Research Report

Parallel Join Algorithms for Shared Memory Multiprocessor Database Systems

Alok N. Choudhary, Richard P. King and Nagui Halim

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, N.Y. 10598

NOTICE: This report will be distributed outside of IBM up to one year after the IBM publication date.
Parallel Join Algorithms for Shared Memory
Multiprocessor Database Systems

A.N. Choudhary, R.P. King, and N. Halim
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract
We present several parallel algorithms for join operations for tightly coupled multiprocessor systems with large shared memory. Our main assumption is that large enough shared memory is available to hold participating relations and corresponding index data structures. The proposed algorithms are categorized into two classes: "Two Index Join" algorithms, in which the two participating relations have ordered index structure, and, "One Index Join" algorithms, with only one relation having an ordered index structure.

We propose methods to partition the relations for parallel processing, with and without using the index structures, and then methods to join the partitions and merge results. We also present techniques to reduce number of searches in the target relation. The proposed algorithms are analyzed and their performance compared. The results show that it is possible to obtain several orders of magnitude improvement in join processing time over conventional disk-based systems if a large amount of main memory is available.
1. Introduction

With the advent of VLSI technology, main memory sizes of gigabytes or more will be feasible. Therefore, databases will become memory resident, avoiding the bottlenecks of disk accesses. In order to take advantage of large main memory, parallel processing to perform large database operations will also become necessary. Furthermore, it will be possible to make use of memory resident index structures to access the relations efficiently. Note that when relations reside on disks, clustering of data on the disks can be done only on one index, which normally is the primary index. Therefore, other index structures on the relations are useless for operations such as join because it is more efficient to access the pages from disks sequentially rather than randomly. However, in main memory, secondary index structures can be exploited to access relations for complex queries involving joins etc.

In this paper, we consider a tightly coupled multiprocessor system with a large main memory. We assume that the memory is large enough to store most of the database and, especially, all of the relations and the index structures participating in a complex query. We present various parallel algorithms for equijoin operations. The algorithms make use of index structures to perform a join operation if they are available on the join attribute. Also, we present several alternative algorithms when one of the relations participating in the join does not have an index structure on the join attribute. We analyze all of the algorithms and discuss their performance. The performance metric is the time to join two relations with various relation sizes. Other parameters include the output relation size (Join Selectivity), balance in the index structures, and duplicate factors in the relations.

There has been significant research in the area of relational database machines [1, 3, 9]. The main goal of the research has been to be able to answer complex queries efficiently. The design approach mainly included a set of parallel processors executing in the close proximity of the data storage units, mainly disks.

Another approach has been to propose and evaluate parallel algorithms on general purpose multiprocessor systems. Since join operation is the most expensive relational database operation, performance of join algorithms generally is a good indication of the overall performance of the multiprocessors systems for relational database operations. In [2], Bitton et al. present parallel algorithms for relational database operations. They consider a simple but realistic model of parallel processing with a three level memory hierarchy: local memory, shared global memory, and disk
storage. They assume limited amounts of shared and local memory, just big enough to store a few pages of each relation. Therefore, all of their algorithms are optimized for I/O, i.e., to minimize the number of disk accesses. The I/O access performance metric overshadows the impact of all other parameters on the performance. Nevertheless, in an I/O intensive environment, optimizing the disk accesses becomes the most important task.

In [8], Valduriez and Gardarin present parallel-join and semi-join algorithms for a more specialized multiprocessor system. Again, due to the presence of disks in the system, the algorithms are optimized for I/O. However, the architecture includes various components to efficiently access and filter the data from disks before the processors work on the parallel-join and semi-join operations.

Shapiro presented algorithms for equijoin on a uniprocessor system [6], but with large main memory. The analyses of the algorithms included the processing cost as well as I/O. The focus of the work was to estimate the minimum amount of main memory needed for the algorithms to work. Also, he discussed the trade-offs between various algorithms when sufficiently large main memory was available.

Valduriez introduced the concept of "Join Indices" in [7]. The join index is a simple data structure which improves performance by storing the surrogates of the two participating relations joined by the join attribute. Join indices are most useful if the output relation is small, and updates (inserts, deletes and modify operations) to the base relations are infrequent.

The effects of data skew and processing-time variations on parallel-join performance are explored by Lakshmi and Yu in [4]. In their model, the database is partitioned among a set of processors, each with their own disks and a small amount of local memory. These processors are interconnected with each other and with the requesting processors by a network with a limited total bandwidth (12 Mbyte/sec). They found, in this environment, that if there is a large imbalance in the workload, then there is very little speedup that can be achieved by adding more parallel processors.

It is possible to obtain several orders of magnitude improvement in the time to perform join just by having large main memories. If parallel processing is used along with large main memories, further speed ups can be obtained in the processing time. We show how the ordered and unordered index structures can be used to partition relations efficiently for parallel processing and how to merge results. Also, we present several techniques to reduce the search in the target relation for join.
The performance comparison and analysis provides us with various conditions under which some of the algorithms perform better than the others, and otherwise.

The rest of the paper is organized as follows. In Section 2, we present and analyze several parallel algorithms for equijoin. Section 3 contains the performance evaluation and comparison of the parallel algorithms. Finally, conclusions and future work are presented in section 4.

2. Parallel Algorithms for Joins

In this section we present parallel algorithms for equijoin operations. The description of the algorithms is followed by a simple analysis of their performance.

Structure of the Parallel Algorithms

Most of the parallel algorithms presented in this report have the following structure, though in some algorithms some steps may be combined or some steps may be implicit or unnecessary.

Preprocessing: The preprocessing step is needed if the relations participating in the join and their corresponding index structures need to be reorganized for partitioning the data for parallel processing. For example, if a relation does not have an index structure on the join attribute, then a hash index structure may be built on the relation before performing the join. However, it should be noted that preprocessing itself may be done in parallel, as we shall observe in the description of the algorithms.

Partitioning: In this step, the input relations participating in the join operation are partitioned onto the available processors. The method used to partition the data also depends on the sizes of the input relation and their corresponding index data structures.

Parallel Join: Following the data partitioning, each processor performs the join on its partition of the data. The method used for performing the join again depends on the input data structures, input relation sizes and desired format of the resultant relation.

Merging Results: Once the participating processors complete the join operation on their partitions, the output data needs to be merged in order to produce the desired joined relation. The merging part may be a part of the parallel join step, or it can be performed separately, if the input data
partitioning is such that data granules assigned to various processors can be independently processed.

Postprocessing: This step is normally needed if the output produced by the algorithm is not in the desired format. For example, it may be efficient for a parallel algorithm to produce a joined relation that is not sorted on the join attribute, then sort the output.

**Terminology**

The two relations to be joined are denoted by R and S. The join attribute is denoted by A and the corresponding columns in R and S are denoted as RA and SA respectively. The join selectivity is denoted by JS and semi-join selectivities for R and S are denoted by JSr and JSs. The cardinality of the two relations (number of tuples in each relation) is denoted by Nr and Ns respectively. The logarithm of any variable is taken on base 2 unless specified otherwise. "Two index join" means that both relations R and S have an order preserving index (such as T trees) on the join attribute (A). "One index join" means that only one of the relations has an order preserving index on the join attribute. The following is a list of symbols used to describe various parameters.

\[ N_r \equiv \text{Cardinality of Relation R.} \]
\[ N_s \equiv \text{Cardinality of Relation S.} \]
\[ N_i \equiv \text{Number of Tuples in Partition i.} \]
\[ N_{pop} \equiv \text{T-Tree node population.} \]
\[ N_o \equiv \text{Cardinality of the Output Relation} \]
\[ JS \equiv \text{Join Selectivity} = \frac{N_o}{N_r \times N_s} \]
\[ JSr \equiv \text{Semi-Join Selectivity of R} = \frac{N_o}{N_r} \]
\[ JSs \equiv \text{Semi-Join Selectivity of S} = \frac{N_o}{N_s} \]
\[ A_{dup} \equiv \text{Average number of duplicates per attribute value} \]
\[ t_w \equiv \text{time to walk through T-tree per tuple} \]
\[ t_s \equiv \text{time to search in the target relation} \]
\[ t_a \equiv \text{time to access a tuple from memory} \]
\[ t_c \equiv \text{time to combine two tuples for join} \]
\[ t_{st} \equiv \text{time to store a joined tuple} \]
\[ t_h \equiv \text{time to compute hash value for a key} \]
$h_c \equiv$ fraction of collisions in the bit vector
$t_{sa} \equiv$ time to scan relation $S$ sequentially
$t_{hr} \equiv$ time to search a hash table for a tuple
$t_{rs} \equiv$ time to search for a key in index structure for $R$
$t_{ts} \equiv$ time to search for a key in index structure for $S$

Note that values of some of the above parameters are different for different algorithms. For example, the search time $t_{rs}$ depends on the nature of the index structure. We will define the algorithms dependent values with the analysis of the algorithms.

2.1 Two Index Join Algorithm (I2SR)

In this algorithm, relations $R$ and $S$ are joined using the indices on these relations, with partitioning of the relations for parallel implementation. One of the two relations, say $R$, is partitioned and scanned, and the other relation, say $S$, is looked up for matches.

Let $P$ be the number of processors. To simplify the discussion assume $P = 2^r$. The index structure is a $T$-tree (a variation of AVL trees with bigger internal node) [5]. Each internal node contains an average of $N_{_{min}}$ elements, with fixed limit to the maximum and minimum number of elements in an internal node. The algorithms are as follows.

Data Partitioning: A designated processor partitions the data using the index structure ($I_r$) on $R$ to $P$ processors as follows. $P_1$ collects pointers to the subtrees of $I_r$ at level $\log(P)$ and distributes the pointers to the processors participating in the join. Also, each processor is given a pointer to the index structure of relation $S$ ($I_s$). Each processor $i$ will perform the join on its partition of size $N_i$.

Figure 1 on page 6 shows the partitioning of the data onto processors. The example shows the $T$-tree index structure. Processor $i$ gets the partition $P_i$ to perform the join as illustrated with a triangle representing the subtree below the root node $P_i$.

Parallel Join: In this step, each processor works on its partition of the data using the subtree of the index assigned to it. It scans its subtree of $I_r$ in order and, for each unique value of join attribute of $A$, it performs a look up into $S$ using $I_s$. If the corresponding attribute is found, then a join is performed for all of the duplicates, if any. Note that the output subrelation produced by each
Figure 1. T-Tree index structure (Data Partitioning and Merging): The leaf nodes are the root nodes passed to processors $P_1$ to $P_M$ for join processing.

processor is in order and the order of the output between processors is preserved due to the above partitioning of the input data.

Parallel Merge: The parent nodes of all of the subtrees (i.e., internal nodes from level 0 to level $\log(p)$) were not considered in the parallel join step. Therefore, along with the parallel binary merge of the output subrelations, a join operation for the nodes left in the previous steps is performed. The merge is performed $\log(p)$ times and each time the number of processors and output relations participating in the merge is reduced by a factor of 2.

There are two possible variations of the above algorithm. In one case the smaller of the two relations is scanned and the larger relation is looked up for join, in the other case the smaller relation
is looked up and the larger relation is scanned. The description of both algorithms is the same, and the only difference is observed in the performance of the two algorithms. The two variations are abbreviated by I2SS and I2SR (Two Index with Scan S and Two Index with Scan R).

Analysis

Assume that relation R is partitioned among the processors to be scanned and relation S is searched for matches using its index (Ib). The analysis is exactly the same for I2SS except with parameters for R and S interchanged.

The data partitioning step involves accessing \( P \times \log(P) \) pointers which takes time \( T_{pw} = P \times \log(P) \times t_r \). The cost of parallel join is computed as follows. Each processor walks through its partition, for each unique entry searching in relation S, and, if the search is successful, then the cost of joining and moving the joined tuple back to the memory is added. That is, the time for parallel Join is

\[
T_{\pi} = \max_{1 \leq i \leq P} \left[ N_i \times \left( t_w + \frac{t_{ss}}{A_{dup}} + J_{Sr} \times \left( t_w \times \frac{(A_{dup} - 1)}{A_{dup}} + t_c + t_{si}\right) \right) \right]
\]

and the cost for merging the results of partial joins in parallel is given by

\[
T_m = \log(P) \times N_{pos} \times \left( t_w + \frac{t_{xx}}{A_{dup}} + J_{Sr} \times \left( t_w \times \frac{(A_{dup} - 1)}{A_{dup}} + t_c + t_{ss}\right) \right)
\]

and the cost of the entire algorithm is given by \( T(I2SR) = T_{pw} + T_{\pi} + T_m \).

One Index Join Algorithms

We have investigated and analyzed several parallel algorithms for this case. Let's assume, without loss of generality, that the relation R has an ordered index on the join attribute while relation S does not. Briefly, the alternative algorithms are to: 1) partition relation R and use a nested loop search on relation S; 2) partition R and use a nested loop search on S but restrict the search in S by pre-processing S; 3) build a simple hash table for S on the join attribute to reduce search time; 4) use pre-processing with the hash table for S to restrict the number of searches in S; or 5) partition relation S, look up relation R for join and then sort the output on the join index to obtain the ordered, joined relation.
The first algorithm, in which for each unique join attribute in R the entire relation S needs to be searched, was very expensive compared to the other algorithms; we do not include any further discussion on that algorithm. We use the following abbreviations for the other four algorithms: Bit Vector Nested Loop (BVNL), Simple hash on S (SH-S), Hash on S with bit vector (SH-BV), and Unordered join and sort (SS-Sort, i.e., scan S and sort).

2.2 Bit Vector Nested Loop (BVNL)

Preprocessing: The motivation for preprocessing is to reduce the number of searches in relation S when performing parallel join. The method is as follows. A bit vector (B) is used, and for each attribute value in relation S a bit is set to 1. A hashing function H is used to compute the bit address for an attribute. For an attribute value A, bit H(A) of B is set to 1. Note that collisions may occur because two different attribute values may result in setting the same bit location. The collision factor depends on the hashing function and the length of the bit vector. Due to the possibility that \( H(x) = H(y) \), \( B(H(x)) = 1 \) does not necessarily imply presence of x. However, \( B(H(x)) = 0 \) definitely implies the absence of x.

The preprocessing step can be performed in parallel. A designated processor partitions the relation S equally among the available P processors. Each processor builds a bit vector from its partition of the data. Then, using a binary merge, the bit vectors are combined to obtain a bit vector for the entire relation S. This bit vector is then broadcast to all of the processors. Note that the probability of collision depends on the length of the bit vector, the relation size and the hashing function. The probability of collision can be reduced arbitrarily by increasing the bit vector length, and by using a combination of bit vectors instead of just one.

Data Partitioning: This is the same as for algorithm 12SR.

Parallel Join: In this step, each processor works on its partition of the data using the subtree of the index assigned to it. First, it scans the subtree of relation R in order. For each unique value A: 1) it checks bit \( B(H(A)) \); 2) if the bit is set, then relation S is scanned sequentially; and 3) if one or more matching tuples exist, then join is performed. If the bit is off, then no corresponding tuples exist. Note that sequential scan is performed for join and the function of the bit vector is only to reduce the number of scans of S.
Parallel Merge: This step is the same as in the two-index algorithms except that the searching method for S has changed due to the absence of an index on S. Note that the output produced by this algorithm is already sorted on the join attribute because relation R is scanned in order.

Analysis

In this algorithm, in addition to the cost of parallel join and merge we have to include the cost of preprocessing. That cost includes accessing the input relation S, computing the hash value for each tuple and updating the bit vector. We assume that there is enough local memory with each processor so that no accesses to the shared memory are needed to update the bit vector. Thus, the cost of preprocessing can be computed as

$$T_{pp} = N_S \times \frac{(t_w + t_a + t_h)}{P}$$

The cost of the parallel join step consists of the same components except that cost of the components is different from the previous algorithms. Also, for each bit vector hit, relation S has to be scanned completely to search for corresponding join tuples. The cost of parallel join and merge is given by

$$T_{pj} = \max [N_l \times (t_w + \frac{t_h + h_r \times t_{sec}}{A_{dup}}) + JSr \times (t_c + t_{stl})]$$

and

$$T_m = \log(P) \times N_{pop} \times (t_w + \frac{t_h + h_r \times t_{sec}}{A_{dup}}) + JSr \times (t_c + t_{stl})$$

and the cost of the entire algorithm is

$$T(BVNL) = T_{pp} + T_{pjr} + T_{pj} + T_m.$$

2.3 Simple Hash on S (SH-S)

In the preprocessing step of this algorithm, a hash table is built on the relation S. The hash table is then used for subsequent searches in the relation S. We will only describe the preprocessing step because the other steps are the same as the previous algorithm's except for the fact that in this algorithm the hash-table look-up is used for possible joins.
Preprocessing: In this step, all processors perform in a tightly coupled fashion since all processors have to work on the same hash table, and, therefore, there is a need for synchronization. We assume that it is possible for the processors to lock hash table locations individually and conflicts between processors occur only if two processors try to access the same location in the hash table. Such a situation will arise only if two processors try to insert entries into the hash table at the same time with join attributes such that their hash values are the same. We assume that the probability of such cases is very low and does not have a significant effect on the performance. This is a reasonable assumption since it is possible to use good hash functions and enough memory to avoid conflicts.

The hash table is built in parallel by P processors, each of which works on its partition of the input relation and updates the common hash table.

Analysis

The analysis of this algorithm is very similar to that of the previous algorithm except for the following differences. In this algorithm the search for a tuple in S is performed using the hash table instead of performing a sequential search on S. However, the preprocessing step involves moving the tuples around to build a hash table explicitly rather than just updating a bit vector. The preprocessing step cost is given by

\[
T_{pp} = P \times \log(P) \times t_a + N_i \times \left( \frac{t_w + 3.5 \times t_a + t_h}{P} \right)
\]

We can observe that the preprocessing cost involves 2 accesses for moving the tuples from the relation onto the hash table and that a collision factor adds 1.5 to the average number of accesses.

The parallel join and merge are different from the previous algorithm in number of accesses and access time. The hash table is accessed for each entry in R, but the access time is reduced from a sequential search to a hash-table look-up. The following are the costs of the parallel join and the merge.

\[
T_{pl} = \max_{1 \leq i \leq P} \left( N_j \times (t_w + \frac{t_h + t_{hs}}{A_{dup}}) + JSr \times \left( \frac{t_a \times (A_{dup} - 1)}{A_{dup}} \right) + t_c + t_i \right)
\]

\[
T_m = \log(P) \times N_{pop} \times (t_w + \frac{t_h + t_{hs}}{A_{dup}}) + JSr \times \left( \frac{t_a \times (A_{dup} - 1)}{A_{dup}} \right) + t_c + t_i
\]
and the cost of the entire algorithm is given by $T(SH - S) = T_{rp} + T_{psr} + T_{rj} + T_m$

### 2.4 Hash on S with bit vector (SH-BV)

This algorithm incorporates the reduced search time of the previous algorithm by using a hash table, and the number of searches is reduced by using a bit vector. The additional cost of constructing a bit vector in the preprocessing is small because the main cost of constructing a bit vector is computing the hash value for each key, which is done anyway while building the hash table.

**Analysis**

The performance of this algorithm is closely related to the performance of the previous one. The preprocessing is more costly, but the parallel join step is cheaper, due to the reduction in the number of searches in relation $S$ by use of the bit vector. The various costs are as follows.

\[
T_{pp} = N_s \times \frac{(t_w + 4.5 \times t_a + t_h)}{p}
\]

\[
T_{pj} = \max_{1 \leq l \leq p} \left[ N_l \times (t_w + \frac{t_h + h_r \times t_h}{A_{dup}} + JSr \times \frac{t_a \times (A_{dup} - 1)}{A_{dup}} + t_c + t_{cd}) \right]
\]

\[
T_m = \log(P) \times N_{pop} \times (t_w + \frac{t_h + h_r \times t_h}{A_{dup}} + JSr \times \frac{t_a \times (A_{dup} - 1)}{A_{dup}} + t_c + t_{gi})
\]

and the cost of the entire algorithm is given by $T(SH - S) = T_{rp} + T_{psr} + T_{rj} + T_m$

### 2.5 Unordered Join and Sort (SS-Sort)

So far we have considered those cases in which the relation with an index structure on the join attribute is partitioned and scanned while the relation without the index structure is searched for the corresponding tuples for join operation. Such schemes result in sorted output because using the ordered index it is possible to scan a relation in order. However, it is possible to partition the relation which does not have an ordered index on the join attribute and search the relation which has an ordered index on the join attribute. The advantages of such a scheme are that using the index structure for search reduces the search time and preprocessing steps required by the previous two algorithms are eliminated. The disadvantage of this scheme is that the output is not sorted on the join attribute and, therefore, a post processing step is needed to sort the output. If the output re-
lation is very large, then sorting can be expensive. Furthermore, since the relation is not scanned in order, the advantages of duplicate clustering are lost, and the time complexity of the algorithm becomes proportional to the number of tuples in the input relation, as opposed to the number of unique tuples.

**Data Partitioning:** A designated processor partitions the data using the primary index structure on S to P processors as follows. It collects P pointers to the subtrees of Ir at level \( \log(P) \) and distributes the pointers to the processors participating in the join. Also, each processor is given a pointer to the index structure of relation R (Ir). Each processor performs the join on its partition. Note that number of elements in each partition will not normally divide the input relation S equally to all of the processors, due to imbalance in the tree structure.

**Parallel Join:** In this step, each processor works on its partition of the data. The partitions of S are scanned, and, for each tuple in relation S, relation R is searched using the index on the join attribute. Note that since the relation S is not scanned in order, the duplicates are not clustered together and, therefore, the search on R needs to be performed for each input tuple in S.

**Parallel Merge:** This step is similar to the merge steps we discussed in the previous algorithms and therefore, we will not discuss it here. The merge is performed \( \log(p) \) times and each time the number of processors and output relations participating in the merge is reduced by a factor of 2.

**Sorting the Output Relation:** In this step, the output relation is divided into equal parts to the available processors. The processors apply a sequential algorithm to sort their partition. Then merging is performed in a binary tree fashion which produces a sorted relation in \( \log(P) \) steps.

**Analysis**

This method enabled us to use the index of relation R to search for tuple for joining with candidate tuples in relation S. Therefore, we avoided any kind of preprocessing and reorganization of the input data in order to obtain an efficient algorithm. The performance of the first three steps is similar to algorithm 12SS because we use the index structure to search the target relation (R). However, since the relation S, which does not have an index on the join attribute, is scanned, the output is not sorted on the join attribute. Therefore, there is a need for postprocessing step to sort the output relation on the join attribute. It is easy to note that this algorithm is attractive when it
is known that the output relation is very small compared to the input relations. The cost of the first three steps of the algorithm is given by the following formulae.

The first step of data partitioning involves accessing $P \times \log(P)$ pointers which results in $P \times \log(P) \times \text{ta}$ time (cost of partitioning $T_{par}$).

$$T_{pj} = \max_{1 \leq i \leq P} \left[ N_i \times (t_w + t_{sr} + JSs \times (t_w \times \frac{(A_{dup} - 1)}{A_{dup}} + t_c + t_{sr})) \right]$$

and cost for merging the results of partial joins in parallel is given by

$$T_m = \log(P) \times N_{pop} \times (t_w + t_{sr} + JSs \times (t_w \times \frac{(A_{dup} - 1)}{A_{dup}} + t_c + t_{sr}))$$

The postprocessing step requires a parallel sort of the output. Each processor performs a sort on its own partition using a sequential algorithm, and then the sorted tuples are merged in a binary tree fashion. The step incurs the overhead

$$T_{sort} = \frac{(N_o)}{P} \times \log\left(\frac{(N_o)}{P}\right) \times 2 \times t_a + 2 \times N_o \times t_a$$

and the cost of the entire algorithm is given by $T(SS - Sort) = T_{par} + T_{pj} + T_m + T_{sort}$

3. Performance

In this section, we discuss and compare the performance of the algorithms. The performance metric is the time taken to join two relations as a function of the relation sizes. Other parameters are output size (which determines the join selectivity), duplicate factor, and balance in the index data structures.

Some of the default values for the above parameters are as follows. The output size is 1000 tuples, the index structures are assumed to be balanced, and the tuples are assumed to be unique. The performance comparison of various algorithms is performed by varying one parameter at a time so that the effect of that parameter can be understood clearly. The other parameter values are summarized below.

$$N_{pop} = 10$$

$$t_w = 10 \text{ instructions/tuple}$$
\( t_a = 0.5 \) microseconds

\( t_e = (1.0 \text{ microseconds} + 15 \text{ instructions})/\text{tuple} \)

\( t_{st} = 0.5 \) microseconds

\( t_b = 150 \) instructions

\( t_{bo} = 2 \times t_a \)

\( t_r = \log(\frac{N_r}{N_{pp}}) \times 20 \) instructions

\( t_{rr} = \log(\frac{N_r}{N_{pp}}) \times 20 \) instructions

\( t_{res} = N_r \times 10 \) instructions

\( h_i = 1.5 \), Means on an average for each set bit in the
bit vector, one third will be set due to collision.

Processor Speed = 15 MIPS

---

Figure 2. Performance for Join: Parameters: Relation R size = 500,000; No duplicates,
Balanced Index Trees, P = 16, Output Size = 1000 tuples.
Figure 3. Parallel Join Performance: Parameters: Relation R size = 500,000; No duplicates, Balanced Index Trees, P = 16, Output Size = 1000 tuples.

Figure 2 shows the time for join as a function of the size of input relation S. Other parameters are shown with the figure. The Y-axis scale is logarithmic. Note that algorithm BVNL (Bit-Vector Nested-Loop) performs poorly compared to all other algorithms. For each hit in the bit vector, the entire relation S needs to be searched sequentially for the possible join tuples. Since this algorithm performs so poorly, we will not include it in our discussions any further.

Figure 3 plots the same data without BVNL. The two-index algorithm is represented by the lines marked I2SR (scan R, look-up in S) and I2SS (scan S, look-up in R). The scan takes time proportional to the size of the relation, while the look-up takes time which is logarithmic in the size. Therefore, it is best to scan the smaller relation, using the index in the larger to support the look-ups.

With the other algorithms, it is assumed that there is an index on R for the join attribute, but not on S. Still, the performance of SS-Sort is nearly identical to that of I2SS. The scan of S takes the
same time, and the sort of the (small) output relation is negligible. This makes SS-Sort the best choice as long as S, the relation without the index, is not too much larger than R, the relation with the index.

In this case, with R fixed at 500,000 tuples, the crossover occurs when S has 600,000 tuples. SH-S and SH-BV, like SS-Sort, scan S, but only to build a hash table, which takes very little time. They then scan R, which is smaller, and do fast look-ups in the hash table for S. This makes SH-BV the best choice when S is much larger than R.

Figure 4 shows join times for the algorithms when relation sizes are increased by an order of magnitude compared to those in the previous figure. The main difference is that SH-BV and SH-S surpass SS-Sort sooner. With R’s size being 5,000,000 tuples, the crossover point has moved to 4,000,000 tuples. This is due to R’s larger size, which has increased the look-up times for SS-Sort.
Figure 5. Join with Large Output: Parameters: Relation R size = 500,000; No duplicates, Balanced Index Trees, P = 16, Output Size = 100K tuples.

Next, we made the output relation size of the same order of magnitude as that of the inputs. (See Figure 5 on page 17.) This makes the sorting time of SS-Sort much larger, with SH-BV the best except when S is smaller than R. Note also that the difference between SH-BV and SH-S is smaller. The large output size implies a large number of hits in SH-BV's bit vector, which reduces the value of building the bit vector in the first place.

Figure 6 on page 18 shows the performance of the algorithms when the average number of duplicates is 5, i.e., for each unique value of the join attribute in the input relation, there are, on the average, 4 more tuples with the same join attribute. SS-Sort is worse than SH-BV and SH-S unless S is very much smaller than R. Since SH-BV and SH-S scan R using the index on the join attribute, the duplicates in R are one after the other, so matches need only be searched for once per set of duplicates, and any matching duplicates in S are grouped together by the hash table. SS-Sort scans S, which has no index on the join attribute, so the duplicates in S are not grouped together. This makes it impossible for them to share the cost of searches in R.
It is interesting to notice that I2SR and I2SS receive the greatest benefits from duplicates. The largest factor in their join times is the T-tree search, so they save a great deal when duplicates allow them to avoid searching.

So far, we have assumed that when input relations are partitioned using the index structure, the processors get equal shares of the input relation. However, the index tree structure may not be completely balanced. We assume that each node can hold up to 10 entries and no fewer than 9 except the leaf nodes, where minimum of 1 is allowed. This allows most internal nodes to be fully packed. Also, from any internal node, the heights of the two subtrees below it may differ by at most one. In the worst case, the largest partition may contain 1.8 times as many tuples as the smallest.

Figure 7 on page 19 shows the performance of the algorithms when the trees are not balanced. The performance of all of the algorithms is degraded, since the largest partition scanned is signif-
Figure 7. Join with Unbalanced Partitions: Parameters: Relation R size = 500,000; No Duplicates; P = 16, Output Size = 1000 tuples.

icantly larger than before. There is, however, a shift in favor of SH-BV and SH-S, which spend proportionally less of their time performing the scan, and so suffer proportionally less when the scan is lengthened.

4. Conclusions

In this paper, we presented several parallel join algorithms for tightly-coupled multiprocessor systems with large main memory. We discussed methods to partition the input relations, perform join in parallel, and merge results in parallel. We described the use of index structures to partition relations and merge the results. Furthermore, we described methods to reduce the number of searches in the target relations if ordered index structures are not present on the target relation.

The results indicate that it is possible to perform large joins very fast, with several orders of magnitude improvement over conventional, disk-based systems. We analyzed and compared performance of the proposed algorithms. In general, if both relations participating in the join have ordered
index structures, then using the index structures to perform the join does the best. Also, we conclude that partitioning and scanning the smaller relation, and using the larger relation for target searches, performs better than scanning the larger relation. If only one relation has an index structure, then preprocessing the data to reduce the number of target searches is a good alternative. However, if it is known that the output relation size is much smaller than the input relation sizes, then partitioning the relation without an index structure and searching the one with the index performs fairly well.

Several extensions to this work are possible. One can generalize these algorithms or propose new algorithms for non-equi joins. We think that algorithms using order preserving index structures will do better than the others. Algorithms for other relational database operations, such as selection, projection, and union, need to be looked at in such an environment. Finally, a general approach to query optimization in multiprocessor systems with large main memory is another avenue to be explored.
References


