SCHEDULING AND PROCESSOR ALLOCATION
FOR PARALLEL EXECUTION OF MULTI-JOIN QUERIES

Ming-Syan Chen, Philip S. Yu and Kun-Lung Wu

IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, New York 10598

ABSTRACT

We explore in this paper inter-operator parallelism within a multi-join query to minimize the execution time of the query in a multiprocessor system. Specifically, the focus is on two major issues: (i) scheduling the execution sequence of multiple joins within a query, and (ii) determining the number of processors to be allocated for the execution of each join operation obtained in (i). For the first issue, we propose and evaluate by simulation several methods to determine the general join sequences, or the bushy execution trees. Despite their simplicity, the heuristics proposed can lead to the general join sequences which significantly outperform the optimal sequential join sequence. In addition, several heuristics to determine the processor allocation, categorized by bottom up and top down approaches, are derived and evaluated by simulation. As confirmed by our simulation, by first using the join sequence heuristics to build a bushy tree and then applying the concept of the synchronous execution time to the bushy tree for processor allocation, we have an efficient two-step approach to schedule and execute multi-join queries in a multiprocessor system.

1. Introduction

In recent years, multiprocessor-based parallel database machines have drawn considerable attention from both the academic and industrial communities due to the high potential for parallel execution of complex database operations [6] [14] [20] [22]. In relational database systems, joins are the most expensive operations to execute, especially with the increasing database sizes [25]. Furthermore, several applications which involve decision support and complex objects usually have to specify their desired results in terms of multi-join queries. As a result, some complex queries in such systems may take hours or even days to complete and the system performance is thus degraded. Consequently, the development of effective schemes to exploit parallelism in the execution of multi-join queries has become increasingly attractive and important.

As pointed out in [18], 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 intra-operator parallelism. Various solutions for exploiting intra-operator 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 [1] [5] [12] [16] [17]. The effect of data skew on the performance of parallel joins has been analyzed in [10] [24]. The second form of parallelism is termed inter-operator 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 inter-query parallelism. It can be seen that to exploit the third form of parallelism, one has to resort to the results derived for inter-operator parallelism within a query. However, despite its importance, inter-operator parallelism has not yet been fully studied, although during the past few years some light has been shed on this issue [4] [8] [15] [18].

In view of this, the objective of this paper is to study and improve the execution of multi-join queries with the focus on deriving efficient schemes to exploit inter-operator parallelism in a multiprocessor system.

Note that different execution sequences of joins in a query will result in different execution costs [19]. Also, in a multiprocessor system the execution time of each join strongly depends on the number of processors allocated for the execution of the join [10]. Thus, the subject of exploiting inter-operator parallelism for the execution of a multi-join query mainly consists of the following two major issues: (i) join sequence scheduling, i.e., scheduling the execution sequence of joins in the query, and (ii) processor allocation, i.e., determining the number of processors for each join obtained in (i) so that the execution time required for the query can be minimized. To facilitate our discussion, 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 closely on the two factors.

For the issue of join sequence scheduling, we develop and evaluate by simulation several heuristics to determine the sequence of joins for a multi-join query with the focus on optimizing the join sequence efficiency, i.e., minimizing the total amount of work required in the join sequence. 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 non-leaf node (internal node) represents the resulting relation from joining its child nodes and the join sequence is performed in the manner of bottom up. For notational simplicity, such an execution tree in Fig. 1a will be denoted by the expression (((R1,R2),R3),R4),R5). 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 [3], or a composite inner [13]. The general join sequence in Fig. 1b can be denoted as \((R_1, R_2, (R_3, R_4, R_5))\). Note that the cost of a join is usually proportional to the sizes of its operands. In view of this and the fact that the size of the resulting relation of a join is often larger than those of its operands, it is important to explore the general join sequence in order to reduce the total cost incurred. Particularly for complex queries, i.e., queries consisting of joins involving many relations. However, prior approaches on executing multi-join queries mostly focused on the use of sequential join sequences [19]. This mainly results from the fact that the problem of finding the optimal general join sequence is inherently difficult, if not infeasible, for complex queries due to its exponential complexity [13] [21], and also that the improvement achievable on the join sequence efficiency by employing general join sequences has not yet been clearly shown. 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 whose join sequence efficiencies are significantly better than that of the optimal sequential join sequence. This is especially true for complex queries. More importantly, it is shown that the join sequence efficiencies obtained by our general join sequence heuristics are rather close to that of the optimal general join sequence, meaning that by employing appropriate heuristics we can avoid excessive search costs and obtain join sequences with very high quality.

Next, we explore the issue of processor allocation to join operations. Note that in the study of intra-operator 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 inter-operator 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 intra-operator 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. Clearly, execution dependency and system fragmentation, as well as the operational point selection, have to be taken into account for a better processor allocation efficiency, 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, which determines the join sequence and processor allocation at the same time, i.e., 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) the top-down approach which is based on the concept of synchronous execution time to determine the processor allocation based on a given bushy tree. The concept of synchronous execution time is to partition the processors allocated to an internal node in the bushy tree into two clusters of processors for the execution of the two joins associated with its two child nodes in such a way that the resulting relations from the two child nodes 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 the schemes on join sequence scheduling and processor allocation can be combined to form a final scheduler for the execution of a multi-join query in a multiprocessor system. As shown by our simulation results, 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 increases. 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 is shown to be 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. Processor allocation is dealt with in Section 4. This paper concludes with Section 5.

2. Preliminaries

We assume that a query is of the form of conjunctions of equi-join predicates. 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. Notice that when a join operation between the two relations \( R_k \) and \( R_q \) in a given query graph is carried out, we can obtain the resulting query graph by shrinking the edges between \( R_k \) and \( R_q \) and merging the two relations together to represent the resulting relation from the join operation.

We use \( |R_k| \) to denote the cardinality of a relation \( R_k \), and \( |A| \) to denote the cardinality of the domain of an attribute \( A \). The notation \( (R_k, R_q) \) is used to mean the join between \( R_k \) and \( R_q \), and \( R_k \ast R_q \) denotes the resulting relation
of \((R_i, R_j)\). As in most prior work, we assume that the execution time incurred is the primary cost measure for the processing of database operations. Also, we focus on the execution of complex queries [21]. Note that there are various join algorithms applicable to execute a join operation in a multiprocessor system, such as the sort-merge join, the hash join and the nested loop join. Different and different multiprocessor systems will result in different execution costs for a join, for which various formulas have been derived in [23]. Since our objective in this paper is to study the execution sequence of multiple joins in a query and the processor allocation for each join so as to minimize the execution time (i.e., the response time) required, we shall focus on the sort-merge join in what follows to have a specific cost model, and the cost function of joining \(R_i\) and \(R_j\) will be expressed by \(|R_i| + |R_j| + |R_i \times R_j|\) as in [24]. This formula is believed to be general and reasonable for joining large relations by sort-merge joins in a multiprocessor system (either with shared disks or with shared everything) as assumed in our discussion. The amount of memory available to execute a join is assumed to be in proportion to the number of processors involved. Clearly, for other join methods the cost model has to be changed accordingly, and additional provisions might be needed to handle some more issues, such as pipelining that is possible under the hash join, which is beyond the scope of this paper. Readers interested in the study of pipelined hash joins are referred to a companion paper [11]. In addition, 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 formulas used in [3], which is given in the Appendix for reference. This assumption is not essential but will simplify our presentation. In the presence of data skew, we only have to modify the formula for estimating the cardinalities of resulting relations from joins accordingly [7].

3. Determining the Join Sequence

For the objective of showing the effect of a join sequence on the total work incurred, in this section assume that the execution of joins is under a single processor system. The join sequence efficiencies of various join sequences can thus be compared with one another.

3.1 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 \(S_{OPT}\). Since \(S\) 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. An algorithmic form of this scheme is shown below.

It can be seen that the complexity of the operation in line 6 of \(S_{GD}\) is \(O(V^2)\), and that of the whole scheme is thus \(O(V^2)\). 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 \(\frac{n!}{2^n}\), 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 the 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 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 \(R_3\) and \(R_4\) is one with the minimal cost among all joins. After the join \((R_3, R_4)\), the resulting query graph is given in Fig. 2b where \(R_3\) now represents the resulting composite. Then, it can be verified that \(R_3\) is the relation which will have the minimal-cost join with \(R_7\), and the execution of \((R_3, R_7)\) in Fig. 2b is executed. Following the above procedure, we have the resulting join sequence by \(S_{GD}\), \(((R_7, R_6), R_5), R_4)\) 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 \(((R_1, R_2, R_3), R_4), R_5)\) whose total cost is 36,135.92, which is less than that required by \(S_{GD}\).

3.2. General join sequences (bushy trees)

To determine a join sequence of the minimal cost, consider the problem of adding up \(n\) numbers using the Huffman algorithm [9], which performs the additions of two numbers in the set that results in the minimal intermediate sum, replaces the two numbers with the intermediate sum and repeats the above procedure until the final sum is obtained. Suppose that the cost of adding two numbers is equal to the value of their sum. The Huffman algorithm is proved to produce the addition sequence that requires the minimal total cost of additions. Note that there is a similarity between this problem and the one to find the join sequence of minimal cost, since in both problems, we would like to minimize the total cost of a sequence of operations (additions and joins, respectively). This similarity suggests us to perform the minimal-cost join first, and then, from the resulting query, choose the minimal-cost join to perform. This procedure repeats until all joins are finished. Based on this heuristic, scheme \(G_{MC}\) where \(G\) means that the resulting sequence is a general join sequence and the subscript \(MC\) is from
### Table 1. Profile for the query in Figure 2a.

<table>
<thead>
<tr>
<th>Rel.</th>
<th>R₁</th>
<th>R₂</th>
<th>R₃</th>
<th>R₄</th>
<th>R₅</th>
<th>R₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card.</td>
<td>118</td>
<td>102</td>
<td>106</td>
<td>100</td>
<td>131</td>
<td>120</td>
</tr>
</tbody>
</table>

(a). Profile for relations

<table>
<thead>
<tr>
<th>Att.</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card.</td>
<td>19</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>16</td>
<td>15</td>
<td>18</td>
</tr>
</tbody>
</table>

(b). The domain size of each attribute

"minimal cost", is outlined below. It can be seen that unlike SGD, the resulting composite of a join by GMC need not participate in the next join.

**Scheme GMC**: /* A scheme to execute the join with the minimal cost */

1. repeat until |V| = 1
2. begin
3. Choose the join (Rᵢ, Rⱼ) from G = (V, E) such that
   \[ \text{cost}(Rᵢ, Rⱼ) = \min \{ \text{cost}(Rᵢ, Rⱼ) \} \]
4. Perform (Rᵢ, Rⱼ).
5. Merge Rᵢ and Rⱼ into Rₙₑₙₑᵢₙₑᵢₙₑᵢₙₑ, in G. Update the profile accordingly.
6. end

For the example query in Fig. 2a, it can be verified that after the first minimal-cost join (R₃, R₄) is performed, the next minimal-cost join to be executed by GMC is (R₁, R₂), rather than (R₃, R₄) as in SGD. The resulting sequence is (((R₃, R₄), (R₁, R₂)), (R₁, R₅)), whose total cost is 13,958.62, significantly less than those required by SGD and Sopr. The execution trees resulted by SGD and GMC are shown in Figs. 3a and 3b, respectively. It can be seen that the complexity in the operation in line 3 of scheme GMC is \(O(|E|)\), and that of the scheme is \(O(|V|^2)\), rather close to \(O(|V|^2)\) required by SGD.

Note that in 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 two heuristic schemes which are variations of GMC, namely the minimal resulting relation (GMR) and the minimal expansion (GME). Instead of finding the minimal-cost join as in GMC, these two schemes look for joins which result in, respectively, the minimal resulting relation and the minimal expansion, where the expansion of a join (Rᵢ, Rⱼ) is defined as the size increase, i.e., \(|Rᵢ^*Rⱼ| - |Rᵢ| - |Rⱼ|\). Clearly, the heuristic schemes GMR and GME are of the same complexity, \(O(|V||E|)\), as scheme GMC. Algorithmic forms of GMR and GME are similar to the one of GMC, except that the statement 3 in GMC is changed to 3A and 3B below for GMR and GME, respectively.

3A. (for GMR): Choose the join (Rᵢ, Rⱼ) from G = (V, E) such that
   \[ |Rᵢ^*Rⱼ| = \min \left\{ |Rᵢ^*Rⱼ| \right\}_{∀Rᵢ ∈ V, ∀Rⱼ ∈ V} \]

3B. (for GME): Choose the join (Rᵢ, Rⱼ) from G such that
   \[ |Rᵢ^*Rⱼ| - |Rᵢ| - |Rⱼ| = \min \left\{ |Rᵢ^*Rⱼ| - |Rᵢ| - |Rⱼ| \right\}_{∀Rᵢ ∈ V, ∀Rⱼ ∈ V} \]

Following GMR, the resulting join sequence for the query in Fig. 2a is (((R₃, R₄), R₁), (R₁, R₅)), 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 GMC. This fact can be further justified by the simulation results in Section 3.3. It can be seen that the resulting join sequence by GME, where the expansion of a join operation is of concern, is also (((R₃, R₄), R₁), (R₅, R₆)) for the query in Fig. 2a. Moreover, to assess the performance of the heuristics, we implemented scheme GOPR to determine the optimal general join sequence for a multi-join query. Same as in Sopr, we enumerate possible candidate sequences in our implementation of GOPR and employ the technique of branch and bound to prune the search. Using GOPR, we obtain that the optimal general join sequence for the query in Fig. 2a is ((R₃, R₄), (((R₁, R₂), R₅), R₆)), with its bushy tree shown in Fig. 3d, requiring only a cost of 13,013.57, which in fact rather close to those obtained by GMC, GMR, and GME. 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 GMC, GMR, and GME, despite their simplicity, perform significantly better than SGD and Sopr, and result in join
sequences whose execution costs are reasonably close to that of the optimal one.

3.3. Simulation results and remarks

To perform the simulation, 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. The occurrence of an edge between two relations in the query graph is determined according to a given probability, denoted by \( p_{\text{prob}} \). Without loss of generality, only queries with connected query graphs are deemed valid and used for our study. The cardinalities of relations and attributes are also randomly generated within a reasonable range. The simulation program is coded in C. In that program, for each query the six scheduling schemes, i.e., SGD, SOPT, GMC, GMR, GM_m and GOPT, 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. The average execution cost for join sequences obtained from each scheme when \( p_{\text{prob}}=0.32 \) is shown in Table 2. Also, we divide the average execution costs of the first five schemes by that of GOPT for a comparison purpose, and show the results associated with Table 2 in Fig. 4.

<table>
<thead>
<tr>
<th>rel. no.</th>
<th>SGD</th>
<th>SOPT</th>
<th>GMC</th>
<th>GMR</th>
<th>GM_m</th>
<th>GOPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n=4 )</td>
<td>6905</td>
<td>6001</td>
<td>5112</td>
<td>5087</td>
<td>5098</td>
<td>4762</td>
</tr>
<tr>
<td>( n=6 )</td>
<td>23917</td>
<td>18135</td>
<td>14192</td>
<td>13927</td>
<td>13940</td>
<td>13141</td>
</tr>
<tr>
<td>( n=8 )</td>
<td>136505</td>
<td>74151</td>
<td>52085</td>
<td>46760</td>
<td>47187</td>
<td>42121</td>
</tr>
<tr>
<td>( n=10 )</td>
<td>606518</td>
<td>237223</td>
<td>112723</td>
<td>104311</td>
<td>107675</td>
<td>84121</td>
</tr>
</tbody>
</table>

Table 2. Average execution cost for various join sequences.

From Table 2 and Fig. 4, it can be seen that except for GOPT, the join sequence efficiency of join sequences obtained by GMC is the best among those obtained by the five remaining schemes, and then, in order, those by GMR, GM_m and SGD. The join sequence efficiencies of the sequences resulted by GMC, GMR and GM_m are quite close to the optimal one and significantly better than those of SGD and SOPT, especially when the number of relations increases. For the sizes of queries simulated here, the run times of SGD, GMC, GMR and GM_m under the DOS/PS2-80 environment are very close to one another whereas those of SOPT and GOPT are larger than them by more than three orders of magnitude due to their exponential complexity.

4. Processor Allocation for Each Join

The execution time required for a join operation within a multiprocessor system depends on the number of processors allocated to perform the join [10]. 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 \( y = 30 \) 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 \( p_{\text{m}} \), and the one which optimizes processor utilization, i.e., minimizes the product of the number of processors and the execution time, referred to as the best efficiency point, denoted by \( p_{\text{opt}} \). For example, \( p_{\text{b}} = 5 \) and \( p_{\text{m}} = 16 \) 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 for processor allocation. The heuristics proposed can be divided into two categories: (1) the bottom up approach in Section 4.1, and (2) the top down approach in Section 4.2.

4.1. Bottom up approach

(a) 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 inter-operator parallelism is absent when this heuristic is used, and the join sequence is the key factor to the performance in such a case.

(b) 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. 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.

(c) Minimum time point (MT): This heuristic is based on the minimum time point in the operational curve, i.e., \( p_{\text{m}} \). 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.
(d). 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, a scheme based on 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, 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, \(k^*p_B + (1-k)^*p_M\), is used to execute each join, 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 GMR, 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 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:**

\[
\text{\#} P = \text{\#} \text{processors available and initialized as the total numbers of processors.}
\]

1. **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.
2. **Step 2:** Allocate \(h_p(J)\) processors to execute the join \(J\). \(P := P - h_p(J)\).
3. **Step 3:** Update the profile by marking \(J\) as an ongoing join.
4. **Step 4:** Determine the completion time of \(J\) and record it in the completion time list of ongoing joins.
5. **Step 5:** Go to Step 1.

**Processor Deallocation:**

1. **Step 1:** From the completion time list, determine the next completion of an ongoing join, say \(J\).
2. **Step 2:** Update the profile to reflect that \(J\) is completed. \(P := P + h_p(J)\).
3. **Step 3:** If there is any executable join in the updated query profile, go to processor allocation.
4. **Step 4:** Go to Step 1.

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 3. For illustration, the operational curve of a join is approximated by the function,

\[
T_{\text{exe}} = \frac{a|R_i| + b|R_j| + c|R_i^*R_j^*|}{N_p} + dN_p
\]

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 [24], and parameter \(d\) is determined by the inter-processor communication protocol. Also, to each join we determine the minimal number of processors required for its execution according to the sizes of its operands, \(p_B\) is thus obtained. Note that the operational area where the number of processors is less than \(p_B\) can be ignored, since only the operational region \([p_B, p_M]\) is of interest in this study. Without loss of generality, \(G_{\text{SE}}\) is used to determine the next join operation to be executed. 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 4a, where the meaning of the column \(W(R_i)\) will be explained in Section 4.2. The bushy tree and its corresponding processor allocation by SE is shown in Fig. 7a. The execution scenarios using TE are shown in Table 63.
4b, where the time-efficiency point used is determined by $0.3p_0 + 0.7p_d$.

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

Note that when an execution tree is built bottom up, the following two constraints have to be followed: (1) execution dependency is observed, and (2) the processor requirement is satisfied according to the processor allocation heuristic employed. As can be seen in Tables 4a and 4b, the above two constraints lengthen the execution time of a query. Naturally, one 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 processor allocation which uses the concept of synchronous execution time to alleviate the two constraints and improve the query execution time.

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, the 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 cumulative execution cost 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,

$$W(R_i) = W(R_x) + W(R_y) + \text{cost}(R_x, R_y).$$

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 4a and 4b, 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,

$$P(R_x) = \frac{W(R_x)}{W(R_x) + W(R_y)} \cdot P(R_i)$$

and

$$P(R_y) = P(R_i) - P(R_x).$$

Since $W(R_i) = 0$ if $R_i$ 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 $p_m$, we shall employ $p_m$ processors to perform that join whereas using $r$ processors for the subsequent

---

*Table 4a. Execution sequence for heuristic SE.*

<table>
<thead>
<tr>
<th>join seq.</th>
<th>start time</th>
<th>end time</th>
<th>$W(R_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R_1, R_2)</td>
<td>0.0</td>
<td>308.9</td>
<td>104.4</td>
</tr>
<tr>
<td>(R_1, R_3)</td>
<td>0.0</td>
<td>345.2</td>
<td>122.6</td>
</tr>
<tr>
<td>(R_2, R_4)</td>
<td>0.0</td>
<td>378.5</td>
<td>139.2</td>
</tr>
<tr>
<td>(R_2, R_5)</td>
<td>0.0</td>
<td>381.8</td>
<td>140.9</td>
</tr>
<tr>
<td>(R_1, R_4)</td>
<td>378.5</td>
<td>1661.5</td>
<td>2027.8</td>
</tr>
<tr>
<td>(R_1, R_5)</td>
<td>1661.5</td>
<td>4002.1</td>
<td>76107.7</td>
</tr>
</tbody>
</table>

*Table 4b. Execution sequence for heuristic TE.*

<table>
<thead>
<tr>
<th>join seq.</th>
<th>start time</th>
<th>end time</th>
<th>$W(R_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R_1, R_2)</td>
<td>0.0</td>
<td>289.2</td>
<td>1044.5</td>
</tr>
<tr>
<td>(R_1, R_3)</td>
<td>289.2</td>
<td>625.3</td>
<td>1409.9</td>
</tr>
<tr>
<td>(R_1, R_4)</td>
<td>625.3</td>
<td>940.5</td>
<td>1226.3</td>
</tr>
<tr>
<td>(R_1, R_5)</td>
<td>940.5</td>
<td>1367.7</td>
<td>3497.8</td>
</tr>
<tr>
<td>(R_2, R_6)</td>
<td>1367.7</td>
<td>2307.7</td>
<td>12084.3</td>
</tr>
<tr>
<td>(R_1, R_6)</td>
<td>2307.7</td>
<td>4636.8</td>
<td>25681.0</td>
</tr>
<tr>
<td>(R_1, R_7)</td>
<td>4636.8</td>
<td>22553.3</td>
<td>565892.7</td>
</tr>
</tbody>
</table>

---

Figure 7. Bottom up processor allocation.

---

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

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 4a (by SE) is larger than that in Table 4b (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 4a and 4b. 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 of synchronous execution time to the bushy tree built by SE, denoted by STSE. For a comparison purpose, we also investigate the use of the synchronous execution time on the bushy tree built by TE, denoted by STTE.

The execution scenario using the heuristic STSE is shown in Table 4c, 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 STSE, 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 STSE outperforms SE despite both of them have the identical bushy trees and the same join sequence efficiency. It can be obtained that the execution time for STTE is 20557.2. The bushy tree and its processor allocation by STTE 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, STSE performs better than STTE, and in fact is the best one among the processor allocation heuristics evaluated in Section 4.3.

### 4.3. Simulation results and remarks

As in Section 3.3, 300 queries with a given number of relations involved were randomly generated. For each query, the six scheduling schemes, according to the heuristics of SE, FS, MT, TE, STSE and STTE, respectively, are performed to determine the number of processors for each join to execute the query, where the cluster size of FS is chosen to be half of the total processors. 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 Fig. 9. It can be seen that SE, i.e., the one using intra-operator parallelism only, performs well when the number of relations is 10, but performs worse when the number of relations increases. Also, 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 TE, especially when the number of relations is large. This can be explained by the reason that execution dependency is eased when the number of relations is large, and the execution time of each join thus becomes the primary factor in determining the execution time of a multi-join query.

Also, from 300 randomly generated queries, the average execution times obtained by the six heuristics for a query of 15 relations are shown in Fig. 10 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, SE suffers from the inefficient use of processors, and is thus outperformed by MT, TE, STSE and STTE 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. 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 intra-operator parallelism (i.e., SE) will suffice to provide a reasonably good performance. On the other hand, for a large number of relations, the cluster size of FS is chosen to be half of the total processors. For a multiprocessor of 48 nodes, the average execution times

Table 4c. Execution sequence for heuristic STSE.

<table>
<thead>
<tr>
<th>Join seq.</th>
<th>Start time</th>
<th>End time</th>
<th>W(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R1, R2)</td>
<td>0.0</td>
<td>289.2</td>
<td>1044.5</td>
</tr>
<tr>
<td>(R3, R4)</td>
<td>0.0</td>
<td>354.8</td>
<td>1409.9</td>
</tr>
<tr>
<td>(R5, R6)</td>
<td>0.0</td>
<td>315.1</td>
<td>1226.3</td>
</tr>
<tr>
<td>(R1, R2)</td>
<td>315.1</td>
<td>742.3</td>
<td>3497.8</td>
</tr>
<tr>
<td>(R3, R4)</td>
<td>289.2</td>
<td>1593.2</td>
<td>12084.3</td>
</tr>
<tr>
<td>(R5, R6)</td>
<td>742.3</td>
<td>1684.7</td>
<td>26961.0</td>
</tr>
<tr>
<td>(R1, R2)</td>
<td>1684.7</td>
<td>19101.2</td>
<td>565892.7</td>
</tr>
</tbody>
</table>

Figure 8. Top down processor allocation (synchronous execution time).
5. Conclusion

In this paper we dealt with two major issues to exploit inter-operator parallelism within a multi-join query: (i) join sequence scheduling and (ii) processor allocation. We explored the general join sequence so as to exploit the parallelism achievable in a multiprocessor system. 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 \(G_{OPF}\), 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. Also, we explored the issue of processor allocation. Several processor allocation heuristics, categorized by bottom up and top down approaches, were proposed and evaluated by simulation. Among all the schemes evaluated, the two-step approach by \(S_{TE}\), 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 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.

Acknowledgement

The authors would like to thank S. Lavenberg, H. Young, M. Lo and J. Chen at IBM for their helpful comments and technical assistance on improving this paper.

REFERENCES


Appendix

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

$$N_T(G_B) = \frac{\prod_{i=1}^{p} |R_i|}{\prod_{i=1}^{p} |A_i|^{m_i-1}}.$$