demand, may be waiting for service at other processors (or I/O servers). This effectively increases transaction completion time. (Recall that a transaction is not considered complete unless all the sub-transactions of that transaction are finished.) In this policy, each transaction is given a priority that is inversely proportional to the total size of the transaction. When a transaction is divided into sub-transactions, they carry this priority information. The scheduling decisions are made based on this information. Thus all sub-transactions receive the same treatment in all queues irrespective of their actual service demand.

Note that for processor scheduling, all four policies could be applied. However, for I/O scheduling, preemptive policies are not applicable. Therefore, we study the impact of the remaining three scheduling policies for I/O scheduling. We have simulated all four policies for processor scheduling. However, it is shown in [Dand91c] that processor scheduling policies have insignificant role in determining the overall system performance. Therefore, in the following, we consider the same scheduling policy for both processor and I/O scheduling (for example, if we use the HPF policy for I/O scheduling then the same policy is used for processor scheduling).

4.1.2. Results

With the input parameters set as in Table 1, several simulation experiments were run by varying the number of locks (*ltot*) from 1 to *dbsize* = 5000 for each run. The number of transactions is fixed at *ntrans* = 200 and the number of processors is fixed at *npros* = 20 for this set of simulation experiments. For these results, FIFO transaction-level scheduling policy is used.

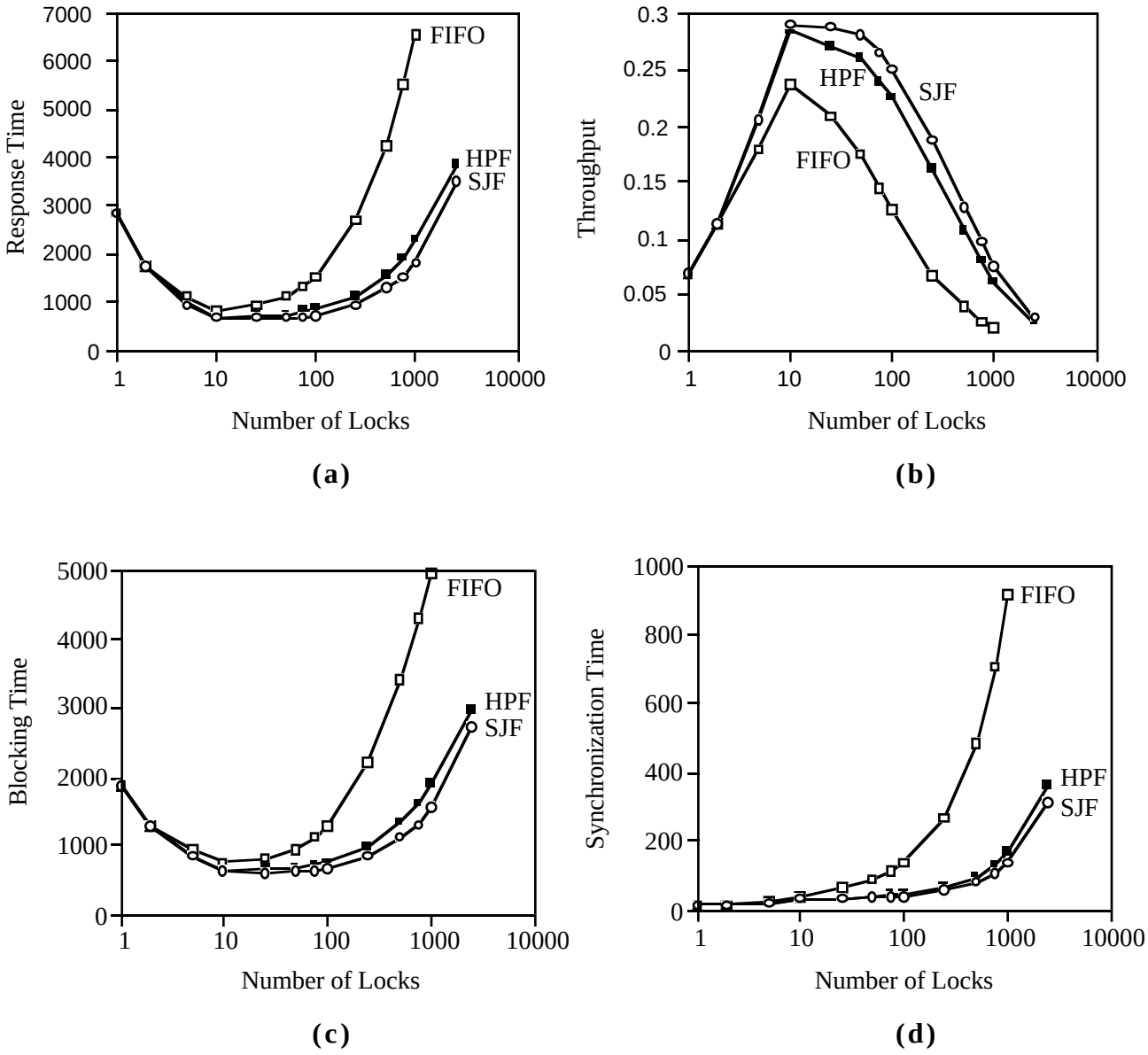

Figure 12 Performance of sub-transaction scheduling algorithms as a function of number of locks

(a) Effect of Number of Locks

The effect of sub-transaction scheduling policies is shown in Figure 12. This figure shows both the system throughput and response time as well as the blocking time and synchronization time. As shown in Figure 12c, it is the blocking time that is the major component of the response time. On the other hand, the synchronization time (shown in Figure 12d) increases substantially only when the number of locks is large. While all the scheduling policies exhibit increased sensitivities to larger number of locks, synchronization time of the FIFO policy increases much more quickly. The reason is that the lock processing overhead increases in direct proportion to the number of locks.

An important observation, however, is that the type of sub-transaction scheduling policy has negligible impact on the desired locking granularity. All three policies support coarse granularity

(dividing the entire database of 5000 entities into 10 lockable entities is sufficient to ensure best performance). However, the penalty for not maintaining the optimum number of locks is more for the FIFO policy compared to that for the other two policies. For example, when the number of locks is changed from 10 to 100, the throughput of the FIFO policy decreases by 50% whereas the corresponding reduction of the STF policy is 10%.

(b) Effect of Transaction Size

The impact of the average transaction size is shown in Figure 13. Note that the number of entities required by a transaction is determined by the input parameter *maxtransize* (maximum transaction size). The transaction size is uniformly distributed over the range 1 and *maxtransize*. Thus the average transaction size is approximately *maxtransize*/2. In this set of experiments, *maxtransize* is set at 100 and 2500. This corresponds to an average transaction size of 1% and 25% of the database, respectively.

Figure 13 shows the throughput as a function of number of locks. Several observations can be made form the data presented in Figure 13. First, the observations made about the data in Figure 12 are generally valid even when the transaction size is varied. That is, the type of sub-transaction scheduling policy has negligible impact on locking granularity. Second, the optimal point (at which the throughput peaks) shifts toward the right (i.e., toward larger number of locks or finer granularity) with decreasing transaction size. However, all these optimal points have a number of locks that is less than 100. This reinforces the previous observation that coarse granularity is sufficient. This conclusion is valid even when there is a large variance in the transaction size (the results are not presented here).

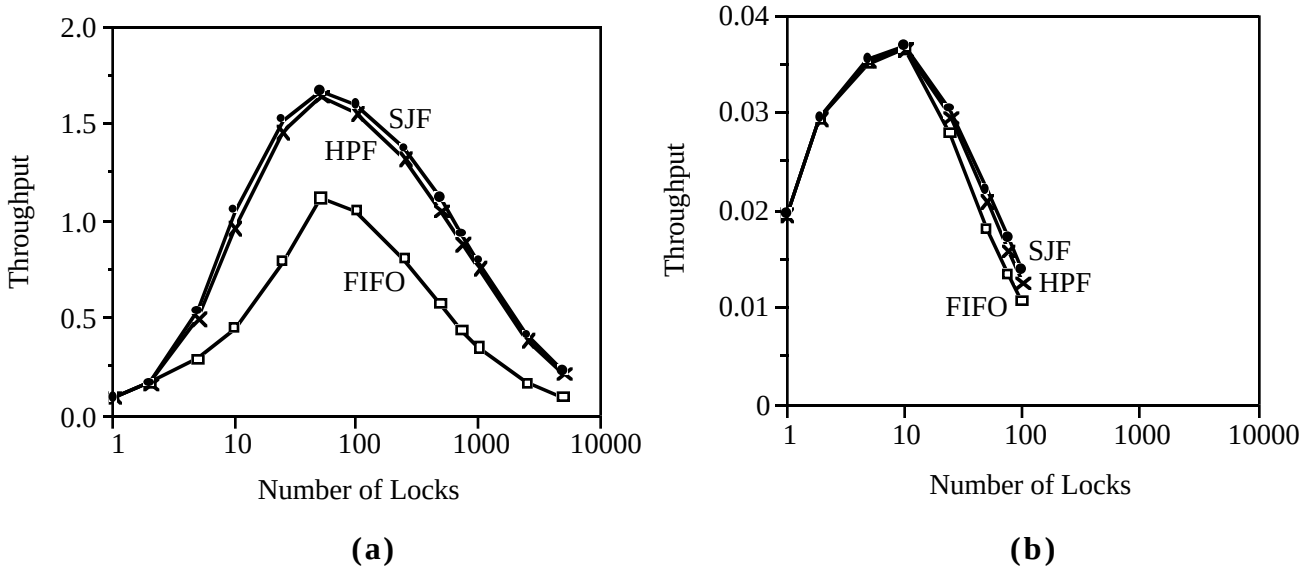

Figure 13 Performance of sub-transaction scheduling algorithms as a function of number of locks **(a)** Small transactions (average transaction size = 1% of database) **(b)** Large transactions (average transaction size = 25% of database)

4.2. Influence of Transaction Scheduling Policies

As shown in Figure 1, transactions wait in the pending queue to be scheduled. The scheduling policy effectively determines which of these waiting transactions should be scheduled next. This section describes the four transaction-level scheduling policies (Section 4.2.1) and their impact on locking granularity (Section 4.2.2).

4.2.1. Scheduling Policies

Transaction scheduling policies can be broadly divided into two classes: policies that work independently of the system state and policies that use the current system state information. We consider two policies belonging to each category - one policy works independently of the transaction characteristics and the other policy requires transaction size information.

System state independent policies

- *First-In-First-Out (FIFO):* The transactions are placed in the order of their arrival to the pending queue. The transaction at the head of the pending queue makes a lock request, the processing of which requires both the processor and I/O resources. If this request is granted, the transaction proceeds to the processing stage; otherwise, it is placed in the blocked queue as explained in Section 2. In either case, the transaction is removed from the pending queue and the transaction behind this transaction makes its lock request next.

- *Smallest Transaction First (STF)*: This policy requires transaction size information and orders the transactions in the pending queue in increasing order of transaction size. Thus at any time, of all the new transactions that have arrived at this queue, only the transaction with the smallest size is allowed to make its lock request. Other than this difference, this policy is similar to the FIFO policy described above. It should be noted that an estimate of transaction size can be obtained. For example, if the transaction involves a select operation, then selectivity characteristics of the relation involved (along with the information on available index structures etc.) can be used to obtain a good estimate of the complexity involved in processing the transaction.

System state dependent policies

Note that a transaction making its lock request for a set of locks takes away processors and I/O servers from transaction processing to lock processing, which is an overhead activity. Since a given database can support only a certain number of transactions depending on the average transaction size, total number of locks in the system etc. when there is large number of transactions in the system this lock processing overhead may cause serious degradation in overall system performance. This degradation increases with increasing number of locks and increasing transaction size and the number of transactions in the system. In order to reduce the adverse effect of this overhead on overall system performance, we use two abstract state dependent threshold policies to study their impact on the desired locking granularity. (Note that policies that are similar in spirit to these abstract policies are used in practice. For example, see [Care90, Moen91].) In these two policies, the transactions waiting in the pending queue are not allowed to make their lock request unless the number of active transactions is less than a specified threshold value. Thus the transactions in the pending queue wait until the system load drops below this value before making their lock requests. This reduces lock processing overhead when there is little chance of having their request granted.

- *Threshold-based First-In-First-Out (T-FIFO):* The transaction at the head of the pending queue is allowed to make its lock request only if the system load (measured in terms of number of active transactions that are currently holding locks) below the threshold level n_a.
- *Threshold-based Smallest Transaction First (T-STF)*: This is the threshold version of the STF policy described above.

The above two policies depend on a threshold value n_a which is computed as follows:

$$n_a = \alpha \frac{dbsize}{avgTransize}$$

where $dbsize$ is the total database size and $avgTransize$ is the average transaction size and α is a parameter that allows n_a to be a fraction of the maximum number of transactions that can be supported by the database.

4.2.2. Results

For these results, FIFO sub-transaction level scheduling policy is used. Other parameters are fixed as in Section 4.1.2.

(a) Effect of Number of Locks

For this set of experiments, α is set at 0.5, which is found to provide peak throughput for the system and transaction parameters used here [Dand91b]. The results are presented in Figure 14.

As discussed in Section 4.1, for the FIFO and STF policies, it is the blocking time that is the major component of the response time. When the number of locks is less than 10, the blocking time associated with the T-FIFO and T-STF policies is the same as that of the FIFO and STF policies, respectively. This is because the threshold value for the two threshold policies is fixed at 10 for the parameters used here. When the number of locks is increased above 10, the blocking time drops and becomes negligible. The reason is that the lock processing overhead is reduced by restricting the transactions waiting in the pending queue from making lock requests when the system load is high.

As in Section 4.1, the synchronization time increases substantially when the number of locks is large. While all the scheduling policies exhibit increased sensitivities to larger number of locks, synchronization time of the FIFO and STF policies increases much more quickly. The reason is that the lock processing overhead increases in direct proportion to the number of locks. Since lock processing has priority over regular transaction processing, the overall response time gets inflated substantially by not only increasing the blocking time but also substantially increasing the synchronization time. For example, when the number of locks is 1000, the synchronization time of the FIFO policy is 20% of the corresponding blocking time.

An important observation is that the type of transaction scheduling policy has only a minor effect on the desired locking granularity. While all the policies support fairly coarse granularity (dividing the entire database of 5000 entities into 10 to 100 lockable entities is sufficient to ensure best throughput), the penalty for not maintaining the optimum number of locks is more with the FIFO and STF policies compared to the other two policies. As shown in Figure 14b the throughput is fairly flat from 10 to 100 locks for the T-FIFO and T-STF policies.

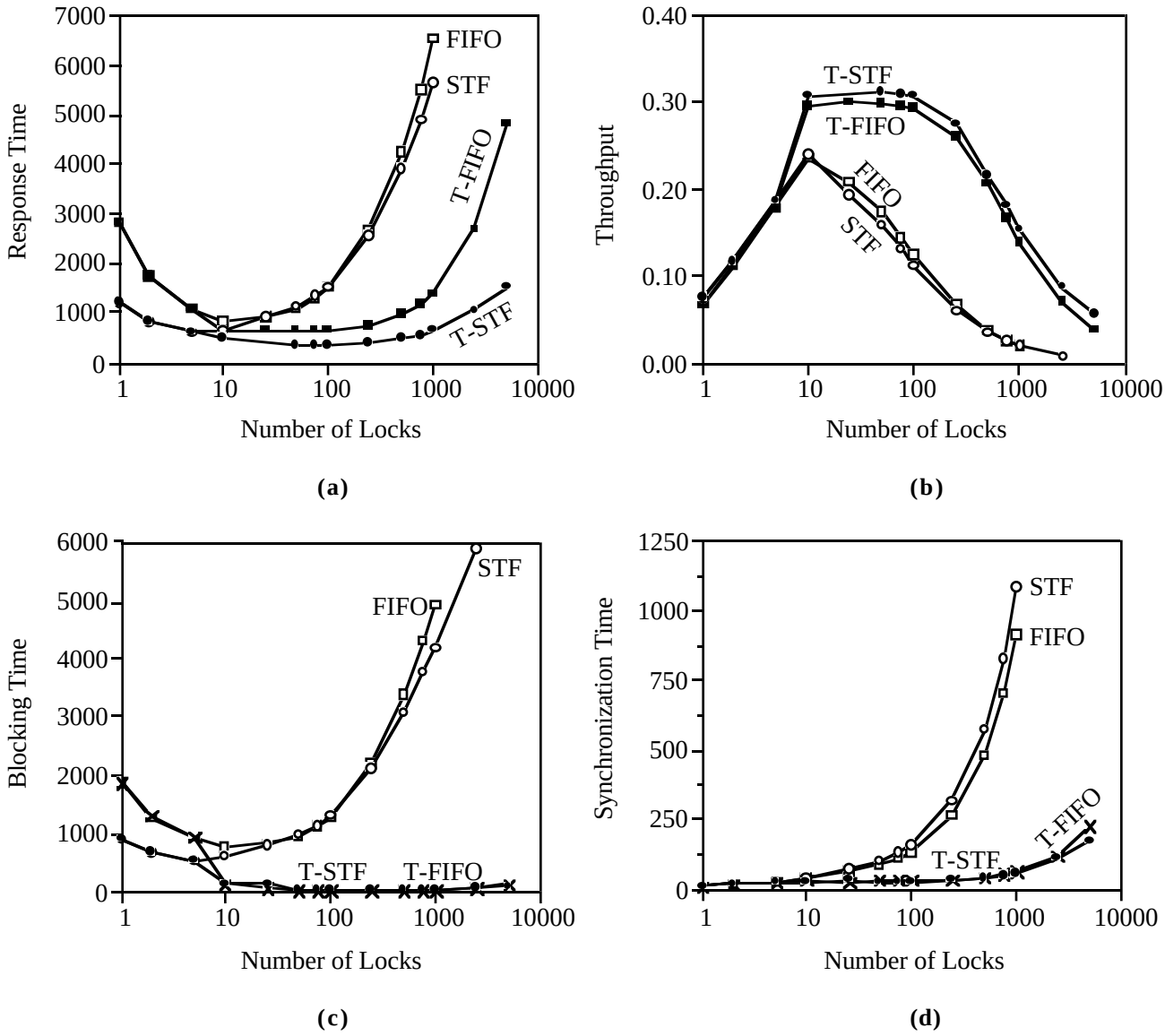

Figure 14 Performance of transaction scheduling algorithms as a function of number of locks

(b) Effects of Transaction Size

The impact of average transaction size is shown in Figure 15. In this set of experiments, as in Section 4.1, the average transaction size is set at 1% and 25% of the database, respectively. For the smaller transaction size workload, α is set at 0.2 and for the larger transaction size workload, α is set at 0.7. Note that these α values were found to provide the peak throughput.

Figure 15 shows the throughput as a function of number of locks. For reasons discussed before, the threshold-based policies perform better. These results show that the observation that the transaction-level scheduling policy has only minor impact on the desired locking granularity is valid even for different transaction sizes. The results further show that for small transactions, the actual transaction scheduling policy has negligible impact on locking granularity. For example, all four transaction scheduling policies provide peak throughput when the number of locks is equal to 50.

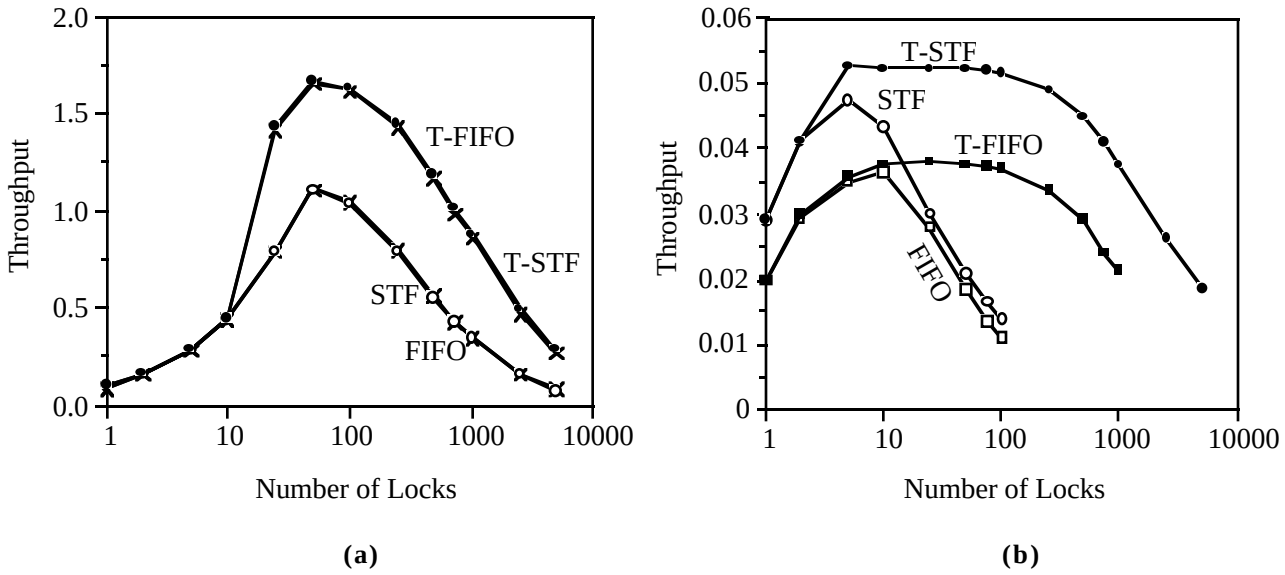

Figure 15 Performance of transaction scheduling algorithms as a function of number of locks **(a)** Small transactions (average transaction size = 1% of database) **(b)** Large transactions (average transaction size = 25% of database)

5. CONCLUSIONS

We have studied the effects of locking granularity on the performance of a class of parallel database systems. In a related study [Ries77, Ries79] on the effects of locking granularity in centralized database systems, it was concluded that coarse granularity is generally preferred except when the transactions access small part of the database randomly (in which case fine granularity is desired).

In general, our results support these conclusions even for large parallel database systems (in terms of number of processors). However, our results show that, for the same workload, finer granularity is desired with increasing system size when the database is accessed sequentially. To increase the performance of parallel database systems, lock contention should be reduced.

When the system is lightly loaded (i.e., the number of transactions in the systems is small) we need to have fine granularity (one lock per database entity) when transactions access the database randomly. However, when transactions access the database sequentially, coarse granularity is desired if the transactions are large (i.e., access large portion of the database); better than coarse granularity is preferred for small and mixed sized (small + large) transactions. For example, providing granularity at the block level and at the file level, as is done in the Gamma database machine, may be adequate for practical purposes.

When the system is heavily loaded (such that the utilization of the CPU and the I/O resources is high) coarse granularity is desirable. For example, providing just file level granularity may be sufficient.

One might think that reducing the lock I/O cost in an I/O bound application might improve the performance substantially. However, results presented here show that reducing the lock I/O cost does not improve the performance of a multiprocessor system substantially (not even when the lock I/O cost is minimized i.e., all information for locking is kept in main memory). Furthermore, our results indicate that horizontal partitioning results in better performance than random partitioning.

We have also shown that the actual scheduling policy used at the sub-transaction level has only marginal effect on the desired locking granularity. In general, FIFO policy exhibits more sensitivity to the locking granularity compared to the other policies considered. At the transaction level, threshold-based scheduling policies perform better than the non-threshold policies. For large transactions, threshold-based policies exhibit a more robust behaviour to locking granularity.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada and Carleton University. A previous version of this paper appeared as [Dand91a].

REFERENCES

[Bern81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Comp. Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.

[Care88] M. J. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proc. VLDB Conf.*, Los Angeles, 1988, pp. 13-25.

[Care89] M. J. Carey and M. Livny, "Parallelism and Concurrency Control Performance in Distributed Database machines," *Proc. SIGMOD Conf.*, 1989, pp. 112-133.

[Care90] M. J. Carey, S. Krishnamurthy, M. Livny, "Load Control for Locking: The 'Half-and-Half' Approach," *ACM PODS Conf.*, 1990.

[Corn86] D. W. Cornell, D. M. Dias, and P.S. Yu, "On Multisystem Coupling Through Function Request Shipping," *IEEE Trans. Software Engrg.*, Vol. SE-12, No. 10, October 1986, pp. 1006-1017.

[Dand91a] S. P. Dandamudi and S. L. Au, "Locking Granularity in Multiprocessor Database Systems," *Proc. Int. Conf. Data Engineering*, Kobe, Japan, 1991, pp. 268-277.

[Dand91b] S. P. Dandamudi and C.Y. Chow, "Scheduling in Parallel Database Systems," *Proc. Int. Conf. Distributed Comp. Systems*, Arlington, Texas, 1991, pp. 116-124.

[Dand91c] S. P. Dandamudi and C.Y. Chow, "Performance of Transaction Scheduling Policies for Parallel Database Systems," *Proc. Int. Conf. Parallel Processing*, St. Charles, Illinois, 1991, Vol. I, pp. 616-620.

[Date89] C. J. Date and C. J. White, *A Guide to DB2*, Addison-Wesley, Reading, Mass., 1989.

[DeWi88] D. DeWitt, S. Ghandeharizadeh, and D. Schneider, "A Performance Analysis of the Gamma Database Machine," *Proc. ACM SIGMOD Conf.*, Chicago, Ill., June 1988, pp. 350-360.

[DeWi92] D. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Comm. ACM*, Vol. 35, No. 6, June 1992, pp. 85-98.

[Eswa76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and L. I. Traiger, "On the Notions of Consistency and Predicate Locks in a Database System," *Commun. ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.

[Fran85] P. Franaszek and J. T. Robinson, "Limitations of Concurrency in Transaction Processing," *ACM Trans. Database Syst.*, Vol. 10, No. 1, March 1985, pp. 1-28.

[Jenq89] B. C. Jenq, B. C. Twichell, and T. Keller, "Locking Performance in a Shared Nothing Parallel Database Machine," *IEEE Trans. Knowledge and Data Eng.*, Vol.1, No. 4, December 1989, pp. 530-543.

[Livn87] M. Livny, S. Khoshafian, and H. Boral, "Multi-Disk Management," *Proc. ACM SIGMETRICS Conf.*, Banff, Alberta., May 1987, pp. 69-77.

[Moen91] A. Moenkeberg and G. Weikum, "Conflict-driven Load Control for the Avoidance of Data-Contention Thrashing," *Proc. Int. Conf. Data Engineering*, Kobe, Japan, 1991, pp. 632-639.

[Poti80] D. Potier and P. Le Blanc, "Analysis of Locking Policies in Database Management Systems," *Commun. ACM*, Vol. 23, No. 10, October 1980, pp. 584-593.

[Ries77] D. R. Ries and M. Stonebraker, "Effects of Locking Granularity in a Database Management Systems," *ACM Trans. Database Syst.*, Vol. 2, No. 3, September 1977, pp. 233-246.

[Ries79] D. R. Ries and M. Stonebraker, "Locking Granularity Revisited," *ACM Trans. Database Syst.*, Vol. 4, No. 2, June 1979, pp. 210-227.

[Rodr76] J. Rodriquez-Rosell, "Empirical Data Reference Behaviour in Data Base Systems," *Computer*, Vol. 9, No. 11, November 1976, pp. 9-13.

[Ryu90] I.K. Ryu and A. Thomasian, "Analysis of Database Performance with Dynamic Locking," *JACM*, Vol. 37, No. 3, July 1990, pp. 491-523.

[Ston86] M. Stonebraker, "The Case for Shared Nothing," *Database Eng.*, Vol. 5, No. 1, 1986, pp. 4-9.

[Tand88] The Tandom Performance Group, "A Benchmark of Non-Stop SQL on the Debit Credit Transaction," *Proc. ACM SIGMOD Conf.*, Chicago, Ill., June 1988, pp. 337-341.

[Tay85] Y. C. Tay, N. Goodman, and R. Suri, "Locking Performance in Centralized Databases," *ACM Trans. Database Syst.*, Vol. 10, No. 4, December 1985, pp. 415-462.

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

[Thom85] A. Thomasian, "Perfromance Evaluation of Centralized Databases with Static Locking," *IEEE Trans. Software Engrg.*, Vol. SE-11, No. 4, April 1985, pp. 346-355.

[Thom90] A. Thomasian and I.K. Ryu, "A Recursive Solution Method to Analyze the Perfromance of Static Locking Systems," *IEEE Trans. Software Engrg.*, Vol. SE-15, No. 10, October 1989, pp. 1147-1156.

[Yao77] S. B. Yao, "Approximating Block Accesses in Database Organizations," *Comm. ACM*, Vol. 20, No. 4, April 1977, pp. 260-261.

[Yu90] P. S. Yu and D. M. Dias, "Concurrency Control Using Locking with Deferred Blocking," *Proc. Int. Conf. Data Engineering*, 1990, pp. 30-36.