Managing Intra-operator Parallelism in Parallel Database Systems*

Manish Mehta
IBM Alamaden Research Center
San Jose, CA, USA.
mmehta@almaden.ibm.com

David J. DeWitt
Computer Sciences Department
University of Wisconsin-Madison
dewitt@cs.wisc.edu

Abstract

Intra-operator (or partitioned) parallelism is a well-established mechanism for achieving high performance in parallel database systems. However, the problem of how to exploit intra-operator parallelism in a multi-query environment is not well understood. This paper presents a detailed performance evaluation of several algorithms for managing intra-operator parallelism in a parallel database system. A dynamic scheme based on the concept of matching the rate of flow of tuples between operators is shown to perform well on a variety of workloads and configurations.

1 Introduction

Highly-parallel database systems are increasingly being used for large-scale database applications. Examples of these systems include products like Teradata DBC/1012 [26], Tandem Himalaya [13], IBM SP1/2 [16], and research prototypes like Bubba [8], Gamma [7], and Volcano [12]. *Intra-operator parallelism* (or partitioned parallelism [4]) is a well-established technique for increasing performance in these systems. Basically, by allowing the input data to be partitioned among multiple processors and memories, this technique enables a database operator to be split into many independent operators each working on a part of the data. However, it is not evident how intra-operator parallelism should be used in a multi-query environment, where multiple concurrent queries are contending for system resources. Selecting low degrees of operator parallelism can lead to underutilization of the system and reduced performance. On the other hand, high degrees of parallelism can give "too many" resources to a single query and lead to high resource contention. This paper explores this problem of determining the "appropriate" degree of intra-operator parallelism for queries in a multi-query parallel database system.

There are two important issues that need to be addressed. First, for each query, the algorithm must determine the degree of parallelism of each operator in the query plan. Second, the algorithm must assign specific processors to execute each operator instance.

The degree of parallelism of an operator should be selected such that the cost of starting and terminating all of the instances of the operator is more than offset by the performance improvement due to parallelism. Since startup and termination costs are a function of the configuration and workload, the degree of parallelism should change for different workloads and configurations. In this paper, we present several algorithms for determining the degree of parallelism of operators. A detailed performance evaluation shows that a new dynamic algorithm based on the concept of matching the rate of flow of tuples between operators provides good performance across a variety of workloads and configurations.

The primary objective of an algorithm for assigning processors to operators is load balancing. Processors should be assigned to operators such that the workload is uniformly distributed and all the nodes are equally utilized. This paper presents several alternative methods of assigning processors to operators. The results show that algorithms that utilize information about the system workload in assignment decisions perform better than algorithms that assign processors stati-

*This research was performed while the first author was at the University of Wisconsin-Madison and was supported by an IBM Research Initiation Grant.

Proceedings of the 21st VLDB Conference
Zurich, Swizerland, 1995

cally.

The rest of the paper is organized as follows. Section 2 presents the system architecture of a typical highly-parallel database system. The algorithms for selecting the degree of parallelism are presented in Section 3 while algorithms for mapping operators to processors are presented in Section 4. Section 5 discusses how these algorithms can be combined to produce a comprehensive processor allocation scheme. The simulation model used for the performance evaluation is described in Section 6 followed by a description of the experimental parameters in Section 7. Section 8 presents a performance evaluation of algorithms for determining the degree of parallelism and Section 9 contains the evaluation of algorithms for mapping the operators to processors. Related work is discussed in Section 10 and Section 11 contains our conclusions and suggestions for future work.

2 System Architecture

Highly parallel systems are typically constructed using a shared-nothing [25] architecture. The system consists of a set of external terminals from which transactions are submitted. The transactions are sent to a randomly-selected scheduling node. The execution of each transaction on the processing nodes is coordinated by a specialized process called the *scheduler*. The scheduler allocates resources (memory and processors) to the transaction and is responsible for starting and terminating all the operators in a transaction. The processing nodes are composed of a CPU, memory, and one or more disk drives[1]. There is no shared memory/disk between the nodes, hence the term Shared-Nothing. All inter-node communication is via message passing on the interconnection network.

3 Determining the Degree of Operator Parallelism

In most shared-nothing database systems, the only way to access data on a node is to schedule a select operator on that node. This implies that the degree of parallelism of select operators is fixed a-priori by the data placement mechanism. However, the degree of parallelism for other operators, like joins and stores, can be chosen independently of the initial data placement. We consider four algorithms for determining the degree of parallelism of such operators.

3.1 Maximum

The degree of parallelism chosen by this algorithm is equal to the number of nodes in the system. Maximum therefore achieves the highest parallelism, but it also has the highest startup and termination costs and leads to the highest resource contention. Moreover, it is a static algorithm that selects the same degree of parallelism for *all* operators regardless of the query type and the workload.

3.2 MinDp

The degree of parallelism selected by this algorithm is equal to the minimum of the degree of parallelism of all the input streams. For example, consider a binary hash-join query where the degrees of parallelism of the selects on the inner and outer relations are Inner-dp and Outer-dp, respectively. The MinDp algorithm will select the join's degree of parallelism as min(Inner-dp, Outer-dp).

3.3 MaxDp

The MaxDp algorithm is sets the degree of join parallelism to be the maximum of the degree of parallelism of the input streams. Note that in the case of unary operators like store, the MaxDp and MinDp algorithms are identical.

3.4 RateMatch

The RateMatch algorithm is based on the idea of matching processing rates of operators. If the rate at which tuples are sent to an operator is much higher than the rate at which the tuples can be processed by the operator, incoming messages will accumulate and the message buffer can overflow, forcing the sender to block. On the other hand, if the rate at which tuples are received by an operator is much lower than the maximum possible processing rate, the operator will frequently be idle and will waste system resources (specifically memory). By matching operator processing rates, the RateMatch algorithm prevents senders from blocking and, at the same time, conserves system resources by avoiding idle operators.

We next present the formulas used by the RateMatch algorithm to calculate the rate at which tuples are processed by the select and hash-join operators. Similar formulas can also be developed for other database operators. These formulas adjust the degree of parallelism based on the current CPU utilization and disk response times, and therefore allow the RateMatch algorithm to adapt to different workloads and configurations. We first develop formulas for a single-user system assuming no buffering of messages, and then incorporate the effect of multiple users and message buffering.

3.4.1 Processing Rate of Select Operators

The total time (in seconds) taken by each select operator to process a data page is

$$T_{select} = MAX(T_{I/O_{Select}}, T_{CPU_{Select}})$$

[1]For the rest of this paper, the term *node* is used to collectively refer to a processor, its local memory and the attached set of disks.

where $T_{I/O_{Select}}$ is the time taken to perform one I/O and $T_{CPU_{Select}}$ is the CPU time taken for processing one data page. Note that the above equation assumes an overlap in CPU and I/O processing. $T_{I/O_{Select}}$ is calculated as the sum of the time taken to initiate an I/O request and the actual time taken to perform the I/O. Therefore,

$$T_{I/O_{Select}} = \frac{I_{I/O}}{CPUSpeed} + AvgDiskReadTime$$

where $I_{I/O}$ is the number of instructions needed to initiate an I/O and CPUSpeed is the speed of the CPU in instructions per second (MIPS * 10^6). $T_{CPU_{Select}}$ includes the time taken by the selects to examine each tuple on the data page, apply the selection predicate, and send the selected tuples to the next operator. Therefore,

$$T_{CPU_{Select}} = \frac{(I_{Read} + I_{Pred}) * TupsPerPage}{CPUSpeed} + \frac{I_{Send} * SelectionSelectivity}{CPUSpeed}$$

where I_{Read} is the number of instructions for reading a tuple in memory, I_{Pred} is the number of instructions for applying a predicate, I_{Send} is the time taken to send a page (including the cost of copying tuples to the network buffer), *TupsPerPage* is the number of tuples in each page, and *SelectionSelectivity* is the fraction of tuples that satisfy the selection predicate.

Therefore the total rate at which pages are processed by the select operators is

$$ProcRate_{Select} = \frac{N_{Select}}{T_{select}}$$

where N_{Select} is the degree of parallelism of the select operator. Similarly, the total rate (in pages/second) at which tuples are produced by the select operators is

$$Rate_{Select} = ProcRate_{Select} * SelectionSelectivity = \frac{N_{Select} * SelectionSelectivity}{T_{select}}$$

Finally, since the rate at which tuples are produced may be different in the build and probe phases due to a different degrees of select parallelism in the build and probe phase, the above calculation is carried out for each phase separately. These rates are denoted as $Rate_{Select_{Build}}$ and $Rate_{Select_{Probe}}$, respectively.

3.4.2 Processing Rate of Hash-Join Operators

The rate at which tuples are processed by a hash-join operator depends on the amount of memory allocated to it. Here, we present formulas only for maximum and minimum join memory allocation. The formulas for intermediate memory allocations can be developed similarly.

Maximum Memory Allocation: In the case of maximum memory allocation, join operators do not perform any I/Os. During the build phase, the join operators read each incoming tuple and insert them into a hash table. Therefore, the time taken to process a data packet in the build phase is

$$T_{Build} = \frac{I_{Rcv} + (I_{Read} + I_{Hash}) * TupsPerPkt}{CPUSpeed}$$

where I_{Rcv} is the number of instructions needed to receive a data packet, I_{Hash} is the number of instructions needed to hash a tuple (this assumes that the tuple is already in memory and no copying is required), and *TupsPerPkt* is the number of tuples in each network packet. In the probe phase, the join reads incoming tuples and probes the hash table for matches. If a match is found, the result tuples are composed and sent to the parent operator. Therefore,

$$IPerTuple = I_{Read} + I_{Probe} + I_{Compose} * JoinSel$$

$$T_{Probe} = \frac{I_{Rcv} + IPerTuple * TupsPerPkt}{CPUSpeed} + \frac{I_{send} * JoinSel}{CPUSpeed}$$

where I_{Probe} is the number of instructions needed to probe the hash table, $I_{Compose}$ is the number of instructions needed to produce a result tuple, and *JoinSel* is the join selectivity.

Minimum Memory Allocation: If the joins are allocated their minimum memory allocation, the incoming tuples are divided into disk-resident partitions. Since select operators send tuples to join operators only during the partitioning phase, the rate calculation matches processing rates only for the partitioning phase. In both the build and probe phases, the incoming tuples are read, hashed and then written to the corresponding disk partition[2]. Therefore,

$$T_{Build/Probe} = AvgDiskWriteTime + \frac{I_{Rcv} + I_{I/O}}{CPUSpeed} + \frac{(I_{Read} + I_{Hash} + I_{Copy}) * TupsPerPkt}{CPUSpeed}$$

where I_{Copy} is the number of instructions needed to copy the tuples to the outgoing disk page.

Once the time taken for processing in each phase has been calculated, the number of join processes needed

[2]Recall that if joins are given their minimum memory allocation, the build and probe phases refer to the initial phases that distribute the inner and outer relations, respectively. The result tuples are produced in a third phase where each participating join processor processes its disk-resident partitions.

to absorb the incoming tuples in the build phase, $N_{Join_{Build}}$, is calculated by equating the rate at which tuples are processed by the joins to the rate at which tuples are produced by the selects.

$$Rate_{Join_{Build}} = \frac{N_{Join_{Build}}}{T_{Build}} = Rate_{Select_{Build}}$$

Therefore:

$$N_{Join_{Build}} = Rate_{Select_{Build}} * T_{Build}$$

Similarly:

$$N_{Join_{Probe}} = Rate_{Select_{Probe}} * T_{Probe}$$

Finally, the number of join sites should be such that select operators do not block in either the build or the probe phase, so:

$$N_{Join} = Min(N_{max}, Max(N_{Join_{Build}}, N_{Join_{Probe}}))$$

where N_{max} is the total number of nodes in the system.

3.4.3 Extension to Multiple Users

The only extension needed in the formulas for a multiple-user system is to modify the value of the CPU CPUSpeed parameter to incorporate the fact that other users in the system will also be using the CPU simultaneously. Note that the effect of multiple users at the disk is already incorporated in the value of the Average Disk Read/Write parameters. Therefore, the value of CPUSpeed is modified to EffectiveCPUSpeed using the formula for service time, S(x), for a task with service demand x on an M/G/1 server with round-robin scheduling [17], i.e.:

$$S(x) = \frac{x}{1 - Utilization}$$

where x is the service demand of the arriving job. Therefore, the time taken for each CPU processing task should be modified using the following equation.

$$T_{CPU} = \frac{CPUInstructions}{EffectiveCPUSpeed} = \frac{CPUInstructions}{CPUSpeed * (1 - Utilization)}$$

3.4.4 Effect of Message Buffer Size

The previous formulas assumed that there is no message buffering and therefore that the processing rates need to match exactly. However, in practice, the operators buffer only a limited number of message packets. In this case, the join processing rate may be slower than the select processing rate as long as there is no message buffer overflow. Let M be the size of the message buffer, T the time taken by the select to process all of the tuples, and N_{Join} and N_{Select} are the the degrees of join and select parallelism, respectively. The total number of message packets accumulated per second at the join operators is the difference in the rate at which tuples are sent by the select operators and the rate at which they are consumed by the join operators. Therefore, the total number of data packets accumulated over the period of the query is given by:

$$\#AccumulatedMsgs = (N_{Select} * Rate_{Select} - N_{Join} * Rate_{Join}) * T$$

If this number is equal to the total message buffer size of the join operators (i.e. $M * N_{Joins}$), there will be no message overflow. Therefore,

$$\#AccumulatedMsgs = M * N_{Joins} \quad (1)$$

The total time taken to process the input is, in turn, estimated as

$$T = \frac{T_{Select} * InputSize}{N_{Select}}$$

where InputSize is the number of pages accessed from the input relation. Substituting the value of T and $\#AccumulateMsgs$ in Equation 1 and simplifying, the degree of join parallelism is calculated as:

$$N_{Join} = \frac{N_{Select} * Rate_{Select} * T_{Select} * InputSize}{M * N_{Select} + Rate_{Join} * T_{Select} * InputSize}$$

where T_{Select} is the time taken by a select operator to process one data page.

4 Processor Assignment Algorithms

Once the degree of parallelism for an operator has been determined, each instance of an operator must be assigned to a specific processor. Six algorithms for processor assignment are considered in this paper:

4.1 Random

The desired number of processors are chosen randomly in this algorithm. Although the Random algorithm is simple to implement, it can lead to load imbalance (since it does not use any information about the present state of the system).

4.2 Round-Robin

The Round-Robin algorithm chooses processors in a round-robin fashion (i.e. if the first operator is executed on nodes 1–10, the next operator is executed on nodes 11 onwards). This algorithm distributes the processing load better than Random, but it can also lead to load imbalances because it ignores the actual distribution of the load in the system.

4.3 Avail-Memory

The third algorithm was proposed in [24] and assumes that the processing load of an operator is proportional to its memory requirement and chooses the processors with the most free memory. This assumption is applicable to memory-intensive operators like joins and sorts. Since memory allocation is performed at the scheduling nodes, memory utilization figures are already available to the query schedulers. Therefore, this algorithm does not entail any extra communication between the scheduling and processing nodes[3].

4.4 CPU-Util

The CPU-Util algorithm was first proposed in [23] and assigns the least-utilized processors to an operator. The performance of this algorithm depends on the frequency with which CPU utilization statistics can be updated at the scheduling nodes. Also, in order to prevent two successive operators from being scheduled on the same set of nodes, once a set of processors have been chosen, their CPU utilizations are increased "artificially". This artificial increase in CPU utilization prevents successive operators from being scheduled on the same set of processors, and it is cancelled the next time the statistics are updated. The amount by which the utilization should be increased is difficult to estimate, however. If the amount is too low, it may not prevent two successive operators from being executed on the same set of nodes. Conversely, if the amount is too high, it can lead to a large difference between the actual utilization of the node and the utilization as seen by the query schedulers, leading the CPU-Util algorithm to schedule queries on nodes that are already more heavily utilized.

In order to select these parameters, we performed a detailed sensitivity analysis of CPU-Util [Meht94]. As a result of the analysis, our simulation model updates utilization statistics at the scheduling nodes every 5 seconds. Also, once an operator is scheduled on a node, its utilization is increased "artificially" by 5%. Note that this algorithm is not useful for operators like store, that do not perform much CPU processing.

4.5 Disk-Util

The Disk-Util algorithm chooses the processors on the nodes with the least disk-utilization. Similar to the CPU-Util algorithm, disk utilization statistics are reported to the scheduling nodes every 5 seconds, and disk-utilization is artificially increased by 5% in-between periods of statistics collection. This algorithm is not useful for operators that do not perform a significant amount of disk I/O. An example is a hash-join operator with maximum memory allocation. Such a join operator performs only CPU processing since the input relations are read by separate select operators.

4.6 Input

The Input strategy can only be used with the MinDp and the MaxDp algorithms. Recall that the degree of parallelism selected by the MinDp and MaxDp algorithms is equal to the degree of parallelism of one of the input operators; the input operator with the maximum parallelism for MaxDp and minimum parallelism for MinDp. The Input strategy executes an operator on the same set of processors as the selected input operator. For example, if the MinDp policy selects the inner relation data stream for a hash-join operator, the Input strategy will assign the join operator to the set of nodes where the inner relation is being accessed.

5 Processor Allocation Strategies

The algorithms for determining the degree of operator parallelism can be combined with the algorithms for processor assignment to obtain a wide variety of processor allocation algorithms. Most of the combined algorithms perform processor allocation in two phases. The degree of parallelism is determined in the first phase. The degree of operator parallelism and the total memory allocation to the operator are then used to determine the memory needed *per processor*. In the second phase, a list is made of candidate processors that have enough memory available in their buffer pools. Finally, the processor assignment algorithm is used to select a subset of the processors from the candidate list. If the number of processors in the candidate list is less than the degree of parallelism of the operator, the query blocks and waits for memory to become available. The only exceptions to this process occur with the Maximum policy, which executes each operator on all processors, and the Input processor assignment policy, which executes the operator on the nodes where the input data stream is produced.

6 Simulation Model

The performance studies presented in this paper are based on a detailed simulation model of a shared-nothing parallel database system. The simulator is written in the CSIM/C++ process-oriented simulation language [Schw90] and models the database system as a closed queueing system. The following sections describe the configuration, database and workload models of the simulator in more detail.

[3] If there are multiple scheduling nodes, some communication is needed among the scheduling nodes to maintain an accurate estimate of memory consumption at the processing nodes. However, these costs are ignored in this paper since inter-scheduler communication for memory management is needed in all processor allocation algorithms.

6.1 Configuration Model

The terminals model the external workload source for the system. Each terminal sequentially submits a stream of transactions. Each terminal has an exponentially distributed "thinktime" to create variations in arrival rates. All experiments in this paper use a configuration consisting of 128 nodes. The nodes are modeled as a CPU, a buffer pool of 16 Mbytes[4] with 8 Kbyte data pages, and one or more disk drives. The CPU uses a round-robin scheduling policy with a 5 millisecond timeslice. The buffer pool models a set of main memory page frames whose replacement is controlled via the LRU policy extended with "love/hate" hints. These hints are provided by the various relational operators when fixed pages are unpinned. For example, "love" hints are given by the index scan operator to keep index pages in memory; "hate" hints are used by the sequential scan operator to prevent buffer pool flooding. In addition, a memory reservation system under the control of the scheduler task allows buffer pool memory to be reserved for a particular operator. This memory reservation mechanism is used by hash join operators to ensure that enough memory is available to prevent their hash table frames from being stolen by other operators.

The simulated disks model a Fujitsu Model M2266 (1 Gbyte, 5.25") disk drive. This disk provides a cache that is divided into 32 Kbyte cache contexts for use in prefetching pages for sequential scans. In the disk model, which slightly simplifies the actual operation of the disk, the cache is managed as follows: each I/O request, along with the required page number, specifies whether or not prefetching is desired. If prefetching is requested, four pages are read from the disk into a cache context as part of transferring the page originally requested from the disk into memory. Subsequent requests to one of the prefetched blocks can then be satisfied without incurring an I/O operation. A simple round-robin replacement policy is used to allocate cache contexts if the number of concurrent prefetch requests exceeds the number of available cache contexts. The disk queue is managed using an elevator algorithm.

The interconnection is modeled as an infinite bandwidth network so there is no network contention for messages. This is based on previous experience with the GAMMA prototype [7] which showed that network contention is minimal in typical shared-nothing PDBs. Messages do, however, incur an end-to-end transmission delay of 500 microseconds. All messages are "point-to-point" and no broadcast mechanism is used for communication. Table 1 summarizes the configuration parameters and Table 2 shows the CPU instruction costs used in the simulator for various database operations.

[4] The simulated buffer pool size is smaller than buffer pools in typical configurations. Unfortunately, simulating a larger buffer pool size would require enormous amounts of resources. Some of our simulations took up to 36 MBytes and ran for 24 hours on an IBM RS/6000 even with 16 Mbytes of memory per node.

Parameter	Value
Number of Nodes	128
Memory Per Node	16 Mbytes
CPU Speed	10 MIPS
Number of Disks per Node	1
Page Size	8 Kbytes
Disk Seek Factor	0.617
Disk Rotation Time	16.667 msec
Disk Settle Time	2.0 msec
Disk Transfer Rate	3.09 Mbytes/sec
Disk Cache Context Size	4 pages
Disk Cache Size	8 contexts
Message Wire Delay	500 μsec

Table 1: Simulator Parameters: Configuration

7 Experimental Parameters

7.1 Configuration

Although we have experimented with both a disk-intensive and CPU-intensive configuration, for the sake of brevity, results are presented in this paper only for a CPU-intensive configuration. The results of the disk-intensive configuration are briefly summarized in each performance section and the interested reader is referred to [20] for detailed experimental results. The CPU-intensive configuration consists of a 10 MIPS CPU and four disks per processor. The CPU speed was chosen to be artificially low so that the processors could be saturated with only 4 disks per node, thus reducing the running time of the simulations time[5].

A message buffer of 256 Kbytes is provided for each operator. This implies that at most 32 8Kbyte pages can be buffered by each operator. Operators stop sending messages when they detect that the receiver's message buffer is full.

[5] For a faster processor, we would need to simulate many more disks per processor. For instance, it takes upto 16 disks with high I/O prefetching to saturate one Alpha AXP processor [5]

Operation	Instr.
Initiate Select Operator	20000
Terminate Select Operator	5000
Initiate Join Operator	40000
Terminate Join Operator	10000
Apply a Predicate	100
Read Tuple from Buffer	300
Probe Hash Table	200
Insert Tuple in Hash Table	100
Start an I/O	10000
Copy a Byte in Memory	1
Send(Receive) an 8K Message	10000

Table 2: Simulation Parameters: CPU Costs

7.2 Database

A simplified database is used in all of the experiments. Each database relation contains five million tuples and is fully declustered. Although the relations are fully declustered, we model range queries to explore the effect of reading data on only a subset of nodes. The tuple size is fixed at 200 bytes and a clustered index is modeled on each relation.

7.3 Workload

The workload contains only binary hybrid hash-join [6] queries. The hybrid hash-join method was chosen since it has been shown to be superior to other join methods. Binary join queries were chosen so that issues, like pipelining and query scheduling, that arise while processing complex queries could be ignored. This is a reasonable simplification as most commercial database systems execute queries comprised of multiple joins as a series of binary joins, and do not pipeline tuples between adjacent joins in the query tree. Each binary join query is composed of two select operators (one for the inner and one for the outer relation) plus a join operator and a store operator. The select operators execute wherever the input relations are declustered. Therefore, processor allocation needs to be determined only for the join and store operators. In order to simplify this performance study, a simplistic processor allocation policy is used for store operators – the store operator of each query executes on the same set of nodes as the join operator of the query. Therefore, the degree of parallelism and specific processor assignments need to be determined only for the join operator in a query. Moreover, the join selectivity has been fixed at 1% to make the size of the join output small to reduce the impact of store operators on the performance results. Based on the results of earlier memory allocation studies [21] [28] [1] [3], joins are given either their maximum or their minimum memory allocation. Three kinds of workloads are considered: Small, Medium and Large. Table 3 summarizes the important parameters of these three workloads.

Workload	Access Method	indexSelectivity
Small	Clustered Index Scan	1%
Medium	Clustered Index Scan	25%
Large	File Scan	N/A

Table 3: Workload Parameters

We also assume the presence of some mechanism that can be used to direct the select operators to only a subset of the nodes.[6]. Therefore, the degree of select operator parallelism is chosen randomly from 1 to 128.

[6]Even though all the relations are fully declustered, select operators can be directed to a subset of nodes in several cases (e.g. when *range declustering* [11] is used to map tuples to relation partitions).

The performance of all of the algorithms is examined under various system loads by increasing the number of query terminals from 10 to 40.

8 Determining Degree of Parallelism

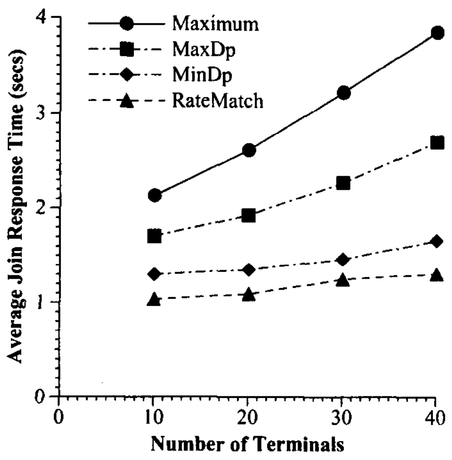
This section presents a performance comparison of algorithms that determine the degree of parallelism. The CPU-Util algorithm is used in all the experiments to perform processor assignment; the reason for using this algorithm here will become evident in Section 9.

8.1 Maximum Memory Allocation

The first experiment compares the performance of the algorithms on the Small workload (clustered index scans with 1% indexSelectivity) when each query is given its maximum memory allocation. We assume that range declustering is used and the degree of parallelism of the select operators varies uniformly between 1 and 128. Figure 1 shows the average query response time and the degree of join parallelism as the load increases from 10 to 40 terminals.

Since the queries in this workload are small, startup and termination costs form a large fraction of the query response time. Therefore, the relative order of the algorithms is determined by the startup and termination costs, which, in turn, are determined by the degree of join parallelism. The Maximum algorithm, which selects the highest degree of join parallelism (128), has the highest startup and termination costs and, consequently, the highest average query response time. The MaxDp and MinDp algorithms choose smaller degrees of parallelism (85 and 38, respectively) and therefore achieve lower average query response times (as compared to the Maximum algorithm). The lowest query response time is achieved by the RateMatch algorithm. This algorithm realizes that the sizes of the inner and outer relations of the join queries are small, and that the join and select processing rates can be matched with a low degree of join parallelism. Moreover, unlike the other algorithms, the RateMatch algorithm dynamically adapts to the query workload: it selects a higher degree of join parallelism as the system load increases because the CPU-utilization increases. The degree of join parallelism increases from 25 to 32 as the load increases from 10 to 40 terminals.

The next experiment explores the relative performance of the algorithms on the Medium workload (clustered index scans with 25% indexSelectivity). Figure 2 shows the average query response time and the degree of join parallelism chosen by the algorithms. Note that, except for the RateMatch algorithm, the degrees of join parallelism chosen by all the other algorithms are the same as in the previous experiment. This is, because, their choice of the degree of paral-



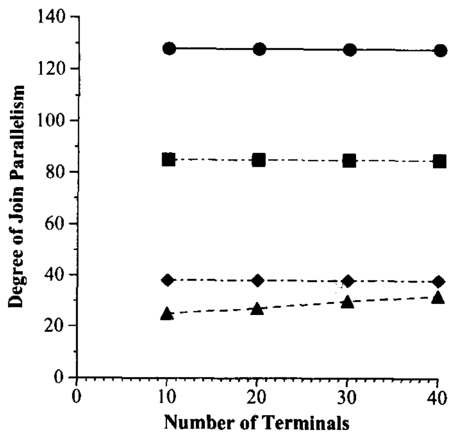
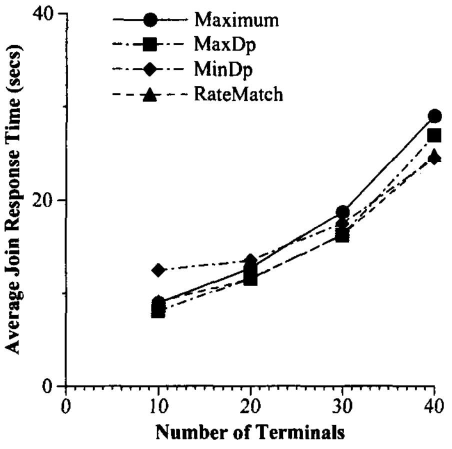


Figure 1: Performance of Algorithms for Determining the Degree of Parallelism – Small Workload Maximum Memory Allocation



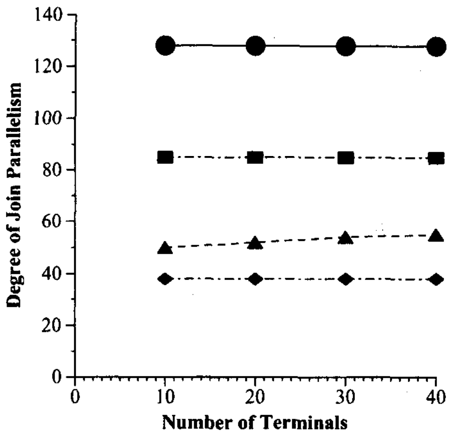


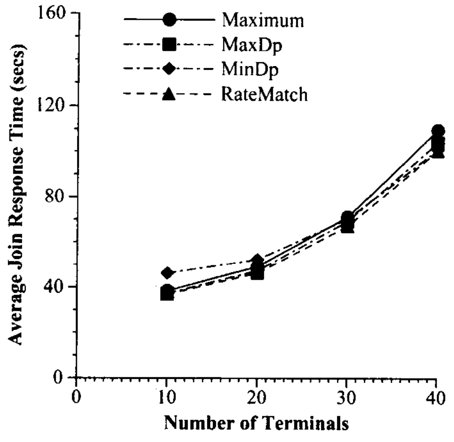
Figure 2: Performance of Algorithms for Determining the Degree of Parallelism – Medium Workload Maximum Memory Allocation

lelism depends only on the data placement and configuration size, both of which remain static for all workloads.

The performance results for the Medium workload are quite different from those of the Small workload. The MinDp algorithm has the *highest* average query response time for the Medium workload for less than 30 terminals, followed by the Maximum, then the MaxDp, and finally the RateMatch algorithm. The important thing to note is that MinDp has the worst performance, even though Figure 2 shows that it chooses the *smallest* degree of join parallelism (38).

The inferior performance of MinDp is due to the fact it selects a degree of join parallelism that is so low that the join operators cannot process tuples at the rate at which they are produced by the selects, so the message buffers of the join operators overflow. This causes the select operators to block leading to lower CPU and disk utilizations and higher query response times. Note that message buffer overflow was not observed in the previous workload since the queries were much smaller (1% indexSelectivity) and the message buffer size (256 KB) was large enough to prevent overflow. The RateMatch algorithm dynamically selects a higher degree of join parallelism for this workload to prevent the message buffers of the join operators from overflowing. The MaxDp and Maximum algorithms avoid message buffer overflow but they incur higher startup and termination costs than the RateMatch algorithm since they select degrees of join parallelism that are "too" high. However, startup and termination costs are a smaller fraction of the total response time of the queries in this workload, so the average query response times achieved by the Maximum and MaxDp algorithms are only 17% and 9% higher, respectively, than those of the RateMatch algorithm.

However, as the number of terminals is increased, the relative performance of the MinDp algorithm improves as the load increases (more than 20 terminals). This is, because, even though some of the op-



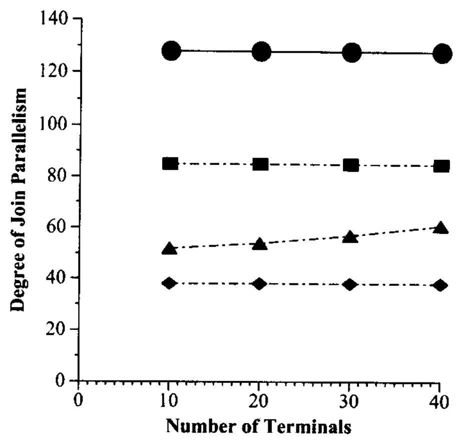


Figure 3: Performance of Algorithms for Determining the Degree of Parallelism – Large Workload Maximum Memory Allocation

erators block in the MinDp algorithm due to message buffer overflow, there is enough concurrent activity in the system to achieve high processor utilization, and there is no increase in the average response time. The Maximum and MaxDp algorithms, on the other hand, perform relatively worse because of their higher startup and termination costs. The RateMatch algorithm chooses a much smaller degree of join parallelism (compared to Maximum and MaxDp), and therefore performs well. At the highest load of 40 terminals, RateMatch results in an average response time that is only 2% higher than the average response time achieved by MinDp.

The next experiment examines the performance of the algorithms on the Large query workload (file scans with 100% selectionSelectivity). Figure 3 shows the average query response time and the degree of join parallelism for each algorithm. The results show that RateMatch still has the best performance, but the performance of the other algorithms is much closer; Maximum and MaxDp provide response times that are only 11% and 9% higher, respectively. The reason is the same as before – file scans increase the execution time of the queries, so the effect of extra startup and termination costs in the MaxDp and Maximum algorithms become less and less significant. The MinDp algorithm behaves similar to the last experiment. It selects a low degree of parallelism, causing the select operators to block; thus, leading to high average query response times at low query loads. As the load increases, the effect of blocking diminishes and MinDp is able to achieve lower average query response times.

It is interesting to note that the degree of join parallelism chosen by the RateMatch algorithm for the Medium and for the Large workloads is nearly identical (even though the Large queries process nearly four times the data processed by the Medium queries). The reason is that the *rate* at which tuples are sent by the select operators is the same for both workloads. The rate depends on the degree of parallelism of the select operators. Since the degree of parallelism of select operators is identical for both workloads, the rate at which tuples are sent to the join operators in the two workloads is also identical. Therefore, the same degree of join parallelism can be used for both the workloads. This implies that any algorithm which chooses the degree of join parallelism based on the *size* of the input relations will be non-optimal. The degree of join parallelism chosen by such an algorithm for the Large workload would be four times that of the Medium workload, while these experiments have shown that the degree of join parallelism should remain the same for both workloads.

The last three experiments have shown that the MinDp algorithm performs well for small-query workloads but can lead to a underutilized system and higher query response times for larger queries (since it can underestimate the proper degree of join parallelism causing select operators to block). The Maximum and MaxDp algorithms perform poorly for small query workloads due to high startup and termination costs but can provide reasonable performance as query sizes increase. Finally, the RateMatch algorithm consistently shows good performance. The RateMatch algorithm performs well for the small query workload because it reduces startup and termination costs. At the same time, it can dynamically increase the degree of parallelism for larger queries to prevent operators from blocking.

8.1.1 Minimum Memory Allocation

In each of the previous experiments, queries were given their maximum memory allocation. The next experiment compares the performance of the algorithms when queries are given their minimum memory allocation. Figure 4 shows the average query response

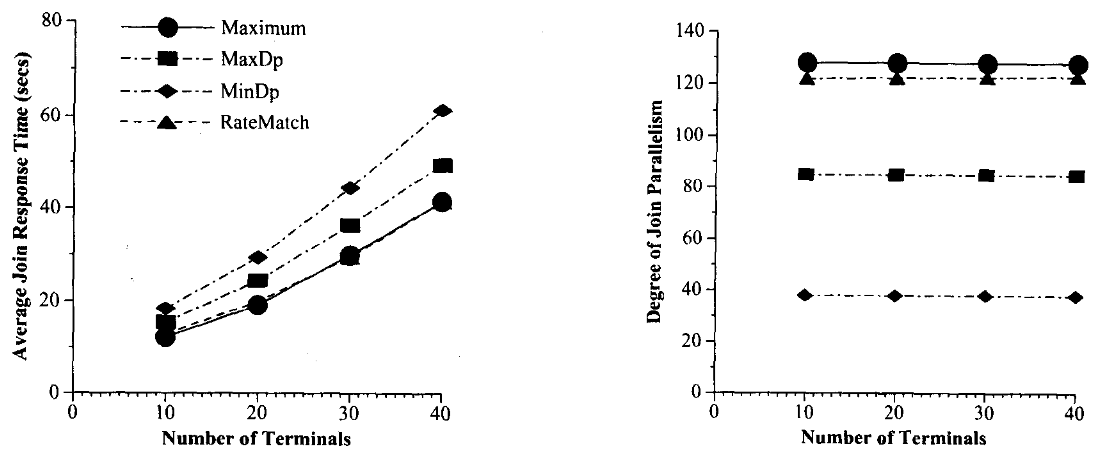


Figure 4: Performance of Algorithms for Determining the Degree of Parallelism – Medium Workload Minimum Memory Allocation

time and the degree of join parallelism for the medium workload. Results are not presented for the Small workload since there is not much of a difference between maximum and minimum memory allocation for queries in the Small workload. Additionally, Large workload results were qualitatively very similar to the Medium workload results and have therefore been omitted.

Figure 4 shows that if queries are allocated their minimum memory allocation, the higher the degree of join parallelism, the lower the average query response time. This occurs for two reasons. First, minimum memory allocation implies that input relations must be partitioned by the join operators into disk-resident buckets. Since writing out tuples to disk is slow, the processing rate of join operators decreases. Therefore, too little join parallelism can cause the select operators to block. Second, once partitioning of the input relations is complete, a higher degree of join parallelism implies faster processing for each disk-resident bucket. Consequently, the Maximum and RateMatch algorithms, which both select high degrees of join parallelism (128 and 122, respectively), provide better performance than the MinDp and MaxDp algorithms, which select lower degrees of join parallelism (85 and 38, respectively).

8.2 Result Summary

The results of the previous experiments have shown that when queries are allocated their maximum memory allocation, the MinDp algorithm performs well for small queries since the cost of startup and termination constitutes a large fraction of their response time. However, the MinDp algorithm's performance can deteriorate as the size of the input relations increases because it can underestimate the degree of operator parallelism, thus causing other operators to block. The Maximum and MaxDp algorithms perform poorly for small query workloads due to high startup and termination costs, but they provide reasonable performance for larger query sizes. On the other hand, the RateMatch algorithm can dynamically adapt the degree of parallelism to provide good performance for both small and large query workloads.

If minimum memory allocation is used for queries, a higher degree of join parallelism improves response times. Therefore, the Maximum and RateMatch algorithms perform well, but the MinDp and MaxDp algorithms lead to higher response times.

The relative performance of the algorithms is also the same in a disk-intensive configuration [20] except that all the algorithms have nearly identical performance when queries are given their maximum memory allocation. Response times are dominated by I/O processing time in a disk-intensive configuration; since all the algorithms perform the same I/O processing if queries are allocated their maximum memory allocation, their performance is also identical.

9 Determining Processor Assignment

So far we have compared the performance of the algorithms for selecting the degree of join parallelism. This section presents a performance evaluation of the six processor assignment algorithms discussed in Section 4.

9.1 Maximum Memory Allocation

The first experiment in this section compares the performance of the algorithms on the Small workload. Maximum memory allocation is used for the queries and the degree of select parallelism varies varies uniformly between 1 and 128. Figure 5 shows the performance of the various processor assignment algorithms when the RateMatch algorithm is used to determine the degree of join parallelism[7]. As explained previously, the Input algorithm cannot be used with the Rate algorithm, so it is not shown in Figure 5. The Disk-Util algorithm is also absent since it is used only with minimum memory allocation.

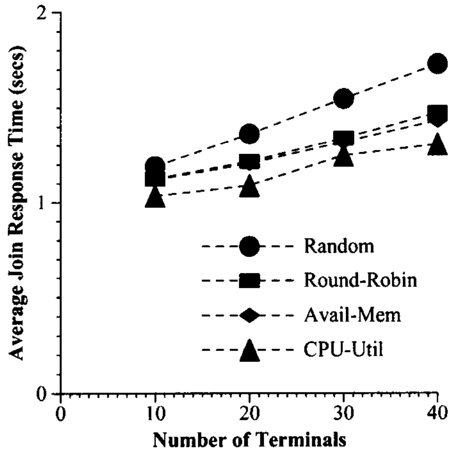


Figure 5: Processor Assignment Algorithms
Small Workload, Maximum Memory Allocation

Figure 5 shows that the Random algorithm leads to the highest response times since it sometimes assigns even heavily loaded processors to a join. The Round-Robin and Avail-Memory algorithms distribute the join workload more uniformly than the Random algorithm and therefore achieve lower response times. However, both these algorithms ignore the CPU load from the select operators, and thus do not perform as well as the CPU-Util algorithm. The CPU-Util achieves the lowest query response times, but it is only about 10% better than the Round-Robin algorithm. This is because the CPU-Util algorithm uses utilization statistics that are updated every 5 seconds. Since queries in this workload are very small (1% selectivity on the input relations), multiple queries often arrive in the system within the 5 second intervals when the CPU-utilization statistics are out-of-date. These queries therefore get executed on nodes that do not necessarily have the lowest CPU-utilization. As a result, the performance of the CPU-Util algorithm is only slightly better than the simpler Round-Robin algorithm.

The relative performance of the processor assignment algorithms is also similar with the Medium workload. Figure 6 shows the performance of the different algorithms when the RateMatch algorithm is used to select the degree of join parallelism. CPU-Util provides the best performance in all the cases, followed by the Avail-Mem, Round-Robin, and the Random algorithms. Results for the Large workload were qualitatively similar and have been omitted.

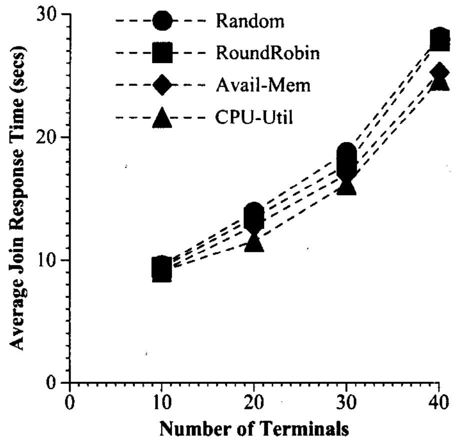


Figure 6: Processor Assignment Algorithms
Medium Workload, Maximum Memory Allocation

The results of these experiments show that the CPU-Util algorithm for processor assignment achieves the lowest response times when queries are given their maximum memory allocation. However, as query sizes increase, simpler algorithms like Round-Robin and Avail-Mem can also perform quite well.

9.2 Minimum Memory Allocation

The next experiment with the CPU-intensive configuration explores the performance of the processor allocation algorithms when joins are given their minimum memory allocation. As in Section 8.1.1, results are reported only for the Medium workload (since there is not much difference between the maximum and minimum memory allocations for the Small workload and the results for the Large workload are similar to the results of Medium). Figure 7 shows the average query response times for the different processor assignment algorithms under various system loads. Since join operators perform I/Os with minimum memory allocation, the performance of the Disk-Util algorithm is also included.

Figure 7 shows that all the processor assignment algorithms have virtually identical performance in this case. This is mainly because the number of processors chosen by RateMatch is high for this workload. The average degree of join parallelism is 122. Therefore, the set of processors chosen by the CPU-Util algorithm, for example, is not very different from the set of processors chosen by the Random algorithm. As a result, all of the algorithms have basically the same performance for this workload. This experiment shows that as the degree of operator parallelism increases, the differences in the performance of the various processor allocation algorithms virtually disappear.

[7][20] also contains experimental results when other algorithms like MaxDp are used determine the degree of join parallelism and the results are qualitatively very similar.

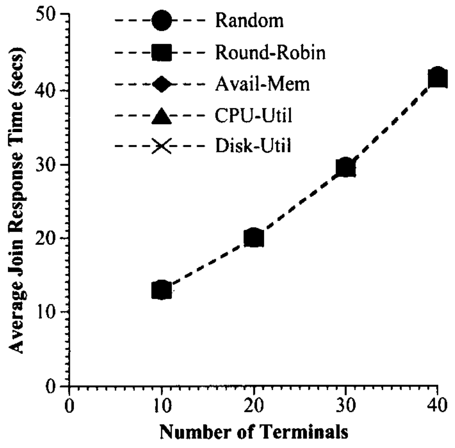


Figure 7: Processor Assignment Algorithms
Medium Workload, Minimum Memory Allocation

9.3 Result Summary

This section has presented the performance of several processor assignment algorithms. The results show that the choice of the processor assignment algorithm has a significant performance impact only if the workload is CPU-intensive and queries are given their maximum memory allocation. In this case, the CPU-Util algorithm, which selects the least utilized processors, achieves the lowest response time.[8] In all other cases[9], the choice of a processor assignment algorithm has a small performance impact and therefore a simple algorithm like Round-Robin is sufficient to obtain reasonable performance. Finally, the experiments show that the choice of the processor assignment algorithm is not as important as the choice of the algorithm used to decide the degree of join parallelism; the differences between the performance of the processor assignment algorithms are smaller than the differences in the performance of the algorithms for selecting join parallelism.

10 Related Work

Intra-operator parallelism and processor assignment has been an active area of database research. The topic has been studied extensively in the context of load balancing in shared-everything systems [14] [19]. Several researchers have focused on processor assignment for queries with multiple join operators [10] [22] [9]. Processor assignment has also been studied examined in the context of distributed database systems [2] [18].

A formula for determining the optimal degree of parallelism of database operators was presented in [27]. The formula assumes that if S is the startup cost of an operation, and P is the per-tuple processing cost, then the optimal degree of parallelism, $n_{opt} = \sqrt{\frac{PN}{S}}$. Note that this formula is based only on the size of the operand and disregards the rate of flow of tuples between operators. The results presented earlier in the paper (Section 8) show that this can lead to excessively high degrees of parallelism. Rahm and Marek [23] study algorithms to determine the degree of parallelism for the join operator and also study processor assignment algorithms. The authors consider only small queris and decrease the degree of join parallelism based on the CPU-utilization in a multi-user environment. Our results, however, show that reducing the degree of parallelism even in CPU-intensive configurations does not affect performance significantly, especially for large query sizes. Moreover, the algorithm presented in [23] uses the optimal parallelism in single-user mode as input. This parameter can be hard to estimate especially since it is a complex function of the configuration, memory, and the memory allocation policy. Algorithms for processor and memory allocation were also studied in [24]. [Murp91] use the concept of matching processing rates of operators in a query plan to determine buffer allocation.

11 Conclusions

This paper has investigated the problem of managing intra-operator parallelism in a multi-query environment for a parallel database system. Four algorithms for deciding the degree of operator parallelism and six algorithms for selecting the assignment of operator instances to processors were considered. A detailed performance evaluation of the algorithms showed that using the RateMatch algorithm for deciding the degree of parallelism and the CPU-Util algorithm for selecting processors achieves the best performance irrespective of the workload and hardware configuration. The RateMatch algorithm calculates the degree of parallelism based on the rate at which tuples are processed by various operators, while the CPU-Util algorithm selects the processor with the least CPU-Utilization. Both algorithms use information about the current system state and can therefore dynamically adapt to different workloads. However, experiments also show that if the workload consists of large queries, or if the configuration is disk-intensive, simpler allocation algorithms like MaxDp can perform equally well. This implies that processor allocation can be significantly simplified in several cases.

In this paper, the RateMatch algorithm was used only to determine the degree of join parallelism for binary join queries in a shared-nothing system. However, we feel that the paradigm of matching the rate of tu-

[8]These results explain the use of the CPU-Util algorithm in all of the experiments comparing algorithms for selecting join performance (Section 8).

[9]The experiments with the disk-intensive configuration [20] also show that all the algorithms perform similarly.

ple flow between operators can be used in other cases also. For example, it can be used in a complex query to match the rate of flow between the operators in a parallel-query pipeline. Similarly, the algorithm can be used to decide the degree of parallelism for other operators like sorts and aggregates. We plan to explore these issues further in the future. Another direction of future research is the application of the RateMatch algorithm to shared-memory and shared-disk systems.

The results presented in this study have also shown the importance of decoupling processor allocation from data placement. The MinDp algorithm, for instance, can underestimate the degree of operator parallelism and cause high query response times. Similarly, the MaxDp algorithm can overestimate operator parallelism and lead to higher startup and termination costs. The decoupling of processor allocation and data placement can have a significant impact on several other areas of research in shared-nothing parallel database systems as well. For example, all of the studies on declustering policies [11] [15] also make the implicit assumption that operations like joins are executed on the nodes where data is accessed. A re-examination of the algorithms proposed in these studies will be required if this assumption is removed.

References

[1] K. Brown, M. Mehta, M. Carey, and M. Livny. Towards automated performance tuning for complex workloads. In *Proc. VLDB Conf.*, Santiago, Chile, September 1994.

[2] M. Carey, M. Livny, and H. Lu. Dynamic task allocation in a distributed database system. In *Proc. Intl. Conf. On Distr. Computing Systems*, May 1985.

[3] D. Davison and G. Graefe. Dynamic resource brokering for multi-user query execution. In *Proc. SIGMOD Conf.*, San Jose, Ca, May 1995.

[4] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6), March 1992.

[5] C. Nyberg et. al. Alphasort: A risc machine sort. In *Proc. SIGMOD Conf.*, Minneapolis, MN, May 1994.

[6] D. DeWitt et. al. Implementation techniques for main memory databases. In *Proc. SIGMOD Conf.*, Boston, MA, June 1984.

[7] D. DeWitt et. al. The gamma database machine project. *IEEE Trans. on Knowledge and Data Engg.*, 2(1), March 1990.

[8] H. Boral et. al. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowledge and Data Engg.*, 2(1), March 1990.

[9] Ming-Ling Lo et. al. On optimal processor allocation to support pipelined hash joins. In *SIGMOD*, pages 69–78, Washington DC, June 1993.

[10] Ming-Syan Chen et. al. Using segmented right-deep trees for the execution of pipelined hash joins. In *VLDB*, Vancouver, Canada, August 1992.

[11] S. Ghandeharizadeh. *Physical Database Design in Multiprocessor Systems.* PhD thesis, University of Wisconsin-Madison, 1990.

[12] G. Graefe. Volcano: An extensible and parallel dataflow query processing system. Technical report, Oregon Graduate Center, June 1989.

[13] Tandem Performance Group. A benchmark of non-stop sql on the debit credit transaction. In *Proc. SIGMOD Conf.*, Chicago, IL, November 1988.

[14] W. Hong. Exploiting inter-operation parallelism in xprs. In *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992.

[15] K. A. Hua and C. Lee. An adaptive data placement scheme for parallel database computer systems. In *Proc. VLDB Conf.*, Brisbane, Australia, 1990.

[16] Int'l Business Machines. *Scalable POWERparallel Systems*, GA23-2475-02 edition, February 1995.

[17] L. Kleinrock. *Queueing Systems - Vol. 2: Computer Applications.* John Wiley and Sons, 1976.

[18] H. Lu and M. Carey. Some experimental results on distributed join algorithms in a local network. In *Proc. VLDB Conf.*, Stockholm, Sweden, August 1985.

[19] H. Lu and K. Tan. Dynamic and load-balanced task-oriented database query processing in paralle systems. In *Proc. EDBT Conf.*, 1992.

[20] M. Mehta. *Resource Allocation for Parallel Shared-Nothing Database Systems.* PhD thesis, University of Wisconsin-Madison, 1994. available at http://www.cs.wisc.edu/ mmehta/mmehta.html.

[21] M. Mehta and D. DeWitt. Dynamic memory allocation for multiple-query workloads. In *Proc. VLDB Conf.*, Dublin, Ireland, August 1993.

[22] P. S. Yu Ming-Syan Chen and K. L. Wu. Scheduling and processor allocation for parallel execution of multi-join queries. In *Proc. of the 8th Int. Conf. on Data Engineering*, Pheonix, AZ, February 1992.

[23] E. Rahm and R. Marek. Analysis of dynamic load balancing strategies for parallel shared nothing database systems. In *VLDB*, Dublin, Ireland, August 1993.

[24] E. Rahm and R. Marek. Dynamic multi-resource load balancing in parallel database systems. Technical Report 2 (1994), University of Leipzig, June 1994.

[25] M. Stonebraker. The case for shared nothing. *Data Engineering*, 9(1), March 1986.

[26] Teradata Corp. *DBC/1012 Data Base Computer System Manual*, document no. c10-0001-02, release 2.0 edition, November 1985.

[27] A. Wilschut, J. Flokstra, and P. Apers. Parallelism in a main-memory dbms: The performance of prisma/db. In *Proc. VLDB Conf.*, Vancouver, Canada, August 1992.

[28] P. Yu and D. Cornell. Buffer management based on return on consumption in a multi-query environment. *VLDB Journal*, 2(1), January 1993.