Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources

Minos N. Garofalakis
University of Wisconsin–Madison
minos@cs.wisc.edu

Yannis E. Ioannidis*
University of Wisconsin–Madison
yannis@cs.wisc.edu

Abstract

Scheduling query execution plans is a particularly complex problem in hierarchical parallel systems, where each site consists of a collection of local time-shared (e.g., CPU(s) or disk(s)) and space-shared (e.g., memory) resources and communicates with remote sites by message-passing. We develop a general approach to the problem, capturing the full complexity of scheduling distributed multi-dimensional resource units for all kinds of parallelism within and across queries and operators. We present heuristic algorithms for various forms of the problem, some of which are provably near-optimal. Preliminary experimental results confirm the effectiveness of our approach.

1 Introduction

In the shared-nothing [7] and the more general hierarchical (or, hybrid) [2] multiprocessor architectures, each *site* consists of its own set of local resources and communicates with other sites only by message-passing. Despite the popularity of these architectures, the development of effective and efficient query processing and optimization techniques to exploit their full potential still remains an issue of concern [9, 25].

Prior work has already demonstrated the importance of resource scheduling during parallel query optimization. One of the main sources of complexity for the problem is the *multi-dimensionality* of the resource needs of database queries. That is, during their execution queries typically require multiple resources, such as memory buffers and CPU and disk bandwidth. This introduces a range of possibilities for effectively scheduling system resources among concurrent query operators, which can substantially increase the utilization of these resources and reduce the response time of the query. Moreover, system resources can be categorized into two radically different classes with respect to their mode of usage by query plan operators:

- Time-Shared (TS) (or, preemptable) resources (e.g., CPUs, disks, network interfaces), that can be sliced between operators at very low overhead [9, 11]. For such resources, operators specify an amount of work (i.e., the effective time for which the resource is used) that can be stretched over the operator's execution time.
- Space-Shared (SS) resources (e.g., memory buffers), whose time-sharing among operators introduces prohibitively high overheads [9]. For such resources, operators typically specify rigid capacity requirements that must be satisfied throughout their execution.

Most previous work on parallel query scheduling has ignored the multi-dimensional nature of database queries and has concentrated on simplified models of SS resources, resulting in unrealistic approaches to the problem. Similar limitations exist in previous efforts within the field of deterministic scheduling theory.[1]

In our earlier work [11], we have presented a multi-dimensional framework for query scheduling in shared-nothing parallel systems with only TS resources, dealing with the full variety of bushy plans and schedules that incorporate *independent* and *pipelined* forms of inter-operation parallelism as well as intra-operation (i.e., *partitioned*) parallelism. Within this framework, we have developed a *provably* near-optimal list scheduling approach for time-sharing system resources among concurrent operators.

In this paper, we extend our previous formulation to include both TS and SS resources, representing query operator costs as pairs of *work* and *demand* vectors with one dimension per TS and SS resource, respectively. We develop a fast resource scheduling algorithm for operator pipelines called PIPESCHED that belongs to the class of *list scheduling algorithms* [14]. We then extend our approach to multiple independent pipelines, using a level-based (or, *shelf-based*) scheduling algorithm [5, 24] that treats PIPESCHED as a subroutine within each level. The resulting algorithm, termed LEVELSCHED, is analytically shown to be near-optimal for given degrees of operator parallelism. Furthermore, we show that LEVELSCHED can be readily extended to handle the operator precedence constraints in a bushy query plan as well as *on-line* task arrivals (e.g., in a dynamic or multi-query execution environment). Preliminary experimental results confirm the effectiveness of our algorithms compared to a lower bound on the optimal solution, showing that our analytical worst-case bounds are rather pessimistic compared to the average performance. Finally, we discuss the implications of our results for the open problem of designing efficient cost models for parallel query optimization [7].

*Partially supported by the National Science Foundation under Grants IRI-9113736 and IRI-9157368 (PYI Award), and by grants from IBM, DEC, HP, AT&T, Informix, and Oracle.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

[1] Due to space constraints, we do not discuss the details of earlier work. For an extensive bibliography, the interested reader is referred to the full version of the paper [12].

2 Problem Formulation

2.1 Definitions

We consider hierarchical parallel systems [2] with *identical* multiprogrammed resource sites connected by an interconnection network. Each site is a collection of d TS resources (e.g., CPU(s), disk(s), and network interface(s) or communication processor(s)) and s SS resources (e.g., memory). Although memory is probably the only SS resource that comes to mind when discussing traditional database query operators, often the distinction between TS and SS resources depends on the needs of a particular application. For example, the playback of a digitized video from a disk requires a specific fraction of the disk bandwidth throughout its execution. Clearly such an operator views the disk as an SS resource although traditional database operators view it as a TS resource. For this reason, we decided to address the scheduling problems for general s rather than restricting our discussion to $s = 1$ (i.e., memory). An obvious advantage of this general formulation is that it allows us the flexibility to "draw the line" between TS and SS resources at any boundary, depending on factors such as application requirements or user view of resources.

An *operator tree* [9, 17] is created as a "macro-expansion" of an execution plan tree by refining each node into a subtree of physical operator nodes, e.g., `scan`, `probe`, `build` (Figure 1(a,b)). Edges represent the flow of data as well as two forms of *timing* constraints between operators: pipelining (thin edges) and blocking (thick edges). Furthermore, blocking edges occasionally imply a SS *resource dependency*, where a parent task must use the same SS resources as its children in order to access their results. For instance, this is the case with the `build` operators of Figure 1(b), which must build their hash tables in memory, so that the corresponding `probe` operators, being executed immediately after them, find those tables in memory. A *query task* is a maximal subgraph of the operator tree containing only pipelining edges. A *query task tree* is created from an operator tree by representing query tasks as single nodes (Figure 1(c)).

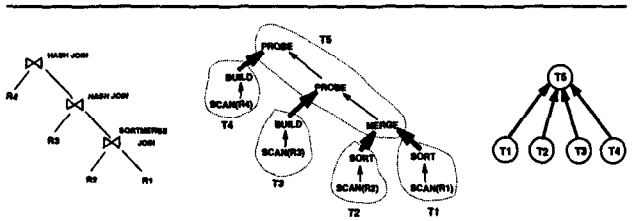

Figure 1: (a) An execution plan tree. (b) The corresponding operator tree. (c) The corresponding query task tree. The thick edges in (b) indicate blocking constraints.

The above trees clarify the definitions of the three forms of intra-query parallelism:

- *Partitioned parallelism:* A single node of the operator tree is executed on a set of sites by appropriately partitioning its input data sets.
- *Pipelined parallelism:* The operators of a single node of the task tree are executed on a set of sites in a pipelined manner.
- *Independent parallelism:* Nodes of the task tree with no path between them can be executed in parallel on a set of sites independent of each other. For example, in Figure 1, tasks T1-T4 may all be executed in parallel, whereas task T5 must await the completion of T1-T4.

The *home* of an operator is the set of sites allotted to its execution. Each operator is either *rooted*, if its home is fixed by data placement constraints (e.g., scanning a materialized intermediate relation or probing a built hash table), or *floating*, if the resource scheduler is free to determine its parallelization.

2.2 Overview

A *parallel schedule* consists of (1) an operator tree and (2) an allocation of system resources to operators. Given a query execution plan, our goal is to find a parallel schedule with minimal response time. Accounting for both TS and SS resource dimensions, our scheduling framework gives rise to interesting tradeoffs with respect to the degree of partitioned parallelism. Coarse grain operator parallelizations [8, 10, 11] are desirable since they typically result in reduced communication overhead and effective parallel executions with respect to TS resource use. On the other hand, fine grain operator parallelizations are desirable since they imply smaller SS requirements for each clone thus allowing for better load balancing opportunities and tighter schedulability conditions. A quantification of these tradeoffs and our resolution for them are presented in Section 3.1.

We have devised an algorithm for scheduling bushy execution plan trees that consists of the following steps:

1. Construct the corresponding operator and task trees, and for each operator, determine its individual resource requirements using hardware parameters, DBMS statistics, and conventional optimizer cost models (e.g., [18, 21]).
2. For each floating operator, determine the degree of parallelism based on the TS vs. SS resource tradeoffs discussed above (partitioned parallelism).
3. Place the tasks corresponding to the leaf nodes of the task tree in the *ready list* L of the scheduler. While L is not empty, perform the following steps:
   - 3.1. Determine a batch of tasks from L that can be executed concurrently and schedule them using a *provably near-optimal* multi-dimensional list scheduling heuristic (pipelined and independent parallelism).
   - 3.2. If there are tasks in the tree whose execution is enabled after Step 3.1, place them in the ready list L.

We prove that our approach is *near-optimal* for scheduling multiple independent pipelines. Further, it can be readily used to handle *on-line* task arrivals (e.g., in a dynamic or multi-query execution environment).

2.3 Assumptions

Our approach is based on the following set of assumptions:

A1. No Time-Sharing Overhead for TS Resources. Following Ganguly et al. [9], slicing a preemptable resource among multiple operators introduces no additional resource costs.

A2. Uniform TS Resource Usage. Following Ganguly et al. [9], usage of a preemptable resource by an operator is uniformly spread over the execution of the operator.

A3. Constant SS Resource Demand. The total SS requirements of an operator are constant and independent of its degree of parallelism. For example, the total amount of memory required by all the clones of a `build` operator equals the size

of a hash table on the build relation. Further, increasing the degree of parallelism does not increase the SS demands of individual clones.

A4. Non-increasing Operator Execution Times. For the range of parallelism considered, an operator's execution time is a non-increasing function of its degree of parallelism, i.e., allotting more sites cannot increase its response time.

A5. Dynamically Repartitioned Pipelined Outputs. The output of an operator in a pipeline is always repartitioned to serve as input to the next one. This is almost always accurate, e.g., when the join attributes of pipelined joins are different, the degrees of partitioned parallelism differ, or different declustering schemes must be used for load balancing.

3 Quantifying Partitioned Parallelism

3.1 A Resource Usage Model

Our treatment of TS resource usage is based on the model of preemptable resources proposed by Ganguly et al. [9], which we briefly describe here. The usage of a single resource by an operator is modeled by two parameters, T and W, where T is the elapsed time after which the resource is freed (i.e., the response time of the operator) and W is the work measured as the effective time for which the resource is used by the operator. Intuitively, the resource is kept busy by the operator only W/T of the time. Although this abstraction can model the true utilization of a system resource, it does not allow us to predict exactly when the busy periods are. Thus, we make assumption A2 which, in conjunction with assumption A1, leads to straightforward quantification of the effects of resource sharing [9].

In our previous work [11], we presented a multi-dimensional version of the model of Ganguly et al. [9] that can quantify the effects of sharing sites with TS resources among query operators. We extend that model and describe the usage by an isolated operator of a site consisting of d TS resources and s SS resources by the triple $(T^{seq}, \overline{W}, \overline{V})$, where:

- T^{seq} is the (stand-alone) sequential execution time of the operator,
- $\overline{W}$ is a d-dimensional *work vector* whose components denote the work done on individual TS resources, i.e., the effective time [9, 11] for which each resource is used by the operator; and
- $\overline{V}$ is an s-dimensional *demand vector* whose components denote the SS resource requirements of the operator throughout its execution. For notational convenience we assume that the dimensions of $\overline{V}$ are *normalized* using the corresponding SS capacities of a single site.

This generalized view of a system site is depicted in Figure 2. Our model assumes a fixed numbering of system resources for all sites; for example, dimensions 1, 2, 3, and 4 of $\overline{W}$ may correspond to CPU, disk-1, disk-2, and network interface, respectively.

Time T^{seq} is actually a function of the operator's individual resource requirements, i.e., its work vector $\overline{W}$ (sometimes emphasized by using $T^{seq}(\overline{W})$ instead of T^{seq}), and the amount of *overlap* that can be achieved between processing at different resources [11]. This overlap is a system parameter that depends on the hardware and software architecture of the resource sites (e.g., buffering architecture for disk I/O) as well as the algorithm implementing the operator. The operator's SS resource requirements

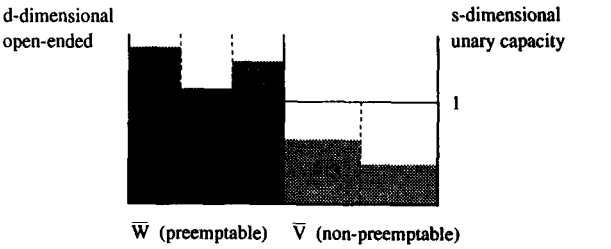

Figure 2: A site with TS and SS resources ($d = 3$, $s = 2$)

($\overline{V}$) depend primarily on the size of its inputs and the algorithm used to implement the operator. On the other hand, the operator's work requirements ($\overline{W}$) depend on both of these parameters as well as its SS resource allotment $\overline{V}$.

Note that, in this paper, we are adopting a somewhat simplified view of the SS resource demands, assuming that components of $\overline{V}$ have fixed values determined by plan parameters. In most real-life query execution engines, operator memory requirements are *malleable*, in the sense that they are typically specified as a *range* of possible memory allotments. This flexibility adds an extra level of difficulty to our scheduling problem. It means that the scheduler also has to select specific SS demand vectors $\overline{V}$ that minimize query response time over all possible $(\overline{W}, \overline{V})$ combinations. We plan to address this more general problem in our future work.

3.2 Quantifying the Granularity of Parallel Execution

As is well known, increasing the parallelism of an operator reduces its execution time until a saturation point is reached, beyond which additional parallelism causes a speed-down, due to excessive communication startup and coordination overhead over too many sites [6]. To avoid operating beyond that point, we want to ensure that the granules of the parallel execution are sufficiently coarse [8, 11]. In the presence of SS resources, any scheduling method is restricted in its mapping of clones to sites by SS capacity constraints, i.e., it is not possible to concurrently execute a set of clones at a site if their total SS requirements exceed the site's capacity (in any of the s dimensions). Clearly, coarse operator clones imply that each clone has SS resource requirements that are relatively large. This means that, when restricted to coarse grain operator executions, a scheduling method can be limited in its ability to balance the total work across sites. Furthermore, coarse SS requests can cause severe fragmentation that may lead to underutilization of system resources. Thus, taking both TS and SS resources into account gives rise to interesting tradeoffs with respect to the granularity of operator clones. Our analytical results in Section 4 clearly demonstrate this effect.

We view the granularity of a parallel operator op as a function of the ratio $\frac{W_p(\mathrm{op})}{W_c(\mathrm{op},N)}$ and $V(\mathrm{op}, N)$, where

- $W_p(\mathrm{op})$ denotes the total amount of work performed during the execution of op on a single site, when all its operands are locally resident (i.e., zero communication cost); it corresponds to the *processing area* [10] of op and is constant for all possible executions of op;
- $W_c(\mathrm{op}, N)$ denotes the total communication overhead incurred when the execution of op is partitioned among N clones; it corresponds to the *communication area* of the partitioned execution of op and is a non-decreasing function of N; and

- $V(\mathrm{op}, N)$ denotes the maximum (normalized) SS resource requirement of any clone when the execution of op is partitioned among N clones; it corresponds to the SS *grain size* of the partitioned execution of op and is a non-increasing function of N.

Note that the execution of op with degree of partitioned parallelism equal to N is feasible only if $V(\mathrm{op}, N) \leq 1$; that is, the partitioning of op must be sufficiently fine-grain for each clone to be able to maintain its SS working set at a site. We only consider such "reasonable" parallelizations in the remainder of the paper.

Definition 3.1 A parallel execution of an operator op with degree of partitioned parallelism equal to N is *λ-granular* if $V(\mathrm{op}, N) \leq \lambda$, where $\lambda \leq 1$.

The following quantification of coarse grain parallelism extends our earlier formulation [11].

Definition 3.2 A parallel execution of an operator op with degree of partitioned parallelism equal to N is *coarse grain with parameter f* (referred to as a CG_f execution) if the communication area of the execution is no more than f times the processing area of op, i.e., $W_c(\mathrm{op}, N) \leq f\, W_p(\mathrm{op})$.

Definition 3.3 A parallel execution of an operator op with degree of partitioned parallelism equal to N is *λ-granular CG_f*, if the communication area of the execution is no more than f' times the processing area of op, i.e., $W_c(\mathrm{op}, N) \leq f'\, W_p(\mathrm{op})$, where f' is the minimum value larger than or equal to f such that $V(\mathrm{op}, N) \leq \lambda$.

The intuition behind this definition is that we may sometimes have to compromise our restrictions on communication overhead to ensure that the parallelization is in the λ-granular region. This is graphically demonstrated in Figure 3.

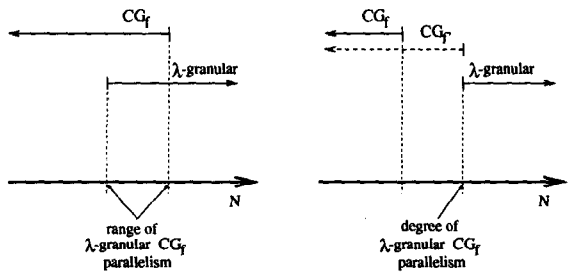

Figure 3: λ-granular CG_f execution: (a) $f = f'$, and (b) $f < f'$.

3.3 Degree of Partitioned Parallelism

Assuming zero communication costs, the TS and SS resource requirements of an operator are described by a d-dimensional work vector $\overline{W}$ and an s-dimensional demand vector $\overline{V}$ whose components can be derived from system parameters and traditional optimizer cost models [21]. By definition, the processing area of the operator $W_p(\mathrm{op})$ is simply the sum of $\overline{W}$'s components, i.e., $W_p(\mathrm{op}) = \sum_{i=1}^{d} W[i]$. Similarly, the SS grain size $V(\mathrm{op}, N)$ can be estimated using traditional optimizer cost models and statistics kept in the database catalogs [20]. Finally, we estimate the communication area $W_c(\mathrm{op}, N)$ using a simple linear model of communication costs that has been adopted in previous studies of shared-nothing architectures [11, 26] and validated on the Gamma research prototype [6]. Specifically, if D is the total size of the operator's input and output transferred over the interconnect, then $W_c(\mathrm{op}, N) = \alpha\, N + \beta\, D$, where α and β are architecture-specific parameters [11].

The following proposition is an immediate consequence of Definition 3.3 and our communication cost model.

Proposition 3.1 The maximum allowable degree of partitioned parallelism for a λ-granular CG_f execution of operator op is denoted by $N_{max}(\mathrm{op}, f, \lambda)$ and is equal to the expression

$$\max\left\{1,\ \left\lfloor \frac{f\, W_p(\mathrm{op}) - \beta\, D}{\alpha} \right\rfloor,\ \min\{N : V(\mathrm{op}, N) \leq \lambda\}\right\}.$$

4 The Scheduling Algorithm

4.1 Notation and Definitions

Table 4.1 summarizes the notation used in this section with a brief description of its semantics. Detailed definitions of some of these parameters are given below. Additional notation will be introduced when necessary.

Table 1: Notation

Parameter	Semantics
P	Number of system sites
d	Number of TS resources per site
s	Number of SS resources per site
B_j	System site (i.e., "bin") i ($i = 1, \ldots, P$)
B_j^W	Set of TS work vectors scheduled at B_j
B_j^V	Set of SS demand vectors scheduled at B_j
$T^{site}(B_j)$	Execution time for all clones at B_j
M	Number of operators to be scheduled
op_i	Operator, e.g., build ($i = 1, \ldots, M$)
N_i	Degree of partitioned parallelism (number of clones) for op_i
$\overline{W}_{\mathrm{op}_i}$	Work vector for op_i (including communication costs for N_i clones)
$\overline{V}_{\mathrm{op}_i}$	Demand vector for op_i
$T^{max}(\mathrm{op}_i, N_i)$	Maximum execution time among the N_i clones of op_i while alone in system
S	Set of (floating) clones to be scheduled
S^W (S^V)	Set of work (demand) vectors for all clones to be scheduled
S^{TV}	Set of *volume* (time × demand) vectors for all clones to be scheduled
$l(\overline{v}), l(S^v)$	Length of a vector $\overline{v}$ or set of vectors S^v

Vector $\overline{V}_{\mathrm{op}_i}$ describes the total (normalized) SS resource requirements of op_i. The components of $\overline{V}_{\mathrm{op}_i}$ are computed using architectural parameters and database statistics. Note that these components are independent of the degree of partitioned parallelism N_i.

Vector $\overline{W}_{\mathrm{op}_i}$ describes the total (i.e., processing and communication) TS resource requirements of op_i, given its degree of parallelism N_i. Using the notions of communication and processing area defined in Section 3, the above is expressed as

$$\sum_{k=1}^{d} W_{\mathrm{op}_i}[k] = W_p(\mathrm{op}_i) + W_c(\mathrm{op}_i, N_i).$$

The individual components of $\overline{W}_{\mathrm{op}_i}$ are computed using architectural parameters and database statistics, as well as the SS allotment for op_i and our model for communication costs.

Given an operator clone with a (stand-alone) execution time of T and a SS demand of $\overline{V}$, we define the *volume vector* of the clone as the product $T \cdot \overline{V}$, i.e., the resource-time product[2] for the clone's execution [3]. S^W, S^V, and S^{TV} are used to denote the set of work, demand, and volume vectors (respectively) for the set S of all the clones to be scheduled. We use the W, V, and TV superscripts in this manner throughout the paper.

The *length of a n-dimensional vector* $\overline{v}$ is its maximum component. The *length of a set* S^v *of n-dimensional vectors* is the maximum component in the vector sum of all the vectors in S^v. More formally, $l(\overline{v}) = \max_{1 \le k \le n}\{v[k]\}$ and $l(S^v) = \max_{1 \le k \le n}\{\sum_{\overline{v} \in S^v} v[k]\}$.

The *performance ratio* of a scheduling algorithm is defined as the ratio of the response time of the schedule it generates over that of the optimal schedule. All the scheduling problems addressed in this paper are non-trivial generalizations of traditional multiprocessor scheduling [14] and, thus, they are clearly $\mathcal{NP}$-hard. Given the intractability of the problems, we develop polynomial time heuristics that are *provably near-optimal*, i.e., with a constant bound on the performance ratio.

Since the parallelization of rooted operators is pre-determined, our algorithms are only concerned with the scheduling of floating operators. Also, for the purposes of this section, the degree of partitioned parallelism for all floating operators is determined based on a granularity condition, as shown in Proposition 3.1. In short, all algorithms presented in this section assume a pre-processing step that places rooted clones at their respective sites and computes the degree of coarse grain parallelism for all floating operators.

4.2 Modeling Parallelism and Resource Sharing

We present a set of extensions to the (one-dimensional) cost model of a traditional DBMS based on the multi-dimensional resource model described in Section 3.1. Our extensions account for all forms of parallelism and quantify the effects of sharing TS and SS resources on the response time of a parallel execution.

4.2.1 Partitioned and Independent Parallelism

In partitioned parallelism, the work and demand vectors of an operator are partitioned among a collection of independent *operator clones* [9]. Each clone executes on a single site and works on a portion of the operator's data. The partitioning of $\overline{W}_{\mathrm{op}_i}$ and $\overline{V}_{\mathrm{op}_i}$ into work and demand vectors for operator clones is determined based on statistical information kept in the DBMS catalogs. Given such a partitioning $< (\overline{W}_1, \overline{V}_1), (\overline{W}_2, \overline{V}_2), \ldots, (\overline{W}_{N_i}, \overline{V}_{N_i}) >$, where $\sum_{k=1}^{N_i} \overline{W}_k = \overline{W}_{\mathrm{op}_i}$ and $\sum_{k=1}^{N_i} \overline{V}_k = \overline{V}_{\mathrm{op}_i}$, a lower bound on the parallel execution time for op_i is the maximum of the sequential execution times of its N_i clones; that is, the parallel execution time for op_i is greater than or equal to

$$T^{max}(\mathrm{op}_i, N_i) = \max_{1 \le k \le N_i} \{ T^{seq}(\overline{W}_k) \}.$$

By our definitions of the TS and SS resource classes, it is obvious that a set of clones $< (\overline{W}_1, \overline{V}_1), \ldots, (\overline{W}_k, \overline{V}_k) >$ can be executed concurrently at some system site *only if* $l(\sum_1^k \overline{V}_i) \le 1$, i.e., their SS requirements do not exceed the capacity of the site. We call such clone collections *compatible*.

Definition 4.1 Given a collection of M independent operators $\{\mathrm{op}_i, i = 1 \ldots M\}$ and their respective degrees of partitioned parallelism $\{N_i, i = 1 \ldots M\}$, a *schedule* is a partitioning of the $\sum_{i=1}^{M} N_i$ operator clones into a collection of compatible subsets $S_1, \ldots, S_n$ followed by a mapping of these subsets to the set of available sites.

The effects of time-sharing the preemptable resources of a site among the clones in a compatible subset S_i can be quantified as follows. Let S_i^W denote the set of work vectors for all clones in S_i. Since all clones are executed concurrently, the execution time for the clones in S_i is determined by the ability to overlap the processing of TS resource requests by different clones. Specifically, under our model of preemptable resources described in Section 3.1, the execution time for all the operator clones in S_i is defined as [11]

$$T(S_i) = \max \left\{ \max_{\overline{W} \in S_i^W} \{T^{seq}(\overline{W})\}, \; l(S_i^W) \right\}.$$

Thus, if we let $S(B_j)$ denote the collection of compatible subsets mapped to site B_j under a given schedule SCHED, the execution time for B_j is

$$T^{site}(B_j) = \sum_{S_i \in S(B_j)} \max \left\{ \max_{\overline{W} \in S_i^W} \{T^{seq}(\overline{W})\}, \; l(S_i^W) \right\}. \quad (1)$$

Clearly, the response time of SCHED is determined by the longest running site; that is,

$$T^{par}(\mathrm{SCHED}, P) = \max_{1 \le j \le P} \{ T^{site}(B_j) \}.$$

4.2.2 Pipelined Parallelism

Pipelined parallelism introduces a co-scheduling requirement for query operators, requiring a collection of clones to execute in producer-consumer pairs using fine-grain/lock-step synchronization. The problems with load-balancing a pipelined execution have been identified in previous work [13]. Compared to our model of a schedule for partitioned and independent parallelism (Definition 4.1), pipelined execution constrains the placement and execution of compatible clone subsets to ensure that all the clones in a pipe run concurrently – they all start and terminate at the same time [16]. This means that it is no longer possible to schedule resources at one site independent of the others, as we suggested in the previous section. Compatible subsets containing clones from the same pipeline must run concurrently. Furthermore, given that the scheduler is not allowed to modify the query plan, scheduling a pipeline is an "all-or-nothing" affair: either all clones will execute in parallel or none will. The implications of pipelined parallelism for our scheduling problem will be studied further in Section 4.4 where a near-optimal solution will be developed.

4.3 Scheduling Independent Operators

In this section, we extend our earlier lower bound on the optimal parallel execution time of independent operators (i.e., operators not in any pipeline) with a new term that accounts for the effect of SS resources. We then demonstrate that a heuristic based on Graham's LPT (Largest Processing Time) *list scheduling* method [14] can guarantee near-optimal schedules for such operators.

[2] The *volume* of an operator is defined as the product of the amount of resource(s) that the operator reserves during its execution and its execution time.

Theorem 4.1 Let $\{\mathrm{op}_i, i = 1, \ldots M\}$ be independent operators with respective degrees of partitioned parallelism $\underline{N} = (N_1, N_2, \ldots, N_M)$. Let S be the corresponding set of clones and define $T^{max}(S) = \max_{i=1,\ldots,M}\{T^{max}(\mathrm{op}_i, N_i)\}$. If $T^{par}(\mathrm{OPT}, P)$ is the response time of the optimal execution on P sites then $T^{par}(\mathrm{OPT}, P) \geq LB(S, P)$, where

$$LB(S,P) = \max\left\{ T^{max}(S) , \frac{l(S^W)}{P} , \frac{l(S^{TV})}{P} \right\}.$$

As with all theoretical results presented here, Theorem 4.1 is stated without proof due to space constraints. The details can be found in the full version of this paper [12]. Compared to our earlier results [11], the lower bound in Theorem 4.1 introduces a third term containing $l(S^{TV})$, i.e., the total *volume* of the parallel execution. We will see that this new parameter plays an important role in our analytical and experimental results.

The basic idea of our heuristic scheduling algorithm, termed OPSCHED, is to construct the partition of clones into compatible subsets incrementally, using a Next-Fit rule [4, 5]. Specifically, OPSCHED scans the list of clones in non-increasing order of execution time. At each step, the clone selected is placed in the site B_i of minimal height $T^{site}(B_i)$ (see Equation (1)). This placement is done as follows. Let S_{i,n_i} denote the topmost compatible subset in B_i. If the clone can fit in S_{i,n_i} without violating SS capacity constraints, then add the clone to S_{i,n_i} and update $T^{site}(B_i)$ accordingly. Otherwise, set $n_i = n_i + 1$, place the clone by itself in a new topmost subset S_{i,n_i}, and set $T^{site}(B_i)$ accordingly. The following theorem establishes an absolute performance bound of $d + 2s + 2$ for our heuristic.

Theorem 4.2 Given a set of clones S, OPSCHED runs in time $O(|S| \log |S|)$ and produces a schedule SCHED with response time $T^{par}(\mathrm{SCHED}, P) \leq (d + 2s + 2) \cdot LB(S, P)$.

4.4 Scheduling with Pipelining Constraints

The co-scheduling requirement of pipelined operator execution introduces an extra level of complexity that OPSCHED cannot address, namely the problem of deciding whether a pipeline is *schedulable* on a given number of sites. Given a collection of operator clones in a pipeline, the schedulability question poses an $\mathcal{NP}$-hard decision problem that essentially corresponds to the decision problem of s-dimensional vector packing [4]. Thus, it is highly unlikely that efficient (i.e., polynomial time) necessary and sufficient conditions for pipeline schedulability exist. Note that no such problems were raised in the previous section, since the clones were executing independently of each other.

In this section, we show that λ-granularity with $\lambda < 1$ for all operator parallelizations can provide an easily checkable sufficient condition for pipeline schedulability[3]. Once schedulability is ensured, balancing the work of the pipeline across sites to minimize its response time still poses an $\mathcal{NP}$-hard optimization problem. We present a polynomial time scheduling algorithm that is within a constant multiplicative factor of the response time lower bound for schedulable λ-granular pipelines. Further, we demonstrate that using a level-based approach, our methodology can be extended to provide a provably near-optimal solution for multiple independent pipelines. Finally, we extend our techniques to handle the data dependencies in a bushy query plan and on-line task arrivals.

[3] We use the term *λ-granular pipeline* to describe a pipeline in which all operator parallelizations are λ-granular.

4.4.1 Scheduling a Single λ-granular Pipeline

We present a near-optimal algorithm for scheduling a pipeline C consisting of λ-granular parallel operators, where $\lambda < 1$. Let S_C denote the collection of clones in C and define S_C^W, S_C^V, and $T^{max}(S_C)$ in the obvious manner. Note that, by our definitions, the pipeline C will require *at least* $l(S_C^V)$ sites for its execution (otherwise, λ would have to be greater than 1). The following lemma provides a sufficient condition for schedulability.

Lemma 4.1 The number of sites required to schedule a λ-granular pipeline C is always less than or equal to $l(S_C^V) \cdot s/(1 - \lambda)$.

Our heuristic, PIPESCHED, belongs to the family of list scheduling algorithms [14]. PIPESCHED assumes that it is given a number of sites P_C that is sufficient for the scheduling of C, according to the condition of Lemma 4.1. The algorithm considers the clones in S_C in non-increasing order of their *work density ratio* $\frac{l(\overline{W}_i)}{l(\overline{V}_i)}$. At each step, the clone under consideration is placed in the least filled (i.e., least work) site that has sufficient SS resources to accommodate it; that is, clone $(\overline{W}_i, \overline{V}_i)$ is packed in bin B such that $l(B^W)$ is minimal among all sites B_j such that $l(B_j^V \cup \{\overline{V}_i\}) \leq 1$. The PIPESCHED algorithm is depicted in Figure 4. The following theorem bounds the worst-case performance ratio of our algorithm[4].

Theorem 4.3 Given a λ-granular pipeline C, PIPESCHED runs in time $O(|S_C| \log |S_C|)$ and produces a schedule SCHED with response time

$$T^{par}(\mathrm{SCHED}, P_C) \leq [d(1+\frac{s}{1-\lambda})+1] \cdot \max\left\{ T^{max}(S_C) , \frac{l(S_C^W)}{P_C} \right\}.$$

Algorithm PIPESCHED(C, P_C)
Input: A set of λ-granular pipelined operator clones S_C and a set of P_C sites, where $P_C \geq \frac{l(S_C^V) \cdot s}{1-\lambda}$ (see Lemma 4.1).
Output: A mapping of the clones to sites that does not violate SS resource capacity constraints.

1. let $L =< (\overline{W}_1, \overline{V}_1), \ldots, (\overline{W}_N, \overline{V}_N) >$ be the list of all clones in *non-increasing* order of $\frac{l(\overline{W}_i)}{l(\overline{V}_i)}$.
2. for $k = 1$ to N do
   2.1. let $SB_k = \{B_j : l(B_j^V \cup \{\overline{V}_k\}) \leq 1\}$, i.e, the set of bins with sufficient SS resources for the k^{th} clone.
   2.2. let $B \in SB_k$ be a site such that $l(B^W) = \min_{B_j \in SB_k}\{l(B_j^W)\}$.
   2.3. place clone $(\overline{W}_N, \overline{V}_N)$ at site B and set $B^W = B \cup \{\overline{W}_N\}$, $B^V = B \cup \{\overline{V}_N\}$.

Figure 4: Algorithm PIPESCHED

The bound established in Theorem 4.3 clearly captures the granularity tradeoffs identified in Section 3. Increasing the degree of partitioned parallelism decreases both $T^{max}(S_C)$ and λ

[4] Note that the volume term does not come into the expression for the performance bound of PIPESCHED. This is because, by definition, all the clones in S_C must execute in parallel, so $l(S_C^V) \leq P_C$. Thus, for the execution of a single pipeline, $\frac{l(S_C^{TV})}{P_C} \leq T^{max}(S_C) \cdot \frac{l(S_C^V)}{P_C} \leq T^{max}(S_C)$.

because of "finer" operator clones, but it also increases the total amount of work $l(S_C^W)$ because of the overhead of parallelism. The importance of such work-space tradeoffs for parallel query processing and optimization has been stressed in recent work [15].

4.4.2 Scheduling Multiple Independent Pipelines

The basic observation here is that the PIPESCHED algorithm presented in the previous section can be used to schedule any collection of independent pipelines as long as schedulability is guaranteed by Lemma 4.1.

Our algorithm for scheduling multiple independent pipelines uses a Next-Fit Decreasing Height (NFDH) policy [5] in conjunction with Lemma 4.1 to identify pipelines that can be scheduled to execute concurrently on P sites (i.e., in one layer of execution). PIPESCHED is then used for determining the execution schedule within each layer. The overall algorithm, LEVELSCHED, is formally outlined in Figure 5.

Algorithm LEVELSCHED($\{C_1, \ldots, C_N\}, P$)
Input: A set of λ-granular operator pipelines $\{C_1, \ldots, C_N\}$ and a set of P sites.
Output: A mapping of clones to sites that does not violate SS resource capacity constraints or pipelining dependencies.

1. Sort the pipelines in non-increasing order of T^{max}, i.e., let $L =< C_1, \ldots, C_N >$, where $T^{max}(S_{C_1}) \geq \ldots \geq T^{max}(S_{C_N})$.
2. Partition the list L in k maximal schedulable sublists: $L_1 =< C_1, \ldots, C_{i_1} >, L_2 =< C_{i_1+1}, \ldots, C_{i_2} >, \ldots, L_k =< C_{i_{k-1}+1}, \ldots, C_N >$ based on Lemma 4.1; that is,

$$l(\cup_{C \in L_j} S_C^V) \leq \frac{P(1-\lambda)}{s} \quad \text{and}$$

$$l((\cup_{C \in L_j} S_C^V) \cup S_{C_{i_j+1}}^V) > \frac{P(1-\lambda)}{s}, \quad \text{for all } j.$$

3. for $j = 1, \ldots, k$ do
   3.1. call PIPESCHED($(\cup_{C \in L_j} C)$, P)

Figure 5: Algorithm LEVELSCHED

The following theorem gives an upper bound on the worst-case performance ratio of LEVELSCHED. Note that the co-scheduling requirement for the clones in a pipe implies that the total volume for all the clones in $\{C_1, \ldots, C_N\}$ is $l(S^{TV}) = l(\sum_1^N T_{C_i}^{max} \cdot \sum_{\overline{v} \in S_{C_i}^V} \overline{v})$, since any clone in C_i will require its share of ss resources for at least $T_{C_i}^{max}$ time. The lower bound in Theorem 4.1 holds using the above definition of volume.

Theorem 4.4 Given a collection of N independent λ-granular pipelines comprising a set of clones S, LEVELSCHED runs in time $O(N|S| \log P|S|)$ and produces a schedule SCHED with response time

$$T^{par}(\text{SCHED}, P) \leq [d^2(1+\frac{s}{1-\lambda})+\frac{2s^2}{1-\lambda}+1] \cdot LB(S, P).$$

It is important to note that, in most cases, the lower bound estimated in Theorem 4.1 will significantly underestimate the optimal response time since it assumes that 100% utilization of system resources is always possible *independent of the given task list*. Thus, the quadratic multiplicative constants in Theorem 4.4 reflect only a worst case that is rather far from the average, as our experimental results have verified as well.

4.5 Data Dependencies and On-Line Task Arrivals

Scheduling arbitrary query task trees must ensure that the blocking constraints specified by the tree's edges are satisfied. The LEVELSCHED algorithm can be readily extended to handle such constraints by ensuring that the (sorted) ready list of tasks L always contains the collection of query tasks that are ready for execution, i.e., they are not blocked waiting for the completion of some other (descendant) task in the task tree. In addition, as mentioned in Section 2, care must be taken to ensure that whenever there is a SS resource dependency between parent and children tasks across blocking edges, the operators of all such children are co-scheduled and those of the parent are treated as *rooted* and scheduled in the immediately following shelf. All this is done by modifying LEVELSCHED as follows (see Figure 5):

1. Any sibling pipelines in the task tree with SS resource dependencies are treated as a unit, i.e., the way individual pipelines are treated in LEVELSCHED. For the purposes of this algorithm, assume that the term 'pipeline' is interpreted as such a unit.
2. Initially, the input set of pipelines $\{C_1, \ldots, C_N\}$ contains exactly the tasks at the leaf nodes of the query task tree.
3. After Step 3.1, determine the set of tasks C that have been enabled (i.e., are no longer blocked) because of the last invocation of PIPESCHED. If C$\neq \emptyset$, then merge the tasks in C into the ready list L and go to Step 2. Otherwise, continue with the next invocation of PIPESCHED.

The exact same idea of dynamically updating and partitioning the ready list L can be used to handle *on-line* task arrivals in a dynamic or multi-query environment. Basically, newly arriving query tasks are immediately merged into L to participate in the partitioning of L into schedulable sublists right after the completion of the current execution layer. Thus, our layer-based approach provides a uniform scheduling framework for handling intra-query as well as inter-query parallelism.

As we have already indicated in our earlier work [11], deriving performance bounds in the presence of data dependencies is a very difficult problem that continues to elude our efforts. The difficulty stems from the interdependencies between different execution layers: scheduling decisions made at earlier layers can impose data placement and operator execution constraints on the layers that follow. We leave this problem open for future research.

5 Experimental Performance Evaluation

5.1 Experimental Testbed

We have experimented with the following algorithms:

- TREESCHED: Level-based scheduling of task trees, observing blocking and data placement constraints (Section 4.5).
- LEVELSCHED: Level-based scheduling of multiple independent query tasks (Section 4.4.2).

For both scheduling scenarios (task trees, independent tasks), we compared the average performance of our scheduling algorithms with a lower bound on the response time of the optimal execution schedule for the given degrees of partitioned parallelism (determined by the granularity parameters λ and f). This lower bound

for LEVELSCHED follows directly from Theorem 4.1, whereas, for TREESCHED, we used the formula:

$$\text{TREEBOUND} = \max\{\ \frac{l(S^W)}{P}\ ,\ \frac{l(S^{TV})}{P}\ ,\ T(\text{CP})\ \},$$

where S is the collection of all operator clones in the task tree and $T(\text{CP})$ is the total response time of the critical (i.e., most time-consuming) path in the task tree. Due to space constraints, we only discuss our results for TREESCHED and refer the interested reader to the full paper [12] for more details.

Some additional assumptions were made to obtain a specific experimental model from the general parallel execution model described in Sections 3 and 4. These are briefly summarized below:

EA1. No Data or Execution Skew: With the exception of startup cost, the TS work and SS demand vectors of an operator are distributed perfectly among all sites participating in its execution. Startup is added to the work components of only one of these sites, the "coordinator" for the parallel execution.

EA2. Uniform TS Resource Overlapping: The amount of overlap achieved between processing at different TS resources at a site can be characterized by a single system-wide parameter $\epsilon \in [0, 1]$ for all query operators. This parameter allows us to express the response time of a work vector as a convex combination of the maximum and the sum of its components, i.e., $T^{seq}(\overline{W}) = \epsilon \max_{1 \le i \le d}\{W[i]\} + (1-\epsilon)\sum_{i=1}^{d} W[i]$. Small values of ϵ imply limited overlap, whereas values closer to 1 imply a larger degree of overlap.

EA3. Simple Hash-Join Plan Nodes: The query plan consists of hash-join nodes, where the memory demand for each join equals the size of the inner relation times a "fudge factor" accounting for the hash table overhead. Note that although it is possible to execute a hash-join with less memory [22], such memory limitations complicate the processing of multi-join pipelines – since `probe` operators cannot keep their entire data sets (i.e., inner hash tables) in memory, it is no longer possible to execute the `probe` pipeline in one pass. This means that intermediate disk I/O has to be performed at one or more pipeline stages, essentially modifying the original plan with the addition of extra blocking and data dependencies. As part of our future work, we plan to investigate the effects of such memory limitations on our scheduling methodology and results.

Finally, we should note that our implementations incorporated an additional optimization to the basic scheme: *After the placement of a schedulable sublist (Lemma 4.1) of ready pipelines, each remaining ready pipeline was checked (in non-increasing order of T^{max}) for possible inclusion in the current level before starting the next execution level.* Although this optimization does not help improve worst-case performance (since Lemma 4.1 is tight), we found that it really helped the average-case performance of the heuristics at the cost of a small increase in running time.

We experimented with tree queries of 10, 20, 30, 40, and 50 joins. For each query size, twenty query graphs (trees) were randomly generated and for each graph an execution plan was selected in a random manner from a bushy plan space. We assumed simple key join operations in which the size of the result relation is always equal to the size of the largest of the two join operands. We used two performance metrics in our study: (1) the *average performance ratio* defined as the response time of the schedules produced by our heuristics divided by the corresponding lower bound and averaged over all queries of the same size; and, (2) the *average response time* of the schedules produced by our heuristics over all queries of the same size. Experiments were conducted with various combinations of values for the λ, f, and ϵ parameters. Since the effects of f and ϵ on scheduler performance were also studied in our prior work [11], the discussion in this paper mostly concentrates on the new parameter λ. (The results presented in the next section are indicative of the results obtained for all values of f, ϵ.)

In all experiments, we assumed system nodes consisting of $d = 3$ TS resources (one CPU, one disk unit, and one network interface) and $s = 1$ SS resource (memory). The work vector components for the CPU and the disk were estimated using the cost model equations given by Hsiao et al. [18]. The communication costs were calculated using the model described in Section 3. The values of the cost model parameters were obtained from the literature [18, 10, 26] and are summarized in Table 5.1.

Table 2: Experiment Parameter Settings

Configuration/DB Catalog Parameters	Value
Number of Sites	10 – 120
CPU Speed	10 MIPS
Memory per Site	16 – 64 MB
Effective Disk Service Time per page	5 msec
Startup Cost per site (α)	15 msec
Network Transfer Cost per byte (β)	0.3 μsec
Fudge Factor (F)	1.4
Tuple Size	128 bytes
Page Size	32 tuples
Relation Size	10^4 – 10^6 tuples
CPU Cost Parameters	**No. Instr.**
Read Page from Disk	5000
Write Page to Disk	5000
Extract Tuple	300
Hash Tuple	100
Probe Hash Table	200

5.2 Experimental Results

Figure 6(a) depicts the average performance ratio of TREESCHED as a function of system size for 40-join queries and 32MB of memory at each site, assuming a coarse-granularity parameter $f = 0.6$ and a resource overlap of 50% (i.e., $\epsilon = 0.5$). Note that our algorithm is consistently within a small constant factor (i.e., less than 2) of the lower bound on the optimal schedule length. Although the distance from the lower bound has certainly increased compared to our results for only TS resources [11], the results clearly demonstrate that the worst-case multiplicative factors derived in our analytical bounds are overly pessimistic as far as average performance is concerned.

Observing Figure 6(a), it appears that TREESCHED performs better for larger values of the memory granularity parameter λ. This is slightly counterintuitive and seems to contradict Section 4.4: "finer" memory requirements should allow our schedulers to obtain better load balancing and, consequently, better schedules. However, although the *performance ratio* of the algorithms improves with larger values of λ, the *actual performance* (i.e., the response time) of the schedules (shown in Figure 6(b)), deteriorates with larger λ, as expected. The explanation of this phenomenon lies in Figure 7, which shows how the three components of the TREEBOUND lower bound vary with the number of sites for our example set of 40-join queries and 16MB of memory per site. (We chose a smaller value for memory because it better

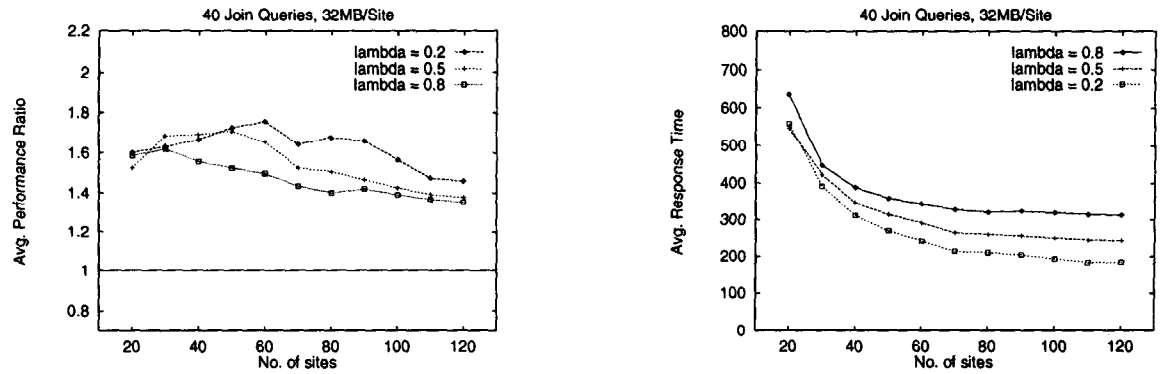

Figure 6: Effect of λ on (a) the average performance ratio of TREESCHED, and (b) the average schedule response times obtained by TREESCHED. ($f = 0.6, \epsilon = 0.5$)

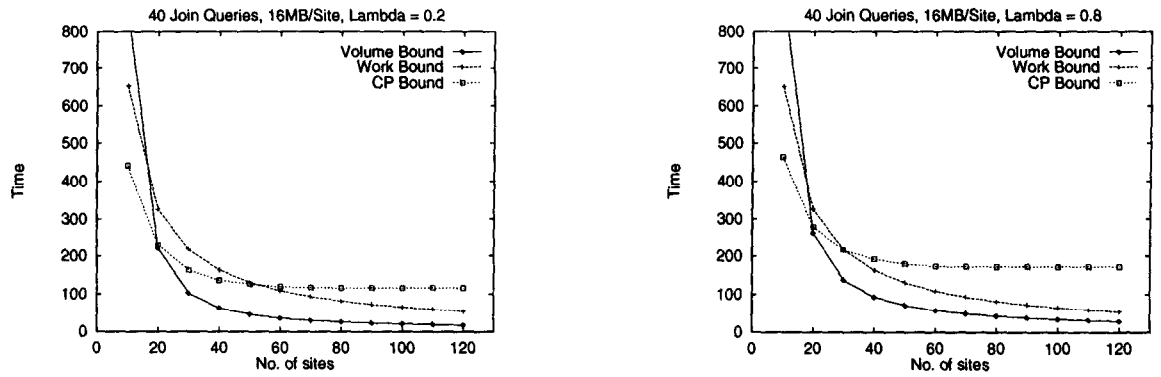

Figure 7: TREEBOUND components for (a) $\lambda = 0.2$, and (b) $\lambda = 0.8$. ($f = 0.6, \epsilon = 0.5$)

illustrates the effect of the volume term for our system setting.) In particular, for small values of the number of sites P, the dominant factor in TREEBOUND is the average volume term ($\frac{l(S^{TV})}{P}$). For larger values of P (and, consequently, increased system memory), TREEBOUND is determined by the average work term ($\frac{l(S^{W})}{P}$). Eventually, as P continues to grow, the critical path term ($T(\text{CP})$) starts dominating the other two terms in the bound. Also, note that as the critical path becomes the dominant factor in the query plan execution, our level-based methods become more accurate in approximating the lower bound. Intuitively, this is because the "height" of each execution level as determined by the plan's critical path will be sufficient to pack the work in that level and, thus, the resource loss due to "shelving" is not important. (This also explains why the average performance ratios for various values of λ all converge to a value close to 1 as the number of sites increases.) For the parameter settings in our experiments, larger values for the memory granularity λ typically imply lower degrees of parallelism for the operators in the plan, which means that the critical path will start dominating the other two factors in TREEBOUND much sooner. Furthermore, the aforementioned effect on the performance ratio becomes more pronounced since the critical path term will be significantly larger for larger λ (Figure 7). Consequently, larger values for λ imply better performance ratios, although the actual schedule response times are worse.

6 Parallel Query Optimization

Perhaps the major difference between parallel query optimization and its well-understood centralized counterpart lies in the choice of *response time* as a more appropriate optimization metric. This choice of metric implies that a parallel query optimizer cannot afford to ignore resource scheduling during the optimization process. Prior work has demonstrated that a *two-phase* approach [17] using the traditional work (i.e., resource consumption) metric during the plan generation phase often results in plans that are inherently sequential and, consequently, unable to exploit the available parallelism [1]. On the other hand, using a detailed resource scheduling model during plan generation (as advocated by the *one-phase* approach [19, 23]) can have a tremendous impact on optimizer complexity and optimization cost. For example, a Dynamic Programming (DP) algorithm must use much stricter pruning criteria that account for the use of system resources [9, 19]. This leads to a combinatorial explosion in the state that must be maintained while building the DP tree, rendering the algorithm impractical even for small query sizes.

The role of the optimizer cost model is to provide an abstraction of the underlying execution system. In this respect, the one- and two-phase approaches lie at the two different ends of a spectrum, incorporating either detailed knowledge (one-phase) or no knowledge (two-phase) of the parallel environment in the optimizer cost metric. The goal is to devise cost metrics that are more realistic than resource consumption, in the sense that they are cognizant of the available parallelism, and at the same time are sufficiently efficient to keep the optimization process tractable. In recent work, Ganguly et al. [8] suggested the use of a novel scalar cost metric for parallel query optimization. Their metric was defined as the maximum of two "bulk parameters" of a parallel query plan, namely the critical path length of the plan tree and the aver-

age work per site. Although the model used in the work of Ganguly et al. was one-dimensional, it is clear that the "critical path length" corresponds to the maximum sum of T^{max}'s in the task tree (over all root-to-leaf paths), whereas the "average work" corresponds to $\frac{l(S^W)}{P}$ with S being all operator clones in the plan.

Based on our analytical and experimental results, there clearly exists a third parameter, namely the *average volume per site* $\frac{l(S^{TV})}{P}$ that is an essential component of query plan quality. The importance of this third parameter stems from the fact that it is the only one capturing the constraints on parallel execution that derive from SS (i.e., memory) resources.

We believe that the triple *(critical path, average work, average volume)* captures all the crucial aspects characterizing the expected response time of a parallel query execution plan. Consequently, we feel that these three components can provide the basis for an efficient and accurate cost model for parallel query optimizers. Finally, note that although Ganguly et al. [8] suggested combining the plan parameters through a $\max\{\}$ function to produce a scalar metric, the way these parameters are used should depend on the optimization strategy. For example, a DP-based parallel optimizer should use our three "bulk parameters" as a 3-dimensional vector and use a 3-dimensional "less than" to prune the search space [9]. Clearly, using only three dimensions turns the Partial Order DP (PODP) approach of Ganguly et al. [9] into a feasible and efficient paradigm for DP-based parallel query optimization.

7 Conclusions

The problem of scheduling complex queries in hierarchical parallel database systems of multiple time-shared and space-shared resources has been open for a long time both within the database field and the deterministic scheduling theory field. Despite the importance of such architectures in practice, the difficulties of the problem have led researchers in making various assumptions and simplifications that are not realistic. In this paper, we have provided what we believe is the first comprehensive formal approach to the problem. We have established a model of resource usage that allows the scheduler to explore the possibilities for concurrent operations sharing both TS and SS resources and quantify the effects of this sharing on the parallel execution time. The inclusion of both types of resources has given rise to interesting tradeoffs with respect to the degree of partitioned parallelism, which are nicely exposed within our analytical models and results, and for which we have provided some effective resolutions. We have provided efficient, near-optimal heuristic algorithms for query scheduling in such parallel environments, paying special attention to various constraints that arise from the existence of SS resources, including the co-scheduling requirements of pipelined operator execution, which has been the most challenging to resolve. Our set of results apply to all types of query plans and even sets of plans that are either provided all at the beginning or arrive dynamically for scheduling. As a side-effect of our effort, we have identified an important parameter that captures one aspect of parallel query execution cost, which should play an important role in obtaining realistic cost models for parallel query optimization.

References

[1] C.K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G.P. Copeland, and W.G. Wilson. "DB2 Parallel Edition". *IBM Systems Journal*, 34(2):292–322, 1995.

[2] L. Bouganim, D. Florescu, and P. Valduriez. "Dynamic Load Balancing in Hierarchical Parallel Database Systems". In *Proc. of the 22nd Intl. VLDB Conf.*, September 1996.

[3] S. Chakrabarti and S. Muthukrishnan. "Resource Scheduling for Parallel Database and Scientific Applications". In *Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures*, June 1996.

[4] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson. "Approximation Algorithms for Bin-Packing – An Updated Survey". In *"Algorithm Design for Computing System Design"*, Springer-Verlag, NY, 1984.

[5] E.G. Coffman, Jr., M.R. Garey, D.S. Johnson, and R.E. Tarjan. "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms". *SIAM Journal on Computing*, 9(4):808–826, 1980.

[6] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. "The Gamma Database Machine Project". *IEEE Trans. on Knowledge and Data Engineering*, 2(1):44–62, 1990.

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

[8] S. Ganguly, A. Goel, and A. Silberschatz. "Efficient and Accurate Cost Models for Parallel Query Optimization". In *Proc. of the 15th ACM PODS Symp.*, June 1996.

[9] S. Ganguly, W. Hasan, and R. Krishnamurthy. "Query Optimization for Parallel Execution". In *Proc. of the 1992 ACM SIGMOD Intl. Conf.*, June 1992.

[10] S. Ganguly and W. Wang. "Optimizing Queries for Coarse Grain Parallelism". Tech. Report LCSR-TR-218, Dept. of Computer Sciences, Rutgers University, October 1993.

[11] M. N. Garofalakis and Y. E. Ioannidis. "Multi-dimensional Resource Scheduling for Parallel Queries". In *Proc. of the 1996 ACM SIGMOD Intl. Conf.*, June 1996.

[12] M. N. Garofalakis and Y. E. Ioannidis. "Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources". Unpublished manuscript, June 1997.

[13] G. Graefe. "Query Evaluation Techniques for Large Databases". *ACM Comp. Surveys*, 25(2):73–170, 1993.

[14] R.L. Graham. "Bounds on Multiprocessing Timing Anomalies". *SIAM Journal on Computing*, 17(2):416–429, 1969.

[15] W. Hasan, D. Florescu, and P. Valduriez. "Open Issues in Parallel Query Optimization". *ACM SIGMOD Record*, 25(3):28–33, 1996.

[16] W. Hasan and R. Motwani. "Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism". In *Proc. of the 20th Intl. VLDB Conf.*, August 1994.

[17] W. Hong. "Exploiting Inter-Operation Parallelism in XPRS". In *Proc. of the 1992 ACM SIGMOD Intl. Conf.*, June 1992.

[18] H.-I Hsiao, M.-S. Chen, and P. S. Yu. "On Parallel Execution of Multiple Pipelined Hash Joins". In *Proc. of the 1994 ACM SIGMOD Intl. Conf.*, May 1994.

[19] R. S.G. Lanzelotte, P. Valduriez, and M. Zaït. "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces". In *Proc. of the 19th Intl. VLDB Conf.*, August 1993.

[20] V. Poosala and Y. E. Ioannidis. "Estimation of Query-Result Distribution and its Application in Parallel-Join Load Balancing". In *Proc. of the 22nd Intl. VLDB Conf.*, September 1996.

[21] 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 *Proc. of the 1979 ACM SIGMOD Intl. Conf.*, June 1979.

[22] L. D. Shapiro. "Join Processing in Database Systems with Large Main Memories". *ACM Trans. on Database Systems*, 11(3):239–264, 1986.

[23] J. Srivastava and G. Elsesser. "Optimizing Multi-Join Queries in Parallel Relational Databases". In *Proc. of the 2nd Intl. Conf. on Parallel and Distributed Information Systems*, January 1993.

[24] J. Turek, J. L. Wolf, K. R. Pattipati, and P. S. Yu. "Scheduling Parallelizable Tasks: Putting it All on the Shelf". In *Proc. of the 1992 ACM SIGMETRICS Conf.*, June 1992.

[25] P. Valduriez. "Parallel Database Systems: Open Problems and New Issues". *Distributed and Parallel Databases*, 1:137–165, 1993.

[26] A. N. Wilschut, J. Flokstra, and P. M.G. Apers. "Parallelism in a Main-Memory DBMS: The Performance of PRISMA/DB". In *Proc. of the 18th Intl. VLDB Conf.*, August 1992.