Optimization of Multi-Way Join Queries for Parallel Execution

Hongjun Lu
National University of Singapore
Singapore 0511

Ming-Chien Shan
Hewlett Packard Laboratory
Palo Alto, CA 94303

Kian-Lee Tan
National University of Singapore
Singapore 0511

ABSTRACT

Most of the existing relational database query optimizers generate multi-way join plans only from those linear ones to reduce the optimization overhead. For multiprocessor computer systems, this strategy seems inadequate since it may reduce the search space too much to generate near-optimal plans. In this paper we present a framework for optimization of multi-way join queries in multiprocessor computer systems. The optimization process not only determines the order and method in which each join should be performed, but also determines the number of joins should be executed in parallel, and the number of processors should be allocated to each join. The preliminary performance study shows that the optimizer usually generate optimal or near-optimal plans when the number of joins is relatively small. Even when the number of joins increases, the algorithm still gives reasonably good performance. Furthermore, the optimization overhead is much lesser compared to exhaustive search.

1. Introduction

With the advent of VLSI technology, the trend of computer architectures is moving towards multiprocessor systems. This trend has great influence to all fields in computer science. For database management systems, as an example, a large amount of work has been done to explore parallel processing of database operations. Special database machines have been designed to obtain increased system performance (response time and throughput) through both inter-query and intra-query parallelism [Bora90, Bult89, DeWi90, Su88, Tera83]. However, most of the existing relational database query optimizers only consider plans in which the execution order is modeled by a linear processing tree. An M-way join query

$$R_1 \bowtie R_2 \bowtie \cdots R_M \bowtie R_{M+1}$$

is visualized as a sequence of 2-way joins of the form

$$(((\cdots (R_1 \bowtie R_2) \bowtie R_3) \cdots)) \bowtie R_{M+1}).$$

This strategy seems adequate in uniprocessor systems [Seli79, Lohm85]. In a multiprocessor environment, however, the number of feasible join plans increases dramatically with new dimensions introduced by parallelism and parallel processing tends to give better overall system performance. The optimal, and even the suboptimal solution may be excluded from the search space by restricting to a linear execution sequence only.

This problem has been recently addressed by researchers from different directions [Kris86, Ston88, Swam88, Swam89, Deen90, Ioan90, Ono90, Schn90]. Krishnamurthy, Boral and Zaniolo proposed heuristics to optimize non-recursive multi-way query with enlarged search space [Kris86]. Swami, Gupta and Ioannidis studied the benefit of using other techniques such as simulated annealing in query optimization to tackle the problem of large search space [Swam88, Swam89, Ioan90]. Ono and Lohman show that even cartesian product should be sometimes considered to generate optimal plans [Ono90]. For multiprocessor systems, Stonebraker et. al. proposed a two step approach to optimize query plan, in which a collection of good sequential plans is first obtained based on the buffer space and the parallelization of this collection of plans is then explored [Ston88]. Schneider and DeWitt studied the behavior of query plans with different type of structures, left-deep, right-deep and bushy, in processing multi-way join queries in shared-nothing architecture [Schn90].

In this paper, we report our study on optimizing non-recursive multi-way join queries in multiprocessor systems. The major difference between our work and the previous ones is that parallelism is explored at two levels : *intra-join parallelism* where several processors may be assigned to one join operation and *inter-join parallelism* where several joins may be performed concurrently. The proposed query optimization algorithm, therefore, not only determines the order in which each join should be executed and the method should be used, but also determines the number of joins should be executed in parallel and the number of processors should be allocated to each join operation.

In the next section, we describe the multiprocessor system architecture and briefly review some results from the first phase of our study which forms the base of our work reported here. The framework of parallelizing multi-way join queries in the multiprocessor system is proposed in section 3. Section 4 describes some results of

a performance study on our optimization algorithms. The last section, section 5, is a summary and discussion about future work.

2. The Basics of Our Work

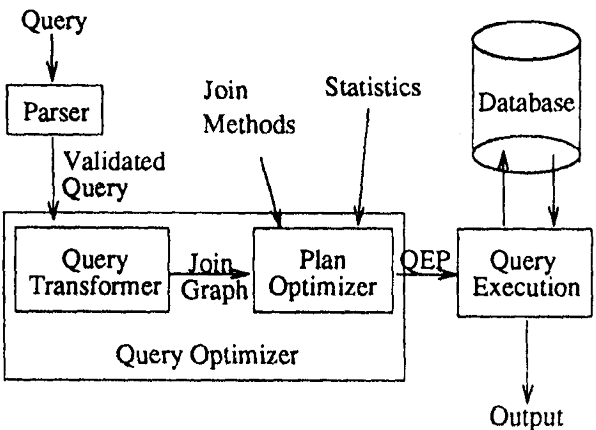


Figure 2.1. Functional components in query processing.

In general, the database query processing and optimization process takes queries in a declarative language as input and generates query execution plans (QEPs). By executing those plans, the results of queries are obtained and delivered to the user. Figure 2.1 depicts the functional components involved in this process. A *parser* is used to validate the input queries. The validated query is first transformed into some semantically equivalent internal representation form such as a *join graph.* During this transformation, some heuristic rules can be applied, such as *push selection down as much as possible* and *perform projections as soon as possible.* The join graph, together with statistics about the participating relations and available join methods, are sent to the core component, *the plan optimizer*, to generate the final query execution plan. The major function of this plan optimizer is to select an optimal or near-optimal query execution plan among all feasible ones based on some optimization objectives.

In most existing database management systems, the primitive join operation is the two-way join. Multi-way join queries are usually treated as a sequence of two-way joins. Two essential tasks of the plan optimizer are to select

1) the order in which the joins are performed, and
2) an appropriate join method for each join operation

to achieve some predefined optimization objectives. In a multiprocessor system, however, there are more dimensions in the search space for optimization if both inter- and intra-join parallelism are to be explored. In addition to the above two tasks, the plan optimizer must also determine

3) the number of joins to be executed concurrently and the relations participating in each of these joins, and
4) the number of processors allocated to each of the concurrent join operations.

Hence, optimization of a query becomes more expensive and complicated in multiprocessor environment.

2.1. Multiprocessor computer system with shared memory

The multiprocessor systems in our study are general purpose systems without any special-purpose hardware for database operations. The number of processors of such system is relatively small compared to some database machines that may consist of a few hundred or even thousands of processors [Tera83]. Each processor that shares a common memory (shared memory) with other processors may also have some buffers dedicated to itself (local memory) for input/output. Different from most previous work where a fixed amount of memory is assumed to be available for a database operation such as join, we assume that the amount of memory available for an operation varies according to the number of processors assigned to the operation. This assumption may cause some difficulty in performance analysis as the effects of the number of processors cannot be isolated. However, this is closer to real situation. In a general purpose computer system, when a processor is assigned some tasks to execute, it is usually allocated a certain amount of memory space. We also assume that if a certain number of processors is allocated to process a query, the control of these processors and related memory will be transferred to the database management system. It is up to the database management system to schedule the processors and to efficiently use the available memory space. Furthermore, though it is expected that main memory sizes of a gigabyte or more will be feasible and perhaps even fairly common within the next decade, we still cannot assume that a whole relation can be read from the mass storage to either the processor's local memory or the shared memory before processing. That is, in general, both the total memory of the processors and the size of the shared memory are not large enough to contain a whole relation.

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. The processors, disks and memory are linked by an interconnection network. We assume that the interconnection network has sufficient bandwidth for the tasks at hand. That is, the contention for the interconnection network is not considered in our analysis. However, we do consider the contention of the shared memory, which is reflected in the cost model of parallel join operations.

2.2. Parallel join methods and their costs

As mentioned above, one of the major task of a query optimizer is to determine the method for each join to be performed since there are usually a number of ways

to perform a particular join with different costs. In the multiprocessor environment, the selection of join methods becomes more complex. First, the number of join methods increases. For uniprocessor system, the sort-merge join, nested loops join and hash-based join are three major join methods. In a multiprocessor system, each of these methods has a number of variations with different performance. Second, there are more parameters that affect the cost of a join in multiprocessor systems than in uniprocessor systems, such as number of processors participating in the join and the architecture of the system (shared nothing, shared everything, etc). Recently, quite a number of research work have been reported on parallel join algorithms [DeWi85, Kits90, Schn89, Vald84, Wolf90]. Performance of different parallel join methods are analyzed. In general, the cost of a parallel join method is a function of the two relations to be joined and the number of processors participating in the join.

In [Lu90], we reported our work on multiprocessor join algorithms. We studied four hash-based multiprocessor join methods: a parallel version of hybrid-hash join and its modification, hash-based nested loops join and simple hash join. The costs of these hash-based join methods are studied in terms of the total processing time and the elapsed time. Since the major purpose of parallel processing is to speed up the computation, the *elapsed time* is taken as the objective of optimization. Hereafter the cost of a join refers to the elapsed time unless otherwise specified. To estimate the elapsed time of a join, the join process is decomposed into sequentially executed *phases*. Most hash-based joins can be decomposed into two such phases: a *partition phase* and a *join phase*. The join phase, limited by the available memory size, is often further divided into iterative *batches*. Each of them joins only a portion of the two relations. These batches are also sequentially executed. In most computer systems, CPU processing, i.e. computation on the data in memory, proceeds concurrently with disk I/O operation, i.e. to find the required data on disk and to bring it into memory. Our calculation of the elapsed time considered this overlap among CPU processing and disk I/O operation. It works as follows: we first compute the required disk I/O time per disk drive and CPU time per processor for each phase i in executing an algorithm, T^i_{IO} and T^i_{CPU}. For a system with d disk drives and p processors, the total processing time for phase i is then

$$T_i = p \times T^i_{CPU} + d \times T^i_{IO} \qquad (2.1)$$

The elapsed time E_i for phase i, which is generally less than $T^i_{CPU} + T^i_{IO}$ due to overlap, will be [1]

$$E_i = \max(T^i_{CPU}, T^i_{IO}) \qquad (2.2)$$

Since phases of an algorithm are executed serially, the elapsed time of an algorithm with n phases is

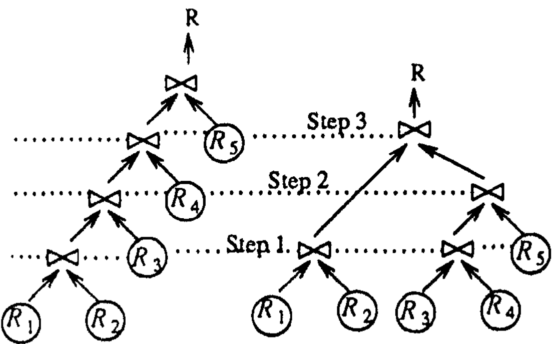
$$E = \sum_{i=1}^{n} E_i \qquad (2.3)$$

For the detailed analysis of the four join methods and formulas of computing T^i_{CPU} and T^i_{IO} for these methods, please refer to [Lu90]. In our implementation of the query optimization algorithms discussed in this paper, we used these formulas. However, the algorithms proposed is independent of the join methods and cost formulas. We will assume that ***the number of join methods provided by the system and the costs associated with them are available*** in the later discussion.

[1] Here, for a CPU-bound phase, the time to read in the initial pages before the processing begins, and the time to write out the final pages of the resulting tuples are ignored. While for an I/O-bound phase, the time to initiate and terminate the processing are ignored.

3. Optimization of Multi-way Join

Intra-join algorithm can be achieved by assigning more than one processor to a join operation as in all proposed parallel join algorithms [DeWi85, Kits90, Schn89, Vald84, Lu90, Wolf90].



(a) Linear QEP (b) Synchronized bushy QEP

Figure 3.1. Query execution plans (QEPs).

Inter-join parallelism among multi-way join queries can be realized by generating query execution plans with bushy structure. The difference between such bushy structured QEPs and the linear structured QEPs is shown in Figure 3.1. In a linear QEP (Figure 3.1a), joins in a multi-way join query are performed one by one. The result relation from the first join of two relations, say R_1 and R_2, is joined with the third relation, R_3, the result of which is then joined with the fourth relation, R_4, and so on. In a bushy structured QEP, a number of pairs of relations may be joined in parallel. In Figure 3.1b, two pairs of relations, (R_1, R_2) and (R_3, R_4), are joined in parallel. The result of $R_3 \bowtie R_4$ is then joined with R_5, the result of which is again joined with the result of $R_1 \bowtie R_2$.

When the bushy structured QEPs are included in the search space of a query optimizer, the number of feasible QEPs increases dramatically. To limit the increase of QEPs in the search space of our multiprocessor query optimizer, we divide QEPs into two groups, *synchronized* and *asynchronized*. By a synchronized QEP, we mean that the whole multi-way join process is

divided into synchronized steps. For each step, a number of joins are executed concurrently. The joins to be performed at the following step will not start to execute until *all* joins in the previous step have been completed. In this section, we are going to propose a greedy multi-way join optimization algorithm which explores inter-join parallelism by considering such *synchronized QEPs* during optimization. By limiting QEPs to synchronized ones, the cost estimation of a QEP is easier. However, there are two possible side effects: (1) the possible pipeline among steps is not taken into account. Instead, the costs of storing and retrieving the intermediate results are added to the plan cost, and (2) some processors that complete one join earlier than others have to wait and the CPU utilization will decrease. As a result, some better plans may be excluded from the search space. However, the second effect could be minimized by carefully allocating processors to the joins to be concurrently executed according to their workload. Furthermore, since linear QEPs are still in the search space, the new optimization algorithm should be at least as good as those that do not consider bushy QEPs.

3.1. Algorithm GP: a greedy multi-way join query optimization algorithm

Our algorithm, GP, the Greedy Parallel multi-way join optimization algorithm is listed in Figure 3.2. Algorithm GP is an iterative algorithm that generates one step in a synchronized QEP during each iteration. It is a greedy algorithm since it always tries to join as many pairs of relations as possible in parallel for the current step. At the beginning, all relations to be joined are included in the working set T. A set of relation pairs, R, is selected for the first step by calling function *Select_rel_pairs*. For subsequent steps i, the same procedure is applied to the reduced working set that consists of the intermediate relations from the last step, step $i-1$, and the relations that have not been joined so far. Graphically, this reduced working set is represented by a *reduced join graph* that is obtained by replacing the relations joined in step $i-1$ by their result relations and merging the edges accordingly. When the working set contains less than four relations, function *Two_way_seq* is called to determine the sequence of sequentially joining those relations.

3.2. Selecting pairs of relations

Function *Select_rel_pairs* in Algorithm *GP* select k pairs of relations from the working set to be joined in parallel for the current step. *Select_rel_pairs* determines concurrently executed relation pairs with given working set (or join graph). The algorithm, shown in Figure 3.3, also uses an iterative approach starting with $k = 1$. During each iteration, it computes the costs of QEPs which concurrently join k pairs of relation at the first step and find the minimum cost by calling function *Minimum_cost*. It terminates when either k is equal to the number pairs in the join graph or such k is found that the minimum cost of QEPs concurrently joining $k+1$ pairs first is greater than the minimum cost of QEPs having k joins evaluated concurrently first.

Algorithm GP

Input : A join graph $G = (T, E)$
where node set T is a set of relations
and edge set E represents the join conditions.
Output : S, the join sequence consisting of relation pairs

```
S ← ∅;
while Size(T) > 3 do {
    R ← Select_rel_pairs (G);
    S ← S ∪ R;
    G ← G  with each pair of relations in R replaced by
             their join results;
}
R ← Two_way_seq (G);
S ← S ∪ R;
```

Figure 3.2. Multi-way join optimization algorithm *GP*.

Algorithm Select_rel_pairs

Input : G, a join graph
Output : R, a set of relation pairs to be joined concurrently

```
begin
  k ← 0;
  repeat
    k ← k+1;
    C_k ← Minimum_cost (G, k, R_k);
    if ( R_k does not contain all relations in G )
    then
      C_{k+1} ← Minimum_cost (G, k+1, R_{k+1});
  until C_{k+1}>C_k or R_{k+1} contains all pairs in G
  if C_{k+1}>C_k
  then return R_k
  else return R_{k+1}
end;
```

Figure 3.3. Function *Select_rel_pairs*.

Function *Minimum_cost* is the core part of the algorithm. It takes the reduced join graph G, and the number of relation pairs to be joined concurrently first, k, as input and returns the minimum cost of those plans that joins k pairs first. At the same time, it determines those k pairs of relations and join methods for each pair of relations. The computation complexity of this function comes from (1) the large number of feasibie QEPs that join k pairs in parallel during the first step; and (2) a large number of combinations of join methods supported and possible processor allocation strategies for a chosen QEP. To simplify the cost evaluation of QEPs and hence to reduce the optimization overhead, we propose two heuristic cost functions that lead to two versions of Algorithm GP: GP_T, an optimization algorithm based on *total cost* and GP_P, an optimization algorithm based on *partial cost*. As the name implies, algorithm GP_T estimates the

total cost of a QEP, C_{plan}, which is the sum of the cost of each step i ($1 \le i \le m$), in the m-step QEP. On the other hand, Algorithm GP_P uses only the cost of the first step (may plus one more join as explained later) $Cost_1$ as the approximation of $Cost_{plan}$. We discuss the details of these two algorithms in the next two subsections.

3.2.1. Greedy Parallel multi-way join optimization based on total cost(GP_T)

In Algorithm GP_T, the total costs of a QEP, $Cost_{plan}$, is computed as the sum of cost of each step $Cost_i$ in the QEP. However, even with a small number of joins in the join graph, it seems still very expensive to search for the minimum cost from all possible combinations of different number of steps and different number of joins at each step. An important heuristic used to limit the search space in GP_T is to consider only those QEPs that execute k joins concurrently at the first step and execute remaining joins sequentially. For those QEPs, the plan cost is

$$Total_k = Par_join_cost(R_k) + Seq_join_cost(T - R_k \cup Join_result(R_k)) \quad (3.1)$$

where $Par_join_cost(R_k)$ returns the cost of joining relations in R_k in parallel and $Seq_join_cost(R)$ returns the cost of joining relations in R sequentially. Since there are a number of different ways to select k pairs of relations from all relations and also a number of different sequences to join the remaining relations sequentially, function $Minimum_cost$, with given k, enumerates the costs returned by two functions Par_join_cost and Seq_join_cost and returns the minimum among them, $MIN(Total_k)$, which is denoted as C_k in function $Select_rel_pairs$.

Figure 3.4. An example join graph.

We will delay the discussion about finding $MIN(Total_k)$ for the moment and use an example to explain how algorithm GP_T works. Consider the join graph shown in Figure 3.4. There are 7 relations R_1 to R_7 such that there are join predicates between R_i and R_{i+1} for $i = 1, \ldots\ldots, 6$. The optimization process using algorithm GP_T is illustrated in Figure 3.5. To make the presentation simpler, we will only focus on determining the number of pairs to be joined and will not identify the actual relations in the pairs. Algorithm GP_T, as an iterative algorithm, starts with the working set T containing all seven relations. In step one (Figure 3.5a), C_1 and C_2, the minimum cost of executing one join and two joins concurrently at the first step respectively, is first computed and compared. Assume that $C_1 > C_2$ so that C_3, i.e. the minimum cost of joining three pairs of relations in parallel during the first step, is computed. Suppose $C_3 > C_2$ and hence $Select_rel_pairs$ returns two pairs of relations that should be joined concurrently in step one. Now, the working set of step two consists of 5 relations — two intermediate results and three original relations (Figure 3.5b). For this working set, C_1 and C_2 are first computed and let $C_1 < C_2$. That means only one pair is joined in step two. Finally, in step three (Figure 3.5c), C_1 is compared with C_2 ($< C_1$). Since there are only four relations, no further computation is needed and two pairs of relations should be joined concurrently in this step. The working set now contains only two relations which is for the last step and the algorithm terminates. The plan generated thus consists of four steps: joining two pairs of pairs concurrently, followed by one pair, followed by two pairs and end with another join (Figure 3.5d).

3.2.2. Greedy parallel multi-way join optimization based on partial cost (GP_P)

In Algorithm GP_T, the minimum *total* cost of QEPs joining a set of relations is to be computed. Although the computation is simplified by limiting the search space to those QEPs that only execute joins in parallel at their first step, it is still quite expensive to compute such total cost at each iteration, especially when the number of joins increases. In order to further reduce the optimization overhead, the second version of Algorithm GP, GP_P only estimates the *partial cost* of a QEP, $Partial_k$, and uses it as the approximation of $Cost_{plan}$. $Partial_k$, the cost of a QEP which joins k pairs of relations concurrently first, is represented by the cost of executing these k pairs in parallel:

$$Partial_k = Par_join_cost(R_k) \quad (3.2)$$

Since two QEPs — one joins k pairs first, and another joins $k+1$ pairs first — as required in function $Select_rel_pairs$, has different number of joins, the comparison of these two QEPs are done by computing

$$C_k = MIN(Partial_k) + C_{one_join}$$

and

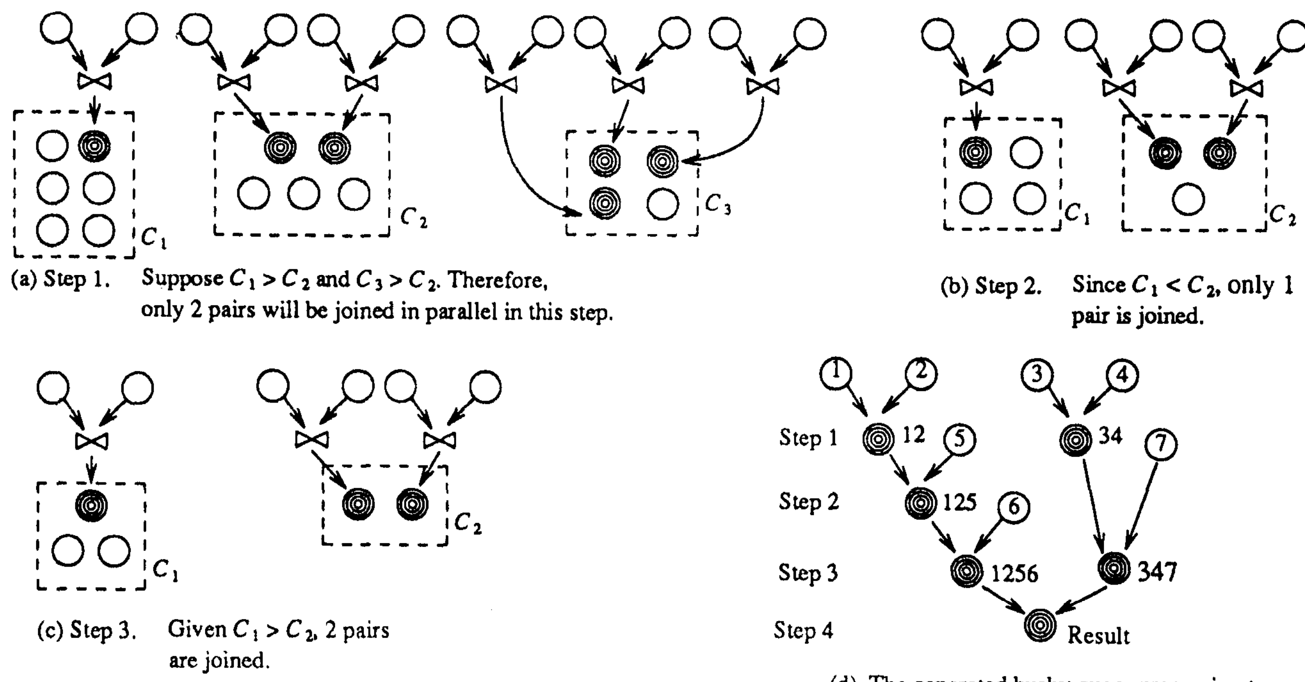
$$C_{k+1} = MIN(Partial_{k+1})$$

where C_{one_join} is the minimum cost of joining two relations from the intermediate results and the remaining relations in the original working set. The number of relation pairs to be joined concurrently, k is determined as follows:

$$k = \begin{cases} 1 & \text{if } C_1 < C_2 \\ N & \text{if } C_{k-1} > C_k \ \ \forall \ k, \ 1 < k \le N \\ k & \text{if } C_{k-1} > C_k \text{ and } C_k < C_{k+1} \end{cases} \quad (3.3)$$

where N is the maximum number of pairs.

Use the same example for Algorithm GP_T, Algorithm GP_P works as illustrated in Figure 3.6 : In step one (Figure 3.6a), the working set contains seven relations and C_1 and C_2 are compared. Assume $C_1 > C_2$ and C_2 and C_3 are then computed with the result $C_2 > C_3$. Since the maximum number of parallel joins is three, so step one should join three pairs in parallel. Next, the working set of step two consists of four relations (Figure



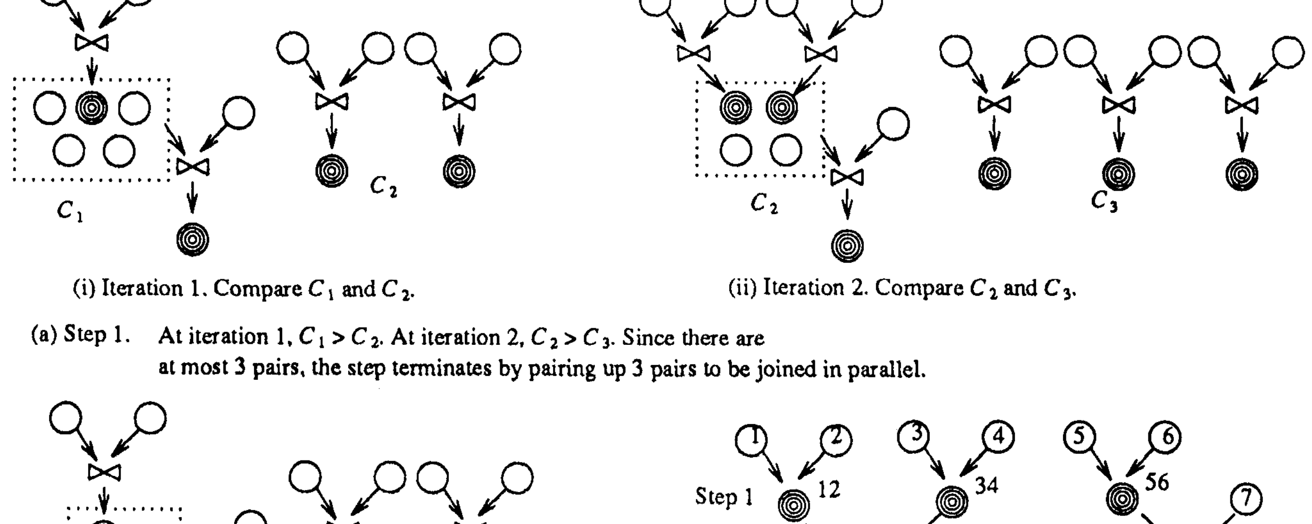
(a) Step 1. Suppose $C_1 > C_2$ and $C_3 > C_2$. Therefore, only 2 pairs will be joined in parallel in this step.

(b) Step 2. Since $C_1 < C_2$, only 1 pair is joined.

(c) Step 3. Given $C_1 > C_2$, 2 pairs are joined.

(d) The generated bushy query processing tree.

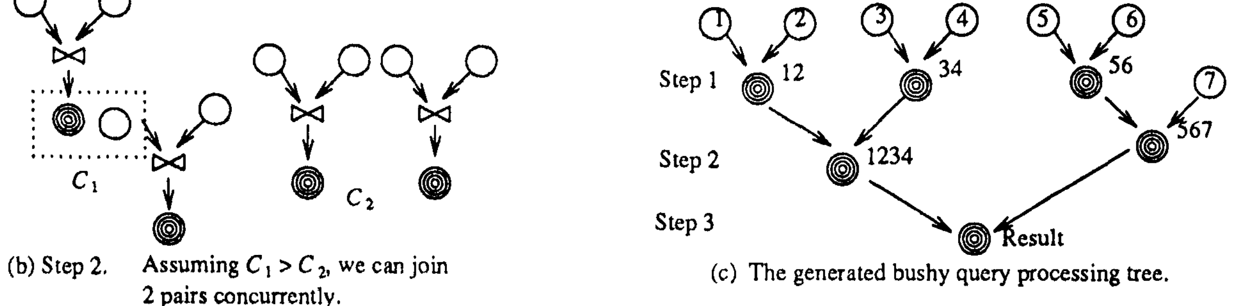
Figure 3.5. Example of total cost evaluation function.



(i) Iteration 1. Compare C_1 and C_2.

(ii) Iteration 2. Compare C_2 and C_3.

(a) Step 1. At iteration 1, $C_1 > C_2$. At iteration 2, $C_2 > C_3$. Since there are at most 3 pairs, the step terminates by pairing up 3 pairs to be joined in parallel.



(b) Step 2. Assuming $C_1 > C_2$, we can join 2 pairs concurrently.

(c) The generated bushy query processing tree.

Figure 3.6. Example of partial cost evaluation function.

intermediate results — relations bounded are executed serially — one relation is selected from bounded box

3.6b). C_1 and C_2 are computed and compared. If $C_1 > C_2$, then step 2 will join two pairs concurrently. The plan will terminate with another join. The plan generated thus comprises joining three pairs concurrently, followed by two pairs in parallel and finally one pair (Figure 3.6c).

3.3. Algorithm to pair up relations

In this section, we will discuss the detailed function repeatedly applied in both algorithms GP_P and GP_T — to select k pairs of relations and to compute the join cost. This problem can be viewed as a *matching problem* [Sysl83]: Given a join graph, select k subgraphs consisting of two nodes with a connecting edge. One straightforward way is to use matching algorithms available to find the maximum matching and then select the required number of pairs. To reduce the overhead of the matching process, we investigate some heuristics that help us to find an initial matching. The algorithm is shown in Figure 3.7. From the given join graph, a subgraph of two relations (nodes) with common join attributes are selected first according to some criterion. The same (or different) criterion is repeatedly applied to the remaining relations to select other pairs until either the desired number of pairs have been found or no more two connected nodes in the join graph can be found. In the latter case, we use the pairs found so far as an initial matching and apply the general matching algorithm, *matching*(G, N, S), to find the desired number of relation pairs. Function *matching* takes the join graph G and the set of initial pairing (less than N pairs) of relations, S, as the input and outputs a set containing N pairs of relations.

Algorithm to pair up relations

Input: G, a join graph, N, the number of pairs desired
Output: S, a set of N pairs of relations

$i \leftarrow 0$
$S \leftarrow \varnothing$

while (there are connected nodes in G) and ($i < N$) do {
    choose one pair { R_{i_1}, R_{i_2}} from G (based on some criterion)
    $S \leftarrow S \cup \{ R_{i_1}, R_{i_2} \}$
    $G \leftarrow G$ - { nodes of chosen pair and edges emitting from them }
    $i \leftarrow i+1$ }
if $i < N$ then
    $S \leftarrow$ **matching**(G, N, S)
return(S)

Figure 3.7. Function to pair up relations.

A number of heuristics have been used in different optimization algorithms to select two relations among others and join them together first with the intention to achieve some optimal ordering. The problem in our case is a little different: the chosen pairs are executed concurrently and their results will participate in subsequent joins. We studied the following four criteria to choose a pair among a set of relations:

1) $\min(N_i)$ and $\min(N_j)$ — select the relation with the smallest cardinality, followed by selecting from among the adjacent relations the one with the minimum cardinality
2) $\min(N_i)$ and $\max(N_j)$ — select the relation with the smallest cardinality, followed by selecting from among the adjacent relations the one with the maximum cardinality
3) $\min(JS_{ij})$ — select the pair of relations according to increasing order of the join selectivity
4) $\min(N_i N_j JS_{ij})$ — select the pair of relations that result in the smallest size of the intermediate result

These criteria aim at minimizing the intermediate relation sizes. Each criterion achieves this in different ways. Criterion 1 minimizes the immediate intermediate relation size by selecting two smallest source relations. Criterion 2, considering that the join chosen will be processed concurrently with others, hopes to achieve a global effect by averaging the intermediate relations. Criterion 3 considers the join selectivities hoping that by joining two relations with low selectivity it will result in a small relation size. Criterion 4 achieves the aim by using the resultant intermediate size as the yardstick. However, since several pairs are selected greedily, the use of any of the criteria suffers from the same shortcoming — the optimal set of pairs may not necessarily be selected. The performance of these criteria will be discussed later.

3.4. Processor allocation and join methods

The last issue to be addressed is the computation of the cost of chosen joins, that is the function of *Par_join_cost(R)* and *Seq_join_cost(R)* in Equation (3.1) and (3.2). As we assumed in Section 2, cost formulas for different join methods should be available for the query optimizer. With these formulas, it is straightforward to compute the cost of sequentially joining a set of relations, *Seq_join_cost(R)* by fixing the number of processors to the total number of processors available and query optimization techniques used in uniprocessor database management systems can be used. In our implementation, the computation starts with selecting two relations with the smallest resulting size and comparing the join costs of different methods. The join method with the minimum cost is added into the total cost. This process is repeated until all relations in the set (R) have been considered.

For N pairs of relations to be joined concurrently, the cost computation was complicated since the number of processors allocated to each of the joins will affect the join cost. For some particular processor allocation A_k, where p_i processors is allocated to the ith join and $\sum_{i=1}^{N} p_i = p$, the total number of processors available, the cost will be

$$Cost_k = \max_{1 \le i \le N} \; join_cost(i^{th}\ join, p_i)$$

The problem of computing *Par_join_cost(R)* becomes that of finding the processor allocation, A_{opt}, among all possible allocations A, and then compute the cost $Cost_{opt}$, such that

$$Cost_{opt} = \operatorname*{MIN}_{k \in A} \; Cost_k$$

In our current implementation of the algorithm, the processors are allocated to the N joins in the following manner: Initially, each join gets an initial assignment, that is a certain number of processors. This assignment is repeatedly adjusted by adding more processors to the most expensive join and/or removing processors from the cheapest one to average out the workload until no improvements can be made to the cost to compute the N joins.

3.5. Summary

In this section we have described two versions of a greedy query optimization algorithm GP, GP_T and GP_P, for multiprocessor computer systems in a top-down manner. The heuristics used to reduce the search space and simplify the cost computation are illustrated. To limit the search space, only synchronized QEPs are considered. To simplify the computation, the cost of a plan (subplan) is approximated by the sum of the cost of a parallel processing step and the cost of subsequent sequential processing cost in algorithm GP_T, and the same cost is approximated by the cost of the first parallel processing cost (or plus the cost of one more join) in algorithm GP_P. Furthermore, the optimization heuristics widely used in uniprocessor systems, such as those determining the order of joins are also applied.

The complexity of the algorithm can be roughly estimated as follows. In each execution of the *while-loop* of GP, we obtain one step of the execution plan. Since there are at most n relations, the loop will not be executed more than n times. The generation of each step of the plan is performed by function *Select_rel_pairs*, where the *repeat-loop* is executed at most $n/2$ times. For each of the execution of the *repeat-loop*, several pairs to be joined are selected. This takes at most $O(n^3)$, which is the complexity of a *matching* algorithm. The allocation of processors has time complexity of $O(p \times nlgn)$, where p is the number of processors. Therefore, the algorithm GP has complexity of

$$O(n^2 \times (n^3 + p \cdot n \cdot lgn)) = O(n^5 \cdot lgn), \quad \text{for } p < n^2$$

Note that the n^3 comes from the matching algorithm used when pairs of relations are selected. In fact, the matching algorithm is called only when the heuristics used cannot find enough number of relation pairs. Most of the time, the *matching function* needs not be called as the heuristic provides the answer. This will reduce the time complexity of selecting relation pairs to $O(n^3 \cdot lgn)$ instead of $O(n^5 \cdot lgn)$.

4. A Performance Study

To evaluate the algorithms described in the above section, an experimental study is conducted with the following purposes:

(1) to compare the four criteria for selection of pairs of relations (used in the algorithm of pairing relations) and

(2) to evaluate and compare the effectiveness of algorithm GP with both heuristics, GP_T and GP_P in generating optimal plans.

The optimization algorithm, algorithm GP, is implemented in our study. However, the input queries are randomly generated according to chosen parameters and execution costs of generated QEPs are calculated according to the developed cost models. Therefore, the results presented here are basically simulation results since no multiprocessor database system is available in our organization yet. We hope that these results can give us some insight into our algorithm and provide us with some experience to implement it in real systems.

Though recent work [Kris86, Swam88, Swam89] have emphasized on large number of joins, we believe that for most traditional applications in a well-designed relational database system, most of the queries will require only a small number of joins. Therefore, we study the proposed algorithm on a small number of joins ($\leq$ 10). We vary the join selectivities, the sizes of the relations, the number of processors and the number of tuples per page. However, our algorithm is also applicable for large number of joins (> 10).

We define the following measure to study the performance of our algorithm :

$$CostMultiplier(A_1, A_2) = \frac{Cost_{A_1}}{Cost_{A_2}}$$

where $Cost_{A_i}$, (i = 1, 2), represents the cost of executing the QEP generated by algorithm A_i. *CostMultiplier*(A_1, A_2) is thus a measure of the relative performance of algorithm A_1 over algorithm A_2.

For the experiments with small number of joins, we are able to compute the optimal solution by enumerating all possible combinations. We therefore use the optimal solution generated by exhaustively trying all possibilities as our basis for comparison. Hence, we have

$$CostMultiplier(GP, OPT) = \frac{Cost_{GP}}{Cost_{OPT}}$$

where $Cost_{OPT}$ and $Cost_{GP}$ are the costs of plans generated by the exhaustive search [2] and the algorithm GP used respectively. It is clear that *CostMultiplier(GP*,

[2] The search space only includes all feasible *synchronized QEPs*. So, the optimal here may not be the real optimal. However, this seems the best reference we can use in our experiment.

OPT) ≥ 1 and a lower bound value of one implies that algorithm *GP* generates the optimal answer.

4.1. Experiment 1 : Criteria for selecting the joining pairs

In this experiment, we conducted several tests to study the criteria used for selection of joining pairs (see Section 3.3). The main parameters of the queries used in the experiment are shown in Table 4.1.

Parameters		Relation Size (in pages)	
		750 — 850	600 — 1000
Join Sel.	0.0008 — 0.002	Test 1	Test 3
	0.0007 — 0.004	Test 2	Test 4

Table 4.1. Experimental setup.

For example, in test 1, the join selectivity is varied from 0.0008 to 0.002 while the relations sizes are in the range of 750 to 850 pages. These are varied according to the uniform distribution such that the final relation size is also in the range of 750 to 850 pages. The other tests are similar except for the parameter settings.

The number of processors are varied from 5 to 32 for the tests. For each test, more than 2500 multi-way join queries with different number of joins, relation sizes, join selectivities and number of processors are generated. A *query generator* is used to generate queries. The QEPs of these queries are generated by applying algorithm *GP* with all the four criteria. The average costs by using different criteria were compared with that of criterion 1 and Table 4.2 summarizes the results. Those numbers greater than one means that the criterion performs worse than the first criterion.

Experiment Set	GP_T			
	C1	C2	C3	C4
Test 1	1.0000	1.0078	0.9912	0.9899
Test 2	1.0000	1.0000	0.9872	0.9872
Test 3	1.0000	1.0037	1.0023	0.9958
Test 4	1.0000	0.9993	0.9782	0.9734

(a) Performance for GP_T

Experiment Set	GP_P			
	C1	C2	C3	C4
Test 1	1.0000	1.0004	0.9976	1.0003
Test 2	1.0000	1.0003	0.9963	0.9963
Test 3	1.0000	1.0024	1.0005	0.9943
Test 4	1.0000	1.0012	0.9919	0.9922

(b) Performance for GP_P

Table 4.2. Comparison of criteria.

From Table 4.2, we note that the performance of the various criteria are relatively close to one another for both heuristics. No single criterion is superior in all situations. Several factors contribute to this — the number of processors, the join selectivities and the relation sizes. By varying these factors, the heuristics when used with one criterion may outperform the others. Criterion 3, which is shown to be the best criterion in [Swam89], no longer dominates. The possible reason is that, in our parallel processing environment, a number of pairs are joined concurrently and the overall performance depends on the combination of the optimality of choosing all these pairs. The policy of choosing one best pair may lead to the situation that the execution costs of other pairs are too high and the overall cost increases. Base on the above results, we use criterion 4, which generates more near optimal plans than others, in subsequent experiments.

4.2. Experiment 2 : The base experiment

We first study the performance of the algorithm *GP* by comparing the results with that of the optimal result. The algorithm *GP*, using both cost evaluation functions, and the exhaustive search program are applied to the queries generated by the query generator. The exhaustive search method used always generate the optimal plan for a given query. The plans generated by algorithm *GP*, using either of the cost evaluation functions, are then tested for optimality by comparing them with the optimal ones. The parameters and their settings that controlled the query generator for the base experiment is shown in Table 4.3. We vary the number of relations from 4 to 7. For each variation, we collect 500 sets of data with different parameter settings. The join selectivity, *JoinSel* and the relation sizes, *RelSizes* are uniformly distributed over 0.0009 — 0.002 and 750 — 850 pages respectively. The number of processors available varies from 5 to 32.

Parameter	Meaning	Setting
JoinSel	Join Selectivities.	0.0009 — 0.002
NumRel	No. of Relations	4 — 7
RelSizes	Size of Relations	750 — 850
NumProAvail	No. of Processors	5 — 32

Table 4.3. Base experiment parameters settings.

Tables 4.4 and 4.5 show the results of this experiment by showing the percentages of QEPs generated by our algorithm, *GP*, that fall in the different ranges of the metric, *CostMultiplier*. For example, in Table 4.4, with a 3-way join (4-R), GP_T generates, in fact, optimal plans. With 4-way join (5-R), the costs of 11.4% of QEP's are less than 1.1 of the cost of the optimal plans. From Table 4.4, we see that GP_T performs well for small number of joins. As expected, the percentage of optimal solution decreases as the number of relations increases since the search space increases drastically (as the number of relations increases). However, even for 6-way join, as high as 70% of the plans generated are optimal. The effectiveness of the algorithm GP_T is apparent since all the QEPs generated have costs no more than 10% over the costs of the optimal plans.

Query Type	CostMultiplier (%)			
	1.0	1.0 — 1.1	1.1 — 1.2	> 1.2
4-R	100	0	0	0
5-R	88.6	11.4	0	0
6-R	77.3	22.7	0	0
7-R	72.0	28.0	0	0

Table 4.4. *CostMultiplier* for GP_T.

Query Type	CostMultiplier (%)			
	1.0	1.0 — 1.1	1.1 — 1.2	> 1.2
4-R	98.7	1.3	0	0
5-R	83.8	16.2	0	0
6-R	45.3	38.0	10.3	6.4
7-R	56.0	40.0	4.0	0

Table 4.5. *CostMultiplier* for GP_P.

Table 4.5 shows that GP_P performs well for small number of joins too. Except for six relations, GP_P generates QEPs with *CostMultiplier* less than 120%. For six relations, the performance is poor due to the greediness of the heuristic. With six relations, when three pairs are joined in the first iteration, subsequently only sequential joins may be done. However, a possibly better plan might be to join two pairs first, follow by two pairs before the final sequential join is performed.

Each cost evaluation function has its own advantages. While GP_T not only generates higher percentage of optimal solutions, it also generates nearer-optimal solutions (that is the values are nearer to optimal than GP_P). This is expected as it considers the total cost to complete the entire M-way joins. On the other hand, GP_P is superior in that it generates a good plan in a shorter time. We have observed in our experiments that, though both approaches generate a plan in less than a second, GP_T takes about twice as long to produce a plan. The exhaustive approach, on the other hand, takes as long as several hours to produce an optimal plan.

4.3. Experiment 3 : Vary sizes of relations

In this experiment, we study how the relation sizes affect the generation of parallel execution plans. The relations sizes are varied over a wider range of values from 600 to 1000. The join selectivities are kept in the same range as the base experiment. However, the selectivities are chosen such that the final result size is in the range of 750 — 850. Tables 4.6 and 4.7 show the results.

Query Type	CostMultiplier (%)			
	1.0	1.0 — 1.1	1.1 — 1.2	> 1.2
4-R	93.3	6.0	0.7	0
5-R	75.2	24.2	0.6	0
6-R	68.0	29.8	2.2	0
7-R	69.0	31.0	0	0

Table 4.6. *CostMultiplier* for GP_T.

Query Type	CostMultiplier (%)			
	1.0	1.0 — 1.1	1.1 — 1.2	> 1.2
4-R	89.3	10.0	0.7	0
5-R	72.6	26.8	0.6	0
6-R	50.2	36.8	9	4.0
7-R	56.5	36.0	7.5	0

Table 4.7. *CostMultiplier* for GP_P.

Compare the results in the above two tables with those in experiment 2, we find that the optimality of QEPs generated decrease. The algorithm, however, remains effective. For GP_T (Table 4.6), in all cases, there are still at least 68% of the plans generated which are optimal and all plans have cost less than 120% of the optimal one. On the other hand, the plans generated by GP_P are at least 50% optimal, 85% with *CostMultiplier* less than 110%.

4.4. Experiment 4 : Combination of small and large relation sizes

In this experiment, we study how a mixture of small and large relation sizes affect the generation of parallel execution plans. The relations sizes are varied from two ranges : 750 — 850 and 5000 — 6000. This mixture simulates the situation where some small size relations may be joined with very large size relations. Tables 4.8 and 4.9 show the results.

Query Type	CostMultiplier (%)			
	1.0	1.0 — 1.1	1.1 — 1.2	> 1.2
4-R	86.0	14.0	0	0
5-R	75.0	21.0	4.0	0
6-R	70.0	25.0	5.0	0
7-R	66.0	33.0	1.0	0

Table 4.8. *CostMultiplier* for GP_T.

Query Type	CostMultiplier (%)			
	1.0	1.0 — 1.1	1.1 — 1.2	> 1.2
4-R	86.0	14.0	0	0
5-R	45.0	35.0	20.0	0
6-R	29.0	50.0	20.0	1.0
7-R	48.0	27.0	23.0	2.0

Table 4.9. *CostMultiplier* for GP_P.

Tables 4.8 and 4.9 indicate that, with large variation of relation size, performance GP_T still performs quite well, but GP_P does not perform so good, especially when the number of relations increases. This implies that if there is a large variance among the relations to be joined, it is better to use GP_T in order to obtain better plans, with the price of high optimization overhead.

4.5. Experiment 5 : Increase the number of joins

From experiments 2 to 4, we see the effectiveness of the proposed algorithm GP. The purpose of this exper-

iment is to see the relative performance of GP_T and GP_P for large number of joins. Since an exhaustive enumeration of the join orderings is computationally expensive, we compare them with one another. We vary the join selectivities and the sizes of the relations. Tables 4.10 and 4.11 show the relative performance of heuristic GP_T over GP_P with parameter settings from experiments 2 and 3 respectively.

Query Type	*CostMultiplier (%)*			
	0.6 — 0.8	0.8 — 0.9	0.9 — 1.0	1.0 — 1.1
10-R	1.4.8	22.8	71.8	1.6
20-R	7.75	26.75	64.25	1.25
30-R	5.5	26.0	67.0	1.5
40-R	4.0	38.25	56.5	1.25
50-R	2.75	33.5	63.0	0.75

Table 4.10. *CostMultiplier* (GP_T, GP_P). (Experiment 2 settings)

Query Type	*CostMultiplier (%)*			
	0.6 — 0.8	0.8 — 0.9	0.9 — 1.0	1.0 — 1.1
10-R	5.6	17.2	75.4	1.8
20-R	4.75	28.75	65.00	1.5
30-R	5.5	30.5	62.75	1.25
40-R	3.0	30.0	65.75	1.25
50-R	4.75	37.25	57.25	0.75

Table 4.11. *CostMultiplier* (GP_T, GP_P). (Experiment 3 settings).

From Tables 4.10 and 4.11, we see that GP_T outperforms GP_P most of the time (> 98%). For more than 50% of the time, GP_P produces results that are close to GP_T. Up to 90% of the results generated by GP_P are 80%-near-GP_T.

5. Conclusion

In this paper, we have examined the problem of generating parallel plans for multi-way join in multiprocessor computer systems comprising conventional, commercially available components without the assistance of any special-purpose hardware components. While traditional optimizers (which do not generate parallel plans) deal with choosing an appropriate join method and the best join ordering, our optimizer that generates parallel plans, must also select the pairs of relations to be joined in parallel and allocate processors to the join operations. We proposed an algorithm, algorithm GP, which employs the greedy paradigm to generate parallel QEPs for multi-way join queries. The plan generated exploits parallelism at two levels : *intra-join parallelism* where several processors may be assigned to a join operation and *inter-join parallelism* where several joins may be performed concurrently.

Two cost evaluation functions were proposed to compute the cost of a QEP with both concurrently and sequentially executed joins. These two cost evaluation functions lead to two versions of GP, GP_T and GP_P. Algorithm GP_T uses the *total* cost of the entire M-way join at each step to determine the number of join operations for each step, while Algorithm GP_P uses the *partial cost*, i.e. cost to execute joins of the current step to compare two QEPs. Our study shows that, for small number of joins, both heuristics always generate plans with cost no more than 120% of the optimal plans. Algorithm GP_T outperforms Algorithm GP_P in most of the cases. However, the time to generate the plan is longer. We also investigated four criteria to guide the selection of pairs of relations to be joined first. The results show that the average performance of the criteria are relatively close to one another in parallel processing environment.

Our work may be extended in the following areas. First, we plan to relax the assumption of our study that the execution of a QEP is synchronized. By restricting to synchronized QEPs, processings at certain step cannot begin until all the joins at the previous step have been completed. This is reasonable if we can minimize the overall execution time at each step by distributing the processors to balance the load. However, this approach has its shortcoming that the effect of pipeline among joins, as in the execution scheme suggested by Schneider and DeWitt [Schn90], is not considered. Second, previous work have applied combinatorial algorithms and simulated annealing as heuristics to the problem of optimization to generate a sequential plan. Such algorithms have been shown to be effective. They may be further studied and modified to produce plans for parallel execution. We may also introduce a k-step (k > 1) look-ahead in our heuristics to enhance their performance. Third, there are a number of issues and implementation details of the algorithm that need to be further studied. For example, for allocating processors to joins, our current implementation may not be very effective and efficient when the number of processors available is large. The last, but very important issue to be addressed is to move as much work as possible from runtime to compilation time. Since the number of processors available is only known until runtime, our algorithm presented in this paper is to be executed at runtime. Although most of our effort is to reduce the overhead by using heuristics, precompilation, or most likely partial precompilation, in this new environment might still be an effective way of reducing the runtime overhead. This possibility is not explored yet.

References

[Bora90] Boral, H., *et. al.*, "Prototyping BUBBA, A Highly Parallel Database System," *IEEE Trans Knowledge and Data Eng.*, Vol. 2, No. 1, Mar. 1990, pp. 4-24.

[Bult89] Bultzingsloewen, G. v., *et. al.*, "Design and Implementation of KARDAMOM — A Set-Oriented Data Flow Database Machine," *Proc. 6th Intl. Workshop on Database Machines*, Springer-Verlag Lecture Notes, Vol. 368, Jun. 1989, pp. 18-33.

[Deen90] 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, Jul. 1990, pp. 92-102.

[DeWi85] DeWitt, D. J., and Gerber, R., "Multiprocessor Hashed-Based Join Algorithms," *Proc. VLDB 85*, Stockholm, Aug. 1985, pp. 151-164.

[DeWi90] DeWitt, D. J., *et. al.*, "The GAMMA Database Machine Project," *IEEE Trans Knowledge and Data Eng.*, Vol. 2, No. 1, Mar. 1990, pp. 44-62.

[Ioan90] Ioannidis, Y. E. and Kang, Y., "Randomized Algorithms for Optimizing Large Join Queries," *Proc. SIGMOD 90*, May 1990, pp. 312-321.

[Kits90] Kitsuregawa, M. and Ogawa, Y., "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)," *Proc. VLDB 90*, Australia, Aug. 1990, pp. 210-221.

[Kris86] Krishnamurthy, R., Boral, H., and Zaniolo, C., "Optimization of Nonrecursive Queries," *Proc. VLDB 86*, Kyoto, Aug. 1986, pp. 128-137.

[Lohm85] Lohman, G. M., *et. al.*, "Query Processing in R^*," in *Query Processing in Database Systems*, Kim, W., *et. al.* (editors), Springer-Verlag, 1985.

[Lu90] Lu, H. J., Tan, K. L. and Shan, M. C., "Hash-based Join Algorithms for Multiprocessor Computers with Shared Memory," *Proc. VLDB 90*, Australia, Aug. 1990.

[Ono90] Ono, K. and Lohman, G. M., "Measuring the Complexity of Join Enumeration in Relational Query Optimization," *Proc. VLDB 90*, Australia, Aug. 1990.

[Schn89] 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, Jun. 1989, pp. 110-121.

[Schn90] Schneider, D. A. and DeWitt, D. J., "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," *Proc. VLDB 90*, Australia, Aug. 1990.

[Seli79] Selinger, P. G., *et. al.*, "Access Path Selection in a Relational Database Management System," *Proc. SIGMOD 79*, Boston, Massachusetts, Jun. 1979, pp. 23-34.

[Ston88] Stonebraker, M., Katz, R., Patterson, D., and Ousterhout, J., "The Design of XPRS," *Proc. VLDB 88*, Los Angeles, Aug. 1988, pp. 318-330.

[Su88] Su, Y. W. Stanley, "Database Computers : Principles, Architectures and Techniques," McGraw-Hill Intl. Edition, Computer Series, 1988.

[Swam88] Swami, A. and Gupta, A., "Optimization of Large Join Queries," *Proc. SIGMOD 88*, Chicago, Illinois, Jun. 1988, pp. 8-17.

[Swam89] Swami, A. , "Optimization of Large Join Queries : Combining Heuristics and Combinatorial Techniques," *Proc. SIGMOD 89*, Portland, Oregon, Jun. 1989, pp. 367-376.

[Sysl83] Syslo, M. M., Deo. N. and Kowalik, J. S., "Discrete Optimization Algorithms with Pascal program," Prentice-Hall, 1983.

[Tera83] Teradata Corporation, DBC/1012 Database Computer Concepts and Facilities, Inglewood, CA, Apr. 1983.

[Vald84] Valduriez, P., and Gardarin, G., "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *ACM Trans. Database Systems*, Vol. 9, No. 1, Mar. 1984, pp. 133-161.

[Wolf90] Wolf, J. L., Dias, D. M. and Yu, P. S., "An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew," *Proc. 2nd Intl. Symp. Databases in Parallel and Distributed Systems*, Dublin, Ireland, Jul. 1990, pp. 103-115.