# Using Shared Virtual Memory for Parallel Join Processing

Ambuj Shatdal
Jeffrey F. Naughton

Technical Report #1139

March 1993

# Using Shared Virtual Memory for Parallel Join Processing[1]

Ambuj Shatdal
Jeffrey F. Naughton

{shatdal,naughton}@cs.wisc.edu
Computer Sciences Department
University of Wisconsin-Madison

*A shortened version of the paper appears in Proc. of 1993 ACM-SIGMOD Conference*

Computer Sciences Technical Report # 1139
March, 1993

[1]This research was supported by a grant from the IBM Corporation and NSF grant IRI-9157357.

**Abstract**

In this paper, we show that shared virtual memory, in a shared-nothing multiprocessor, 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. The implementation of the algorithm requires few modifications to existing shared-nothing parallel database systems.

# 1 Introduction

The next generation of shared-nothing multiprocessors are expected to be equipped with shared virtual memory (henceforth called SVM) providing a globally shared address space (e.g. the Intel Paragon product literature states that it will provide SVM). 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 paper, 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 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 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. 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.

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 load sharing 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.

The rest of the paper is organized as follows. Section 2 introduces the problem in parallel join algorithms caused by skew in the data and discusses previous solutions to it. Section 3 gives a brief overview of the idea of SVM. The problems in naively using SVM, and our algorithms as a solution are described in section 4. Performance evaluation is presented in section 5 and section 6 offers the conclusions.

## 2 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. We discuss the prominent ones in brief.

Walton et al. [WDJ91] 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, selectivity skew, redistribution skew, and join product skew.

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.

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 sample-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.

# 3 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.

## 3.1 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.

## 3.2 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 try to minimize updatable shared data and maximize processor locality at the same time.

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.

# 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.

## 4.1 Naive Approaches

To show why simply implementing a shared memory algorithm in 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. 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.

### 4.2 A Dual-Paradigm Algorithm

Our dual-paradigm algorithm begins exactly like the basic parallel hybrid hash algorithm [DGS+90]. 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 join 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 a 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 forward the probe tuples to each of the processors participating in its probe phase. $p_1$ 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 probing tuples; however, in today's multicomputers, with up to 200 MB/sec interconnects, the cost of this redistribution will be in the noise when compared to the cost of the read off the disk. A simplified pseudocode version of the join operator code appears in figure 1.

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. As discussed later, this problem can be overcome partly by sampling and using range (instead of hash) partitioning of building relation to get more uniform load in build phase.

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

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

```
/* hash table for a bucket */
shared HASHTABLE_t hash[N];
/* status variable indicating processor status for polling */
shared STATUS_t status[N];
/* processor → {processor} forwarding map */
shared INT_set map[N];/* initially map[a] = { a } */
foreach bucket do{
        while (not end of build relation){
                read tuples from it
                insert t in the hash table hash[p]
        }
        while (not end of probe relation){
                read tuples from it
                if (other processors in map[p])
                    forward the tuples to them in a round robin way
                probe the hash table hash[p]
                if (match)
                    send the result tuple to some appropriate processor
        }
        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
                map[lp] = map[lp] ∪ { p }
                /* indicate to the node that I too am working for lp */
                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 some appropriate processor
                }
            }
            wait at barrier
        }
}
```

Figure 1: Simplified Pseudocode for the Algorithm

almost as well as more complex kinds (e.g. those involving estimated work left).

### 4.3 A Sampling Variant

The algorithm as described in the previous subsection deals elegantly with join product skew, 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 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.

### 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.

## 5 Performance Evaluation

We compare the performance of four algorithms.

**HHJ** Basic Parallel Hybrid Hash Join.

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

**SVMHHJ** Basic SVM Hybrid Hash Join. This is the dual-paradigm algorithm discussed in Subsection 4.2.

**SVMRPHHJ** Sampling based SVM Range Partitioning Hybrid Hash Join. This is the sampling variant referred to in Subsection 4.3.

Our experiments show that if the skew in building relation is not severe, then SVMHHJ performs best, with performance coming close to ideal in many cases. Unlike virtually all previously proposed skew handling algorithms, it suffers almost no performance degradation under the no skew case. If the skew in building relation is extreme, then SVMRPHHJ does best. This is because it mitigates redistribution skew by doing range partitioning.

[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. | Configuration/Node Parameter | Value |
|---|---|---|---|
| Initiate Select | 20000 | Tuple Size | 100 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

### 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.

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 $\theta$ which determines the skew. Unfortunately, increasing value of $\theta$ imply decreasing skew, and hence all of the graphs use $1-\theta$ as the unit of X axis. At $\theta = 1.0$ the relation is uniformly distributed and at $\theta = 0.0$ the relation is significantly skewed[3]. Here we must emphasize that by very nature of Zipf distributions, $1-\theta$ values less than 0.5 signify very small skew and that skew grows quickly with increasing values of $1-\theta$. 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 4. 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 occurences of a single value with $1-\theta$ indicating above mentioned behavior of Zipf distribution.

[3]To keep up with intuition we will use $1-\theta$ as unit for X axis and as a parameter in further discussion. The Y axis unit is seconds.

| $1-\theta$ | 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 | 52990 |

Table 2: Modeling Skew with Zipf Distribution

The size of the relations in our study, except where noted, is 1 Million tuples. This relation size coupled with our hardware configuration meant that all joins were single bucket joins, except for the multi-bucket join experiment.

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 Intel Paragon) 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 Paragon 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 KByte 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[4]. 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

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

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 disks models a slightly simplified Fujitsu Model M2266 (1 GB, 5.25") disk drive. This 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 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 disk characteristics approximate those of the Fujitsu Model M2266 disk drive, as described earlier.

## 5.2 Experiments with Varying Data Skew

### Single Relation Skew

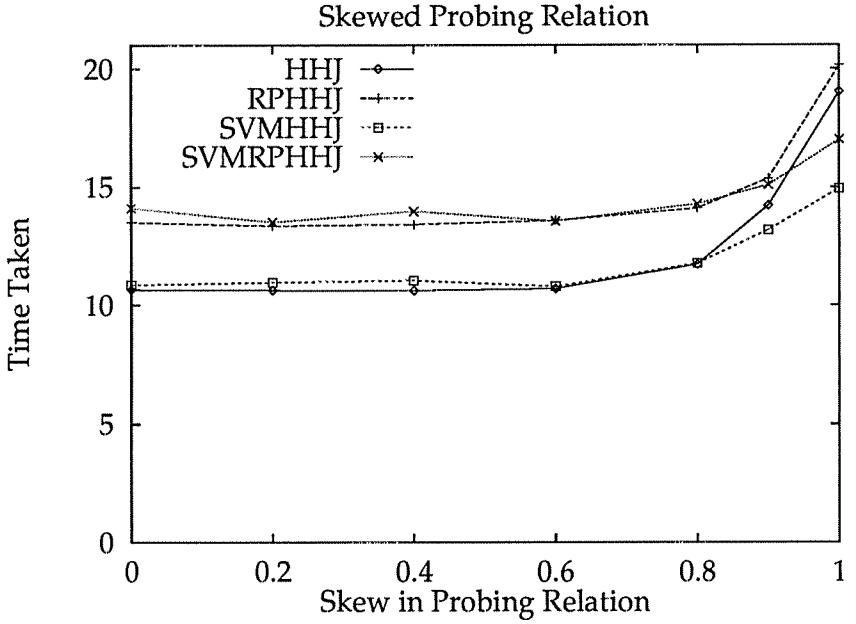


Figure 2: Single Relation Skew: Probe Relation

Figure 2 shows the performance of the four algorithms with no skew in the building relation and varying the skew in the probing relation. We see that when only 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 HHJ and SVMHHJ perform well. In higher skew case we see the load sharing taking effect and the SVMHHJ and SVMRPHHJ both do better than HHJ and RPHHJ respectively.

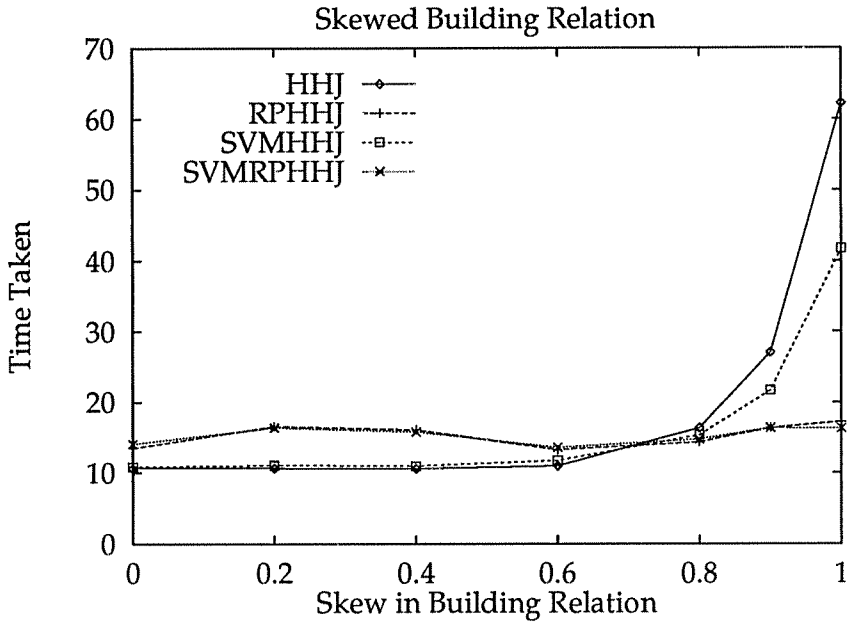


Figure 3: Single Relation Skew: Build Relation

Figure 3 shows the performance of the four algorithms with no skew in the probing 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 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 SVMHHJ algorithm because overflow resulting from skewed build file results in too much imbalance. In both cases, we see that the SVM versions do better the algorithms without SVM in the high skew case.

### Join Product Skew

Figures 4 and 5 consider the case of join product skew. The two graphs represent the same experiment; in figure 5 the lines for HHJ and RPHHJ 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 win 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.

In the figure 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 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 so no equivalent affect was present.

### Other Variations

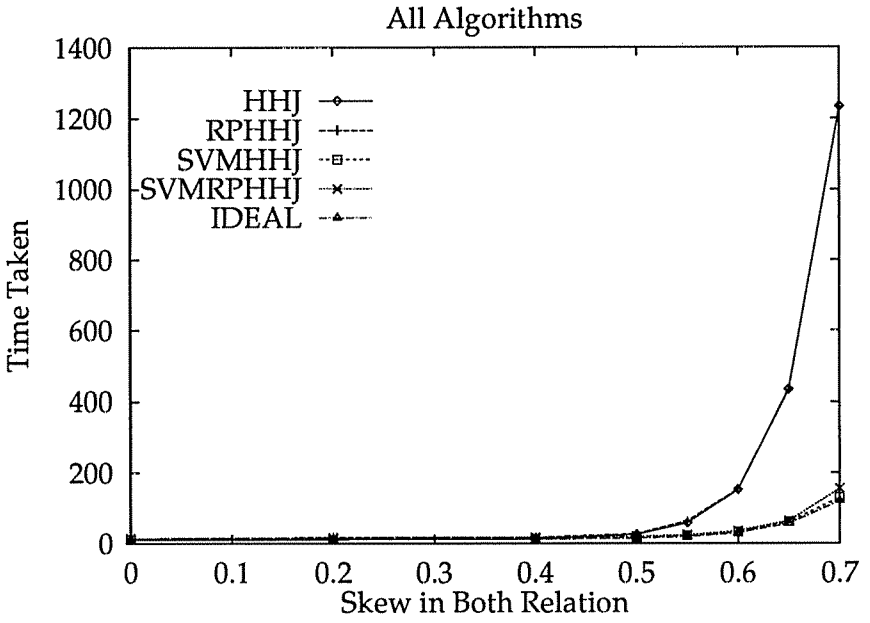


Figure 4: Join Product Skew, Both Relations are Equally Skewed: All Algorithms

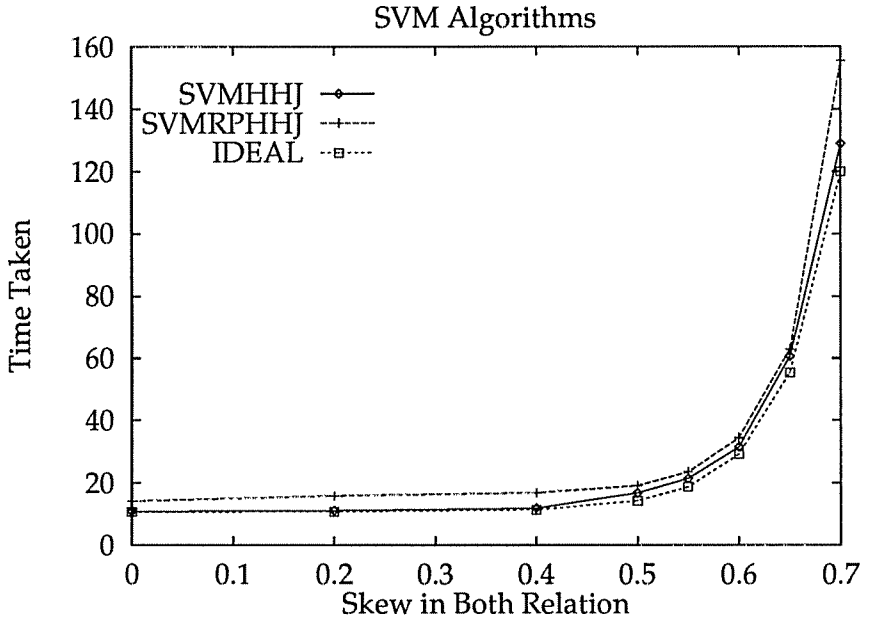


Figure 5: Join Product Skew, Both Relations are Equally Skewed: SVM Algorithms

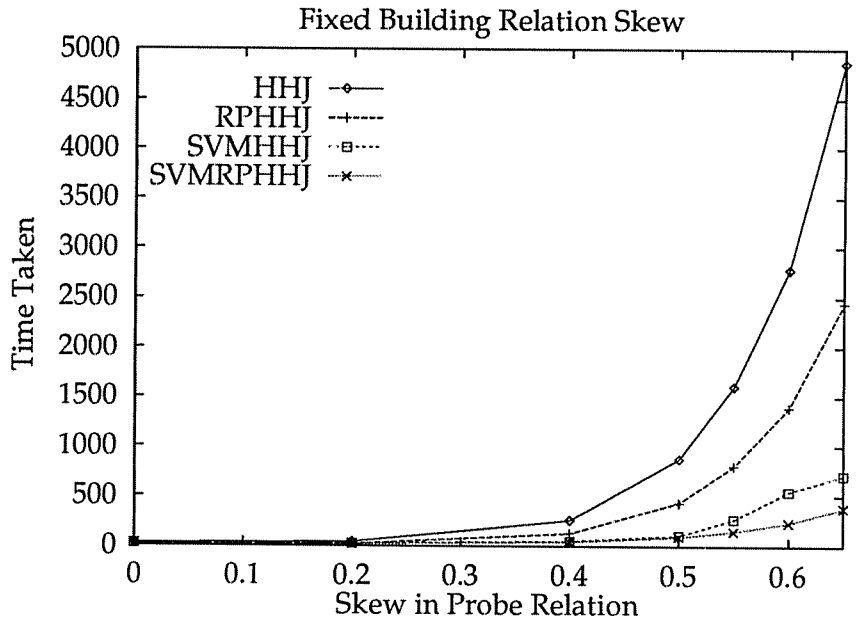


Figure 6: Double Skew, One Relation skewed at $1 - \theta = 0.9$

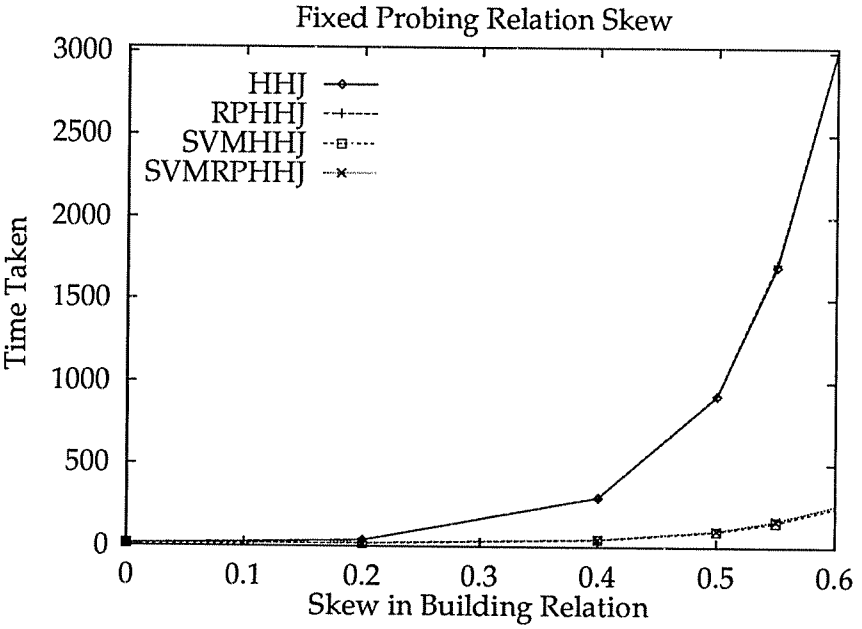


Figure 7: Double Skew, Probe Relation skewed at $1 - \theta = 0.9$

To get a better feel of the performance of the algorithms we also studied other variations. In first experiment, we fixed the skew in building relation at $1 - \theta = 0.9$ and varied the skew in probing relation (see figure 6 (left)). In the second, we fixed the skew in the probing relation at $1 - \theta = 0.9$ and varied the skew in building relation. The results are shown in figure 6 and 7. Both the figures reconfirm our earlier results viz. 1. sampling helps if the building relation is highly skewed making SVMRPHHJ perform the best, e.g. when build relation is skewed with $1 - \theta = 0.9$; if the build relation is not extremely skewed then SVMHHJ does better, and 2. that SVM algorithms do significantly better than their pure message passing counterparts.

### 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 8). The memory size of a node was reduced to 1 MB resulting 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 SVMRPHHJ performing better than SVMHHJ for moderate to large skew.

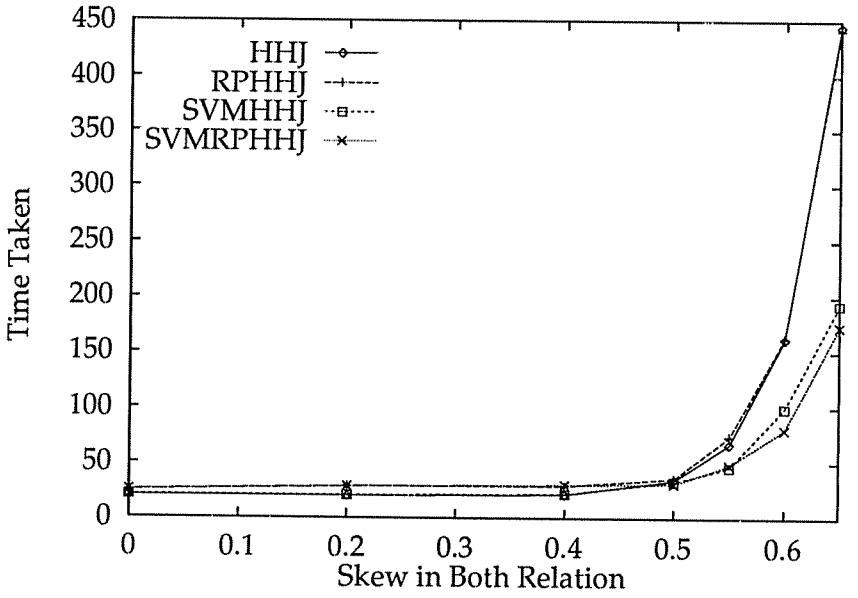


Figure 8: Multi-Bucket Join, Both Relations are Equally Skewed

### 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 number of processor. To show that our algorithms are scalable, we performed speedup and scaleup experiments at the moderate skew of $1 - \theta = 0.5$ in both relations. For the scaleup experiment relation sizes were appropriately scaled from 1 million

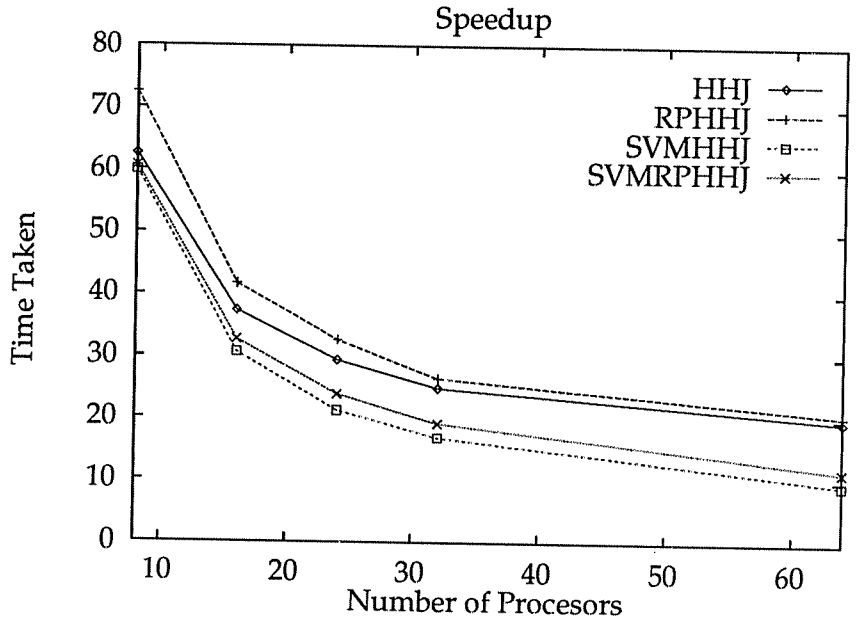


Figure 9: Speedup

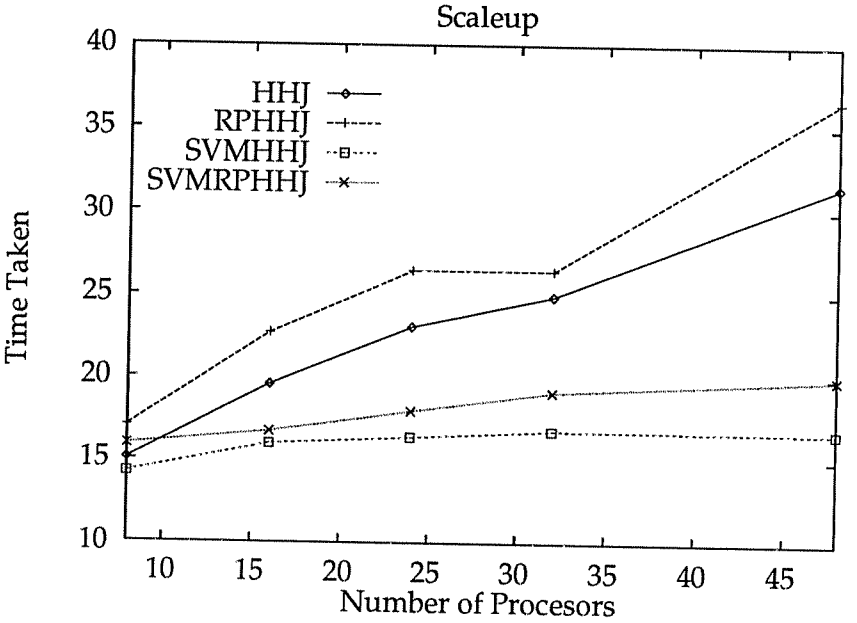


Figure 10: Scaleup

tuples/32 processor configuration. The results of these experiments are shown in figure 9 and 10. The graphs show that the SVM algorithms have better speedup and scaleup characteristics than the original algorithms without the SVM load balancing.

# 6 Conclusions

The algorithms presented in this paper show a simple way of using SVM to drastically improve the performance of join processing on existing parallel database systems in presence of data skew. The needed changes required to existing parallel database systems built for shared-nothing multiprocessors are simple and localized.

The proposed algorithm achieves the performance with little degradation of performance in the no skew case, a significant problem with all other load balancing algorithms. Unlike previous skew handling algorithms, which require statistics about their input relations making it hard to extend the algorithms to multi-way join case, the SVMHHJ algorithm supports multi-way joins as given.

One way to combine the sampling and non-sampling variants of the algorithm to get close to optimal performance is to use the following approach.

```
if (skew is large)
    use SVMRPHHJ
else
    use SVMHHJ
```

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. This approach takes care of the case when too large a skew in building relation worsens the performance of SVMHHJ without the slight performance degradation exhibited by SVMRPHHJ in the lower skew case.

A final result of the paper is the exposition of a clean way of using both SVM and messages in a shared nothing environment. The streams using messages are used for data flow, while shared data structures are maintained in shared virtual memory. This seems to be a good way of getting best of both the programming paradigms, and may be useful in other parallel database applications.


## Acknowledgment

The idea of looking at shared virtual memory for join processing was originally suggested to us by David DeWitt. We also thank the numerous writers of the original Gamma simulator without which the work would not have been possible.


# References


[BBDW83] D. Bitton, H. Boral, et al. Parallel algorithms for the execution of relational database operations. *ACM Trans. on Database Systems*, 8(3), September 1983.

[CBZ91] J. B. Carter, J. K. Bennet, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 1991 Symp. on Operating System Principles*, 1991.

[DG85] D. J. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. of the 12th VLDB Conf.*, 1985.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. Schneider et al. The Gamma database machine project. *IEEE Trans. on Knowledge and Data Engineering*, 2(1), March 1990.

[DNSS92] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. of the 19th VLDB Conf.*, August 1992.

[ELZ86] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5), May 1986.

[ESW78] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational database system. In *Proc. of the ACM-SIGMOD Conf.*, 1978.

[GD90] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proc. of the 16th VLDB Conf.*, September 1990.

[HD91] H. I. Hsiao and D. J. DeWitt. A performance study of three high-availability data replication strategies. In *Proc. of the 1st Int'l Conf. on Parallel and Distributed Systems*, December 1991.

[HL91] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. of the 17th VLDB Conf.*, August 1991.

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

[HT89] M. Hsu and V.-O. Tam. Transaction synchronization in distributed shared virtual memory systems. Technical Report TR-05-89, Center for Research in Computing Technology, Harvard Univ., 1989.

[KO90] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the Super Database Computer (SDC). In *Proc. of the 16th VLDB Conf.*, August 1990.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[LKB87] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. In *Proc. of the 1987 ACM-SIGMETRICS Conf.*, May 1987.

[LT91] H. Lu and K.-L.Tan. A dynamic and load-balanced task-oriented approach to parallel query processing. DISC Technical Report TRC7/91, National University of Singapore, July 1991.

[LTS90] H. Lu, K.-L. Tan, and M.-C. Shan. Hash-based join algorithms for multiprocessor computers with shared memory. In *Proc. of the 16th VLDB Conf.*, September 1990.

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

[Omi91] E. Omiecinski. Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. In *Proc. of the 17th VLDB Conf.*, September 1991.

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

[SD89] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. of the ACM-SIGMOD Conf.*, June 1989.

[SD90] D. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proc. of the 16th VLDB Conf.*, September 1990.

[WDJ91] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. of the 17th VLDB Conf.*, September 1991.

[WDYT90] J. L. Wolf, D. M. Dias, et al. An effective algorithm for parallelizing hash joins in the presence of data skew. IBM T. J. Watson Research Center Tech Report RC 15510, 1990.