Processing Multi-join Query in Parallel Systems

Kian-Lee Tan **Hongjun Lu**

Department of Information Systems and Computer Science
National University of Singapore
Singapore 0511

Abstract

In parallel systems, a number of joins from one or more queries can be executed either serially or in parallel. While serial execution assigns all processors to execute each join one after another, the parallel execution distributes the joins to clusters formed by certain number of processors and executes them concurrently. Both approaches employ parallelism to improve system performance. However, data skew may result in load imbalance among processors executing the same join and some clusters may be overloaded with time-consuming joins. As a result, the completion time will be much longer than what is expected. In this paper, we propose an algorithm to further minimize the completion time of concurrently executed multiple joins. For this algorithm, all the joins to be executed concurrently are decomposed into a set of tasks that are ordered according to decreasing task size. These tasks are dynamically allocated to available processors during execution. Our performance study shows that the proposed algorithm outperforms the previously proposed approaches, especially when number of processors increases, high skewness is present in the relations to be joined and relation sizes are large.

Introduction

Today's DBMS will have to deal with complex queries which take a long time to complete. The conventional von-Neumann architecture will soon reach its speed limit, and parallelism represents the most feasible alternative to achieve any significant breakthrough in performance. With the advances in hardware technology and computer architecture, a large number of parallel computers are already being employed to solve database applications [3, 4, 5, 19, 21].

In relational database systems, the most costly and frequently used (and hence important) operation is the *join* operation. With novel applications, such as artificial intelligence, graphics and geometric modeling, it will not be uncommon to have queries involving many relations (and therefore many joins). Object-oriented database systems are another class of potential applications that will generate many joins. Even in relational systems, the use of views can lead to an increase in the number of joins in the query being processed. In all cases, we have a set of joins which can be processed collectively. While optimization of multi-join queries has been extensively studied in uniprocessor environment [6, 7, 18, 20], the development of effective schemes to exploit parallelism to process multi-join queries, in particular inter-join parallelism, are only beginning to be explored [2, 11]. Intra-join parallelism has also received much attention in recent years [10, 15, 16, 17, 22, 24].

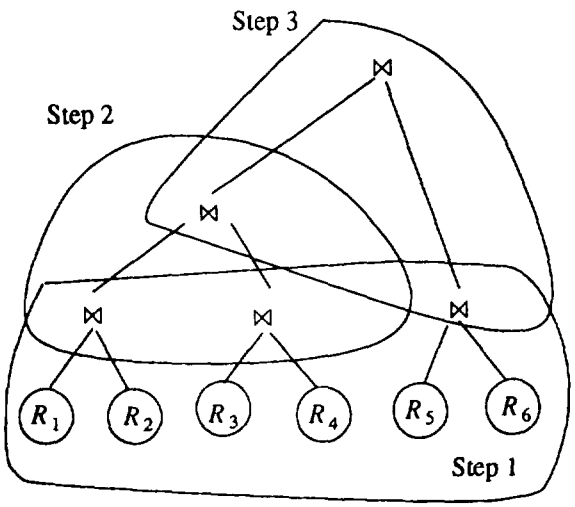


Figure 1. A sample multi-join query plan.

In this paper, we assume that we are given a query execution plan where a set of joins could be executed concurrently. For example, in Figure 1, the query plan consists of 3 sets of joins that could be executed con-

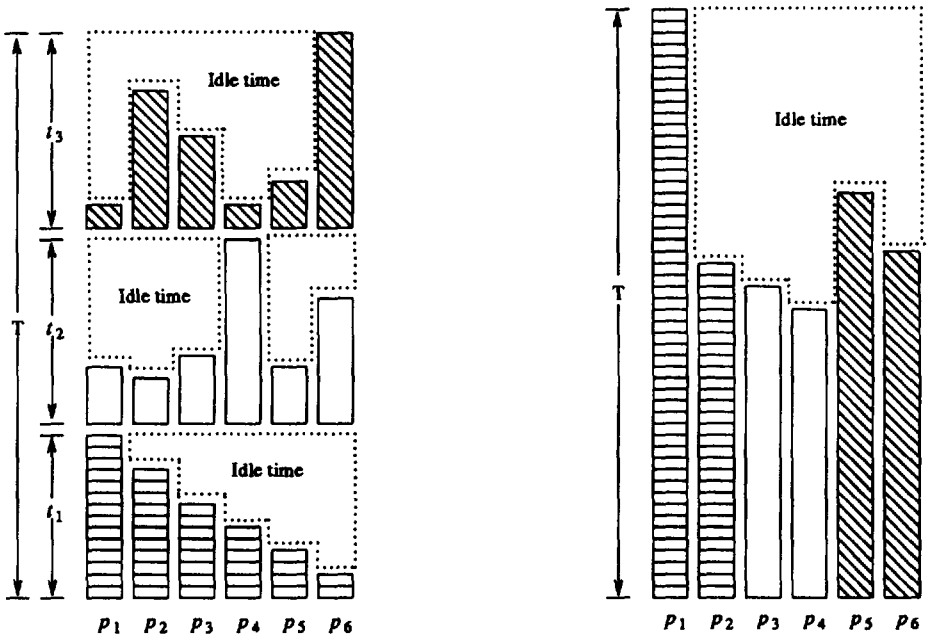


Figure 2. Conventional multi-join strategies.

currently. Each set is executed in a step and the steps are ordered based on the dependency between the joins in the sets. The cost of a plan is thus

$$Cost(QEP) = \sum_{i=1}^{n} Cost_i$$

where $Cost_i$ is the cost for step i and is equal to the cost of the most expensive join operation in the step. Since each step is executed in the same manner, it suffices for us to examine only a step of the query plan as is done in [2]. Thus, we reduce the multi-join query processing problem to the following problem:

> *Given a set of joins,* $J = \{ J_1, J_2, \ldots\ldots, J_n \}$ *where* $J_i = R_i \bowtie S_i$, *that are scheduled to be performed concurrently on p processors, find the least completion time.*

The two approaches which has been adopted in the literature to solve this problem are

1) Execute the set of joins as a series of single joins one after another. This corresponds to the *one-parallel strategy* in [2] where all processors form a single cluster and a parallel join algorithm is employed to execute each join. If the elapsed time for J_i is t_i, for $1 \le i \le n$, then the completion time for J is

$$T = \sum_{i=1}^{n} t_i$$

2) Partition the set of processors into n clusters of processors with $p_1, \ldots\ldots, p_n$ processors respectively, such that $\sum_{i=1}^{n} p_i = p$, and the join J_i is allocated to the i^{th} cluster, for $1 \le i \le n$. Within each cluster, a parallel join algorithm is used to perform the join. Thus, both inter- and intra-join parallelism are employed in this strategy. In [2], the *n-parallel strategy* distributes the number of processors evenly across the number of joins, that is each cluster contains (approximately) the same number of processors. Lu, Shan and Tan employ a load-balancing scheme in the allocation of processors to joins [11]. Their approach estimates the time for each join and allocates more processors to time-consuming joins. If the i^{th} processor takes t_i time to complete the load it is allocated, then the completion time for J is

$$T = \max_{i=1}^{p}(t_i)$$

Example 1. Let $J = \{ J_1, J_2, J_3 \}$ and $p = 6$. Figure 2 shows the two approaches to execute J. In Figure 2(a), the *one-parallel strategy* is employed. Assuming a hash-based join algorithm, each join is split into, say, 6 tasks and these tasks are then allocated to the 6 processors. We define a *task* to be a *single* operation and the data associated with the operation. Hence, the *join* operation between relations R_1 and R_2 is a task. However, using the definition of a task recursively, we can generate

more tasks from a single task by duplicating the operation and splitting its data such that some conditions (which depends on the operations) be satisfied. If we perform the join operation using hash-based algorithm, the join of a bucket of R_1 with R_2 forms a task and the number of tasks is equivalent to the number of buckets. Suppose the join J_i needs time t_i to execute, the completion time for J is $T = t_1 + t_2 + t_3$. In Figure 2(b), the *three-parallel strategy* is used and two processors are assigned to perform a join. Within each cluster, the join load is split among the two processors. The completion time for J is determined by processor p_1.

The two approaches are effective only when the processing load at each processor is approximately the same. In this case, the completion time to execute J is near-optimal if not optimal. However, in real situation, the processing load across the processors will not be the same. In such cases, underloaded processors will soon become idle. In *one-parallel* approach, the skew in the data may cause a load-imbalance across the processors [9, 23]. Several algorithms that handle skew have been proposed in the literature [14, 24]. The situation is worse in multi-join scenario. As shown in Figure 2(a), even when the amount of idle time is small for a single join, the amount of idle time accumulated could be substantial as the number of joins increases. When the *n-parallel* strategy is used, load imbalance arises when some clusters may be assigned more time-consuming joins (each may involve two large relations and the join selectivity is high). Moreover, as in the *one-parallel* case, within each cluster, data skew will result in some processors within the cluster being heavily loaded. For example, in Figure 2(b), the join allocated to processors p_1 and p_2 is the most time-consuming. The data skew for the join further worsen the overall performance of the system. All these indicate the need to develop new algorithms to evenly distribute the processing load across the processors so as to minimize the completion time of executing the set of joins in J by reducing the idle time within each processor.

In this paper, we propose an algorithm — *M*ulti-join *I*nterleave e*X*ecution — that better utilizes the system resources when executing multiple joins. *MIX* first generates a set of tasks from all the joins. At runtime, these tasks are acquired by idle processors and executed. The main feature of *MIX* is that there is no predefined set of processors to execute a join. Tasks from a join may be processed by several processors and all processors may be processing different tasks from different joins at any one time. Our performance study shows that *MIX* performs best for large number of processors, large relation sizes and high skewness of join attributes.

In the next section, we describe the proposed algorithm *MIX*. In section 3, we study the effectiveness of the algorithms. We summarize this research and briefly discuss some future work in section 4.

Multi-Join Processing Strategy

In this section, we describe the interleaved approach for multi-join processing in parallel systems. We assume the system architecture to be *shared disk (SD)*. In SD system, each processor has its own private memory but each processor can directly access any disk. The processors cooperate by message passing through an interconnection network. Each relation is horizontally partitioned across the disks in the system. In [1], it is shown that SD system performs as well as a *shared nothing (SN)* system. Moreover, designing a SD system is easier than designing a SN system because the database partitioning problem does not arise. In addition, a SD machine is more amenable to load-balancing [12]. Though the *shared everything (SE)* system is shown to outperform SN system and SD system, it is limited by the bus and memory bandwidths. Thus, SD system is more favorable over the SE system when the throughput requirements were too large for SE. We assume that the system has a certain amount of global memory and the global memory is used to keep up-to-date information of the tasks. This assumption may be relaxed by using a shared-disk instead (I/O cost incurred in accessing this information will be small compared to that for the data). We also assume a locking mechanism is used to regulate access to the global memory.

It is assumed that the system uses conventional disk drives for secondary storage and databases (relations) are stored on these disk storage devices. Both disks and memory are organized in fixed-size pages. Hence, the unit of transfer between the secondary storage and memory is a page. We assume that the relations to be joined are, initially, evenly distributed among the processors to facilitate full concurrent access to the relation.

MIX is an extension of the task-oriented approach to query processing. The task-oriented query processing approach comprises three phases: 1) *task generation* phase where a query is decomposed into independent tasks, 2) *task allocation* phase where these tasks are allocated to processors based on some criterion such as balancing the tuple sizes of data associated to tasks [13] or balancing the estimated execution time of tasks [24], and 3) task *execution* phase in which all the processors execute the tasks allocated to it concurrently and independently. *MIX*, however, has two phases and is different in the following ways. First, in the task generation phase, *all* the joins to be executed in parallel are decomposed into tasks before phase 2 begins. In this way, we have a large collection of tasks from all joins to be

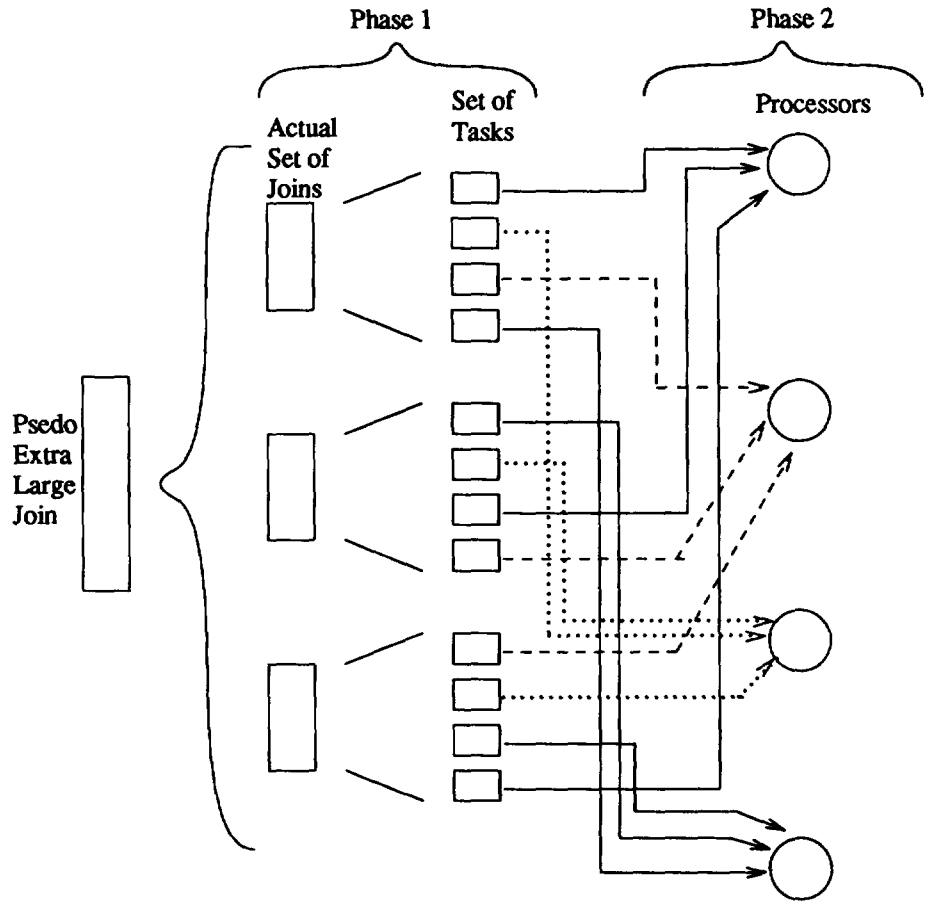
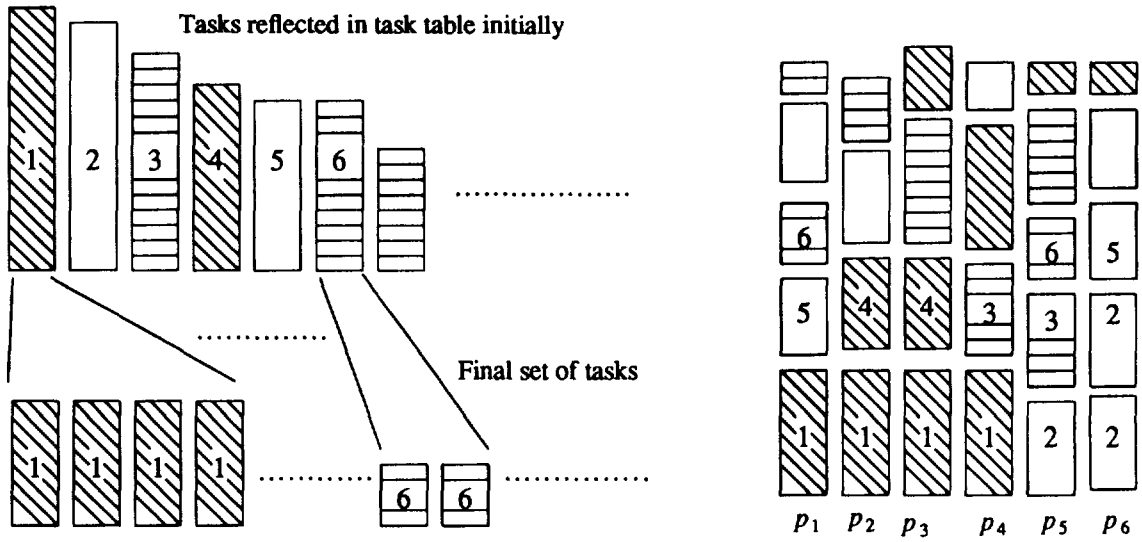


Figure 3. Algorithm *MIX*.

acquired in phase 2. Second, we combine phases 2 and 3 of the task-oriented approach to arrive at a *dynamic* method in which task allocation becomes *demand-driven*. That is instead of allocating all the tasks to processors prior to their execution as in previous approach, in our approach, a processor *acquires* a task to be processed at runtime. Each processor has, at any time, at most one task to process and it does not know which is the next task until the current task finishes. Once the current task is completed, the processor will acquire another task until there is no more tasks. In this way, a processor is idle only when all tasks are allocated and not, as in previous approaches, when tasks of a particular join are exhausted. Third, since we have a collection of tasks from all joins, joins may be executed in interleaved fashion, that is tasks from a join may be executed by several processors and all processors may execute different tasks from different joins at any one time. Hence, there is no predefined set of processors to perform a join and the number of processors that perform a join varies at runtime. In addition, both inter-join and intra-join parallelism are exploited.

The main idea of the algorithm is to view the set of joins to be executed concurrently as (large) tasks of a single (*pseudo extra large*) join query. Each of these tasks (joins), however, needs to be further partitioned to generate more (sub-)tasks. If one task is very costly, then the processor that acquires the task may be the one to determine the execution time of the joins. Algorithm *MIX* remedies this problem using a load-balancing approach similar to that employed by Omiecinski [14]. In [14], a task may be allocated to several processors. However, we generate a number of (sub-)tasks instead of allocating the tasks to the processors. This is done as follows:

> Let *mem* denotes the size of memory available at each processor sufficient for an in memory hash table to be built for join processing. Then a task with the smaller bucket size *bsize* may produce k (sub-)tasks where
>
> $$k = \left\lceil \frac{bsize \cdot f}{mem} \right\rceil \qquad (1)$$
>
> where $bsize \cdot f$ represents the size of the hash



(a) Task splitting

(b) Task acquisition

Figure 3. Task splitting and acquisition.

table for a bucket of size *bsize*.

Once all tasks are generated, these tasks are acquired by processors and executed. Figure 3 illustrates this process. We describe the two phases of algorithm *MIX* here.

Phase 1. Task generation. In the task generation phase, all the joins are partitioned into tasks one after another. For each join, $R^i \bowtie S^i$, each processor will reads approximately $|R^i|/p$ pages of $|R^i|$ and hashes the tuples to the appropriate output buffers (at least one for each task). Once an output buffer is full, it is written out to the disks. The pages of a task is striped across all the disks in the system to avoid disk "hotspot". This can be done by writing the x^{th} page to disk $(x-1)\ MOD\ p+1$. The same hash function is used to create partitions for relation S^i. Let R^i_j (S^i_j) denotes the subset of R^i (S^i) that resides at disk j, for $1 \le j \le d$. We call the portions of R^i_j (S^i_j) that hash into the k^{th} output buffer the sub-buckets of R^i_j and denote them as R^i_{jk} (S^i_{jk}), where $1 \le j \le d$ and $1 \le k \le B$. Thus $\bigcup_{j=1}^{d} R^i_{j1}, \bigcup_{j=1}^{d} R^i_{j2}, \ldots, \bigcup_{j=1}^{d} R^i_{jB}$ are the partitions of R^i. The join result of R^i and S^i corresponds to the union of the join results of the respective partitions of R^i and S^i, that is

$$R^i \bowtie S^i = \bigcup_{k=1}^{B} ((\bigcup_{j=1}^{d} R^i_{jk}) \bowtie (\bigcup_{j=1}^{d} S^i_{jk}))$$

The join execution of the pair of partitions $(\bigcup_{j=1}^{d} R^i_{jk}) \bowtie (\bigcup_{j=1}^{d} S^i_{jk})$ corresponds to an independent subtask that generates portion of the join result.

During the partitioning of data, each sub-bucket on each disk has associated with it a *directory*. The directory for a sub-bucket stores the *disk identifier* and *page identifier* for pages belonging to a partition. With this directory, a processor assigned a partition will have direct access to the pages. Moreover, such a structure has been shown to facilitate load-balancing in the joining phase [12]. It should be noted that the sizes of the directories is not large. Assuming the disk identifier requires 1 byte and the page identifier requires 2 bytes, a 4 Kbytes page can house the addresses for more than 1300 pages of data.

Once the partitioning is performed, each processor will store the information of the sub-buckets in the global memory. Such information includes 1) the join number, 2) the task number, 3) the size of the sub-buckets of the source and target relations and 4) the addresses to the directories of the sub-buckets. A processor (with smallest index) will then build a single *task table* containing the task number with its associated information for a partition of R^i and S^i. The size of each partition is the sum of the corresponding sub-buckets. The addresses of the directories are stored as it is.

Next, the same processor (with smallest index) will sort the task table in non-ascending order of the size (in pages) of the smaller of the two buckets of the tasks. Where there is a tie, the order is according to the non-ascending order of the size (in pages) of the larger of the two buckets. Once the tasks are sorted, for every task whose smaller bucket size exceeds the memory available, the task is split into k subtasks (as determined by Equation (1)) by dividing the smaller bucket size into k buckets and duplicating the larger bucket size. In this way, the

set of tasks is increased by $k - 1$ tasks. It should be noted that while a task may produce k (sub-)tasks, these tasks may not necessarily be processed by k different processors. This is so because phase 2 is dynamic.

Phase 2. Task acquisition and execution. This is the joining phase of the algorithm. During the joining phase, a free processor reads the global information and acquires a task to process. Once a task is available, the directories associated with the sub-buckets of R^i and S^i that correspond to this task are collected at the processor. The directories from all disks are then linked together. In this way, each processor has the addresses to all pages that correspond to the task. The task is then executed, that is the join is performed, as in a uniprocessor environment. Any uniprocessor join algorithm may be employed. In our study, we use the *hash-based nested loops join* (HNL) algorithm for simplicity. The result can be easily generalized. This algorithm is shown to be superior over the other algorithms when we handle medium-sized source relations, that is the size of the source relation is no more than 5 times the size of the memory [13]. The algorithm comprises two phases: 1) Several pages of the source relation, which is determined by the available memory, is read and the hash table is built; 2) The whole of target relation is read a page at a time and each tuple is probed for joinability with the partially staged hash table. These two phases are repeated until all the pages of the source relation are read.

Whenever a processor finishes the processing of the allocated task, it acquires a new task. At the same time, it will update the global information — increment the counter for the join that indicates the number of tasks completed. This process of *acquisition and execution* of tasks is repeated until all the B tasks have been allocated. The execution time to perform the joins is then determined by the processor that finishes last.

Figure 4 illustrates how tasks are "chopped" and allocated to processors (on hind-sight, that is when processings are completed). There are 3 joins to be executed concurrently, each of which produces 6 tasks (tasks from the same join are shown with the same fillings). The sorted order of the collections of tasks is represented by the height of the tasks (we assume in this Figure that the height also corresponds to the execution time). Assume tasks labeled 1 — 6 in Figure are split into 4, 3, 2, 2, 2, 2 tasks, we have the allocation as shown in Figure. We see that the 3 (sub-) tasks of task 2 are processed by 2 processors only. We also see that the 2 tasks from the same join in processor 1 are not executed consecutively.

An advantage of the algorithm *MIX* is that the number of processors allocated to a query may vary at runtime. This is also dependent on the collection of joins to be processed concurrently. The same join, when executed with different set of joins may be allocated different number of processors. This is possible for the following reasons: a) there is no predefined set of processors to perform any join, b) the collection of and ordering of tasks may be different and c) the acquisition and execution phase is dynamic.

A Performance Study

To study the effectiveness of the proposed algorithms, we conducted a simulation study. We vary the skewness of the join attribute (which follows the Zipf-distribution), the number of processors involved and relation sizes of joins. We also vary the number of joins to be performed in parallel. We also use the following two algorithms as references:

Algorithm Seq. This algorithm uses intra-join parallelism only, that is all processors are used to perform a join. All the joins are executed serially.

Algorithm Par. This algorithm employs both intra-join and inter-join parallelism. It distributes the number of processors into clusters of (approximately) equal size and allocate each join to a cluster.

To evaluate the performance of the proposed algorithm, we assume that the values of the join column follow a *Zipf-like distribution* [8]. For a relation R with a domain of D distinct values, the i^{th} distinct join column value, for $1 \le i \le D$, has such number of tuples as given by the following expression:

$$\|D_i\| = \frac{\|R\|}{i^{\theta} \cdot \sum_{j=1}^{D} \frac{1}{j^{\theta}}} \qquad (2)$$

where θ is the skew factor. When $\theta = 0$, the distribution becomes *uniform*. With $\theta = 1$, it corresponds to the highly skewed *pure* Zipf distribution [25]. Though the join column is skewed, we assume that the relations to be joined are, initially, evenly distributed among the processors to facilitate full concurrent access to the relation.

We also study the effect of different correlations between the skew values in the two relations. Two types of correlation are modeled: *Ordered correlation* and *random correlation*. For ordered correlation, the values in both the attributes have the same ranking sequences. For example, the highest ranked value in attribute R_A of relation R is also the highest ranked value in the corresponding attribute S_A in relation S. On the other hand, random correlation randomly correlates the attribute in R and S.

As in [24], we substitute the *actual* distribution of data into the cost formulas for the join algorithm HNL to compute the elapsed time. That is, the actual number of pages (and tuples) of the source, target and result relations of each partition are used in the computation. The

distribution of data for the source and target relations is generated using Equation (2). A hashing function is then used to partition the relations into B partitions. Two hashing functions are used — *range-based* where partition i contains tuple in the range $\left\lfloor 1+(i-1)\times\frac{D}{B}\ ,\ i\times\frac{D}{B}\right\rfloor$, for $1 \le i \le B$, and *modulo-based* where tuple with value tu belongs to partition [tu MOD B + 1]. Both hashing functions show similar results. To compute the result size, we introduce a selectivity factor of the join operation, *Sel*. *Sel* gives the number of distinct tuples in the source relation that has a matching value in the target relation. For example, $Sel = 0.5$ means that for each tuple in the source relation, there is a probability of 0.5 that it will find a match in the target relation. *Sel* is modeled using a uniform distribution UD(0,1). When the ordered correlation is used, for each distinct value of the source relation examined, if the value of UD(0,1) is in the range 0 — *Sel*, the corresponding distinct value in the target relation is a match and the join result size equal to the product of the number of tuples with this value in the source and target relations. Otherwise, the result size is 0. For random correlation, for each distinct value of the source relation examined, a random distinct value is picked from the target relation and the result size is computed in the same manner as that of ordered correlation.

For purpose of illustrating the performance study here, the following test values are used: The system has 16 disks. The number of processors is varied from 2 — 32 and each processor has 512K of memory. The CPU processing rate is 10 MIPS and the I/O bandwidth 10 Mbytes/s. It takes 50 instructions to compare two keys/attributes. The computation of hash function of a key costs 100 instructions. The time to move a tuple in memory is 500 instructions. We also vary the number of tuples in relation R (S) from 20K — 300K. Each page of relation R(S) and the resultant relation contains 50 tuples. The number of distinct tuples in relations R and S is 10K. The skew factors are varied from 0.1 — 1.0. The selectivity factor (*Sel*) is 0.5. The number of joins considered here are 2 and 3 and each join is partitioned into 50 buckets initially. We also vary the number of joins and obtained similar results as reported here.

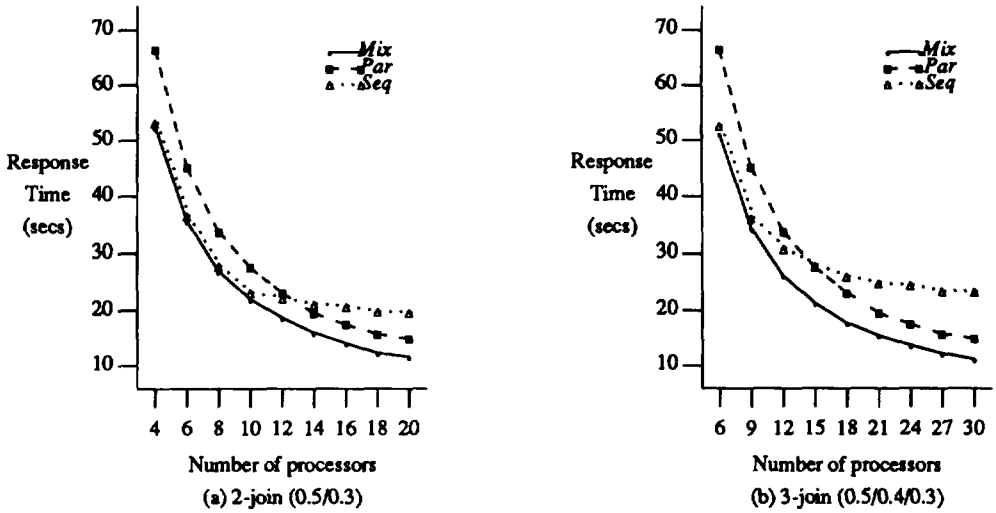
Experiment 1: Vary the number of processors

In this experiment, we study the performance of the algorithms as the number of processors vary. Graph 1 shows the result of the experiment. For both 2-joins and 3-joins, the response time for all the algorithms decreases as the number of processors increases. We also made several interesting observations. First, algorithm *Seq* and *Par* outperforms each other depending on the number of processors. This result differs from that presented in [2]. Deen, in his experiments for 2-joins and 3-joins, concluded that it is better to form a single cluster of parallel processors and to carry out the multi-join in series of single joins one after another. This is because the study is conducted on a small-scale multiprocessor system, that is the number of processors is small. We agree with this result for small number of processors. For *Seq*, the load is spread across all the processors. For *Par*, the cluster size for a join is even smaller. This will result in the load imbalance across clusters being more significant. However, we have observed in our experiment that with large number of processors, *Par* is superior. For *Seq*, when the number of processors p goes beyond a certain number, say q, the most expensive task dominates performance. By increasing the number of processors beyond q will provide no significant gain in response time. On the other hand, when the number of processors p is large, it becomes beneficial for k joins to be executed concurrently using p/k processors per join since the dominating task may still remains dominating within the cluster.

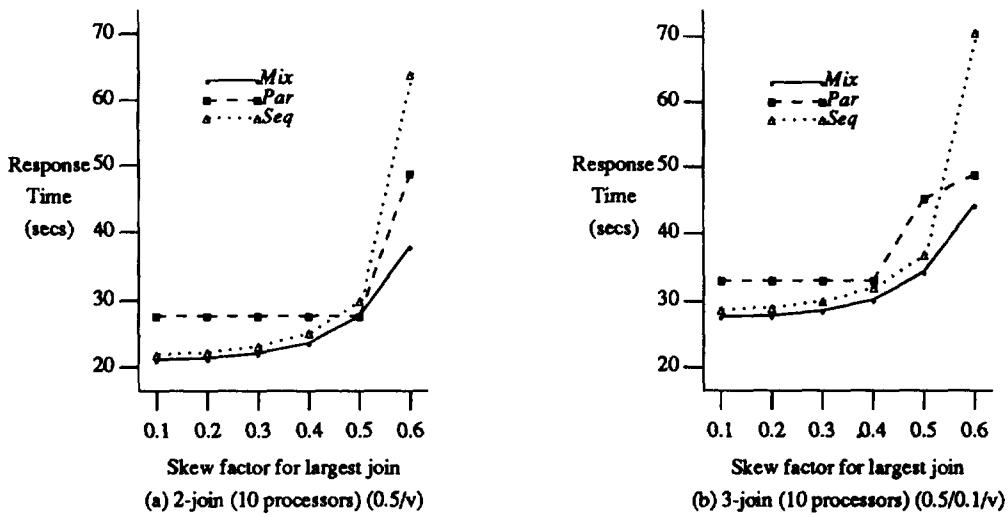
Second, algorithm *MIX* outperforms both *Seq* and *Par*. This shows that interleaving the execution of tasks from different joins better utilizes resources and reduces the idle time. For *Seq*, since joins are performed one after another, a processor becomes idle as soon as all tasks from a join are acquired. If there are still joins to be processed, the processor will be busy only when these joins are executed. For *Par*, a processor within a cluster becomes idle when all tasks of the join allocated to the cluster are exhausted. Even if there may be tasks from other joins at other clusters, no transfer of task is permitted. On the other hand, for *MIX*, all processors acquire tasks from a large collection of tasks (from all joins). In this way, a processor will be freed only when there is no more tasks in the system. Next, for *Seq* or *Par*, a heavy load (where task size > memory available), which may dominate performance, is acquired and processed by a single processor. However, in *MIX*, the same load is spread across several processors, thus balances the load, that is any dominating task would have been "chopped" into several smaller tasks and executed by several processors.

Experiment 2: Effects of load imbalance due to skew factor

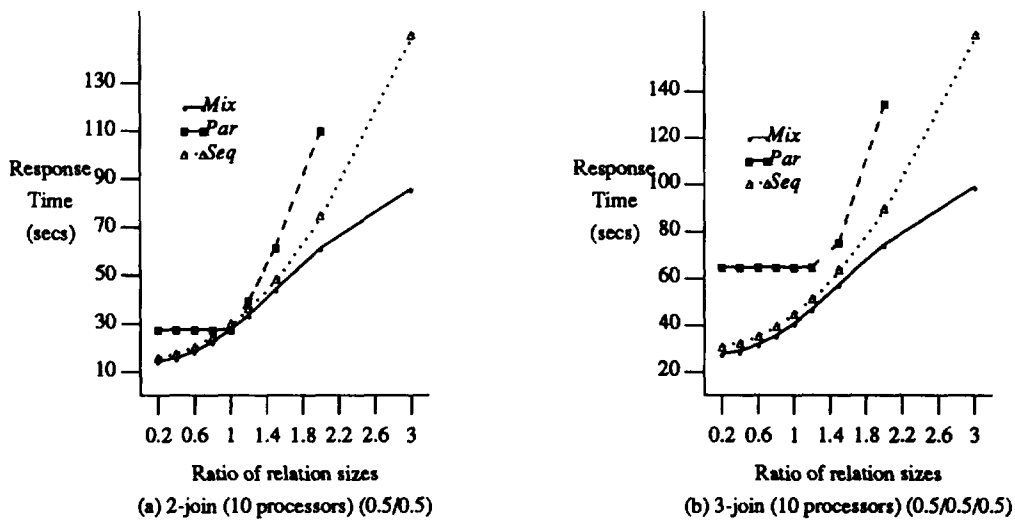
The skewness of data is an important factor that will cause load imbalances. This experiment studies the behaviors and relative performance of the algorithms when the skew factor varies. In our experiment, for m-joins, we fix the skew factors for the $(m-1)$-joins and vary the skew factor of one join. Graph 2 shows the result of our simulation. The horizontal-axis represents the skew factor of the data for the join for which we vary. We have



Graph 1. Experiment 1: Vary the number of processors.



Graph 2. Experiment 2: Effects of skew factor.



Graph 3. Experiment 3: Effects of varying relation sizes.

two observations regarding algorithm *Par*. FIrst, algorithm *Par* perform worse than algorithm *Seq* when the skew factors for one of the joins is medium and the rest low. The reason is that when a smaller cluster (number of processors) is assigned the join with medium skew factor, the cluster becomes the more loaded cluster. On the other hand, algorithm *Seq* is able to spread the load of the expensive join across a larger cluster. Second, the curve is flat initially. This is so since the loaded cluster determines the execution time.

When the skew factors are high (for 2 of the joins), some tasks (buckets) dominate performance. Hence the execution time for such tasks is approximately the same for both algorithm *Seq* and *Par*. However, since *Par* distributes the join across clusters of processors, the completion time remains dominated by the loaded tasks. On the other hand, *Seq* executes the joins serially, thereby incurring higher completion time.

Algorithm *MIX* performs best again in this experiment since it divide the load of an expensive task across several processors. It is able to exploit the large collection of tasks from all joins and attempts to balance these tasks across all processors.

Experiment 3: Effects of load imbalance due to relation sizes

For this experiment, we would like to study the effects of load imbalance due to relation sizes. We present here the result when the relations involved in a join has the same relation sizes, that is only the joins involved have different relation sizes. The other case when each relation of each join has different sizes shows similar behavior.

Graph 3 shows the result of the experiment. For the experiment, we vary only the relation sizes for one join and keeps the other relation sizes the same. From the graph, we see that algorithm *MIX* is superior to the other algorithms.The performance gain over *Seq* is small when the relation sizes are small. This is so since there is little opportunity for splitting tasks. On the other hand, when relation sizes becomes large, the number of tasks that needs to be split increases, allowing *MIX* to balance the load across the processors. It should be pointed out that the performance of *Par* is worse than *Seq* since the number of processors is small. The reason is the same as that in Experiments 1 and 2.

Conclusion

In this paper, we address the problem of minimizing the execution time for a collection of joins in parallel systems. Such a collection may be obtained when several single join queries or when complex queries (such as multi-join queries) which can be decomposed to multiple join queries are to be executed. The conventional method is to form clusters of processors and allocate the joins to the clusters. In the *serial* algorithm, all the processors form a cluster and all joins are executed serially one at a time. Thus only intra-join parallelism are exploited. On the other hand, the *k-parallel* algorithm exploit both intra-join and inter-join parallelism by distributing the k joins into k clusters of processors; each cluster has (approximately) equal number of processors. Our proposed algorithm, *MIX*, is an extension of the task-oriented approach. Each of the join is first decomposed into tasks. These large collection of tasks are then ordered according to the smaller of the relation sizes of the task. Large tasks may be "chopped" into k smaller (sub-)tasks such that a in memory hash table may be built. Hence, k processors (not necessarily unique) are needed to execute these (sub-)tasks. At runtime, these tasks are dynamically acquired by available processors one at a time. In this way, tasks from different joins may be interleaved during execution. Once a processor finished the execution of the assigned task, it request for another one until there is no more tasks. The main feature of *MIX* is that inter-join and intra-join parallelism are exploited without predefining the set of processors to perform the joins. The number of processors may vary at runtime.

We study the performance of algorithm *MIX* in shared-disk environment. The data skew is modeled by Zipf-like distribution. Moreover, we use the actual data distribution of each task to measure the elapsed time in our simulation. Our results show that algorithm *MIX* is superior to conventional methods in all cases — with different number of processors, skew factors and relation sizes. We also observe that distributing joins to clusters of processors may outperform that of serial execution when the number of processors is large.

We plan to extend this study in several ways. First, we have assumed a set of joins to be executed concurrently. This, however, may not be suitable when the number of joins is large. We may explore how to determine the number of joins to be performed at each step. Second, we have considered only joins. We are planing to extend our study to queries. An immediate problem is how the ordering of tasks within queries may affect algorithm *MIX*. Third, we would like to study the possibility of pipelining.

References

[1] Bhide, A., "An Analysis of Three Transaction Processing Architectures," *Proc. 14th VLDB Conf.*, Los Angeles, CA., Aug. 1988, pp. 339-350.

[2] Deen, S. M., Kannangara, D. N. P. and Taylor, M. C., "Multi-join on Parallel Processors," *Proc. 2nd Intl. Symp. Databases in Parallel and Distributed*

Systems, Dublin, Ireland, July 1990, pp. 92-102.

[3] DeWitt, D. J., *et al.*, "The GAMMA Database Machine Project," *IEEE Trans. Knowledge and Data Engineering*, Vol. 2, No. 1, Mar. 1990, pp.44-62.

[4] Englert, S., *et al.*, "A Benchmark of Nonstop SQL Release 2 Demonstrating Near-linear Speedup and Scaleup on Large Databases," Tandem Tech. Rep. 89.4, May 1989.

[5] Hsiao, D. K., Advanced Database Machine Architecture, Prentice Hall, 1983.

[6] Ioannidis, Y. E. and Kang, Y., "Randomized Algorithms for Optimizing Large Join Queries," *Proc. SIGMOD 90*, May 1990, pp. 312-321.

[7] Krishnamurthy, R., Boral, H., and Zaniolo, C., "Optimization of Nonrecursive Queries," *Proc. VLDB 86*, Kyoto, Aug 1986, pp. 128-137.

[8] Knuth, D. E., The Art of Programming, Vol. 3: Sorting and Searching, Addison-Wesley, 1973.

[9] Lakshmi, M. S., and Yu, P. S., "Effectiveness of Parallel Joins," *IEEE Trans. Knowledge and Data Engineering*, Vol. 2, No. 4, Sept. 1990, pp.410-424.

[10] Lu, H., Tan, K. L. and Shan, M. C., "Hash-based Join Algorithms for Multiprocessor Computers with Shared Memory," *Proc. VLDB 90*, Brisbane, Australia, Aug. 1990, pp. 198-209.

[11] Lu, H., Shan, M. C., and Tan, K. L., "Optimization of Multi-Way Join Queries for Parallel Execution," *Proc. VLDB 91*, Barcelona, Spain, Sept. 1991.

[12] Lu, H. and Tan, K. L., "Dynamic and Load-balanced Task-Oriented Database Query Processing in Parallel Systems," *EDBT 92*, Mar. 1992.

[13] Nakayama, M. and Kitsuregawa, M., "Hash-partitioned Join Method Using Dynamic Destaging Strategy," *Proc. 14th VLDB Conf.*, Los Angeles, CA., Aug. 1988, pp. 468-478.

[14] Omiecinski, E., "Performance Analysis of a Load Balancing Relational Hash-Join Algorithm for a Shared Memory Multiprocessor," *Proc. VLDB 91*, Barcelona, Spain, Sept 1991.

[15] Qadah, G. Z., and Irani, K. B., "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine," *IEEE Trans. Software Eng.* , vol. 14, no. 11, Nov. 1988, pp. 1668-1683.

[16] Richardson, J. P., Lu, H., and Mikkilineni, K., "Design and Evaluation of Parallel Pipelined Join Algorithms," *Proc. SIGMOD 87*, San Francisco, May 1987, pp.399-409.

[17] Schneider, D. A. and DeWitt, D. J., "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *Proc. SIGMOD 89*, Portland, Oregon, June 1989, pp. 110-121.

[18] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G., "Access Path Selection in a Relational Database Management System," *Proc. SIGMOD 79*, Boston, Massachusetts, Jun 1979, pp. 23-34.

[19] Su, S. Y. W., Database Computers, McGraw-Hill, 1988.

[20] Swami, A. and Gupta, A., "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques," *Proc. SIGMOD 89*, Portland, Oregon, June 1989, pp. 367-376.

[21] Teradata corporation, DBC/1012 Data Base Computer Concepts and Facilities, Teradata Document C02-0001-05, Los Angeles, CA, 1988.

[22] Valduriez, P., and Gardarin, G., "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *ACM Trans. Database Syst.*, vol. 9, no. 1, March 1984, pp. 133-161.

[23] Walton, C. B., Dale, A. G., and Jenevein, R. M., "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins," *Proc. VLDB 91*, Barcelona, Spain, Sept. 1991.

[24] Wolf, J. L., *et. al.*, "An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew," *Proc. 8th Data Engineering Conf.*, Japan, Apr. 1991, pp. 200-209.

[25] Zipf, G. K., Human Behavior and the Principle of Least Effort, Addison Wesley, 1949.