Optimization of Parallel Execution for Multi-Join Queries

Ming-Syan Chen, *Senior Member, IEEE*,
Philip S. Yu, *Fellow, IEEE*, and Kun-Lung Wu, *Member, IEEE*

Abstract—In this paper, we study the subject of exploiting interoperator parallelism to optimize the execution of multi-join queries. Specifically, we focus on two major issues: 1) scheduling the execution sequence of multiple joins within a query, and 2) determining the number of processors to be allocated for the execution of each join operation obtained in 1). For the first issue, we propose and evaluate by simulation several methods to determine the general join sequences, or bushy trees. Despite their simplicity, the heuristics proposed can lead to the general join sequences that significantly outperform the optimal sequential join sequence. The quality of the join sequences obtained by the proposed heuristics is shown to be fairly close to that of the optimal one. For the second issue, it is shown that the processor allocation for exploiting interoperator parallelism is subject to more constraints—such as execution dependency and system fragmentation—than those in the study of intraoperator parallelism for a single join. The concept of synchronous execution time is proposed to alleviate these constraints. Several heuristics to deal with the processor allocation, categorized by bottom-up and top-down approaches, are derived and are evaluated by simulation. The relationship between issues 1) and 2) is explored. Among all the schemes evaluated, the two-step approach proposed, which first applies the join sequence heuristic to build a bushy tree as if under a single processor system, and then, in light of the concept of synchronous execution time, allocates processors to execute each join in the bushy tree in a top-down manner, emerges as the best solution to minimize the query execution time.

Index Terms—Bushy trees, synchronous execution time, multi-join query, execution dependency, system fragmentation.

1 Introduction

There has been a growing interest in applying general purpose parallel machines to database applications [7], [8], [12], [19], [34], [43], [50]. Several research systems have been developed to explore this trend, including GAMMA [16], XPRS [49], DBS3 [4], GRACE [31], and BUBBA [6]. Relational databases have a certain natural affinity to parallelism. Relational operations are set oriented and this provides the query optimizer lots of flexibility in selecting the parallelizable access path. In relational database systems, joins are the most expensive operations to execute, especially with the increases in database size and query complexity [11], [27], [30], [39], [53]. For future database management, parallelism has been recognized as a solution for the efficient execution of multi-join queries [1], [17], [18], [25], [36], [42], [52], [54], [55].

As pointed out in [46], the methods to exploit parallelism in the execution of database operations in a multiprocessor system can be divided into three categories. First, parallelism can occur in each operator within a query in such a way that several processors can work, in parallel, on a single database operation. This form of parallelism is termed intraoperator parallelism and has been studied extensively. Various solutions for exploiting *intraoperator* parallelism in multiprocessor database systems have been reported in the literature. Several algorithms were proposed for parallel execution of two-way joins in multiprocessor systems [15], [38], [44], [45]. Some researchers further concerned themselves with multiprocessors of particular architectures such as rings and hypercubes [3], [40]. The effect of data skew on the performance of parallel joins has also been analyzed in [14], [32], [51]. The second form of parallelism is termed *interoperator* parallelism, meaning that several operators within a query can be executed in parallel. Third, parallelism can be achieved by executing several queries simultaneously within a multiprocessor system, which is termed *interquery* parallelism [48]. It can be seen that to exploit the third form of parallelism, one has to resort to the results derived for interoperator parallelism within a query. During the past few years some light has been shed on this issue [13], [21], [22], [37], [42], [46]. As an effort toward this trend, the objective of this paper is to study and improve the execution of multi-join queries, and devise efficient schemes to exploit interoperator parallelism to minimize the query execution time in a multiprocessor-based database system.[1]

Note that different join execution sequences for a query will result in different execution costs [47]. Also, the execution time of a join in a multiprocessor system strongly depends on the number of processors allotted for the execution of that join [32]. For instance, a 40 second execution time of a join on four processors may increase to 60 seconds if only two processors are used. Thus, the subject of ex-

- *M.-S. Chen is with the Electrical Engineering Department, National Taiwan University, Taipei, Taiwan, Republic of China. E-mail: mschen@cc.ee.ntu.edu.tw.*
- *P.S. Yu and K.-L. Wu are with the IBM Thomas J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598. E-mail: {psyu, klwu}@watson.ibm.com.*

Manuscript received: Apr. 25, 1995.
For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K96033.

1. The execution time of a query in this paper, similar to that in most related work, means the response time to complete the query, rather than the total execution time of all processors.

1041-4347/96$05.00 ©1996 IEEE

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

ploiting interoperator parallelism for the execution of a multi-join query comprises two major issues:

1) join sequence scheduling, or query plan generation, i.e., scheduling the execution sequence of joins in the query, and
2) processor allocation, i.e., determining the number of processors for each join obtained in 1) so that the execution time required for the query can be minimized.

Clearly, the join method affects the optimization procedure to exploit parallelism. Under hash joins, we have the opportunity of using pipelining to improve the performance [15], [21], whereas such an opportunity is not available when a join method like the sort-merge join is employed. Note that pipelining causes the effects on join sequence scheduling and processor allocation to be entangled, and the resulting cost model of each join and the criteria for processor allocation in the presence of pipelining will thus be intrinsically different from those developed without using pipelining. As a result, join methods without and with pipelining have to be dealt with separately for best optimization results. In this paper, we shall focus on the join methods without pipelining, such as the sort-merge join that is in fact the most prevalent join method in existing database softwares, and develop a specific solution procedure. Readers interested in optimization on pipelining multiple joins, which calls for a different procedure due to its different problem formulation, are referred to [9], [26], [35], [55]

First, for the issue of join sequence scheduling, we develop and evaluate by simulation several heuristics to determine the join sequence for a multi-join query with the focus on minimizing the total amount of work required.[2] Specifically, we investigate two sorts of join sequences, namely, *sequential join sequences* and *general join sequences*. A join sequence in which the resulting relation of an intermediate join can only be used in the next join is termed a sequential join sequence. An example of a sequential join sequence can be found in Fig. 1a where every nonleaf node (internal node) represents the resulting relation from joining its child nodes. A join sequence in which the resulting relation of a join is not required to be only used in the next join is termed a general join sequence. For example, the sequence of joins specified by the join sequence tree in Fig. 1b is a general join sequence. Such an execution tree of a general join sequence is called a *bushy tree* [22], or *composite inners* [41].

Note that the bushy tree join sequences did not attract as much attention as sequential ones in the literature. As a matter of fact, it was generally deemed sufficient, by many researchers, to explore only sequential join sequences for desired performance in the last decade. This can be in part explained by the reasons that in the past the power/size of a multiprocessor system was limited, and that the query structure used to be too simple to require further parallelizing as a bushy tree. It is noted, however, that these two limiting factors have been phased out by the rapid increase in the capacity of multiprocessors and the trend for queries to become more complicated nowadays, thereby justifying the necessity of exploiting bushy trees. Consequently, we propose and evaluate by simulation several join sequence heuristics in this paper to efficiently determine general join sequences of good efficiency. As can be seen from our results, the heuristics proposed, despite their simplicity, result in general join sequences which significantly outperform the optimal sequential join sequence. This is especially true for complex queries. More importantly, it is shown that the quality of the general join sequences obtained by the proposed heuristics is fairly close to that of the optimal general join sequence, meaning that by employing appropriate heuristics we can avoid excessive search cost and obtain join sequences with very high quality.

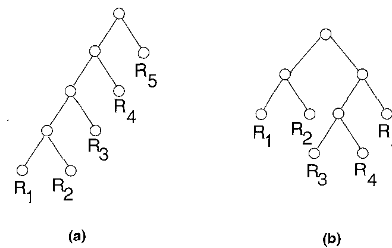


Fig. 1. Illustration of different join sequences.

Next, we explore the issue of processor allocation for join operations. In the study of intraoperator parallelism, the objective is usually to determine the processor allocation which achieves the minimum execution time of a single join. Such a selection is referred to as *operational point selection* in this paper. However, in exploiting interoperator parallelism, we, in contrast, are dealing with the execution of a complex query with multiple joins where different joins are allowed to be executed in parallel in different clusters of processors. As will be seen later, minimizing the execution time of a multi-join query, in addition to the operational point selection as in the study of intraoperator parallelism, requires more factors, such as *execution dependency* and *system fragmentation*, to be considered. Execution dependency means that some joins cannot be performed until their operands generated by prior joins are available. Also, after a sequence of processor allocation and release, there might be a few processors left idle since they do not form a cluster large enough to execute any remaining join efficiently. This phenomenon is termed system fragmentation [11]. Clearly, execution dependency and system fragmentation, as well as the operational point selection, have to be taken into account for a better processor allocation strategy, thus complicating the minimization procedure for the query execution time. To deal with this problem, we propose and evaluate several heuristics to determine the number of processors for each join. The processor allocation heuristics proposed can be divided into two categories:

1) the bottom-up approach, where the number of processors allocated to each internal node (join) in a bushy tree is determined as the bushy tree is being built bottom-up, and

2. Note that "minimizing the total amount of work in a join sequence" is only for join sequence scheduling, and should not be confused with the overall objective to minimize the query execution time.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

2) the top-down approach, which, in light of the concept of *synchronous execution time*, determines the processor allocation based on a given bushy tree.

The concept of synchronous execution time is employed to deal with processor allocation so that input relations for each join can be made available approximately the same time. It is shown that the concept of synchronous execution time will significantly alleviate execution dependency and system fragmentation, and hence improve the query execution time.

Note that to conduct performance study on the execution of a multi-join query, the schemes on join sequence scheduling and processor allocation are integrated to form a final scheduler. As shown by our simulation results, the join sequence scheduling is in general the dominating factor for the query execution time whereas processor allocation becomes significant as the number of processors and query complexity increase. Thus, as confirmed by our simulation, among all the schemes investigated, the two-step approach of first applying the join sequence heuristics to build a bushy tree as if under a single processor system, and then determining the processor allocation in light of the concept of synchronous execution time for the bushy tree built emerges as the best solution to minimize the query execution time.

This paper is organized as follows. The notation and assumptions used are given in Section 2. In Section 3, we study several join sequence heuristics. Sequential and general join sequences are addressed in Sections 3.1 and 3.2, respectively, and simulation results are presented in Section 3.3. Processor allocation is dealt with in Section 4. Bottom-up and top-down approaches are respectively developed in Sections 4.1 and 4.2, followed by their simulation results in Section 4.3. The paper concludes with Section 5.

2 Preliminaries

In this study, we assume that a query is of the form of conjunctions of equi-join predicates and all attributes are renamed in such a way that two join attributes have the same attribute name if and only if they have a join predicate between them. A join query graph can be denoted by a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each node in a join query graph represents a relation. Two nodes are connected by an edge if there exists a join predicate on some attribute of the two corresponding relations. An edge between R_i and R_j in a query graph is said to be *shrunken* if that edge is removed from the graph and R_i and R_j are merged together. Notice that when a join operation between the two relations R_i and R_j in a given query graph is carried out, we can obtain the resulting query graph by shrinking the edges between R_i and R_j and merging the two relations together to represent the resulting relation from the join operation.

We use $|R_i|$ to denote the cardinality of a relation R_i and $|A|$ to denote the cardinality of the domain of an attribute A. The notation (R_i, R_j) is used to mean the join between R_i and R_j, and $R_i \bowtie R_j$ denotes the resulting relation of (R_i, R_j). For notational simplicity, we denote the execution tree in Fig. 1a as $((((R_1, R_2), R_3), R_4), R_5)$, and that in Fig. 1b as $((R_1, R_2), ((R_3, R_4), R_5))$. As in most prior work on the execution of database operations in multiprocessor systems, we assume that the execution time incurred is the primary cost measure for the processing of database operations. In that sense, it has been shown that the join is the most expensive operation and that the cost of executing a join operation can mainly be expressed in terms of the cardinalities of the relations involved. Also, we focus on the execution of complex queries, which becomes increasingly important nowadays in real databases due to the use of views [53]. As mentioned earlier, different join methods and different multiprocessor systems will result in different execution costs for a join, and we shall address the join methods without utilizing pipelining, such as the sort-merge join, in this paper. The effect of pipelining is examined in [9], [35]. It is worth mentioning that due to its stable performance, the sort-merge join is the most prevalent join method used in some database products to handle both equal and nonequal join queries [2], [24]. The hash-based join, though having good average performance, suffers from the problem of hash bucket overflow and is thus avoided by many commercial database products. The architecture assumed in this study is a multiprocessor-based database system with shared disks and memory [5]. The cost function of joining R_i and R_j can then be expressed by $|R_i| + |R_j| + |R_i \bowtie R_j|$, which is general and reasonable for joining large relations by the sort-merge join [28], [29], [51]. Also, all the processors are assumed to be identical and the amount of memory available to execute a join is assumed to be in proportion to the number of processors involved.

To facilitate our discussion, the performance of a scheduling scheme is assessed by the average execution time of plans generated by this scheduling scheme. The efficiency of the join sequence, measured by its execution on a single processor system, is termed *join sequence efficiency*, and the effectiveness of processor allocation, determined by the speedup achieved over the single processor case, is termed *processor allocation efficiency*. The overall efficiency for dealing with the above two issues then depends on the two factors. Note that to best assess the performance impact of certain factors in a complicated database system, it is generally required to fix some factors, and evaluate only the interesting ones. Similarly to most other work in performance, we adopt the above approach in this paper and concentrate on investigating the effects of join sequence scheduling and processor allocation. It is hence assumed that we have several shared disks and enough disk bandwidth for I/O operations. The effect of resource (i.e., disk/network bandwidth) contention, which is modeled in [18], is assumed to have similar effects on the schemes evaluated, and thus not addressed in this paper. It is noted that even when the disk bandwidth is a bottleneck, join sequence scheduling schemes generating smaller intermediate relations will in general tend to have better performance. In that case, however, the data placement in disks will become an important issue for performance improvement, which is beyond the scope of this paper. Also, we refer readers interested in such issues as execution for a single sort-merge join and the use of indices to improve one join operation to

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

prior work on intraoperator parallelism [28], [29], [51]. Optimization on these issues is system dependent, and in fact orthogonal to the relative performance among schemes evaluated in this paper. Besides, we assume that the values of attributes are uniformly distributed over all tuples in a relation and that the values of one attribute are independent of those in another. The cardinalities of the resulting relations from join operations can thus be estimated according to the formula in [10], which is given in the Appendix for reference. Note that this assumption is not essential but will simplify our presentation. Also, all tuples are assumed to have the same size. In the presence of certain database characteristics and data skew, we only have to modify the formula for estimating the cardinalities of resulting relations from joins accordingly [20], [23] when applying our join sequence scheduling and processor allocation schemes. Results on the effect of data skew can be found in [27], [51].

3 Determining the Execution Sequence of Joins

In this section, we shall propose and evaluate various join sequence heuristics. Specifically, we focus on sequential join sequences in Section 3.1 and general join sequences, i.e., bushy trees, in Section 3.2. Simulation results by different heuristics are given in Section 3.3. For the objective of showing the effect of a join sequence on the total work incurred, we in this section consider the execution of joins under a single processor system. The join sequence efficiencies of various join sequences are compared with one another. Clearly, our results in this section on improving the join sequence efficiency are applicable to both multiprocessor and single processor systems. The combined effects of join sequence scheduling and processor allocation are discussed in Section 4.

3.1 Schemes for Sequential Join Sequences

First, we investigate the sequential join sequences resulted by the following two methods:

1) the greedy method, denoted by S_{GD}, and
2) the optimal permutation, denoted by S_{OPT}, where S means "sequential join sequence" and the subscripts mean the methods used.

The greedy scheme S_{GD} can be outlined as follows. First, the scheme starts with the join which requires the minimal execution cost. Then, the scheme tries to join the composite with the relation which has the minimal-cost join with the existing composite. The above step is repeated until all joins are finished. It can be seen that the complexity of S_{GD} is $O(|V|^2)$. Moreover, we also investigate the optimal sequential join sequence which can be obtained by the optimal permutation of relations to be joined. It can be seen that the number of different sequential join sequences for a query of n relations is $\frac{n!}{2}$, which is half of the total number of permutations of n objects since the first two relations can be interchanged. To evaluate the optimal sequential join sequence in Section 3.3 where different join sequences are compared by simulation, we implemented scheme S_{OPT} in which the technique of branch and bound is used to avoid exhaustive enumeration and reduce the cost of search. For better readability, the implementation detail of S_{OPT}, which is irrelevant to the quality of the join sequence resulted, is not included in this paper.

To show the resulting join sequences by S_{GD} and S_{OPT}, consider the query in Fig. 2a whose profile is given in Table 1. From the operations of S_{GD} and the formula in the Appendix, it can be seen that the join between R_2 and R_4 is the one with the minimal cost among all joins. After the join (R_2, R_4), the resulting query graph and its profile are given respectively in Fig. 2b and Table 2 where R_2 now represents the resulting composite. Then, it can be verified that R_5 is the relation which will have the minimal-cost join with R_2, and the execution of (R_2, R_5) in Fig. 2b is performed. Following the above procedure, we have the resulting join sequence by S_{GD}, $(((((R_2, R_4), R_5), R_6), R_3), R_1)$ whose total cost is 45,246.43. On the other hand, it can be obtained that for the query in Fig. 2a, the optimal sequential join sequence by S_{OPT} is $(((((R_1, R_3), R_6), R_5), R_2), R_4)$ whose total cost is 36,135.92, which is less than that required by S_{GD}. It is interesting to see that the first join performed by S_{OPT} is (R_1, R_3), rather than (R_2, R_4) which is the first one chosen by S_{GD}.

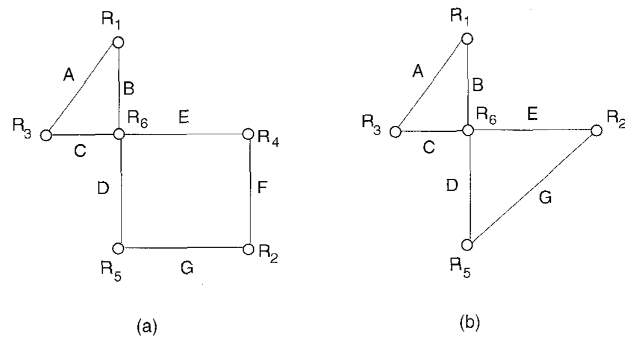


Fig. 2. Two states of an example query graph: (a) the original graph, (b) the resulting graph after joining R_2 and R_4.

TABLE 1
The Profile of the Query in Fig. 2a

relation R_i	R_1	R_2	R_3	R_4	R_5	R_6
cardinality	118	102	106	100	131	120

(a) Cardinalities of relations

attribute	A	B	C	D	E	F	G
cardinality	19	15	17	19	16	15	18

(b) Cardinalities of attributes

TABLE 2
The Profile of the Query in Fig. 2b

relation R_i	R_1	R_2	R_3	R_5	R_6
cardinality	118	680	106	131	120

(a) Cardinalities of relations

attribute	A	B	C	D	E	G
cardinality	19	15	17	19	16	18

(b) Cardinalities of attributes

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

3.2 Schemes for General Join Sequences

It can be seen from the cost function presented in Section 2 that the joins whose operands are of larger sizes usually have higher costs. This observation suggests the following heuristic to explore the general join sequence in order to reduce the total cost incurred. First, we perform the minimal-cost join, and then, from the resulting query, choose the minimal-cost join to perform. This procedure repeats until all joins are finished. Note that this heuristic, though efficient, is greedy in nature in that only "local optimality" is considered, and thus need not lead to a resulting join sequence with the minimal cost. Based on this heuristic, scheme G_{MC}, where G means that the resulting sequence is a general join sequence and the subscript MC stands for "the join with minimal cost," is outlined below. It can be seen that unlike S_{GD}, the resulting composite of a join by G_{MC} need not participate in the next join.

```
Scheme G_MC: /* A scheme to execute the join with
  the minimal cost. */
begin
1.  repeat_until |V| = 1
2.  begin
3.    Choose the join R_i ⋈ R_j from G = (V, E) such
        that cost(R_i, R_j) = min_{∀R_p,R_q∈V} {cost(R_p, R_q)}.
4.    Perform R_i ⋈ R_j.
5.    Merge R_i and R_j to R_min(i,j). Update the profile
        accordingly.
6.  end
end
```

For the example query in Fig. 2a, it can be verified from Fig. 2b and Table 2 that after the first minimal-cost join (R_2, R_4) is performed, the next minimal-cost join to be executed by G_{MC} is (R_5, R_6), rather than (R_2, R_5) as in S_{GD}. The resulting sequence is $(((R_2, R_4), (R_5, R_6)), (R_1, R_3))$ whose total cost is 13,958.62, significantly less than those required by S_{GD} and S_{OPT}. The execution trees resulted by S_{OPT} and G_{MC} are shown in Fig. 3a and 3b, respectively. It can be seen that the complexity of the scheme is $O(|V||E|) < O(|V|^3)$, rather close to $O(|V|^2)$ required by S_{GD}.[3]

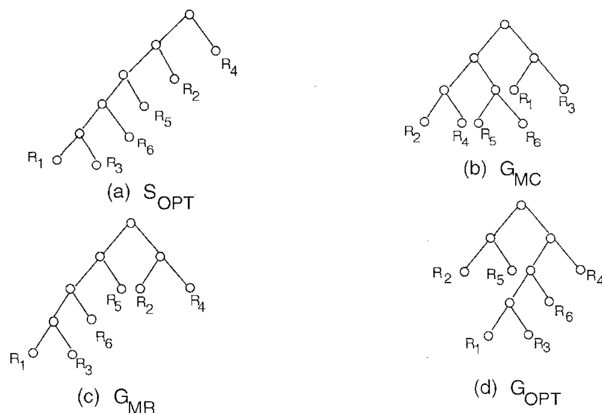


Fig. 3. Different execution trees resulted from different join sequence heuristics.

3. For the simulation in [11], the run times required by the two schemes are almost the same.

Note that in a sequence of joins, the cardinalities of intermediate relations resulting from the early joins affect the costs of joins to be performed later. Since the objective taken is to minimize the total cost required to perform a sequence of joins, one may want to execute the joins which produce smaller resulting relations first. In view of this fact, we develop and evaluate the following heuristic scheme which is a variation of G_{MC}, namely the minimal resulting relation (G_{MR}). Instead of finding the minimal-cost join as in G_{MC}, the scheme G_{MR} searches for the join which results in the minimal resulting relation.[4] Clearly, the heuristic scheme G_{MR} is of the same complexity, $O(|V||E|)$, as scheme G_{MC}. Algorithmic form of G_{MR} is similar to the one of G_{MC}, except that the statement 3 in G_{MC} is changed to 3A below:

3A. (for G_{MR}): Choose the join (R_i, R_j) from $G = (V, E)$ such that $\left|R_i \bowtie R_j\right| = \min_{allR_p,R_q \in V}\left|R_p \bowtie R_q\right|$.

Following G_{MR}, the resulting join sequence for the query in Fig. 2a is $((((R_1, R_3), R_6), R_5), (R_2, R_4))$, whose bushy tree is shown in Fig. 3c. The associated cost is 13,288.38, showing a better join sequence efficiency than the one obtained by G_{MC}. This fact can be further justified by the simulation results in Section 3.3. Moreover, to assess the performance of the heuristics, we implemented scheme G_{OPT} to determine the optimal general join sequence for a multi-join query. Same as in S_{OPT}, we enumerate possible candidate sequences in our implementation of G_{OPT} and employ the technique of branch and bound to prune the search. Using G_{OPT}, we obtain that the optimal general join sequence for the query in Fig. 2a is $((R_2, R_5), (((R_1, R_3), R_6), R_4))$ with its bushy tree shown in Fig. 3d, requiring only a cost of 13,013.57, which is in fact rather close to those obtained by G_{MC} and G_{MR}. Clearly, such an optimal scheme, though leading to the optimal solution sequence, will incur excessive computational overhead which is very undesirable in some applications and might outweigh the improvement it could have over the heuristic schemes. As can be seen in the following, the heuristic schemes G_{MC} and G_{MR}, despite their simplicity, perform significantly better than S_{GD} and S_{OPT}, and result in join sequences whose execution costs are reasonably close to that of the optimal one.

3.3 Simulation Results for Join Sequence Heuristics

Simulations were performed to evaluate the heuristic schemes for query plan generation. The simulation program was coded in C, and input queries were generated as follows. The number of relations in a query was predetermined. The occurrence of an edge between two relations in the query graph was determined according to a given probability, denoted by *prob*. Without loss of generality, only queries with connected query graphs were deemed valid and used for our study. To determine the structure of a query and also the cardinalities of relations and attributes

4. Another heuristic choosing the join (R_i, R_j) with the minimal expansion (i.e., $(|R_i \bowtie R_j| - |R_i| - |R_j|) = \min_{allR_pR_q \in V} |R_p \bowtie R_q| - |R_p| - |R_q|)$ was also evaluated [11], and found to provide mediocre performance. Its results are thus not reported in this paper.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

involved, we referenced prior work on workload characterization [53] and a workload obtained from a Canadian insurance company. To make the simulation be feasibly conducted, we scaled the average number of tuples in a relation down from one million to two thousand. The cardinalities of attributes were also scaled down accordingly so that the join selectivities could still reflect the reality.

Based on the above, the cardinalities of relations and attributes were randomly generated from a uniform distribution within some reasonable ranges. The number of relations in a query, denoted by n, is chosen to be 4, 6, 8, and 10, respectively. For each value of n, 300 queries were randomly generated. For each query, the five scheduling schemes, i.e., S_{GD}, S_{OPT}, G_{MC}, G_{MR}, and G_{OPT}, are performed to determine join sequences to execute the query. When two relations not having join predicates are to be joined together, a Cartesian product is performed. From our simulation, we found that relative performance of these schemes is not sensitive to the density of the query graph, i.e., the number of edges in the graph.[5] The average execution cost for join sequences obtained from each scheme when *prob* = 0.32 is shown in Table 3. Also, we divide the average execution costs of the first four schemes by that of G_{OPT} for a comparison purpose, and show the results associated with Table 3 in Fig. 4.

TABLE 3
THE AVERAGE EXECUTION COST FOR JOIN SEQUENCES OBTAINED BY EACH SCHEME

relation no.	S_{GD}	S_{OPT}	G_{MC}	G_{MR}	G_{OPT}
n = 4	6,905.0	6,001.2	5,112.4	5,087.7	4,762.3
n = 6	23,917.5	18,135.2	14,192.8	13,927.9	13,141.5
n = 8	136,505.7	74,151.2	52,085.5	46,760.8	42,131.4
n = 10	606,518.1	237,223.4	112,723.2	104,311.0	84,121.8

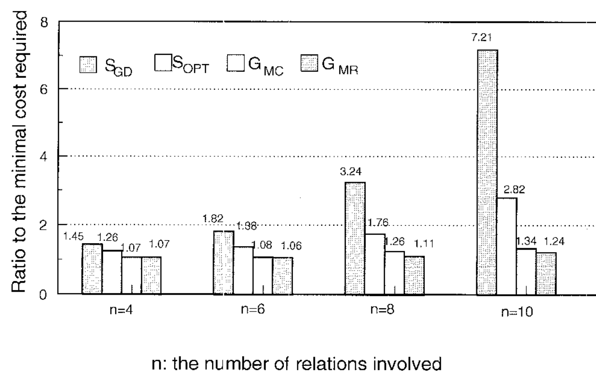


Fig. 4. Performance results of different join sequence heuristics.

From Table 3 and Fig. 4, it can be seen that except for G_{OPT}, the join sequence efficiency of join sequences obtained by G_{MR} is the best among those obtained by the four remaining schemes, and then, in order, those by G_{MC}, S_{OPT}, and S_{GD}. The join sequence efficiencies of the sequences resulted by G_{MC} and G_{MR} are quite close to the optimal one and significantly better than those by S_{GD} and S_{OPT}, especially when the number of relations increases. For the sizes of queries simulated here, the run times of S_{GD}, G_{MC}, and G_{MR} under the RS/6000 environment are very close to one another whereas those of S_{OPT} and G_{OPT} are larger than them by more than three orders of magnitude due to their exponential complexity.

5. Note that the "absolute" performance of these scheduling schemes is highly dependent on the query complexity. Discussions on this issue can be found in [33], [41].

4 PROCESSOR ALLOCATION FOR EXECUTING EACH JOIN

As pointed out in Section 1, to minimize the execution time of a multi-join query, it is necessary to address the following three issues: operational point selection, execution dependency, and system fragmentation. Note that the execution time required for a join operation within a multiprocessor system depends on the number of processors allocated to perform the join, and their relationship can be modeled by an operational curve,[6] as evidenced in prior results on intraoperator parallelism [32], [51]. Basically, increasing the number of processors will reduce the execution time of a join until a saturation point is reached, above which point adding more processors to execute the join will, on the contrary, increase its execution time. This is mainly due to the combining effects of limited parallelism exploitable and excessive communication and coordination overhead over too many processors. An example of an operational curve for this phenomenon is shown by the solid curve in Fig. 5, where a dotted curve xy = 30 is given for reference. In such a curve, the operational point chosen from the curve, depending on the design objective, is generally between the point which minimizes the execution time of the join, referred to as the *minimum time point*, denoted by P_M, and the one which optimizes *execution efficiency*, i.e., minimizes the product of the number of processors and the execution time, referred to as the *best efficiency point*, denoted by p_B.

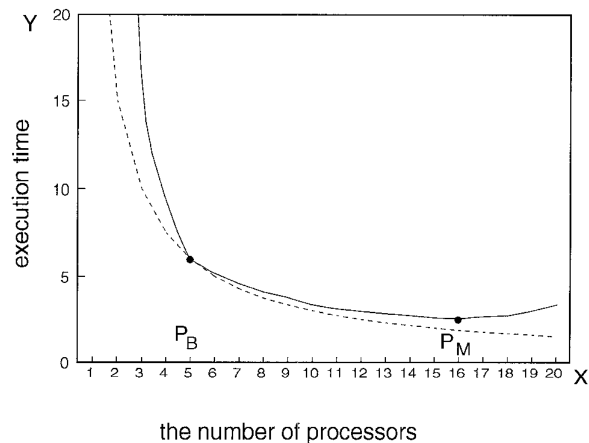


Fig. 5. An example operational curve of a join in a multiprocessor system.

6. Note that every join has its operational curve.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

Formally, the execution efficiency of allocating k processors to execute a join is defined as

$$\frac{\text{exe. time on one proc.}}{\text{k * exe. time on k proc.}}$$

to represent the efficiency of such an allocation. For example, $p_B = 5$ and $p_M = 16$ for the operational curve in Fig. 5. To improve the processor allocation efficiency, we not only have to utilize the information provided in the operational curve for the operational point selection, but are also required to comply with execution dependency and avoid system fragmentation as much as possible so as to minimize the execution time of the query.

Consequently, we propose and evaluate in the following several heuristics to determine the number of processors allocated for the execution of each join. The heuristics proposed can be divided into two categories:

1) the bottom-up approach, presented in Section 4.1, determines the join sequence and processor allocation at the same time, i.e., processors are allotted when a bushy tree is being built, and
2) the top-down approach, presented in Section 4.2, determines the processor allocation based on a given bushy tree.

The effectiveness of these heuristics will be evaluated by simulation in Section 4.3.

4.1 Bottom-Up Approach for Processor Allocation

We introduce below four heuristics for the bottom-up approach to determine the processor allocation.

1) Sequential execution (SE):
This heuristic is to use all the processors in the system to execute each join in the query sequentially. It can be seen that interoperator parallelism is absent when this heuristic is used, and the join sequence is the key factor to the performance in such a case.
2) Fixed cluster size (FS):
This heuristic is to allocate a fixed number of processors for the execution of each join to avoid system fragmentation. Clearly, by taking the total number of processors as the cluster size, we have a special case equivalent to heuristic SE.
Note that by using the above heuristics, system fragmentation is avoided since a fixed number of processors are always released together for a later use. Moreover, under heuristic SE, execution dependency is inherently observed, since join operations are executed sequentially. However, the two heuristics may suffer from poor operational point selection because the information provided by the operational curve is not utilized to determine the operational point of a join.
3) Minimum time point (MT):
This heuristic is based on the minimum time point in the operational curve, i.e., the number of processors used to execute the corresponding join operation is p_M. Note that even though this operational point obtains the minimum execution time for each join, it may not minimize the execution time of a multi-join query as a whole due to the effect of execution dependency and system fragmentation.
4) Time-efficiency point (TE):
Recall that the best efficiency point is the operational point where processors are most efficiently used to execute the join. However, as can be seen in Fig. 5, a scheme based on the the best efficiency point might suffer from execution dependency, since some join operating at its best efficiency point might take a long execution time to complete due to a small number of processors used to execute the operation, thus causing long waiting time for subsequent joins. On the other hand, a scheme based on MT may not use processors efficiently since it may require too many processors to reach the minimum time point. Clearly, the number of processors associated with an operational point which can strike a compromise between the execution time and the processor efficiency should be within the region $[p_B, p_M]$. In view of this, we shall use a combination of the minimum time point and the best efficiency point, termed as the *time–efficiency point*, as a heuristic for our study, i.e., the number of processors, $k^*p_M + (1 - k)^*p_B$, is used to execute each join operation, where $0 \leq k \leq 1$.

Note that the above heuristics for processor allocation can be combined with the schemes for scheduling join sequences developed in Section 3 to form a final scheduler which handles the scheduling and processor allocation of a multi-join query in a multiprocessor system. That is, we use a join sequence heuristic, say G_{MR}, to determine the next join to be considered and employ the appropriate processor allocation heuristic to determine the number of processors to be allocated for the execution of that join. The operations for the processor allocation and deallocation can be outlined as follows where the processor allocation heuristic, denoted by h_P, can be any of *SE*, *FS*, *MT*, and *TE* described above and $h_P(J)$ is the number of processors allocated to execute a join J under the heuristic h_P.

Processor Allocation:

/* P is the number of processors available and initialized as the total numbers of processors. */

STEP 1: Use the join sequence heuristic to determine the next join operation J such that $h_P(J) \leq P$ and execution dependency is observed, i.e., the two input relations of J are available then. If no such a join exists, go to processor deallocation.

STEP 2: Allocate $h_P(J)$ processors to execute the join J. $P := P - h_P(J)$.

STEP 3: Update the profile by marking J as an ongoing join.

STEP 4: Determine the completion time of J and record it in the completion time list of ongoing joins.

Step 5: Go to Step 1.

Processor Deallocation:

STEP 1: From the completion time list, determine the next completion of an ongoing join, say J.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

STEP 2: Update the profile to reflect that J is completed. P: = P + h_P(J).

STEP 3. If there is any executable join in the updated query profile, go to processor allocation.

STEP 4: Go to Step 1.

It can be seen that using the above procedures, the execution tree can be built bottom-up. To demonstrate the processor allocation and deallocation, we shall show the operations using heuristics SE and TE. The operations by FS and ME follow similarly. Consider the query in Fig. 6 with the profile in Table 4. In light of the results on parallelizing sort and join phases [29], [51], the operational curve of a join can be modeled as a hyperbolic function below,

$$T_{exe} = \frac{a|R_i| + b|R_j| + c|R_i \bowtie R_j|}{N_p} + dN_p,$$

where N_p is the number of processors employed, parameters a, b, and c are determined by the path length of the system in processing and joining tuples [28], [32], [51], and parameter d is determined by the interprocessor communication protocol. Also, as observed in [32], for sort-merge join, runs for sorting are usually memory intensive. In view of this and the fact that the amount of memory available is in proportion to the number of processors involved, we have, for each join, the minimal number of processors required for its execution according to the sizes of its operands. p_B for each operational curve formulated above can thus be determined for our study. We then ignore the operational area where the number of processors is less than p_B, and consider only the operational region $[p_B, p_M]$ for efficient execution. Without loss of generality, G_{MR} is used to determine the next join operation to be executed.[7] Then, for heuristics SE in a multiprocessor system of 32 nodes with a = b = c = 1 and d = 20, we have the execution sequence as shown in Table 5a, where the column $W(R_i)$ is the *cumulative execution cost* of R_i, and will be used in Section 4.2 to implement top-down approaches. The bushy tree and its corresponding processor allocation by SE is shown in Fig. 7a. The execution scenarios using the time-efficiency point are shown in Table 5b, where the time-efficiency point used is determined by $0.3p_B + 0.7p_M$.[8] The bushy tree and its corresponding processor allocation by TE is shown in Fig. 7b. Note that though the same scheme G_{MR} is used to determine the next join to be performed in both cases, the resulting join sequences are different from each other due to different processor allocation scenarios. It can be seen that the bushy tree in Fig. 7b is different from the one in Fig. 7a.

4.2 Top-down Approach for Processor Allocation

It can be seen that when an execution tree is built bottom-up, the following two constraints have to be followed:

1) execution dependency is observed, i.e., the operands of the join selected to be performed next do not depend on the resulting relation of any ongoing join, and
2) the processor requirement is satisfied according to the processor allocation heuristic employed, i.e., the number of processors required by that join is not larger than the number of processors available then.

As can be seen in Tables 5a and 5b, the above two constraints lengthen the execution time of a query and degrade the performance of a scheduler since the first constraint causes long waiting time for some operands, and the second can result to the existence of idle processors. In view of these, one naturally wants to achieve some degree of execution synchronization, meaning that processors are allocated to joins in such a way that the two input relations of each join can be made available approximately the same time. Also, idleness of processors should be avoided. As a result, we propose the top-down approach for the processor allocation which uses the concept of synchronous execution time to alleviate the two constraints and improve the query execution time.

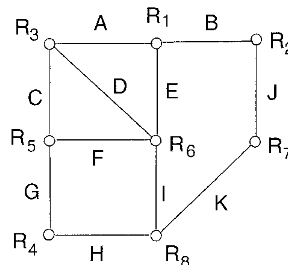


Fig. 6. A query to show processor allocation.

TABLE 4
THE PROFILE OF THE QUERY IN FIG. 6

relation R_i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
cardinality	100	85	93	106	102	90	101	94

(a) *Cardinalities of relations*

attribute	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*
cardinality	9	8	7	9	9	10	9	7	7	10	8

(b) *Cardinalities of attributes*

To describe the processor allocation using the synchronous execution time, consider the bushy tree in Fig. 7a for example. Recall that every internal node in the bushy tree corresponds to a join operation, and we determine the number of processors allocated to each join in the manner of top-down. Clearly, all processors are allocated to the join associated with the root in the bushy tree since it is the last join to be performed. Then, those processors allocated to the join on the root are partitioned into two clusters which are assigned to execute the joins associated with the two child nodes of the root in the bushy tree in such a way that the two joins can be completed approximately the same time. The above step for partitioning the processors for the root is then applied to all internal nodes in the tree in a top-down manner until each internal node (join) is assigned with a number of processors. More formally, define the

7. The corresponding simulation results by using G_{MC} do not provide additional information, and are thus omitted in this paper.

8. Different values for k have been evaluated. The choice for k = 0.3 is made for its reasonably good performance.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

cumulative execution costs of an internal node as the sum of the execution costs of all joins in the subtree under that internal node. Also, define the cumulative execution cost of a leaf node (an original relation) as zero. Let R_i be a relation associated with an internal node in the bushy tree, and R_x and R_y be the relations corresponding to its two child nodes. Then, the cumulative execution cost of the node with R_i, denoted by $W(R_i)$, is determined by

$$W(R_i) = W(R_x) + W(R_y) + cost(R_x, R_y).$$

Note that the cumulative execution cost of each node can be determined when the bushy tree is built bottom-up. The cumulative execution costs of internal nodes for the bushy trees in Figs. 7a and 7b can be found in Tables 5a and 5b, respectively. Then, it is important to see that to achieve the synchronous execution time, when partitioning the processors of a node into two clusters for its child nodes, one has to take into account the cumulative execution costs of the two child nodes, rather than the execution costs of the two joins associated with the two child nodes. Let R_i be a relation associated with an internal node in the bushy tree and R_x and R_y be the relations corresponding to its two child nodes such that $W(R_x) \geq W(R_y)$. Denote the number of processors allocated to perform the join generating R_i as $P(R_i)$. Then, $P(R_x)$ and $P(R_y)$ are determined, respectively, by

$$P(R_x) = \left\lceil P(R_i) \frac{W(R_x)}{W(R_x) + W(R_y)} \right\rceil \text{and } P(R_y) = P(R_i) - P(R_x).$$

Since $W(R_y) = 0$ if R_y is an original relation, we know that when only one child node corresponds to a join and the other is a leaf node, the former inherits all processors. Note that if the number of processors allocated to an internal node (join) of a bushy tree, say r processors, exceeds that required for the minimum time point, we shall employ p_M processors to perform that join whereas using r processors for the subsequent partitioning for the subtree under that internal node. Also, when the number of processors passed to an internal node in a lower level of the tree is too few to be further partitioned for efficient execution of joins, sequential execution for the joins in its child nodes is employed for a better performance. Clearly, there are many different bushy execution trees for a query. It can be seen that the problem of determining the optimal bushy tree to minimize the execution time by the concept of synchronous execution time is of exponential complexity. For an efficient solution, we apply the concept of synchronous execution time to the bushy trees obtained by the heuristics introduced in Section 4.1.

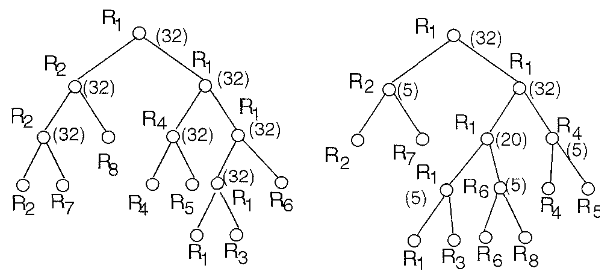


Fig. 7. Bottom-up processor allocation.

TABLE 5
EXECUTION SEQUENCE FOR DIFFERENT HEURISTICS

join sequence	proc. no.	starting time	end time	resulting R_i	$W(R_i)$
(R_2, R_7)	32	0.0	289.2	R_2	1,044.5
(R_4, R_5)	32	289.2	625.3	R_4	1,409.9
(R_1, R_3)	32	625.3	940.5	R_1	1,226.3
(R_1, R_6)	32	940.5	1,367.7	R_1	3,497.8
(R_2, R_8)	32	1,367.7	2,307.7	R_2	12,084.3
(R_1, R_4)	32	2,307.7	4,636.8	R_1	26,961.0
(R_1, R_2)	32	4,636.8	22,553.3	R_1	565,892.7

(a) SE

join sequence	proc. no.	starting time	end time	resulting R_i	$W(R_i)$
(R_2, R_7)	5	0.0	308.9	R_2	1,044.5
(R_1, R_3)	5	0.0	345.2	R_1	1,226.3
(R_6, R_8)	5	0.0	378.5	R_6	1,392.5
(R_4, R_5)	5	0.0	381.8	R_4	1,409.9
(R_1, R_6)	20	378.5	1,661.5	R_1	20,278.7
(R_1, R_4)	32	1,661.5	4,002.1	R_1	76,107.7
(R_1, R_2)	32	4,002.1	21,695.6	R_1	622,866.5

(b) TE

join sequence	proc. no.	starting time	end time	resulting R_i	$W(R_i)$
(R_2, R_7)	10	0.0	289.2	R_2	1,044.5
(R_4, R_5)	6	0.0	354.8	R_4	1,409.9
(R_1, R_3)	16	0.0	315.1	R_1	1,226.3
(R_1, R_6)	16	315.1	742.3	R_1	3,497.8
(R_2, R_8)	10	289.2	1,593.2	R_2	12,084.3
(R_1, R_4)	22	742.3	1,684.7	R_1	26,961.0
(R_1, R_2)	32	1,684.7	19,101.2	R_1	565,892.7

(c) ST_{SE}

join sequence	proc. no.	starting time	end time	resulting R_i	$W(R_i)$
(R_2, R_7)	1	0.0	2,064.5	R_2	1,044.5
(R_1, R_3)	14	0.0	315.1	R_1	1,226.3
(R_6, R_8)	15	0.0	334.0	R_6	1,392.5
(R_4, R_5)	2	0.0	944.6	R_4	1,409.9
(R_1, R_6)	29	334.0	1,523.0	R_1	20,278.7
(R_1, R_4)	31	1,523.0	3,863.6	R_1	76,107.7
(R_1, R_2)	32	3,863.6	20,557.2	R_1	622,866.5

(d) ST_{TE}

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

As pointed out before, different bottom-up processor allocation heuristics used may result in different bushy trees even when the same join sequence heuristic is applied. It is important to see that although execution time for the sequence in Table 5a (by SE) is larger than that in Table 5b (by TE), the join sequence efficiency of the bushy tree in Fig. 7a is in fact better than that of the tree in Fig. 7b, as shown by their cumulative execution costs in Tables 5a and 5b. Note that the constraints on execution dependency can get introduced when a bushy tree is being built by heuristic TE, as well as by FS and MT. Such constraints are absent when heuristic SE is employed to form the bushy tree. (This explains why the tree in Fig. 7a is different from that in Fig. 7b.) Thus, the bushy tree by SE is in fact superior to those by other heuristics in that the former has a better join sequence efficiency owing to full exploitation of the join sequence heuristics. Therefore, we shall apply the concept the synchronous execution time to the bushy tree built by SE, denoted by ST_{SE}. For a comparison purpose, we also investigate the use of the synchronous execution time on the bushy tree built by TE, denoted by ST_{TE}.

The execution scenario using the heuristic ST_{SE} is shown in Table 5c, and the corresponding bushy tree and processor allocation is shown in Fig. 8a. In spite of the fact that the bushy tree in Fig. 8a is the same as that in Fig. 7a, the resulting execution times differ due to the difference in processor allocation. It can be seen that under ST_{SE}, processors are allocated to the execution of each join in such a way that two joins generating the two operands for a later join can be completed approximately the same time, thus alleviating execution dependency. Moreover, since the processors allocated to a node in a bushy tree are partitioned for the allocation to its child nodes, system fragmentation is eased. This explains why ST_{SE} outperforms SE despite both of them have the identical bushy trees and the same join sequence efficiency. The execution scenario using the heuristic ST_{TE} execution time is shown in Table 5d. The bushy tree and its processor allocation by ST_{TE} is shown in Fig. 8b which has the same bushy tree as the one in Fig. 7b, but differs from the latter in processor allocation. It is important to see that despite TE outperforms SE, ST_{SE} performs better than ST_{TE}, and in fact is the best one among the processor allocation heuristics evaluated in Section 4.3.

4.3 Simulation Results for Processor Allocation

The query generation scheme employed in Section 3.3 is used to produce input queries for simulation in this subsection. As in Section 3.3, 300 queries with a given number of relations involved were randomly generated with the occurrence of an edge in the query graph also determined by a given probability *prob*. For each query, the six scheduling schemes, according to the heuristics of SE, FS, MT, TE, ST_{SE}, and ST_{TE}, respectively, are performed to determine the number of processors for each join to execute the query. As in Section 3.3, the simulation results here also indicate that the above heuristics are not sensitive to different values of *prob*. Thus, we shall only show the results for *prob* = 0.30 in the following. For a multiprocessor of 48 nodes, the average execution times obtained by each heuristic for queries of 10, 15, 20, and 25 relations are shown in Table 6a. It can be seen that heuristic SE, i.e., the one using intraoperator parallelism only, performs well when the number of relations is 10, but performs worse when the number of relations increases. This agrees with our intuition since as the number of relations increases, the opportunity to exploit interoperator parallelism increases and the constraint imposed by execution dependency becomes relatively less severe. Also, heuristic FS is in general outperformed by others due mainly to execution dependency and poor operational points selection. Among the heuristics on bottom-up approaches, the shortest execution time is usually achieved by heuristic TE, especially when the number n is large. This can be explained by the same reason as mentioned above, i.e., that execution dependency is eased when the number of relations is large, and TE thus performs best for its best usage of processors.

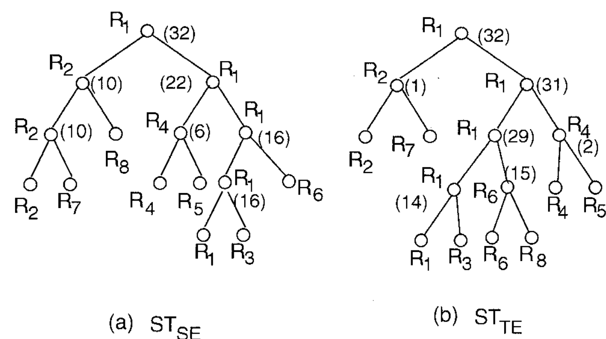


Fig. 8. Top-down processor allocation (synchronous execution time).

Also, from 300 randomly generated queries, the average execution times obtained by the six heuristics for a query of 15 relations is shown in Table 6b where the number of processors in the system is varied from 16 to 64. It can be seen that when the number of processors increases, heuristic SE suffers from the inefficient use of processors, and is thus outperformed by heuristics MT, TE, ST_{SE}, and ST_{TE} by a wide margin. It can also be observed that heuristic TE which uses processors efficiently to achieve a nearly minimum execution time performs well when the number of processors is large. Clearly, the more processors are in the system, the more parallelism can be exploited by heuristic TE. However, MT performs better than TE when pn = 64, which can be explained by the fact that when the supply of processors is sufficient, achieving minimum time point (by MT) becomes a better heuristic than using processors efficiently (by TE). In all, when the number of processors is small, utilizing intraoperator parallelism (i.e., SE) will suffice to provide a reasonably good performance. On the other hand, for a large multiprocessor system, one has to resort to interoperator parallelism to fully exploit the resources in the system. Note, however, that without using synchronous execution time, ME and TE, though having a good operational point selection for each join, cannot improve the query response time in a global sense due to the nature of a bottom-up approach, and are thus outperformed by ST_{SE} and ST_{TE}. This fact strongly justifies the necessity of taking execution dependency and system

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

TABLE 6
THE AVERAGE EXECUTION TIME FOR EACH HEURISTIC

relation no.	SE	FS	MT	TE	ST_{SE}	ST_{TE}
n = 10	5,151.1	15,817.,5	4,698.2	4,685.7	4,337.5	4,440.6
n = 15	9,041.1	20,828.7	7,659.9	7,135.4	5,990.2	6,284.5
n = 20	13,803.6	25,114.,5	11,905.2	10,159.8	7,951.0	8,296.2
n = 25	16,716.6	26,105.4	13,648.1	12,682.8	9,600.3	10,066.1

(a) When the number of processors is 48.

Proc. no.	SE	FS	MT	TE	ST_{SE}	ST_{TE}
pn = 16	15,925.1	50,965.9	15,744.1	15,473.7	14,663.8	14,916.4
pn = 32	11,295.9	41,060.7	11,109.2	10,832.7	9,737.2	9,899.2
pn = 48	9,041.1	20,828.7	7,695.9	7,135.4	5,990.2	6,284.5
pn = 64	9,021.1	14,575.5	6,808.2	6,912.1	5,683.0	6,152.7

(b) When the number of relations is 15.

fragmentation into consideration when interoperator parallelism is exploited.

As mentioned earlier, although SE is outperformed by TE due to its poor operational point selection, ST_{SE} remedies this defect by properly reallocating processors using the concept of synchronous execution time. ST_{SE} can thus outperform ST_{TE}. It is worth mentioning that the sequential join sequences, such as the one shown in Fig. 3a, will not benefit from the concept of synchronous execution time, since in this case, joins have to be executed sequentially and there is no interoperator parallelism exploitable. This fact, together with the fact that the sequential join sequences usually suffer from poor join sequence efficiency, accounts for the importance of exploring the general join sequences.

Note that similar to the heuristics in Section 3, the heuristics we investigated here are very straightforward and require little implementation overhead. In all, our results showed that the join sequence efficiency is in general the dominating factor for the query execution time whereas the processor allocation efficiency becomes significant as the number of processors and query complexity increase. This suggests that for an efficient solution, one can attempt to optimize the join sequence efficiency by building a good bushy tree first and then improve the processor allocation efficiency by appropriately allocating processors for the execution of each join. This is in fact how the heuristic ST_{SE} is constructed.

5 CONCLUSION

In this paper we dealt with two major issues to exploit interoperator parallelism within a multi-join query:

1) join sequence scheduling and
2) processor allocation.

For the first issue, we explored the general join sequence so as to exploit the parallelism achievable in a multiprocessor system. Heuristics G_{MC} and G_{MR} were derived and evaluated by simulation. The heuristics proposed, despite their simplicity, were shown to lead to general join sequences whose join sequence efficiencies are close to that of the optimal one (G_{OPT}), and significantly better than what is achievable by the optimal sequential join sequence (S_{OPT}), particularly when the number of relations in the query is large.

Moreover, we explored the issue of processor allocation. In addition to the operational point selection needed for intraoperator parallelism, we identified and investigated two factors: execution dependency and system fragmentation, which are shown to be important when exploiting interoperator parallelism. Several processor allocation heuristics, categorized by bottom-up and top-down approaches, were proposed and evaluated by simulation. To form a final scheduler to perform a multi-join query, we combined the results on join sequence scheduling and processor allocation. Among all the schemes evaluated, the two-step approach by ST_{SE}, which

1) first applies the join sequence heuristic to build a bushy tree to minimize the total amount of work required as if under a single processor system, and then,
2) in light of the concept of synchronous execution time, allocates processors to the internal nodes of the bushy tree in a top-down manner, is shown to be the best solution to minimize the query execution time.

APPENDIX: EXPECTED RESULTING CARDINALITIES OF JOINS

PROPOSITION. Let G = (V, E) be a join query graph. $G_B = (V_B, E_B)$ is a connected subgraph of G. Let $R_1, R_2, \ldots, R_q$ be the relations corresponding to vertices in V_B, $A_1, A_2, \ldots, A_r$ be the distinct attributes associated with edges in E_B and m_i be the number of different vertices (relations) that edges with attribute A_i are incident to.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

Suppose R_M is the relation resulting from all the join operations between relations in G_B and $N_T(G_B)$ is the expected number of tuples in R_M. Then,

$$N_T(G_B) = \frac{\Pi_{i=1}^{q}\left|R_i\right|}{\Pi_{i=1}^{r}\left|A_i\right|^{m_i-1}}.$$

REFERENCES

[1] P. America, "Parallel Database Systems," *Proc. PRISMA Workshop, LNCS 503*, Springer-Verlag, 1991.

[2] C. Baru et al., "An Overview of DB2 Parallel Edition," *Proc. ACM SIGMOD*, pp. 460–462, May 1995.

[3] C.K. Baru and O. Frieder, "Database Operations in a Cube-Connected Multiprocessor System," *IEEE Trans. Computers*, vol. 38, no. 6, pp. 920–927, June 1989.

[4] B. Bergsten, M. Couprie, and M. Lopez, "DBS3: A Parallel Database System for Shared Store," *Proc. Second Int'l Conf. Parallel and Distributed Information Systems*, pp. 260–262, Jan. 1993.

[5] A. Bhide and M. Stonebraker, "A Performance Comparison of Two Architectures for Fast Transaction Processing," *Proc. Fourth Int'l Conf. Data Eng.*, pp. 536–545, Feb. 1988.

[6] H. Boral et al., "Prototyping Bubba, A Highly Parallel Database System," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 1, pp. 4–24, Mar. 1990.

[7] K. Bratbergsengen and T. Gjelsvik, "The Development of the CROSS8 and HC16-186 (Database) Computers," *Proc. Sixth Int'l Workshop Database Machines*, June 1989.

[8] L. Chambers and D. Cracknell, "Parallel Features of NonStop SQL," *Proc. Second Int'l Conf. Parallel and Distributed Information Systems*, pp. 69–70, Jan. 1993.

[9] M.-S. Chen, M.-L. Lo, P.S. Yu, and H.C. Young, "Applying Segmented Right-Deep Trees to Pipelining Multiple Hash Joins," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 4, pp. 656-668, Aug. 1995.

[10] M.-S. Chen and P. S. Yu, "Combining Join and Semijoin Operations for Distributed Query Processing," *IEEE Trans. Knowledge and Data Eng.*, vol. 5, no. 3, pp. 534–542, June 1993.

[11] M.-S. Chen, P. S. Yu, and K.-L. Wu, "Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries," *Proc. Eighth Int'l Conf. Data Eng.*, pp. 58–67, Feb. 1992.

[12] D. Clay, "Informix Parallel Data Query," *Proc. Second Int'l Conf. Parallel and Distributed Information Systems*, pp. 71–72, Jan. 1993.

[13] S.M. Deen, D.N.P. Kannangara, and M.C. Taylor, "Multi-Join Parallel Processors," *Proc. Second Int'l Symp. Databases in Parallel and Distributed Systems*, pp. 92–102, July 1990.

[14] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri, "Practical Skew Handling in Parallel Joins," *Proc. 18th Int'l Conf. Very Large Data Bases*, pp. 27–40, Aug. 1992.

[15] D.J. DeWitt and R. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proc. 11th Int'l Conf. Very Large Data Bases*, pp. 151–162, Aug. 1985.

[16] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.I. Hsiao, and R. Rasmussen, "The Gamma Database Machine Project," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 1, pp. 44–62, Mar. 1990.

[17] D. J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Comm. ACM*, vol. 35, no. 6, pp. 85–98, June 1992.

[18] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query Optimization for Parallel Execution," *Proc. ACM SIGMOD*, pp. 9–18, June 1992.

[19] Gardarin et al., "Design of a Multiprocessor Relational Database System," *Proc. Information Processing 83*, pp. 363–367, 1983.

[20] D. Gardy and C. Puech, "On the Effect of Join Operations on Relation Sizes," *ACM Trans. Database Systems*, vol. 14, no. 4, pp. 574–603, Dec. 1989.

[21] R. Gerber, "Dataflow Query Processing Using Multiprocessor Hash-Partitioned Algorithms," Technical Report No. 672, Computer Science Dept., Univ. of Wisconsin-Madison, Oct. 1986.

[22] G. Graefe, "Rule-Based Query Optimization in Extensible Database Systems," Technical Report Technical Report No. 724, Computer Science Dept., Univ. of Wisconsin-Madison, Nov. 1987.

[23] P. Haas and A. Swami, "Sequential Sampling Procedures for Query Size Estimation," *Proc. ACM SIGMOD*, pp. 341–350, June 1992.

[24] D.J. Haderle and R.D. Jackson, "IBM Database 2 Overview," *IBM Systems J.*, vol. 23, no. 2, pp. 112–125, 1984.

[25] W. Hong, "Exploiting Inter-Operator Parallelism in XPRS," *Proc. ACM SIGMOD*, pp. 19–28, June 1992.

[26] H.-I. Hsiao, M.-S. Chen, and P.S. Yu, "On Parallel Execution of Multiple Pipelined Hash Joins," *Proc. ACM SIGMOD*, pp. 185–196, Minneapolis, Minn., May 1994.

[27] K.A. Hua, Y.L. Lo, and H. Young, "Considering Data Skew Factor in Multi-Way Join Query Optimization for Parallel Execution," *VLDB J.*, vol. 2, no. 3, pp. 303–330, July 1993.

[28] B. Iyer and D.M. Dias, "System Issues in Parallel Sorting for Database Systems," *Proc. Sixth Int'l Conf. Data Eng.*, pp. 246–255, 1990.

[29] B. Iyer, G. Ricard, and P. Varman, "Percentile Finding Algorithm for Multiple Sorted Runs," *Proc. 15th Int'l Conf. Very Large Data Bases*, pp. 135–144, Aug. 1989.

[30] M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys*, vol. 167, no. 2, pp. 111–152, June 1982.

[31] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Architecture and Performance of Relational Algebra Machine GRACE," *Proc. Int'l Conf. Parallel Processing*, pp. 241–250, Aug. 1984.

[32] M. S. Lakshmi and P. S. Yu, "Effectiveness of Parallel Joins," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 4, pp. 410–424, Dec. 1990.

[33] R. Lanzelotte, P. Valduriez, and M. Zait, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces," *Proc. 19th Int'l Conf. Very Large Data Bases*, Aug. 1993.

[34] B. Linder, "Oracle Parallel RDBMS on Massively Parallel Systems," *Proc. Second Int'l Conf. Parallel and Distributed Information Systems*, pp. 67–68, Jan. 1993.

[35] M.-L. Lo, M.-S. Chen, C.V. Ravishankar, and P.S. Yu, "On Optimal Processor Allocation to Support Pipelined Hash Joins," *Proc. ACM SIGMOD*, pp. 69–78, May 1993.

[36] R.A. Lorie, J.-J. Daudenarde, J.W. Stamos, and H.C. Young, "Exploiting Database Parallelism In a Message-Passing Multiprocessor," *IBM J. Research and Development*, vol. 35, nos. 5/6, pp. 681–695, Sept./Nov. 1991.

[37] H. Lu, M.-C. Shan, and K.-L. Tan, "Optimization of Multi-Way Join Queries for Parallel Execution," *Proc. 17th Int'l Conf. Very Large Data Bases*, pp. 549–560, Sept. 1991.

[38] H. Lu, K.L. Tan, and M.-C. Shan, "Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory," *Proc. 16th Int'l Conf. Very Large Data Bases*, pp. 198–209, Aug. 1990.

[39] P. Mishra and M. H. Eich, "Join Processing in Relational Databases," *ACM Computing Surveys*, vol. 24, no. 1, pp. 63–113, Mar. 1992.

[40] E.R. Omiecinski and E.T. Lin, "Hash-Based and Index-Based Join Algorithms for Cube and Ring Connected Multicomputers," *IEEE Trans. Knowledge and Data Eng.*, vol. 1, no. 3, pp. 329–343, Sept. 1989.

[41] K. Ono and G. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization," *Proc. 16th Int'l Conf. Very Large Data Bases*, pp. 314–325, Aug. 1990.

[42] H. Pirahesh, C. Mohan, J. Cheng, T.S. Liu, and P. Selinger, "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches," *Proc. Second Int'l Symp. Databases in Parallel and Distributed Systems*, pp. 4–29, July 1990.

[43] D. Reiner, "The Kendall Square Query Decomposer," *Proc. Second Int'l Conf. Parallel and Distributed Information Systems*, pp. 36–37, Jan. 1993.

[44] J. Richardson, H. Lu, and K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms," *Proc. ACM SIGMOD*, pp. 399–409, May 1987.

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

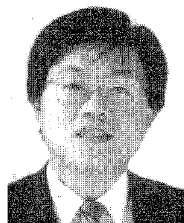
[46] D. Schneider and D.J. DeWitt, "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines," *Proc. 16th Int'l Conf. Very Large Data Bases*, pp. 469–480, Aug. 1990.

[47] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD*, pp. 23–34, 1979.

[48] T. Sellis, "Multiple Query Optimization," *ACM Trans. Database Systems*, vol. 13, no. 1, pp. 23–52, Mar. 1988.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.

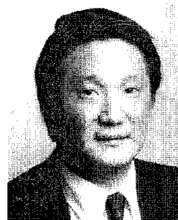
[49] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS," *Proc. 14th Int'l Conf. Very Large Data Bases*, pp. 318–330, 1988.
[50] "DBC/1012 Database Computer System Manual Release 2.0," Technical Report Document No. C10-0001-02, Teradata Corp., Nov. 1985.
[51] J.L. Wolf, D.M. Dias, and P.S. Yu, "A Parallel Sort Merge Join Algorithm for Managing Data Skew," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 1, pp. 70–86, Jan. 1993.
[52] J.L. Wolf, J. Turek, M.-S. Chen, and P.S. Yu, "A Hierarchical Approach to Parallel Multiquery Scheduling," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 6, pp. 578–590, June 1995.
[53] P.S. Yu, M.-S. Chen, H. Heiss, and S.H. Lee, "On Workload Characterization of Relational Database Environments," *IEEE Trans. Software Eng.*, vol. 18, no. 4, pp. 347–355, Apr. 1992.
[54] P.S. Yu, M.-S. Chen, J.L. Wolf, and J.J. Turek, "Parallel Query Processing," *Advanced Database Systems*, chapter 12, N. Adam and B. Bhargava, eds., Lecture Notes in Computer Science 759, pp. 239–258, Springer-Verlag, Dec. 1993.
[55] M. Ziane, M. Zait, and P. Borla-Salamet, "Parallel Query Processing in DBS3," *Proc. Second Conf. Parallel and Distributed Information Systems*, pp. 93–102, Jan. 1993.



Ming-Syan Chen (S'87-M'88-SM'93) received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1982; and the MS and PhD degrees in computer, information, and control engineering from the University of Michigan, Ann Arbor, in 1985 and 1988, respectively.

Dr. Chen is a faculty member in the Electrical Engineering Department at National Taiwan University. Previously, he was a research staff member with the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, from 1988 to 1996, involved in projects related to database systems and multimedia systems. From 1985 to 1988, he was a member of the Real-Time Computing Laboratory at the University of Michigan, Ann Arbor, where he conducted research on multiprocessor systems. His current research interests include multimedia technologies, database systems, and Internet applications.

Dr. Chen has served as a program committee member, section chair, and tutorial lecturer at many conferences on his research areas, and is currently a guest editor of *IEEE Transaction on Knowledge and Data Engineering* on a special issue on mining of databases that is scheduled for publication in 1996. He invented technology that has been granted, or that has been applied toward granting, 15 U.S. patents in the areas of interactive video playout, video server design, data mining, and scalable video placement. He has authored or coauthored more than 65 papers for refereed journals and conference proceedings. He received the Outstanding Innovation Award from IBM in 1994 for his contribution to parallel transaction design and implementation for a major database product, plus numerous awards for his inventions and patent applications. Dr. Chen is a member of the Association for Computing Machinery and a senior member of the IEEE.



Philip S. Yu (S'76-M'78-SM'87-F'93) received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, Republic of China, in 1972; the MS and PhD degrees in electrical engineering from Stanford University in 1976 and 1978, respectively; and the MBA degree from New York University in 1982.

Since 1978, Dr. Yu has been with the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. Currently, he is manager of the Architecture Analysis and Design Group. His research interests include transaction and query processing, database systems, data mining, parallel and distributed processing, multimedia systems, disk arrays, computer architecture, performance modeling, and workload analysis. He has published more than 180 papers in refereed journals and conference proceedings, and in excess of 110 research reports and 80 invention disclosures. He holds, or has applied for, 22 U.S. patents.

Dr. Yu is a member of the ACM and a fellow of the IEEE. He is an editor of *IEEE Transactions on Knowledge and Data Engineering*. In addition to serving as a program committee member for various conferences, he served as program chair of the Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, and as program cochair of the 11th International Conference on Data Engineering. He has received several IBM and non-IBM honors, including a Best Paper Award, IBM Outstanding Innovation Awards, an Outstanding Technical Achievement Award, a Research Division Award, and 12 Invention Achievement Awards.

Kun-Lung Wu (S'85-M'90) received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1982; and the MS and PhD degrees in computer acience from the University of Illinois at Urbana-Champaign in 1986 and 1990, respectively.

From 1985 to 1989, he was a research assistant at the Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois. In the summer of 1986, he also worked as a consultant to Texas Instruments, Dallas, Texas. Since March 1990, he has been with the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, where he is currently a research staff member in the Architecture Analysis and Design Group. His research interests include database transaction and query processing, performance analysis, parallel and distributed processing, memory and I/O management, multimedia applications, and network-centric information services.

Dr. Wu has published many refereed journal and conference-proceedings papers in his research areas. He has received awards from IBM for his patented inventions. He has also served as an organizing/program committee member for various IEEE conferences. Dr. Wu is a member of the ACM, Phi Kappa Phi, and the IEEE.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:17 from IEEE Xplore. Restrictions apply.