Design and Evaluation of Parallel Pipelined Join Algorithms

James P Richardson
Hongjun Lu
Krishna Mikkilineni

Honeywell Inc
Corporate Systems Development Division
Golden Valley, Minnesota

ABSTRACT

The join operation is the most costly operation in relational database management systems Distributed and parallel processing can effectively speed up the join operation In this paper, we describe a number of highly parallel and pipelined multiprocessor join algorithms using sort-merge and hashing techniques Among them, two algorithms are parallel and pipelined versions of traditional sort-merge join methods, two algorithms use both hashing and sort-merge techniques, and another two are variations of the hybrid hash join algorithms The performance of those algorithms is evaluated analytically against a generic database machine architecture The methodology used in the design and evaluation of these algorithms is also discussed

The results of the analysis indicate that using a hashing technique to partition the source relations can dramatically reduce the elapsed time hash-based algorithms outperform sort-merge algorithms in almost all cases because of their high parallelism Hash-based sort-merge and hybrid hash methods provide similar performance in most cases With large source relations, the algorithms which replicate the smaller relation usually give better elapsed time Sharing memory among processors also improves performance somewhat

This research was sponsored in part by Rome Air Development Center contract F30602-85-C-0215

1. Introduction

As enterprises use database management systems to manage more of their information, the size of existing databases is increasing rapidly Databases of over 100 gigabytes now exist, terabyte databases, if they do not exist now, will appear in the next few years Managing these large databases will require more powerful architectures than are in common use today Technology now permits the construction of multiprocessor database management architectures with tens or hundreds of processors, a gigabyte or more of main memory, and disk capacity in the terabyte range The Teradata DBC/1012 is an example of such an architecture [Tera83]

The join operation is an important operation for relational database systems and a large amount of work has been done to develop efficient algorithms to perform the join operation It will become even more important as logic-based inference capabilities are added to these systems In this paper, we describe a number of multiprocessor join algorithms The algorithms use sort-merge and hashing techniques, and are highly parallel and pipelined The algorithms are designed to execute on a multiprocessor architecture that is parameterized in the degree of memory sharing, so that tightly-coupled, loosely-coupled, and intermediate architectures can be modeled Other architectural parameters include the number of processors, number of disks, amount of main memory, and interconnection network bandwidth We model analytically the performance of the algorithms to determine elapsed time, resource utilization, and other quantities as functions of the workload and architectural parameters The join algorithms overlap computation, disk transfers, and interconnection network transfers An important feature of our analysis is that this overlap of different system resources is modeled in the analysis of the elapsed time This analysis also identifies bottlenecks that limit the algorithms' performance We do not model multiple simultaneous join operations and therefore do not compute system throughput

Based on this analysis, we will answer the following questions

- How do the algorithms compare in performance?

When does one outperform another?

- How does elapsed time vary as a function of the architectural parameters?
- How does elapsed time vary with the workload?
- Does shared memory help algorithm performance? To what extent?
- What are the architectural bottlenecks? How could they be alleviated?

In the following sections, we describe the multiprocessor hardware architecture and the join algorithms, develop cost formulas for the algorithms, compare the algorithms' performance under various workloads and hardware configurations, and summarize the results of our investigation

2. Multiprocessor Data Management Architecture

Many specialized architectures have been proposed for high performance relational database management These architectures include logic-on-disk machines [Schu79, Su79], VLSI-based special purpose processors [Kits83, Shib84], and loosely- and tightly-coupled multiprocessor architectures [DeWi79, Tera83, DeWi86] We believe that commercially viable database machines must be constructed principally from the commodity components such as general purpose microprocessors and conventional disk storage devices This belief is based on the superior price/performance and reliability of commodity components compared to custom components Therefore, we consider in this study a multiprocessor architecture with the following characteristics

- The architecture uses a large number (tens to hundreds, at least) of processors to obtain the necessary performance This assumes that the processors can be used effectively The Teradata DBC/1012 appears to have demonstrated that this is possible
- The architecture can use large amounts (hundreds of megabytes to hundreds of gigabytes) of semiconductor memory In the next few years, this amount of memory will be feasible as well as cost-effective
- The architecture can support an aggregate disk capacity of a terabyte or more, only a small fraction of the total database can be accommodated in main memory We assume further that many of the individual database relations will typically not fit in main memory

Figure 1 shows a block diagram of our architecture The architecture consists of a set of *clusters* linked by an intercluster bus or ring Each cluster consists of a set of processors, a shared memory bank addressable by all the processors in the cluster, and a set of disk storage units and associated controllers Processors read and write the shared memory in units of a few bytes, with little contention The processors may have local caches to reduce memory contention, but this is invisible to the data management software except possibly for the need to flush the cache occasionally

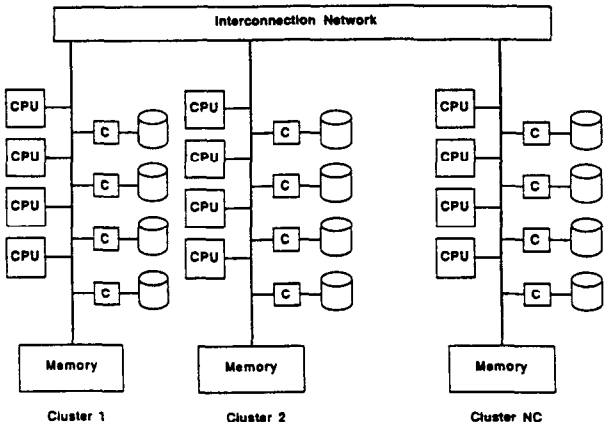

Figure 1 Multiprocessor Data Management Architecture

Transfers between disk and memory, and between cluster memories over the bus are a page at a time, where a page is a few kilobytes or more in size A specific configuration of this architecture is determined by the following parameters

NC	number of clusters
NP	number of processors *per cluster*
ND	number of disks *per cluster*
M	pages of main memory available *per cluster*
PG	page size in bytes

These parameters can be varied to determine the effect of architectural changes For instance, if the CPU is a bottleneck, more CPUs can be added per cluster or each CPU can be made faster (CPU speed is defined in terms of execution times for basic operations associated with the join algorithms, such as tuple move) If the disk is a bottleneck, the page size can be increased, the disk transfer rate can be increased, or more disks can be added per cluster We have assumed that the network is a single bus, so the only architectural cure for a network bottleneck is to increase the network transmission rate

3. Join Algorithm Descriptions

The problem that each join algorithm solves is the following Given relations R and S, compute their natural join on a specified pair of attributes, giving output relation O R and S are assumed to be uniformly partitioned across all disks on all clusters according to attributes that are often used for selections or joins of these relations However, to make the join difficult, we assume that the join attributes for R and S are not the partitioning attributes This forces tuples to be transmitted between clusters to perform the join The partition is not determined by the values of the join attributes, so that tuples must be transmitted between clusters to perform the join Let R_i and S_i denote the fragments of R and S, respectively, stored on the

disks at cluster C_i The result of the join can be partitioned across the clusters, it need not be collected on one cluster No projection is performed on the result except to remove the redundant copy of the join attribute, so no duplicate tuples are produced

Assume without loss of generality that S is larger (in bytes) than R Some of the algorithms transmit only one of the relations over the interconnection network, in these cases, they transmit R, the smaller relation

Each algorithm consists of one or more *phases* The phases are executed one after another, the activity in one phase completes before the next phase starts Within each phase, a fixed set of processes execute in parallel, passing data to each other (possibly over the network) and reading from and writing to disk in pipelined fashion Each cluster has dedicated send and receive processes to act as intermediaries between communicating processes on different clusters

Processes communicate with each other via streams of data pages Each process may have several input and output streams We chose this granularity of communication to minimize the interprocess communication and synchronization overhead The cost of passing a page of data between processes on the same cluster is assumed to be negligible in comparison to the cost of producing or consuming it In some algorithms, processes on the same cluster read concurrently from the same page buffer or other memory area, concurrent reading and writing is not used because it would require a high synchronization overhead The interprocess communication system uses flow control to match the speeds of the producing and consuming processes and to prevent buffer overflow Enough buffers are allocated to allow all processes to execute concurrently

In this section we present six algorithms with the above assumptions The algorithms come in pairs the first of each pair transmits tuples from both R and S over the network, while the second transmits only tuples from R, reducing communications cost while increasing computation The two algorithms comprising the first pair are parallel versions of the the basic sort merge join, they are most similar to algorithms described in [Bitt83] [Vald84] The other four use hashing to decompose R and S into buckets, and then use either a sort-merge or a hashing technique to join each pair of buckets They are similar to algorithms described in [DeWi84] [Kits83]

3.1 Parallel Sort Merge Algorithms

3.1.1 Parallel Sort-Merge Join Type 1 (SMJ1)

In the basic sort-merge join, each relation is first sorted on its join attribute Then, the two sorted relations are merge-joined The merge-join operation matches tuples in the two relations by their join attributes and generates the result tuples It is pipelined much like a merge operation, except that a tuple in either input relation can be used to construct multiple output tuples

Previously published algorithms have presented parallel algorithms for the sort phase Bitton *et al* describe join algorithms employing a parallel binary merge sort and a block bitonic sort [Bitt83] The former can be improved, memory permitting, by using a general multi-way merge [Vald84] Both algorithms start by generating a set of sorted runs from the original unsorted relation These runs are generated using a main-memory sorting algorithm or a priority queue The latter is preferable because it generates runs that are on average twice the size of the main memory dedicated to the priority queue, and hence twice the size of the runs generated by a main-memory sorting algorithm [Knut73] In addition, run generation by priority queue is inherently a pipelined operation, permitting better overlap between CPU and I/O than run generation by main memory sorting Once the runs are generated, the final sorted output is generated either by merging the runs or by using a block bitonic algorithm

The sort-merge join algorithm just described parallelizes the sort operations, but the final merge-join operation is still performed sequentially over the entire length of both relations In the algorithm described below, the final merge-join is partitioned into multiple parallel processes so that no single process must pass over all of either relation This is accomplished by generating $NFRUN_R$ final runs of relation R and $NFRUN_S$ final runs of relation S, and merge-joining each of the final runs of R with each of the final runs of S, all in parallel One of these merge-joins is executed at each cluster, leaving the joined relation O partitioned across the clusters There is an obvious constraint

$$NFRUN_R \; NFRUN_S = NC$$

For convenience in describing the algorithm, let the clusters be named C_{ij} for $1 \le i \le NFRUN_R$ and $1 \le j \le NFRUN_S$ The clusters are logically arranged as a two-dimensional array with $NFRUN_R$ rows and $NFRUN_S$ columns, though their physical interconnection is unchanged (Assume for now that there are at least two rows and two columns The degenerate case is discussed below) The portion of R stored on C_{ij} is called R_{ij} The algorithm has three major phases

Phase 1. At each cluster C_{ij}, NP processes generate initial runs of R_{ij} and write them back to disk Each process has its own priority queue for generating the runs

Phase 2: Each cluster C_{ij} generates initial runs of S_{ij} in the same way

Phase 3: Merge the initial runs of R into $NFRUN_R$ final runs, and the initial runs of S into $NFRUN_S$ final runs Also, merge-join each final run of R with each final run of S to produce the result The merging and merge-joining form a five-stage pipeline The final runs of R are produced in two stages The first stage of the merge occurs at each cluster C_{ij}, where a merge process merges all initial runs of R_{ij} into a single sorted version of R_{ij} One of the clusters in each row, the *row pivot cluster,* executes the second

stage, merging the sorted R_{ij}'s into a final sorted run for the row The final runs of S are generated in the same way except that a *column pivot cluster* executes the second merge stage for clusters in its column The row (column) pivot clusters send their final runs to the other clusters in the row (column), each cluster merge-joins the final run of R for row i with the final run of S for column j to produce O_{ij}, a fragment of the final result By choosing C_{ii} as the row pivot cluster for row i and C_{j+1j} as the column pivot cluster of column j, no cluster is both a row and a column pivot

3.1.2. Parallel Sort-Merge Join Type 2 (SMJ2)

It may be wasteful to sort both relations completely before merge-joining them, especially if the join selectivity is low At each stage of the merging process, tuples are being processed that may not participate in the final join The following algorithm makes only one pass over the larger relation (S) In the description, we revert to single subscripts on clusters and relation fragments The algorithm has three phases

Phase 1: Generate local runs of R_i at each cluster C_i, as in algorithm SMJ1

Phase 2: Merge these runs into a single sorted version of R This is done with a two-stage merge Each cluster executes the first stage, merging the local runs into a sorted version of R_i The results from each cluster are sent to C_1, which executes the second stage of the merge and broadcasts the resulting sorted R to all clusters Each cluster writes this run to disk

Phase 3: Generate runs of S at each cluster using a priority queue However, instead of writing these runs to disk, merge-join them immediately with R to produce the output tuples Let there be NP processes at each cluster executing the run generation stage, paired with an equal number of processes executing the merge-join stage

The advantage of this algorithm is that it produces the join results with one pass over the S relation When S is large, this presumably saves much of the I/O and processing that would otherwise be required to produce the final runs of S The disadvantage is that R must be read repeatedly from disk to be joined against the runs of S (If R fits entirely in main memory, this is unnecessary However, hash-based algorithms are probably superior in this case We do not attempt to fit all of R in main memory)

3.2. Hash Partitioning Join Algorithms

These algorithms all use a hash partitioning technique described in [DeWi84] to decompose a join of two large relations into a sequence of smaller joins They partition tuples of R and S into *batches* RB_0, , RB_{NBATCH} and SB_0, , SB_{NBATCH} and join the respective batches The partitioning is based on the value of a hash function applied to the join attribute, so that joining the respective batches generates all required result tuples The batches are sized so that each batch join can be performed in main memory Batches 1, , $NBATCH$ of each relation are written to disk during partitioning Batches RB_0 and SB_0 are joined during or immediately after partitioning, depending on the particular algorithm If sufficient memory is available, $NBATCH=0$ and no batches must be written to disk Otherwise, batches 1, , $NBATCH$ are read from disk and joined one after the other The number of batches can be computed from the size of the relations and available memory, see Section 4 This partitioning technique minimizes the amount of intermediate data that must be written to disk

All of the algorithms further partition the batches into *buckets* and join the buckets within each batch in parallel Two of the algorithms, HSM1 and HH1, join respective buckets using a sort-merge technique similar to the GRACE algorithm [Kits83] The other two, HH1 and HH2, use the in-memory hash table technique described in [DeWi84] [Brat84] The type 1 algorithms, HSM1 and HSM2, transmit both R and S over the network, the type 2 algorithms, HSM2 and HH2, transmit only R , the smaller relation

3.2.1. Hash-Based Sort-Merge Join Type 1 (HSM1)

This algorithm partitions each batch of R and S further into NC NP_{join} buckets that are joined in parallel by NP_{join} processes on each of the NC clusters A hash function applied to the join attribute of each tuple determines the batch, cluster, and process in which it will be joined Each pair of buckets is joined using a sort-merge join

Let RB_{ij} denote the subset of RB_j derived from R_i, the subset of R stored at C_i Define SB_{ij} similarly The algorithm has three phases

Phase 1: At each cluster C_i, NP_{join} processes read R_i from disk a page at a time The assignment of pages to processes is arbitrary The processes hash each tuple in a page to determine the batch, cluster, and process in which it will be joined If the tuple is in batch $j \neq 0$, it is placed in a buffer to be written to a disk file for RB_{ij} If the tuple belongs to batch 0 but is to be joined on a different cluster, it is placed in a buffer to be sent to that cluster If the tuple is to be joined by a different process on the same cluster, it is placed in a buffer to be sent to the correct process When a process fills one of these buffers, it writes it to disk or sends it to another cluster or process as appropriate When a page is sent to another cluster, it is received by an arbitrary process on that cluster, the tuples in the page are rehashed and sent to another process in the cluster if necessary Once the tuple arrives at the correct process, it is inserted into a binary search tree The search tree will be traversed inorder in the next phase to produce a sorted version of the bucket When all of R_i has been read, S_i is processed in the same way

Phase 2: This phase is repeated for j ranging from 1 to $NBATCH$ If $NBATCH=0$, this phase is omitted At each cluster C_i, each process performs an inorder traversal of its R and S search trees to produce a sorted stream of tuples for each bucket It

merge-joins these tuples to produce the join results As tuples are consumed, the space they occupied is freed As the memory is freed, the processes read the file for batch RB_{ij}, hash each tuple, send the tuples to the appropriate cluster and process, and insert them into search trees occupying the newly freed space When all of RB_{ij} has been read, SB_{ij} is processed in the same way

Phase 3: In this phase, the buckets in batch *NBATCH* are joined as described for phase 2 No data remains to be read from disk and partitioned

3.2.2. Hash-Based Sort-Merge Join Type 2 (HSM2)

This algorithm differs from Algorithm HSM1 in that relation S is not sent over the network Instead, each cluster C_i joins all of R with its portion S_i of relation S A hash function on the join attribute of each tuple determines the batch to which the tuple belongs, and the process on each cluster (in the case of R) or the process on cluster C_i (in the case of S_i) that will do the joining

The algorithm has three phases Only the first phase will be described, the other phases should be clear from the description of phase 1 and Algorithm HSM1

Phase 1: At each cluster C_i, NP_{hash} processes read R_i They hash each tuple to determine the batch and process number Each tuple is buffered either to be written to disk or to be broadcast to the correct process on each cluster Each process constructs a binary search tree of tuples belonging to its own bucket When all of R_i has been read, S_i is processed in the same way, except that tuples from S_i are not transmitted to other clusters, only to other processes on the same cluster

3.2.3. Multiprocessor Hybrid Hash Join Type 1 (HH1)

Algorithm HH1 partitions R and S into batches and buckets in the same ways that Algorithm HSM1 does However, it uses a hash-based algorithm to join each pair of buckets An in-memory hash table is constructed for each bucket of R This table is then probed using tuples from the corresponding bucket of S to produce the join results The concurrent processing of consecutive batches performed in phase 2 of Algorithm HSM1 is not possible here because the hash table for a bucket of R cannot be deallocated until it has been probed by all the tuples in the corresponding S bucket On the other hand, Algorithm HSM1 requires memory to hold S buckets, while this algorithm does not The algorithm has four phases

Phase 1: At each cluster C_i, NP_{join} processes read R_i from disk They hash each tuple and copy it to the appropriate buffer if it must be written back to disk or sent to another cluster or process, as in Algorithm HSM1 Each process constructs a hash table of tuples belonging to its own bucket

Phase 2: At each cluster C_i, the NP_{join} processes read S_i from disk They hash each tuple and buffer it to be written to disk or sent to another cluster if necessary It is not necessary to send a tuple from one process to another on the same cluster, however Once a tuple is at the correct cluster, any process can probe the appropriate hash table to generate the join results

Phases 3 and 4 are repeated for j ranging from 1 to *NBATCH* If $NBATCH=0$, these phases are omitted

Phase 3: This is similar to phase 1 except that each cluster C_i reads RB_{ij} from disk instead of R_i, and performs no disk writes

Phase 4: This is similar to phase 2 except that each cluster C_i reads SB_{ij} from disk instead of S_i, and performs no disk writes

3.2.4. Multiprocessor Hybrid Hash Join Type 2 (HH2)

This algorithm is to Algorithm HH1 as Algorithm HSM2 is to Algorithm HSM1 It has four phases similar to those of Algorithm HH1 However, tuples of S are not transmitted over the network In fact, they are not even transmitted between processes on the same cluster since any process can probe a hash table on the same cluster

3 3. Discussion

The algorithms described above represent the latest versions in a sequence of algorithms These versions provide better overlap in the usage of different resources than earlier versions For example, in all four hash partitioning algorithms, the communications load is spread as evenly as possible over the duration of the algorithm execution Tuples are sent across the network only when they are about to participate in a join One of our earlier versions transmitted all tuples to the joining cluster when the relations were being partitioned, as in DeWitt and Gerber's algorithm [DeWi85] We found that this could cause a network bottleneck during partitioning, the disks and CPUs were not well utilized Spreading the communications load over the duration of the algorithms reduced their execution time

The overlapping among disk I/O, CPU processing and data transfer over the network gives algorithm designers opportunities to tune the algorithms to obtain the trade-off between the elapsed time, total processing cost and memory usage that is best for their system Another example regarding this is the way the hash-based algorithms store tuples in batches 1 — *NBATCH* One possibility is to use one file per batch at each cluster, we chose to use one file per remote cluster at each cluster In the former case, no repartitioning is needed in the later phases, but more buffer pages have to be allocated in the partitioning phase In the latter case, the tuples in the same batch have to be rehashed to determine its bucket number, but far fewer buffer pages are needed to hold the tuples rewritten to disks

4. Performance Comparisons

This section presents performance comparisons of the join algorithms described in Section 3 The main purpose of the performance comparison is to get some insight into the behavior of different algorithms The novelty of this performance analysis lies in two facts First, there are few comprehensive performance studies of join algorithms in the multiprocessor-multidisk architectures [DeWi85] Second, most studies do not model overlap among computation, disk transfers and interconnection network transfers Unless this overlap is modeled, total processing time and elapsed time are equivalent performance metrics We do model this overlap so that we can locate bottlenecks in the algorithms One of the goals of parallel join algorithm design should be to overlap the usage of various system resources as much as possible, thus reducing elapsed time while holding total resource usage constant

The tests conducted can be categorized into three groups that investigate (1) the effects of communication speed, (2) the effects of system configurations, and (3) the effects of data sizes In this section, we first describe the methodology used in the analysis Then the details of the tests, including the parameter settings and test results, are discussed

4.1. Analysis Methodology

For each algorithm, we will compute the following quantities

T	elapsed time
T_{cpu}	total CPU time
T_{disk}	total disk transfer time
T_{net}	total network transfer time

Each of these is the sum over all phases i of the corresponding per-phase quantities T^i, T^i_{cpu}, T^i_{disk}, and T^i_{net} Resource utilization percentages are easily derived from these basic measures The analysis uses the following times for basic operations

t_{comp}	CPU time to compare two attributes
t_{hash}	CPU time to compute hash function of a key
t_{move}	CPU time to move a tuple in memory
t_{swapp}	CPU time to swap two pointers in memory
$t_{build-tuple}$	CPU time to build a join result tuple
t_{send}	CPU time to send a page over network
t_{recv}	CPU time to receive a page over network
t_{net}	network hardware page transfer time
t_{disk}	disk page transfer time

We first compute the following basic quantities for each phase i

P^i_{disk}	the number of disk transfer pages
P^i_{send}	the number of pages sent over the network
P^i_{recv}	the number of pages received over the network
T^i_{join}	CPU time unrelated to disk or net transfers

P^i_{recv} can be greater than P^i_{send} due to broadcasting Then,

$$T^i_{cpu} = T^i_{join} + P^i_{send}\, t_{send} + P^i_{recv}\, t_{recv}$$

$$T^i_{disk} = P^i_{disk}\, t_{disk}$$

$$T^i_{net} = P^i_{send}\, t_{net}$$

We assume that the CPU time attributable to disk transfers is negligible For network communication, we consider both the CPU time and the hardware transfer time, either is a potential bottleneck

The elapsed time T^i for phase i will in general be significantly less than $T^i_{cpu}+T^i_{disk}+T^i_{net}$ due to overlap In our analysis, we assume that sufficient buffering is provided to permit all processes to execute in parallel with each other and with disk and network transfers

- One buffer is allocated for each input and output stream for each process
- One buffer is allocated for each disk to contain the data being read or written
- One buffer is allocated for the network send process at each cluster to hold the next page to be transmitted and one buffer for the network receive process to hold the next incoming page

We also assume that the time for data to flow from the beginning of a pipeline of processes to the end is negligible compared to the total elapsed time for the phase, so that the pipeline is in steady state for most of the phase Under these assumptions, the elpased time T^i can be approximated by the maximum of

- The total disk transfer time of any disk If I/O is spread evenly over all disks, this quantity is $T^i_{disk}/(NC\ ND)$
- The total network transfer time T^i_{net} for the phase
- The total CPU time for any single process, including network send and receive processes
- The total CPU time for any cluster, divided by NP, the number of processors per cluster If processing is spread evenly over all clusters, this quantity is $T^i_{cpu}/(NC\ NP)$ This quantity models processor sharing among the processes at a cluster

The detailed formulas derived in the analysis are not here They can be found in another version of this paper [Rich87]

4 2 Parameter Settings

Three types of parameters are used in the comparisons architectural parameters, timing parameters, and workload parameters The parameter values used are listed in Table 1

4 3. Tests and Results

Now we describe the tests conducted in the performance comparisons Since our results showed that the sort-merge algorithms were generally much slower than the hash-based algorithms, only curves for the the hash-based join algorithms are shown in the figures in this section except for Figure 3, where we include the results of sort-merge algorithms as an example showing the performance difference between sort-

Parameter	Range	Typical Value
Architectural Parameters		
NC	2 - 64	16
ND	2 - 20	8
NP	2 - 64	8
M	16 - 512	128
PG	32K bytes	
Timing Parameters		
t_{comp}	5 μs	
t_{hash}	3 μs	
t_{move}	10 μs	
t_{swapp}	1μs	
$t_{build-tuple}$	20 μs	
t_{send}	0 1 - 3 ms	1 ms
t_{recv}	0 1 - 3 ms	1 ms
t_{net}	0 4 - 2 56 ms (100 - 600 Mbps)	2 56 ms (100 Mbps)
t_{disk}	10 - 30 ms	15 ms
Workload Parameters		
$\|R\|$	50 - 400K pages	100K pages
$\|S\|$	$\|R\|$ - 10 $\|R\|$	$\|R\|$
JS	10^{-8}	

Table 1 Parameter Settings

merge and hash-based algorithms

4 3.1. Communication versus Performance

In this test, the bandwidth of the communications line was varied from 100 Mbps to 600 Mbps to study the effects of the data transfer rate on the performance of the algorithms The results are shown in Figure 2 The elapsed times of type 1 algorithms drop dramatically when the bandwidth increases from 100 Mbps to 300 Mbps This is because the system is network bound with our typical parameter settings for these algorithms In other words, the data transfer was the bottleneck and the bandwidth of the communications line determined their elapsed times In contrast, the elapsed times of type 2 algorithms did not change at all when the bandwidth was varied in the range In these two algorithms, the amount of data transferred equals the size of the small relation *R* It is not a bottleneck when the bandwidth is greater than 200 Mbps

4.3.2. System Configuration versus Performance

The first group of tests that investigated the effects of system configurations on performance consists of the following five tests

- The total hardware resources, that is, the total number of disks (*NC ND*), processors (*NC NP*), and memory size (*NC M*) are kept constant The number of clusters in the system (*NC*) is varied
- The configuration of each cluster is kept the same (i e , fix *NP* and *ND*), but the number of clusters in the system varies
- The number of disks at each cluster, *ND*, is varied, and other parameters are kept constant

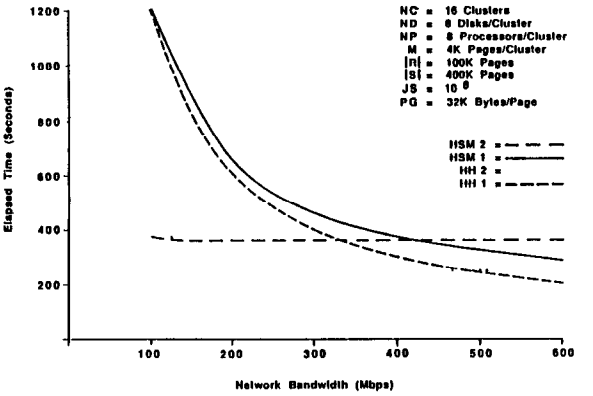

Figure 2 Elapsed Time vs Network Bandwidth

- The number of processors at each cluster, *NP*, is varied, and other parameters are kept constant
- Memory size *M* at each cluster is varied, while other parameters were kept constant

We will describe these tests in more detail

(1) *The elapsed time versus different configurations under the same total hardware resources* One extreme of the spectrum is that all resources form a single large cluster Another extreme is a system such as Gamma [DeWi86], where each cluster has only one processor and one disk The results are shown in Figure 3 The later case is not shown in the figure since the trend is already shown when the system consists of 64 clusters having two disks and two processors each That is the case when each cluster has two disks and two processors When the system consists of 128 clusters with one processor and one disk, the elapsed time for HSM2 is almost doubled compared to the 64-cluster case The other three curves remained flat (not shown in the figure)

From Figure 3, it can be seen that a huge single cluster provides the best performance since the communications cost is eliminated The curves of HH1 and HSM1 are flat since the system is network-bound This will be seen more clearly later The total amount of data transferred in the

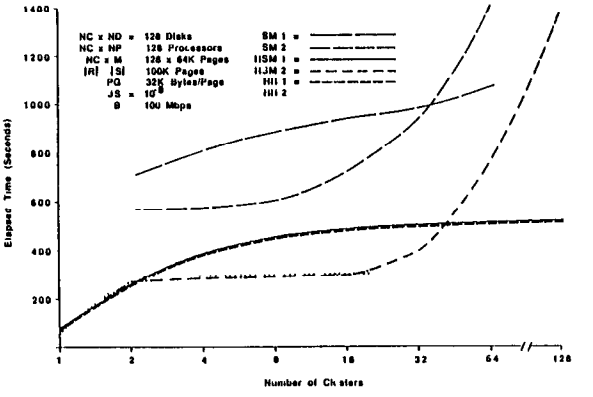

Figure 3 Elapsed Time vs Number of Clusters (Fixed Hardware Resources)

type 2 algorithms equals the size of relation R, which is a constant when the system configuration is changed With the large number of clusters (> 32), HSM2 performs poorly since replicating relation R increases the total processing cost The increasing CPU cost makes the system CPU bound When the number of clusters is doubled, the elapsed time is also doubled For the type 1 algorithms, increasing the number of clusters increases the number of pages sent over the network, $\frac{NC-1}{NC}(|R|+|S|)$ When the number of clusters in the system increases from 2 to 4, this amount increases one third This is reflected in the increase of the elapsed time We ignore the memory contention and the cost of synchronizing the concurrent access of disks in our analysis The huge single cluster case is just an indication of the lower bound of the elapsed time It is impractical to put a large number of disks and processors with shared memory in one cluster

Figure 3 was obtained with a 100 Mbps network With 600 Mbps, the performance is a little different In this case, the type 1 algorithms performed better than their counterparts Figure 4 shows the results As in Figure 3, the type 2 algorithms performed very poorly when the number of clusters exceeded 64

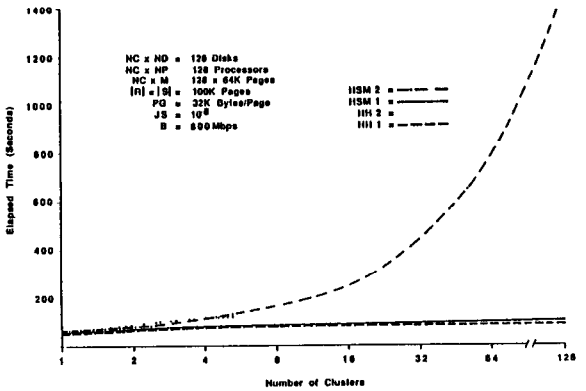

Figure 4 Elapsed Time vs Number of Clusters (Fixed Cluster Configuration)

(2) *The elapsed time versus the number of clusters* In this test, the configuration of each cluster was kept the same and the number of clusters in the system was varied The result of this variation is to increase the parallel processing power of the system and also introduce more data transfer for some algorithms since we assume that the original data is scattered around the system The result of this test is shown in Figure 5

(3) *The elapsed time versus the number of disks at each cluster* In these tests, the number of disks are varied and other parameters is kept as constant Figure 6 shows the result It can be seen from the figure that it is unnecessary to attach more disks to a cluster when the bottleneck is not disk I/O When the number of disks was more than eight in the tested case, increasing the number of disks did not bring real performance benefit

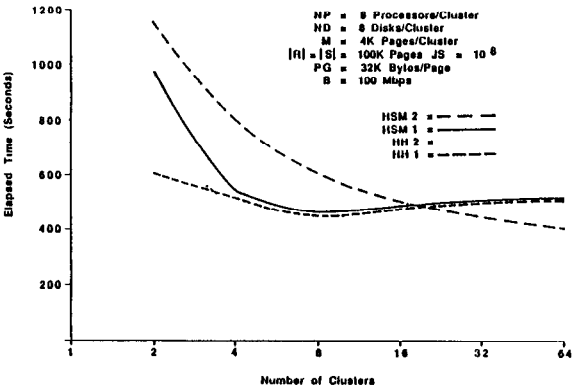

Figure 5 Elapsed Time vs Number of Clusters (Fixed NP, ND, and M)

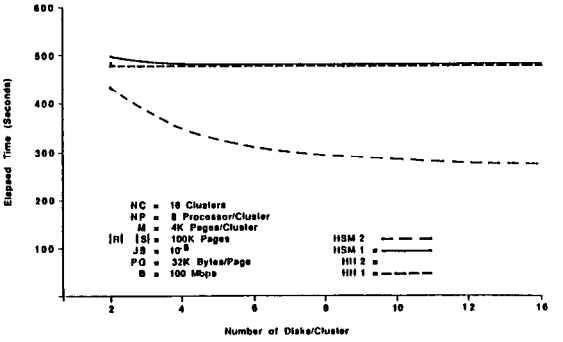

Figure 6 Elapsed Time vs Number of Disks

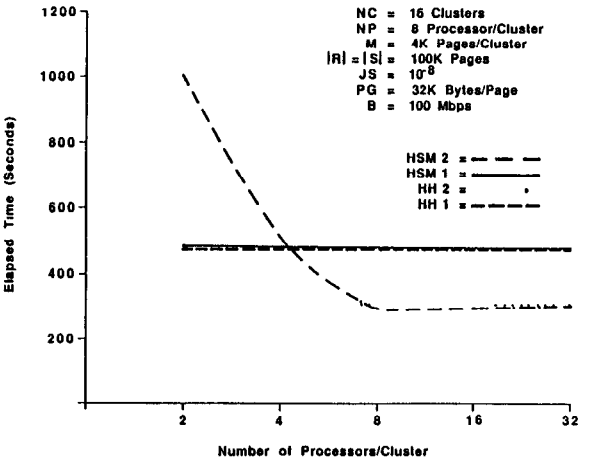

Figure 7 Elapsed Time vs Number of Processors

(4) *The elapsed time versus the number of processors at each cluster* In these tests, the number of processors at each cluster was varied and the results are shown in Figure 7 It can be seen from the figure that CPU processing was not the bottleneck even with two processors at each cluster The only exception was the HSM2 algorithm, which is the most CPU-intensive CPU of these algorithms However, with more than eight processors per cluster, the elapsed time did not decrease further when more processors were added to the clusters Another observation is that, in our buffer alloca-

tion scheme, the number of buffers needed increases proportionally to the square of the number of processors, (not linearly) The large number of processors may cause insufficient memory for executing the algorithms

(5) *The elapsed time versus the amount of memory at each cluster* In this group of tests, the amount of memory at each cluster was varied From the results, shown in Figure 8, it can be seen that the type 2 algorithms required more memory space for buffers That is, the minimum memory requirement is more strict for them However, as long as the memory was big enough to start the algorithm, there was not a big difference in the elapsed time with different memory sizes This can be explained as follows The only benefit a large size memory provide is to save the disk I/O and related rehashing of buckets of 1 — *NBATCH* The processing of the remaining buckets will not be affected by the bucket sizes, which are determined by memory size If the processing cost of the first batch is not the dominant factor of the total processing, or disk I/O is not the bottleneck in the first phase, the memory size will not significantly affect the elapsed time, as seen from the figure

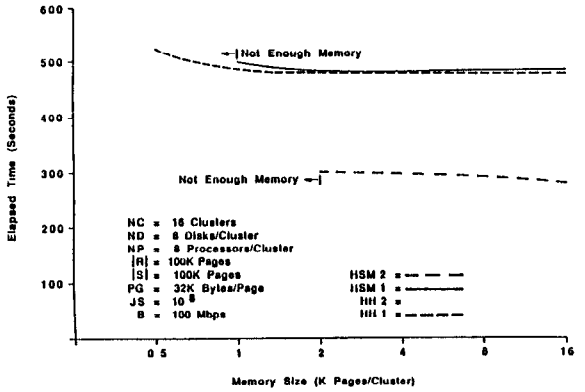

Figure 8 Elapsed Time vs Memory Size

4 3 3 Data Sizes versus Performance

The third group of tests studied the effects of the data size on performance of the algorithms The size of relation R ranged from 1 75 10^9 bytes to 25 10^9 $||S||$ ranged from $||R||$ to 10 $||R||$ Figure 9 depicts the relationship between the elapsed time and the relation size Along with the increase of the size of the two relations, the elapsed time of all algorithms also increased However, the type 1 algorithms were more sensitive to this increase The elapsed time increased linearly with the size of the relations The reason for this is that the bottleneck in these tests is the network The amount of data transferred increases when the relation sizes increases Figure 10 shows the same system and relation sizes with a high bandwidth network (600 Mbps) The first observation is that the type 1 algorithms outperform the type 2 algorithms when the relations were small Second, the elapsed time of all algorithms increases to some extent when the relations become larger The type 1 algorithms were still more sensitive to the relation sizes When the relation sizes become larger, their performance become worse than the type 2 algorithms

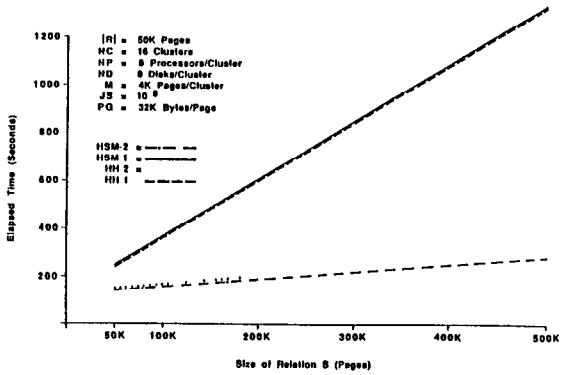

Figure 9 Elapsed Time vs Relation Sizes

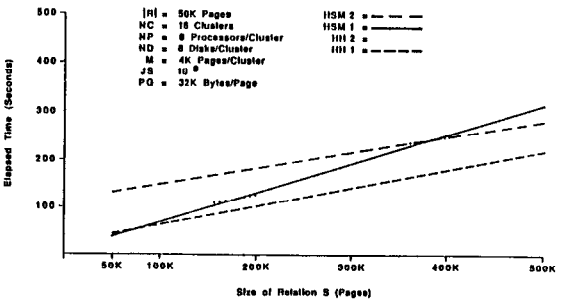

Figure 10 Elapsed Time vs Relation Sizes (Faster Network)

5 Conclusions

In this paper, we have described six different parallel and pipelined join algorithms Some results from our analysis are also presented to compare those algorithms The results of our performance evaluation reiterate the relative performance superiority of the hash-based algorithms compared to sort-based algorithms Our results also show how the effects of overlaps among the different steps of an algorithm effect its relative performance We calculate the bottlenecks in the alternative join algorithms and show that the performance of an algorithm improves by distributing the tasks across the various non-overlapping stages of the algorithm so that maximum overlap and equitable resource utilization are achieved Our results show that intercluster communication bandwidth is typically a bottleneck, and thus, the algorithm or system configuration that reduces intercluster data transfer is preferred The replicated versions of the algorithms typically perform better than their non-replicated counterparts because of the reduced intercluster data transfer

From this study, we can state some basic conclusions about parallel processing of join operations in the multiprocessor environment

(1) The different performance shown by the algorithms studied indicates that it is important to choose appropriate algorithms for a particular join operation with a given system configuration Furthermore, with a given system and relations to be joined, the query optimizer has to carefully determine the number of clusters, the number of disks and the number of processors that will be used in the join Generally speaking, the hash-based algorithms outperform the sort-merge algorithms if the output tuples are not required in the sorted order However, in the case that the source relations are already sorted, or the applications require the output tuples be sorted on the join attributes, the sort-merge algorithms may be advantageous One possibility not mentioned is that of using an order-preserving hash function in the hash-based sort-merge algorithms The sorted order is maintained between different buckets, and the final output tuples can thus be in the desired sorted order The use of an order-preserving hash function should not introduce heavy extra cost

(2) In multiprocessor-multidisk systems, high parallelism can be achieved by dividing the total processing task among processors and disks and executing the subtasks concurrently However, in some algorithms, such as the sort-merge algorithms evaluated in this study, the parallel processing becomes difficult for some steps (final merge, for example) Increasing of the number of processes cannot speed up the processing On the other hand, the hash-based algorithms are naturally parallelizable Both the partitioning and joining phase can be concurrently executed by all participating processors This is the main factro that explains why the hash-based algorithms outperform the sort-merge algorithms with regard to the elapsed time

(3) Among the three major system resources, CPU, disk and communication network, CPU seems to not be the bottleneck of the processing pipeline in general (only in some steps of the sort-merge joins as mentioned above) For hash-based algorithms a small number of processors at each cluster is enough to provide the necessary processing power On the other hand, disk I/O can be the bottleneck of the pipeline, although we intentionally used large page size (32K) and very high disk-memory transfer rate in our study One possible approach is to increase the number of disks at each clusters This multi-disk system can efficiently remove the bottleneck caused by slow disk I/O However, the number of disks that can be attached to one cluster must be limited by the complexity of control

For joins with small or moderate size relations, communications cost should not be a dominant factor in local area networks [Lu85], there is, however, still the possibility that the communications line becomes a bottleneck when a large amount of data has to be transferred in a very large system through the communications line This is especially true for the algorithms where a large amount of data transfer is required (such as HSM1 and HH1)

(4) One key point in the design of a parallel processing algorithm is to achieve maximum overlap among operations requiring different resources in order to increase the parallelism and reduce the effect of the resource that is the bottleneck of the pipeline For example, in the hash-based algorithms, the remotely processed tuples can be transferred either during partitioning or right before their use in the joining phase The total communication cost is the same in these two schemes, while their overlapping with disk I/O is different In the first scheme, all communication occurs while the relations are partitioned The second scheme distributes the communications cost, each relatively small amount of data transfer overlaps with disk I/O and CPU processing in the joining phases Which scheme is better will depend on the relative speed of the disk I/O and data transfer over the network This example reminds us that parallelism between different type of resources can be further increased by tuning the processing steps carefully for each algorithm Furthermore, the precise analysis of such a parallel processing algorithm is very difficult Some simulation or tests in real systems would be useful

Since the system configuration, that is, the number of clusters, the number of processors, the number of disks, and the size of memory used in a join operation, affects the performance along with the relation size and selectivities, query optimization in this multiprocessor environment could be more complicated, and also more important It might be a useful exercise to more thoroughly investigate the relative behavior of different algorithms with regard to the parameters and derive some heuristics to use in the query processing process for such a data flow database machine

References

[Bitt83] Bitton, D, *et al*, "Parallel Algorithms for the Execution of Relational Database Operations," *ACM TODS*, vol 8, no 3, Sept 1983, pp 324-353

[Brat84] Bratsbergsengen, K, "Hashing Methods and Relational Algebra Operations," *Proc Tenth Int'l Conf on Very Large Data Bases*, Singapore, Aug 1984, pp 323-333

[DeWi79] DeWitt, D J, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Managment Systems," *IEEE Trans Computers*, vol C-28, no 6, 1979

[DeWi84] DeWitt, D J, *et al*, "Implementation

Techniques for Main Memory Database Systems," *Proc SIGMOD '84*, June 1984, pp 1-8

[DeWi85] DeWitt, D J, and Gerber, R, "Multiprocessor Hash-Based Join Algorithms," *Proc VLDB 85*, Stockholm, Aug 1985, pp 151-164

[DeWi86] DeWitt, D J, *et al*, "GAMMA - A High Performance Dataflow Database Machine," *Proc VLDB 86*, Kyoto, Japan, Aug 1986, pp 228-237

[Kits83] Kitsuregawa, M, Tanaka, H, and Motooka, T, "Application of Hash to Data Base Machine and its Architecture," *New Generation Computing*, vol 1, no 1, 1983, pp 63-74

[Knut73] Knuth, D E, *The Art of Computer Programming, Volume 3 (Sorting and Searching)*, Addison-Wesley, Reading, MA, 1973

[Lu85] Lu, H, and Carey, M J, "Some Experimental Results on Distributed Join Algorithms in a Local Network," *Proc VLDB 85*, Stockholm, Aug 1985

[Rich87] Richardson, J P, Lu, H, and Mikkilineni, K, *Design and Evaluation of Parallel Pipelined Join Algorithms*, Honeywell Corporate Systems Development Division Technical Report (to appear)

[Schu79] Schuster, S A, *et al*, "RAP 2 - An Associative Processor for Databases and its Applications," *IEEE Trans Computers1, vol C-28, no 6, 1979*

[Shib84] Shibayama, S, *et al*, "A Relational Database Machine with Large Semiconductor Disk and Hardware Relational Algebra Processor," *New Generation Computing*, vol 2, no 2, 1984

[Su79] Su, S Y W, *et al*, "The Architectural Features and Implementation Techniques of the Multicell CASSM," *IEEE Trans Computers*, vol C-28, no 6, 1979

[Tera83] Teradata Corporation, *DBC/1012 Data Base Computer Concepts and Facilities*, Inglewood, CA, Apr 1983

[Vald84] Valduriez, P, and Gardarin, G, "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *ACM TODS*, vol 9, no 1, March 1984, pp 133-161