Optimizing Queries for Coarse Grain Parallelism

Sumit Ganguly  Weining Wang
Department of Computer Sciences
Rutgers University
New Brunswick, NJ 08904
e-mail: \{sumit,weinwang\}@cs.rutgers.edu
Tel: (908) 932-4974
Fax: (908) 932-0537

Abstract

We consider the problem of optimizing select-project-join relational queries for minimum response time on parallel machines. The design of the optimizer is based on three ideas: (1) the concept and quantification of degree of coarse grain parallelism for an execution tree, (2) the design of a parallelizing scheduler for a tree of coarse grain operations which is provably near optimal, and (3) the analysis of the scheduling algorithm to obtain a cost formula for parallel execution time. The search algorithm of the optimizer is presented as a multi-dimensional dynamic programming algorithm. We present two three-dimensional search algorithms for the case when placement of relations in the parallel machine do not overlap. We propose the tree placement strategy and demonstrate, by means of examples, how the number of dimensions in the search can be significantly reduced, thereby increasing the efficiency of the search algorithm.

1 Introduction

Parallel computers are increasingly becoming an economic reality. A seemingly obvious way of decreasing the response time of database queries is to run them on a parallel computer. For most parallel programs, there is a significant gap between the actual speedups obtained on a parallel machine and the potential speedup (factor of \(n\) on \(n\) processors). This is due to the non-trivial nature of the resource scheduling problem, inherent data dependencies in the problem and the communication overhead.

Current database systems allow users to present queries in a declarative fashion, using a high level language such as SQL. Consequently, an important operation of the database
systems is to translate such a high-level specification into an efficient query processing strategy. An important factor in the success of relational database technology in a sequential computing environment has been the successful design of these query optimizers.

However, the design of query optimizers for parallel machines has only recently received attention. An important experimental result due to Hong [9] and Stonebraker [8] is that the optimal plan for a shared everything parallel machine is the best parallelization of the optimal plan for a sequential machine. As discussed in [4, 5, 15], this result does not generalize to shared nothing parallel architectures.

Ganguly, Hasan and Krishnamurthy [5] present a framework for optimizing queries for parallel machines. They observe that optimizing for parallel executions requires extending the System R dynamic programming algorithms to work with more than one dimension. However, they do not present any specific practical method of designing the dimensions. Ganguly [4] presents a two dimensional search algorithm for optimizing in the presence of data dependencies without considering the problem of scheduling resources.

The approach taken by Srivastava and Elssesser [15] is interesting for shared nothing architectures. They observe that it is not possible to design a good optimizer that does not take the design of a resource scheduler into account creating an apparent circular dependence. They break this circular dependence, by an exhaustive solution, which, however, is computationally expensive. More work is needed to justify the optimality of their heuristic algorithm. Lu, Shan and Tan [11] present a greedy heuristic for optimization of multi-way joins. The optimality of the proposed method is not clear.

In this paper, we concentrate on the problem of optimizing the response time (also called parallel time) of a given select-project-join query of relational algebra. Our model of a parallel machine is a number of identical resource units connected by an interconnection network. Each resource unit may consist of one or more CPUs with memory and one or more disks. We assume a simple and widely accepted model of communication costs (cost of sending a message of length $M$ bytes is $\alpha + M\beta$, where $\alpha$ and $\beta$ are architecture-specific parameters). Relations are assumed to be horizontally fragmented with, possibly, more than one fragmentation. Our contributions are the following:

1. We quantify the concepts of coarse grain parallel computation of an execution tree and the degree of coarse grain parallelism for each operator in the execution tree. We present a local solution (i.e., without reference to other nodes in the tree) to the problem of estimating the coarse grain degree of parallelism for each operation in the execution tree. We also present a local sufficient condition for determining when an operation is coarse grain. All results in the paper assume that each operation in the tree is coarse grain according to our quantification.

2. We present a parallelizing scheduler that schedules operations in a coarse grain execution tree. For the class of executions considered, this scheduling algorithm is provably near optimal. The design of the algorithm is based on the rectangle packing algorithm.
presented in [1].

3. We analyze the scheduling algorithm to obtain two cost formulae (one more accurate than the other) for the parallel completion time of an execution tree. We use this formula to design a multi-dimensional dynamic programming search algorithm that finds the minimum response time plan among the space of coarse grain parallel executions. In a scenario where the relations do not overlap on resource units, this results in a 3-dimensional dynamic programming search algorithm. Our design solves the circular dependence problem referred to by Srivastava and Elsesser [15] without using an exhaustive search algorithm.

4. Overlap between the “home sites” of database relations increases the number of dimensions. We observe that symmetry in the overlap patterns can be exploited to keep the search algorithm efficient. We propose a novel placement strategy, called the tree placement strategy, and demonstrate, by means of examples, how the number of dimensions can be significantly reduced, for a given margin of error, thus increasing the efficiency of the search algorithm.

The rest of the paper is organized as follows. Section 2 discusses the execution space and Section 3 discusses our model of the parallel machine. Section 4 presents the concept of the degree of coarse grain parallelism. Section 5 presents the scheduling algorithm and Section 6 presents three dimensional dynamic programming search algorithms. Section 7 discusses the tree placement strategy and its ramifications. Finally, we conclude in Section 8.

2 Execution Space

In this section, we briefly discuss the execution space, review the notion of join trees and operator trees and state our assumptions.

We consider the space of left-deep join trees [14], where each node is annotated to represent a specific method of joining the operands and of accessing the relations. Each join operation may be a composition of several operations, for example, sorting and merging files, building a hash table and probing it etc. Given a join tree, an operator tree [4, 5, 8, 9, 12, 15] is an expansion of each node of the join tree into its constituent operations. Data dependency between adjacent operations in the operator tree is placed on the edges of the tree. We consider two kinds of data dependencies between operations: sequential and pipelined. Sequential dependency arises in various situations, for example, (a) when producer or a consumer is a sort operation, or (b) the producer is a build hash table operation and the consumer is a probe hash table operation etc. Pipeline dependency arises in situations such as (a) when producer is a merge operation and the consumer is a probe operation etc, or (b) the producer is a probe hash table and the consumer is the index scan of a nested
loop etc. Since properties of relational implementation operators are well known, it is easy to construct a table of data dependencies between all pairs of relational operators [4, 5].

*Single materialization on right subtree:* In this paper, we assume that for each right subtree in the operator tree, there is at most one sequential dependency.

All examples that require a materialization in the right subtree, for example, a sort operation or a build hash table or a build index operation, do not require another materialization before the binary join operation. We therefore believe that the assumption that there is at most a single materialization point in the right subtree of the operator tree includes most interesting executions in the space of left-deep join trees.

We assume that there is a centralized cost model (which is physically transparent[4]) that can estimate the time of computation of each of the operators in an operator tree and the sizes of the inputs and the outputs for each operator.

In this paper, we do not model memory explicitly and leave it to future work. Also, modeling network congestion and related issues such as network latency is left for future work.

3 A Model of Parallelism

In this section, we discuss our model of the parallel machine and our model of the cost of parallel computation.

We view the parallel machine as a collection of *identical resource units* interconnected by a communication network. An example of a resource unit could be one CPU, memory unit and a disk. Another example could be two CPUs, memory unit and one disk. We assume that each resource unit is the unit of scheduling, i.e., resources are handed out by the scheduler in quantum of resource units.

A parallel execution typically refers to the parallel execution of an operator tree. An operation signifies a node in the operator tree.

We first discuss the concepts of the processing area of a general parallel computation and parallel processing time of a single operation. We then discuss the concepts of communication area of a general parallel computation and the parallel communication time of a single operation. Finally, we discuss the concept of computation area of parallel execution and the parallel time of an operation.

3.1 Processing Area and Parallel Processing Time

In this section we present our model of parallel processing area and time.

Given an operator tree, let $A_{seq}$ denote the time required to execute the tree on one
resource unit. We define the processing area of a parallel execution of the tree as follows.

**Definition 1:** The *processing area* of a parallel execution is the sum of the computation time spent by each resource unit assigned to the parallel execution and is denoted by $A^p$.

*It should be emphasized that the computation area does not include communication overhead.* Hence it is reasonable to assume that $A^p \leq A_{seq}$. For simplicity, we assume that $A^p = A_{seq}$ for any parallel execution of an operator tree.\(^1\)

Let $a_{seq}$ denote the time required to execute an operation, i.e., a node of an execution tree on one resource unit. Let $pt^p(n)$ denote the time taken for the parallel execution of this operation on $n$ resource units. We model $pt^p(n)$ as

$$pt^p(n) = \frac{a_{seq}}{n}$$

### 3.2 Communication Area and Parallel Communication Time

In this section, we discuss the concepts of communication area and the parallel communication time.

We model the cost of communicating a message of $M$ bytes from one sender to one receiver as

$$Communication \ time = \alpha + M \times \beta$$

where

- $\alpha$ is the startup cost, measured in microseconds (for example, for nCUBE 2, $\alpha = 200$ microseconds [3]).
- $\beta$ is the time spent per unit of data sent (for example, for nCUBE 2, $\beta = 0.6$ microseconds per byte [3]).

Thus, the communication costs can be divided into two parts, the startup costs (the $\alpha$ component) and the message costs ($\beta$ component). Below we define the concept of communication area.

**Definition 2:** For a given parallel execution, the communication area, denoted by $A^c$ is defined as the sum of the communication time spent by each resource unit assigned to the parallel execution.

In parallel database processing, the startup cost of communicating $M$ bytes between a pair of operators depends on

\(^1\)The case $A^p < A_{seq}$ signifies a superlinear speedup and is possible since the increase in available memory in a parallel execution may reduce the total disk I/O costs. All results in this paper can be applied to the case of superlinear speedups with a slight modification [17].
1. The pattern of communication, whether one-to-few or all-to-all.

2. The nature of data dependency, whether sequential or pipelined.

We discuss each of the above items below.

**Dependence of Communication Costs on Pattern of Communication.** The two most frequent kinds of communication patterns in parallel database processing are:

1. *All-to-all.* The output of one operation has to be sent to the resource units computing the next operation after partitioning the data. Consider a sequence of two joins, the first one is parallelized by a partition on the "name" attribute whereas the second is parallelized using a partition on the "salary" attribute. Then, the output of the first join must be repartitioned (i.e., either hash partitioned, range partitioned etc.) and then sent over the network. We call this communication pattern all-to-all communication. If there are \( m \) senders and \( n \) receivers in an all-to-all communication, then the startup cost for each sender is proportional to \( an \) and the startup cost for each receiver is proportional to \( am \).

2. *One-to-few.* The output of one operation does not have to be repartitioned. For example, consider a sequence of two joins both of which use a partition on the "name" attribute. In this case, the repartition of the output of the first join may be unnecessary (depending on the expected skew in the result). We call this one-to-few communication. If there are \( m \) senders and \( n \) receivers in a one-to-few communication, then we assume that the startup cost for each sender and receiver is proportional to \( a \).

**Dependence of Communication Costs on Data Dependency.** The data dependency between operations also influences the startup costs. If data is pipelined between two operations scheduled on different sets of resource units, then messages from the sender cannot be pooled until the sender's execution is complete. Rather, a minimum of \( \lambda \) messages must be sent from any sender to a receiver. We assume that \( \lambda \geq 10 \), i.e., a message is sent after an average of 10% of the computation. This scenario is not imperative for sequential dependency.

Our model of startup cost for communication between \( m \) senders and \( n \) receivers is summarized in the table given in Figure 1.

### 3.3 Computing Area and Parallel Computing Time

In this section, we discuss the concepts of the computation area of a parallel execution and parallel time.

**Definition 3:** The area of a parallel computation, denoted by \( A \), is defined as the sum of the computation area, \( A^p \) and the communication area, \( A^c \).
<table>
<thead>
<tr>
<th>Pattern of Communication</th>
<th>Pipelined</th>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost per Sender</td>
<td>Cost per Sender</td>
</tr>
<tr>
<td></td>
<td>Cost per Receiver</td>
<td>Cost per Receiver</td>
</tr>
<tr>
<td>All-to-all</td>
<td>$\lambda n$</td>
<td>$\lambda n$</td>
</tr>
<tr>
<td>One-to-few</td>
<td>$\lambda \alpha$</td>
<td>$\alpha$</td>
</tr>
</tbody>
</table>

Figure 1: Startup costs for communication

\[
\text{Computation Area} = \text{Processing Area} + \text{Communication area}
\]

or

\[
A = A^p + A^c
\]

**Definition 4:** The parallel time of a parallel execution, denoted by $PT$ is the time required to complete the parallel execution. □

The parallel time of the parallel execution of a single operation using $n$ resource units is denoted by $pt(n)$ and is modeled as

\[
pt(n) = pt^p(n) + pt^c(n)
\]

## 4 Degree of Coarse Grain Parallelization

In this section, we discuss our concept of coarse grain parallelism.

Consider the performance of a typical operation as a function of the number of resource units $n$ used to execute it. Let us trace the speedup as a function of $n$, where speedup is defined as the ratio of the time taken to finish the operation on 1 resource unit to the time taken to finish the operation on $n$ resource units. When $n$ is small, the speedup function for most operations, and particularly for joins, is linear and is close to $n$. As $n$ increases, the speedup function becomes non-linear although monotonically increasing. However, if $n$ is still increased, then communication costs overwhelm computation costs and there is typically a speed down. Hence, the number of resource units that are assigned to an operation should be such that the communication costs do not overwhelm the computation costs. We call such a computation coarse grain and define it below.

**Definition 5:** A parallel execution of set of operations is said to be *coarse grain* (with parameter $f$), if the communication area is no more than $f$ times the processing area, i.e., $A^c \leq f \times A^p$. □
It should be emphasized that the concept of coarse grain parallelism compares “communication area” costs with sequential computing cost. It does not directly relate parallel communication time to parallel computation time.

The parameter of coarse grain computation is closely related to the efficiency [10] of a parallel computation, which is defined below.

**Definition 6:** The efficiency \( \eta \) of a parallel computation is defined as the ratio of the sequential computation time to the computation area of the parallel execution, i.e.,

\[
\eta = \frac{A_{seq}}{A}
\]

Consider a coarse grain parallel computation with parameter \( f \) on \( n \) resource units. Then

\[
\eta = \frac{A_{seq}}{A} = \frac{A^p}{A^p + A^c} \geq \frac{A^p}{A^p + f \times A^p} = \frac{1}{1 + f}
\]

Hence, the relation between the efficiency \( \eta \) and the parameter \( f \) of coarse grain parallel computation is:

\[
\eta \geq \frac{1}{1 + f}
\]

Parallel computations with good efficiency are desirable. We hypothesize that \( f \) should be small, i.e., \( 0 \leq f \leq 1 \), which gives \( 0.5 \leq \eta \leq 1 \).

**Definition 7:** The degree of parallelism \( \delta \) of an operation is the number of resource units assigned to the operation in a parallel execution. \( \square \)

Given an operator tree, it is an important to obtain the degree of parallelism for each operator such that the parallel execution is coarse grain. For a given coarse-grain parallel execution, let \( \alpha S_i \) be the total startup component of the communication overhead at node \( i \) in the operator tree. Let \( I_i \) be the sum of the input and output sizes for communication at node \( i \). We assume that each operation sends its output to and receives its input from the interconnection network. ²

Then, the total communication cost = \( \alpha S_i + \beta I_i \). Of the total communication cost, \( \beta I_i \) is independent of the degree of parallelism for node \( i \) or its predecessors or successors. For each node \( i \), let

\[
B_i = \frac{f \times A^p_i - \beta I_i}{\alpha} \quad \ldots (1)
\]

²Clustering adjacent operators will be considered in a future work. It is beneficial to cluster adjacent operations that have a one-to-few communication pattern and a sequential dependency, whereas it is not beneficial to cluster adjacent operations that have an all-to-all communication pattern.
Figure 2: Examples for start up cost expression

where $f$ is the parameter of coarse granularity. Then the condition for coarse grain parallel execution of the operator tree with $m$ nodes may be stated as

$$\sum_{i=1}^{m} S_i \leq \sum_{i=1}^{m} B_i$$

Let $\alpha S_i$ denote the startup cost of the parallel execution at node $i$ and $\delta_i$ denote the degree of parallelism for the node $i$. Given the operator tree and the values for $B_i$ at each node $i$, an expression for $S_i$ can be obtained from the table in Figure 1. We illustrate this by means of an example.

**Example 1:** Consider Figure 2 in which node $i$ is connected to other nodes in three different ways as shown and with different patterns of communication and data dependency. Consider Figure 2(a). Node $i$ participates as a receiver in an all-to-all communication from node $i-1$ with a sequential dependency with the sender. Hence the startup cost for each resource unit assigned to node $i$ is $\alpha \delta_{i-1}$ from the table in Figure 1. Hence, the total startup cost for this communication = $\alpha \delta_{i-1} \delta_i$.

Node $i$ also participates as a sender in a one-to-few communication with node $i-1$. Hence the total start up cost for this communication is given by $\alpha \delta_i \delta_{i+1}$. The total start up cost is the sum of the startup costs for every communication that node $i$ participates in. Hence, the expression for $S_i$ is given by $\delta_i \delta_{i+1} + \delta_i \delta_{i-1}$. (Note that start up cost is defined as $\alpha S_i$).

The expressions for $S_i$ for Figure 2(b) and (c) may be obtained similarly are shown below.

1. $S_i = \lambda \delta_i \delta_{i-1} + \delta_i$, for Figure 2(b).
2. \( S_i = \delta_{i-1} \delta_i + \lambda \delta_{i-2} \delta_i + \lambda \delta_i \), for Figure 2(c). \( \square \)

Given an operator tree, the startup cost expressions for each node \( i \) may be derived. Coarse granularity at each node may be enforced using the constraint

\[
S_i \leq B_i \quad \text{for } 1 \leq i \leq m
\]

However, such a set of inequations is not desirable for three reasons:

1. An objective function to test whether one solution is better than another is not easily formulated.

2. The expressions for \( S_i \) are non-linear and therefore computationally expensive to solve, even if an objective function is designed.

3. We are interested in the coarse granularity of the entire computation, and not necessarily for each node in the tree.

**Local Solution to Coarse Grain Parallelism**

We now present a solution for each \( \delta_i \) such that the overall computation is coarse grain. Suppose that the expression for \( S_i \) has \( q_i \) quadratic terms and \( l_i \) linear terms. Without loss of generality, we rearrange the terms so that all the quadratic terms precede the linear terms, i.e.,

\[
S_i = \sum_{j=1}^{q_i} a_{ij} \delta_j \delta_i + \sum_{j=q_i+1}^{q_i+l_i} a_{ij} \delta_i
\]

We represent the solution of \( \delta_i \), for all \( i \) by \( \hat{\delta}_i \), which is obtained as follows.

\[
\hat{\delta}_i = \min(\{ \hat{x}_i B_i \}, P)
\]

where \( P \) is the number of resource units and \( \hat{x}_i \) is the unique positive root (which always exists) of the quadratic equation

\[
B_i (\sum_{j=1}^{q_i} a_{ij}) x_i^2 + \sum_{j=q_i+1}^{q_i+l_i} a_{ij} x_i = 1 \quad \ldots (3)
\]

If \( \hat{\delta}_i = 0 \), then we consider the operation to be fine grain. A sufficient condition for the coarse granularity of a tree is given by \( \hat{\delta}_i \geq 1 \), which is equivalent to the following. Note that the solution for \( \hat{\delta}_i \) is local, i.e., it depends only on the characteristics of node \( i \).

**Local Condition for Coarse Granularity:** A tree is coarse grain if each node \( i \) satisfies the following property.

\[
\sum_{j=1}^{q_i+l_i} a_{ij} \leq B_i \quad \ldots (4)
\]
Theorem 1 proves that the choice of \( \delta_i = \hat{\delta}_i \), for each node in the tree, ensures that the computation is coarse-grain. Again, note that the solution is local.

**Theorem 1** Suppose that each node in the tree satisfies the local condition for coarse granularity. Then, the choice of \( \delta_i = \hat{\delta}_i \), for each node in the tree implies that the computation of the entire tree is coarse grain.

**Proof:** Suppose that there are \( m \) nodes in the tree numbered from 1 to \( m \).

\[
\sum_{i=1}^{m} S_i = \sum_{i=1}^{m} \left( \sum_{j=1}^{q_i} a_{ij} \hat{\delta}_i \hat{\delta}_{ij} + \sum_{j=q_i+1}^{q_i+l_i} \hat{\delta}_i a_{ij} \right)
\]

\[
\leq \sum_{i=1}^{m} \left( \sum_{j=1}^{q_i} \hat{\delta}_i^2 a_{ij} + \sum_{j=q_i+1}^{q_i+l_i} a_{ij} \hat{\delta}_i \right)
\]

For every term of the kind \( \hat{\delta}_i \hat{\delta}_{ij} \), there is an identical term in the expression for \( S_{ij} \). Hence the above expression is equal to

\[
= \sum_{i=1}^{m} \hat{\delta}_i^2 \left( \sum_{i=1}^{q_i} a_{ij} \right) + \hat{\delta}_i \sum_{j=q_i+1}^{q_i+l_i} a_{ij}
\]

Since \( \hat{\delta}_i \leq x_i B_i \) and \( x_i \) is a root of (3), it follows that the above expression

\[
\leq \sum_{i=1}^{m} B_i
\]

Hence

\[
\sum_{i=1}^{m} S_i \leq \sum_{i=1}^{m} \frac{f \times A_i^p - \beta B_i}{\alpha}
\]

or

\[
\sum_{i=1}^{m} (\alpha S_i + \beta B_i) \leq \sum_{i=1}^{m} f \times A_i^p
\]

It remains to be shown that \( \hat{\delta}_i \geq 1 \), for each node \( i \). In order to do so, it is sufficient to show that \( \hat{x}_i B_i \geq 1 \). Let \( h_i(x_i) \) denote the following function.

\[
h_i(x_i) = B_i \left( \sum_{j=1}^{q_i} a_{ij} x_i^2 + \sum_{j=q_i+1}^{q_i+l_i} a_{ij} x_i \right) - 1
\]

It follows that \( h_i(x_i) \) has a unique positive root and is a monotonic, increasing and unbounded function in the range \( x_i \geq 0 \). The local condition for coarse-grain computation may be rewritten as

\[
h_i \left( \frac{1}{B_i} \right) \leq 0
\]
Hence, there is a root $\hat{x}_i$ of $h_i$ which is no less than $\frac{1}{B_i}$, i.e., $\hat{x}_i B_i \geq 1$. \qed

We say that a tree of operations is \textit{locally coarse grain} if each node in the tree satisfies the local condition for coarse granularity presented in inequation (4). The following sections that present the design of scheduler and search algorithm assume that the tree is locally coarse grain. Note that the concept of coarse granularity, degree of parallelism and the proof of Theorem 1 do not assume anything specific about trees and can be applied to general computation graphs.

5  Scheduling Algorithm

In this section, we present a simple algorithm for scheduling a tree of operators on the parallel machine. This scheduling algorithm approximates the performance of the optimum algorithm to within a reasonable factor. Further, this algorithm presents us with a formula for the completion time of an operator tree, which is used in the design of the optimization algorithm, presented in Section 6.

We first describe our model of a scheduler and then present the scheduling algorithm. The scheduling algorithm is first described for independent and pipelined operations and then for a tree of operations.

5.1  Model of Scheduler

We assume that individual resource units of the parallel machine are multi-threaded and a scheduler can dispatch more than one operation to a resource unit. Once an operation is dispatched to a resource unit it is not migrated to another resource unit. For simplicity, we ignore the cost of context switch between concurrently running threads on the same resource unit of a parallel machine. Following this model, two long duration operations should be preferably assigned to different resource units. Hence there is a need for an intelligent assignment of operations to resource units.

Consider a set of pipelined and independent operations. Given an assignment of operations to resource units, the scheduler dispatches each of the operations to the resource units it is assigned to. Each operation runs immediately, thus providing us with a schedule corresponding to the resource assignment. The parallel time of this schedule, which is the earliest time of completion of all the operations, is no worse than the parallel time of a schedule with the same resource assignment in which operations are not multi-threaded. This follows from our simplifying assumption that the cost of context switch between threads can be ignored.

For tree scheduling, the scheduler dispatches each operator to the resource units allocated for each operation. The operation is \textit{executed as soon as its input becomes available}, i.e., following the dataflow paradigm. As discussed above, a resource assignment for each operator in the tree results in a schedule whose parallel time is no worse than the parallel
time of a schedule with the same resource assignment in which operations do not time-share the resource units.

5.2 Rooted and Floating Operations

In this section, we classify operations into two classes, rooted and floating.

There are two kinds of operations in an operator tree:

1. Rooted Operations. These are the scan (indexscan etc.) operators which must use a certain given set of resource units (the roots of the operation). In other words, resource units cannot be arbitrarily assigned to such operations.

2. Floating operations. Operations that are not rooted are called floating. These operations are candidates for resource assignment.

For simplicity, the scheduling algorithms are presented for the case when an operator tree consists of floating operations only. However, the algorithms and their properties remain unchanged when the tree consists of rooted operations [17].

5.3 Scheduling Independent and Pipelined Operators

In this section, we describe an algorithm that assigns resources to a set of independent and pipelined operations.

Figure 4 presents Algorithm 1 that assigns resources to a set of independent or pipelined operations $T_i$, $1 \leq i \leq nops$. Each operation is characterized by four fields, namely, (1) oid for the operation id, (2) degree, which is equal to the degree of parallelism $\delta_i$, (3) area, which is equal to $a_i$, i.e., the parallel computing area when the operation is parallelized using $\delta_i$ resource units, and (4) time, which is equal to $area/degree = a_i/\delta_i$. The algorithm uses the procedure $assign\_operation(oid, rid, np)$ that assigns resource units $rid, rid + 1, \ldots, rid + np - 1$ to operation specified by oid. The algorithm is an implementation of the following rule:

Next Fit Decreasing Height scheduling rule: [1] Choose the operation, $T$, with the largest completion time (i.e., $a_i(\delta_i)/\delta_i$). If there are $\delta_T$ resource units available, then assign the first $\delta_T$ resource units to $T$ and repeat the rule. Otherwise, wait until all the operators currently running have completed and repeat the rule until all operations have been scheduled. ❑

We show an example of the schedule produced by the algorithm below. This example also shows how the scheduling algorithm can be improved to give shorter schedules in the average case. The improved algorithms are discussed in [17].
**Example 2:** Consider the problem of scheduling 6 independent tasks into 10 resource units, where the parallel computing area and the degree of parallelism of each task is tabulated below.

<table>
<thead>
<tr>
<th>Task</th>
<th>$\delta$</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>T3</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>T5</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>T2</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>T4</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>T6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 3(a) shows the schedule generated by Algorithm 1. The parallel time of the schedule is 19 units.

![Schedule Example](image.png)

Figure 3: An Schedule Example of Independent Tasks

The schedule chart in Figure 3(a) has some gaps which can be reduced using one of two techniques: (1) increase the parallel time of an operation by using fewer number of processors than the degree of parallelism, thus creating free resource units and (2) by assigning tasks with smaller areas into idle resource units without increasing the parallel time. A better schedule, with a parallel time of 15 units, for the same problem is shown in Figure 3(b).

Note that the algorithm proceeds by constructing a schedule by assigning at most one operation to any resource unit at any time (single-threaded execution). Our assumption of multi-threaded resource units implies that the parallel time of the multi-threaded version of this schedule is no worse than the parallel time of this schedule.

The following theorem states that the parallel time of the schedule generated by this algorithm is within a factor of $3 + 2f$ of the parallel time of the optimum schedule.

**Theorem 2** Let $PT_s$ be the parallel time of the schedule generated by Algorithm 1 and let
Input: ready[] is an array of operations to be scheduled.
nops is the number of operations in the array.
P is the number of resource units.
Output: A resource assignment for each operation in ready[].

int freep; /* number of free resource units left */
int firstp; /* first available resource unit*/
begin
    sort ready[] in decreasing order of the time field.
    \( j = 1; \)
    \( freep = P; \)
    \( firstp = 1; \)
    \( u = 0; \)
    while \( (j \leq nops) \) {
        \( u = u + 1; \)
        assign_operation(ready[j].oid, firstp, ready[j].degree);
        \( firstp = firstp + ready[j].degree; \)
        \( freep = freep - ready[j].degree; \)
        \( i = i + 1; \)
        while ((i \leq nops) and ready[j].degree < freep) {
            assign_operation(ready[j].oid, firstp, ready[j].degree);
            \( firstp = firstp + ready[j].degree; \)
            \( freep = freep - ready[j].degree; \)
            \( j = j + 1; \)
        }
    }
end

Figure 4: Algorithm 1: Scheduling independent and pipelined tasks
$PT_{\text{optimum}}$ be the parallel time of any schedule. Then

$$PT_s \leq (3 + 2f - \frac{1}{P})PT_{\text{optimum}}$$

where $f$ is the parameter of coarse grain computation.

**Proof:** Let $0 \ldots l$ be the values assumed by the variable $u$ in Algorithm 1. Let $S_i$ be the set of operators assigned to resource units when program variable $u = i$. Let $A_i$ be the parallel computation area of the operators in $S_i$ and $T_i$ be the largest completion time among the completion times of all the operators in $S_i$.

Then, for $1 \leq i \leq l - 1$,

$$A_i + A_{i+1} \geq P \times T_{i+1}$$

$$\sum_{i=1}^{l-1} A_i + \sum_{i=1}^{l-1} A_{i+1} \geq P \sum_{i=1}^{l-1} T_{i+1}$$

or

$$2A - A_i - A_1 \geq P(PT_s - T_1)$$

where $A =$ total area of the operators in the schedule, including the communication costs.

$$PT_s \leq T_1 + \frac{2A - A_1 - A_1}{P} \leq T_1 + \frac{2A - A_1}{P} \leq T_1 + \frac{2A - T_1}{P} = T_1(1 - \frac{1}{P}) + \frac{2A}{P}$$

Let $A_{seq}$ denote the sequential computing time. Since the parallel execution is coarse grain, it follows that

$$A \leq (1 + f)A_{seq}$$

where $A_{seq}$ denotes sequential computing time. Let $PT_{\text{optimum}}$ and $A_{\text{optimum}}$ denote the parallel time and the parallel computing area of the optimum parallel time schedule.

$$A_{\text{optimum}} \geq A_{seq}$$

hence

$$PT_{\text{optimum}} \geq \frac{A_{\text{optimum}}}{P} \geq \frac{A_{seq}}{P} \geq \frac{A}{P(1 + f)}$$

Also

$$PT_{\text{optimum}} \geq T_1$$

Hence,

$$PT_s \leq (3 + 2f - \frac{1}{P})PT_{\text{optimum}}$$
5.4 Scheduling Tree of Operators

In this section, we consider the problem of assigning resources to a given operator tree.

We restrict our attention to operator trees that have at most one sequential dependency on any right subtree. The scheduling algorithm proceeds in two stages. In the first stage, all adjacent piped operations are aggregated together to obtain the aggregated operator tree. Then operations of the aggregated operator tree are scheduled using a variant of Algorithm 1.

An outline of the aggregation algorithm is presented in Figure 6. Each vertex in the aggregate tree is a set of vertices in the operator tree which consists of a maximal pipeline. An example operator tree and its corresponding aggregate tree is shown in Figure 5.

Figure 7 presents the tree scheduling algorithm. Informally, the algorithm is an implementation of the following rule:

Tree Scheduling Rule: Find the set of ready operations. Choose the operation, $T$, with the largest completion time (i.e., $a_i(\delta_i)/\delta_i$). If there are $\delta_T$ resource units available, then assign the first $\delta_T$ resource units to $T$ and repeat the rule. Otherwise, wait until all the operators currently running have completed and repeat the rule until all operations are scheduled.

The algorithm makes use of the function get_ready_operations that returns an array of ready operations sorted in decreasing order of the completion time. It also returns the number of ready operations. The function delete_operation(oid) deletes the operation oid
Input: An operator tree.

Output: An aggregate operator tree.

1. Create an aggregate vertex \{x\} for each vertex \(x\) in the operator tree.

2. If \(x \in X\) and \(y \in Y\) and there is a pipeline dependency between \(x\) and \(y\) in the operator tree, then combine the aggregate vertices \(X\) and \(Y\) (i.e., delete the vertices \(X\) and \(Y\) and insert a vertex \(X \cup Y\)). Repeat this step until there is no change. This gives the vertex set of the aggregate tree.

3. If \(x \in X\) and \(y \in Y\) and there is a sequential dependency from \(x\) (child) to \(y\) (parent), then introduce an edge and a sequential dependency from \(X\) (child) to \(Y\) (parent). Repeat this step until there is no change.

Figure 6: Algorithm 2: Outline of an algorithm to generate the aggregate tree corresponding to an operator tree from the aggregate tree.

Analogous to the discussion in Section 5.3, we argue that the tree scheduling algorithm assigns operations to resource units by constructing a schedule in a single-threaded context. The parallel time of this schedule is no less than the parallel time of the corresponding multi-threaded version, where each operator runs as soon as it is ready. The following theorem states that the parallel time of the schedule generated by the tree scheduling algorithm is within a factor of \((4 + 2f - 2/P)\) of the length of the optimum schedule, \(PT_{\text{optimum}}\).

**Theorem 3** Given an operator tree with at most one sequential dependency in the right subtree, let \(PT^*_s\) be the parallel time of the schedule generated by the tree scheduling algorithm for the tree. Let \(PT_{\text{optimum}}\) be the parallel time of the optimum schedule for the tree. Then

\[
PT_s \leq (4 + 2f - \frac{2}{P}) \times PT_{\text{optimum}}
\]

**Proof:** Let \(0 \ldots l\) be the range of values assumed by the variable \(u\) in the tree scheduling algorithm. Let \(S_i\) be the set of operators assigned to resource units when \(u = i\). If an operator is assigned resources when program variable \(u = i\), then we say that the operator is assigned at stage \(i\). Let \(A_i\) be the parallel computing area of operators in \(S_i\) and \(T_i\) be the largest completion time among the completion times of all the operators in \(S_i\). Let \(g_0 = 1, g_1, \ldots, g_m\) be the values of \(i\) such that \(A_{i-1} + A_i < P \times T_i\). Let \(U_k\) denote the set of operations scheduled when the program variable \(u\) takes values \(\geq g_{k-1}\) and \(< g_k\). Let \(PT_k\) denote the parallel time of \(U_k\), i.e.,

\[
PT_k = \sum_{i=g_{k-1}}^{g_k-1} T_i
\]

18
Input: An aggregate operator tree and the number of resource units P.
Output: A resource assignment for each operation in the corresponding operator tree.

begin
  nready = get.ready.operations(ready);
  u = 0;
  firstp = 1;
  freep = 1;
  while (nready > 0) {
    j = 1;
    u = u + 1;
    assign.operation(ready[1].oid, firstp, ready[1].degree);
    firstp = firstp + ready[1].degree;
    freep = freep - ready[1].degree;
    j = j + 1;
    while (j < nready) and (ready[j].degree < freep) {
      assign.operation(ready[j].oid, firstp, ready[j].degree);
      firstp = firstp + ready[j].degree;
      freep = freep - ready[j].degree;
      delete.operation(ready[j].oid);
    }  // end while
    j = j + 1;
  }  // end while
  nready = get.ready.operations(ready);
end.

Figure 7: Algorithm 3: Tree Scheduling Algorithm
Let $A'_k$ denote the sum of the parallel computing areas of the operations in $U_k$.

Following the proof of Theorem 2

$$PT_k \leq T_{g_{k-1}}(1 - \frac{1}{P}) + 2\frac{A'_k}{P}$$

Hence

$$PT_s = \sum_{k=0}^{m} PT_k \leq (1 - \frac{1}{P}) \sum_{k=0}^{m} T_{g_k} + 2\frac{A}{P}$$

where $A$ is the total parallel computing area of all the operators in the tree. Since $PT \geq (A/P)(1 + f)$, it follows that

$$PT_s \leq (1 - \frac{1}{P}) \sum_{k=0}^{m} T_{g_k} + (2 + 2f)PT$$

There are two possibilities:

1. $T_{g_k} > T_{g_{k-1}}$. In this case, the operation scheduled in stage $g_k$ with completion time $T_{g_k}$ could not have been scheduled in stage $g_k - 1$. Hence, this operation, say $O_k$, depends sequentially on some operation that is scheduled in stage $g_k - 1$. Hence, $O$ belongs to an internal vertex of the aggregate tree, say $V_k$, none of whose vertices are scheduled before stage $g_k$.

2. $T_{g_k} \leq T_{g_{k-1}}$. Let $O_k$ be the operation scheduled in stage $g_k$ with completion time $T_{g_k}$. Recall that the numbers $g_k$ is defined such that $A_{g_k-1} + A_{g_k} < P\times T_{g_k}$. We can conclude that the scheduling algorithm is unable to schedule $O_k$ in stage $g_k - 1$ although there are enough processors left in stage $g_k - 1$. Hence, there is data dependency between $O_k$ and some operation that is scheduled in stage $g_k - 1$. Hence, $O_k$ belongs to an internal vertex $V$, none of whose vertices are scheduled before stage $g_k$.

In either case, for each $g_k$, there exists an internal vertex which is scheduled at stage $g_k$ and not before. Hence, with each $g_k$, we can associate a unique internal vertex. Let $L$ denote the sum of the maximum parallel operations in each internal vertex of the aggregate tree. Then,

$$\sum_{k=0}^{m} T_{g_k} \leq T_1 + L$$

Hence

$$PT_s \leq (1 - \frac{1}{P})(T_1 + L) + 2\frac{A}{P} \quad \cdots (5)$$

Clearly, $T_1 \leq PT_{\text{optimum}}$, $L \leq PT_{\text{optimum}}$ and $A/P \leq PT_{\text{optimum}}(1 + f)$. Hence,

$$PT_s \leq 2(1 - \frac{1}{P})PT_{\text{optimum}} + (2 + 2f)PT_{\text{optimum}}$$

or, $$PT_s \leq (4 + 2f - 2\frac{1}{P})PT_{\text{optimum}} \quad \Box$$

20
6 Search Algorithm

In this section, we design a multi-dimensional dynamic programming search algorithm (multi-dimensional dynamic programming is called partial order dynamic programming in [5, 4]). Given a query, the search algorithm finds the plan with the least value of parallel time among the space of all locally coarse grain parallel executions. Recall that a plan is locally coarse grain, if each node of the corresponding operator tree satisfies inequation (4), which specifies the local condition for coarse granularity.

We first design a formula for estimating the parallel time of an execution tree by analyzing the tree scheduling algorithm. We then present three dimensional search algorithms.

6.1 Cost Formula for Parallel Time

In this section, we present two cost formulae for estimating the parallel time of an operator tree.

Equation (5) gives us an approximate formula for estimating the parallel time of a tree which is rewritten below.

\[ EPT \equiv (1 - \frac{1}{P})(M + L) + 2\frac{A}{P} \ldots (6) \]

where

- \( M \) is the largest parallel time among all leaves in the aggregate tree.
- \( L \) is the sum of the largest parallel time of each internal node in the aggregate tree.
- \( A \) is the parallel computing area of the tree, and
- \( P \) is the number of resource units.

Clearly, for any scheduling algorithm \( A \) that is no worse than the tree scheduling algorithm (i.e., \( PT_A(T) \leq PT_s(T) \) for all trees \( T \))

\[ PT_A(T) \leq EPT(T) \leq (4 + 2f - \frac{2}{P})PT_A(T) \]

justifying the "goodness" of \( EPT \) as an estimate of parallel time.

An approximation of \( M + L \) in the equation defining \( EPT \) is the length of the longest critical path in the aggregate tree, denoted by \( C \). This gives the following heuristic for the estimation of parallel time

\[ CPT \equiv (1 - \frac{1}{P})C + 2\frac{A}{P} \ldots (7) \]
6.2 Partial Orders for Dynamic Programming

In this section, we design partial orders which can be used to optimize for the minimum parallel time using the definition of \( EPT \) or \( CPT \). In this section, we assume that the resource units on which database relations reside do not overlap. The case when there is an overlap is considered in Section 7.

Let \( e \) be an extension of the operator tree, i.e., a sequence of nodes with exactly one node in the spine of the operator tree. Figure 8 illustrates the notation. Let \( T \) be an operator tree. Let \( A(e) \) and \( A(T) \) be the area of operators in \( e \) and \( T \) respectively. Let \( T \circ e \) denote the new tree formed by extending \( T \) with \( e \). Then,

\[
A(T \circ e) = A(T) + A(e)
\]

In order to simplify the presentation of the metrics, we define the parallel time of a vertex of the aggregate tree and the length of a path in an aggregate tree.

**Definition 8:** The parallel time of a vertex in an aggregate tree is the maximum parallel time among all the vertices of the operator tree that are included in the aggregate vertex. The length of a path in the aggregate tree is the sum of the parallel times of the vertices in the path. □

Recall that the operator tree has at most one materialization in the right subtree which implies that the corresponding aggregate tree has a unique spine (a sequence of internal nodes). We define the following metrics needed for formalizing the partial order for dynamic programming search. Let \( T \) be an aggregate operator tree and \( T_{agg} \) be the aggregate tree corresponding to \( T \).
1. Let \( \text{int}(T) \) denote the length of the unique path in \( T_{agg} \) consisting of internal nodes only without including the root. Let \( \text{int}_A(T) = (1 - 1/P)\text{int}(T) + (2/P)A(T) \).

2. Let \( r(T) \) denote the parallel time of the root of \( T_{agg} \).

3. Let \( \text{Int}(T) \) denote the length of the path consisting of internal nodes in \( T_{agg} \).

4. Let \( M(T) \) denote the largest parallel time of the set of leaf nodes in \( T_{agg} \).

5. Let \( C(T) \) denote the length of the critical path in \( T_{agg} \).

6. Let \( c(T) \) denote the length of the fragment of critical path in \( T_{agg} \) that excludes the root.

Clearly, \( \text{Int}(T) = r(T) + \text{int}(T) \) and \( C(T) = r(T) + c(T) \).

The definitions for each of the metrics \( \text{int}, r \) etc. apply equally well to an extension \( e \) since an extension is a sequence, which is a special case of a tree.

The inductive equations for computing the metrics are given below. There are two cases, depending on whether the data dependency \( d \) between the root of the tree and the root of the extension is sequential or pipelined.

- \( d \) is sequential.

\[
\begin{align*}
c(T \circ e) &= \max(c(T) + r(T), c(e)) \\
\text{int}_A(T \circ e) &= \text{int}_A(T) + (1 - 1/P)r(T) + (2/P)A(e) \\
r(T \circ e) &= r(e) \\
M(T \circ e) &= \max(M(T), \text{int}(e))
\end{align*}
\]

- \( d \) is pipelined

\[
\begin{align*}
c(T \circ e) &= \max(c(T), c(e)) \\
\text{int}_A(T \circ e) &= \text{int}_A(T) + 2/PA(e) \\
r(T \circ e) &= \max(r(T), r(e)) \\
M(T \circ e) &= \max(M(T), \text{int}(e))
\end{align*}
\]

Equations (6) and (7) may be rewritten as

\[
EPT = \text{int}_A(T) + (1 - \frac{1}{P})(r + M)
\]

\[
CPT = 2\frac{A}{P} + (1 - \frac{1}{P})(c + r)
\]

The partial orders for optimizing \( EPT \) and \( CPT \) are given below. Let \( p_1 \) and \( p_2 \) be two plans for the same subquery. The notation \( \text{int}_A(p_1) \), denotes \( \text{int}(T) \), where \( T \) is the operator tree corresponding to \( p_1 \) etc. The meanings of \( r(p_1), M(p_1), M(p_2) \) etc. are similar.

Let \( p_1 \) and \( p_2 \) be plans for the same subquery.
• **Partial order for EPT:** $p_1 \leq_{EPT} p_2$ if

$$int_A(p_1) \leq int_A(p_2) \text{ and } r(p_1) \leq r(p_2) \text{ and } M(p_1) \leq M(p_2).$$

Since there are three conjuncts in the partial order, we say that the corresponding dynamic programming algorithm is a three-dimensional search algorithm.

• **Partial Order for CPT:** $p_1 \leq_{CPT} p_2$ if

$$c(p_1) \leq c(p_2) \text{ and } r(p_1) \leq r(p_2) \text{ and } A(p_1) \leq A(p_2).$$

Since there are three conjuncts in the partial order, we say that the corresponding dynamic programming search algorithm is three-dimensional.

Note that each of the inductive equations is monotonic in their parameters. Hence, the partial orders $\leq_{CPT}$ and $\leq_{EPT}$ satisfy the principle of optimality.

### 7 Symmetry in Placement Strategies

The justification of the correctness of the partial order metrics in Section 6 is based on inductive equations for metrics $int, r$ etc. These inductive equations do not necessarily hold when placement of database relations on resource units overlap. In this section, we address this problem by showing that symmetry in data placement can reduce the number of dimensions significantly. The complexity of multi-dimensional dynamic programming increases significantly with the increase in the number of dimensions, hence the motivation for reducing the number of dimensions.

**Definition 9:** The placement of a database relation $R$ is the set of resource units used to store the relation and is denoted by $\text{placement}(R)$. $\square$

If the placement of relations overlap, then the basic problem is to estimate the largest parallel time of a set of $n$ scan operations denoted by $\text{RPT}$ (Rooted Parallel Time), where $n$ is no more than the number of relations in the query.

It is clear that if there are $n$ relations, then $\text{RPT}$ is a monotonic function of the access times for each of the relations, irrespective of the overlaps between the relations. Hence, we can model the largest rooted time as an $n$-dimensional vector.

Note that the inductive definitions of metrics $int_A$, $A$ and $c$ remain unaffected by overlapping placement of relations. However, in general, metrics $r$ and $M$ require an $n$-dimensional estimation for $\text{RPT}$ and another dimension for estimating parallel floating time, i.e., an $n + 1$ dimensional estimation. Thus, equations (6) and (7) imply that optimizing for $\text{EPT}$ requires $2n + 3$ dimensions and optimizing for $\text{CPT}$ requires $n + 3$ dimensions. Below,
we show how symmetry in data placement can significantly reduce the number of dimensions. We present the tree placement strategy next.

**Definition 10:** The placement of a set of relations is a tree placement scheme if the following condition is satisfied: for every pair of relations, either their placements do not overlap or the placement of one subsumes the placement of the other. □

The tree placement scheme is named after the fact that the resulting placement can be represented in the form of a tree. The placement tree is a weighted tree (i.e., nodes have weights) and is defined as follows. Let $R_1, R_2, \ldots, R_n$, be the set of relations.

1. The vertices of the placement tree are $\text{placement}(R_i)$. The weight of a node is equal to the number of relations with an identical placement.

2. There is an edge from $\text{placement}(R_i)$ to $\text{placement}(R_j)$ if (1) $\text{placement}(R_j)$ is a strict subset of $\text{placement}(R_i)$ and (2) there does not exist $R_k$ such that $\text{placement}(R_k)$ is a strict subset of $\text{placement}(R_i)$ and is a strict superset of $\text{placement}(R_j)$.

In the tree placement scheme, every node has a unique parent and hence the graph resulting from the above construction is a tree (or a forest). The maximum overlap occurs only at the leaf nodes. The number of leaf nodes in a tree on $n$ nodes is bounded by $n - 1$ and in a forest is bounded by $n$. This justifies the “symmetry” of the tree placement scheme. An example of a tree placement scheme is presented in Figure 9. The following examples
Figure 10: An example of a tree placement policy

illustrate how \( RPT \) may be estimated with a few dimensions given a tree placement of relations and an acceptable factor of tolerance \( \epsilon \).

Algorithm to find the minimum number of dimensions required to estimate \( RPT \), given a tolerance margin of \( \epsilon \) is presented in [18].

**Example 3:** Consider TREE1 shown in Figure 10. It represents a particular placement of 10 relations, 5 of which have an identical placement, say \( R_1 \) through \( R_5 \) and the remainder 5 relations, i.e., \( R_6 \) through \( R_{10} \) have pairwise disjoint placement and are contained within the placement of \( R_1 \).

The formula for the largest rooted parallel time is:

\[
RPT(T) = \max \left( \begin{array}{c}
a_1 + a_2 + a_3 + a_4 + a_5 + a_6 \\
a_1 + a_2 + a_3 + a_4 + a_5 + a_7 \\
a_1 + a_2 + a_3 + a_4 + a_5 + a_8 \\
a_1 + a_2 + a_3 + a_4 + a_5 + a_9 \\
a_1 + a_2 + a_3 + a_4 + a_5 + a_{10} \\
\end{array} \right).
\]

\( RPT \) is a monotonic function of each of the rows in the above matrix and hence can be estimated using 5 dimensions (equal to the number of leaves in the tree). The above formula can be rewritten as

\[
RPT(T) = a_1 + a_2 + a_3 + a_4 + a_5 + \max(a_6, a_7, a_8, a_9, a_{10})
\]

This shows us that \( RPT \) can be estimated using only two dimensions, viz., The dimensions are:

1. \( a_1 + a_2 + a_3 + a_4 \)
2. \( \max(a_6, a_7, a_8, a_9, a_{10}) \)
This implies that $EPT$ can be estimated using $2 \times 2 + 3 = 7$ dimensions (instead of $2 \times 10 + 3 = 23$ dimensions) and $CPT$ may be estimated using $2 + 3 = 5$ dimensions (instead of $10 + 3 = 13$ dimensions). \hfill \Box

The following example shows that, given an $\epsilon$, which represents the tolerance in error, the number of dimensions for estimating $EPT$ and $CPT$ is much less than $2n + 3$ and $n + 3$ respectively.

**Example 4:** Consider TREE2 in Figure 10. Let $a_1 \ldots a_4$ be the time of access of the relations in root, $a_5$ and $a_6$ be the time of access of the relations in the second level and $a_7$ through $a_{14}$ be the time of access of the leaf relations. There are 8 leaves and a straightforward implementation would require 8 dimensions. The following formula may be used to predict the largest rooted time.

$$RPT(T) = a_1 + \ldots + a_4 + \max(\max_{i=7}^{10}(a_5 + a_i), \max_{i=11}^{14}(a_6 + a_i))$$

$$\approx RPT(T) = (a_1 + \ldots + a_4) + \max_{i=6}^{14}(a_i)$$

where the approximation is within a factor of 2, i.e. $1 \leq \frac{RPT(T)}{\tilde{RPT}(T)} \leq 2$.

Hence, if an error of a factor of 2 is tolerable, then there is a 2-dimensional prediction of the largest rooted time. The dimensions are:

1. $a_1 + \ldots + a_4$
2. $\max(a_5, a_6, \ldots, a_{14})$.

This implies that $EPT$ can be estimated using $2 \times 2 + 3 = 7$ dimensions (instead of $2 \times 14 + 3 = 31$) and $CPT$ may be estimated using $2 + 3 = 5$ dimensions (instead of $14 + 3 = 17$) to within a factor of 2. \hfill \Box

Note that if the placement of all relations have a non-empty overlap, then $EPT$ can be estimated using 5 dimensions and $CPT$ can be estimated using 4 dimensions without incurring any error.

## 8 Conclusions and Future Work

In this paper, we have designed an an optimizer that minimizes the parallel execution time for relational database queries. This optimizer is designed using the following methodology.

1. We first quantify the notion of coarse grain parallel execution and the degree of parallelism. We present local conditions for checking when an execution is coarse grain and estimate the degree of parallelism for each operation in an execution tree.
2. We design a scheduler with a near-optimal performance for parallelizing a coarse grain tree. This is based on the scheduling strategies for rectangle packing presented in [1].

3. We analyze the scheduling algorithm to obtain cost estimates (for example, one estimate is the computing area/number of resource units + length of the critical path of execution) for the parallel time of an execution tree. We use these estimates to design a three dimensional dynamic programming search algorithm for the case when relations do not overlap.

4. Overlap between the "home sites" of database relations increases the number of dimensions. We observe that symmetry in the overlap patterns can be exploited to keep the search algorithm efficient. We propose a novel placement strategy, called the tree placement strategy, and demonstrate, by means of examples, how the number of dimensions in the search algorithm can be significantly reduced, thus increasing its efficiency.

Future work involves evaluating the performance of the scheduler and the optimizer (and its variants) [17]. Better schedules may be obtained by clustering neighboring operations in an operator tree and is being studied in [17]. A complete algorithm for minimizing the dimensions for a tree placement scheme and other symmetric placement schemes may be found in [18].

Acknowledgements

We thank Don Smith for helping us understand the communication model for parallel machines. We thank Tao Yang for useful comments on the section on coarse granularity and the model of communication.

References


28


