Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism

Waqar Hasan
Stanford University
and
Hewlett-Packard Laboratories
hasan@cs.stanford.edu

Rajeev Motwani*
Department of Computer Science
Stanford University
Stanford, CA 94305
rajeev@cs.stanford.edu

Abstract

We address the problem of finding parallel plans for SQL queries using the two-phase approach of join ordering followed by parallelization. We focus on the parallelization phase and develop algorithms for exploiting pipelined parallelism. We formulate parallelization as scheduling a weighted operator tree to minimize response time. Our model of response time captures the fundamental tradeoff between parallel execution and its communication overhead. We assess the quality of an optimization algorithm by its *performance ratio* which is the ratio of the response time of the generated schedule to that of the optimal. We develop fast algorithms that produce near-optimal schedules – the performance ratio is extremely close to 1 on the average and has a worst case bound of about 2 for many cases.

1 Introduction

We address the problem of *parallel query optimization*, which is to find optimal parallel plans for executing SQL queries. Following Hong and Stonebraker [HS91], we break the optimization problem into two phases: join ordering followed by parallelization. We focus on the parallelization phase and develop optimization algorithms for exploiting pipelined parallelism.

Our model of parallel execution captures a fundamental tradeoff – *parallelism has a price* [Gra88, PMC+90]. Two communicating operators may either incur the communication overhead and run on distinct processors, or share a processor and save the communication overhead.

We model parallelization as scheduling an operator tree [GHK92, Hon92b, Sch90] on a parallel machine. We represent the resource requirements of operators and communication as node and edge weights respectively. Our optimization objective is to find a schedule (i.e. a parallel plan) with minimal response time.

This paper concentrates on algorithms for exploiting pipelined parallelism. In general, pipelining is a useful supplement to partitioned parallelism [DG92] but is sometimes the *only* way of speeding up a query. Consider decision support queries that join a large number (say 10) of relations and apply external functions, grouping and aggregation. Selection predicates may localize access to single partitions of each relation. If each reduced relation is small, partitioned parallelism ceases to be a viable option and pipelined parallelism is the only source of speedup. Algorithms for managing pipelined parallelism are thus an *essential* component of an optimizer.

To the best of our knowledge, scheduling theory does not provide algorithms that handle communication costs. Scheduling pipelines constitutes a nontrivial generalization of the classical problem of multiprocessor scheduling [GLLK79]. Prior work in parallel query optimization [Hon92a, SE93, LCRY93, LST91, SYT93, SYG92, TWPY92, WFA92] ignored the communication overhead of exploiting parallelism. This was sometimes justified by restricting to situations where communication overhead is low: shared-memory architectures with omission of pipelined parallelism.

Brute force algorithms are impractical for scheduling pipelines due to the extremely large search space. A query that joins 10 relations leads to an operator tree with about 20 nodes. The number of ways of scheduling 20 operators on 20 processors exceeds 5×10^{13}.

Algorithms that simply ignore communication over-

*Supported by NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation and Xerox Corporation.

**Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994**

head are unlikely to yield good results. We show that one such naive algorithm produces plans with twice the optimal response time on average, and is arbitrarily far from optimal in the worst case.

We will measure the quality of optimization algorithms by their *performance ratio* [GJ79] which is the ratio of the response time of the generated schedule to that of the optimal. Our goal is to develop algorithms that are near-optimal in the sense that the average performance ratio should be close to 1 and the worst performance ratio should be a small constant.

We develop two algorithms called *Hybrid* and *GreedyPairing*. We experimentally show that both algorithms, on the average, find near-optimal plans for small operator trees. Experiments with larger operator trees proved impossible for the very reason that it required the practically infeasible task of computing the optimal schedule.

Thus, one motivation for our worst-case analytical results was the need to provide performance guarantees independent of the size of the operator tree or the number of processors. Another motivation was to provide guarantees that do not depend on the choice of the experimental benchmark and are valid across all database applications. Finally, worst-case bounds on the performance ratio apply to each schedule and thus guarantee that the scheduling algorithm will *never* produce a bad plan.

We show, for p processors, the performance ratio of *Hybrid* is no worse than $2-1/p$ for operator trees which are paths and no worse than $2+1/p$ for stars. We also show the performance ratio of *GreedyPairing* to be no worse than $2-\frac{2}{p+1}$ when communication costs are zero.

Hybrid outperforms *GreedyPairing* almost uniformly but the difference is significant only when operator trees are large and communication costs are low. On the other hand, *GreedyPairing* is a simple algorithm which can be extended naturally to take data-placement constraints into account.

Section 2 develops a cost model for response time and provides a formal statement of the optimization problem. Section 3 summarizes our approach and discusses why a natural algorithm called *Naive LPT* does not perform well.

Section 4 develops the *GreedyChase* algorithm to identify parallelism that is simply not worth exploiting irrespective of the number of processors. Use of *GreedyChase* as a *pre-processing* step leads to improved schedules. The *Modified LPT* algorithm is developed and shown to be near-optimal for star-shaped operator trees.

Section 5 focuses on the restricted class of *connected* schedules. We show the optimal connected schedule to be a near-optimal schedule for path-shaped operator trees. We develop a fast polynomial algorithm called *BalancedCuts* to find the *optimal* connected schedule.

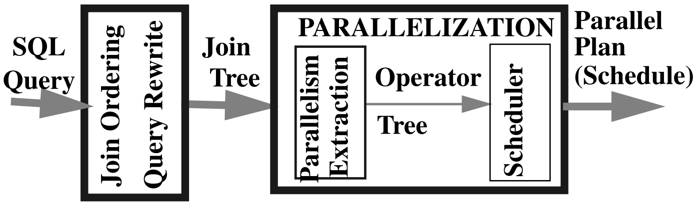

Figure 1: Two-phase Parallel Query Optimization

Section 6 develops algorithms for the general problem. The *Hybrid* algorithm is devised by combining the best features of connected schedules and the *modified LPT* algorithm. Finally, the *GreedyPairing* algorithm is developed and experimentally compared with *Hybrid*. Section 7 summarizes our contributions and provides directions for future work.

2 A Model for the Problem

Figure 1 shows a two-phase approach for compiling an SQL query into a parallel plan. The first phase is similar to conventional query optimization, and produces an annotated join tree that fixes aspects such as the order of the joins, join methods and access methods. The second phase is a parallelization phase that converts a join tree into a parallel plan.

We define a parallel plan to be a schedule consisting of two components: (1) an *operator tree* that identifies the atomic units of execution (operators) and the timing constraints between them; and, (2) an allocation of machine resources to operators.

We consider parallelization itself to consist of two steps. The first step translates an annotated join tree to an operator tree. The second step is a scheduling step that allocates resources to operators.

Section 2.1 refines prior notions of operator trees [GHK92, Hon92b, Sch90] by reducing timing constraints between operators to parallel and precedence constraints. Section 2.2 shows how resource requirements of nodes and edges may be derived from conventional cost models. Section 2.3 describes a cost model for response time. Finally, Section 2.4 provides a formal statement of the optimization problem addressed in this paper. A full description of the cost model and its justification can be found in [Has94a, Has94b].

2.1 Operator Trees

An operator tree is created as a "macro-expansion" of an annotated join tree (Figure 2). Nodes of an operator tree represent operators. Edges represent the flow of data as well as timing constraints between operators.

An operator is an atomic piece of code that takes zero or more input sets and produces a single output set. Operators are formed by appropriate factoring of the code that implements the relational operations specified in an annotated join tree. A criteria in

Figure 2: Macro-expansion of a Join Tree to a Weighted Operator Tree

designing operators is to reduce inter-operator timing constraints to simple forms, i.e. parallel and precedence constraints.

It is often possible to run a consumer operator in parallel with an operator that produces its input. This is termed *pipelined parallelism*. In order to identify such opportunities for speedup, we classify the inputs/output of operators into two idealized categories:

• *blocking*: the set of tuples is produced/consumed as a *whole set*. A blocking output produces the entire output set at the instant the operator terminates. An operator with a blocking input requires the entire input set before it can start.

• *pipelining*: the set is produced/consumed "tuple at a time" and the tuples are uniformly spread over over the *entire* execution time of the operator.

Opportunities for pipelined parallelism exist only along edges that connect a pipelining output to a pipelining input.

Definition 2.1 If an edge connects a pipelining output to a pipelining input, it is a *pipelining edge*; otherwise it is a *blocking edge*. □

In Figure 2 blocking edges are shown as thick edges.

Pipelined execution is typically implemented using a *flow control* mechanism (such as a table queue [PMC+90]) to ensure that a fixed amount of memory suffices for the pipeline. This constrains all operators in a pipeline to run concurrently - the pipeline executes at the pace of the slowest operator. Thus, pipelining edges represent parallel constraints, and blocking edges represent precedence constraints.

Definition 2.2 Given an edge from operator i to j, a *parallel constraint* requires i and j to start at the same time and terminate at the same time. A *precedence constraint* requires j to start after i terminates. □

Note that a pipelining constraint is symmetric in i and j. The direction of the edge indicates the direction in which tuples flow but is immaterial for timing constraints.

The code for join and access methods is expected to be broken down into operators such that inputs/outputs are easily classifiable as blocking or pipelinable. For example a simple hash join may be broken into `build` and `probe` operators. The `build` operator produces a "whole set," i.e., the hash table while `probe` produces output "tuple at a time." If the inputs/output are not easily classifiable, the operator should be further broken down. As an example consider the merge-sort algorithm that consists of repeatedly forming larger runs followed by a merge of the final set of runs. The output is neither produced at the instant the operator terminates nor is it spread out over the entire execution time of the operator. This situation is handled by breaking the algorithm into two operators: `form-runs` that produces a blocking output, i.e. the final runs, and `merge-runs` that takes a blocking input and produces a pipelining output.

2.2 Resource Requirements as Weights

The weight t_i of node i is the time to run the operator in isolation assuming all communication to be local. The dominant overhead of using distinct processors for two communicating operators is the additional instructions that must be executed for communicating data [Gra88, PMC+90]. The weight c_{ij} of an edge between nodes i and j is the *additional* execution time incurred by both operators if they are assigned distinct processors.

Conventional models (such as System R [SAC+79]) use statistical models to estimate the sizes of the intermediate results. These sizes are then used to determine the CPU, IO and communication requirements of relational operations such as joins. Since relational operations are broken down into operators, the same formulas are easily extended for operators.

Given the CPU, IO and communication requirements of an operator, its execution time may be estimated by a formula that combines the individual resource requirements using functions such as *weighted sum* and *max*. The exact formula depends on the hardware and software architecture of the system in consideration. The nature of the formula is not important in solving the parallelization problem, it suffices that such formulas exist.

2.3 Response Time of a Schedule

In the rest of this paper, we shall restrict ourselves to operator trees with only pipelining edges and these edges will be considered undirected since the direction is immaterial for the timing constraints. For example, if the indexes (hash tables) pre-exist, the operator tree of Figure 2 would have no blocking edges and would reduce to the one shown in Figure 3(a).

A schedule (i.e. parallel plan) allocates operators to processors. Since we assume processors to be identical, a schedule may be regarded as a partition of the set of operators.

Definition 2.3 Given p processors and an operator tree $T = (V, E)$, a *schedule* is a partition of V, the set of nodes, into p sets $F_1, \ldots, F_p$. □

Suppose F is the set of operators allocated to some processor. The cost of executing F is the cost of executing all nodes in F plus the overhead for communicating with nodes on other processors. It is thus the sum of the weights of all nodes in F and the weights of all edges that connect a node within F to a node outside. For convenience, we define $c_{ij} = 0$ if there is no edge from i to j.

Definition 2.4 If F is a set of operators, $cost(F) = \sum_{i \in F}[t_i + \sum_{j \notin F} c_{ij}]$. □

Definition 2.5 If F is the set of operators allocated to processor p, $load(p) = cost(F)$. □

Since our goal is to minimize the response time R of a parallel plan, we now derive a formula for R given an operator tree $T = (V, E)$ and a schedule $S = F_1, \ldots, F_p$.

The pipelining constraint forces all operators in a pipeline to start simultaneously (time 0) and terminate simultaneously at time R. Fast operators are forced to "stretch" over a longer time period by the slow operators. Suppose operator i is allocated to processor p_i and uses fraction f_i of the processor. The pipelining constraint is then:

$$f_i = \frac{1}{R}[t_i + \sum_{j \notin F_{p_i}} c_{ij}] \quad \text{for all operators } i \in V \quad (1)$$

The utilization of a processor is simply the sum of utilizations of the operators executing on it. Since at least one processor must be saturated (otherwise the pipeline would speed up):

$$\max_{1 \le l \le p} [\sum_{i \in F_l} f_i] = 1$$

$$\Rightarrow \quad R = \max_{1 \le l \le p} cost(F_l) \qquad \text{using equation (1)}$$

Definition 2.6 The response time $rt(S)$ of schedule $S = F_1, \ldots, F_p$ is $\max_{1 \le l \le p} cost(F_l)$. □

Example 2.1 Figure 3(a) shows a schedule for a pipelined operator tree. Notice that the join tree of Figure 2 would expand to exactly this tree if indexes pre-exist. Edges are shown as undirected since the direction is immaterial for the purposes of scheduling. The schedule is shown by encircling the set of operators assigned to the same processor. The cost of each set is underlined. For example $\{probe\}$ costs 8 by adding up its node weight (7) and the weight of the edge (1) connecting it to its child. Figure 3(b) shows a Gantt chart of the execution specified by the schedule. The fraction of the processor used by each operator in shown in parenthesis. ■

2.4 Formal Problem Statement

The pipelined operator tree scheduling problem may now be stated as follows.

Input: Operator Tree $T = (V, E)$ with positive real weights t_i for each node $i \in V$ and c_{ij} for each edge $(i, j) \in E$; number of processors p

Output: A schedule S with minimal response time i.e., a partition of V into $F_1, \ldots, F_p$ that minimizes $\max_{1 \le l \le p} \sum_{i \in F_l}[t_i + \sum_{j \notin F_l} c_{ij}]$.

This problem is intractable since the special case in which all edge weights are zero is the NP-complete problem of *multiprocessor scheduling* [GJ79, GLLK79].

Since the number of ways of partitioning n elements into k disjoint non-empty sets is given by $\left\{ {n \atop k} \right\}$, which denotes Stirling numbers of the second kind [Knu73], we have

Lemma 2.1 *The number of distinct schedules for a tree with n nodes on p processors is* $\sum_{1 \le k \le p} \left\{ {n \atop k} \right\}$

This number of schedules is about 1.2×10^5 for $n = p = 10$, 1.4×10^9 for $n = p = 15$, and 5.0×10^{13} for $n = p = 20$.

3 Overview of Our Approach

Scheduling pipelined operator trees is an intractable problem and the space of schedules is super-exponentially large. Thus any algorithm that finds the *optimal* is likely to be too expensive to be usable. Our approach is to develop *approximation algorithms* [GJ79, Mot92], i.e., *fast* heuristics that produce *near-optimal* schedules.

We first discuss our methodology for evaluating algorithms. We then show why a naive algorithm does not perform well. Finally, we provide a road map to algorithms in the rest of the paper.

The proofs of lemmas and theorems are omitted due to space constraints. The interested reader is referred to the full version of this paper [HM94].

3.1 Methodology

We will evaluate algorithms based on their *performance ratio* which is defined as the the ratio of the response time of the generated schedule to that of the optimal.

Our goal is to devise algorithms that satisfy two criteria. Firstly, the *average* performance ratio should be close to 1. Secondly, the *worst* possible performance ratio should be *bounded*. In other words, the performance

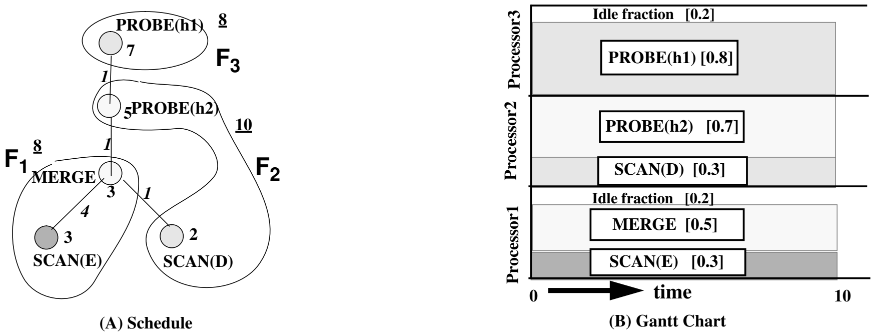

Figure 3: A parallel plan and its execution

ratio should not keep increasing with the problem size (number of processors or size of the operator tree).

We provide analytical proofs for worst-cases of performance ratios. Results on the average-case of performance ratios are based on experiments.

All experiments were done by random sampling from spaces specified by four parameters: *shape*, *size*, *edgeRange* and *nodeRange*. Three kinds of shapes could be specified: trees, paths and stars. The size specified the number of nodes in the tree to be generated. The edgeRange and nodeRange specified the range of integers from which node and edge weights could be drawn.

Definition 3.1 A star is a tree in which one node, called the center, is directly connected to all other nodes called leaves. □

3.2 The Naive LPT Algorithm

The *LPT* algorithm is one of the best algorithms for scheduling independent jobs with a known running time on a multiprocessor. It assigns the job with the largest running time to the least loaded processor, repeating this step until all jobs are assigned. For p processors, *LPT* has a worst case performance ratio of $\frac{4}{3} - \frac{1}{3p}$, i.e., an *LPT* schedule differs from the optimal by 33% in the *worst* case [Gra69].

LPT is a natural candidate since our problem reduces to multiprocessor scheduling when all communication costs are zero. One way of applying *LPT* to our problem is to simply use the *cost* of each node (i.e. the node weight plus weights of all incident edges) as its running time.

This naive algorithm performs poorly since it is unaware of the tradeoff between parallelism and communication cost. Consider two operators each of weight t connected by an edge of weight c. To obtain a schedule for 2 processors, *Naive LPT* will consider the cost of each operator to be $t + c$ and place them on separate processors resulting in a schedule with a response time of $t + c$. *LPT* can never come up with the schedule that saves communication cost by placing both operators on a single processor, thus achieving a response time of $2t$. Since cheap operators and expensive communication can make the ratio $\frac{t+c}{2t}$ arbitrarily large:

Lemma 3.1 *The worst case performance ratio of Naive LPT is unbounded.*

The average case performance of *Naive LPT* is also poor. On the average, generated schedules had twice the optimal response time (see Figure 6 and experiments described in Section 6.3.1).

3.3 A Road Map to Subsequent Algorithms

Section 4 develops the *GreedyChase* algorithm to identify parallelism that is simply not worth exploiting. This leads to the *modified LPT* algorithm which consists of running *GreedyChase* followed by *LPT*. We show that this algorithm produces near-optimal schedules for star-shaped operator trees but not for paths.

Section 5 develops a fast algorithm called *BalancedCuts* to find the optimal "connected" schedule. Further, we show that the optimal connected schedule is a near-optimal schedule for path-shaped operator trees.

Section 6 develops the *Hybrid* algorithm by combining *modified LPT* and *BalancedCuts*. Another algorithm called *GreedyPairing* is separately developed. Both algorithms use *GreedyChase* as a pre-processing step.

4 Identifying Worthless Parallelism

In this section we develop an understanding of the tradeoff between parallelism and communication cost. We develop the *GreedyChase* algorithm that "chases down" and removes parallelism that is "worthless" irrespective of the number of processors.

The reason that maximal use of parallelism does not necessarily yield minimal response time is that adding more operators to a processor can in fact *reduce* its load. This non-monotonicity arises when savings in communication cost offset the additional cost of the added operators. In Figure 3, the set {MERGE} has a cost of 9 seconds. When SCAN(E) is added to the set, the cost reduces to 8 since 4 seconds of communication are saved while only 3 seconds of computation are added.

In Section 4.1, we identify a class of trees that we call *monotone*. Such trees have no worthless parallelism in the sense that maximal use of parallelism is in fact optimal. In Section 4.2, we characterize *worthless edges* whose communication overhead is relatively high enough to exceed any benefits from parallelism.

In Section 4.3, we show that repeatedly "collapsing" worthless edges results in a monotone tree. This provides us with fast and simple *GreedyChase* algorithm that removes *all* and *only* worthless parallelism. Finally, we use *GreedyChase* as a pre-processing step to design the *modified LPT* algorithm which is shown to do well on star shaped operator trees.

4.1 Monotone Trees

Definition 4.1 An operator tree is *monotone* iff any connected set of nodes, X, has a lower cost than any connected superset, Y, i.e., if $X \subset Y$ then $cost(X) < cost(Y)$. □

Monotonicity guarantees that adding more operators to a processor will always increase processor load. Thus maximal use of parallelism is indeed optimal. The following lemma shows that a monotone tree does not have any parallelism which if exploited would result in *increasing* response time.

Lemma 4.1 *If there are at least as many processors as the number of operators then a schedule that allocates each operator to a distinct processor is an optimal schedule for a monotone operator tree.*

Thus monotone trees only have beneficial parallelism and the challenge (dealt with in subsequent sections) is to pick what parallelism to exploit when the number of processors is not sufficient to exploit all parallelism.

We now define the operation of *collapsing* two nodes. It is simply a way of constraining two nodes to be assigned to the same processor. We shall use collapsing as a basic operation in converting an arbitrary tree into a monotone tree.

Definition 4.2 $Collapse(i_1, i_2)$ collapses nodes i_1 and i_2 in tree T. T is modified by replacing i_1 and i_2 by a single new node i. The weight of the new node is the sum of the weights of the two deleted nodes, i.e. $t_i = t_{i_1} + t_{i_2}$. If there is an edge between the nodes, it is deleted. All other edges that were connected to either i_1 or i_2 are instead connected to i. □

Lemma 4.2 *If a schedule places both of nodes i_1 and i_2 on processor k, the load on all processors is invariant when i_1 and i_2 are collapsed, and the new node is placed on processor k.*

Note that collapsing two nodes in a tree could create cycles in the resulting graph, and therefore the new graph need not be a tree. Moreover, the two nodes may have edges to the same vertex, leading to the creation of multiple edges between a pair of nodes. The multiple edges can be removed by replacing them with a single edge whose weight is the sum of their weights.

In this section, we will only need to collapse adjacent pairs of nodes (which amounts to collapsing edges) and therefore the result is always a tree.

Lemma 4.3 *Given an operator tree T, and adjacent nodes i_1, $i_2 \in V$, the output of $Collapse(i_1, i_2)$ will also be an operator tree T' but with one fewer node.*

Definition 4.3 Collapsing an edge (i_1, i_2) is defined as collapsing its end-points i_1 and i_2. □

4.2 Worthless Edges

High edge costs can offset the advantage of parallel execution. In Figure 3, the cost incurred by MERGE in communicating with a remotely running SCAN(E) exceeds the cost of SCAN(E) itself. It is thus always better for the processor executing MERGE to simply execute SCAN(E) locally rather than communicate with it. We now generalize this observation.

Definition 4.4 The (ordered) pair of nodes $\ll i, j \gg$ is said to be a *worthless pair* iff the cost c_{ij} of the edge connecting them is no smaller than the sum of t_j and the costs of the remaining edges incident with node j, i.e. $c_{ij} \geq t_j + \sum_{k \in (V - \{i\})} c_{jk}$ or equivalently $cost(\{i\}) \geq cost(\{i, j\})$. □

Definition 4.5 An edge (i, j) is said to be worthless iff either $\ll i, j \gg$ or $\ll j, i \gg$ is a worthless pair. □

The following theorem shows that our definition of worthless indeed captures edges whose high communications costs offset the advantages of parallel execution.

Theorem 4.1 *Given p processors and an operator tree T with worthless edge (i, j), there exists an optimal schedule of T for p processors in which nodes i and j are assigned to the same processor.*

4.3 The Greedy Chase Algorithm

We now establish an important connection between worthless edges and monotone trees. The following theorem allows us to transform *any* tree into a monotone tree simply by collapsing all worthless edges.

Theorem 4.2 *A tree is monotone iff it has no worthless edges.*

More importantly, we can schedule the monotone tree rather than the original tree. This follows since collapsing worthless edges does not sacrifice optimality (Theorem 4.1) and the schedule for the original tree can be recovered from the schedule for the transformed tree (Lemma 4.2).

Algorithm 4.1 The *GreedyChase* Algorithm

Input: An operator tree
Output: A monotone operator tree
1. **while** there exists some worthless edge (i, j)
2. Collapse(i,j)
3. **end while**

Since each collapse reduces the number of nodes, *GreedyChase* must terminate. The check for the existence of a worthless edge is the crucial determinant of the running time. When a worthless edge is collapsed, adjacent edges may turn worthless and thus need to be rechecked. The algorithm may therefore be implemented to run in time $O(nd)$, where n is the number of nodes and d is the maximum degree of any node. Experimentally, the running time of our implementation of *GreedyChase* was virtually linear in n.

4.4 The Modified LPT Algorithm

The *modified LPT* algorithm simply pre-processes away worthless parallelism by running *GreedyChase* before running *LPT*. As shown by the following example this results in a significant improvement.

Example 4.1 Figure 4(A) shows how *GreedyChase* collapses worthless edges (worthless edges are hatched). Note that edges that are not worthless may turn worthless as a result of other collapses.

If we have two processors, *modified LPT* will produce the schedule (B) with a response time of 11. *Naive LPT* on the other hand could produce the schedule (C) which has a response time of 25. ■

Modified LPT performs well on operator trees that are star-shaped. All edges in such a tree have low communication costs since the weight of an edge cannot exceed the weight of the leaf without making the edge worthless. In fact, we can show the following theorem.

Theorem 4.3 *The worst-case performance ratio of the* modified LPT *algorithm is less than* $2+1/p$ *for stars.*

The algorithm is still oblivious to the tradeoff between parallelism and communication. Edges in a monotone path can have high weights and the algorithm is unaware of the savings that can accrue when two nodes connected by an edge with a large weight are assigned the same processor. In fact, we can show:

Lemma 4.4 *The worst-case performance ratio of the* modified LPT *algorithm is unbounded for paths.*

5 Connected Schedules

In this section, we develop the *BalancedCuts* algorithm for finding the optimal *connected* schedule. A connected schedule requires the nodes assigned to any processor to be a connected set. This restriction is equivalent to only considering schedules that incur communication cost on $p-1$ edges (the minimal possible number) when using p processors.

This problem is of interest since it offers a way of finding near-optimal general schedules. Section 5.4 will show that the optimal connected schedule is a near-optimal general schedule when the operator tree is a path. We shall use this fact in the design of the *Hybrid* algorithm in Section 6.1.

In Section 5.1, we show that finding an optimal connected schedule can be reduced to deciding which edges of the tree should be "cut" and which should be collapsed. In Section 5.2, we develop the *BalancedCuts* algorithm for finding connected schedules when all communication costs are zero. In Section 5.3 we show how the *GreedyChase* algorithm helps in generalizing *BalancedCuts* to account for communication costs.

5.1 Cutting and Collapsing Edges

A connected schedule for p processors divides the operator tree into $k \leq p$ *fragments* (i.e. connected components). We define a notion of *cutting* edges under which a connected schedule with k fragments is obtained by cutting $k-1$ edges and collapsing the remaining edges.

Definition 5.1 $Cut(i,j)$ modifies a tree by deleting edge (i,j) and adding its weight to that of the nodes i and j, i.e. $t_i^{new} = t_i^{old} + c_{ij}$ and $t_j^{new} = t_j^{old} + c_{ij}$. □

The following lemma follows directly from this definition.

Lemma 5.1 *If a schedule places nodes i and j on distinct processors, the load on all processors is invariant when the edge* (i,j) *is cut.*

The above lemma along with Lemma 4.2 establishes that we can view a connected schedule as cutting inter-fragment edges and collapsing intra-fragment edges. Figure 5 illustrates the translation between a connected schedule consisting of 3 fragments and a graph with 3 nodes and no edges. This suggests that one way of finding a connected schedule is examine all $O(2^n)$ combinations of cutting/collapsing edges. The next section shows how we can do better.

5.2 Connected Schedules when Communication is Free

We now develop an algorithm for finding the optimal connected schedule for trees in which all edge weights are zero. The algorithm is generalized to handle edge weights in the next section.

We will develop the algorithm in two steps. First, given a bound B and number of processors p, we develop an efficient way of finding a connected schedule with a response time of at most B, if such a schedule exists. Second, we show that starting with B set to a lower bound on the response time, we can use a small number of upward revisions to get to the optimal connected schedule.

5.2.1 Bounded Connected Schedules

Definition 5.2 A schedule is (B,p)-*bounded* iff it is a connected schedule that uses at most p processors and has a response time of at most B. □

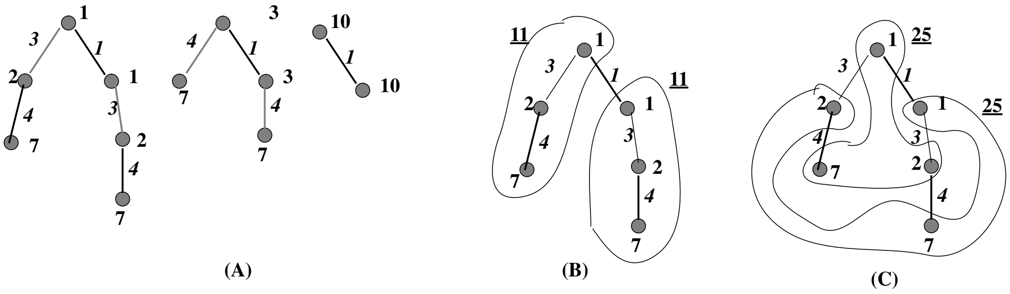

Figure 4: (A) Trace of *GreedyChase* (worthless edges hatched) (B) modified LPT schedule (C) naive LPT schedule

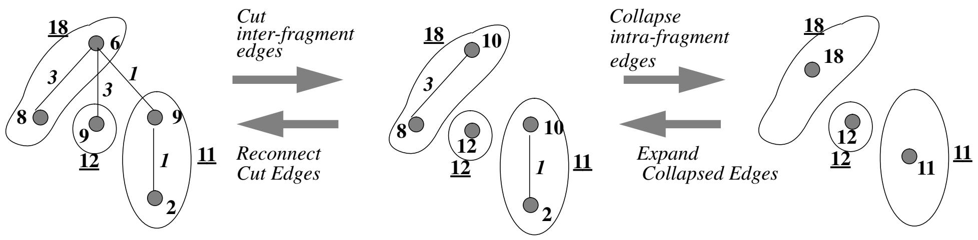

Figure 5: Connected Schedule as Cutting and Collapsing Edges

We first consider the simple case of a leaf node r with parent m to see how the decision to cut or collapse an edge can be made. Suppose $t_r + t_m > B$. Clearly, the edge (m, r) should be cut since otherwise we shall exceed the bound. Now suppose instead $t_r + t_m \leq B$. We claim that the edge (m, r) can be collapsed. Since r is connected only to m, if the connecting edge were cut, r would get a processor, say p_r, to itself. Putting m on p_r reduces the total work for other processors without causing the bound to be exceeded on p_r, and thus can never hurt. This basic idea will be generalized to derive an efficient algorithm. The algorithm bears a similarity to that of Hadlock [Had74] for a related but different problem.

Definition 5.3 A node is a *mother node* iff all adjacent nodes with at most one exception are leaves. The leaf nodes are termed the children of the mother node. □

The following lemmas narrow the set of schedules we need to examine. We assume m is a mother node with children $r_1, \ldots, r_d$ in the order of non-decreasing weight, i.e. $t_{r_1} \leq t_{r_2} \leq \ldots \leq t_{r_d}$.

Lemma 5.2 *If a (B,p)-bounded schedule places m and r_j in the same fragment and r_i in a different fragment where $i < j$ (i.e. $t_{r_i} \leq t_{r_j}$), then a schedule in which r_j and r_i exchange places is also (B,p)-bounded.*

Repeated application of Lemma 5.2 results in:

Lemma 5.3 *If there exists a (B,p)-bounded schedule, then there exists a (B,p)-bounded schedule such that (1) if (m, r_j) is collapsed then so is (m, r_{j-1}) (2) if (m, r_j) is cut then so is (m, r_{j+1})*

Let l be the largest number of children that can be collapsed with m without exceeding bound B, that is, the maximum l such that $t_m + \sum_{1 \leq i \leq l} t_{r_i} \leq B$

Theorem 5.1 *If there exists a (B,p)-bounded schedule, then there exists a (B,p)-bounded schedule such that (1) (m, r_j) is collapsed for $1 \leq j \leq l$ (2) (m, r_j) is cut for $l < j \leq d$*

Theorem 5.1 gives us a way of finding a (B,p)-bounded schedule or showing that no such schedule exists. We can simply pick a mother node and traverse the children in the order of non-increasing weights. We collapse children into the mother node as long the weight of the mother stays below B and then cut off the rest. We repeat the process until no more mother nodes are left. If the resulting number of fragments is no more than p, we have found a (B,p)-bounded schedule, otherwise no such schedule is possible.

Algorithm 5.1 The *BpSchedule* Algorithm

Input: Operator tree T with zero edge wts, bound B
Output: Partition of T into fragments $F_1, \ldots, F_k$
    s.t. $cost(F_i) \leq B$ for $i = 1, \ldots, k$

while there exists a mother node m
    Let m have children $r_1, \ldots, r_d$ s.t. $t_{r_1} \leq \ldots \leq t_{r_d}$
    Let $l \leq d$ be the max l s.t. $t_m + \sum_{1 \leq i \leq l} t_{r_i} \leq B$
    for $j = 1$ **to** l **do**
        collapse(m, r_j)
    for $j = l + 1$ **to** d **do**
        cut(m, r_j)
end while
return resulting fragments $F_1, \ldots, F_k$

5.2.2 The BalancedCuts Algorithm

We will find the optimal connected schedule by setting B to a lower bound on the response time and repeatedly revising B by as large an increment as possible while making sure that we do not overshoot the optimal value. For each such value of B we run *BpSchedule* and check whether the number of fragments is at most p.

A lower bound on response time can be derived by observing that some processor must execute at least the *average* weight and every node needs to be executed by some processor.

Lemma 5.4 *The maximum of* $\lceil \sum_{i \in V} t_i / p \rceil$ *and* $\max_{i \in V} t_i$ *is a lower bound on the response time of any schedule (not just connected schedules).*

We can use an unsuccessful run of *BpSchedule* to derive an improved lower bound. For each fragment F_i produced by *BpSchedule*, let B_i be the cost of the fragment plus the weight of the *next* node that was not included in the fragment (i.e. the value t_{l+1} when a cut is made in line 7 of *BpSchedule*). The following lemma is based on the intuition that at least one fragment needs to grow for the number of fragments to reduce.

Lemma 5.5 *Let* B^* *be the smallest of the* B_i*. Then,* B^* *is a lower bound on the optimal response time.*

Using the lower bound given by Lemma 5.4 and the revision procedure given by Lemma 5.5, we devise the algorithm shown below.

Algorithm 5.2 The *BalancedCuts* Algorithm

Input: Operator tree T with zero edge weights,
  number of processors p
Output: Optimal connected schedule
$B = \max\left(\lceil \sum_{i \in V} t_i / p \rceil, \max_{i \in V} t_i\right)$
repeat forever
  $F_1, \ldots, F_k$ = BpSchedule(T, B)
  if $k \leq p$ **return** $F_1, \ldots, F_k$
  Let $B_i = cost(F_i)$ + wt of next node not in F_i
  $B = \min_i B_i$
end repeat

BalancedCuts may be shown to terminate in at most $n-p+1$ iterations and have a running time of $O(n(n-p))$.

5.3 BalancedCuts with Communication Costs

Generalizing *BalancedCuts* to take care of communication requires two changes. Firstly, the input tree must be pre-processed by running *GreedyChase*. Secondly, *BpSchedule* must consider the children of a mother node in the order of non-decreasing $t_i - c_{im}$. Both changes are required to make *BpSchedule* work correctly.

BpSchedule assumes that adding more nodes to a fragment can only increase its cost. The monotone trees produced by *GreedyChase* guarantee exactly this property. Since the schedule for the original tree can be recovered from the schedule for the "pre-processed" tree (see Section 4.3), it suffices to schedule the monotone tree.

BpSchedule greedily "grows" fragments by collapsing children with their mother node as long as the fragment cost remains bounded. The children were ordered by non-decreasing weights, and the weight of each child was a measure of how much the weight of the fragment would increase by collapsing the child into the mother node. With non-zero edge weights, the mother node must pay the cost of communicating with the child when it is a different fragment. Thus collapsing the child i with the mother m increases the cost of the fragment by $t_i - c_{im}$. Simply ordering the children of the mother node in the order of non-decreasing $t_i - c_{im}$ suffices to generalize Lemmas 5.2 and 5.3 and Theorem 5.1.

5.4 Approximation using Connected Schedules

The optimal connected schedule is a good approximation for paths but not for stars.

Theorem 5.2 *The worst-case performance ratio in using the optimal connected schedule is at most* $2 - 1/p$ *for paths.*

Connected schedules are not a good approximation for stars since all fragments except the one containing the center are forced to consist of a single node.

Lemma 5.6 *The worst-case performance ratio in using the optimal connected schedule is unbounded for stars.*

6 Near-Optimal Scheduling

This section develops two near-optimal algorithms for scheduling pipelined operator trees and compares them experimentally.

We have the interesting situation in which the *modified LPT* algorithm works well for stars but not for paths, while connected schedules are a good approximation for paths but not for stars. This naturally motivates the combination of the two algorithms into a *Hybrid* algorithm (Section 6.1). In Section 6.2, we discuss the *GreedyPairing* algorithm which has the advantage of being extremely simple. Finally, we experimentally compare *GreedyPairing*, *Hybrid* and *Naive LPT*.

6.1 A Hybrid Algorithm

BalancedCuts performs poorly on stars since the constraint of connected schedules is at odds with load balancing. While the algorithm is cognizant of communication costs, it is poor at achieving balanced loads. On the other hand, *LPT* is very good at balancing loads but unaware of communication costs.

One way of combining the two algorithms is to use *BalancedCuts* to cut the tree into many fragments and then schedule the fragments using *LPT*. *LPT* can be expected to "cleanup" cases such as stars on which connected schedules are a bad approximation.

Algorithm 6.1 The *Hybrid* Algorithm

Input: Operator tree T, number of processors p
Output: A schedule
1. T' = GreedyChase(T)
2. **for** i = p **to** n **do**
3. $F_1, F_2, \ldots, F_i = BalancedCuts(T', i)$
4. schedule = $LPT(\{F_1, F_2, \ldots, F_i\}, p)$
5. **end for**
6. return best of schedules found in steps 2 to 5

Note that *Hybrid* has a performance ratio no worse than that obtained by using *BpSchedule* or by *modified LPT*. This is because the case $i = p$ will provide an optimal connected schedule, while the case $i = n$ will behave as the modified LPT algorithm. Thus the performance ratio is no worse than $2 - 1/p$ for paths and no worse than $2 + 1/p$ for stars.

6.2 The Greedy Pairing Algorithm

We now describe an algorithm which is based on greedily collapsing that pair of nodes which leads to the least increase in response time.

GreedyPairing starts by first pre-processing the operator tree into a monotone tree by running *GreedyChase*. Then it chooses the pair of nodes, i and j , such that $cost(\{i, j\})$ is the minimum possible and collapses them. Ties are broken by favoring the pair which offers the greatest reduction in communication. This process is continued until the number of nodes is reduced to p, and then each node is assigned a distinct processor. Note that collapsing two (non-adjacent) nodes in a tree will not necessarily maintain the property of being a tree.

We believe that this algorithm has a worst-case performance ratio close to 2. At this point, we can prove this result only in the case of zero edge weights.

Theorem 6.1 *The* GreedyPairing *algorithm has a worst-case performance ratio of* $.2 - 2/(p + 1)$ *when all edge weights are zero.*

6.3 Experimental Comparison

In Section 6.3.1 we compare the average performance ratios of *GreedyPairing*, *Hybrid* and *Naive LPT*. This experiment was practical only for small operator trees since it required computing the optimal schedule. In Section 6.3.2, we compare *GreedyPairing* and *Hybrid* with each other rather than with the optimal. This made it possible to experiment with large numbers of processors and large operator trees

6.3.1 Comparison with Optimal

We measured the performance ratios of *GreedyPairing*, *Hybrid* and *Naive LPT* for trees with up to 12 nodes on up to 12 processors. We ran 1000 trials for each combination of number of nodes and number of processors and measured the average and maximum performance ratios.

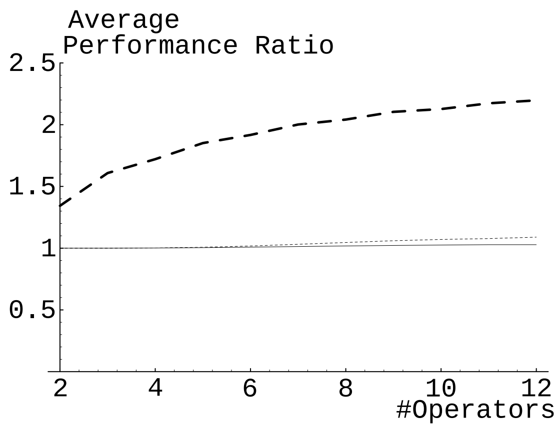

Figure 6: Average Performance Ratios of *Naive* (bold dashed), *GreedyPairing* (dashed) and *Hybrid* (solid)

Figure 7 shows the average performance ratios for two processors with *edgeRange* and *nodeRange* both set to $1 \ldots 10$. *Naive LPT* clearly performs poorly and we shall not discuss it further in the rest of this section. Both *GreedyPairing* and *Hybrid* have average performance ratios extremely close to 1. The averages increase (though gradually) with the size of the operator tree. *Hybrid* performs slightly better than *GreedyPairing*.

Varying the *shape*, *edgeRange* and *nodeRange* parameters had no significant effect. Larger numbers of processors resulted in *better* ratios for *GreedyPairing* and *Hybrid*.

Over *all* our experiments, the maximum performance ratio for *GreedyPairing* was observed to be 1.45 for a tree with 9 operators scheduled on 2 processors. For *Hybrid*, it was 1.36 for a tree with 9 operators scheduled on 3 processors.

6.3.2 Hybrid versus GreedyPairing

We directly compared *Hybrid* and *GreedyPairing* since computing the optimal schedule turned out to be practically impossible for large operator tree.

Letting g be the response time of the *GreedyPairing* schedule and h that of *Hybrid*, we measured $g/min(g, h)$ and $h/min(g, h)$. Each measurement was made over 1000 samples and had a relative error of less than 5% with a confidence of more than 95%.

We found the average value of $h/min(g, h)$ to be exactly 1 over a large range of experiments. Deviations were rare, and, if present, were very small (eg: 1.000021). In other words *Hybrid outperforms GreedyPairing almost uniformly.*

However, $g/min(g, h)$ was also found to be extremely small, except when edge weights were relatively small compared to node weights. Figure 7 plots g/h for 2,

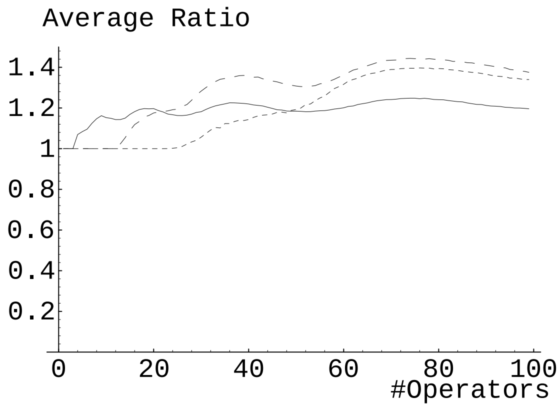

Figure 7: Average Ratio g/h of *GreedyPairing* to *Hybrid* for 2 processors (solid), 10 processors (dashed) and 20 processors (dotted)

10 and 20 processors when *edgeRange* was $1 \ldots 10$ and *nodeRange* was $11 \ldots 100$.

The explanation for behavior is that the problem essentially reduces to multiprocessor scheduling (i.e. zero edge weights) when edge weights are relatively small. *Hybrid* in that case reduces to the *LPT* algorithm with a worst case performance ratio of $\frac{4}{3} - \frac{1}{3p}$ or about 1.33 while *GreedyPairing* has a worst case performance ratio of $2 - \frac{2}{p+1}$ or about 2.

7 Conclusion

We have developed algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. Our algorithms are efficient and produce near-optimal schedules – the performance ratio is extremely close to 1 on the average and has a worst case bound of about 2 for many cases.

We focused on a subproblem of the parallelization problem that needs to be solved by a real optimizer. This subproblem is *exactly* the problem that needs to be solved for some queries and is thus of practical value. Our work also constitutes a non-trivial generalization of the classical problem of multi-processor scheduling [GLLK79].

Our model of parallelization as scheduling a weighted operator tree is general enough to be applicable to SQL queries that contain operations other than joins, such as aggregation or foreign functions. Further, our work is likely to be applicable to any hardware/software architecture where there is a tradeoff between parallelism and its communication overhead.

We showed that a naive use of the classical *LPT* algorithm results in an unbounded performance ratio. We developed a $O(nd)$ *GreedyChase* algorithm for preprocessing away worthless parallelism from an operator tree. This lead to the *Modified LPT* algorithm that runs *GreedyChase* followed by *LPT*. The performance ratio of *Modified LPT* was found to have a worst-case bound of $2 + 1/p$ for operator trees which are stars. However, the ratio was found to be unbounded for paths.

We then investigated the use of the optimal *connected* schedule as an approximation. We devised a $O(n(n-p))$ algorithm called *BalancedCuts* to find the optimal connected schedule. The performance ratio in using the optimal connected schedule was found to have a worst-case bound of $2 - 1/p$ for paths. The ratio was found to be unbounded for stars.

The *GreedyPairing* and *Hybrid* algorithms are both $O(n^3)$ algorithms for the general problem. Both have average performance ratios extremely close to 1. The performance ratio of *Hybrid* is no worse than $2 - 1/p$ for paths and no worse than $2 + 1/p$ for stars. The performance ratio of *GreedyPairing* is no worse than $2 - 2/(p+1)$ when communication costs are zero.

Hybrid was found to be almost uniformly superior to *GreedyPairing* but the differences were significant only for large operator trees with low communication costs. *GreedyPairing* offers the advantage of being a simple algorithm which can naturally be extended to take data-placement constraints into account.

The general problem of parallel query optimization remains open [DG92, Val93] and our work suggests several directions for further work.

In shared-nothing systems, the *leaf* operators of an operator tree are constrained to execute where the data is placed. While *GreedyPairing* can naturally take such constraints into account, we have not yet investigated its performance under this constraint.

Blocking edges in operator trees place precedence constraints between operators and partitioned parallelism allows several processors to compute a single operator. Prior work has addressed these problems only for situations where communications costs can be ignored.

Finally, it is challenging to devise richer models of the parallelization problem. We modeled the cost of an operator as an integer. Such costs are obtained by combining the time spent on different classes of resources using functions such as *weighted sum* and *max*. A more general model would represent cost as a multi-dimensional vector with a dimension per resource class. As a first step, Hong and Stonebraker [Hon92a] devised a scheduling algorithm for *two* operators (one IO-bound and one CPU-bound) with *no* timing constraints between them under the assumption that perfect speedup is obtainable by partitioned parallelism. Multi-dimensional scheduling algorithms that minimize response time for multiple classes of resources, each with an independent resource limit, are an open problem.

Acknowledgements: We thank Jim Gray for advice on modeling parallel query optimization and Jeff Ullman for many useful discussions on solving the problem. Thanks are also due to Surajit Chaudhuri, Umesh Dayal, Ashish Gupta, Stephanie Leichner, Marie-Anne Neimat, Donovan Schneider and Arun Swami for useful comments. The first author also thanks Hector Garcia-Molina, Ravi Krishnamurthy, Jeff Ullman and Gio Wiederhold for intellectual support.

References

[DG92] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, June 1992.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman and Company, 1979.

[GLLK79] R.L Graham, E.L. Lawler, J.K. Lenstra, and A.H.G Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[Gra69] R.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March 1969.

[Gra88] J. Gray. The Cost of Messages. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 1–7, Toronto, Ontario, Canada, August 1988.

[Had74] F.O. Hadlock. Minimum Spanning Forests of Bounded Trees. In *Proceedings of the 5th Southeastern Conference on Combinatorics, Graph Theory and Computing*, pages 449–460. Utilitas Mathematica Publishing, Winnipeg, 1974.

[Has94a] W. Hasan. A Model for Parallelization of SQL Queries, 1994. Submitted for publication.

[Has94b] W. Hasan. *Optimizing Response Time of Relational Queries by Exploiting Parallel Execution.* PhD thesis, Stanford University, 1994. In preparation.

[HM94] W. Hasan and R. Motwani. Optimization Algorithms for Pipelined Parallelism. Technical report, HP Laboratories, 1994. HPL-94-50.

[Hon92a] W. Hong. Exploiting Inter-Operation Parallelism in XPRS. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, June 1992.

[Hon92b] W. Hong. *Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays.* PhD thesis, University of California, Berkeley, August 1992.

[HS91] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December 1991.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol 1: Fundamental Algorithms.* Addison-Wesley, 2nd edition, 1973.

[LCRY93] M-L. Lo, M-S. Chen, C.V. Ravishankar, and P.S. Yu. On Optimal Processor Allocation to Support Pipelined Hash Joins. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 69–78, June 1993.

[LST91] H. Lu, M-C. Shan, and K-L. Tan. Optimization of Multi-Way Join Queries for Parallel Execution. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[Mot92] R. Motwani. *Lecture Notes on Approximation Algorithms (Volume I).* Computer Science Report STAN-CS-92-1435, June 1992.

[PMC+90] H. Pirahesh, C. Mohan, J. Cheung, T.S. Liu, and P. Selinger. Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. In *Second International Symposium on Databases in Parallel and Distributed Systems*, Dublin, Ireland, 1990.

[SAC+79] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1979.

[Sch90] D. A. Schneider. *Complex Query Processing in Multiprocessor Database Machines.* PhD thesis, University of Wisconsin—Madison, September 1990. Computer Sciences Technical Report 965.

[SE93] J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.

[SYG92] A. Swami, H.C. Young, and A. Gupta. Algorithms for Handling Skew in Parallel Task Scheduling. *Journal of Parallel and Distributed Computing*, 16:363–377, 1992.

[SYT93] E. J. Shekita, H.C. Young, and K-L Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, 1993.

[TWPY92] J. Turek, J.L. Wolf, K.R. Pattipati, and P.S. Yu. Scheduling Parallelizable Tasks: Putting it All on the Shelf. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, June 1992.

[Val93] P. Valduriez. Parallel Database Systems: Open Problems and New Issues. *Distributed and Parallel Databases: An International Journal*, 1(2):137–165, April 1993.

[WFA92] A.N. Wilschut, J. Flokstra, and P.M. Apers. Parallelism in a main-memory dbms: The performance of prisma/db. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada, August 1992.