PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Optimization and Execution of Large Join Queries

Eileen Tien Lin *
Edward Omiecinski

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Sudhakar Yalamanchili

School of Electrical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

Optimizing large join queries that consist of many joins has been recognized as NP-hard. In this paper, we examine the feasibility of exploiting the inherent parallelism in optimizing large join queries, on a hypercube multiprocessor. This includes not only using the multiprocessor to answer the large join query, but also to optimize it. Two heuristics are provided for generating an initial solution, which is further optimized by an iterative local-improvement method. The entire process of parallel query optimization and execution is simulated on an Intel iPSC/2 hypercube machine.

1 Introduction

A large join query consists of a series of relational database join operations. The order in which these joins are executed has a great impact on the response time. The fundamental problem with optimizing large join queries is searching the large solution space of possible query execution plans.

In [IK84], the optimization of N-relational joins, using the nested-loop join method is proven to be NP-complete. In [KBZ86], a generic cost formula is assumed applicable to the join methods used. They extend a polynomial time optimization algorithm for tree queries[IK84] to the more general case. This algorithm is also improved to an $O(N^2)$ solution where N is the number of relations in the query.

Several researchers have been studying the feasibility of applying general *combinatorial optimization techniques*, such as Simulated Annealing and Iterative Local-Improvement to avoid exhaustive enumeration of all plans. In [SG88], the solution space consists of only *outer linear join processing trees* where at most one intermediate result is active and the inner relation is always a base relation. Furthermore, they assumed that the database resides in main memory. Later in [Swa89], they propose a set of heuristics to be combined with the combinatorial techniques in order to improve the performance. In [IK90], a new Two Phase Optimization algorithm is presented which runs Iterative Local-Improvement for a small period of time and uses the output of this phase as the initial solution for the second phase that runs Simulated Annealing.

In [DKT90] and [SD90], different strategies for processing large join queries in a parallel environment are discussed. In [DKT90], the authors study how to execute a large join query on a shared memory parallel computer. In [SD90], they show how a different representation of a query tree can affect the degree of parallelism within a query and performance. Specifically, they compare *left-deep* and *right-deep* tree representations.

In this paper, we investigate the issue of using the inherent parallelism in a hypercube multiprocessor to optimize large join queries. Both *inter-join* and *intra-join* parallelism are exploited in forming the plan, which implies that a join can be performed on a subcube of any size and more than one join can be performed at a time.

*Currently at IBM Corporation, 555 Bailey Avenue, San Jose, California 95150

2 The Parallel Query Processing Model

Our parallel query processing model is predicated on the following parallel architecture model. We have P = 2^d processors interconnected in a d-dimensional binary hypercube. Each processor, with address $p_{d-1}, p_{d-2}, \ldots, p_i, \ldots, p_1, p_0$, is connected to every other processor whose address is $p_{d-1}, p_{d-2}, \ldots, \overline{p_i}, \ldots, p_1, p_0, \forall i$, where $\overline{p_i}$ is the bit complement of p_i. Communication between non-adjacent processors is realized by routing messages between intermediate nodes. Every processor (*node*) has its own memory and interacts with other processors via message passing. In this paper, we refer to an n-dimensional hypercube of 2^n nodes as an n-cube. A subcube is a subset of processors that forms a smaller hypercube. For the purposes of our study we assume the complete hypercube is available for performing the joins.

Based on this architecture model, the query processing model consists of the following steps.

1. The host preprocesses a query and transforms it into an internal form such as a join graph.
2. The host accesses the global database dictionary for relevant statistics for each relation and each join.
3. The host selects a query optimization strategy for each node. The query and the selected strategy are sent to all nodes in the system.
4. Each node follows its specified strategy to generate an initial plan and to optimize it to make the best parallel query execution plan. This plan is then reported to the host.
5. The host selects the best plan from all of the nodes, schedules the query, and sends the plan to all participating nodes. Each node then executes the plan.

A distinct feature of our research is to exploit the inherent parallelism of the optimization step, instead of relying on only the host to generate a good plan.

3 Assumptions

1. Each relation is *horizontally partitioned* over a subcube within the system. Relations may be allocated to different subcubes of different sizes. Tuples are assumed to be *uniformly* distributed across nodes within a subcube.
2. The queries considered only involve natural joins, i.e. equi-joins. For simplicity, we consider only two-way joins that use the Cube Hybrid-Hash join algorithm[OL89].
3. The system is assumed to be dedicated to this application. Every node is available for both optimization and computation of the joins.
4. A join can be performed on any subcube of any size. More than one join can be simultaneously performed in disjoint subcubes.
5. The values of attributes are distributed *uniformly and independently* of each other. This implies that the size of $R \bowtie S \bowtie T$ can be estimated by multiplying the cardinalities of the three relations and the two join selectivity factors.

4 Definitions

Parallelism in the execution of joins requires the allocation/deallocation of subcubes to relations and join computations. In our approach we use the *binary buddy* system to manage subcubes for relations and join operations. In the binary buddy system the hypercube is recursively partitioned into subcubes. The subcubes can be represented by a binary tree as follows. Associate with each node a status bit that is 1 (0) if the processor is available (busy). The leaf nodes represent the status bits of the nodes. The status bit associated with any interior node is 0 if any of the leaf nodes in the corresponding subtree is 0, and 1 otherwise. The root is at level 0 and the nodes at level i are associated with subcubes of dimension $n-i$. When a request for 2^k processors arrives, nodes at level $n-k$ in the tree are searched to find the first available one. If found, it is allocated and the status bits of all the parent nodes are adjusted accordingly. Similar updates to this structure take place on the deallocation of nodes.

The set of all processors that form a subcube of any size in a hypercube can be identified by a unique *cube identifier*. A cube identifier is an address mask. For example the 2-cube consisting of processing elements 0,2,8 and 10 are uniquely identified by the mask (*0*0), where * represents a don't care. Subcubes that can be allocated under the binary buddy system can be identified with cube identifiers of the form $p_{d-1}, p_{d-2}, \ldots, * * \ldots * *$, where the number of *'s is equal to the dimension of the subcube. This assumes that the recursive decomposition proceeds highest dimension first, followed by the next highest dimension, and so on.

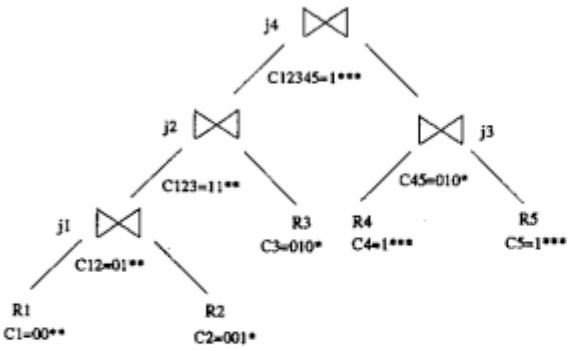

Figure 1: The Tree Representation of a Parallel Plan.

The following definitions are necessary to describe the dependencies between two joins.

Definition 4.1 *A join i is* **data-dependent** *on a join j, where $i \neq j$, if at least one of the operands in i depends on the result of j.*
A join i is **immediately-data-dependent** *on a join j, where $i \neq j$, if the following two conditions are true:*

1. *i is data-dependent on j.*
2. *j is the most immediate join prior to i that produces one of the operands in i.*

Definition 4.2 *A join i is* **location-dependent** *on a join j, where j is to be performed before i, if $C_i \cap C_j \neq \emptyset$, and i (j) is performed on cube C_i (C_j).*
A join i is **immediately-location-dependent** *on a set of joins J, where $i \notin J$, if the following two conditions are true:*

1. *i is location-dependent on k, $\forall k$, where $k \in J$.*
2. *$\nexists k_1$ that is location-dependent on k_2, where $k_1 \in J$ and $k_2 \in J$.*

Definition 4.3 *A join i is* **transfer-dependent** *on a cube C_j if all of the following conditions are true:*

1. *At least one of the operands for i, R_{1i}, is currently stored on C_j.*
2. *$C_i \neq C_j$, where join i is to be performed on C_i.*
3. *Join i is the first join that involves R_{1i} in the plan.*

As an example, consider the following query for a 4-cube, q_i,

$$R_1 \bowtie_{R_1.A=R_2.A} R_2 \bowtie_{R_2.B=R_3.B} R_3$$

$$\bowtie_{R_3.C=R_4.C} R_4 \bowtie_{R_4.D=R_5.D} R_5$$

Figure 1 is an arbitrary parallel plan for q_i. C_1 refers to the cube where relation R_1 is currently stored. C_{12} refers to the cube where the join between R_1 and R_2 is to be performed. The following dependencies exist:

1. j_1 is transfer-dependent on C_1 and C_2.
2. j_2 is immediately-data-dependent on j_1 and transfer-dependent on C_3.
3. j_3 is immediately-location-dependent on j_1 and transfer-dependent on C_4 and C_5.
4. j_4 is immediately-location-dependent on j_2 and immediately-data-dependent on j_2 and j_3.

5 The Optimization Process

The optimization process on every node consists of two steps as in [Swa89]. First, every node applies the specified heuristic to produce a feasible plan. Each of these initial plans is subject to combinatorial optimization in the second step.

In the following discussion, we represent a join plan as a binary tree, where all the non-leaf nodes represent join operations and the leaf nodes represent base relations. The cost of a plan refers to the execution time to complete this plan. The height of a plan (tree) refers to the number of join operations on the longest path from a node to the root.

5.1 Evaluating a Parallel Large Join Plan

This algorithm is used in the generation of an initial plan, and is used iteratively in the subsequent optimization of this plan.

We provide an algorithm for estimating the cost of a parallel large join plan. We require that all transfer-dependencies associated with every node be resolved before any join can be performed on this node.

To expedite the evaluation of a parallel large join plan, the following information is necessary: i, R_{1i}-ready, R_{2i}-ready, C_{12i}-ready, Start_time and Complete_time. The first two ready fields are used to indicate the existence of any immediate-data-dependency or transfer-dependency. R_{1i}-ready is set to j if i is immediately-data-dependent on j. R_{1i}-ready can also be set to C_k, if i is transfer-dependent on the cube

C_k. This means that i cannot be started before relation R_{1i} is transferred from C_k, and that i is the first join that involves R_{1i}. In this case, we require that R_{1i} be transferred to C_{12i} before any join can be performed on C_k. C_{12i}-ready is set to J if i is immediately-location-dependent on a set of joins J. When a join i is free of any immediate dependencies, all three ready fields are set to -1.

Start_time(i) is used to record the time at which i is started. This is set to the earliest time that i is free of any dependency. If i is immediately-data-dependent on j, its earliest Start_time will be the Complete_time of j plus the time to transfer the result of j. If a join i is transfer-dependent on a cube C_k because of R_{1i}, its earliest Start_time will be after R_{1i} is transferred from C_k. The earliest Start_time for a join j on a cube C_j is the earliest time when any of the nodes in C_j is done transferring data for any transfer-dependency. The time to complete this plan is thus the largest Complete_time among all joins.

The following notation is used in the following discussion, for a join i :

- i is immediately-data-dependent on joins $d1$ and $d2$.
- Transfer_time(R_{d1}) stands for the time to transfer the join result of $d1$ for i after $d1$ is completed. Notice that Complete_time(-1) and Transfer_time(R_{-1}) are both 0.
- i is immediately-location-dependent on the set of joins J.
- i is transfer-dependent on a cube C_{1k} because of R_{1k}, and on C_{2k} because of R_{2k}. The first join to occupy C_{1k} is the join $1k$, and the first join to occupy C_{2k} is the join $2k$.
- i is to be performed on C_i.
- Transfer_time(R_{1k}) stands for the time to transfer the relation $1k$ from C_{1k} to C_i.

Algorithm 5.1 *This algorithm is used to estimate the cost of a parallel large join plan.*

1. *Set all the ready fields by checking if there is any immediate data or location dependency between a join and all other joins prior to it, or any transfer-dependency between a join and any cube.*
2. *Initialize all Start_time's to* 0.
3. *Initialize all Complete_time's to infinity.*
4. *For every join i , $1 \leq i \leq n-1$, i in ascending order, n is the number of relations :*
   For every transfer-dependency of i on C_{jk}, $1 \leq j \leq 2$, do the following :
   (a) *Update the Start_time of i, to be* $Transfer\_time(R_{jk}) + \max(Start\_time(i), Start\_time(jk))$.
   i cannot start before the transfer of R_{jk} is completed. Notice that a previously positive Start_time(i) indicates that i was previously transfer-dependent on another cube or C_i was previously involved in some transfer-dependency.
   (b) *Update the Start_time of jk, to be* $Start\_time(i)$.
   We assume that jk cannot start before the transfer of R_{jk} is completed.
   (c) *Update the corresponding ready field for i and the locations of the associated relations.*
5. *Repeat Steps 6 and 7 until all joins are completed :*
6. *Compute the Complete_time for every uncompleted join that is dependency-free. Update the locations of the involving relations.*
7. *For every join i that has not been completed, $1 \leq i \leq n-1$, i in ascending order*
   For every immediate-data-dependency on dj, $1 \leq j \leq 2$,
   If dj has been completed, do the following :
   (a) *Reset the corresponding ready field.*
   (b) *Update the Start_time of i, to be* $\max(Start\_time(i), Complete\_time(dj)) + Transfer\_time(R_{dj})$.

   If i is free of any immediate-data-dependency, and i has immediate-location-dependencies on the set of joins J,
   if every join in J has been completed, do the following :
   (a) *Reset the corresponding ready field.*
   (b) *Update the Start_time of i, to be* $max(Start\_time(i), \forall_{l \in J} Complete\_time(l))$.

5.2 Initial Plan Generation

The complexity of large join optimization on a hypercube multiprocessor does not only involve finding

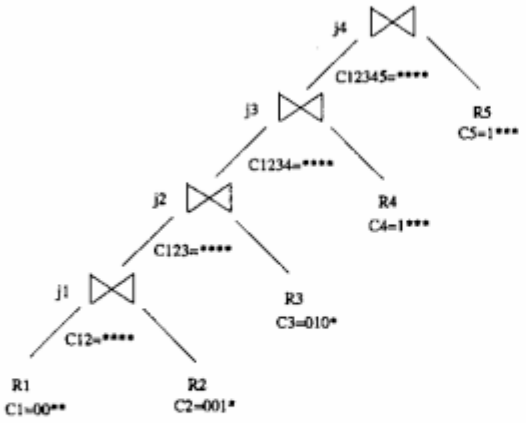

Figure 2: An Example of a Parallel Plan using the Maximum Intra-Join-Parallelism Heuristic.

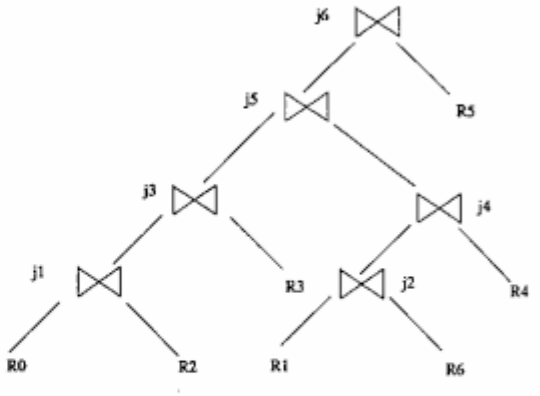

Figure 3: A Plan Generated by the Maximum Inter-Join-Parallelism Heuristic for q_j.

the best order for executing the join operations, but also the best mapping between a join and the subcube where it is to be performed. We provide the following two heuristics for mapping join operations to subcubes in order to produce an initial solution. The Maximum Inter-Join-Parallelism heuristic tries to reduce the height of the tree as much as possible. The Maximum Intra-Join-Parallelism heuristic always uses the largest cube to perform each join. These heuristics can be categorized as *greedy* heuristics.

5.2.1 Maximum Inter-Join-Parallelism

This heuristic tries to invoke as many joins in parallel as possible at the same time, and therefore, increases the degree of inter-join-parallelism. By invoking more joins in parallel, each join is allocated a smaller cube, however the height of the plan is reduced. Even though this plan may have a smaller height, each join may incur a higher cost.

The parallel plan for query q_i in Section 4, which is shown in Figure 1, could be produced by this heuristic. Due to constraints in the query, at most two joins can be performed in parallel. Therefore, j_1 and j_2 share the cube. Since the following two joins have to be performed sequentially, the largest cube is allocated to each join.

5.2.2 Maximum Intra-Join-Parallelism

The maximum degree of parallelism is applied to every join to reduce the individual cost, but at the cost of redistributing the relations prior to all the joins. Since every join uses the same set of nodes, a chain of immediate-location-dependencies is formed. As a result, all joins are forced to be performed sequentially.

Figure 2 is a possible parallel plan for the query q_i in Section 4 using this heuristic. Every join is allocated the largest cube, i.e., the whole system. Note that not all plans generated by this approach are necessarily binary linear processing trees[KBZ86] as shown in Figure 2, in which at most one intermediate relation is used as an input to subsequent join operation.

5.3 Combinatorial Optimization

Any plan for a large join query can be thought of as a *state* in a solution space which includes all possible plans. The ultimate goal of any optimization process is to find a state with the globally optimum cost.

We use a simple combinatorial optimization technique, *Iterative Local-Improvement*. In this technique, the current plan is transformed into a new plan by performing one move such as swapping the relative orders of two joins. If the new plan has a lower cost (as computed by Algorithm 5.1), it becomes the current plan. This process in general continues until a local optimum is found.

We now discuss how we optimize plans generated by each of the heuristics described in the previous section. Note that a local optimum is reached by a node when further local improvement is not possible. The best plan chosen by the nodes is selected as the parallel large join plan.

5.3.1 Maximum Inter-Join-Parallelism

Following the initial application of this heuristic, there is a limit to what extent the height of the plan (tree) can be reduced for a given query. Consider the query q_j, joining 7 relations, R_0 through R_6, where R_6 joins with relations R_1,R_3,R_4 and R_5; and R_2 joins with R_3 and with R_0. Since R_6 is to be joined with R_1,R_3,R_4 and R_5, only one of these four joins can be performed at a time. Figure 3 is a possible plan which has j_1 and j_2 performed in parallel. Following that, j_3 and j_4 are performed in parallel. Finally j_5 is performed followed by j_6. We optimize plans generated by the application of the Maximum Inter-Join-Parallelism heuristic as follows :

- Globally, each node chooses a different maximal independent set of relations, such that if two joins i and j are in the same independent set, we can perform i and j at the same time on two disjoint cubes.
- Locally, each node can swap the join locations of two randomly chosen joins for every independent set until a local optimum is reached.

5.3.2 Maximum Intra-Join-Parallelism

To optimize plans generated by this heuristic, we take an approach similar to that described using the Maximum Locality heuristic. However, since the join locations are fixed, i.e., the entire system, there is no need to alter the join locations.

6 Performance Evaluation

In this section, we present our experimental evaluation of the two heuristics for parallel large join query optimization.

6.1 System Description

The different heuristics and the entire process were coded in C and the experiments were run on a 16-node Intel iPSC/2 hypercube. To simulate a disk per node, we implemented a disk module for every node based on the single MAXTOR XT-8760S disk in our hypercube.

Since the Intel hypercube uses the circuit switching approach, we had to alter our cost model[OL89] used in estimating the cost of a plan. The model[OL89] assumes a packet switching approach. In addition, to simplify the evaluation, attributes are not added with each successive join and sufficient main memory is assumed to be available to guarantee that hash table overflow will not occur.

6.2 Query Characteristics

We categorize our queries into four groups based on the number of tuples per relation (relation cardinality), the relative locations of the relations, and the join selectivity factors:

1. All relation cardinalities are uniformly distributed between 100 and 625 so that every relation is stored on only one node. All join selectivity factors are uniformly distributed between 10^{-4} and 10^{-3}.
2. All relation cardinalities are uniformly distributed between 5001 and 10,000 so that every relation is stored on the entire 4-cube. All join selectivity factors are uniformly distributed between 10^{-5} and 10^{-4}.
3. All relation cardinalities are uniformly distributed between 100 and 10,000. All join selectivity factors are uniformly distributed between 10^{-4} and 10^{-3}.

6.3 Experimental Results

In order to compare the average performance of the two heuristics, we use the *scaled cost* instead of the real cost measured in seconds. The scaled cost is the ratio of the cost of the best plan produced by a heuristic for a given query, to the minimum cost of plans produced by the two heuristics for the same query. The reason we do not compare the actual timing is that the order of the costs for different queries can vary greatly and a scaled cost provides an objective measure of the relative advantage of a specific heuristic. A scaled cost of 1.0 means that this plan has the lowest cost among the plans produced by the two heuristics.

The execution costs of these plans on the hypercube are measured and the average scaled costs are compared to see if the result is consistent with the estimate produced by Algorithm 5.1. For simplicity, the execution cost only reflects the duration from the time a node receives a plan, until it finishes performing the entire join plan. This is the cost predicted by Algorithm 5.1.

Optimization of each of the individual plans will take a varying amount of time. The overhead in initiating the optimization process at each of the nodes, and the transfer of the results back to the host are approximately equal. Therefore, we only consider the longest and shortest durations of the optimization algorithms performed by the nodes.

For each table of results, several experiments were run. We compare the average scaled cost of the initial

Table 1: Performance of Queries in Category 1 for 20 Relations.

Cost	Inter-Join-Par.	Intra-Join-Par.
Best Initial	1.00	1.52
Best Optimized	1.00	1.50
Real Execution	1.00	2.20
Total	1.03	1.14

Table 2: Performance of Queries in Category 2 for 20 Relations.

Cost	Inter-Join-Par.	Intra-Join-Par.
Best Initial	3.42	1.00
Best Optimized	3.28	1.00
Real Execution	2.33	1.00
Total	1.13	1.16

plan, the average scaled cost of the optimized plan, the average scaled execution cost, and the scaled total cost which includes both the optimization time and execution time for the two heuristics.

6.3.1 Comparison of Algorithms

Table 1 shows the performance of the three heuristics when all relations are very small and scattered. The Maximum Inter-Join-Parallelism heuristic has the best performance since it enables many joins to be performed in parallel. In addition, for the Maximum Intra-Join-Parallelism heuristic, the longest time spend for query optimization was 93.17 seconds and for the Maximum Inter-Join-Parallelism heuristic it was only 87.34 seconds. The shortest time spend for query optimization was 31.49 seconds for the Maximum Intra-Join-Parallelism heuristic and 45.24 seconds for the Maximum Inter-Join-Parallelism heuristic. By assigning the entire cube to every join, the Maximum Intra-Join-Parallelism heuristic has to spend more time in resolving all the transfer-dependencies in the beginning since all relations have to be re-distributed over the entire cube.

When all relations are very large and stored on the entire cube, the performance of the two heuristics is summarized in Table 2. The Maximum Intra-Join-Parallelism heuristic is superior to the Maximum Inter-Join-Parallelism heuristic. Although the Maximum Inter-Join-Parallelism heuristic provides a higher degree of inter-parallelism, for this type of queries, each join takes a longer time in addition to the overhead in resolving the transfer-dependencies. In addition, for the Maximum Intra-Join-Parallelism heuristic, the longest time spent for query optimization was 94.11 seconds and for the Maximum Inter-Join-Parallelism heuristic it was only 79.77 seconds. The shortest time spend for query optimization was 25.51 seconds for the Maximum Intra-Join-Parallelism heuristic and 40.60 seconds for the Maximum Inter-Join-Parallelism heuristic.

Table 3: Performance of Queries in Category 3a for 20 Relations.

Cost	Inter-Join-Par.	Intra-Join-Par.
Best Initial	2.42	1.00
Best Optimized	4.65	1.00
Real Execution	6.35	1.03
Total	2.15	1.17

Table 3 summarizes the general case where the relation cardinalities are uniformly distributed. In general, the Maximum Intra-Join-Parallelism heuristic has the best initial and optimized costs. With respect to the longest and shortest optimization times, a similar trend appeared as with the previous experiments.

6.3.2 Query Optimization Time

In general, the longest time and the shortest time to reach a local optimum among the different starting solutions generated by different nodes are quite far apart. This makes it possible to improve the performance with the 2PO (Two Phase Optimization) method described in [IK90]. Those nodes that have reached a local optimum earlier can use the current best solution as the input to the second phase, which uses a modified simulated annealing method. This can better utilize the idle nodes and further improve the quality of their solutions.

6.3.3 Query Execution Time

Most of the average scaled costs for the query execution time on the hypercube are shown to be consistent with the estimated plan costs. That is, if a plan produced by a heuristic is shown to have the best estimate, its actual execution cost is most likely to be the best as well.

This is mainly due to the fact that for these two categories, every relation has to be re-distributed over the entire cube. This in turn results in increased link contention, and therefore communication delays.

6.3.4 Total Time

By examining the total time spent in both optimization and execution, we have a better picture of the performance of a heuristic. For category 1 in Table 1, the Maximum Inter-Join-Parallelism heuristic not only has the best execution cost but also the best total time. However in Table 2, the Maximum Inter-Join-Parallelism heuristic has the best total cost although it has the worst execution cost. One possible reason is that for this heuristic, the difference between the time that the earliest and latest node reaches a local optimum is significantly smaller than the other two heuristics; this compensates for the inferior quality of its optimized plan. Another possible explanation is that this heuristic may be able to handle queries whose relations are of similar sizes better than the other two heuristics. From Table 3 we can see that the Maximum Intra-Join-Parallelism heuristic has the best total time in general together with the best execution time.

7 Summary

In this paper, we examine the issue of optimizing large join queries on a hypercube multiprocessor. Two heuristics are proposed to produce an initial plan for a given query. We adapt the iterative local improvement procedure to the query optimization process in a hypercube multiprocessor to improve the plan produced by the application of the two heuristics. Our simulation of the query optimization and the query execution process show that the performance of these heuristics depends on the characteristics of the queries.

Optimizing complex queries in parallel will reduce the bottleneck in the query processor and will also improve the quality of the query execution plan. However, it is important to realize that the overhead can be substantial, and therefore the number of processors used in optimizing a query should depend on each individual query.

References

[DKT90] S. M. Deen, D. N. P. Kannanagara, and M. C. Taylor. Multi-Join on Parallel Processors. In *Proceedings of Second International Symposium on Databases in Parallel and Distributed Systems*, pages 92–102, 1990.

[IK84] T. Ibaraki and T. Kameda. On the Optimal Nesting Order for Computing N-Relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proceedings of ACM SIGMOD-International Conference on Management of Data*, pages 312–321, May 1990.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 128–137, 1986.

[OL89] E. Omiecinski and E. T. Lin. Hash-Based and Index-Based Join Algorithms for Cube and Ring Connected Multicomputers. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):329–343, September 1989.

[SD90] D. Schneider and D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proceedings of the 16th VLDB Conference*, August 1990.

[SG88] A. Swami and A. Gupta. Optimizing Large Join Queries. In *Proceedings of ACM SIGMOD- International Conference on Management of Data*, pages 8–17, September 1988.

[Swa89] A. Swami. Optimizing Large Join Queries : Combining Heuristics and Combinatorial Techniques. In *Proceedings of ACM SIGMOD- International Conference on Management of Data*, pages 367–376, June 1989.