Architectural Considerations for Parallel Query Evaluation Algorithms

Ambuj Shatdal

Technical Report #1321

August 1996

ARCHITECTURAL CONSIDERATIONS FOR PARALLEL QUERY EVALUATION ALGORITHMS

By

Ambuj Shatdal

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

Abstract

Parallelism is key to high performance relational database systems. Since there are several parallel architectures suitable for database systems, a few interesting problems arise, mostly from an emphasis on the differences among the architectures. Specifically, in the literature, differences rather than similarities between the architectures are pointed out, and the specific details of a particular architecture, crucial to high performance, are generally ignored. In this thesis we have attempted to remedy this situation by emphasizing the similarities and a deeper understanding of two popular parallel architectures, shared nothing and shared memory, from a database perspective. We show that there is complementarity and similarity in the two architectures by showing that software shared-memory support can be used to improve performance on shared-nothing hardware and by showing that shared-nothing software can run on shared-memory hardware with performance comparable to that of "native" algorithms. We also show that by understanding the architectural details and tradeoffs, we can design algorithms that have superior performance. We illustrate this via examples of hash join algorithms on shared-memory hardware that exploit cache memories, hash aggregation algorithms on shared-nothing hardware that tradeoff communication for memory consumption, and hash aggregation algorithm on shared-memory hardware that tradeoff computation for reduced latch conflicts. All these algorithms show performance superior to the previously known algorithms.

Acknowledgements

I want to take this opportunity to express my gratitude to all those have helped me through my graduate studies.

I was fortunate to be able to work and study with Prof. Jeffrey F. Naughton, one of the most wonderful persons I have known in my life. This thesis would have been a very difficult task without his generous help and support. Profs. David DeWitt and Yannis Ioannidis took time to carefully read my thesis; their insights and constructive cricism helped me to improve it considerably. All my teachers here in the Computer Sciences Department—Profs. Charles Fischer, Mary Vernon, Bart Miller, David Wood, Mark Hill—helped me to create a foundation for graduate work. The teachers of numerous other courses that I took here at the University—Prof. David Lindberg, Prof. Malcolm Forster, Prof. Alfred Senn, Prof. Arthur Glenberg, Michael Argue, Lise Rempel, Debbie Berg, Denise Mjelde, Judy Burnell, Shelly, and Dave—helped me broaden my horizons (and get distracted). My friends Anthory D'Silva, Subbarao Palacharla, Biswadeep Nag, T. N. Vijaykumar, Satish Chandra, Francis Valiyaveetil, Trevor Schaid, Dhrubajyoti Borthakur, Chander Kant, Sumeer Goyal, Aimee Dobbs, Ersin Ozugurlu, Kevin Cherkauer and Kristy Chaudoir, made my stay in Madison a pleasant one. I am indebted to all of them.

I also wish to thank the National Science Foudation, for the reseach grants that supported my work, and the National Center for Supercomputing Applications, for letting me use their machines.

I dedicate this work to my family—my parents, Vishwanath and Sheela; my siblings, Rajeev, Pankaj, Neerja and Arvind; and my wife, Heather—for without their eternal love and support this work would not have been possible.

Contents

Abstract i

Acknowledgements ii

1 Introduction 1

1.1 Background and Motivation 1

1.2 A Review of Parallel Architectures 2

1.2.1 The Shared Nothing Architecture 3

1.2.2 The Shared Memory Architecture 4

1.3 Thesis Contributions and Organization 6

2 Hash Join using Shared Virtual Memory 7

2.1 Introduction 7

2.2 Brief Overview of SVM 9

2.2.1 Structure of SVM 9

2.2.2 Performance and Cost 9

2.3 Data Skew: Problem and Previous Solutions 10

2.4 Join Processing using SVM 12

2.4.1 Naive Approaches 13

2.4.2 A Dual-Paradigm Algorithm 13

2.4.3 A Sampling Variant 16

2.4.4 Is SVM Necessary? 18

2.5 Performance Evaluation 18

2.5.1 Simulation Methodology 19

2.5.2 Experiments with Varying Data Skew 22

2.5.3 Multi-Bucket Join 26

2.5.4 Speedup and Scaleup 26

2.6 Concluding Remarks 28

3 Hash Join on Shared Memory Hardware 30
3.1 Introduction . . . 31
3.2 The Experimental Setup . . . 33
3.3 Parallel Hash Join Algorithms . . . 34
3.3.1 The Shared Nothing Algorithm . . . 34
3.3.2 The Shared Memory Approach . . . 38
3.3.3 The Hybrid Approach . . . 40
3.3.4 Cache Conscious Hash Join . . . 42
3.4 Performance Summary . . . 44
3.5 Concluding Remarks . . . 46

4 Hash Aggregation on Shared Nothing Hardware 48
4.1 Introduction . . . 48
4.2 The Background . . . 50
4.2.1 Centralized Two Phase Algorithm . . . 52
4.2.2 Two Phase Aggregation . . . 54
4.2.3 Repartitioning Algorithm . . . 55
4.3 The Algorithms . . . 57
4.3.1 Sampling Based Approach . . . 57
4.3.2 Adaptive Two Phase Algorithm . . . 59
4.3.3 Adaptive Repartitioning Algorithm . . . 61
4.4 Analytical Results . . . 62
4.5 Implementation Results . . . 66
4.6 The Effect of Data Skew . . . 67
4.6.1 Input Data Skew . . . 67
4.6.2 Output Data Skew . . . 68
4.7 Concluding Remarks . . . 70

5 Hash Aggregation on Shared Memory Hardware 71
5.1 Introduction . . . 72
5.2 The Experimental Setup . . . 73
5.3 Classical Algorithms . . . 73

5.3.1 Simple Parallel Aggregation 74

5.3.2 Shared Nothing Two Phase Approach on SMP 75

5.4 The Proposed Algorithms 77

5.4.1 The Adaptive Two Step Algorithm 78

5.4.2 The Adaptive Simple Algorithm 79

5.4.3 Performance Evaluation 80

5.5 Shared Nothing Algorithms on SMP 81

5.6 Concluding Remarks 82

6 Conclusions **84**

Bibliography **87**

Chapter 1

Introduction

1.1 Background and Motivation

Parallel relational database systems have slowly come of age. The basic technology is now in place and there are several commercial systems based on it. The key techniques for parallelism comprise *data partitioning*, where the tuples of a relation are partitioned over several disks (and processors), *pipelining*, where the tuples "flow" from one operator to another without being written temporarily to disk, and *partitioned execution*, where the same operation is performed in parallel on different partitions of the data. However, even though the key ideas are established, the diversity of parallel architectures results in several interesting problems. First, there is the merit and suitability issue for any parallel architecture. Second, the diversity results in plurality of software and algorithm designs. As we get caught up in the differences, there arises an inability to use what each of the architectures has to offer. Furthermore, the proposed query processing algorithms have generally tended to ignore the details of the relevant parallel architecture and have, therefore, resulted in algorithms that perform suboptimally.

The two popular and commercially successful parallel architectures are shared nothing, where nodes comprising CPU, memory, and disks are connected by a high performance network, and shared memory (or shared everything), where the CPU's share the memory and have access to all the disks.

In the past, there have been studies comparing or advocating the relative merits of different parallel architectures. For example, Bhide [Bhi88] compares the shared-memory, the shared-nothing and the shared-disk architectures for on-line transaction processing (OLTP) queries. Frequently such studies compare significantly different architectures, and tautologically claim that they are different. There is no database study comparing the different software architectures[1] and on the same or similar hardware platforms. Since it is possible to provide

[1] Software architecture is the programming paradigm used in parallel programming, the common ones being "message passing" and "fine-grain shared memory". Though many times the software architectures go hand in

any software architecture on any hardware, one should compare the different software architectures on similar hardware architectures as done in [CLR94] or compare different architectures that are dollar cost-wise equivalent.

First, we start with the belief and the argument that both of these hardware architectures differ only in the speed at which a different processor's memory can be reached (as compared to the local processor). This results in the observation that shared-nothing algorithms can always be used on shared-memory machines because shared-nothing algorithms assume the worst (that sharing is expensive). On the other hand, though one can not really use naive shared-memory algorithms on shared-nothing hardware with a software shared-memory support, it is possible to use some of the shared-memory techniques in shared-nothing algorithms.

Second, we show that a naive understanding of these architectures results in algorithms that perform suboptimally. Understanding the exact tradeoffs involved in any parallel architecture helps in improving the performance. We illustrate this via the examples of parallel hash based aggregation algorithms. We show that whereas on the shared-nothing architecture, "expensive" redistribution of data can improve performance by reducing the amount of I/O, on the shared-memory architecture the inexpensive latching (and waiting) can actually be quite expensive in practice.

1.2 A Review of Parallel Architectures

Most commercial parallel architectures comprise several CPU's (and cache memory units) connected to one or more main (random access) memory units. Parallel database architectures can be broadly classified as shared-nothing architectures, where the main memory is private to a CPU and is not accessible to another CPU, and the shared-memory architectures, where all CPU's can access the entire main memory on the machine. The hardware architectures, in practice, are also closely linked to the software architectures in the machines. The following sections review the basic structure, cost and performance of the shared-nothing and shared-memory architectures.

hand with the hardware architecture, that need not always be the case and increasingly the trend is towards supporting all software architectures on a hardware platform.

1.2.1 The Shared Nothing Architecture

As mentioned, the shared-nothing hardware architecture is characterized by the fact that CPU of a node can not access the memory or disks of another node directly in hardware. The specific architecture varies as to the how independent the nodes might be. In loosely coupled systems like clusters of workstations, the nodes are almost totally independent, whereas in tightly coupled systems like the CM-5, the nodes are connected by a custom high speed network and the operating system allows certain optimizations like user-level message passing. A generic shared-nothing architecture is illustrated in Figure 1.

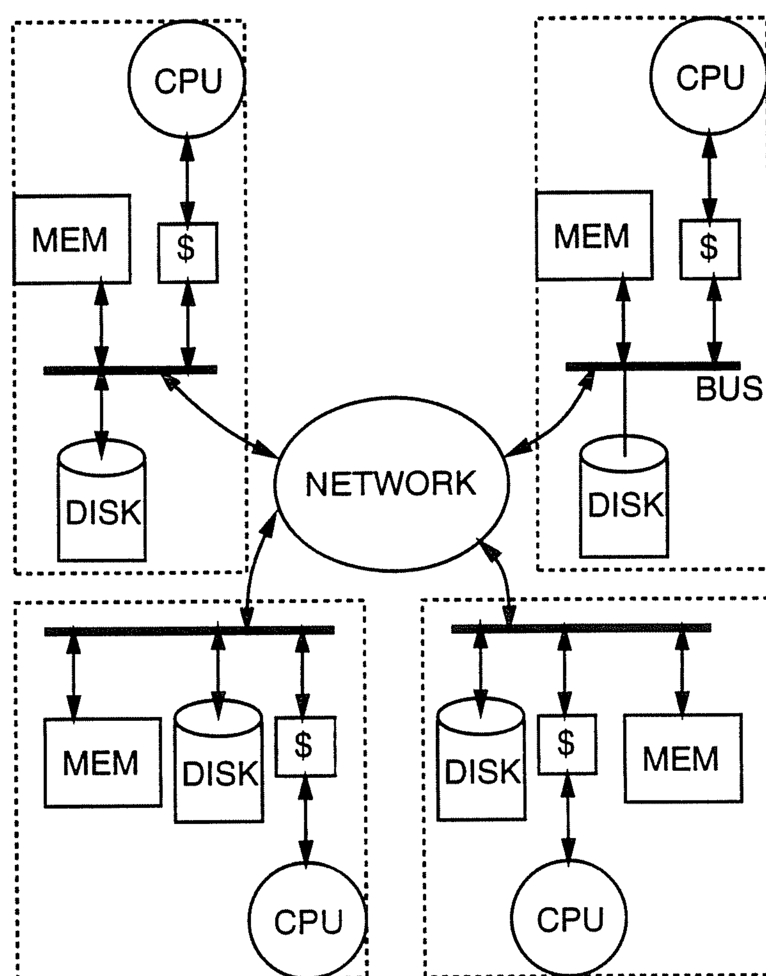

Figure 1: Block Architecture of a Shared Nothing Machine

As evident in the figure each node comprising the CPU, cache, private memory, and disks is connected to the other nodes via an interconnection network. In most modern shared-nothing machines, the interconnection networks are very fast and optimized for large bulk transfers.

Such a hardware architecture suits well the message passing or shared-nothing software architecture. Since private memory of a node is not directly accessible, all requested information has to be sent to the node making the request. Most algorithms designed for the shared-nothing hardware assume that message passing is the only way of communication between processors. This bias is further aggravated by the fact that most shared-nothing machines are optimized for large bulk (message based) transfers [Cha96].

However, it is possible to build shared-memory software architecture on a shared-nothing hardware. Shared virtual memory [LH89, CBZ91] provides a single virtual address space shared by all the processors in a shared-nothing architecture. This is achieved through memory mapping managers that implement the mapping between shared virtual memory and local (physical) memories. The structure and cost/performance of shared virtual memory is discussed further in Chapter 2.

1.2.2 The Shared Memory Architecture

The generic structure of a shared memory multiprocessor is as follows. There is a shared address space and each processor has a private cache memory for speeding up the access to the shared address space. Thus the processors rely on the cache memory for achieving high performance as going to the main memory is fairly expensive. Commonly, in the SMP architecture the processors share the memory bus and the main memory as illustrated in Figure 2.

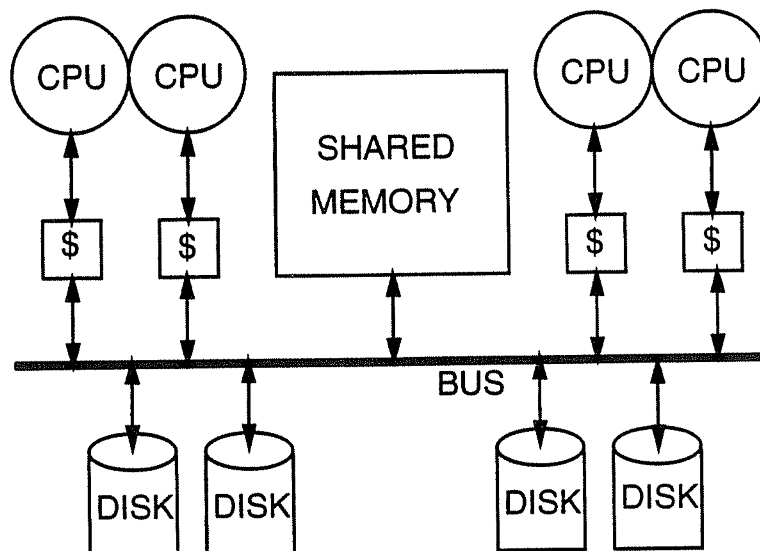

Figure 2: Block Architecture of a Shared Memory Machine

The data in the private cache memory is kept coherent. Coherence implies that the data values are guaranteed to be consistent under some coherence semantics, the most common being the sequential semantics i.e. a read returns the value of the most recent preceding write. There are several cache coherence algorithms mentioned in the literature [AB86]. For SMPs, which have a shared bus, a possible algorithm is the snooping bus write invalidate. In this algorithm, all cache controllers listen on the system bus and when a write occurs on a cached address, the cached value is invalidated. This is possible because the cache is write through i.e. all writes go to the memory. On a read, the value is read into the cache from the shared memory.

For example, a "message," i.e. a write to an address followed by a read of the same address on a different processor, necessarily requires two (slow) memory accesses under the write invalidate protocol, one for write and another for read as any cached copy will be invalidated on the write. This makes the communication of the data value comparatively slow compared to non-shared memory accesses most of which are expected to be hits in the local cache.

The naive theoretical PRAM (Parallel Random Access Memory) model [FW78] of parallel computation assumes that all memory accesses are of uniform cost. This assumption is commonly made in design and analysis of query processing algorithms for shared memory. However, it is clear that SMPs do not match the PRAM model. In fact, another, and perhaps better, way to look at the shared-memory machine is to assume that the machine is a distributed-cache machine and that there is communication between two processors, with its associated costs, when the two processors access the same address at different points of time.

Shared memory software architecture usually bases itself on the PRAM or somewhat improved memory models like Blocked PRAM. It allows any processor to access any memory at any time. Such architecture naturally promotes fine grain sharing. However, for improved performance, one is expected to follow better memory models and program accordingly. Unfortunately, such limitations only erode one of the most touted benefits of shared-memory programming model i.e. ease of programming.

Implementing the message passing programming model on top of shared-memory hardware is fairly easy. In fact, some shared-memory machines now provide the standard MPI message passing interface [Mes94]. Unfortunately, message passing, implemented naively and in a transparent way usually shows inferior performance than using even inefficient shared memory.

Efficient message passing must have no extra memory copies (a bane of most message passing systems). Implementing it requires that the message buffer allocation and deallocation is done outside the encapsulation of the message passing library. Some details are described in Chapter 3.

1.3 Thesis Contributions and Organization

This thesis comprises a study of architectural considerations for parallel hash-based join and aggregation algorithms. The two threads that bind the chapters together are 1) an emphasis on the similarity of the shared-nothing and shared-memory architectures, and 2) an emphasis on an improved understanding of the interaction of the two architectures and hash-based join and aggregation algorithms, and the resulting performance dividends.

In Chapter 2, we begin by showing how software shared-memory (also called shared virtual memory) on shared-nothing hardware can be used in hash join processing for better load balancing, a forte of shared-memory architecture. This shows that the two architectures are not entirely disparate. Then in Chapter 3 we show that both shared nothing and shared-memory hash join algorithms perform well on shared-memory hardware showing that the two software architectures are quite comparable. We also propose join algorithms that explicitly exploit the shared-memory architecture showing how better understanding of the architecture results in improved performance.

The following two chapters, Chapter 4 and 5, study hash based aggregation on shared-nothing and shared-memory architectures respectively. In Chapter 4, we show how our common-sense notion—message passing in a shared-nothing system should be minimized—can result in a suboptimal algorithm. Algorithms are proposed that exploit the tradeoff in memory consumption and communication, and perform better than traditional approaches. Finally, in Chapter 5, again via the example of hash-based aggregation we show that our common-sense notion—latching and waits for latches in shared-memory are not that expensive—results in suboptimal algorithms. Algorithms that take into account the latch conflicts and hence perform better on a shared-memory machines are proposed. The chapter also reconfirms the conclusions of Chapter 3 by showing that shared-nothing algorithms perform comparably to shared-memory algorithms on shared-memory hardware. Conclusions and future work are presented in Chapter 6.

Chapter 2

Hash Join using Shared Virtual Memory

This chapter shows that software shared memory support in a shared-nothing multiprocessor, also called shared virtual memory, facilitates the design and implementation of parallel join processing algorithms that perform significantly better in the presence of skew than previously proposed parallel join processing algorithms. We propose two variants of an algorithm for parallel join processing using shared virtual memory, and perform a detailed simulation to investigate their performance. The algorithm is unique in that it employs both the shared virtual memory paradigm and the message-passing paradigm used by current shared-nothing parallel database systems. Thus we show that the two software architectures: shared-nothing and shared-memory are not antagonistic and that each can benefit from the other.

2.1 Introduction

Shared-nothing multiprocessors can easily be equipped with shared virtual memory (henceforth called SVM) providing a globally shared address space. Since shared-nothing multiprocessors have emerged as the platform of choice for scalable multiprocessor database systems, it is natural to ask if multiprocessor database systems can make good use of SVM. In this chapter, we argue that the answer is yes; specifically, we show that SVM facilitates the design and implementation of parallel join processing algorithms that perform significantly better in the presence of skew than previously proposed parallel join processing algorithms. We propose two variants of an algorithm for parallel join processing with SVM, and investigate their performance in a detailed simulation of a shared-nothing/SVM multiprocessor database system.

Shared virtual memory is an attractive facility since it provides the illusion of a shared memory where there is no actual shared memory in the underlying hardware. This is similar to

the way in which standard virtual memory provides the illusion of a large memory even when the actual physical memory is limited. It is interesting to speculate on how one might build a parallel database system from scratch given the availability of SVM. While our work sheds some light on that topic, our present goal in the thesis is much more modest: we wanted to see if existing parallel database systems, designed and implemented for the shared-nothing paradigm, can be modified easily to take advantage of SVM showing the similarity and complementarity of the two architectures. One interesting result of our investigation was the development of a dual-paradigm algorithm. Our parallel join processing algorithm uses both message passing and SVM. Briefly, stream-oriented processing is handled by message passing, while access to shared data structures is provided in SVM.

Both of our parallel join processing algorithms are based upon the parallel hybrid hash join [DG85, SD89]. In the absence of skew, this algorithm has been shown to have the best performance. However, in the presence of skew, the performance of hybrid hash join degrades since the response time of the parallel join is limited by that of the slowest processor in the join. Our solution to this problem is *load sharing*—whenever a processor finishes, it checks to see if any other processors are still running. If there are other processors still running, the newly idle processor picks a busy processor and begins to share that processor's portion of the join processing. SVM provides an ideal mechanism by which to implement this load sharing thus showing the need and use of shared-memory techniques in a shared-nothing architecture.

While the idea behind this scheme is straightforward, some care must be taken in the design of the algorithm or truly abysmal performance will result. Naive implementations of shared-memory join processing algorithms in SVM suffer from (1) network thrashing due to multiple processors updating shared-memory pages, and (2) disk thrashing due to too many pages being sent to a single processor. Our dual-paradigm algorithm is specifically designed to avoid both of these problems. Further desirable properties of this algorithm are that, unlike most previously proposed skew-handling join processing algorithms, in the "no skew" case the performance is virtually identical to that of parallel hybrid hash, and that the algorithm generalizes easily to handle multi-way joins.

2.2 Brief Overview of SVM

Shared virtual memory [LH89, CBZ91] provides a single virtual address space shared by all the processors in a shared-nothing architecture. This is achieved through memory mapping managers that implement the mapping between SVM and local (physical) memories.

2.2.1 Structure of SVM

Memory mapping managers are responsible for keeping the memory coherent, i.e. the value returned by a read operation is always the same as the value returned by the most recent write operation. One way to ensure coherence is to use the write-invalidate protocol. This protocol allows multiple reader processes to share a page by replication but a writer process obtains an exclusive copy by invalidating the other copies. There are several alternatives to implement the mapping managers using this idea. These are discussed at length in [LH89].

When a process wants to access a page which is not in its physical memory, it suffers a *page fault.* The missing page is then brought in either from the memory of some other processor or from the disk. This is done by the mapping manager.

2.2.2 Performance and Cost

The performance of parallel programs in such a system depends on two factors.

1. The number of parallel processes.
2. The amount of updating of shared data

Non-shared and read-only pages have relatively little effect on the performance. Furthermore, updating shared data does not cause thrashing if the program exhibits *processor locality* of reference, that is, all recent references to a data object come from the same processor. A parallel algorithm for such a machine, therefore, must attempt to minimize updatable shared data and maximize processor locality.

The cost of an SVM system lies in the interconnect traffic involved in servicing the page faults. The traffic consists of

1. Control messages to locate the page and to ensure coherence.
2. Actual data transfers required to maintain consistency.

Different coherence algorithms differ mainly in number of control messages required to service a page fault. This is mainly because page faults depend principally on program properties and only indirectly on the coherence mechanism.

In recent years, research into SVM have produced significant performance enhancements by exploiting specific patterns of sharing in the system [CBZ91]. This results in a reduction in the number and size of messages. Using weaker forms of consistency for shared data also allows buffering of updates, further reducing the number of messages required to maintain consistency.

2.3 Data Skew: Problem and Previous Solutions

Initial work in parallel join algorithms implicitly assumed that values in the base relations were uniformly distributed over the domain. In practice, this implied that given any unbiased partitioning strategy, each processor was likely to have the same amount of work to do. With time this assumption has been challenged (see, e.g. [LY90, SD89]) by the claim that many real data sets are not uniform but suffer from data skew. In the presence of such skew, an unbiased partitioning strategy like hashing will result in unequal load on participating processors. This worsens the response time of the algorithm since other processors have to wait for the loaded processor(s) to finish.

In the past, several algorithms have been proposed to meet this challenge. Unfortunately, most of these algorithms either perform much worse than hash-partitioned parallel hybrid hash in the no skew case, or only perform well for limited classes of data skew. Also, these algorithms have no obvious extension to handling multiway joins without storing the intermediate relations. A notable exception is a recently proposed algorithm by Poosala and Ioannidis [PI96]. We discuss the prominent ones in brief.

Walton et al. [WDJ91b] present a taxonomy of skew in parallel databases. They made the distinction between *attribute value skew*, which is skew inherent in the dataset, and *partition skew*, which occurs in parallel machines when the load is not balanced between the nodes. Different kinds of partition skew can be classified as *initial tuple placement skew* where initial placement of the tuples on nodes is skewed, *selectivity skew* where different nodes select different number of tuples based on the "where" clause, *redistribution skew* where redistribution causes different number of tuples to be processed at each node, and *join product skew* where the join selectivity is different across the nodes i.e. the result sizes are different across the nodes.

The algorithm of Wolf et al. [WDYT90] analyzes the base relations by doing an initial scan on them. This information is used in the actual repartitioning of the relations. Using an analytical model to compare the scheduling hash-join algorithm of [WDYT90] and the hybrid hash-join algorithm of Gamma [DG85, SD89, DGS+90], Walton et al. conclude that scheduling hash effectively handles redistribution skew while hybrid hash degrades and eventually becomes worse than scheduling hash as redistribution skew increases. However, unless the join is significantly skewed, the absolute performance of hybrid hash is significantly better than that of scheduling hash due to the absence of the initial scan of relations.

Schneider and DeWitt [SD89] conclude that the parallel hash-based join algorithms (Hybrid, Grace, and Simple) are sensitive to redistribution skew resulting from attribute value skew in the "building" relation (due to hash table overflow) but are relatively insensitive to redistribution skew in the "probing" relation. This result is confirmed in our work.

The bucket-spreading parallel hash join [KO90] and its variant tuple interleaving hash join [HL91] balance the redistribution skew by ensuring that processors get approximately same number of tuples for the final join phase. This involves sending the tuples twice over the network. Adaptive load balancing hash join [HL91] algorithm attempts to balance the load statically by relocating buckets after initial partition. Its extended version [HL91] is similar to tuple interleaving hash join but avoids the extra network cost by storing tuples locally. All these algorithms suffer from two major problems: (1) they can not exploit memory the way hybrid hash join algorithm does thus not performing optimally (this difference is analogous to the difference between hybrid hash and GRACE algorithm); and (2) they do not handle join product skew.

Omiecinski [Omi91] proposed a load balancing hash-join algorithm for systems running on shared physical memory multiprocessors. The algorithm is based on the bucket-spreading algorithm of Kitsuregawa and Ogawa [KO90]. Analytical and limited experimental results from a 10 processor Sequent machine show that the algorithm is effective in limiting the effects of attribute value skew for double-skew joins, but again the author did not compare the performance of the algorithm with basic parallel join algorithms.

Lu and Tan [LT91] present a dynamic task-oriented algorithm for load balancing in a shared-disk and hybrid environment having both private and shared memory. Our algorithm also uses the idea of load sharing for skew handling, but the details of how the load sharing is accomplished

(as well as the hardware platforms for which the algorithms were designed) are quite different. The Lu and Tan algorithm requires a preprocessing phase which scans both the relations to create tasks, compared to our algorithms which scan and redistribute both the joining relations exactly once. While they did not implement their algorithm nor simulate its performance, their analytical model shows that it is effective in handling attribute value skew.

Poosala et al. [PI96] use query-result size estimation using histograms for load balancing in parallel hybrid hash join algorithm. The general idea is to partition the work, estimate the join load of each partition partition using the histogram, and then assign the partitions to processors such that the total load on each processor is about equal. Performance evaluation shows the algorithms to be quite promising in handling data skew.

DeWitt et al. [DNSS92] investigate the use of sampling coupled with range partitioning (instead of hash partitioning) to balance the work among processors by mitigating redistribution skew. The algorithm was very successful in handling redistribution skew, but much less successful in dealing with join product skew. Briefly, the reason for this is that it is very difficult to detect and quantify join product skew by sampling. In this paper, we show a way of integrating this approach of sampling-based range partitioning with SVM for load balancing. The result is an algorithm that deals successfully with both join product skew and redistribution skew.

Shared virtual memory has received very little attention in previous database literature—the only work of which we are aware is some early work by the Hsu et al. on transaction processing in an SVM system (see [HT89, HT88] for examples of this work). To our knowledge, no work has appeared on query processing in systems with SVM.

2.4 Join Processing using SVM

In this section we consider approaches to using SVM for join processing. First, we show that true shared-memory algorithms do not perform well in an SVM environment because (1) they are not careful to ensure processor locality in the building phase of hash join processing, and (2) they exhibit a tendency to "swamp" the system by replicating all hash table pages throughout the system in the probing phase of hash join processing. Then we present our approach, which uses both SVM and messages, and avoids both these problems.

2.4.1 Naive Approaches

To show why simply implementing a shared-memory algorithm in an SVM environment does not work, consider the parallel hybrid hash join algorithm. A shared-memory version of this algorithm would proceed in two phases:

1. All processors scan the building relation, adding tuples to a global hash table as they go, then

2. all processors scan the probing relation, doing a lookup in the global hash table for each scanned tuple.

In an SVM system, phase 1 of this algorithm would cause terrible thrashing between processors as each processor competes to update the pages holding the shared hash table entries as the to-be-updated-pages have to be moved to the processor making the update. Phase two of the algorithm would be no better, since at this point the hash table pages are read-only, hence they would be replicated throughout the multiprocessor. Unless the memory of each processor was large enough to hold the entire global hash table, many of these hash table pages would thrash to and from the local disks of the processors. Other hash based algorithms [LTS90, Omi91] suffer the same fate for the same reasons, lack of processor locality and swamping due to replication of hash tables.

When we began our work on this problem we developed a series of algorithms that attempted to use the SVM carefully to avoid these two problems. However, we soon realized that in some places the algorithms were merely attempting to mimic the message-based parallel hybrid hash algorithm in SVM—that is, a message send was accomplished by a write into a specific page followed by the setting of a shared flag, a message receive was accomplished by a check on the shared flag followed by a read of the specified page. This realization led us to develop the dual-paradigm algorithm described in the next subsection.

2.4.2 A Dual-Paradigm Algorithm

Our dual-paradigm algorithm begins exactly like the basic parallel hybrid hash algorithm [DGS+90] which is described below. Suppose the join is of relations R and S, say with the join condition $R.A = S.B$, and that R has been chosen as the building relation. At a high level, the basic parallel hybrid hash join [DGS+90] proceeds in two stages:

1. Build.

   Each processor p_i scans its local fragment of R. As each tuple r in R is processed, the processor computes a hash function $h_1(r.A)$. This hash value is used to determine to which processor r should be sent by a lookup in a data structure called "split table," which is just a list of (hash value, processor number) pairs. The tuples of R received at processor p_i form the partition R_i.

   As a processor p_i receives an incoming R tuple r (a member of R_i), p_i applies another hash function $h_2(r.A)$, which determines to which local hash bucket r belongs. The hybrid hash algorithm keeps one local hash bucket in memory (bucket zero), and spools the rest to its local disk. An in-memory hash table is built out of the tuples that fall into local bucket zero, for use in the probing phase of the join. We will refer to this in-memory hash table as a "local bucket hash table" in the following.

2. Probe.

   Each processor p_i scans its local fragment of S. As each tuple s in S is processed, the processor computes the hash function $h_1(s.B)$ and looks up this value in a split table to determine to which processor s should be sent.

   When a processor p_i receives an incoming S tuple s, it applies the hash function $h_2(s)$ to determine to which local bucket s belongs. If this local bucket is bucket zero, p_i immediately probes the local bucket hash table for any matching tuples; otherwise, s is spooled to disk into the appropriate local bucket of S_i.

   After S has been redistributed, the join of the local bucket zero's has been completed. Then each processor p_i repeatedly reads a local bucket of R_i into memory, builds a local bucket hash table out of the R tuples in R_i, then scans the corresponding local bucket of S_i to find all joining tuples, until all local buckets have been processed.

In our dual paradigm algorithm, the algorithm changes in a few ways. First, the hash tables built out of the local buckets are built in SVM. That is, p_1 builds all of its local bucket hash tables in SVM; p_2 builds all of its local bucket hash tables in SVM; and so forth. Note that while these hash tables are being built, they are only updated locally. That is, p_1 does not insert any tuples into p_2's local hash table. This is critical to maintaining processor locality.

To see how the algorithm proceeds, for simplicity of exposition, suppose for the moment that there is only one local bucket at each processor (bucket zero). Suppose that some node, say p_1, finishes its local join processing for bucket zero.

At this point, p_1 checks to see if there are other busy nodes. If there are busy nodes, p_1 chooses a busy node, say p_2, and takes over some of the join processing for p_2.

The way this works is as follows: p_2 maintains a forwarding table that contains only p_2 initially. The purpose of the forwarding table of a processor is to distribute the probe tuples to each of the processors participating in its probe phase in a round-robin manner. p_1 inserts itself in the forwarding table of p_2 changing it to $\{p_1, p_2\}$[1]. The meaning of this change is that during the redistribution of S, any tuple s whose hash value indexed into processor p_2 could now be forwarded to p_1 with equal probability. Thus, effectively, the stream of tuples in S_2 is now evenly divided between p_1 and p_2.

Now consider a tuple s that would have been sent to p_2 originally but is now sent to p_1 because of the updated forwarding table. Processor p_1 then proceeds to probe the local bucket hash table built out of bucket zero of S_2; that is, it probes the hash table on p_2. Note that this is trivially possible since the hash table was built in SVM. Initially, this probe is likely to cause a SVM fault. However, eventually there will be no more SVM faults since the hash table pages will have been replicated to p_1. Note that no thrashing will occur, since the hash table pages are read-only at this point. Also, since p_1 holds only replicated copies of pages for the hash buckets built on p_2, there is no danger of replicated pages swamping its memory. Effectively, the local bucket hash table at p_2 has been replicated to p_1 in a lazy, "copy-on-reference" fashion.

Note that this accomplishes a dynamic subset-replicate [ESW78] join of R_2 and S_2, with R_2 being replicated (in the hash tables) and S_2 being subsetted (by the random selection of p_1 or p_2.) In this way, once p_1 has updated the forwarding table entry for p_2, p_1 and p_2 split the work that was originally allocated to p_2.

The only remaining modification concerns buckets 1 through n. The required change to the basic parallel hybrid hash is that the local reads that scan the buckets of the S_i fragments from disk must also send these buckets through the forwarding table, instead of immediately probing the local hash tables for the buckets of R_i. This is to allow the subsetting of the S buckets other than bucket zero. It is possible that this could require an extra redistribution of some

[1]Set valued split tables were also used in [DNSS92].

probing tuples; however, in today's multicomputers, with up to 200 MB/sec interconnects, the cost of this redistribution will be insignificant when compared to the cost of the read off the disk. A simplified pseudocode version of the join operator code appears in Figure 3.

We see that this algorithm does not suffer the faults of other shared-memory algorithms. Almost all updates to shared variables, which are few to begin with, are done locally. Remote accesses to status variables are required only rarely (for example, when a processor actually looks for another busy processor). Also, synchronization is kept to a minimum. The major sharing, that of hash tables, is only in read-only mode and hence it causes no further traffic beyond its one-time replication. The building phase is done individually per processor to ensure that two processors never update a hash table simultaneously. This results in a small chance that a processor will be idle because if the other processor is not done building, the joining processor will have to wait. In order to redistribute the build relation more evenly, an estimate of the value distribution of the join attribute of the build relation is required. This can be done in various ways: for example, by sampling the relation and estimating the distribution, as discussed later, or by using detailed statistics, like histograms, on the relation.

Thus, instead of precomputing the load at each node and balancing it, as is done in many previous proposals for skew handling join algorithms, each node follows the policy of "don't be idle if there is work left" using a simple heuristic for selecting a processor. Eager et al. [ELZ86] shows and we verify that a simple heuristic, like random, for selecting a busy processor works almost as well as more complex kinds (e.g. those involving estimated work left).

2.4.3 A Sampling Variant

The algorithm as described in the previous subsection deals elegantly with join product skew by being adaptive to the observed join work load, but it is still vulnerable to redistribution skew in the building phase. To see this, note that there is no provision for moving hash table pages from one processor to another during the building phase.

To handle this problem, we can use the technique proposed in [DNSS92]. The idea is that instead of using hashing to partition the relation among the processors of the system, we use range partitioning. The cutoff values for the ranges can be found approximately by sampling at a very low cost; these cutoff values are chosen so as to equalize the number of tuples sent to each processor. The modification of the dual paradigm algorithm to incorporate range partitioning

```
/* hash table for a bucket, NPROCS is the number of nodes */
shared HASHTABLE_t hash[NPROCS];
/* status variable indicating processor status for polling */
shared STATUS_t status[NPROCS];
/* processor → {processor} forwarding map */
shared INT_set map[NPROCS];/* initially map[a] = { a } */
/* code for processor p */
foreach bucket do{
        while (not end of build relation){
                read tuples from it
                insert tuple t in the hash table hash[p]
        }
        while (not end of probe relation){
                read tuples from it
                if (other processors in map[p])
                    distribute the tuples to them in a round robin way
                probe the hash table hash[p] if the tuple is not forwarded to a different processor
                if (match)
                    send the result tuple to the next operator
        }
        if (I am last to finish)
            wait at barrier
        else {
            while (other processors are still at work){
                    map[p] = φ
                    find some busy processor, let it be lp
                    /* indicate to the node that I too am working for lp */
                    map[lp] = map[lp] ∪ { p }
                    while (not end of input from the node lp){
                            read tuples from it/* originally for processor lp */
                            probe the hash table hash[lp]
                            if (match)
                                send the result tuple to the next operator
                    }
            }
            wait at barrier
        }
}
```

Figure 3: Simplified Pseudocode for the Basic/SVM Algorithm

by sampling is straightforward; merely replace $h_1(r.A)$ in the preceding discussion by a lookup that determines in which range $r.A$ falls.

2.4.4 Is SVM Necessary?

The main reasons why we consider SVM necessary for the design and implementation of the algorithms described above are (1) ease of coding and (2) ease of conceptualization in terms of shared data structures. In principle, of course, one can always simulate the entire SVM behavior in the program itself. But just considering how one would implement a simple shared status variable (which other processors can look at) by message passing[2] should be enough to convince the necessity of basic SVM support for implementing any dynamic load balancing algorithm. Sharing a data structure would be all the more complex. For example, just shipping a hash table would involve marshalling the whole data structure in a message (all of which must be sent at once, not like the lazy shipping SVM provides) and must be unmarshalled by the receiving node.

2.5 Performance Evaluation

We compare the performance of four algorithms.

Basic Basic Parallel Hybrid Hash Join.

Sampling Sampling based Range Partitioning Hybrid Hash Join. This is the algorithm from [DNSS92].

Basic/SVM Basic SVM Hybrid Hash Join. This is the dual-paradigm algorithm discussed in Section 2.4.2.

Sampling/SVM Sampling based SVM Range Partitioning Hybrid Hash Join. This is the sampling variant referred to in Section 2.4.3.

Our experiments show that if the skew in building relation is not severe, then Basic/SVM algorithm performs best, with performance coming close to ideal in many cases. Unlike most other previously proposed skew handling algorithms, except one proposed by Poosala and Ioannidis in [PI96], it suffers almost no performance degradation under the no skew case. If the skew

[2]Every update will result in a message to all the nodes. The nodes must have a separate thread waiting for such message and upon receiving the message take appropriate action.

CPU Cost Parameter	No. Instr.	Config./Node Parameter	Value
Initiate Select	20000	Tuple Size	104 bytes
Initiate Join	40000	Number of Disks	1
Initiate Store	10000	CPU Speed	15 MIPS
Terminate Store	5000	Memory Size	16 MB
Terminate Join	10000	Page Size	8 KB
Terminate Select	5000	Latency for 8K Message	1.8 msec
Read Tuple	300	Disk Seek Factor	0.617
Write Tuple into Output Buffer	100	Disk Rotation Time	16.667 msec
Probe Hash Table	200	Disk Settle Time	2.0 msec
Insert Tuple in Hash Table	100	Disk Transfer Rate	3.09 MB/sec
Hash Tuple using Split Table	500	Disk Cache Context Size	4 pages
Apply a Predicate	100	Disk Cache Size	8 contexts
Copy 8K Message to Memory	10000	Disk Cylinder Size	83 pages
Message Protocol Costs	1000	Sampling Overhead	0.6 sec

Table 1: Simulation Parameter Settings

in building relation is extreme, then Sampling/SVM algorithm does best. This is because it mitigates redistribution skew by doing range partitioning.

2.5.1 Simulation Methodology

The simulator is based upon an earlier, event-driven simulation model of the Gamma parallel database machine [DGS+90]. The earlier model was a useful starting point since it had been validated against the actual Gamma implementation; it had also been used extensively in previous work on parallel database machines [GD90, SD90, HD91]. The new simulator, which is much more modular, is written in the CSIM/C++ process-oriented simulation language [Sch90]. The simulator accurately captures the algorithms and techniques used in Gamma. The remainder of this section provides a more detailed description of the relevant portions of the current simulation model, and concludes with a table of the simulation parameter settings used for this study. Since this study was completed in December 1992, some of the technology driven parameters, like CPU and disk speeds, have changed. However, we believe that these changes do not affect our results significantly.

A join query to be processed is sent by a simulated terminal to the Scheduler process for execution (described in the next paragraph) and then waits for the result. Base relations over which simulation is performed are drawn from Zipf distribution with parameter θ which determines the skew. At $\theta = 0.0$ the relation is uniformly distributed and at $\theta = 1.0$ the relation

is significantly skewed. Here we must emphasize that by the very nature of Zipf distributions, θ values less than 0.5 signify very small skew and that skew grows quickly with increasing values of θ. As a result, the resources required to simulate higher skew cases were exorbitant in some cases and they did not give us any further insight. Also, plotting for higher skew values resulted in losing the differences in the low skew case as plotting the high skew values stretched the scale of the graph so much that all the curves just about coincided for the smaller values of skew e.g. in Figure 6. Therefore, some of the graphs are plotted until only a moderately high skew value like 0.65. Table 2 shows the variation of the maximum number of repeated occurrences of a single value with θ, when both the domain of the Zipf distribution and the relation size are one million, indicating the above mentioned behavior of Zipf distribution.

θ	Max Repeated Value
0.0	1
0.2	13
0.4	155
0.5	518
0.6	1670
0.7	5091
0.8	14282
0.9	35546
1.0	75721

Table 2: Modeling Skew with Zipf Distribution

The size of the relations in our study, except where noted, is one million tuples. Except for the multi-bucket join experiment, the hardware configuration was chosen such that all joins were single bucket joins needing about 4 MB of memory. The number of processors in our study, except where noted, is 32, with one disk per node as mentioned before.

The simulator assumes an interconnection network of infinite bandwidth but limited speed. The decision to model such a network was made as a result of our experience with the Intel iPSC/2 Hypercube, on which Gamma runs, where network bandwidth has never been an issue. Given that the next generation of parallel processors (like the CM-5 and IBM SP-2) provide even higher capacity networks, we feel that this is a reasonable assumption. Communications packets do, however, incur a "wire" delay corresponding to the delay encountered on machines like the IBM SP-2 and the CM-5, and CPU costs for sending and receiving messages are included

in the model.

The Scheduler is a special process that accepts queries from terminals and decomposes each query into several communicating lightweight processes, one for each operator in the query plan (e.g., a two-way hash join consists of a pair of selects and a join process). Communication channels between operator processes are set up by the Scheduler before the query is executed, and operators pass data to each other in units of 8 KB messages (each of which incurs a simulated memory-to-memory copy operation). As each query arrives, the scheduler calculates the query's memory requirements and initiates its execution after resource allocation if sufficient memory is available.

In this work we are only concerned with select and join operators. Any result tuples from the queries are "sent back to the terminals," an operation that consumes some small amount of processor cycles for network protocol overheads[3]. The default join algorithm used in the simulator is the hybrid hash join algorithm [DG85, SD89].

The database itself is modeled as a set of relations. All relations are declustered [RE78, LKB87] (horizontally partitioned) across all the disks in the configuration. A hashed partitioning strategy is used, where a randomizing function is applied to the key attribute of each tuple to select a particular disk drive.

The performance of SVM is modeled explicitly for the shared hash tables under assumptions similar to those made in [LH89, CBZ91]. The status variables are ignored because they are updated very infrequently. Counts from the simulation indicate that they are updated about 8 times per node throughout the entire course of the join, so even if updates to these variables took some tens of milliseconds (which is an order of magnitude too high) the effect of these updates would not be detectable in the total run times.

The simulated disk provides a 256 KB cache that we divide into eight 32 KB cache contexts for use in prefetching pages for sequential scans. The CPU is scheduled using a round-robin policy. The buffer pool models a set of main memory page frames. Page replacement in the buffer pool is controlled via the LRU policy extended with "love/hate" hints.

The important parameters of the simulated DBMS are listed in table 1. The CPU speed, memory configuration, and network speed were chosen to reflect the characteristics of the

[3]Storing results to disk is ignored because all algorithms have the same cost for that step. Also, if extended to non-bushy multi-way joins, the intermediate results can be pipelined and not stored.

current generation of commercially available multiprocessors (e.g. the Thinking Machines CM-5). The software parameters are based on instruction counts taken from the Gamma prototype when the previous simulator was validated. The number of nodes in the simulated system is 32 with one disk per node. The relation size for both relations was one million 104 byte tuples each.

2.5.2 Experiments with Varying Data Skew

Single Relation Skew

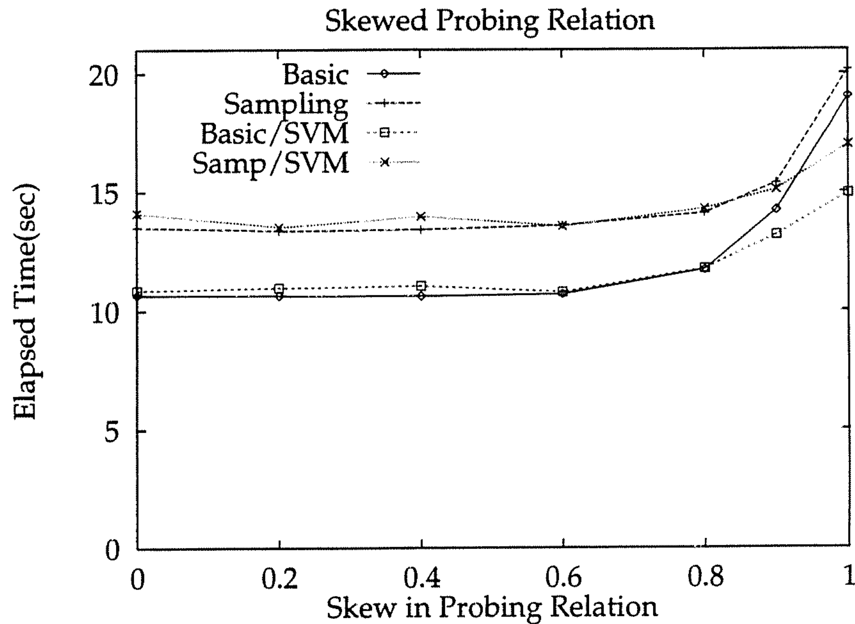

Figure 4: Single Relation Skew: Probe Relation

Figure 4 shows the performance of the four algorithms with no skew in the building relation and varying the skew in the probing relation using 32 nodes for one million tuple relations. We see that when only the probing relation is skewed sampling does not help (and in fact could make things worse, since the range partitioning is only approximate), and therefore initially Basic algorithm and Basic/SVM algorithm perform well. In the higher skew case we see the load sharing taking effect and the Basic/SVM algorithm and Sampling/SVM algorithm both do better than Basic algorithm and Sampling algorithm respectively.

Figure 5 shows the performance of the four algorithms with no skew in the probing relation

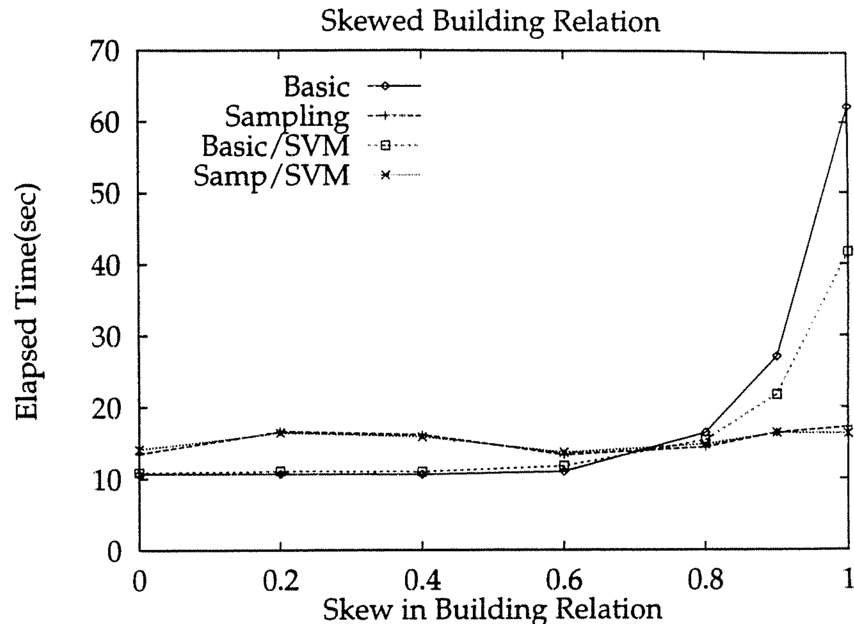

Figure 5: Single Relation Skew: Build Relation

and varying the skew in the building relation. Again we see than under low skew cases, sampling can make things perform a little worse for the reasons mentioned above. But under moderate to high skew it helps tremendously because it balances the building load on each processor. As mentioned earlier, this case is not handled well by the Basic/SVM algorithm because overflow resulting from skewed build file results in too much imbalance. In all cases, however, the SVM counterpart of the algorithm outperforms the algorithm without using SVM.

Join Product Skew

Figures 6 and 7 consider the case of join product skew using 32 nodes for one million tuple relations. The two graphs represent the same experiment; in Figure 7 the lines for Basic algorithm and Sampling algorithm have been removed so that the differences between the remaining algorithms would be apparent. From the Figure, it is clear that join product skew is the case where SVM based algorithms perform better by a great margin. Even for moderate skew the proposed algorithms do significantly better than the algorithms that do not use SVM. This is because the CPU load is shared cleanly in this approach without any significant overhead, and since all processors are working almost all the time work gets done in almost the minimum time possible.

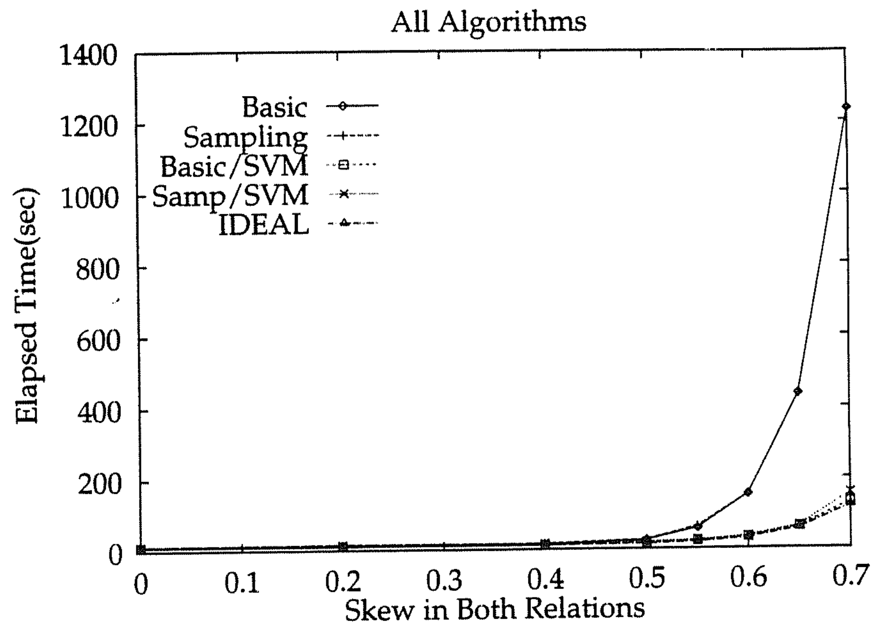

Figure 6: Join Product Skew, Both Relations are Equally Skewed: All Algorithms

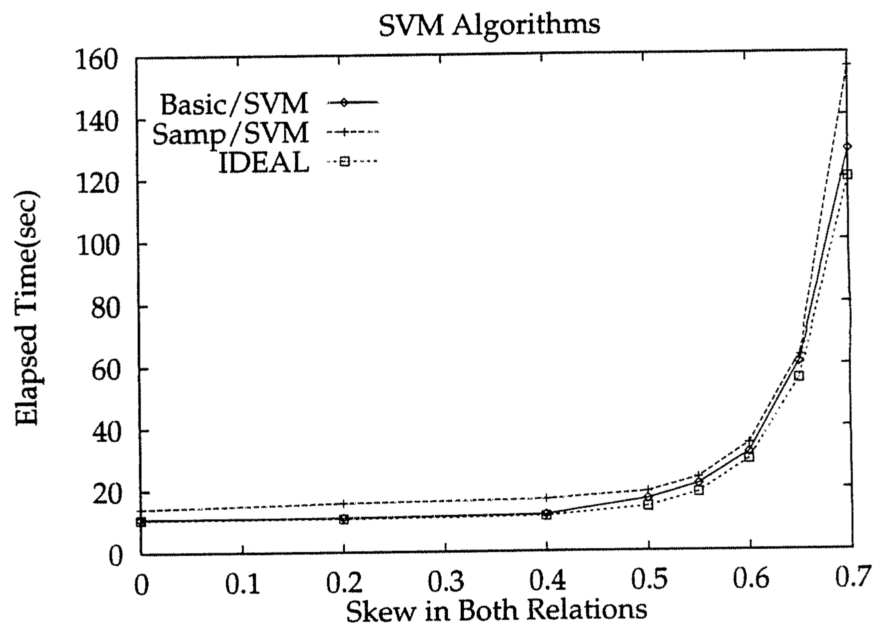

Figure 7: Join Product Skew, Both Relations are Equally Skewed: SVM Algorithms

In the figures the line labeled "IDEAL" was generated by simulating a join with no skew that produced the same number of result tuples as the skewed join. We included the line to emphasize that most of the increase in the running time of the SVM algorithms, with increasing value of θ, is due to the increase in the result size (even if it is not stored, it takes time to compute). Note that in the single skew experiments the join output size was constant across the different values of θ, so no equivalent affect was present.

Other Variations

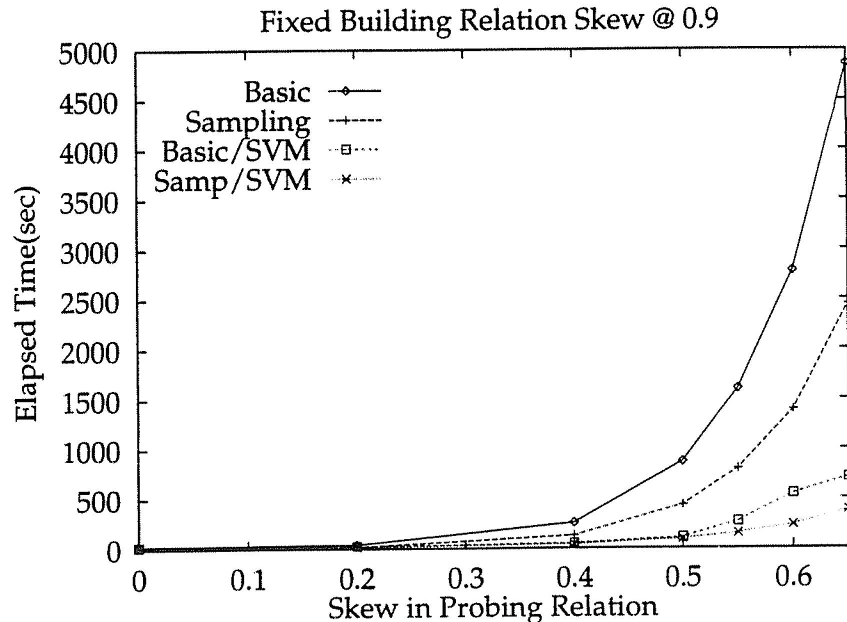

Figure 8: Double Skew, Build Relation skewed at $\theta = 0.9$

To get a better feel of the performance of the algorithms we also studied other variations using 32 nodes and one million tuple relations. In the first experiment, we fixed the skew in the building relation at $\theta = 0.9$ and varied the skew in the probing relation. In the second, we fixed the skew in the probing relation at $\theta = 0.9$ and varied the skew in the building relation. The results are shown in Figures 8 and 9. Both figures reconfirm our earlier results viz. 1) sampling helps if the building relation is highly skewed making Sampling/SVM algorithm perform the best, e.g. when build relation is skewed with $\theta = 0.9$; if the build relation is not extremely skewed then Basic/SVM algorithm does better, and 2) that SVM algorithms do significantly better than their pure message passing counterparts.

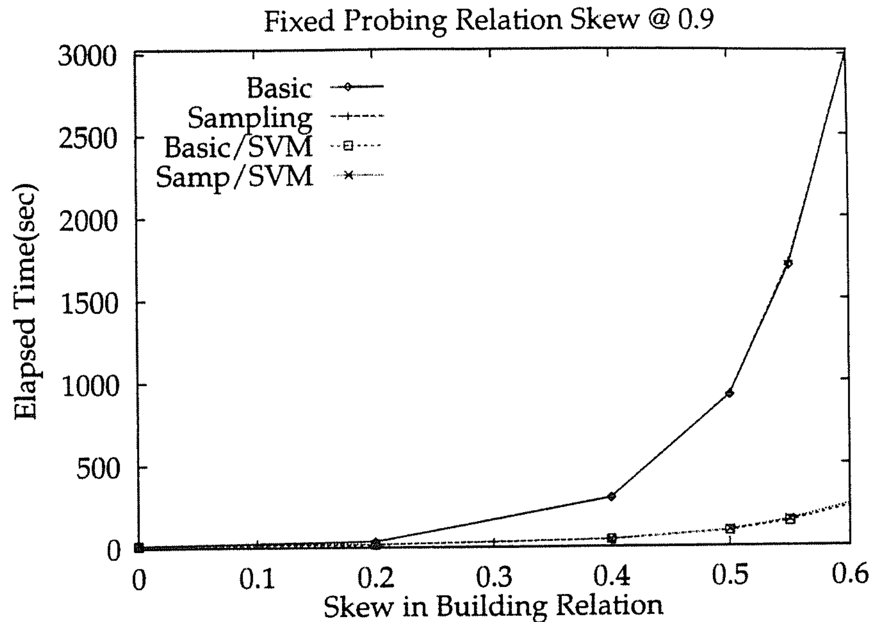

Figure 9: Double Skew, Probe Relation skewed at $\theta = 0.9$

2.5.3 Multi-Bucket Join

To show that the algorithm performance is quite independent of the number of buckets in the join processing, we ran a multi-bucket join experiment (see Figure 10). The memory size of a node was reduced to 1 MB which resulted in a four bucket join. The results of the experiment again verify that the SVM algorithms do significantly better than the originals. However, range partitioning becomes more important here because of the greater need to balance the I/O on the nodes as the amount of I/O per node is increased. Hence we see the Sampling/SVM algorithm performing better than Basic/SVM algorithm for moderate to large skew.

2.5.4 Speedup and Scaleup

Any parallel algorithm, especially the load balancing ones, must be scalable as the effect of skew worsens with an increase in the number of processors. To show that our algorithms scale, we performed speedup and scaleup experiments at the moderate skew of $\theta = 0.5$ in both relations. The base case for the experiments is 8 processors as that is the minimum number of processors required for a single bucket join given our configurations. Using a smaller number of processors would require a significantly slower multi-bucket join making the comparison unfair. From

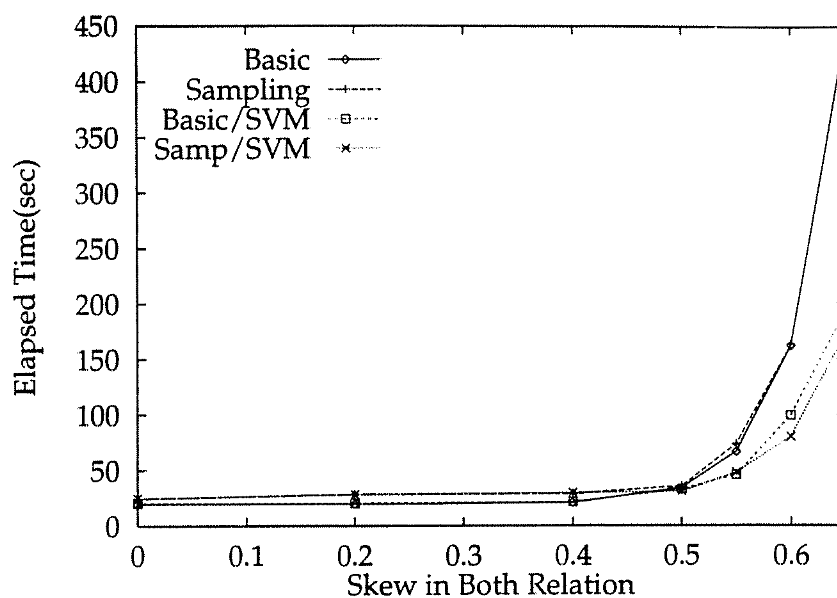

Figure 10: Multi-Bucket Join, Both Relations are Equally Skewed

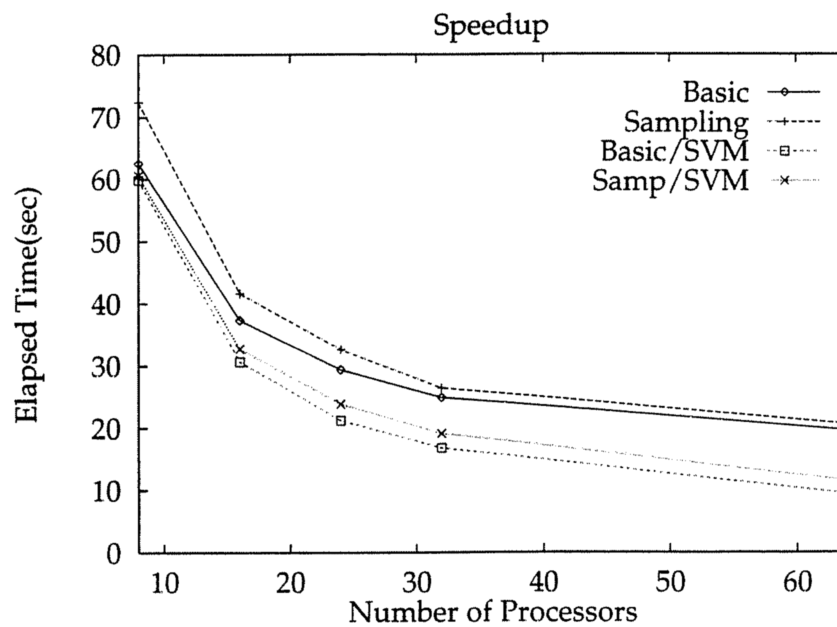

Figure 11: Speedup

Figure 11 it is evident that the SVM algorithms have better speedup characteristics than the original ones.

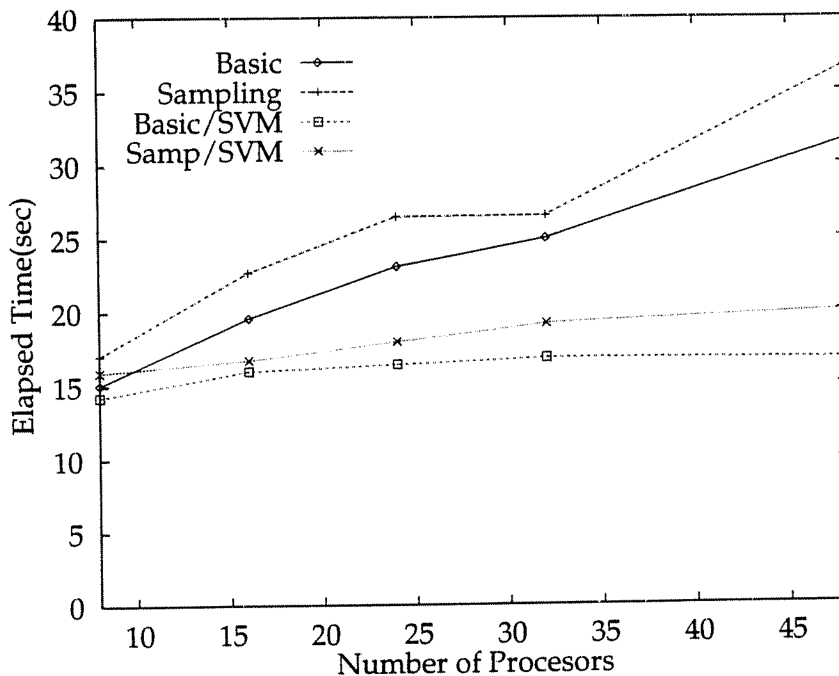

Figure 12: Scaleup

Again starting with the base of 8 nodes, for the scaleup experiment the relation size used is 31250 tuples per node. Figure 12 shows that although the SVM algorithms do not have ideal scaleup characteristics, they have better scaleup characteristics than the original algorithms without the SVM load balancing. One possible reason for nonideal scaleup is that given the same skewed data set, the amount of skew (ratio of most loaded to average) increases with the increase in number of processors. This is illustrated by the following example which divides 1000000 data values with identical domain size at $\theta = 0.9$, among one to 16 partitions, and plots the ratio of the most loaded partition to the average load in Figure 13.

2.6 Concluding Remarks

The algorithms presented above show a simple way of using SVM to improve the performance of join processing on existing parallel database systems in the presence of data skew. The changes required to existing parallel database systems built for shared-nothing multiprocessors are simple and localized. Thus we show that there are benefits to be gained from combining our views of the two architectures. We show a clean way of using both SVM and messages in

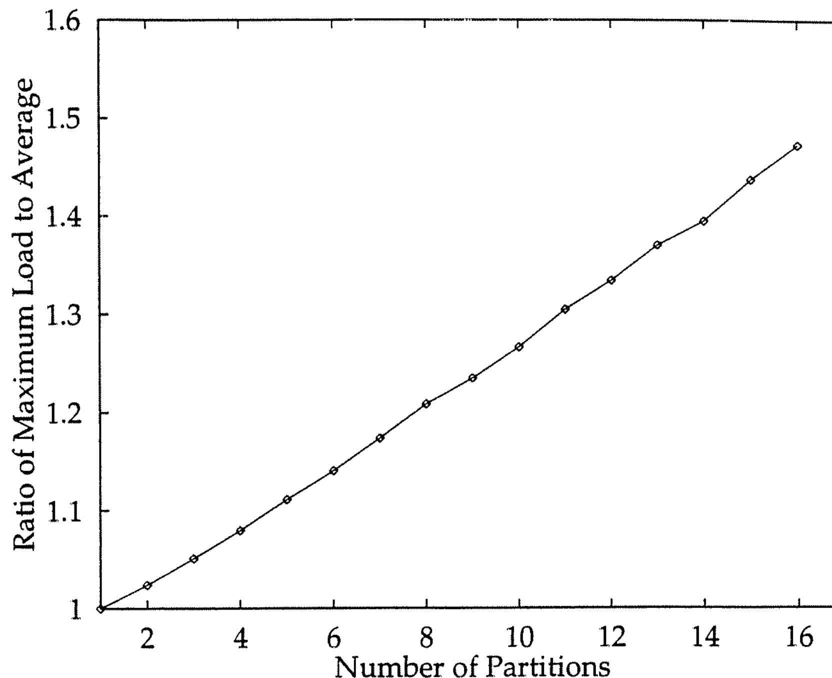

Figure 13: Ratio of Load of the Skewed Partition to the Average

a shared-nothing environment where the streams using messages are used for data flow, while shared data structures are maintained in shared virtual memory. The above seems to be a good way of getting the best of both programming paradigms, and may be useful in other parallel database applications.

The proposed algorithms achieve the superior performance when the data is skewed with little degradation of performance in the no skew case, a significant problem with most other load balancing algorithms. Unlike most previous skew handling algorithms, which require runtime statistics about their input relations making it hard to extend the algorithms to multi-way join case, the Basic/SVM algorithm supports multi-way joins as given. A notable exeception is the load balancing algorithm proposed by Poosala and Ioannidis [PI96] which uses precomputed histograms on the relations for load balancing.

One way to combine the sampling and non-sampling variants of the algorithm to get close to optimal performance is to use the Sampling/SVM algorithm when the skew is large and use the Basic/SVM algorithm otherwise. The decision of whether the skew is large or not can be based upon stored statistics for the relations or upon statistics computed by sampling. Thus we can avoid paying the overhead of sampling in the lower skew case but can still take advatange of sampling when the skew in the building relation is large.

Chapter 3

Hash Join on Shared Memory Hardware

While most scalable database systems are designed for a shared-nothing architecture, advances in hardware technology suggest that shared memory multiprocessors (SMPs) are becoming capable of handling all but the largest applications. Their increasing popularity raises two important questions.

1. How scalable database systems should be architected; in particular, do SMPs require specially targeted algorithms, or will algorithms developed for shared-nothing hardware suffice?

2. Can we design algorithms that exploit the SMP architecture suitably and outperform the traditional SMP algorithms?

As a first step towards answering the questions, in this chapter, we investigate the performance of hash join algorithms on SMPs. First, we show that the shared-nothing hash join algorithm, when ported to an SMP by using a transparent message passing library, performs a little worse compared to the its implementation using an optimized message passing library that requires minor changes in the code. Next we study the performance of a commonly proposed simple parallel join algorithm designed for shared memory and find that it performs only marginally better than the optimized shared-nothing approach. Hence, it seems that at least for the basic query evaluation algorithms, shared-nothing software can run efficiently on shared-memory hardware i.e. the two software architectures are not are as different as is commonly believed. This is further supported in the study of hash aggregation algorithms discussed in the next chapter.

Finally, we attempt to answer the second question above by showing that further performance improvements are possible over the simple shared-memory algorithm by employing

optimizations designed to maximize processor cache hits and minimize memory coherence traffic.

3.1 Introduction

There is an increasing interest in using shared-memory multiprocessors, henceforth called SMPs, for high performance database systems. Since the scale of SMPs lies between those of uniprocessors and the shared-nothing large scale parallel machines, it makes them suitable for all but the largest systems. To the limit of the scalability, they offer superior performance for the same dollar cost than other parallel architectures as marginal performance gain in terms of parallelism and throughput due to addition of a CPU tends to be slightly higher than the cost of the additional hardware. Also, the shared address space allows natural extension of uniprocessor algorithms into the parallel domain making the SMPs easier to use than other parallel platforms. Finally, even without the technical arguments for SMPs, the reality is that today most parallel machines sold are SMPs.

Today, however, most scalable parallel database systems are designed for the shared-nothing hardware paradigm. This includes Informix XPS [Ger95], IBM DB2/PE [BFG+95], Sybase Navigation Server [Syb], Tandem [Tan87], and Teradata [Ter83]. While one can certainly run a shared-nothing system on an SMP (using the shared memory as a fast communication network), SMPs offer alternatives to algorithm design not present in shared-nothing machines which include load balancing and easier management (as there are fewer independent nodes). The question is: should we redesign all the query evaluation algorithms specifically for the SMPs or will the shared-nothing algorithms suffice? In other words, do we need to maintain two software architectures, one for SMPs and the other for shared-nothing hardware? Also, clusters of SMPs seem promising as the basis for scalable parallel database systems. There the question is whether we need a two-level algorithm, shared-memory within a node and shared-nothing between nodes, or can we get away with one single shared-nothing algorithm? Moreover, since SMPs are getting powerful enough to handle all except the largest systems, another important question that arises is: portability considerations aside, how can we exploit the SMP architecture so as to get the best performance?

The answers to the above questions are not clear. On one hand, SMPs allow simple algorithms due to shared memory; they require none of the massive data redistribution that is

inherent in shared-nothing parallel join algorithms. This suggests that shared-memory algorithms should dominate shared-nothing algorithms on SMPs. On the other hand, naive shared-memory algorithms have synchronization overheads and poor cache behavior due to their lack of locality of memory reference. Shared-nothing algorithms have better locality of reference, which suggests that they could be faster than shared-memory algorithms. Hence it is not clear how big the difference in performance between shared-memory and shared-nothing algorithms on SMPs will be; it is not even clear which class of algorithm will dominate.

To begin to answer the questions, we studied one part of parallel DBMS performance: the memory resident portion of join evaluation on an SMP, with an implementation on the SGI PowerChallenge multiprocessor[1]. Since the goal of our study is to compare options for architecting the memory resident portion of the join processing, by not including I/O costs in our performance study, we are perhaps exaggerating the differences between the algorithms. That is, if we added I/O to all of the algorithms, the running time of each would shift upwards by the same amount, making the differences between the running times a smaller fraction of the total running time. The main conclusion of the study, detailed below, is that optimized shared-nothing algorithms and shared-memory algorithms have similar performance on SMPs; this conclusion would only be supported more strongly if we added in I/O costs. To put it another way, our study is biased to emphasize differences between algorithms. It was surprising to us to find that even given this bias the algorithms did not differ significantly except where memory copy overheads dominated.

We first studied the performance of the shared-nothing algorithms on an SMP. Initially we implemented and used a transparent shared-nothing-like message passing library for the shared-nothing algorithm. Then we optimized the message passing library by explicitly exploiting the shared memory so as to avoid any extra memory copies. This required breaking the encapsulation of the message passing library by making the algorithm do explicit message buffer allocation and deallocation (i.e. a restriction that a send-buffer could not be reused and that the receiver had to explicitly free the buffer after it was used). The performance of the algorithm was better with the optimized library as there were no extra memory copies.

Then we studied the performance of the traditional shared-memory hash join algorithm that extends from the uniprocessor hash join where all nodes build a shared hash table and

[1]The I/O costs were not considered as the machine had poor I/O support for a realistic study.

then each node probes its tuples in the shared hash table [LTS90]. The traditional algorithm implemented naively performed poorly. The cause of the poor performance was located in a variable which was possibly suffering from *false-sharing*, i.e. the variable was on a cache line which had other variable beings accessed by different processors. After padding the variable suffering from false-sharing to the cache-block size, the algorithm performs marginally better than the shared-nothing algorithm with optimized message passing. Next we attempted to optimize the shared-memory algorithm in terms of better memory contention and locality by using repartitioning in shared memory, instead of using messages. However, these algorithms performed only comparably to the traditional simple approach. Finally we exploited the fact that the repartitioning algorithm can be cache optimized as the join processing after repartitioning is local to a processor [SKN94]. The cache optimized algorithms performed a little faster than the traditional shared-memory algorithm.

In general, therefore, we found that the shared-nothing and the shared-memory algorithms are comparable in performance on an SMP. This is because despite our simplifications that exaggerated the differences between algorithms, there was not much difference in performance.

In related work, Lu et al. in [LTS90] did an analytical performance of algorithms in a shared-everything environment. However, they considered different basic algorithms comparing hybrid hash, hash loop, etc. The analytical model did not truly reflect an SMP environment. Furthermore, the considered data domain was too restricted which resulted in misleading conclusions. Our previous work on cache conscious algorithms [SKN94] showed that the cache misses on a uniprocessor machine are expensive as the data has to be fetched from (slow) main memory and therefore the algorithms must be designed taking the cache into account. An access to a shared variable is even more likely to result in a cache miss because of the maintenance of cache coherence. Consequently, cost of data sharing can not be ignored while designing efficient query processing algorithms for the SMPs. Chandra et al. in [CLR94] showed that for some scientific applications, optimized shared-memory and message passing programs perform comparably on comparable hardware.

3.2 The Experimental Setup

We compare the performance of the algorithms by implementing them on a SGI PowerChallenge SMP. The hardware configuration of the SMP is 16 MIPS R8000 processors, 4 GB of shared

address space and 4 MB off-chip private cache memory per processor. There is a 16KB data cache on-chip.

The relative performance of the algorithms is sensitive to the join selectivity, which affects how repeated-memory-access intensive the join is. Hence we varied the result size by systematically changing the join selectivity. We varied the result size from about 120,000 to about 31 million tuples. That is, a tuple of S, on average, matched a tuple of R about $\frac{1}{16}$ to 16 times. Other workloads studied did not give any further insight. The relation size for both relations was 200 MB (104 bytes/tuple). We used 12 processors in our experiments. The join attribute was 16 bytes long. The timing runs were obtained by running the algorithm five times and taking the average. Since the relations were memory-resident and the parallel threads were pre-forked, the start up overhead in the algorithms was insignificant. We also ran scaleup experiments using the workload in which one tuple of R matched one tuple of S on average, varying the number of processors from 1 to 12.

3.3 Parallel Hash Join Algorithms

The basis of all parallel hash join algorithms is the uniprocessor hash join. The basic uniprocessor in-memory hash join algorithm works as follows. Assume R and S are the two relations (or fragments of relations of a bigger join) being joined. Furthermore, the relations are now in the main memory after having been read from the disk or are in the memory of a main memory DBMS. In the hash join algorithm, first a hash table of the tuples of R is created by hashing them on the join attribute. Then the tuples of the relation S are probed in the hash table by hashing them on the join attribute and searching the attribute value in the hash table. The matching tuples are output.

First we studied the performance of the shared-nothing algorithm on the SMP to evaluate its viability.

3.3.1 The Shared Nothing Algorithm

Shared nothing algorithms are generally based on the assumption that the communication between processors is slow. A common, though not necessary, feature of a shared-nothing hardware is that the communication in the algorithms is through message passing. Two factors influence the performance of any message passing library: 1. the underlying communication

hardware and 2. how well the library is implemented. Though it is common to ignore the second effect, it is nevertheless important and much work has gone into making the libraries more efficient.

As evident from the SMP architecture discussed earlier, the communication between two processors in an SMP is roughly equivalent to doing main memory accesses exchanging data, instead of private cache accesses, accompanied with some cache coherence protocol overheads. This is generally much faster than speeds achievable in a traditional shared-nothing interconnection network. Since the hardware is fairly fast, the software cost of message passing becomes important and we find the same in our experiments detailed below.

A shared-nothing hash join algorithm, as in Gamma [DGS+90], first repartitions the relations by hashing the tuples on the join attribute, each hash value being assigned a specific node. Each node then joins the partitions of the relations locally as if it were the only node in the system. A property of the algorithm is that a tuple is moved only once across the nodes minimizing communication between nodes. Tuples are actually sent in large (4 KB in our case) batches, except the last one, to minimize the communication cost. The algorithm, in two threads running in parallel, is as follows.

```
/* node i, repartitioning thread, R_i is the partition of R on node i */
foreach t in R_i
  hash t to find the destination node p
  send t to node p

foreach t in S_i
  hash t to find the destination node p
  send t to node p


/* node i, join thread */
while (t in R not exhausted)
  receive t of R sent to me
  insert t in the hash table

while (t in S not exhausted)
  receive t of S sent to me
  probe t in the hash table
  if (match)
    generate output tuple
```

In shared-nothing architectures, the repartitioning involves sending the tuple over an interconnection network across the node boundaries using a message passing library. The question

as mentioned earlier is: how will the algorithm perform if the SMP provides a message passing library just like a shared-nothing architecture?

Since no standard implementation of message passing, like MPI [Mes94], was available on the machine, we implemented two different message passing libraries to evaluate the performance of the shared-nothing algorithm. In the first, sending and receiving a message involves the following steps:

1. copying the source buffer provided by send() to a library buffer which is augmented with some header information like sender id,
2. letting the receiving process know that it has a pending message by putting the message in a queue for that process,
3. upon a receive() copying the library buffer to the buffer supplied by the receive() function and returning some of the relevant header information like length of the message.

The library is quite transparent to the rest of the algorithm and is therefore similar to a message passing library on a shared-nothing system. However, it has two memory copy operations for each send/receive pair. The shared-nothing algorithm using the transparent library is henceforth called the Shared Nothing/Transparent algorithm.

In the second version we exploited the shared-memory and eliminated all extra memory copies. We call this version the Shared Nothing/Optimized algorithm. In this version, the sending and receiving of a message involves the following steps:

1. the sending process allocates a message buffer to be used in the message
2. the buffer supplied by the send() can't be reused by the sending process and is augmented with some header information like sender id,
3. the receiving process is notified of a pending message by placing the message in a queue for the process,
4. upon a receive(), a pointer to the buffer and some relevant information, like length of the message, is returned to the receiving process and the receiving process can use the buffer as its own.

5. when the received buffer is no longer needed, the receiving process explicitly frees the message buffer.

As is evident, there are no memory copies involved in the sending of a message. However, this approach is not transparent because the sending process cannot reuse a sent buffer as buffer address now "belongs" to the receiving process. This, of course, is not the case in a real message passing system where the address spaces are completely disjoint, but should reflect how efficient we can be if we want to directly use shared-nothing algorithms on the SMPs.

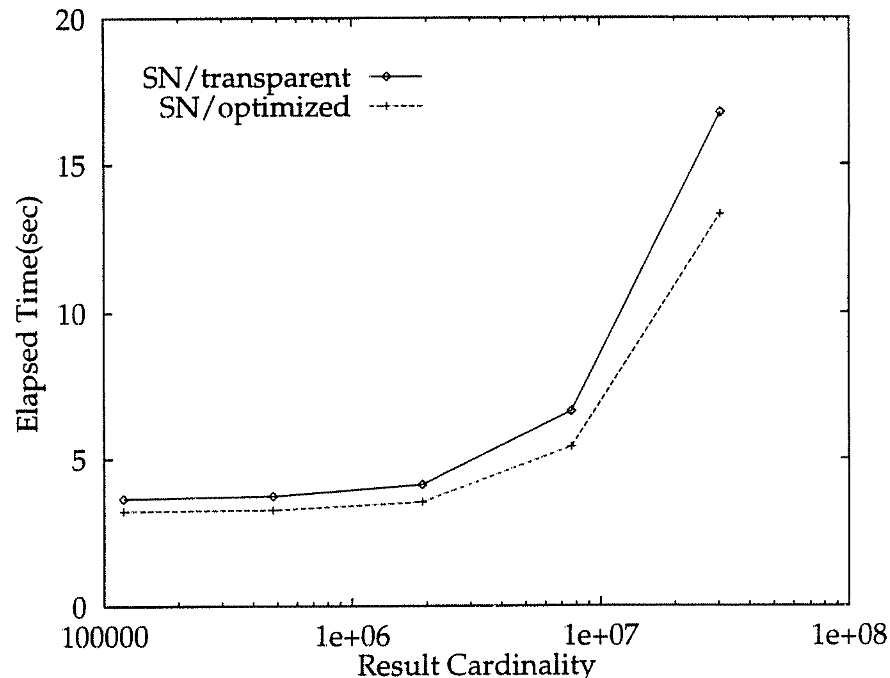

Figure 14: Performance of the Shared Nothing Approach

The performance of the shared-nothing algorithm with the two libraries is shown in Figure 14 for two 200 MB relations using 12 processors[2]. It shows that, as expected, the Shared Nothing/Optimized algorithm performs significantly better, being 13–26% faster, than the Shared Nothing/Transparent algorithm, because it does fewer memory copy operations. As shown later, the Shared Nothing/Optimized algorithm compares favorably with the other shared-memory algorithm showing that the shared-nothing approach is viable on an SMP.

In order to investigate the relative performance of the shared-nothing algorithm further, we

[2]All the experiments, in this and the later chapters, were run with the machine in stand-alone mode. We are omitting presenting the 95% confidence intervals because they were very small, typically less than 1% of the mean.

evaluated the performance of the simple shared-memory approach.

3.3.2 The Shared Memory Approach

The uniprocessor hash join algorithm is easily extended for SMPs as follows. All processors read their partition of the relation R and build the (global) hash table by hashing the tuples on the join attribute. The access conflicts to the hash table are taken care of by latching. Thereafter, all processors read their partition of relation S, probing the shared hash table. This approach is explicitly used in [LTS90] and is implicit in the XPRS database system [HS91]. This Shared Memory/Naive algorithm is detailed below. Under the constant time memory access (PRAM) model this algorithm is the most efficient as it does the least amount of "work" i.e. it has least number of memory accesses (and instructions).

```
/* node i, R_i is the partition of R on node i */
foreach t in R_i
  hash t to find the bucket b of the hash table
  latch bucket b to avoid conflicts
    insert t in the hash table bucket b
  unlatch bucket b

foreach t in S_i
  hash t to find the bucket b of the hash table
  probe t in the hash table bucket b
  if (match)
    generate output tuple
```

At the outset, this algorithm looks ideally parallel, except for the small latching overhead to handle access conflicts to the hash table in the build phase. However, a look inside the functioning of the algorithm on an SMP shows a few shortcomings.

1. Latching overhead may be non-trivial as latching can be expensive in SMPs.

2. Access to the shared hash table has extremely poor processor locality of access because any processor at random may access a bucket of the hash table.

3. A very small portion of the hash table can fit in the cache, effectively reducing the speed at which the hash table may be accessed.

These problems aside, as mentioned earlier, we discovered that the algorithm when implemented naively had a variable, used to temporarily store the join attribute, that suffered

from false-sharing signficantly affecting the performance of the algorithm. We eliminated the false-sharing by padding the variable to the cache line size. We call this the Shared Memory/Optimized algorithm.

As mentioned earlier, the Shared Memory/Naive algorithm is expected to perform the best. Optimizing it by eliminating false-sharing improves its performance signficantly as evident in Figure 15. We notice that the Shared Nothing/Optimized algorithm performs only a little worse, and in fact better for the last data point where the locality of accesses becomes more important, than the Shared Memory/Optimized algorithm thus showing that it is a viable alternative on an SMP.

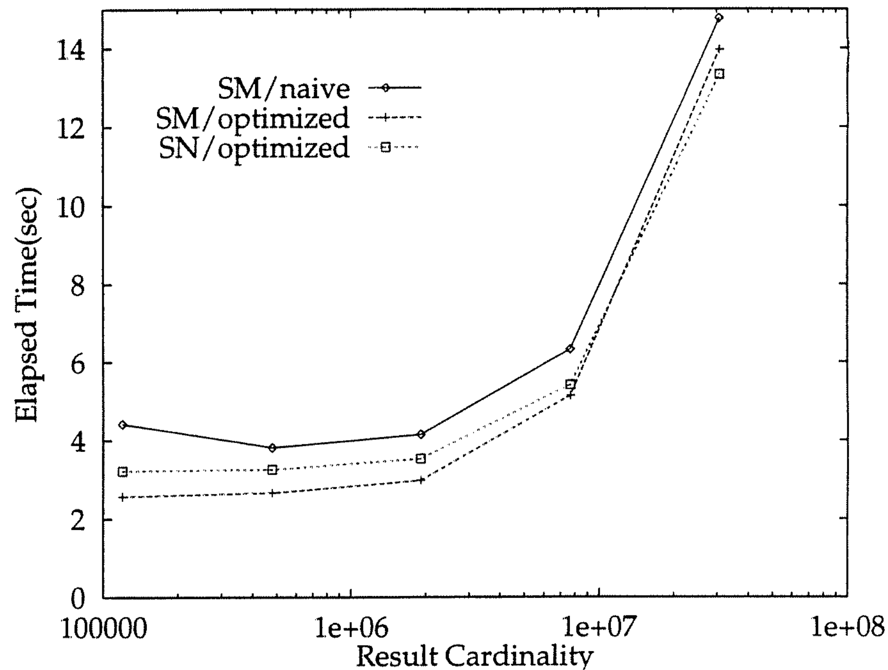

Figure 15: Performance of the Shared Memory Approach

However, the traditional approach is not the only way of performing a hash join on an SMP. In order to achieve superior performance, we attempted eliminating the shortcomings of the shared-memory approach. The first two shortcomings which are present in the shared-memory technique can be removed by borrowing some characteristics of the shared-nothing approach. This mainly includes repartitioning the relations using shared memory instead of sending messages before performing the local join.

3.3.3 The Hybrid Approach

The hybrid approach adapts the shared-nothing paradigm of repartitioning and then performing the local join on SMPs. The algorithms first repartition the relations by hashing on the join attribute. Then each processor performs a local join independently as in a shared-nothing algorithm as detailed in pseudo-code below.

```
    /* node i, R_i is the partition of R on node i */
    foreach t in R_i
      hash t to find the destination node p
1     repartition t by putting it in a "run" for node p

    foreach t in S_i
      hash t to find the destination node p
2     repartition t by putting it in a "run" for node p

    foreach t of R from each processor "run" for node i
      insert t in the hash table

    foreach t of S from each processor "run" for node i
      probe t in the hash table
      if (match)
        generate output tuple
```

Whereas in shared-nothing machines, the repartitioning involves sending the tuple itself to the buffer pool of the destination node, the shared memory permits different ways of repartitioning the relations. The details of the repartitioning, lines 1 and 2 above, differentiate the two algorithms.

1. Repartitioning the relation by copying the tuples to the local buffer pool of the destination node, henceforth called the Tuple-copy Hybrid algorithm. This is similar to doing message passing in terms of memory copy costs.

2. Leaving the tuples in the original (shared) buffer pool and creating partitions on the destination node by having pointers to the tuples in their original buffer pool, henceforth called the Pointer-based Hybrid algorithm.

The performance of the two variants in Figure 16 shows that the overhead of repartitioning is large and it eclipses the gain in the performance of the join due to removal of latching and improvement in processor locality. Hence the algorithms perform only comparably to the Shared Memory/Optimized algorithm. It also shows the following.

1. Though Pointer-based Hybrid algorithm has less partitioning overhead, it performs poorly because one has to go through the pointer to access the join attribute and also to access the tuple, both possibly off the local cache.

2. The Tuple-copy Hybrid algorithm performs poorly most of the time because of the overhead of tuple copying. However, when the result size is quite large, this overhead is somewhat compensated for by the increase in the locality of access.

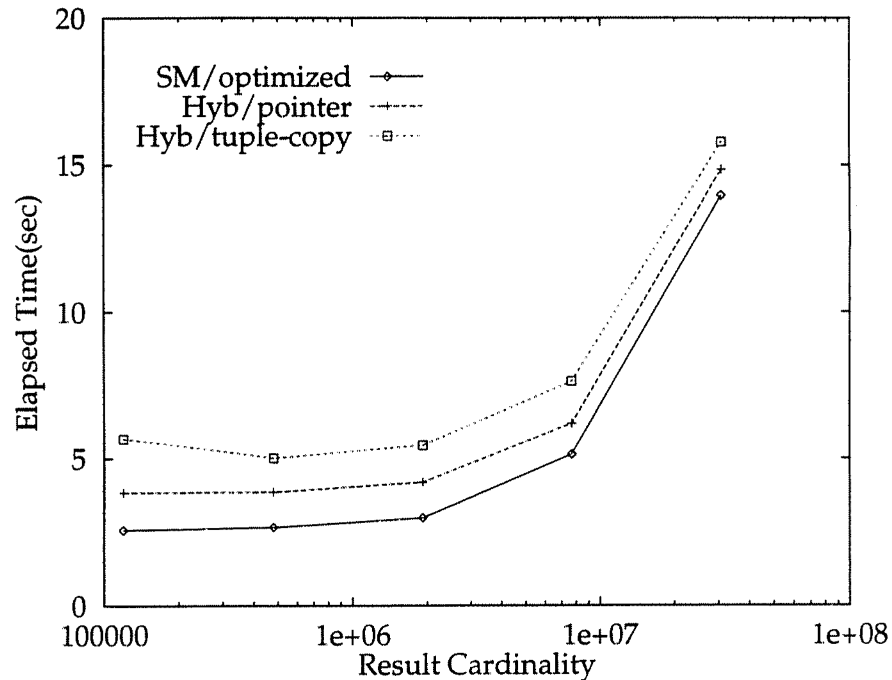

Figure 16: Performance of the Hybrid Approach

The main reason for the poor performance is that the algorithms still have very poor cache behavior though they have incurred the overhead of repartitioning. This is because even though the tuples are partitioned, the partitions are large enough that only a small portion of the hash table can fit in the cache, effectively reducing the speed at which the hash table may be accessed. Therefore, though the misses occurring in cache due to cache coherence are reduced by the repartitioning, the algorithm still suffers. Partitioning the data further into small cache-size partitions is likely to mitigate this shortcoming, as now the entire partitioned hash table is likely to fit in the cache while being accessed [SKN94]. Thus, the hash table is likely to be entirely cache resident while being accessed. Note that this optimization does not add any noticeable additional overhead over the hybrid approach but should improve the algorithm's

performance resulting in the following cache conscious algorithm.

3.3.4 Cache Conscious Hash Join

In this algorithm, instead of just hash repartitioning the relations on the join attribute across nodes, we further divide each of the partitions into cache size units by hashing on the join attribute. Each processor then repeatedly processes each of the corresponding cache size units of R and S. The partitioning of the relations is achieved by letting each processor partition its tuples and write them in a "run" for each cache size work unit. After the partitioning phase is over, a processor will have to read all the runs belonging to one cache size unit.

```
/* node i, R_i is the partition of R on node i */
foreach t in R_i
  hash t to find the node p, and cache-size partition r
  repartition t by putting it in a "run" for node p, partition r

foreach t in S_i
  hash t to find the node p, and bucket b of the hash table
  repartition t by putting it in a "run" for node p, partition r

foreach cache-size partition r on my node
  foreach t of R from each processor "run" for node i, partition r
    insert t in the hash table
  foreach t of S from each processor "run" for node i, partition r
    probe t in the hash table
    if (match)
      generate output tuple
```

We used the pointer-based approach for doing the repartitioning because as shown previously this has the best performance for the hybrid approach.

We expect this algorithm, called the Partitioned Cache Conscious algorithm, to do better because, first, it minimizes data sharing across the processors as partitioned tuples never need to be accessed by two processors and second, the partitioned hash tables are likely to be cache resident thus making the access to the hash table effectively much faster. If the overhead of partitioning is sufficiently small, the algorithm is expected to outperform the other algorithms.

In a variation of the algorithm, called the Queue-based Cache Conscious algorithm, detailed below, a partition of R and the corresponding partition of S are considered as a work unit. Instead of being preallocated to processors, as above, these partitions are placed in a work queue. Each processor, when idle, picks up the next unit from the queue and processes it. The

processing stops when the work queue gets empty. The work queue approach is possible due to the shared memory. This algorithm could outperform the first one if there is a small non-uniformity in the distribution of work load or if the system does not allow the processors to run as evenly as is theoretically possible because of OS functions or other system/user processes. In such cases the inherent load balancing of the algorithm, despite the small overhead of accessing the queue, is likely to result in superior performance.

```
/* node i, R_i is the partition of R on node i */
foreach t in R_i
  hash t to find the cache-size partition r
  repartition t by putting it in a "run" for cache partition r

foreach t in S_i
  hash t to find the node p, and bucket b of the hash table
  repartition t by putting it in a "run" for cache partition r

while there is a cache-size partition r remaining in the queue
  foreach t of R from each processor "run" for cache partition r
    insert t in the hash table
  foreach t of S from each processor "run" for cache partition r
    probe t in the hash table
    if (match)
      generate output tuple
```

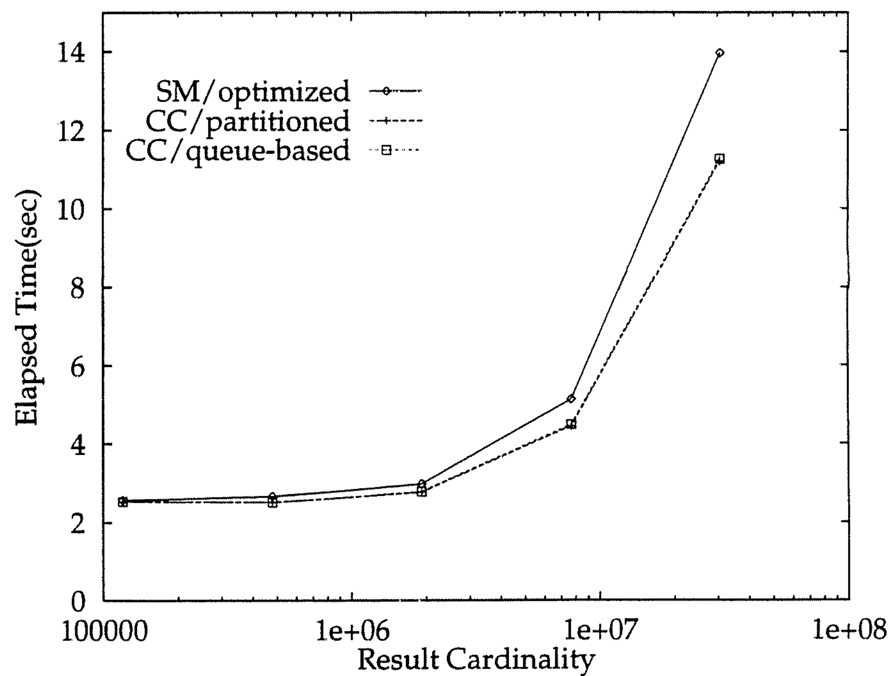

Figure 17: Performance of the Cache Conscious Algorithms

Figure 17 compares the performance of the two cache conscious approaches and the Shared Memory/Optimized algorithm. It shows that the both Queue-based Cache Conscious algorithm and the Partitioned Cache Conscious algorithm show similar performance. Both algorithms outperform the Shared Memory/Optimized algorithm by 1–24% in CPU costs (i.e. ignoring I/O). Note that the repeated-memory-accesses increase as the join selectivity increases because the same tuple will be accessed multiple times to build the result tuple. The cache optimizations, as expected, help significantly when there are lots of repeated memory accesses but not as much when a tuple is looked up only once or twice.

3.4 Performance Summary

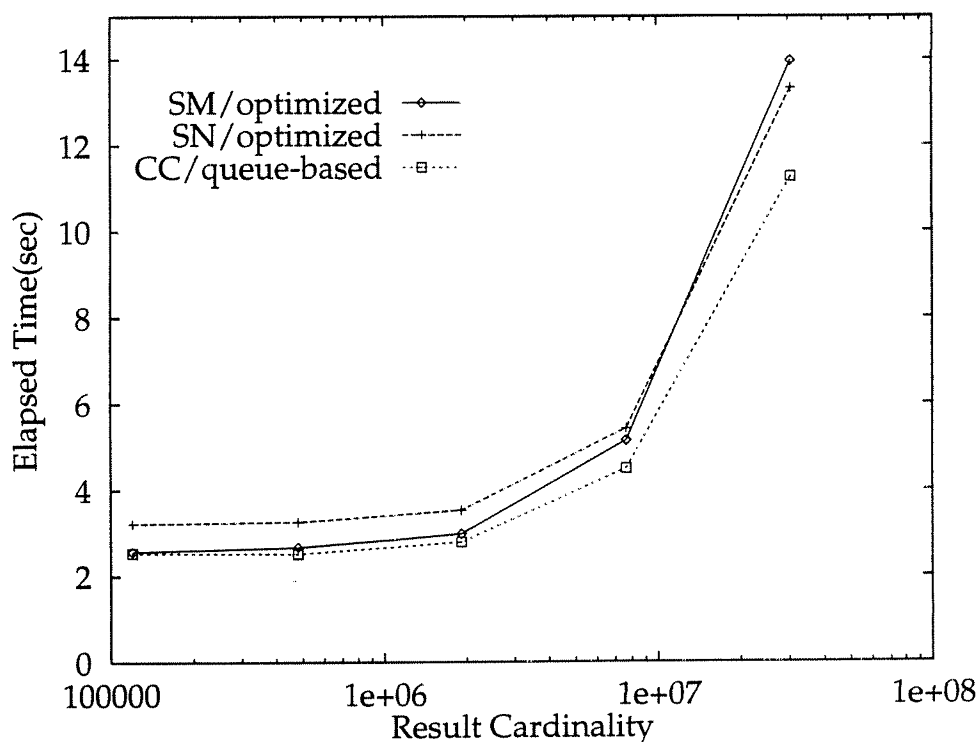

Figure 18: Relative Performance of the Algorithms

The above discussion is summarized in Figure 18. It compares the performance of the Shared Nothing/Optimized algorithm, the Shared Memory/Optimized algorithm and the Queue-based Cache Conscious algorithm. It shows that the Shared Nothing/Optimized algorithms performs comparably to the Shared Memory/Optimized algorithm. It also shows the performance gained via cache optimizations.

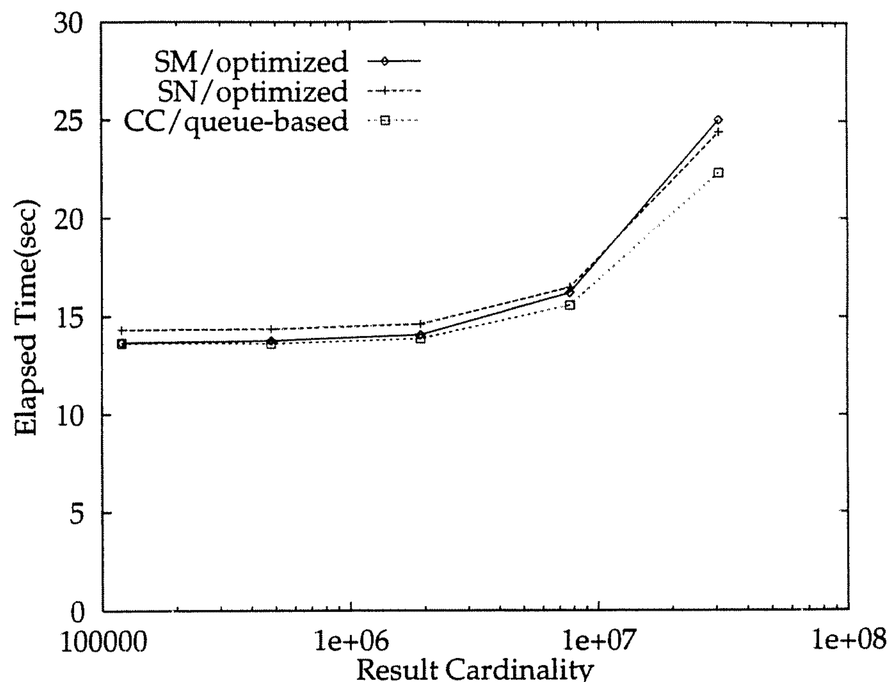

Figure 19: Performance of the Algorithms with Added Estimated I/O Costs

In order to further emphasize the relative performance of algorithms, in Figure 19 we show the performance with an estimated I/O cost (at 1 disk/CPU, 3 MB/sec transfer rate, it takes about 11 seconds to read the 400 MB of data) added to read in the relations.

From the figures it is evident that:

1. The Shared Nothing approach is viable on an SMP and it shows better performance with an optimized message passing library that minimizes memory copies.

2. The Shared Memory/Optimized algorithm performs marginally better for most workloads. However, when the workload is memory access intensive having a high join selectivity, the Shared Nothing/Optimized algorithm performs better because it has better memory access contention and locality.

3. Specializing algorithms for the SMPs architecture, exemplified by the Cache Conscious algorithms, does have performance dividends.

Figure 20 shows the scaleup characteristics of the Partitioned Cache Conscious, the Shared Nothing/Optimized and the Shared Memory/Optimized algorithms using the work load where one tuple of R matched one tuple of S on average. The relation sizes were about 16.7 MB

(104 byte/tuple) per processor. It shows that all three algorithms have reasonable scaleup performance. However, the Partitioned Cache Conscious algorithm shows superior scaleup performance because it is able to minimize the traffic on the system bus which is the main reason for the loss of performance when additional processors are used.

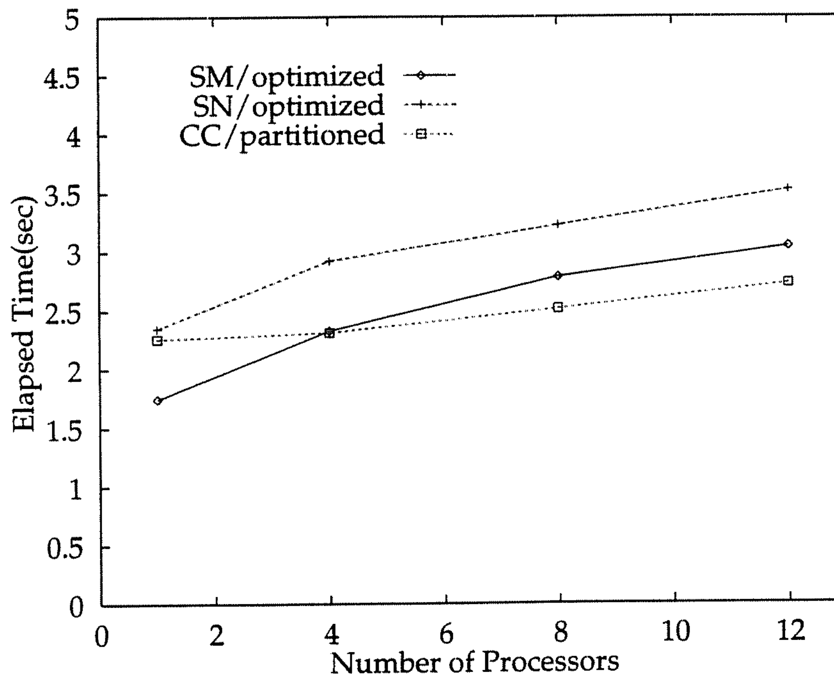

Figure 20: Scaleup Characteristics of the Algorithms

3.5 Concluding Remarks

In this chapter we attempted to answer two questions raised in the beginning: 1) Is shared-nothing software architecture viable on SMP hardware, and 2) How can we best exploit the SMP architecture?

Our study shows that running traditional shared-nothing hash join algorithm on SMPs performs acceptably well. It shows competitive performance when the message passing library is optimized to remove unnecessary memory copies. The shared-memory join algorithm perform marginally better than the shared-nothing algorithm. The performance of the shared-memory join algorithms can be enhanced by optimizing the algorithms by ensuring that the bulk of the join processing takes place in private cache memories of the processors avoiding the cache misses and the coherence traffic. This is achieved by partitioning the workload suitably. In the

next chapter we attempt to answer similar questions for the hash based aggregation algorithm.

One argument against shared-nothing algorithm has been their poor performance when the data is skewed [WDJ91a]. However, techniques that have proven effective for shared-nothing algorithms, e.g. [SN93], would trivially apply to SMPs.

Chapter 4

Hash Aggregation on Shared Nothing Hardware

The traditional parallel aggregation algorithms for shared-nothing architectures perform poorly when the number of groups is large, as expected in decision support and online analytical processing (OLAP) work loads. The not-so-obvious approach—because it uses a lot more communication—is to repartition the relation on the GROUP BY attributes first. However, though it works well when the number of groups is large, it performs poorly when the number of groups is small. This illustrates a possible tradeoff on shared-nothing systems, memory consumption (and hence I/O) vs. communication, that can be exploited. In this chapter, we propose new algorithms that exploit this tradeoff and dynamically adapt, at query evaluation time, in response to the observed number of groups in the relation being aggregated. This illustrates that a better understanding of parallel architectures can have significant performance dividends.

4.1 Introduction

The traditional algorithms for parallel aggregation and duplicate removal[1] are as follows. The standard parallel algorithm, henceforth called Centralized Two Phase, is for each node in the multiprocessor to first do aggregation on its local partition of the relation. Then these partial results are sent to a centralized coordinator node, which merges these partial results to produce the final result. The second approach, henceforth called the Two Phase, overcomes the potential uniprocessor bottleneck of the first by parallelizing the second phase of the Centralized Two Phase algorithm. Though through the years the development in interconnection network technology has resulted in high bandwidth networks, the communication cost is still a signicant component of the parallel query processing costs. Consequently, both these approaches are

[1]Duplicate elimination is algorithmically similar to aggregation except that the number of "groups" is relatively much larger in most cases.

motivated by the belief that communication should be minimized in a shared-nothing machine.

An approach that is a polar opposite of the one above is to first redistribute the relation on the GROUP BY attributes and then do aggregation on each of the nodes producing the final result in parallel, henceforth called Repartitioning. This algorithm outperforms the traditional approaches when the number of groups (i.e. the result size) is large as could be the case in duplicate elimination and several decision support and OLAP queries. The reason being that when the number of groups is large, this algorithm consumes less memory (as a group value is accumulated at just one node) and suffers less overhead (at the expense of more communication in the repartitioning) than the Two Phase algorithms. However, when the number of groups is small, the second step of the aggregation in the Two Phase algorithms has little overhead and hence they show superior performance. Thus, as shown later in detail, the algorithms only work well for a certain range of work loads (in terms of number of groups).

The above discussion brings out one form of the memory consumption (and hence I/O) and communication tradeoff often found in parallel database systems. In the remainder of the chapter, we propose hybrid algorithms that exploit this tradeoff and overcome the shortcomings of both algorithms. The proposed algorithms work well independent of the number of groups being computed. All of the algorithms change (decide) the method of computation depending on the workload, the number of groups in the relation and the amount of available memory, and are, therefore, able to adapt to different work loads. An interesting aspect of the non-sampling based adaptive algorithms is that each processor involved in the computation adapts based on what it observes, independently of what all the other processors are doing. This aspect of the algorithms allows them to proceed adaptively without any global synchronization.

There has been little work reported in the literature on aggregate processing. Epstein [Eps79] discusses some algorithms for computing scalar aggregates and aggregate functions on a uniprocessor. Bitton et al. [BBDW83] discuss two sorting based algorithms for aggregate processing on a shared-disk cache architecture. The first algorithm is somewhat similar to the Two Phase approach mentioned above in that it uses local aggregation. The second algorithm of Bitton et al. uses broadcast of the tuples and lets each node process the tuples belonging to a subset of groups. This is impractical on today's multiprocessor interconnects, which do not efficiently support broadcasting. Su et al. [SM82] discuss an implementation of the traditional approach. Graefe [Gra93] discusses one optimization for dealing with bucket overflow for the Two Phase

algorithm.

4.2 The Background

An SQL aggregate function is a function that operates on groups of tuples. Its basic form is:

select [group by attributes] aggregates **from** {relations}
[**where** {predicates}]
group by {attributes}
[**having** {predicates}]

We note that in practice the aggregate operation is often accompanied by the GROUP BY operation[2]. Thus the number of result tuples depends on the selectivity[3] of the GROUP BY attributes. We find that this selectivity does indeed vary quite a lot resulting in widely varying result sizes. For example, in the TPC-D benchmark we found the result size for aggregation varying from 2 tuples to as large as 0.28 million and 1.4 million tuples for a 100 GB database. In particular, in duplicate elimination the result sizes can be comparable to the input sizes. Hence a truly effective algorithm must cover the entire range of grouping selectivity: i.e. scalar aggregation to result sizes which could be almost identical to the input sizes.

A properly constructed HAVING clause, i.e. one that can't be converted to a WHERE clause, is evaluated after the processing of the GROUP BY clause and it does not directly affect the performance of the aggregation algorithms we are trying to study. Hence we will assume that the query does not have a HAVING clause. In the remainder of the chapter we will assume that aggregation is always accompanied with GROUP BY and that scalar aggregation can be considered as a special case where the number of groups is 1.

Further, we assume a Gamma [DGS+90] like architecture where each relational operation is represented by operators. The data "flows" through the operators in a pipelined fashion as far as possible. For example, a join of two base relations is implemented as two select operators followed by a join operator. Aggregation can be implemented by one or two operators, as needed, which are fed by some child operator, e.g. a select or a join, and the result is sent to some parent operator, e.g. a store. In our study we assume that the child operator is a scan/select and the parent operator is a store. However, with the exception of the sampling

[2] In the TPC-D benchmark 13 out of 15 queries with aggregates have GROUP BY.

[3] GROUP BY or grouping selectivity is the ratio of the result relation cardinality (i.e. number of groups) to input relation cardinality.

Sym.	Description	Values
N	number of processors	32
mips	MIPS of the processor	40
R	size of relation	800 MB
$\|R\|$	number of tuples in R	8 Million
$\|R_i\|$	number of R tuples on node i	$\|R\|/N$
P	page size	4 KB
IO	time to read a page (seq. IO)	1.15ms
rIO	time to read a random page	15.0ms
p	projectivity of aggregation	16%
t_r	time to read a tuple	300/mips
t_w	time to write a tuple	100/mips
t_h	time to compute hash value	400/mips
t_a	time to process a tuple	300/mips
S	GROUP BY selectivity	$\frac{1}{\|R\|}$ to 0.5
S_l	phase 1 selectivity in 2-Phase	max(S*N,1)
S_g	phase 2 selectivity in 2-Phase	max($\frac{1}{N}$,S)
t_d	time to compute destination	10/mips
m_p	message protocol cost/page	1000/mips
m_l	time to send a page	2.0 ms
M	default max. hash table size	10K entries

Table 3: Parameters for the Analytical Models

based algorithm, all of our algorithms extend naturally to the case where the child and parent are other operators.

We present analytical cost models of the basic approaches and the proposed algorithms as an aid to understanding the costs and tradeoffs in the algorithms. The cost models developed below are quite simple and as such they should not be interpreted to predict exact running times of the algorithms. The intention is that although the models will not be able to predict the actual running times, they will be good enough to predict the relative performance of the algorithms under varying circumstances. The simplifying assumptions in the model include no overlap between CPU, I/O and message passing, and that all nodes work completely in parallel thus allowing us to study the performance of just one node. Even this simple model generates results that are qualitatively in agreement with measurements from our implementation.

We assume that the aggregation is being performed directly on a base relation stored on disks as in the example query. The parameters of the study are listed in Table 3 unless otherwise specified. These parameters are similar to those in previous studies e.g. [BCL93]. The CPU

speed is chosen to reflect the characteristics of the current generation of commercially available microprocessors. The I/O rate was as observed on the SUN disk on the SUN SparcServer20/51. We model a high speed, high bandwidth network as in commercial multiprocessors like IBM SP-2. It is modeled only by the latency to send a message i.e. it has unlimited bandwidth. The software parameters are based on instruction counts taken from the Gamma parallel DBMS. The projectivity, p, is the part of a tuple relevant to aggregate computation. The grouping selectivity, S, is the ratio of result size to the input size and it varies from the result size of one (scalar aggregate) to the result size as large as half the input relation, as possible in duplicate elimination. In the following we assume that aggregation on a node is done by hashing.

The underlying uniprocessor hash based aggregation works roughly as follows.

1. The tuples of the relation are read and a hash table is built by hashing on the `GROUP BY` attributes of the tuple. The first tuple that hashes to a new value adds an entry to the hash table and the subsequent matches update the cumulative result as appropriate. It is easy to see that the memory requirement for the hash table is proportional to the number of distinct group values seen.

2. If the entire hash table is unable to fit in the allocated memory, the tuples are hash partitioned into multiple (as many as necessary to ensure no future memory overflow) buckets, and all but the first bucket are spooled to disk.

3. The overflow buckets are processed one by one as in step 1 above.

In the following sections we briefly describe the two traditional approaches to parallel aggregation. The first being the Centralized Two Phase algorithm [DGS+90].

4.2.1 Centralized Two Phase Algorithm

Considering the structure of the shared-nothing architecture, the most intuitive approach for parallel aggregation is for each node to do aggregation on the locally generated (or read) tuples in the first phase and then merge these local aggregate values at a central coordinator in the second phase. It is intuitive because it minimizes the communication required for the parallelization. Thus all nodes do as much processing locally as possible, i.e. without any communication over the interconnect. Intuitively, this will work well when the sequential merging step is small,

i.e. when the number of groups is small, but how the algorithm performs when the number of groups becomes large is not intuitive.

The analytical cost model of the algorithm is as follows. In the first phase each node processes the tuples residing locally.

- scan cost (IO): $(R_i/P) * IO$
- select cost, extracting tuples off the data page: $|R_i| * (t_r + t_w)$
- local aggregation involving reading, hashing and computing the cumulative value: $|R_i| * (t_r + t_h + t_a)$
- the tuples not processed in first pass need an extra read/write: $(1 - M/S_l) * p * R_i/P * 2 * IO$
- generate result tuples: $|R_i| * S_l * t_w$
- message cost for sending result to coordinator: $(p * R_i * S_l/P) * (m_p + m_l)$

In the second phase these local values are merged by the coordinator. The number of tuples that arrive at the coordinator are: $|G| = \sum |R_i| * S_l = |R| * S_l$ and $G = p * R * S_l$.

- receive tuples from local aggregation operators: $(G/P) * m_p$
- compute the final aggregate value for each group. This involves reading and computing the cumulative values: $|G| * (t_r + t_a)$
- the tuples not processed in first pass need an extra read/write: $(1 - M/S_g) * G/P * 2 * IO$
- generate final result: $|G| * S_g * t_w$
- I/O cost to store the result tuples: $(G * S_g/P) * IO$

As shown later, the single coordinator becomes a bottleneck when the number of groups is large, especially when the hash-table size becomes too large to fit in the allocated memory. The sequential bottleneck can be overcome by parallelizing the merging phase resulting in the Two Phase algorithm [Gra93]. This algorithm will still have the same communication requirements as the previous one but the messages go to different nodes (and not to the central coordinator).

4.2.2 Two Phase Aggregation

This algorithm is similar to the Centralized Two Phase algorithm described earlier, except that the merging phase is parallelized by hash-partitioning on the `GROUP BY` attribute. Each node aggregates the locally read (or generated) tuples. The local aggregate values are then hash-partitioned on the `GROUP BY` attributes and the nodes merge these local aggregate values in parallel. This is expected to be efficient even when the number of groups is large as the merging step will not be a bottleneck. The analytical cost model of the second phase of the algorithm becomes:

- receive tuples from local aggregation operators: $(G_i/P) * m_p$ where $G_i = p * R_i * S_l$ and $|G_i| = |R_i| * S_l$.
- compute the final aggregate value for each group arriving to this node: $|G_i| * (t_r + t_a)$
- generate result tuples: $|G_i| * S_g * t_w$
- the tuples not processed in first pass need an extra read/write: $(1 - M/S_l) * G_i/P * 2 * IO$
- I/O cost to store the result tuples: $(G_i * S_g/P) * IO$

However, even now there are two problems that get aggrevated when the number of groups is large. First, there is additional aggregation work in the merging phase which becomes pronounced as the number of groups increase. This is best understood by the following example.

Example 4.2.1 Assume a 4 node system aggregating 80 tuples (20 on each node) forming 4 groups. Each node will do 20 aggregate operations in first phase, and then 4 aggregate operations to merge its share of the local aggregate values, resulting in a total of 24 operations per node for processing the 80 tuples.

Now assume that number of groups being formed is 8. Then each node will do 20 aggregate operation in the first phase, and then 8 aggregate operations are required to merge its share of the local results in the second phase, resulting in a total of 28 aggregation operation per node.

Clearly, total number of operations now is more (28 operations vs. 24 earlier). □

Second, since the value of each group is being accumulated on potentially all the nodes, each of the nodes may have a group entry for every group. Thus the memory requirement per node can be same as in a uniprocessor system (and not decrease as more nodes are added).

Hence, despite the increased consumption of memory the potential for intermediate I/O due to hash table overflow does not decrease considerably. Considering the following simple example.

Example 4.2.2 Assume in the uniprocessor case the memory is sufficient to hold a quarter of the hash table. That means that about three-quarters of the relation can't be processed in the first pass and will have to be written out and read back in later.

Assume a 4 node system. In this algorithm, since all nodes could potentially be accumulating the value of each group, even allocating the same amout of memory as the uniprocessor per node (i.e. 4 times total memory as the uniprocessor), does not reduce the amount of intermediate I/O required i.e. even now three-quarters of the relation will be written out and read back in later.

However, note that a uniprocessor with 4 times as much memory could avoid the overflow processing entirely as the entire table could fit in memory. □

The third approach, counterintuitively, trades off network traffic in order to solve these two problems resulting in the following algorithm.

4.2.3 Repartitioning Algorithm

The Repartitioning algorithm first partitions the data on the `GROUP BY` attributes and then aggregates the partitions in parallel. It eliminates the overhead of the second phase as each tuple is processed for aggregation just once. Memory requirements are also reduced as each group value is stored in one place only. This is important because as we increase the number of nodes, the amount of memory required per node decreases, reducing the potential for intermediate I/O and improving performance.

The cost model of the algorithm is as follows.

- scan cost (IO): $(R_i/P) * IO$
- select cost involving reading, writing, hashing and finding the destination for the tuple: $|R_i| * (t_r + t_w + t_h + t_d)$
- repartition send/receive: $p * R_i/P * (m_p + m_l + m_p)$
- aggregate by reading and computing the cumulative sum: $|R_i| * (t_r + t_a)$
- the tuples not processed in first pass need an extra read/write: $(1 - M/S) * p * R_i/P * 2 * IO$

- generate result tuples: $|R_i| * S * t_r$
- I/O for storing result tuples: $(p * R_i * S_g/P) * IO$

However, if the number of groups is less than the number of processors then, even in the best case, not all processors can be utilized. That is $R_i = R * max(S, \frac{1}{N})$ in the best case. Another potential problem is that the network cost can overshadow the benefit gained but this is not likely to be a problem with current-day high-speed, high-bandwidth networks.

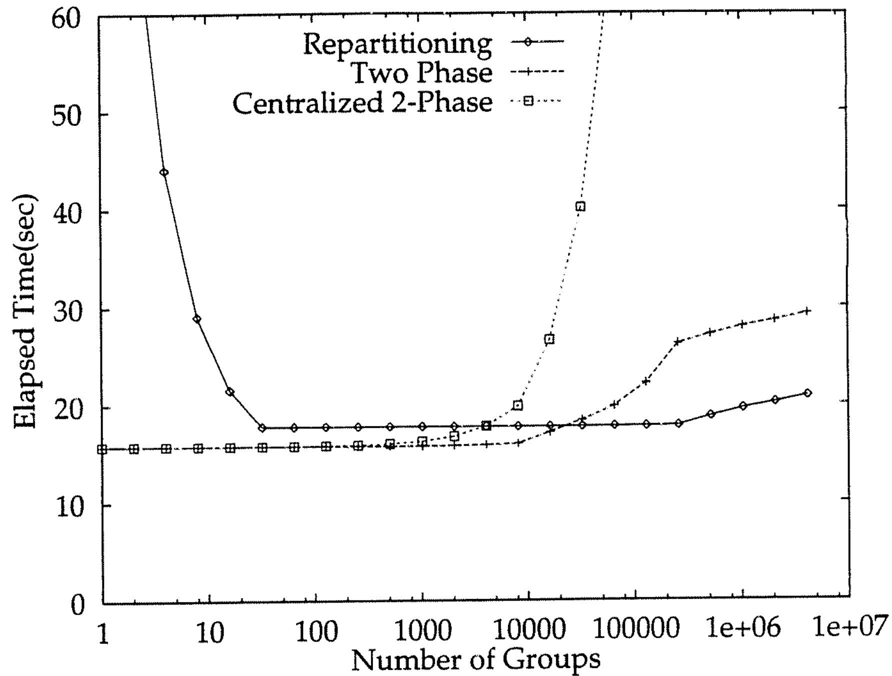

Figure 21: The Performance of Traditional Algorithms

The main performance characteristics of the Centralized Two Phase, the Two Phase and the Repartitioning algorithms are summarized in Figure 21 for the 32 processor, one disk/node, configuration aggregating 8 million (800 MB) tuples. The table size is restricted to 10000 groups, beyond which overflow processing is necessary. It shows that, as expected, the Two Phase algorithms do well when the number of groups is small but the Repartitioning performs better when the number of groups is large. It also shows that the network cost of repartitioning in the Repartitioning algorithms is not a serious problem if the interconnection has sufficient bandwidth (as in the IBM SP-2) but that wasted processors result in suboptimal performance when the number of groups is small. Finally, it is evident that each one of the algorithms performs poorly for some range of grouping selectivities and, therefore, none of these algorithms

by themselves will suffice for the entire range of grouping selectivities. To further motivate the need for including the Repartitioning algorithm, Figure 22 shows the performance of the algorithms with no I/O costs as would be the case when aggregation or duplicate elimination is performed in a pipeline of operators.

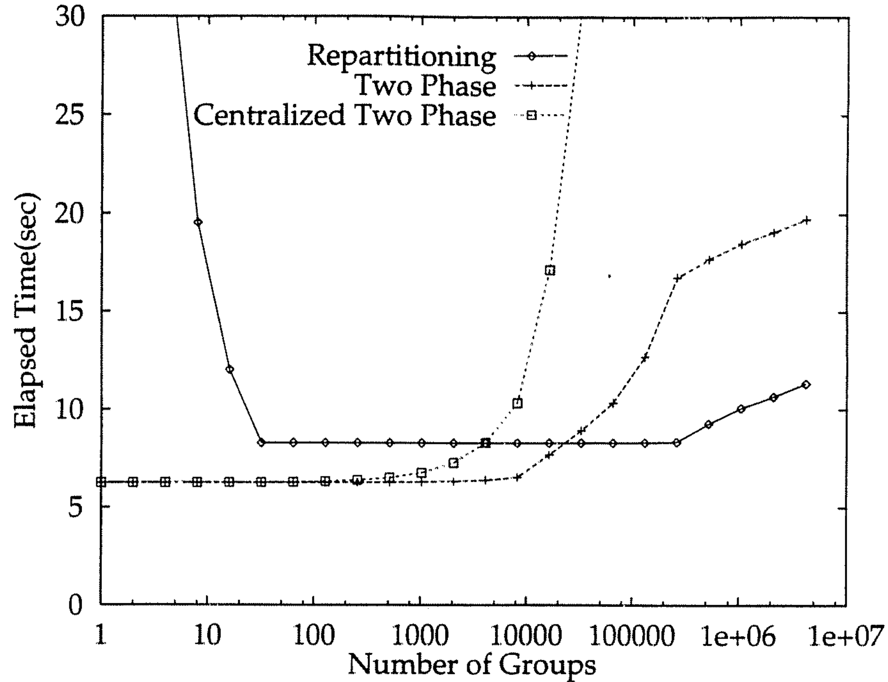

Figure 22: The Performance in an Operator Pipeline

4.3 The Algorithms

The natural question that arises from the previous discussion is: given that the Two Phase and the Repartitioning algorithms work well for different grouping selectivities, can we somehow merge the two to get the best of both? Fortunately, the answer is yes and the following sections present three approaches to it. The first algorithm, Sampling uses sampling to decide whether to use the Two Phase or the Repartitioning algorithm.

4.3.1 Sampling Based Approach

If we know the number of groups in the relation being aggregated then we can use the Two Phase algorithm if the number of groups is smaller than the amout of memory available, and the Repartitioning algorithm otherwise. Unfortunately, the number of groups is not usually known

beforehand. Sampling has been used effectively in past to estimate DBMS parameters [Ses92]. The general problem of accurately estimating the number of groups is similar to the projection estimation problem. However, in our case we only need to decide efficiently whether the number of groups in the relation is small or not because we have some leeway in the middle range where both algorithms are expected to perform well. The middle range, as expected, lies between the points where the repartitioning algorithm can exploit all the processors, and where the memory required for holding the hash table proves insufficient. This does not require an accurate estimate of the number of groups, especially when it is large, making the problem significantly simpler than the general estimation problem which is fairly complex [BF93]. The basic scheme is as follows. First the optimizer will decide what is an appropriate switching point of the algorithms depending on the system characteristics. A reasonable number of groups for switching would be some where in the middle range where both algorithms perform well. For example, 100 times the number of processors available on the small side, or on the large side, the number of groups such that the allocated memory would be just about sufficient. Call this the *crossover threshold.* Then, it can use the following algorithm to decide which scheme to use for aggregation.

```
sample the relation
find the number of groups in the sample
if (number of groups found < crossover threshold)
  use Two Phase
else
  use Repartitioning
```

The sampling can be implemented by letting each node randomly sample relation pages on its local disk. Page-oriented random sampling has been shown to be quite effective if there is no correlation between tuples in a page [Ses92]. Then the tuples in the sampled pages are aggregated using, possibly, the Centralized Two Phase algorithm. The number of groups obtained from the sample provides a lower bound on the number of groups in the relation. The trade-off here is between choosing a small crossover threshold and therefore a small overhead in sampling and decision making cost at the expense of using the Repartitioning algorithm when the number of groups is not too large. This trade-off is explored later.

It can be shown that the number of samples required is fairly small (about 10 times the crossover threshold) [ER61]. For example, for a crossover threshold of 320 this is approximately

2563. This is likely to be less than 1% of any reasonably sized relation for small crossover thresholds.

The cost of this algorithm depends upon which algorithm is actually chosen. The cost of sampling and estimation is approximately as follows.

- let s be the sample size per node in bytes and $|s|$ be the number of tuples in the sample
- scan cost (IO): $(s/P) * rIO$
- select cost, getting tuple off data page: $|s| * (t_r + t_w)$
- local aggregation involving reading, hashing and computing the cumulative value: $|s| * (t_r + t_h + t_a)$
- generate result tuples: $|s| * S_l * t_w$
- message cost for sending result to coordinator: $(p * s * S_l/P) * (m_p + m_l)$

In the second phase these local values are merged by the coordinator. The number of tuples arriving at the coordinator are: $|G| = \sum |s| * S_l = |s| * N * S_l$ and $G = p * s * S_l$.

- receive tuples from local aggregation operators: $(G/P) * m_p$
- compute the number of groups reading the tuples: $|G| * t_r$

However, since the sample is much smaller than the relation itself, the sampling cost is not significant.

4.3.2 Adaptive Two Phase Algorithm

The main idea in the Adaptive Two Phase algorithm is to start with the Two Phase algorithm under the common case assumption that the number of groups is small. However, if the algorithm detects that the number of groups is large it switches to the Repartitioning algorithm. The switching point can be determined as follows. The performance of the Two Phase algorithm worsens rapidly when it has to do intermediate I/O because of memory overflow. The crossover occurs when the intermediate I/O compensates for the additional network cost incurred in the Repartitioning algorithm. For high speed, high bandwidth networks the I/O cost will dominate. Hence, the number of groups at which the in-memory hash table in the

local aggregation phase of the Two Phase algorithm is full and intermediate I/O is required seems to be a good switching point. (This is also the case from an implementation point of view as we avoid overflow processing in the first phase.) As mentioned earlier, the Repartitioning algorithm has better memory utilization and hence will much less likely require intermediate I/O.

Switching from the Two Phase algorithm to the Repartitioning algorithm is done when a node detects that its hash table is full. It first partitions and sends the local results accumulated till that point to the (global) aggregation phase thus freeing the memory. Then it proceeds to read and partition the remaining tuples and sending them to the nodes that they hash to.

The second (global aggregation) phase now can receive two kinds of "tuples". It can receive locally aggregated values and it can receive "raw" (perhaps projected) tuples that partition to the node. Both kinds of tuples can be merged into the same hash table: the aggregated values will have to be processed as in the global aggregation phase (e.g. for SQL average, the sum and the count will have to be added to the currently accumulated value), and the raw tuples will be processed as usual (e.g. for SQL average, the value will be added to the sum and the count will be incremented by one). The hash table will have the final aggregated values for all group values that hash to that node.

Here we must mention that [Gra93] points out another optimization to the Two Phase algorithm. It suggests that in the local aggregation phase, if the hash table is full then the locally generated tuples are hash partitioned and forwarded to the local aggregation phase. Hopefully, there might already be an entry there for that group which will save on I/O costs. If not, then the overflow processing is done at the destination node. As is evident, this optimization will improve the performance of the Two Phase algorithm, provided disk I/O is slower than network, because it may reduce disk I/O. However, optimized Two Phase is inferior to the Adaptive Two Phase algorithm because:

1. There is a chance that there may be no group entry for a forwarded tuple on the destination node i.e. no saving on disk I/O and incurring an additional network cost.

2. All tuples still go through the local and global phases (despite the repartitioning) resulting in additional aggregation work.

3. Unlike the Adaptive Two Phase algorithm, which frees the memory used by the local

aggregation phase as soon as the memory overflow is detected, this optimization continues to use it as it maintains the local hash table until all tuples are exhausted.

The Adaptive Two Phase algorithm, in practice, can have complex behavior as at any given point in time one set of processors in the computation may be executing the Two Phase algorithms while others are executing the Repartitioning algorithm. In the simple scenario, where all nodes execute the same algorithm, the approximate cost model of the Adaptive Two Phase algorithm is as follows. The first M/S_l tuples are processed like the Two Phase algorithm and the remaining tuples, if any, are processed like the Repartitioning algorithm. The cost model can be derived from this to be as follows:

- let $|P_i| = \min(\frac{M}{S_l}, |R_i|)$
- $TwoPhase(P_i)$: cost of processing P_i tuples as in the Two Phase algorithm
- $Repart(R_i - P_i + S_l * P_i)$ where the selectivity of the Repartitioning algorithm is replaced by $\frac{S*|R|}{|R|-|P_i|+|P_i|*S_l}$, if $|R_i| > |P_i|$ as it sees both the original tuples and partially aggregated values.

4.3.3 Adaptive Repartitioning Algorithm

Another way to combine the Two Phase and the Repartitioning algorithm is to start with the Repartitioning algorithm and switch to Two Phase if the number of groups is not large enough to justify using the Repartitioning algorithm. The Adaptive Repartitioning algorithm is useful when the optimizer believes that the expected number of groups is large enough to use the Repartitioning algorithm. We expect that the algorithm will outperform the Adaptive Two Phase when the number of groups is very large as the first segment of tuples (the ones processed before switching) will not go through the additional phase of the Adaptive Two Phase algorithm.

However, in order to overcome the optimizer estimation errors, we must have a way switching to the Two Phase (actually Adaptive Two Phase) approach. The Adaptive Repartitioning algorithm does exactly that. It starts off with the Repartitioning approach. However, if it detects that the number of groups is very small after a certain number of tuples have been seen, it switches to the above mentioned Adaptive Two Phase algorithm. (Other methods for detecting potentially under-utilized processors can be used too.)

Switching between algorithms is performed as follows. When a node detects that it has seen too few groups given the number of tuples it has processed, it sends an "end-of-phase" message to all the nodes. Other nodes, upon receiving this message, follow suit by switching to the Adaptive Two Phase algorithm and sending their own "end-of-phase" message. All processes now perform the Adaptive Two Phase algorithm where the global aggregation phase uses the hash table left by the repartitioning phase.

An approximate and simplified cost model can be outlined as follows. Assume that first $initSeg$ tuples seen by the first Repartitioning phase are used to judge if it is alright to continue with the repartitioning or not. Therefore, the first $(initSeg * N)$ tuples are processed with the cost incurred in the repartitioning algorithm cost. The remaining tuples are processed either with the cost of repartitioning algorithm or with the cost of Adaptive Two Phase algorithm. The actual cost of switching itself is negligible as the end-of-phase message is piggy-backed on the tuples being forwarded. The cost model is detailed below (assuming it correctly decides if the switching takes place of not).

- if $(S * |R_i| > threshold)$ then cost is same as that of the Repartitioning algorithm. Otherwise...
- $Repart(initSeg)$: $initSeg$ tuples are processed as in the Repartitioning algorithm
- $AdaptiveTwoPhase(|R_i| - initSeg + initSeg * S)$ tuples are processed as in Adaptive Two Phase algorithm with the S_g of the second phase is replaced by $\frac{S_g*(|R_i|*S_l)}{N*S_g*(|R_i|*S_l)+(S_l*initSeg)}$ as it can see tuples generated in the first repartitioning phase.

4.4 Analytical Results

We studied the performance of the three algorithms, the Two Phase and the Repartitioning algorithm using the analytical models described before. The main performance characteristics of the algorithms are shown in Figure 23. It shows the performance characteristics for the configuration with 32 nodes with 1 disk/node and 8 million tuples (800 MB). The main observation is that all three algorithms are able to use the correct approach according to the workload demands. The Sampling algorithm has a constant (sampling) overhead for deciding which algorithm to use. The Adaptive Two Phase algorithm is able to track the appropriate algorithm with minimal overhead. This indicates that switching at the memory overflow point

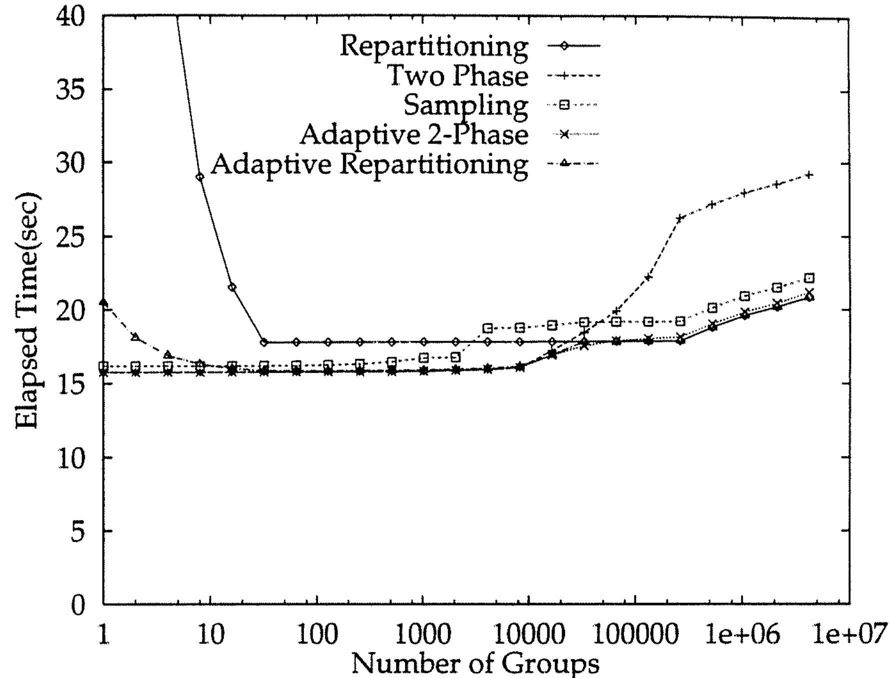

Figure 23: Relative Performance of the Approaches

is a reasonable decision. Finally the Adaptive Repartitioning algorithms has no overhead for its common case but does suffer a little when there are too few groups (as not all processors are used fully till the algorithm switches to Two Phase).

We also illustrate the expected scaleup of the algorithms under the low and high grouping selectivity in Figures 24 and 25 respectively. We are focusing on the extremes of the work load, where it is most important to choose the correct algorithm. The size of relation was 250 K tuples/node and we varied the number of nodes from 1 to 128.

The sampling overhead in the Sampling algorithm, as modeled and implemented, is proportional to the number of processors (i.e. it is a constant per processor). This is because the crossover threshold is proportional to the number of processors (as we want to ensure that the Repartitioning algorithm is not selected when it can't exploit all the processors). In our case, it is $100 * N$, where N is the number of processors. This overhead causes the Sampling algorithm to have small suboptimal scaleup in all cases as evident from Figures 24 and 25.

Both adaptive algorithms (Adaptive Two Phase and Adaptive Repartitioning) show almost ideal scaleup when the number of groups is small, 2 per node, (Figure 24). Adaptive Two Phase works well here because it sticks with the Two Phase algorithm, which has good performance in this range; Adaptive Repartitioning works well because it quickly switches from Repartitioning

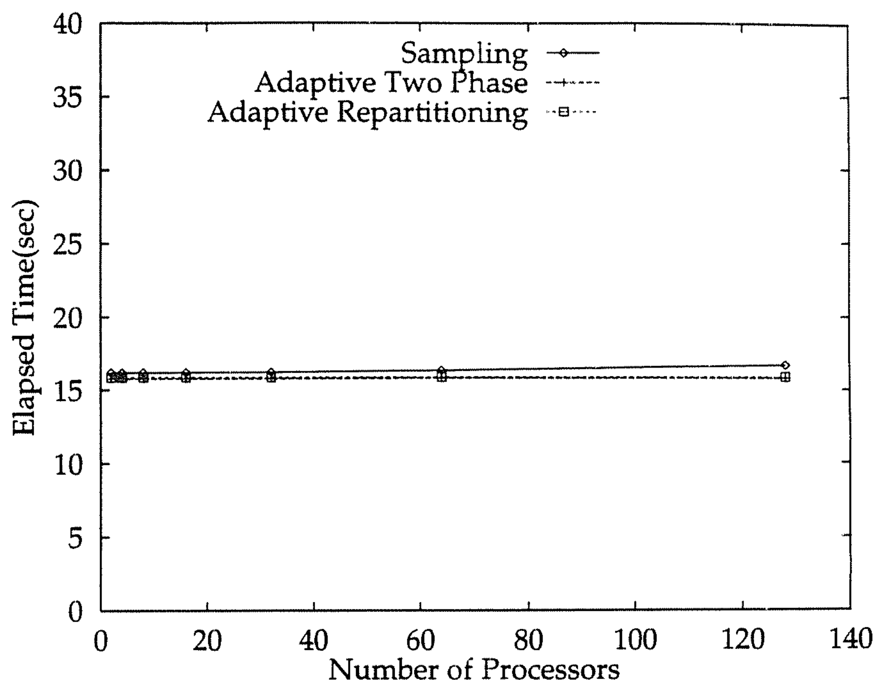

Figure 24: Scaleup of Algorithms: # Groups = 2 per node

to Two Phase when it discovers that there are few groups. When the number of groups is larger than than the memory available to hold the hash table, 125000 per node, (Figure 25), the performance of these algorithms is also good, because in this case Adaptive Two Phase switches to Repartitioning while Adaptive Repartitioning sticks with Repartitioning, which is the correct algorithm for this scenario.

Finally, we studied the trade-off between the sample size and the performance of the Sampling algorithm. Figure 26 clearly indicates the trade-off between sample size (and hence sampling cost) and the penalty of using the incorrect algorithm for a 32 processor configuration. Increasing the sample size makes it possible to determine the number of groups more accurately. Hence, one could have a larger crossover threshold at an increased sampling cost. In summary, for fast networks, one could use a small sample size as both the Two Phase and Repartitioning algorithms work efficiently and the sampling overhead can exceed the difference. However, if the network has low bandwidth, one could afford to increase the sample size to ensure that one doesn't use the Repartitioning algorithms when the number of groups in the relation is small.

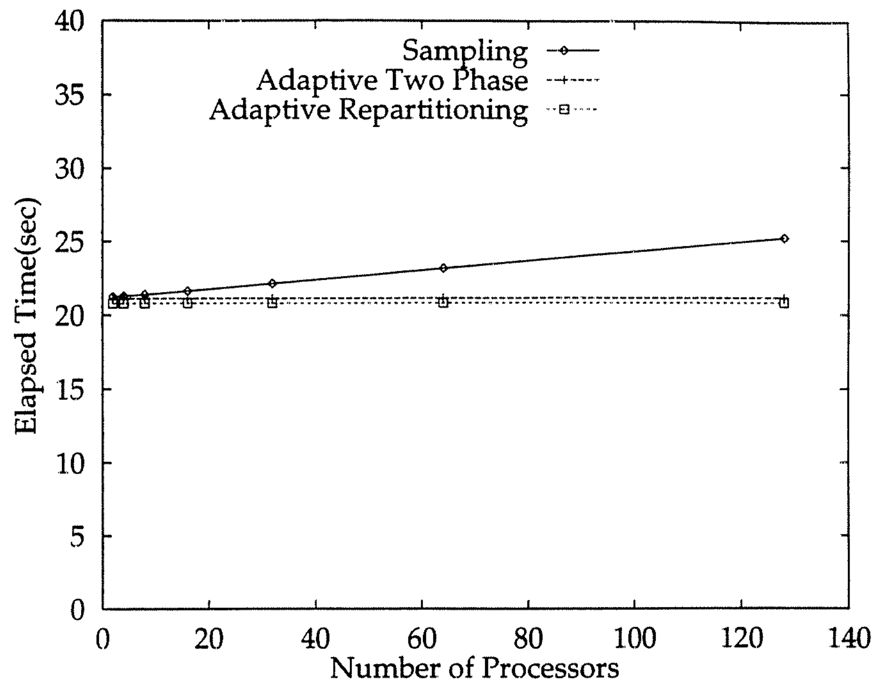

Figure 25: Scaleup of Algorithms: Number of Groups = 2 million

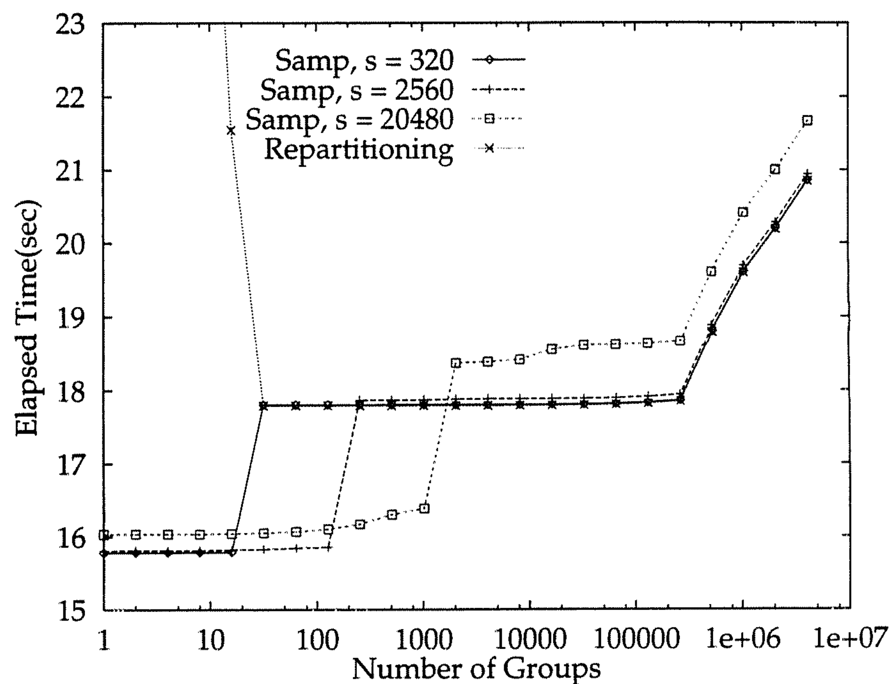

Figure 26: the sample size, performance trade-off

4.5 Implementation Results

In order to validate the general conclusions from the analytical model, and to show that the new algorithms are straightforward to implement, we implemented the algorithms on an IBM SP2 multicomputer using 8 nodes and one disk per node. The nodes are connected by a high-performance switch. We implemented the algorithms on top of the UNIX file system using the PVM parallel library [Oak93]. Our implementation had no concurrency control and did not use the buffer pool. Hence the algorithms are significantly more CPU efficient than what would be found in a complete database system.

The four million 104 byte tuple relation was partitioned in a round-robin fashion. Thus each node had 50 MB of relation. For efficiency reasons, we decided to "block" the messages into 4 KB pages. Memory allocated is sufficient hold a hash table for 10000 groups.

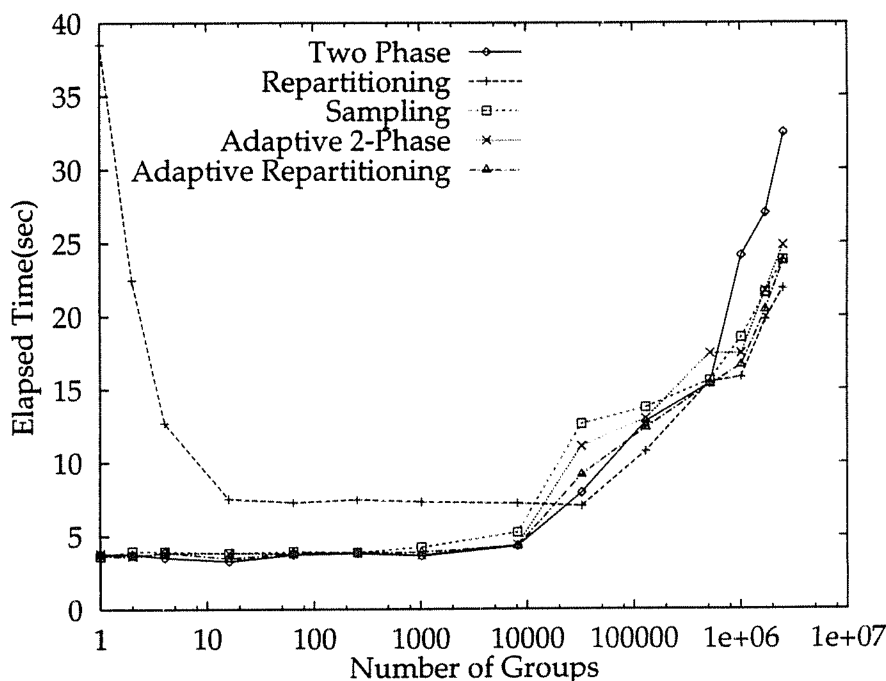

Figure 27: Relative Performance of the Approaches

The algorithms performed almost as expected from the analytical model. Figure 27 shows the performance of the Repartitioning, Two Phase, Sampling, Adaptive Two Phase and the Adaptive Repartitioning algorithms.

These results demonstrate that both the Adaptive Two Phase and Adaptive Repartitioning algorithms do well and are able to change the method of computation on demand. The switching

on memory full criterion for Adaptive Two Phase algorithm seems to perform well as it is able to switch gracefully from Two Phase to Repartitioning approach.

4.6 The Effect of Data Skew

All the performance measures reported so far assume that the attribute values are drawn from a uniform distribution. In this section we examine how data skew affects the performance of the algorithms. As both the input relation size and the number of groups (output size) affects the performance of the algorithms, there can be two kinds of data skew in aggregate processing.

1. The number of groups/node is the same but the number of tuples/node varies. We call this *input skew.*

2. The number of tuples/node is the same but the number of groups/node varies. We call this *output skew.*

This can be contrasted with data skew in the join algorithms, where placement skew is analogous to input skew and join product skew is analogous to output skew[WDJ91b]. However, since hash partitioning is not necessary for aggregate processing, the effect of data skew is significantly different.

The data skew, in essence, results in one or more nodes having to do more work than in the uniform distribution case. These skewed nodes then determine the overall performance of the algorithm. The expected effects of the different types of skew are as follows.

4.6.1 Input Data Skew

As mentioned, input data skew arises because of variability in the number of tuples each node initially has. Input skew mainly affects the input I/O cost if the relation is a base relation. The bulk of the processing is affected only slightly in the algorithms if the input relation is memory resident or is being generated on-the-fly because much of the cost of the aggregate processing depends on the number of groups which is unaffected.

There are two cases of input skew:

1. When the number of groups is small, all algorithms (except Repartitioning) will eventually use the Two Phase approach. The skewed node will have to process more tuples (and do

more I/O) severely affecting its performance. Repartitioning will be slightly less affected as it only has to redistribute the excess tuples, but the I/O cost to read the extra tuples for the skewed node is likely to dominate in case the aggregation is being performed on a disk-resident relation.

2. When there is a large number of groups, each of the algorithms except the Two Phase algorithm will use the Repartitioning approach (the Adaptive Repartitioning might switch to Two Phase before returning to Repartitioning). The skewed node will have to do more I/O, in case of a disk resident relation, and redistribute more tuples than others. The Two Phase algorithm will be affected even more as the skewed node has to process more tuples locally.

4.6.2 Output Data Skew

Output skew provides a more interesting scenario. The skewed nodes have a larger number of distinct group values than the rest of the nodes. There are two cases for the skewed nodes.

1. When the number of groups is small, all the algorithms (except Repartitioning) use the Two Phase approach. Since the total number of tuples is the same and there is no overflow processing, there is expected to be minimal performance degradation.

2. When the number of groups is large, the Adaptive Two Phase algorithm will change the skewed nodes to the Repartitioning approach, improving their performance. The other nodes will still use the Two Phase algorithm. Since the nodes with several groups do repartitioning, the amount of intermediate I/O will be significantly reduced, improving the overall performance of the algorithm.

   The Adaptive Repartitioning algorithm will first change to Adaptive Two Phase algorithm as it detects nodes with too few groups in them. Adaptive Two Phase will proceed as mentioned, resulting in similar performance.

   The Two Phase algorithm will suffer as the skewed node will have to do more I/O (than the uniformly distributed case). This is because the large number of groups on the skewed node may require more intermediate I/O for the processing.

Both the Sampling (which will choose Repartitioning) and Repartitioning algorithms will suffer more than Adaptive Two Phase but less than the Two Phase algorithm. Repartitioning will make all nodes do an equal amount of work because all the group values will be more uniformly distributed. However, the algorithms will be unable to save on messages which is possible in the Adaptive Two Phase approach.

This indicates that the Adaptive Two Phase (and Adaptive Repartitioning) algorithm can outperform all of the static traditional approaches in this scenario. In other words, the new algorithms can be better than the best of the Two Phase and Repartitioning algorithms because they allow nodes to make independent decisions about which algorithms to run.

We evaluated the performance of the algorithms under output skew. The skew is modeled by assigning four of the eight nodes all but four of the groups in the skewed relation. That is, four nodes have only one group value each, and the rest of the tuples are distributed among the remaining nodes. Figure 28 shows that both adaptive algorithms, Adaptive Two Phase and Adaptive Repartitioning, are able to do better than the traditional approaches when the number of groups is significantly larger than the memory allocated. Again, this is due to their ability to choose the correct approach for each node individually, something not possible with the traditional algorithms.

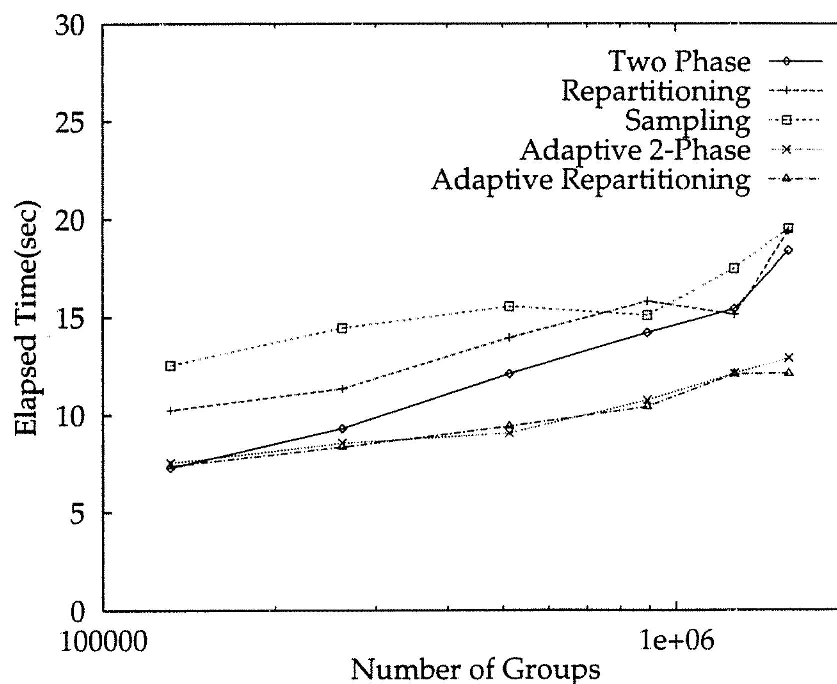

Figure 28: Performance under Output Skew

4.7 Concluding Remarks

The study of hash-based aggregation algorithm in this chapter shows the following.

1. Since communication is a significant component of the cost in shared-nothing query processing algorithms, the algorithm design is motivated to minimize it. Algorithms designed on this naive basis of the communication costs in a shared-nothing machine may perform poorly.

2. The communication minimizing Two Phase algorithms perform well when the number of groups is small enough so that the hash table fits in memory. However, since each group value is potentially accumulated at all the nodes, the amount of memory requirement per node does not decrease with a increase in number of processors. Hence the algorithm can be forced to do overflow processing, writing and later reading the part of the relation that couldn't be processed in the first pass.

   Since communication is significantly faster than disk I/O, the Repartitioning algorithm uses communication, in the form of repartitioning the relation, to reduce the memory usage (and consequently reduce the potential for and the amount of overflow processing required).

3. The Adaptive algorithms are successful at trading off communication and reduced memory usage (and hence disk I/O). They are able to cope efficiently with unknown and varying grouping selectivities. Hence we are able to overcome the shortcomings of the traditional approaches, which do not work well for all the grouping selectivities. These algorithms also show better performance under output data skew than the Two Phase and the Repartitioning algorithms.

Chapter 5

Hash Aggregation on Shared Memory Hardware

As mentioned earlier, shared-memory machines (SMPs) now seem to be the platform of choice for all but the largest database systems. In this chapter, we focus on the hash-based aggregation algorithm for SMPs because with the new emphasis on decision support queries and on-line analytical processing, the performance of `GROUP BY` aggregation operation is becoming very important. The focus is two-fold:

1. A study of shared-memory aggregation algorithms showing that, like the shared-nothing case where none of the traditional approaches perform well for the entire range from very small to extremely large numbers of groups, on shared-memory machines also the traditional approaches do not perform well for the entire range. A uniprocessor hash-based aggregation algorithm, when parallelized for an SMP, does not perform well when the number of groups is small. The Centralized Two Phase shared-nothing algorithm (see Section 4.2.1)), when adapted for shared memory, does not perform well when the number of groups is large. We propose two hash-based aggregation algorithms that perform well independent of the number of groups being aggregated.

2. A study of shared-nothing hash aggregation algorithms on SMP hardware and showing that they perform comparably to the shared-memory algorithms, further confirming our results from Chapter 3.

The performance of the algorithms is evaluated via an implementation on the SGI Challenge SMP.

5.1 Introduction

In Chapter 3 we discussed why shared-memory multiprocessors are becoming the platform of choice for all but the largest systems and potentially as nodes of larger scalable systems. This is, in part, because the shared address space allows the natural extension of uniprocessor algorithms into the parallel domain, which makes the SMPs easier to use than other parallel platforms.

Unfortunately, the natural extension of the uniprocessor hash-based aggregation algorithm, henceforth called Simple, performs poorly on a shared-memory machine when the number of groups in the aggregation is small. This is due to the large number of latch conflicts incurred while accessing the few entries in the hash table. The natural solution would be to use a shared-nothing style approach, henceforth called Two Step, in which the aggregation is performed in two steps: first, each processor performs aggregation on part of the relation "locally." In the second step the local results are merged to form the final aggregates. However, this algorithm performs poorly compared to the Simple algorithm when the number of groups is large because of the extra work required in the second step.

In this chapter, we will attempt to answer the following questions.

1. From the above discussion, it is evident that there is a tradeoff between incurring latch conflicts (in Simple algorithm) and the extra aggregation work (in Two Step algorithm). Can we exploit this tradeoff and have algorithms that perform well independent of the number of groups in the workload?

2. How do the basic shared-nothing algorithms using an efficient message passing library compare with the shared-memory algorithms?

We propose two algorithms that have none of the disadvantages of the classical solutions and perform well independent of the number of groups. This is achieved by seemlessly combining the two approaches in a way that the algorithm uses the correct approach depending on the observed work load. This is similar in spirit to the adaptive algorithms proposed in Chapter 4 for a shared-nothing architecture. Furthermore, we show that traditional shared-nothing algorithms perform comparably to the shared-memory algorithms confirming the results of Chapter 3 viz. the shared-nothing algorithms are viable on shared-memory hardware.

The performance of the algorithms is studied via an implementation on the SGI Challenge

SMP. The I/O cost of reading the relation was ignored as there was insufficient hardware support to do the parallel I/O. However, that should not affect our results significantly as the amount of I/O is the same for the algorithms unless the number of groups is large enough to require hash table overflow. From the discussion in Chapter 4 (see Section 4.2.2), it is clear that the algorithm requiring less memory will outperform the one requiring more memory when the number of groups is large enough to require hash table overflow. Since the Simple parallel algorithm requires the same amount of memory as the uniprocessor algorithm and the Two Step algorithm necessarily requires larger amount of aggregate memory, the Simple algorithm will outperform the Two Step algorithm when the number of groups is large.

To our knowledge, there has been no work reported in literature on aggregate processing on shared-memory systems. Other related work on aggregate processing is mentioned in Chapter 4. Chapter 4 also provides basic background on aggregate processing.

5.2 The Experimental Setup

We compare the performance of the classical and proposed algorithms by implementing them on a SGI Challenge SMP. The hardware configuration of the SMP is 12 MIPS R4400 processors, 1 GB of shared address space and 1 MB off-chip private cache memory per processor. There is a 16KB data cache on-chip.

Our workload consists of `GROUP BY` aggregation of a 200 MB relation (104 byte/tuple) using 8 processors. The `GROUP BY` attribute was 16 bytes long and the value being aggregated was an integer. The number of groups varied from 4 to 261987. The hash table size was chosen to ensure that no hash bucket chaining would be expected and was by default set to the size of the relation. The timing runs were obtained by running the algorithm seven times on the same input and taking the average after ignoring the best and worst runs. Since the relations were memory-resident and the parallel threads were pre-forked, the start up overhead in the algorithms was insignificant.

5.3 Classical Algorithms

The underlying uniprocessor hash based aggregation works roughly as follows. The tuples of the relation are read and a hash table is built by hashing on the `GROUP BY` attributes of

the tuple. The first tuple hashing to a new value adds an entry to the hash table and the subsequent matches update the cumulative result as appropriate. It is easy to see that the memory requirement for the hash table is proportional to the number of distinct group values seen.

If the entire hash table is too large to fit in the memory allocated, the tuples are hash partitioned into multiple (as many as necessary to ensure no future memory overflow) buckets, and all but the first bucket are spooled to disk. The overflow buckets are then processed one by one as in step 1 above.

In our study, since the Simple and the proposed algorithms consume the same amount of memory, the amount of I/O requirement for all of them would be the same. The Two Step algorithm necessarily requires a larger amount of memory for in-memory processing of the same work load. Hence, restricting the algorithms to the case where the hash table fits in memory does not affect the relative performance of the algorithms; it only makes the Two Step algorithm look better.

In the following sections we briefly describe the classical approaches to parallel aggregation on a shared-memory multiprocessor.

5.3.1 Simple Parallel Aggregation

The uniprocessor hash aggregation algorithm is easily extended for SMPs as follows. All processors read their partition of the relation R and build the (global) hash table by hashing the tuples on the `GROUP BY` attribute. Access conflicts to the hash table are taken care of by latching.

```
/* node i, R_i is the partition of R on node i */
foreach t in R_i
  hash t to find the bucket b of the global hash table
  latch bucket b to avoid conflicts
    if (group entry already exists)
      modify the cumulative value
    else
      insert a new entry in the global hash table bucket b
  unlatch bucket b

go through the hash table generating output tuples
```

The performance of the algorithm in Figure 29 shows that the algorithm performs well when the number of groups is large with the response time increasing with increasing result sizes.

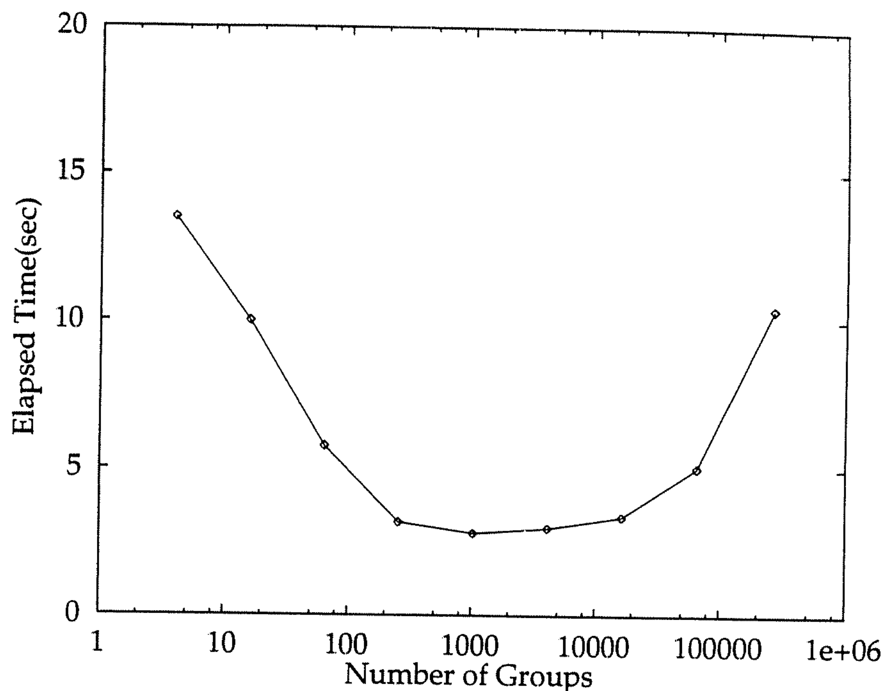

Figure 29: Performance of the Simple Algorithm

However, when the number of groups is small the algorithm performs rather poorly. This is due to the latch conflicts in accessing the same group entry (and hence the same hash bucket entry), and the resulting wait. When the number of groups is small resulting in a small number of hash bucket entries, this overhead becomes significant. The conflict eases when the number of groups is large as the probability of two processors trying to access the same group entry (and hence the same hash bucket) decreases. This is illustrated in Figure 30 which plots the average number of conflicts observed for different result sizes.

One way to minimize the number of conflicts in accessing the hash table is to let each processor process a part of the relation independently, as done in the shared-nothing style algorithms [DGS+90]. However, that requires an additional merge phase as detailed below.

5.3.2 Shared Nothing Two Phase Approach on SMP

The traditional shared-nothing Centralized Two Phase algorithm(see Section 4.2.1) works as follows. In the first step each node does aggregation on the locally generated tuples (or the local partition of the relation). Then these partial results are sent to a coordinator node which merges these partial results to produce the final result.

This algorithm can be adapted for the SMP platform by changing the second step. In the

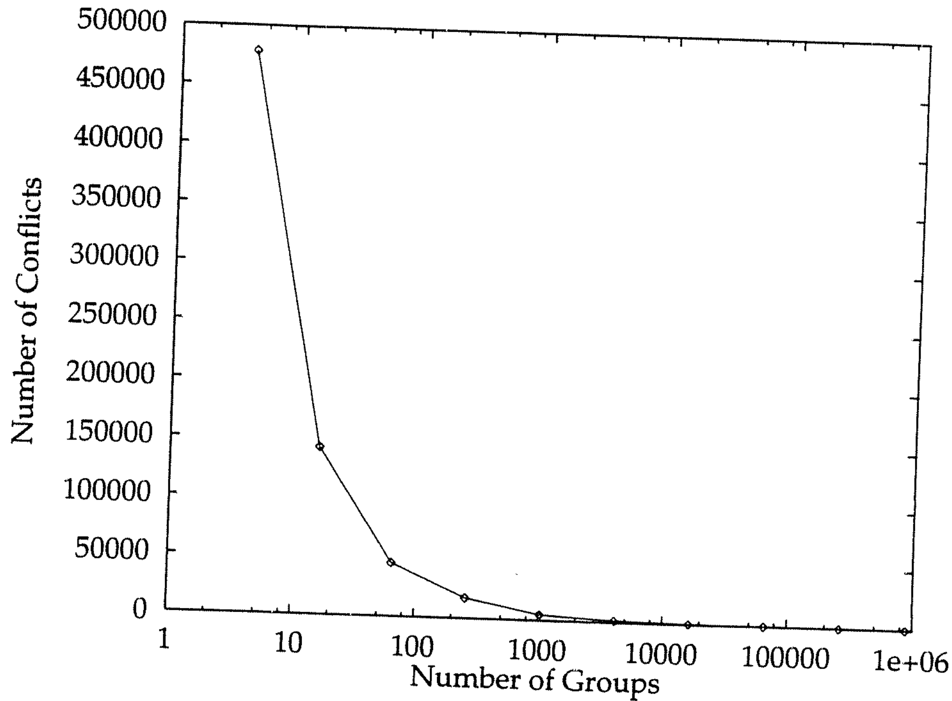

Figure 30: Number of Latch Conflicts in the Simple Algorithm

second step, all processors can combine their local results by building a shared global hash table. However, in this step there are expected to be fewer conflicts as each processor "inserts" a particular group value only once as all local tuples with that group value have already been aggregated.

```
/* node i, R_i is the partition of R on node i */
foreach t in R_i
  hash t to find the bucket b of the local hash table
  if (group entry already exists)
    modify the cumulative value
  else
    insert a new entry in the local hash table bucket b

  foreach entry e with hash bucket b in local hash table
    latch bucket b to avoid conflicts
    if (group entry already exists)
      modify the cumulative value
    else
      insert a new entry in the global hash table bucket b
    unlatch bucket b

  go through the global hash table generating output tuples
```

The performance of the Two Step algorithm is shown in Figure 31. It shows that algorithm successfully overcomes the conflict problem in the Simple algorithm when the number of groups

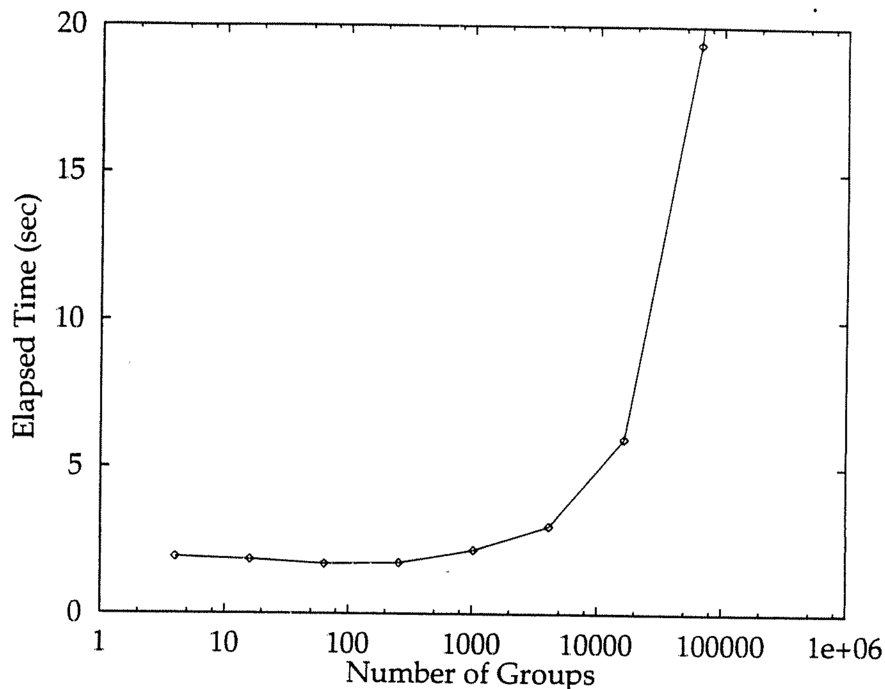

Figure 31: Performance of Two Step Algorithm

is small. However, when the number of groups is large, the additional second step becomes an overhead that is avoided in the Simple algorithm. Hence the performance of the algorithm degrades as the number of groups in the relation increases.

5.4 The Proposed Algorithms

From the above discussion, it should be clear that none of the classical approaches to hash based `GROUP BY` aggregation are satisfactory. What we need is an algorithm that will avoid the conflicts when the number of groups is small but not suffer the overhead in the Two Step algorithm when the number of groups is large. The following algorithms describe two ways of achieving this goal. Both algorithms are based on the idea that when the number of groups is small leading to a high latching conflict rate, one should use the conflict minimizing Two Step algorithm, and when the number of groups is large the Simple algorithm because it avoids the unnecessary second step. Hence, both algorithms decide what to do on the understanding that at some point (in terms of number of groups), one of the two approaches, Simple or Two Step, is better. This switching point will vary from machine to machine depending on how expensive is the latching and waiting for the latches as opposed to the cost of the rest of the

operations involved in aggregation (e.g. hashing, inserting in the hash table, etc.). However, on one machine and one implementation of aggregation operation, the switching point is fairly constant for a given number of processors which determines how many conflicting accesses can happen at any time. In general, the switching point can be estimated empirically by comparing the performance of the two algorithms and determining where the performance curves intersect. In our setup, this switching point was somewhere around one to two thousand groups.

5.4.1 The Adaptive Two Step Algorithm

The first algorithm we propose, called the Adaptive Two Step Algorithm, works as follows. First the algorithm does what essentially amounts to doing the Two Step algorithm. However, it monitors the number of group entries being found. At the point that it finds that the number of groups is larger than the predetermined switching point, it skips the first step for the remaining tuples and inserts them directly in global hash table (already needed for the second step of the Two Step algorithm). Thus it does the remainder of the processing using the Simple algorithm. The partial results that are already in the local hash table in the first step are also merged with the global hash table in a separate step. Hence, the algorithm does the processing exactly like the Two Step algorithm except when the number of groups is large.

```
/* node i, R_i is the partition of R on node i */
foreach t in R_i while number of groups seen is small
  hash t to find the bucket b of the local hash table
  if (group entry already exists)
    modify the cumulative value
  else
    insert a new entry in the local hash table bucket b, increment # group values seen

/* first merge the current local hash tables in the global hash table */
foreach entry e (hash bucket b) in the local hash table
  latch bucket b of the global hash table to avoid conflicts
  if (group entry already exists)
    modify the cumulative value
  else
    insert a new entry in the global hash table bucket b
  unlatch bucket b of the global hash table
```

```
/* then do the Simple algorithm for the remaining tuples, if any */
foreach remaining t in R_i
  hash t to find the bucket b of the global hash table
  latch bucket b to avoid conflicts
  if (group entry already exists)
    modify the cumulative value
  else
    insert a new entry in the global hash table bucket b
  unlatch bucket b

go through the global hash table generating output tuples
```

5.4.2 The Adaptive Simple Algorithm

The second algorithm, called Adaptive Simple Algorithm, works as follows. Initially the algorithm processes the tuples like the Simple algorithm building a global hash table. However, it monitors the number of group entries being found. If, given the number of tuples seen it finds that the number of groups is fairly small, it stops building the global hash table. Instead, it switches to the Two Step algorithm retaining the currently build global hash table for the second merge step of the Two Step algorithm. Thus, if the number of groups is actually large the algorithm proceeds exactly like the Simple algorithm, otherwise it does the bulk of the processing—after it detects that the number of groups being formed is too small given the number of tuples seen—using the Two Step algorithm.

```
/* node i, R_i is the partition of R on node i */
foreach t in R_i provided we haven't switched to Two Step
  hash t to find the bucket b of the global hash table
  latch bucket b to avoid conflicts
  if (group entry already exists)
    modify the cumulative value
  else
    insert a new entry in the global hash table bucket b, increment # group values seen
  unlatch bucket b
  switch if number of groups for tuples processed is too small

/* retain the global hash table for the second step of the Two Step algorithm */
if (switched)
  /* do the Two Step algorithm for the remaining tuples, if any */
  foreach remaining t in R_i
    hash t to find the bucket b of the local hash table
    if (group entry already exists)
      modify the cumulative value
    else
      insert a new entry in the local hash table bucket b
```

```
foreach entry e (hash bucket b) in local hash table
  latch bucket b of the global hash table to avoid conflicts
  if (group entry already exists)
    modify the cumulative value
  else
    insert a new entry in the global hash table bucket b
  unlatch bucket b

go through the global hash table generating output tuples
```

5.4.3 Performance Evaluation

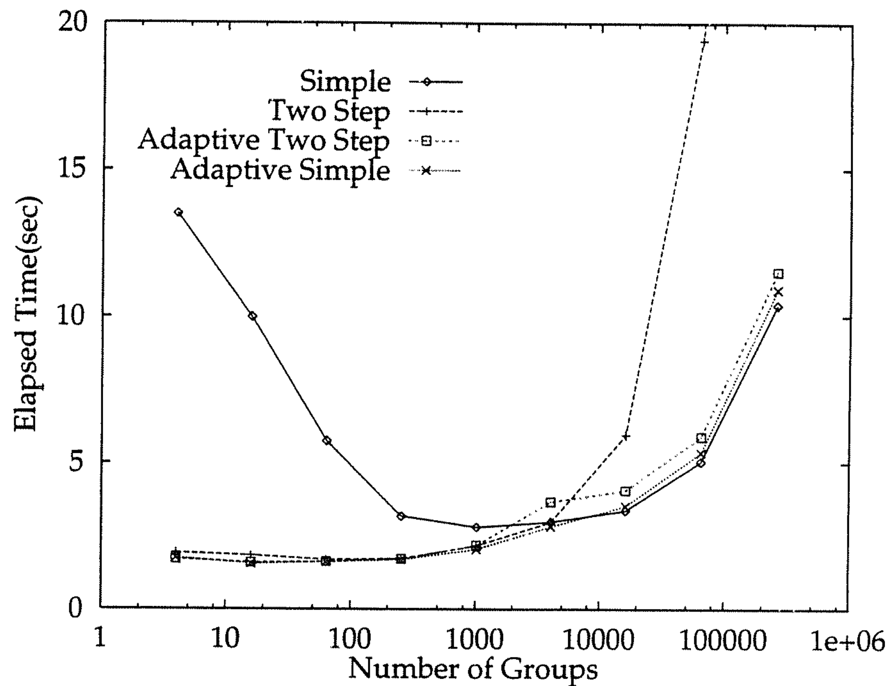

Figure 32: Performance of the Adaptive Algorithms

The performance of the two Adaptive algorithms descibed above is shown in Figure 32. It shows that both algorithms successfully overcome both the conflict problem in the Simple algorithm when the number of groups is small, and the overhead problem of the Two Step algorithm when the number of groups is large, by choosing the correct approach depending on the observed workload. The Adaptive Two Step algorithm performs a little worse when the number of groups is large because it starts with the incorrect approach and has to process a significant number of tuples, using the inappropriate Two Step approach, before switching. The

Adaptive Simple algorithm, on the other hand, can quickly decide if it is using the incorrect approach and doesn't suffer much even when it has to switch to the Two Step approach.

Scaleup Characteristics

We also studied the scaleup characteristics of both algorithms. Figure 33 shows the scaleup of the algorithms when the number of groups is small, 2 per processor. It shows that both of the algorithms scale fairly well when the number of groups is small. Figure 34 shows the scaleup of the algorithms when the number of groups is large, 2048 per processor. The scaleup characteristics of both the algorithms are comparable, though not ideal. Both Adaptive algorithms use the Simple algorithm in this case. The algorithm is expected perform better with fewer nodes as it suffers less conflicts (due to fewer processors competing for latches).

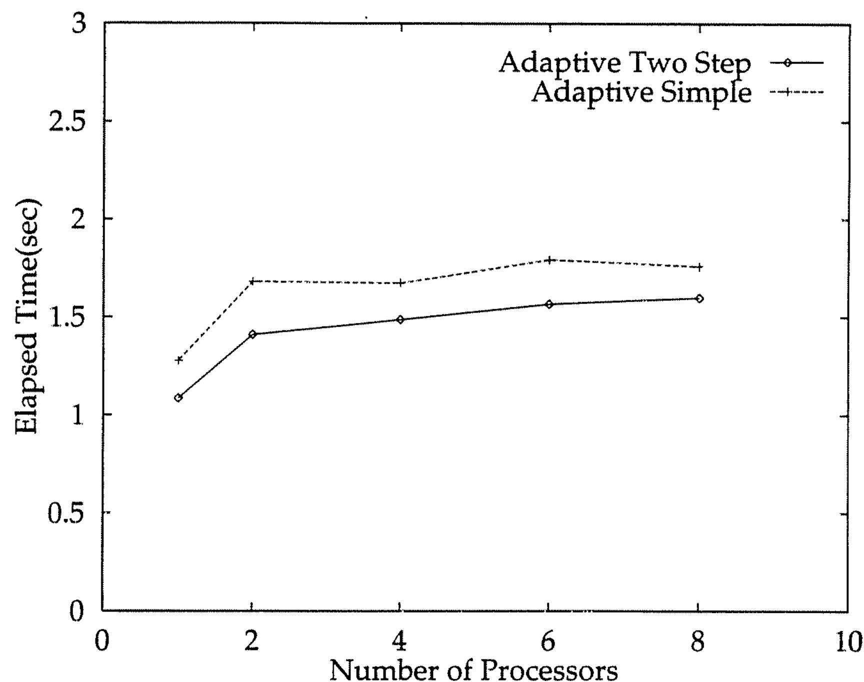

Figure 33: Scaleup Characteristics the Algorithms, 2 groups per CPU

5.5 Shared Nothing Algorithms on SMP

In this section we discuss the performance of the two shared-nothing algorithms, Centralized Two Phase (Section 4.2.1) and Repartitioning (Section 4.2.3). In Chapter 3 we have already shown that using a transparent message passing library results in poor performance of the

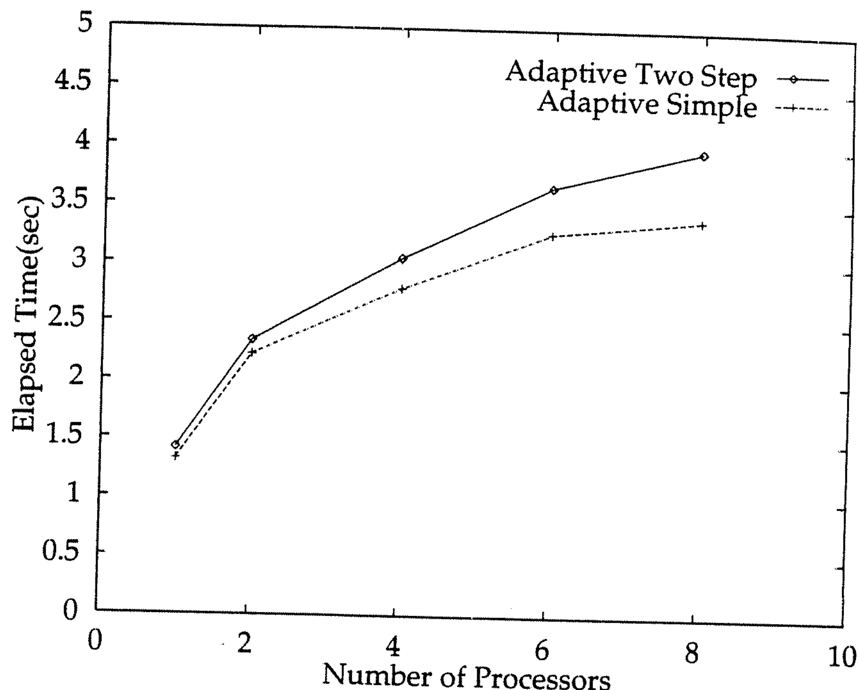

Figure 34: Scaleup Characteristics the Algorithms, 2048 groups per CPU

shared-nothing join algorithms. Hence we study the performance of these two aggregation algorithms using the efficient message passing library.

Figure 35 shows that the Centralized Two Phase algorithm, which is very similar to its adaptation for shared memory, the Two Step algorithm, perform almost identically. The Repartitioning algorithm performs better than the Simple algorithm when the number of groups is small but its performance is relatively worse when the number of groups is large, a point where Simple algorithm performs very well. The other algorithms proposed in Chapter 4 are composed of these elemental algorithms and consequently not studied here.

5.6 Concluding Remarks

We have shown the following via the study of hash based aggregation algorithms on SMPs.

1. Our common-sense intuition—latching and latch wait overhead is typically small and negligible, reflected in the uniform cost memory access (PRAM) model commonly used in the design of algorithms—can result in algorithms that may perform suboptimally. A deeper understanding of the costs and tradeoffs is necessary for design of efficient algorithms.

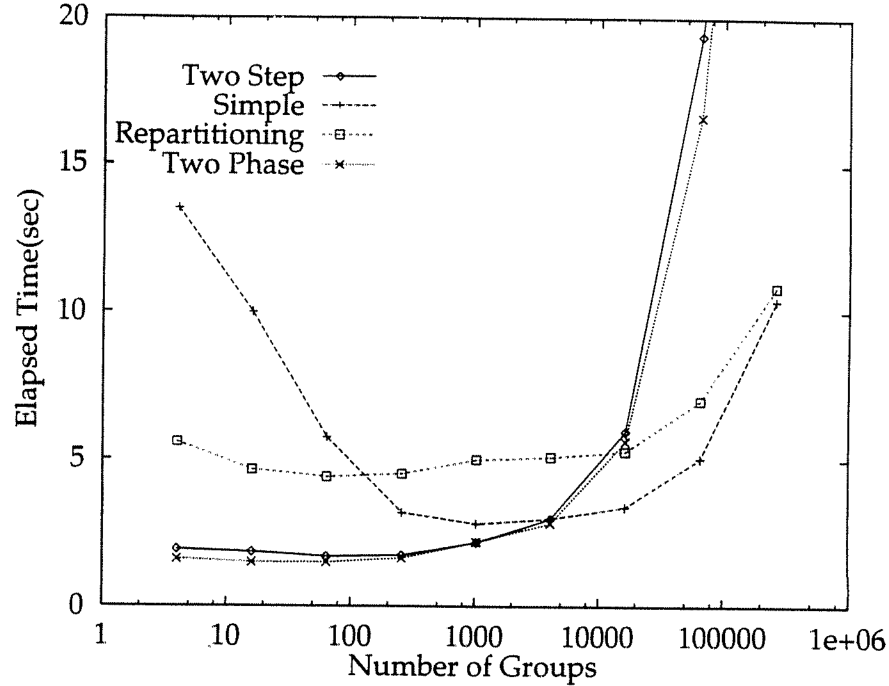

Figure 35: Performance of Shared Nothing Algorithms

2. That shared-nothing algorithms can perform comparably to the shared-memory algorithms on SMP hardware.

Furthermore, we show that whereas each of the classical algorithms is not able to perform well on the entire spectrum of the GROUP BY aggregation work load, in terms of number of groups, the new algorithms are able to cope efficiently with unknown work load. We are able to overcome the shortcomings of the classical approaches, the latching conflicts for the Simple algorithm and the overhead of the second step for the Two Step algorithm. This is made possible by combining the two approaches in a dynamic way where the algorithm decides what to do depending on the observed work load.

If the work load of the system is dominated by GROUP BY's with small number of groups, then one should use the Adaptive Two Step algorithm. If the work load is dominated by GROUP BY's with large number of groups (and duplicate elimination), the Adaptive Simple algorithm would be the preferred algorithm.

Chapter 6

Conclusions

In this thesis we have attempted to demonstrate the following.

1. Shared-nothing and shared-memory software architectures are not disparate and antagonistic, as is generally believed. In fact, they are complementary in design—shared-nothing for coarse-grain parallelism and large bulk transfers and shared-memory for fine-grain parallelism and load balancing—and can have similar performance on the same hardware when used with caution.

   In particular, we showed that shared virtual memory on shared-nothing systems can be used to mitigate the data skew problem commonly associated with the shared nothing architecture. This is possible because of the data sharing afforded by the software shared-memory support. However, the very nature of it requires that sharing be done with caution, for example any algorithm with fine-grain sharing will perform terribly. On the other hand, we showed that shared nothing algorithms, both hash join and hash aggregation, perform comparably to the "native" shared-memory algorithms provided that the message passing, i.e. the shared-nothing layer, was implemented efficiently.

2. Both shared-nothing and shared-memory algorithms benefit from a better understanding of the architecture and architectural tradeoffs. Ignoring architecture details in algorithm design can result in algorithms that may perform sub-optimally.

   Via the example of hash based aggregation algorithms, we showed that our common-sense understanding of the two architectures may result in algorithms that perform suboptimally for a range of workloads. In particular, we showed that the traditional shared-nothing hash aggregation performs poorly when the number of groups in the `GROUP BY` is large. We proposed algorithms that repartition the relations first (i.e. use more communication) in order to avoid the memory overflow (i.e. less I/O) when the number of groups is large. This shows that communication is not always a "bad thing." On the other hand, the

naive approach to shared-memory hash aggregation performs poorly when the number of groups is small because of the increased probability of latch conflicts on the shared hash table. We proposed algorithms that aggregate the tuples in two steps, first each processor builds private local hash tables with no latching and then these local results are merged in the second step, when the number of groups is small. Thus the algorithm does extra computation to avoid latch conflicts. This shows that even efficient latching (and conflicts) can be bad if there are too many of them and steps must be taken to minimize them. Finally, the example of hash join on shared-memory showed that one could design better algorithms by exploiting explicit architectural details like cache memories and considering costs of cache coherence.

3. Algorithms adaptive to the observed work load may be designed with the following paradigm: begin with one algorithm and if the observed work load is not the expected one, change to another algorithm that handles that work load better. This is illustrated via the examples of the hash-based aggregation algorithms and the hash-join algorithm using shared virtual memory.

As part of the study we have developed the following algorithms.

- A load sharing hash join for shared-nothing/SVM architecture (Section 2.4.2).
- Cache conscious hash join algorithm for shared-memory multiprocessors (Section 3.3.4).
- Adaptive hash aggregation algorithms for shared-nothing architecture (Sections 4.3.2 and 4.3.3).
- Adaptive hash aggregation algorithms for shared-memory multiprocessors (Sections 5.4.1 and 5.4.2).

Future Work

Several of the new algorithms proposed in this thesis are adaptive to the observed work load, i.e. they change their behavior depending on the work load observed. This leads to an interesting cost estimation problem for the query optimizer as it does not know initially (and that's the point in having pushed that decision to the runtime) what the work load will be and hence how expensive would it be to run the algorithm.

Though we have touched upon issues of load imbalance and data skew, a more thorough study needs to be done to evaluate the impact of skew on the performance and offer solutions overcoming it on both shared-nothing and shared-memory architectures.

Our work also partially addresses the issue of algorithms for a cluster of SMPs by showing that shared-nothing algorithm performs sufficiently well within an SMP. Therefore, potentially, there need to be just one shared-nothing algorithm within and across the nodes. Investigating the performance of algorithms on a cluster of SMPs would prove very fruitful. Furthermore, transaction processing work loads need to studied on the SMPs.

CM-5, SP-2, iPSC/2, Challenge, PowerChallenge, and UNIX are trademarks of Thinking Machines, IBM, Intel, SGI, SGI, and X/Open respectively.

Bibliography

[AB86] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[BBDW83] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324–353, September 1983.

[BCL93] Kurt P. Brown, Michael J. Carey, and Miron Livny. Managing Memory to Meet Multiclass Workload Response Time Goals. In *Proc. of 19th VLDB Conf.*, pages 328–341, 1993.

[BF93] J. Bunge and M. Fitzpatrick. Estimating the Number of Species: A Review. *Journal of the American Statistical Association*, 88(421):364–373, March 1993.

[BFG+95] Chaitanya Baru, Gilles Fecteau, Ambuj Goyal, Hui i Hsiao, Anant Jhingran, Sriram Padmanabhan, and Walter Wilson. An Overview of DB2 Parallel Edition. In *Proc. of the 1995 ACM-SIGMOD Conference*, San Jose, CA, 1995.

[Bhi88] Anupam Bhide. An Analysis of Three Transaction Processing Architectures. In *Proc. of 14th Int'l Conference on Very Large Data Bases*, pages 339–350, Los Angeles, CA, August 1988.

[CBZ91] John B. Carter, J. K. Bennet, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 1991 Symposium on Operating System Principles*, pages 152–164, 1991.

[Cha96] Satish Chandra, June 1996. Personal Communication.

[CLR94] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? In *Proc. of the 6th ASPLOS Conference*, pages 61–75, October 1994.

[DG85] David M. DeWitt and Robert Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the 12th International Conference on Very Large Databases*, pages 151–164, Stockholm, Sweden, 1985.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[DNSS92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of the 19th International Conference on Very Large Databases*, Vancouver, British Columbia, August 1992.

[ELZ86] Derek L. Eager, E. D. Lazowska, and J. Żahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[Eps79] Robert Epstein. Techniques for Processing of Aggregates in Relational Database Systems. Memorandum UCB/ERL M79/8, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, February 1979.

[ER61] Paul Erdös and Alfred Rényi. On a Classical Problem of Probability Theory. *MTA Mat. Kut. Int. Közl*, 6A:215–220, 1961. Also in Selected Papers of Alfred Rényi, volume 2, pages 617–621, *Akademiai Kiado, Budapest.*

[ESW78] Robert Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1978.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proc. of the 10th Annual Symposium on Theory of Computing*, 1978.

[GD90] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, September 1990.

[Ger95] Bob Gerber. Informix Online XPS. In *Proc. of the 1995 ACM-SIGMOD Conference*, San Jose, CA, 1995.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[HD91] H. I. Hsiao and David J. DeWitt. A performance study of three high-availability data replication strategies. In *Proceedings of the 1st International Conference on Parallel and Distributed Systems*, Miami, FL, December 1991.

[HL91] Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 525–535, Barcelona, Spain, August 1991.

[HS91] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proceedings of the 1st Int'l Conf. on Parallel and Distributed Information Systems*, Miami, Florida, December 1991.

[HT88] Meichun Hsu and Van-On Tam. Managing databases in distributed virtual memory. Technical Report TR–07–88, Aiken Computation Lab., Harvard Univ., March 1988.

[HT89] Meichun Hsu and Va-On Tam. Transaction synchronization in distributed shared virtual memory systems. Technical Report TR-05-89, Center for Research in Computing Technology, Harvard University, 1989.

[KO90] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the Super Database Computer (SDC). In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, England, August 1990.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LKB87] Miron Livny, S. Khoshafian, and Haran Boral. Multi-disk management algorithms. In *Proceedings of the 1987 ACM-SIGMETRICS Conference*, Banff, Alberta, Canada, May 1987.

[LT91] Hongjun Lu and Kian-Lee Tan. A dynamic load-balanced task-oriented approach to parallel query processing. Technical Report Discs Publication TRC7/91, Department of Information Systems and Computer Science, National University of Singapore, July 1991.

[LTS90] Hongjun Lu, Kian-Lee Tan, and Ming-Chien Shan. Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory. In *Proceedings of the 16th VLDB Conference*, pages 198–209, Brisbane, Australia, September 1990.

[LY90] M. Seetha Lakshmi and Philip S. Yu. Effectiveness of parallel joins. *IEEE Transactions on Knowledge and Data Engineering*, 2(4), December 1990.

[Mes94] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. *Int'l Journal of Supercomputing Applications*, 8(3/4), 1994.

[Oak93] Oak Ridge National Laboratory. *PVM 3 User's Guide and Reference Manual*, May 1993.

[Omi91] Edward Omiecinski. Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[PI96] Viswanath Poosala and Yannis Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. To appear in VLDB '96, July 1996.

[RE78] D. Ries and R. Epstein. Evaluation of distribution criteria for distributed database systems. UCB/ERL Technical Report M78/22, University of California, Berkeley, May 1978.

[Sch90] Herb Schwetman. Csim users' guide. MCC Tech Report ACT-126-90, Microelectronics and Computer Technology Corp., March 1990.

[SD89] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 110–121, Portland, Oregon, June 1989.

[SD90] Donovan Schneider and David J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, September 1990.

[Ses92] S. Seshadri. *Probabilistic Methods in Query Processing.* PhD thesis, University of Wisconsin-Madison, Computer Sciences Department, 1992.

[SKN94] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of 20th Int'l Conference on Very Large Data Bases*, pages 510–521, Santiago, Chile, September 1994.

[SM82] Stanley Y. W. Su and Krishna P. Mikkilineni. Parallel Algorithms and Their Implentation in MICRONET. In *Proc. of 8th VLDB Conf.*, pages 310–324, 1982.

[SN93] Ambuj Shatdal and Jeffrey F. Naughton. Using Shared Virtual Memory for Parallel Join Processing. In *Proc. of the 1993 ACM-SIGMOD Conference*, pages 119–128, Washington, D.C., May 1993.

[Syb] Sybase Inc. Sybase Navigation Server . URL: http://www.sybase.com/Offerings/Servers/navserver.html.

[Tan87] Tandem Database Group. Nonstop SQL, A Distributed, High-Performance, High-Reliability Implementation of SQL. In *Workshop on High Performance Transaction Systems*, Asilomar, CA, 1987.

[Ter83] Teradata Corp. *Teradata: DBC/1012 Database Computer Concept and Facilities*, 1983.

[WDJ91a] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th VLDB Conference*, pages 537–548, Barcelona, Spain, September 1991.

[WDJ91b] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[WDYT90] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. IBM T. J. Watson Research Center Tech Report RC 15510, 1990.