An Effective Parallelization of Execution of Multijoins in Multiprocessor Systems

Xuemin Lin and Simon Fox
Department of Computer Science
The University of Western Australia, Nedlands, WA 6010, Australia
e-mail: {lxue, sfox}@cs.uwa.oz.au

Abstract

In this paper, we study a synchronous execution strategy for parallel join computation in multiprocessor systems. Through a further comprehensive investigation of the processor allocation problem and inter-operator parallelization problem, we present a new algorithm for producing an effective parallelization plan for processing multijoins. Besides theoretical analysis, the efficiency and effectiveness of our new algorithm are supported by our experiments.

Key Words: Databases, Multijoins, Parallel Processing, Processor Allocation, Response Time.

1 Introduction

A relational multijoin is the most expensive operation to be processed in distributed database systems. There are two paradigms for processing a distributed multijoin:

1. distributed computation approach, or

2. parallel execution approach.

The former aims to minimize data transmission cost among remote sites, while the later aims to speed up query response time.

In this paper, we restrict our interests in parallel execution of multijoins in a *multiprocessor* system (either share disks or share everything). The interested readers may refer to [1,11,12,24] for a detailed discussion of a distributed processing approach.

To efficiently compute a multijoin in parallel, two factors should be considered [3,9,22]: 1) *intra-operator* parallelism where several processors may be designated for processing one binary join in parallel, 2) *inter-operator* parallelism where several binary joins may be executed concurrently. Nevertheless, an investigation of these two problems should be based on a good processor allocation strategy. In the light of these three aspects, the problem of finding an optimal parallel execution plan is computationally intractable [18,19].

In recent developments for support of non-standard applications, the use of complex data models and the availability of high-level interfaces lead to the generation of complex queries that may contain larger numbers of joins between relations. Consequently, the development of execution strategies for parallel processing multijoins has attracted a great deal of attention. A number of parallel execution heuristics [2,3,5,6,9,15,17,20,21,22] have been recently proposed.

These currently proposed heuristics can be classified into four kinds of strategies: a) Sequential Parallel Execution [22], b) Segmented Right-Deep Execution [2,15,17], c) Synchronous Execution [3,9], d) Full Parallel Execution [20,21,22]. A discussion about advantages and disadvantages for using those four strategies is outside of the coverage of this paper. The interested readers are recommended to read [22] for a detailed discussion. In this paper, we will restrict our interests in the synchronous parallel execution strategy.

Consider that the problem of finding an optimal synchronous parallel execution plan for multijoins is computationally intractable. [9] provided the first heuristic in synchronous parallel execution, which refines a sequential execution plan produced by system R algorithm [14] in combining with a *horizontally synchronous* execution technique in processor allocation and an exploration of inter-operator parallelism.

Simulation results in [3] suggested a severe problem in the optimally sequential execution tree produced by system R algorithm: the minimum total execution cost of an optimally sequential execution tree is, on average, much larger than the minimum total cost of an optimal execution tree. This will potentially degrade the performance of the algorithm in [9], because of the importance (stated in [3,5,22]) of having an execution tree with small total cost. Further, [3] proposed a two-phase optimization strategy for multi-joins. The first phase determines an execution tree that has the lowest total execution cost and the second phase finds a suitable parallelization for this tree. Particularly, in the first phase a greedy heuristic has been applied to obtaining an execution tree with small total cost; and in the second phase, a *hierarchically synchronous* execution strategy is applied to processor allocation to find a suitable parallelization.

In our investigation, we find that in an execution plan for a multijoin, both of inter-operator parallelism and total execution cost have a great impact on minimizing query response time. Further, the maximization of inter-operator parallelization degree and the minimization of total execution cost are conflicting for minimizing response time: an example will be

1087-4089/96 $5.00 © 1996 IEEE

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:20 from IEEE Xplore. Restrictions apply.

shown in Section 4. Therefore, a good trade-off between inter-operator parallelism and total execution cost must be carried out in building an execution plan.

Consider that the the algorithm in [3] is to first provide an execution tree with the lowest total execution cost, and then to explore a suitable inter-operator parallelization for the tree through developing a good processor allocation algorithm. This may limit an achievement of a good trade-off between inter-operator parallelism and total execution cost.

In this paper, we will propose a new hierarchically synchronous execution algorithm. The proposed algorithm can be divided into two phases as follows. The first phase is the same as that in [3] to obtain an execution tree. In the second phase, we will allow modifications to the execution tree in order to provide a good trade-off between inter-operator parallelism and total execution cost, rather than retaining the execution tree structure in the second phase [3]. Our experiment results suggest that the proposed algorithm greatly improve the performance of the algorithm in [3].

The second phase of our algorithm follows the same refinement schedule as that in [9]. However, new optimization techniques and a better processor allocation algorithm are proposed in our algorithm to be tailored to the inputs which have a different nature to that in [9]. This differs our work with that in [9].

The rest of the paper is organized as follows. In section 2, we introduce the necessary background knowledge. In section 3, our processor allocation algorithm is presented, and then a comparison with the existing allocation algorithms will be made through simulation experiments. Section 4 presents a new hierarchically synchronous execution algorithm, as well as our experiment results. In our experiments, we simulate the performances of both our algorithm and the algorithm in [3]. This is followed by a conclusion.

2 Multijoin Execution and Costs

In this paper, we study only *natural* multijoins [14]. For each relation R_i, $|R_i|$ denotes the cardinality of R_i (i.e, the number of tuples in R_i). $|A|$ denotes the cardinality of the domain of a set A of attributes. A *join graph* (V, E, w, c) is used to specify the relationship among relations in a multijoin, where V is the set of nodes and E is the set of edges. Each node represents a referenced relation in the multijoin. Two nodes are connected by an edge if there exists a set of join attributes between the two relations. w defines the cardinality of each referenced relation, such that $w(R_i) = |R_i|$. c defines the cardinality of a join attribute set between two relations, such that for each edge $e = (R_i, R_j)$, $c(e)$ is the cardinality of the join attribute set between R_i and R_j.

Developing a precise estimation of the result size for a join is a difficult task. In this paper, we use the following simple formula [3,14] to evaluate the cardinality of a multijoin $\bowtie_{i=1}^{n} R_i$ result.

$$|\bowtie_{i=1}^{n} R_i| = \frac{\prod_{i=1}^{m} w(R_i)}{\prod_{j=1}^{k} |c(e_j)|}. \quad (1)$$

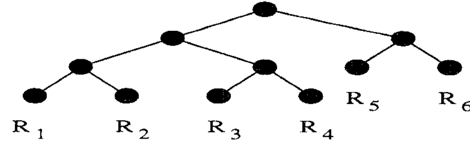


Figure 1: A join execution sequence

Here, $\{e_i : 1 \leq i \leq k\}$ is the set of join attribute sets for all pairs of referenced relations.

An execution plan of a multijoin consists of a *join execution sequence* and a processor allocation for the join execution sequence. A join execution sequence of a multijoin specifies a partial ordering for processing several binary joins to deliver the final multijoin result, and can be represented as a tree (similar to a *relational algebra tree* in [14]). In this paper, a join execution sequence is called an *execution tree*. For instance, according to the execution tree in Figure 1, we should:

compute $\tilde{R} = \tilde{R}_1 \bowtie \tilde{R}_3$ after $\tilde{R}_1 = R_1 \bowtie R_2$ and $\tilde{R}_3 = R_3 \bowtie R_4$, and compute $\tilde{R} \bowtie \tilde{R}_5$ after $\tilde{R}_5 = R_5 \bowtie R_6$ and $\tilde{R} = \tilde{R}_1 \bowtie \tilde{R}_3$.

Using either sort-merge join approach or hash join approach, the cost (time) for computing a join $R_i \bowtie R_j$ can be approximately represented as [3,18]:

$$a|R_i| + b|R_j| + c|R_i \bowtie R_j|, \quad (2)$$

for a single processor. Here, a, b, and c are determined by the path length of the system in processing and joining tuples [23,3,18]. In a multiprocessor system, the cost (time) for computing $R_i \bowtie R_j$ by N processors can be also approximately represented [3,18] as

$$\frac{a|R_i| + b|R_j| + c|R_i \bowtie R_j|}{N} + dN, \quad (3)$$

where d is determined by the inter-processing communication protocol. Note that in the presence of *data skew* [23] in parallel execution of a binary join, (3) may have to be modified. However, in case that either there is no data skew, or data skew problem has been solved, (3) is still valid (we apply this assumption in the rest of the paper).

3 A New Processor Allocation Algorithm

Algorithm design of processor allocation to a given execution tree aims to minimize response time. This problem is well-known for its NP-hardness in distributed jobs scheduling [13].

Consider that in most real applications, the number of processors in a multiprocessor system is usually larger than half the number of referenced relations in a multijoin. This assumption was adopted in the previous research [3,9]. We also use this assumption in the paper. Further, to avoid a complex search space, two synchronization standards, *horizontal* synchronization

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:20 from IEEE Xplore. Restrictions apply.

[9] and *hierarchical* synchronization [3], have been proposed. In horizontal synchronization, we

> iteratively compute in parallel all currently *executable* joins from the bottom of an execution tree, such that at each level, each join computation can be finished at the same time.

On the other hand, in hierarchical synchronization:

> with respect to each node t in an execution tree T, say that t_1 and t_2 are two sub-trees of T with t as their parents, try to finish in parallel the computation of the operations in t_1 at the same time as that in t_2.

In these two synchronization standards, an assumption is made that within the same synchronization level each processor can belong only to one binary join, and processors can be released for use in the next synchronization level only after the computation in the current level has been completely finished.

Further, in hierarchical synchronization, synchronization levels are assigned such that in an execution tree, only siblings are in the same synchronization level. The processors owned by a node can be used only by its children, and then by its desecendants.

A bottom-up processor allocation heuristic is presented in [9] according to horizontal synchronization. Later, a top-down processor allocation heuristic is given in [3] using hierarchical synchronization. In the following, we propose another processor allocation algorithm regarding hierarchical synchronization.

Note that with respect to (3), it is not always true that the response time for parallel computation of a join $R_i \bowtie R_j$ can be reduced along with the increment of the number of processors. In fact, the relationship between the number of processors and the response time can be illustrated in Figure 2. Figure

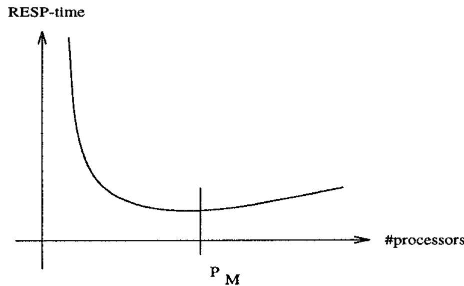


Figure 2: Response time VS processor number

2 shows that the response time is decreasing as the number of processors increases before the point P_M. After the point P_M, the response time is increasing as the number of processors increases. This is caused by inter-processor communication overhead. We call P_M the *minimum time point* P_M to compute $R_i \bowtie R_j$ in parallel. The minimum time point is the number of processors which leads to the minimum value of (3).

It can be immediately obtained, using basic calculus, that P_M will be an integer in the following interval

$$[\lfloor \sqrt{\frac{a|R_i| + b|R_j| + c|R_i \bowtie R_j|}{d}} \rfloor, \lceil \sqrt{\frac{a|R_i| + b|R_j| + c|R_i \bowtie R_j|}{d}} \rceil].$$

Our processor allocation algorithm, HPAA (stands for hierarchical processor allocation algorithm), consists of two steps for a given execution tree:

Step 1: Initialization.

- Assign one processor to each internal node v with two children that are both leaves.
- Then, for each other internal node v, add up the processors which are assigned to the subtree with v as its root. Assign these processors to v. (Note that the number of the processors assigned to v may be, sometimes, larger than its minimum time point: in this case the parallel computation of the join in v needs only to use the minimum time point.)

Go to Step 2.

Step 2: Refinement of the initialization. We assign the remaining processors one by one to reduce the response time:

- With respect to the current processor allocation and (3), find a path l_p from the top to a leaf, such that

  each node v on l_p chooses its child with the longer computation time to be included in l_p.

- If all nodes on l_p have already reached their minimum time points, the algorithm stops and return the current processor allocation. Otherwise, propagationally assign a processor to all nodes on l_p.

The algorithm runs in time $O(Nl + n)$ where N is the number of processors, n is the number of referenced relations in a multijoin, and l is the length of an execution tree ($\log n \leq l \leq n$).

We have simulated the performances of the processor allocation algorithms BU [9], TD [3], and our HPAA. Our experiment results suggest that HPAA outperforms the algorithms BU and TD (the detailed experiment reports may be found in [10]). Especially as both of numbers of the processors and the referenced relations are increasing the performance of HPAA is getting better and better than those of BU and TD.

4 Generating Parallel Execution Plan

As mentioned in Section 2, an execution plan for processing a multijoin consists of an execution tree and a processor allocation for the tree. The processor allocation problem has been discussed in last section. Consequently, we concluded that the algorithm HPAA is the best choice. In this section, we will discuss how to produce a good execution tree.

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:20 from IEEE Xplore. Restrictions apply.

The total cost of an execution tree is defined as the total computation cost by a single processor for the tree. It can be immediately calculated according to (2). As noted by several researchers [22], an execution tree with the lowest total execution cost may favour the minimization of response time for parallel execution. Although finding an execution tree with the lowest total cost is known NP-hard [18], a greedy heuristic in [3] has been shown working well.

We believe that both the minimization of total cost and the maximization of the inter-operator parallelization degree are very important in evaluation of the goodness of an execution tree. However, sometimes they are conflicting. For instance, two execution trees

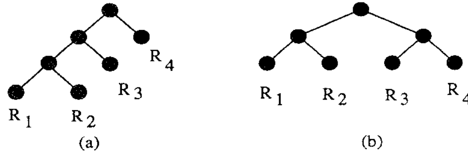


Figure 3: Total cost vs parallelism

for processing a multijoin $\bowtie_{i=1}^{4} R_i$ are illustrated in Figure 3. Suppose that in $\bowtie_{i=1}^{4} R_i$, the following parameters are applied.

Example 1. $|R_i| = 100$ for $1 \leq i \leq 4$. A_1 is the join attribute between R_1 and R_2, A_2 is the join attribute between R_2 and R_3, and A_3 is the join attribute between R_3 and R_4. $|A_1| = |A_2| = |A_3| = 200$. There are 8 processors. Regarding (3), $a = b = c = 1$, $d = 20$. □

One can immediately verify that the execution tree in Figure 3(a) has the minimum total cost, while the execution tree in Figure 3(b) has the maximal inter-operator parallelization degree, but a larger total cost. In this case, the three processor allocation algorithms BU, TD and HPAA yield the same result: the response time for the execution tree in Figure 3(b) is smaller than that in Figure 3(a). However, if we change some of the parameters in Example 1 into:

Example 2. $|R_i| = 8000$ for $1 \leq i \leq 4$, and $|A_i| = 16000$ for $1 \leq i \leq 3$. □

In Example 2, the execution tree in Figure 3(a) leads to a smaller total cost than that in Figure 3(b), as well as a smaller response time.

Unlike the greedy heuristic in [3] concentrating only on reducing total cost, our algorithm for finding an execution tree will be based on a trade-off of minimizing total cost and maximizing inter-operator parallelization degree. Further, it should be clear that such a trade-off must be made according to a good processor allocation algorithm. Thus, we will apply HPAA. Furthermore, our algorithm will use the greedy algorithm GD in [3] as one component. Next, we will present our algorithm. First, let us review the greedy heuristic GD.

4.1 A Greedy Heuristic

The greedy heuristic GD suggests to build an execution tree by iteratively selecting a binary join with the minimum cost according to some measurement. Each time after choosing a binary join, two referenced relations will be merged into the intermediate relation - the join result.

A number of cost measurements may be used in algorithm GD when choosing a join: 1) computation cost of the join, 2) join result size, 3) total size of the referenced relations of the join, and 4) difference between the join result size and its referenced relation sizes.

A performance study in [3] suggested that the measurement 2) is the best choice. In this paper, we will use this measurement when applying the algorithm GD. Hence, the algorithm GD can be precisely described as follows.

Algorithm GD:
Input: a join graph G.
Output: an execution tree T.

$T := \emptyset$;
while G has more than one vertex **do**
{ find a pair of R_i and R_j with
the minimal $|R_i \bowtie R_j|$ in G;
$T := T \cup \{R_i \bowtie R_j\}$;
Merge R_i and R_j into $\tilde{R}_i$ in G; }

In the Examples 1 and 2, the execution tree in Figure 3(a) is an output of the algorithm GD. Note the algorithm GD runs in time $O(nm)$ where n is the number of referenced relations, and m is the number of edges in a join graph.

4.2 A New Algorithm for Effective Execution Parallelization

From a given join graph $G = (V, E, w, c)$, we can derive a *join cardinality graph* $GCA = (V, E, \tau)$ where for each edge (R_i, R_j), $\tau(R_i, R_j) = |R_i \bowtie R_j|$.

The algorithm GD can be viewed as iteratively choosing an edge e (a join) from a GCA with the minimum value of $\tau(e)$. Thus, it does not necessarily guarantee a good inter-operator parallelization degree; for instance, Figure 3(a). However, this drawback can be overcome if we allow a choice of several disjoint edges (joins) each time whenever it is necessary. This is the basic idea of our algorithm NEW. Note that a set of disjoint edges in a graph is a *matching* [8] in the graph.

The algorithm NEW suggests to iteratively choose the "best" matching instead of the best edge from GCA. Particularly, the algorithm NEW works as flows in each iteration, for choosing a suitable matching from the current GCA:

Step 1: For $1 \leq k \leq l$ where l is the maximal cardinality of a matching in GCA, we choose a matching with the minimum sum of edge weights from all matchings in GCA with k edges. Go to Step 2.

Step 2: For each k, once such k edges are chosen, we merge these k pairs of nodes into k intermediate results in the join graph. Then, run the algorithm GD on the resulted join graph G_k to produce an execution tree T_k. Using the processor allocation

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:20 from IEEE Xplore. Restrictions apply.

algorithm HPAA, we can calculate the response time t_k of each T_k for $1 \leq k \leq l$. Go to Step 3.

Step 3: Comparing those t_k for $1 \leq k \leq l$, we choose a j such that t_j is minimized. Then, we replace G by G_j; and GCA is the derived cardinality graph of G_j. We enter into next iteration until GCA has only one node. (Note that if in the next iterations j is always chosen to 1, T_j will be the final output. Otherwise, T_j may be refined in the next several iterations.)

It is clear that if in each iteration j is always chosen as 1 then the algorithm NEW is equivalent to the algorithm GD. For instance, applying the algorithm NEW to Example 2, the output is Figure 3(a). However, the output of the algorithm NEW will be Figure 3(b) for Example 1.

A precise description of the algorithm NEW is as follows.

Algorithm NEW
Input: G (a join graph) and GCA (the derived cardinality graph).
Output: an execution tree TA with a processor allocation.

$T := \emptyset$ (an execution tree); $TA := \emptyset$; $n := |V|$;
Initialization:
for $k = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do** $\{T_k := \emptyset;\ TA_k := \emptyset;\ t_k := \infty\}$
while $n > 1$ **do**
{ **for** $k = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
{ Run Initialization;
choose the minimal weighted matching π_k with k edges;
if such matching exists **then**
{ $T_{k,1} := \{R_i \bowtie R_j : (R_i, R_j) \in \pi_k\}$;
merge R_i and R_j into $\tilde{R}_i$ in G for each $(R_i, R_j) \in \pi_k$;
$w(\tilde{R}_i)$ is assinged to $|R_i \bowtie R_j|$;
(the resulted graph by the above modification of G is denoted by G_k.)
Run greedy algorithm GD on G_k to output $T_{k,2}$;
$T_k := T \cup T_{k,1} \cup T_{k,2}$;
Run HPAA on T_k: obtain TA_k and t_k }
}
choose a TA_k with the minimum t_k;
$T := T \cup \{R_i \bowtie R_j : (R_i, R_j) \in \pi_k\}$;
$TA := TA_k$;
$G := G_k$;
derive GCA from G using (2);
$n := n - 2k$
}

In algorithm NEW, it is clear that every step runs in polynomial time, except that the computational complexity of the step for finding a minimum weighted matching with k edges is unclear. In the next subsection, we will show that it can also be done in polynomial time.

4.3 Minimum Weighted Matching with a Given Cardinality Solvable in Cubic Time

In this section, we show the following problem can be solved in polynomial time.

Minimum k-Matching Problem (MKMP):
Given a weighted graph $G = (V, E, w)$, find a matching with k edges such that the sum of the weights over all matchings with k edges is minimized.

Here we suppose that k is smaller than the maximal cardinality of a matching in G. In the literature [8], the following Weighted Matching Problem can be solved in polynomial time.

Weighted Matching Problem (WMP):
Given a weighted graph $G = (V, E, w)$, find a matching such that the sum of the weights of the edges in the matching is maximized.

Particularly, the WMP problem can be solved in $O(|V|^3)$ [8]. Next we show that MKMP problem can be translated to WMP, and then solved by the algorithm for WMP.

For each instance $G = (V, E, w)$ and k in MKMP, we can construct an instance $G' = (V', E'w')$ in WMP as follows:

- Add $|V| - 2k$ new vertices $\{u_i : 1 \leq i \leq |V| - 2k\}$, that is, $V' = V \cup \{u_i : 1 \leq i \leq |V| - 2k\}$.
- For each pair of vertices v_i and v_j which are bother in V, $(v_i, v_j) \in E'$ if and only if $(v_i, v_j) \in E$. Also, $w'((v_i, v_j)) = N - w((v_i, v_j))$ where

$$N = \sum_{(v_i, v_j) \in E} w((v_i, v_j)) + 1.$$

- For each new vertex $u_i \in \{u_i : 1 \leq i \leq |V| - 2k\}$ and old vertex $v_j \in E$, $(u_i, v_j) \in E'$ and $w'(u_i, v_j) = M$, where $M = (|E| + 1)N$.

The following two theorems can be obtained [10].

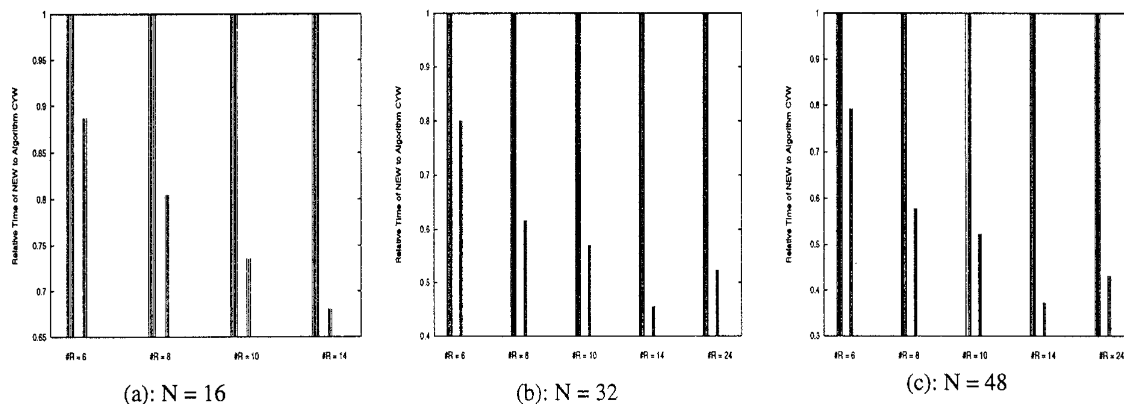
Theorem 1 *Suppose that an instance G and k in MKMP is given where G has n vertices. G' is derived from G as above. Then, in any solution to WMP for G': 1) there must be $n - 2k$ edges from $\{(u_i, v_j) : 1 \leq i \leq n - 2k, 1 \leq j \leq |V|\}$, and 2) the other edges are from E.*

Theorem 2 *Suppose that a weighted graph $G = (V, E, w)$ and $k \leq \frac{|V|}{2}$ are given, and $|V| = n$. G' is derived from G as above. Then, there exists a solution to MKMP if and only if there exists a solution S' to WMP with respect to G' such that $|S'| = n - k$.*

As consequences of Theorems 1 and 2,

- if the cardinality of the solution S' to WMP with respect to G' is not equal to $n - k$, there is no matching in G with k edges;

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:20 from IEEE Xplore. Restrictions apply.



(a): N = 16 (b): N = 32 (c): N = 48

Figure 4: Response times of the algorithm NEW relative to CYW

- otherwise we need only to choose those k edges from S' which are in G as the solution to MKMP;
- MKMP can also be solved in time $O(|V|^3)$.

4.4 Complexity and Experiment Results

Suppose that there are N processors, a multijoin has n referenced relations, and the join graph has m edges. The algorithm NEW has $O(n)$ iterations. In each iteration, we need to consider $O(n)$ cases. In each case, the algorithm HPAA runs in time $O(Nn)$, the MKMP is solvable in time $O(n^3)$, and the algorithm GD runs in time $O(nm)$. Thus, the algorithm NEW runs in time $O(n^3N + n^3m + n^5)$. Note that $m \leq n(n-1)$. Hence, the algorithm NEW runs in time $O(n^5 + n^3N)$.

The algorithm CYW in [3] suggests to use the algorithm GD first to produce an execution plan, and then apply the algorithm TD to the resulted execution plan. We have simulated the performances of our algorithm NEW and the algorithm CYW. In our experiments, for each pair of number of referenced relations and number of processors we randomly generate 1000 examples (multijoins). Then, for each multijoin we record the ratio of the response time using the algorithm NEW to that using the algorithm CYW. For each given number $(16, 24, 48)$ of processors, we output the average ratio over 1000 random multijoins on a given number of referenced relations. In our implementation,

- we choose $a = b = c = 1$ and $d = 20$ for simplicity;
- each relation cardinality is randomly chosen from 500 tuples to 4000 tuples;
- the edge set in a join graph is also randomly chosen;
- the cardinality of a join attribute set $(c(e))$ between R_i and R_j is also randomly chosen from 50 to $\max\{|R_i|, |R_j|\}$.

The experiment results are reported in Figure 4(a)-(c). They showed that our algorithm NEW greatly out-performs the algorithm CYW.

5 Conclusion

In this paper, we study the synchronous strategy for processing multijoins in mutiprocessor systems. We first proposed a new processor allocation algorithm HPAA. Our experiments showed that the algorithm HPAA out-perms the existing processor allocation algorithms. Then, by exploring a suitable inter-operator parallelization in multijoins, we present a new synchronous algorithm NEW for parallel computation of multijoins in multiprocessor systems. The algorithm NEW makes the use of the algorithm HPAA as one component. Our experiments also showed that the algorithm NEW out-performs the latest synchronous algorithm CYW.

Although in this paper we consider only the case that either there is no data skew or the data skew problem has been solved, the results in this paper could be easily modified to cover a general case.

Acknowledgement

This project was partially supported by IRG at UWA. The authors would like to thank DR. C. McDonard and DR. C. P. Tsang for many helps in setting up a pragramming environment.

References

[1] M.-S. Chen and P. S. Yu, Interleaving a Join Sequence with Semijoins in Distributed Query Processing, *IEEE Transactions on Parallel and Distributed Systems*, 3(5), 611-621, 1992.

[2] M.-S. Chen, M. L. Lo, P. S. Yu, and H. C. Young, Using Segmented Right-Deep Trees for the Execution of Pipelined Hashjoins, *18th VLDB*, 15-26, 1992.

[3] M.-S. Chen, P. S. Yu, and K.-L. Wu, Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries, *IEEE Conference on Data Engineering*, 58-67, 1992.

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:20 from IEEE Xplore. Restrictions apply.

[4] D. J. DeWitt and R. Gerber, Multiprocessor Hash-based Join Algorithms, *VLDB*, 151-162, 1985.

[5] W. Hong and M. Stonebraker, Optimization of Parallel Query Execution Plans in XPRS, *1st PDIS Conference*, 1991.

[6] H. I. Hsaio, M. S. Chen, and P. S. Yu, On Parallel Execution of Multiple Pipelined Hash-Joins, *ACM-SIGMOD*, 185-196, 1994.

[7] S. Ganguly, W. Hasan and R. Krishnamurthy, Query Optimization for Parallel Execution, *SIGMOD Record*, 21(2), 9-18, 1992.

[8] E. L. Lawler, *Combinatorial Optimization Networks and Matroids*, Holt, Rinehart and Winston, 1976.

[9] H. Lu, M. C. Shan, and K. L. Tan, Optimization of Multi-Way Join Queries for Parallel Execution, *Proceedings of VLDB 91*, 549-560, 1991.

[10] X. Lin and S. Fox, An Effective Parallelization Scheme for Multijoin Computation, Research Report, CS, University of Western Australia, 1996.

[11] X. Lin and M. Orlowska, An Efficient Processing of a Chain Join with the Minimum Communication Cost in Distributed Database Systems, *Journal of Distributed and Parallel Databases*, 3(1), 69-84, 1995.

[12] X. Lin, M. Orlowska, and X. Zhou, Using Parallel Semi-Join Reduction to Minimize Distributed Query Response Time, *IEEE International Conference on Algorithms and Architectures for Parallel Processing*, IEEE CS Press, 517-526, 1995.

[13] S. Lam and R. Seth, Worst Case Analysis of Two Scheduling Algorithms, *SIAM Journal on Computing*, 6(3), 518-537, 1977.

[14] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, 1991.

[15] D. A. Schneider, *Complex Query Processing in Multiprocessor Database Machines*, PHD Thesis, Computer Science Technical Report 965, University of Wisconsin, Madison, 1990.

[16] D. A. Schneider and D. J. DeWitt, A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment, *SIGMOD*, 110-121, 1989.

[17] D. A. Schneider and D. J. DeWitt, Tradeoffs in processing complex join queries via hashing in multiprocessor database machines, *16th VLDB*, 469-480, 1990.

[18] D. Shasha and T. L. Wang, Optimizing Equijoin Queries in Distributed Databases Where Relations are Hash Partitioned, *ACM Transactions on Database Systems*, 16(2), 279-308, 1991.

[19] A. Swami, Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques, *Proceedings of SIGMOD 89*, 367-376, 1989.

[20] A. N. Wilschut and P. M. G. Apers, Dataflow Query Execution in a Parallel, Main-Memory Environment, *PDIS Conference*, 68-77, 1991.

[21] A. N. Wilschut, P. M. G. Apers, and J. Flokstra, Parallel Query Execution in PRISMA/DB, *PRISMA Workshop on Parallel Database Systems*, 1991.

[22] A. N. Wilschut, J. Flokstra, and P. M. G. Apers, Parallel Evaluation of Multi-Join Queries, *ACM-SIGMOD*, 115-126, 1995.

[23] J. L. Wolf, D. M. Dias, and P. S. Yu, An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew, *Second International Symp. on Databases in Parallel and Distributed Systems*, 1990.

[24] C. T. Yu and C. C. Chang, Distributed Query Processing, *ACM Computing Surveys*, 16(4), 1984.

[25] P. S. Yu, M.-S. Chen, H. Heiss, and S. H. Lee, On Workload Characterization of Relational Database Environments, IEEE Transactions on Software Engineering, 18(1), 1992.

Authorized licensed use limited to: UNSW Library. Downloaded on August 10, 2009 at 07:20 from IEEE Xplore. Restrictions apply.