Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory

Hongjun Lu[*] *Kian-Lee Tan*[*] *Ming-Chien Shan*[†]

[*] Department of Information Systems and Computer Science, National University of Singapore
[†] Hewlett Packard Laboratory, Palo Alto, CA 94303

ABSTRACT

This paper studies a number of hash-based join algorithms for general purpose multiprocessor computers with shared memory where the amount of memory allocated to the join operation is proportional to the number of processors assigned to the operation and a global hash table is built in this shared memory. The concurrent update and access to this global hash table is studied. The elapsed time and total processing time for these algorithms are analyzed. The results indicate that, hybrid hash join that outperforms other hash-based algorithms in uniprocessor systems does not always performs the best. A simpler algorithm, hash-based nested loops join, performs better in terms of elapsed time when both the relations are of similar sizes.

1. Introduction

In database query processing, join is a very time consuming operation and thus a large amount of work has been done to develop efficient algorithms to perform the join operation. With the trend moving towards multiprocessing environment, several parallel join algorithms have been proposed and studied [DeWi85, Qada88, Rich87, Schn89, Vald84]. These algorithms are parallel versions of the traditional nested loops, sort-merge, hashing techniques or their combinations [Brat84, DeWi84, Shap86]. Though the parallelized nested loops and sort-merge join methods are simple and easy to implement, work by [DeWi85, Rich87, Schn89] have shown that hash-based join algorithms outperform them under most conditions. These algorithms, however, were mostly developed and studied in the environment of uniprocessor computers with large main memory [DeWi84, Shap86], shared-nothing multiprocessor systems [Schn89] and database machines. The observations that multiprocessor computers are getting popular and that most such machines are for general purpose computing and not dedicated to database applications motivated our study on database query processing and optimization for general purpose multiprocessor computer systems. This paper presents the result of the first phase of our study — the performance of hash-based join algorithms in such computer systems.

The major differences between our study and previous work are as follows: First, the number of processors is a major architectural parameter and it can only be determined when a join is to be performed. Furthermore, considering the memory management mechanism used by most operating systems, the amount of memory available for join processing is assumed to be proportional to the number of processors allocated to the join operation. That is, increase in number of processors for a join operation implies that memory available for the operation is also increased. Second, memory available to a join operation is organized as a memory pool shared by all processors participating in the operation. This memory pool is managed by the database management system and a global hash table is built for hash-based join algorithms. A locking mechanism is used to regulate any concurrent write to this global hash table with the assumption that the architecture permits concurrent read but exclusive write. Finally, though join algorithms overlap computations and disk transfers, most of the previous work do not consider the overlap. In our study, both the total processing time and the elapsed time of different join algorithms are analyzed. The elapsed time is taken as the maximum of disk I/O time and CPU time so that the overlap is taken into account.

Our results show that the uniprocessor Hybrid Hash Join algorithm is not always the best in a multiprocessor environment. It does not exploit the memory well during the partitioning phase resulting in high contention. A modified version, which eliminates contention in the partitioning phase, is proposed. The Hash-based Nested Loops Join algorithm has better elapsed time performance than the Hybrid Hash-Join algorithm when both relations are of similar sizes. We also see that an algorithm with low elapsed time may not necessarily be the better algorithm as it may require a higher total processing cost.

In the following section, we describe the architectural model for our multiprocessor system. In section 3, we present the various hash-based join algorithms with their cost formulas. Section 4 compares the performance of the algorithms. Finally, our conclusions and suggestions for future research are contained in section 5.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

2. The Multiprocessor Computer System

2.1. The Architecture

The multiprocessor systems we are concerned are general purpose systems without any special-purpose hardware for database operations such as sorting of relations. Enslow summarized the salient characteristics of a multiprocessor system as follows [Ensl77] :

- the system has a set of general-purpose processors with identical capabilities,
- all the I/O devices and I/O channels are shared by all the processors,
- all processors have accessed to a common pool of memory modules,
- the processors, the I/O devices and the shared memory modules are connected by an interconnection network,
- the operation of the entire system is controlled by one operating system.

Such a general multiprocessor organization is shown in Figure 2.1.

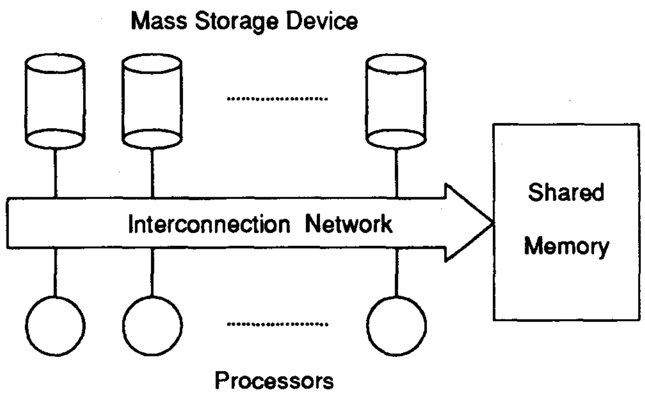


Figure 2.1. A multiprocessor computer system.

The number of processors of such system is relatively small compared to some database machines that may consist of a few hundred or even thousands of processors [Tera83]. Each processor shares the common memory (shared memory) with other processors. It may also have some buffers dedicated to itself (local memory) for input/output. It is reasonable to assume that, when a processor is assigned some task to execute, it is also allocated a certain amount of memory space. From the view point of database operations, if a certain number of processors is allocated to process a query, the control of these processors and related memory will be transferred to the database management system. It is up to its buffer management subsystem to efficiently use the available memory space. Such a multiprocessor architecture can provide both inter- and intra-request parallelism. That is, the processors may either independently execute different relational database operations, or execute the same database operation at the same time.

The machine uses conventional disk drives for secondary storage and databases (relations) are stored on these disk storage devices. Both disks and memory are organized in fixed-size pages. Hence, the unit of transfer between the secondary storage and memory is a page. The processors, disks and memory are linked by an interconnection network. This network allows different processors to read from the same page of the shared memory at the same time (broadcasting). For writes, different processors can only write to *different* pages at the same time. We assume that a locking mechanism is used to enforce this concurrent access policy and the locking granularity is a page. We also assume that there is no I/O cost associated with locking, that is, the lock table is assumed to be in main memory. Under this mechanism, a processor has to obtain a lock on a memory page to which it intends to write. The lock is released after data is written to the page. Since concurrent read is allowed, there is no need to lock a page if the operation is a *read* operation. However, it is assumed that the interconnection network has sufficient bandwidth for the tasks at hand and the contention for the interconnection network is not considered in our following analysis.

Finally, though it is expected that main memory sizes of a gigabyte or more will be feasible and perhaps even fairly common within the next decade, we cannot assume that a whole relation can be read from mass storage to either the processor's local memory or the shared memory before processing. That is, in general, both the total memory of the processors and the size of the shared memory are not large enough to contain a whole relation.

2.2. Concurrent write to the shared-memory

One major issue in analyzing the performance of multiprocessor computers with shared memory is the possible contention that happens when more than one processor intends to write to the same memory page concurrently. In the case of hash-based join algorithms, there are two possible ways in which this might happen. First, when a global hash table is used to explore the benefit of the shared memory, it is likely that different processors may hash different tuples into the same page at the same time. Second, more than one processor may output tuples of the same partition to the same buffer page at the same time during the partitioning phase.

As we mentioned above, a locking mechanism is used to enforce our memory sharing policy. When a processor cannot obtain the lock for writing, it must wait. This implies that, if p processors should write to *the same page at the same time* (i.e. with contention), the processing cost (time) will be p times the processing cost without contention, since the request would be queued for processing serially. Therefore, by letting the expected number of processors that write to the same page at the same time be ξ, we have

$$Proc_c = Proc_{nc} \times \xi$$

where $Proc_c$ and $Proc_{nc}$ are CPU processing cost with and without contention respectively.

To determine the expected concurrent writes, ξ, we formulate the problem as follows:

> *Given $||R||$ tuples and M memory pages, (1 < M ≤ $||R||$), if p tuples (p ≤ $||R|| - ||R||/M$) are randomly selected from the $||R||$ tuples, find the expected number of pages with at least one tuples to be written to.*

This is none other than the problem of characterizing the number of granules (blocks) accessed by a transaction [Yao77, Lang82]. The solution to the above problem is given by Yao's theorem [Yao77] which states that the expected number of blocks hit is given by

$$\zeta = M \cdot \left[1 - \prod_{i=1}^{p} \frac{||R|| \times D - i + 1}{||R|| - i + 1} \right] \quad where\ D = 1 - \frac{1}{M}$$

Therefore, the expected number of tuples falling on a page at the same time can be expressed as $\xi = \frac{p}{\zeta}$ It should be noted that when $M = 1$, $\xi = p$. Since ξ is dependent on $||R||$, M, and p, we also denote it as $C(||R||, M, p)$ in our later analysis.

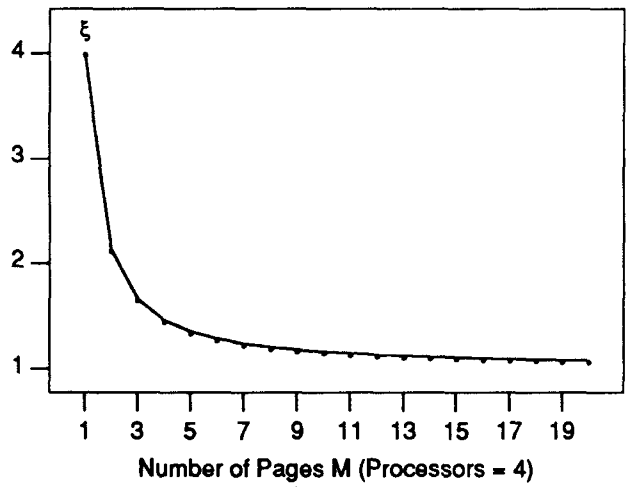


Figure 2.2. Contention ξ versus number of pages M.

Figure 2.2 depicts the value of ξ with respect to the number of pages M. The number of processors is 4 and the relation contains 1000 pages with 40 tuples per page. So, for $M = 1$, that is, all tuples are supposed to be written to the same page, ξ equals 4, that is, all writes must be done sequentially. When M increases, the expected number of tuples falling into the same page decreases dramatically. We will see the effects of this contention factor in later analysis.

3. The Hash-Based Join Algorithms

In this section, we discuss the hash-based join algorithms for the system described in the previous section. We first categorize the hash-based algorithms followed by the general methodology used in our analysis. The description of the algorithms and their cost formulas are then presented.

3.1. Categorization of hash-based join algorithms

Given two relations, R and S, the basic approach of hash-based join methods is to build a hash table on the join attributes for one relation, say R, and then to probe this hash table using the hash values on the join attributes of tuples from the other relation, S. The join result is formed by the matching tuples. Since we assume that memory available is usually much smaller than the size of relations to be joined, it is impossible to build the hash table for the entire relation R. The hash-based join algorithms usually process the join in *batches*. In each batch, only a portion of R is read into memory and the corresponding hash table is built. There are a few possible ways to form portions from relation R.

	Partitioning		S: prior	S: on-fly	S: no
R	prior	keep R_0 in memory	Hybrid-Hash		
		R_0 not in memory	GRACE		
	on-fly			Simple	Simple w/toss
	no				Hash-Loops

Table 3.1 : Categorization of hash-based join algorithms

1) ***To partition the relation prior to join process.*** All tuples in the relation are read, hashed on the join attributes and written back to disk as ***partitions*** according to the hash values in such a way that tuples of each partition can fit in memory. Each batch of the processing will work on one partition in the subsequent join process.
2) ***To partition the relation on-fly.*** The partitioning of the relation can also be done on-fly. That is, during each batch, the tuples that have not been used to build hash tables are read in and hashed. Those tuples that belong to the current partition are inserted into the hash table while other tuples are either tossed away or written back to the disk and processed in later batches.
3) ***No partitioning according to hash value.*** The simplest way to partition a relation is to read in pages sequentially until memory is fully occupied by the hash table.

Relation S that is used to probe the hash tables can be treated in the same ways. The benefit of prior partitioning of S is that only those tuples from the corresponding batches need to be compared to form the join results. Otherwise, all tuples from relation S has to be used to probe the hash table during each batch. One variation of prior partitioning is to retain the first partition in memory to reduce some disk I/O. The hash-based join algorithms proposed so far can thus be categorized as shown in Table 3.1. DeWitt and Gerber [DeWi85] reported some experimental results on the performance of four algorithms listed in Table 3.1, the Grace algorithm [Good81, Kits83], the Hybrid algorithm [DeWi84], the Simple hash and the Hash loops algorithms [DeWi85]. In the same paper, they presented some simulation results on the performance of the multiprocessor versions of the Hybrid and Grace algorithms.

3.2. The elapsed time versus total processing time

In most of the previous work on analytical modeling of join algorithms [Bitt83, DeWi84, Vald84], the elapsed time is used as a criteria to evaluate the performance of an algorithm. In such cases, the best algorithm is that which minimizes the elapsed time. Moreover, most analysis assumed that there is no overlap in disk transfers and computations. The elapsed time is essentially the sum of the computation and disk transfers times. An exception is the work of Richardson, Lu and Krishna [Rich87] which models overlap in computation, disk transfers and interconnection network transfers.

In our analysis, we like to emphasis two of our observations. First, in a multiprocessor or parallel processing environment, it is possible to increase parallelism by duplicating part of computation among different processors. Some algorithms deliberately use this duplication to minimize the elapsed time. As the result, an algorithm that achieves minimum elapsed time may require high total processing time. This is different from what we have in uniprocessor systems where shorter elapsed time means lesser total processing time. An obvious implication of high total processing time is that more resources are tied down to the particular task and hence may decrease the system's overall performance. Hence, both the elapsed time and the total processing time are important in choosing a suitable multiprocessor algorithm. Second, the overlap between different resources is quite important. The overlap should be taken into account because we should not only model real systems more closely but also understand the behavior of an algorithm in more detail. It is highly desirable that an algorithm make good use of both CPU and disk resources. Especially for most parallel processing algorithms, some subtasks can be done in parallel but others have to be in sequential. Whether a subtask is CPU-bound or I/O-bound becomes an important factor in determining the overall performance of an algorithm.

With the above observations, both the elapsed time and the total processing time of an algorithm are analyzed in our study. The process of computing a join is divided into *phases* that are executed one after another. Within each phase, there may be several passes of a series of operations. In other words, phases are executed in sequential while within a phase different tasks can be either parallelized among processors or be overlapped among CPU and disks. To evaluate an algorithm, we first compute the required disk I/O time per disk drive and CPU time per processor for each phase i in executing an algorithm, T^i_{IO} and T^i_{CPU}. For a multiprocessor system with d disk drives and p processors, the total processing time for phase i is then

$$T_i = p \times T^i_{CPU} + d \times T^i_{IO} \qquad (1)$$

The total processing for the algorithm that requires n phases to complete the computation is

$$T = \sum_{i=1}^{n} T_i \qquad (2)$$

The elapsed time E_i for phase i will in general be less than $T^i_{CPU} + T^i_{IO}$ due to overlap. It can be explained as follows. In each phase, the processors can begin its processing as soon as some pages from both relations are in memory. Moreover, while the processors are computing the join operation, the other pages may be read in at the same time. Hence, the elapsed time can be computed as the maximum of the above I/O time and CPU time:

$$E_i = \max(T^i_{CPU}, T^i_{IO}) \qquad (3)$$

That is, if a phase is CPU-bound, the elapsed time equals to the CPU time needed and if it is I/O-bound, the elapsed time equals to the I/O time required. Here we assume that, for a CPU-bound phase, the time to read in the initial pages before the processing begins, and the time to write out the final pages of the resulting tuples are negligible compared to T^i_{CPU}, while for an I/O-bound phase, the time of processing the last few pages are negligible compared to T^i_{IO}. Since all phases of an algorithm are executed in sequential, the elapsed time of an algorithm with n phases is

$$E = \sum_{i=1}^{n} E_i \qquad (4)$$

3.3. Algorithms and analysis

We analyzed four hash-based join algorithms for the multiprocessor system described in section 2: Hybrid Hash-Join(HHJ), modified Hybrid Hash-Join (MHHJ), Simple Hash Join (SHJ) and Hash-based Nested Loops Join (HNLJ). HHJ and MHHJ are variations of multiprocessor Hybrid Hash-Join algorithms with different memory allocation strategies during the partitioning phase. They represent the algorithms that partition both relations before the join process. HNLJ was chosen as the representative of algorithms that do not partition S. For this group of algorithms, the process of probing the hash tables using tuples from S is the same but both prior and on-fly partitioning require extra work to partition R. It is therefore expected that the Hash-based Nested Loops performs the best among them. The Simple Hash was chosen as the representative of the algorithms that partition relations on-fly. In the following discussion, we only present the I/O and CPU times for each phase, T^i_{IO} and T^i_{CPU}, and the elapsed times, E_i and E, and the total processing times, T_i and T, can be easily computed using equations (1) — (4). The detailed derivation of the results can be found in [Lu90]. The parameters and their values used in our analysis are listed in Table 3.2.

Hybrid Hash-Join

The Hybrid Hash-Join algorithm [DeWi84] is a variation of the GRACE Hash-Join algorithm [Kits83] . Both algorithms comprises of two phases — the *partitioning* and the *joining* phase. The former phase divides the relations R and S into disjoint buckets such that the buckets of relation R are of approximately equal size. The latter phase performs the join of the corresponding buckets of

Primitive Parameters		
t_{probe}	time to compare two keys	0.003 ms
t_{comp}	time to compare two attributes	0.003 ms
t_{hash}	time to compute hash function of a key	0.006 ms
t_{move}	time to move a tuple in memory	0.05 ms
t_{lock}	time to acquire a lock	0.01 ms
t_{unlock}	time to release a lock	0.01 ms
IO_{seq}	time to perform a sequential I/O operation	15 ms
IO_{rand}	time to perform a random I/O operation	25 ms
JS	join selectivity, defined by size (R JOIN S)/($\lvert R\rvert \times \lvert S\rvert$)	0.001 (0.00025)
$\lvert M\rvert$	size of shared memory available for the join process (in pages)	32 x p (64 x p)
$\lvert R\rvert$	size of relation R (in pages)	1000
$\lvert S\rvert$	size of relation S (in pages)	1000 (4000)
$\lVert R\rVert$	number of tuples in relation R	
$\lVert S\rVert$	number of tuples in relation S	
tu_R	number of tuples per page of relation R	40
tu_S	number of tuples per page of relation S	40
d	number of disk drives	6 (12)
p	number of processors available for query processing	
q	proportion of R that falls into partition R_0 for Hybrid hash-join	
B	number of partitions	
F	fudge factor	1.4
F_R	average number of probes to insert an R tuple	1.2
F_S	average number of probes to find a match for an S tuple	1.2

Derived Parameters		
EIO_{seq}	effective time for a sequential disk I/O	$\frac{IO_{seq}}{d}$
EIO_{rand}	effective time for a random disk I/O	$\frac{IO_{rand}}{d}$
$\lVert Res\rVert$	number of tuples in the join result	$\lVert R\rVert \times \lVert S\rVert \times JS$
t_{build_tuple}	time to build a result tuple	$2 \times t_{move}$
t_{insert}	time to insert a tuple of R into hash table (lock the page, probe the hash table, move the tuple and release the lock)	$t_{lock} + F_R \times t_{probe} + t_{move} + t_{unlock}$
t_{output}	time to move a tuple into the output buffer	$t_{lock} + t_{move} + t_{unlock}$
t_{find_match}	time to find a match for an S tuple	$F_S \times (t_{probe} + t_{comp})$

Table 3.2. Parameters and test values.

R and S. The GRACE algorithm uses one page of the memory as a buffer for each bucket of R so that there are at most $|M|$ buckets with each bucket containing $\lceil |R|/|M| \rceil$ pages. This means that each bucket requires $\lceil F \times |R|/|M| \rceil$ pages of shared memory to construct a hash table. Thus, the algorithm requires that

$$|M| \geq \sqrt{F \times |R|}$$

This restriction on the minimum amount of memory required is still necessary for the Hybrid Hash-Join algorithm. However, whereas the GRACE algorithm partitions the relations into $|M|$ buckets, the Hybrid algorithm chooses the number of buckets such that the tuples in each bucket will fit in the memory so as to exploit the additional memory to begin joining the first two buckets. The extra memory, if any, is used to build a hash table for a partition of R that is processed at the same time while R is being partitioned. The corresponding partition of S is used to probe the hash table while S is being partitioned. Hence, this partition is not rewritten back to disk and processed again in the second phase. Let B+1 be the number of buckets that relation R is partitioned into, where

$$B = \max\left\{ 0, \left\lceil \frac{F \times |R| - |M|}{|M| - 1} \right\rceil \right\}$$

as given in [DeWi84]. The sizes of R_0 and R_i $(1 \leq i \leq B)$ are $|R_0| = \frac{|M| - B}{F}$ and $|R_i| = \frac{|M|}{F}$ respectively.

The execution of the hybrid hash join can be divided into four phases that are executed serially : (1) to partition R, (2) to partition S, (3) to build the hash table for tuples from R and (4) to probe the hash tables using tuples from S and to form the join results. The required disk I/O and CPU processing time for each phase, T^i_{IO} and T^i_{CPU} $(1 \leq i \leq 4)$ can be computed as follows.†

† In fact, for each phase, there are more than one batches. The formulas sum up the processing times of all batches in the same phase. This is the same for all subsequent analysis.

Phase 1: To partition relation R

$$T_{IO}^{1} = |R| \times EIO_{seq} + |R| \times (1-q) \times EIO_{rand}$$

$$T_{CPU}^{1} = \frac{||R|| \times q}{p} \times (t_{hash} + t_{hash} + t_{insert} \times \xi_1)$$

$$+ \frac{||R|| \times (1-q)}{p} \times (t_{hash} + t_{output} \times \xi_1)$$

where $\xi_1 = q \times C(||R||, |M| - B, p) + (1-q) \times C(||R||, B, p)$

Phase 2: To partition S and join R_0 and S_0

$$T_{IO}^{2} = (|S| + q \cdot |Res|) \times EIO_{seq} + |S| \times (1-q) \times EIO_{rand}$$

$$T_{CPU}^{2} = \frac{||S|| \times q}{p} \times (t_{hash} + t_{hash} + t_{find\_match})$$

$$+ \frac{||S|| \times (1-q)}{p} \times (t_{hash} + t_{output} \times \xi_2)$$

$$+ \frac{q \times ||Res||}{p} \times t_{build\_tuple}$$

where $\xi_2 = (1-q) \times C(||S||, B, p) + q$

Phase 3: To build hash tables for R

$$T_{IO}^{3} = (1-q) \cdot |R| \times EIO_{seq}$$

$$T_{CPU}^{3} = \frac{||R|| \times (1-q)}{p} \times (t_{hash} + t_{insert} \times \xi_3)$$

where $\xi_3 = C(\frac{||R|| \times (1-q)}{B}, |M|, p)$

Phase 4: To join S with R

$$T_{IO}^{4} = (1-q) \cdot (|S| + |Res|) \times EIO_{seq}$$

$$T_{CPU}^{4} = \frac{||S|| \times (1-q)}{p} \times (t_{hash} + t_{find\_match})$$

$$+ \frac{||Res|| \times (1-q)}{p} \times t_{build\_tuple}$$

Modified Hybrid Hash-Join

In the above Hybrid Hash-Join algorithm, each bucket is allocated one buffer page during the partitioning phase. When the number of buckets is relatively small to the number of processors, the contention due to conflicting writes to the buffer during the partitioning phase results in long waiting time. The modified Hybrid Hash-Join algorithm (MHHJ) intends to reduce this contention by allocating more output buffer pages during the partition phase. That is, $p \times B$ buffer pages are used for writing out the tuples in buckets R_i, $(1 \le i \le B)$. In this way there are no buffer contention in the partitioning phase. The available memory for R_0, however, decreases and the value of B becomes

$$B = \max\left\{0, \left\lceil \frac{F \times |R| - |M|}{|M| - p} \right\rceil\right\}$$

and the sizes of R_0 and R_i change accordingly.

The execution of MHHJ is also divided into four phases. For the first phase and the second phase, ξ will be 1 since there are no contention. However, an extra cost was introduced to merge the unfilled pages together for each bucket before writing back to disk. In the worst case, half the number of pages are moved for each bucket for each processor. We have the following cost formulas.

Phase 1: To partition relation R

$$T_{IO}^{1} = |R| \times EIO_{seq} + |R| \times (1-q) \times EIO_{rand}$$

$$T_{CPU}^{1} = \frac{||R|| \times q}{p} \times (t_{hash} + t_{hash} + t_{insert} \times \xi_1)$$

$$+ \frac{||R|| \times (1-q)}{p} + \frac{B}{2} \cdot \frac{tu_R}{2} \times t_{move}$$

where $\xi_1 = q \times C(||R||, |M| - p \times B, p) + (1-q)$

Phase 2: To partition S and join R_0 and S_0

$$T_{IO}^{2} = (|S| + q \cdot |Res|) \times EIO_{seq}$$

$$+ (|S| \times (1-q)) \times EIO_{rand}$$

$$T_{CPU}^{2} = \frac{||S|| \times q}{p} \times (t_{hash} + t_{hash} + t_{find\_match})$$

$$+ \frac{||S|| \times (1-q)}{p} \times (t_{hash} + t_{output})$$

$$+ \frac{q \times ||Res||}{p} \times t_{build\_tuple}$$

$$+ \frac{B}{2} \cdot \frac{tu_S}{2} \times t_{move}$$

The cost formulas for the third phase and the fourth phase of MHHJ are exactly the same as those for HHJ and are not repeated here.

Hash-based Nested Loops Join Algorithm

The Hash-based Nested Loops Join algorithm is a modified version of the traditional nested loops algorithm — hash tables are built on the join attributes of the outer relation to efficiently find the match tuples. When the available memory is smaller than the size of the outer relation, more than one pass is needed to complete the join. During each pass, only $H = min(|R|, \frac{|M|}{F})$ pages of the outer relation R are read into the memory and a hash table is constructed. Then the entire inner relation S is scanned and each tuple is used to probe the hash table. The use of a hash table avoids the exhaustive scanning of all the R tuples in memory for every tuple in S as is done in the nested loops algorithm. Though the entire inner relation S is scanned at every pass of the algorithm, the entire relation R is scanned only once.

The number of passes required, k, is given by

$$k = \left\lceil \frac{|R| \times F}{|M|} \right\rceil$$

The cost of HNLJ can be computed as two phases, one phase is to read in relation tuples of R and to insert them into the hash table, and another phase is to scan S and to probe the hash tables and to output the results.

Phase 1: To read in R and to construct hash tables

$$T^1_{IO} = |R| \times EIO_{seq}$$

$$T^1_{CPU} = \frac{||R||}{p} \times (t_{hash} + t_{insert} \times \xi) \text{ where } \xi = C(||R||, H \times F, p)$$

Phase 2: To read in S, to probe the hash table, and to output the result

$$T^2_{IO} = (k \times |S| + |Res|) \times EIO_{seq}$$

$$T^2_{CPU} = k \times \frac{||S||}{p} \times (t_{hash} + t_{find\_match}) + \frac{||Res||}{p} \times t_{build\_tuple}$$

Simple Hash-Join Algorithm

The Simple Hash Join analyzed in our study is just a multiprocessor version of the simple hash algorithm proposed in [DeWi84]. As in Hash-based Nested Loops Join algorithm, the join is completed in a number of passes. During each pass, a hash table is built for part of relation R, and relation S is then scanned to probe that hash table. The Simple Hash Join, however, does the on-fly partitioning of R and S. That is, tuples that do not belong to the current partition are written back to the disk. With this partitioning, the number of S tuples to be scanned during each phase decreases. The cost is that unprocessed R and S have to be written back to disk. From this description, it is easy to see that the number of passes required to complete the join, k, will be

$$k = \left\lceil \frac{|R| \times F}{|M|} \right\rceil$$

and on the i^{th} pass, $i = 1, 2, ..., k-1$, the number of tuples of R that remains to be processed is $||R|| - i \times \frac{|M| \times tu_R}{F}$. Let $H = min(\frac{|M|}{F}, |R|)$. The processing time can still be computed as two phases.

Phase 1: To read in R tuples, to insert portion of them into the hash table and to write others back to disks

$$T^1_{IO} = \left[|R| + 2 \times ((k-1) \times |R| - \frac{k \times (k-1)}{2} \times H) \right] \times EIO_{seq}$$

$$T^1_{CPU} = \frac{||R||}{p} \times (t_{hash} + t_{insert} \times \xi) + k \times \frac{p}{2} \times \frac{tu_R}{2} \times t_{move}$$
$$+ \frac{1}{p} ((k-1)\, ||R||$$
$$- \frac{k\,(k-1)}{2} \times (H \times tu_R)) \times (t_{hash} + t_{move})$$

where $\xi = c(||R||, H, p)$

Phase 2: to read in S tuples, to join them with R, and to write unprocessed tuples back to disks.

$$T^2_{IO} = \left[|S| + 2\,((k-1) \cdot |S| - \frac{k\,(k-1)}{2} \cdot H \cdot \frac{|S|}{|R|}) + |Res| \right]$$
$$\times EIO_{seq}$$

$$T^2_{CPU} = \frac{||S||}{p} \times (t_{hash} + t_{find\_match}) \times t_{move}$$
$$+ \frac{||R||}{p} \times t_{build\_tuple} + \frac{1}{p} \times ((k-1) \times ||S||$$
$$- \frac{k \times (k-1)}{2} \times (H \times tu_S \times \frac{|S|}{|R|})) \times (t_{hash} + t_{move})$$

4. Performance Studies

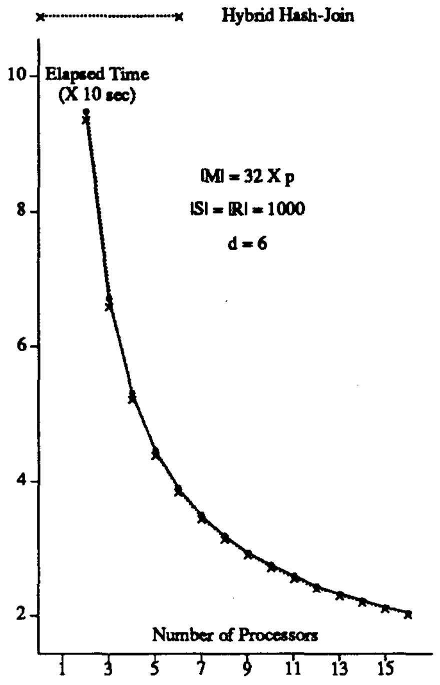
The relative performance of the various algorithms was studied using the cost formulas presented in section 3. The elapsed and total processing times of the algorithms are compared under various conditions as the number of processors increases. In particular, the performance is compared by varying the amount of memory allocated to each processor, the size of the larger relation relative to the smaller one and the number of disks available in the system. The parameter settings is given in Table 3.1. In this section, we present and discuss the results.

4.1. Output buffer pages for Hybrid Hash-Join

The major difference between the Hybrid Hash Join (HHJ) and the Modified Hybrid Hash-Join (MHHJ) is the number of buffer pages allocated to processors during the partitioning phase. Figure 5.1 shows the elapsed and total times of these two algorithms. It can be seen that HHJ has smaller elapsed times than MHHJ. This is because the partition phases of both algorithms are I/O-bound with the given parameters. Since more memory pages are used as output buffers, the size of the R_0 partition is smaller for MHHJ than HHJ. Thus, for MHHJ, more pages need to be written back to disk, incurring more I/O costs. Another factor is that, when the number of processors is small, the joining phase of both algorithms are CPU-bound. Since R_0 is smaller for MHHJ, it needs to process more pages during the joining phase, resulting in higher CPU time too. The total effect is that the elapsed time for MHHJ is longer than that for HHJ.

However, from the view point of total processing time, MHHJ incurs less total processing time than HHJ. The difference becomes larger as the number of processors increases. This is due to conflicting writes during partitioning. Algorithm HHJ only allocates one page for each bucket. As the number of processors increases, the amount of shared memory available increases and hence the number of partitions decreases. This implies that more tuples is to be written to the same buffer page and the expected contention increases. In the extreme case where R is partitioned into two buckets (R_0 and R_1), 50% of the tuples have to be written to the buffer page by p processors and a long waiting time is expected. This increase in CPU time results in higher total processing time. On the other hand, MHHJ eliminates the contention for writing to the output buffers.

Buffer allocation is always a sensitive issue where performance is concerned. Algorithms HHJ and MHHJ are using two extreme strategies — either one buffer page per bucket or p pages per bucket. There might be some better strategies that could give better elapsed time than MHHJ but with lesser total processing time than HHJ. Since the difference between the elapsed times of the two algorithm is not significant, in the subsequent performance comparisons, we will only present the



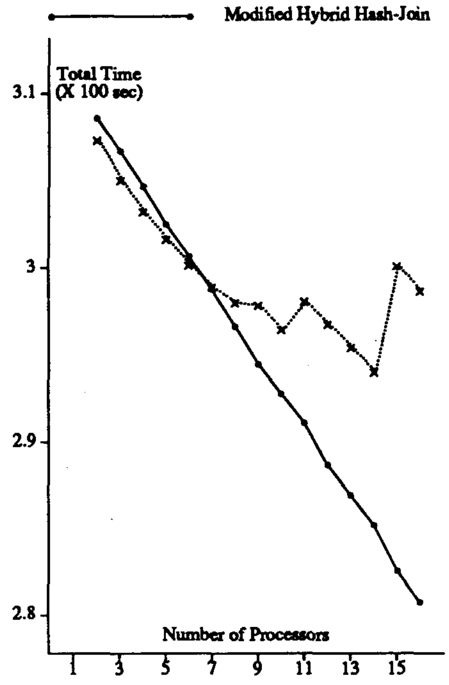


Figure 5.1. Comparison of hybrid and modified hybrid hash-join.

results of the MHHJ algorithm rather than both.

4.2. Relative performance

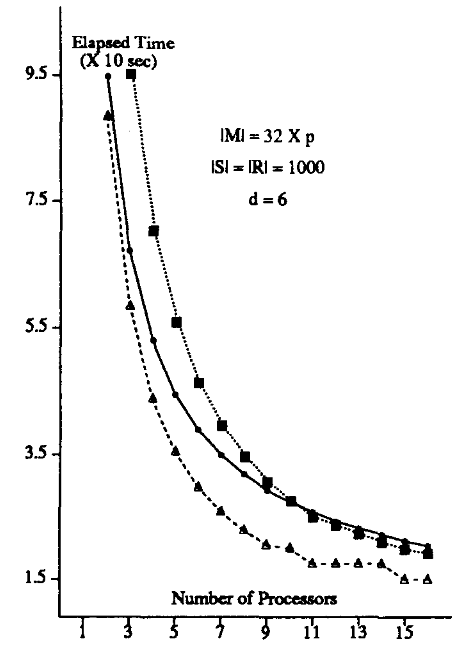
Figures 5.2 — 5.5 show the elapsed times and total times of the Modified Hybrid Hash-Join, the Hash-based Nested Loops and the Simple Hash-Join algorithms with different amount of available memory, different relation sizes and different number of disks in the system.

Among the three algorithms, the Simple Hash Join algorithm performs worst in most cases. When the number of processors increases and the amount of memory available increases, its performance can be comparable with the other algorithms but it is always bounded by either the Hybrid Hash-Join or the Hash-based Nested Loops Join. The major reason is that the Simple Hash Join partitions the relations on-fly. It reads and writes the relations repeatedly and thus incurs a large number of disk I/Os. The Hybrid Hash-Join also partitions the relations but it passes through them less than three times. As for the Hash-based Nested Loops Join, it has to scan relation S several times, but it only scan R once. However there are some instances that SHJ performs a little better than MHHJ. This can be explained as follows: First, although the number of disk I/Os for SHJ may be higher, but all the disk I/Os are sequential reads and writes. For MHHJ, some writes are random accesses. The time needed for random disk I/O is 5/3 times that for sequential disk I/O in our analysis. Second, there are $p \times B$ output buffer pages to be merged before writing out while there are only p pages in SHJ case.

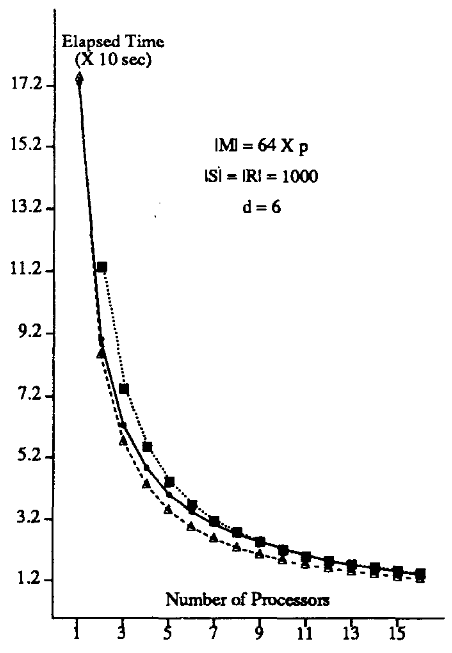
The performance of all the three algorithms, both for the elapsed time and the total processing time, become better when the number of processors increases. This is mainly because of the architectural assumptions of our model. In our system, an increase in the number of processors means both increase in the amount of memory available for the join operation and the CPU processing power. For The Modified Hybrid Hash-Join, large amount of memory means large R_0 and reduces the number of pages written back to the disks and reread during the joining process. the number of disk I/O's and thus reduces both the elapsed time and the total processing time. For both the Simple Hash Join and Hash-based Nested Loops Join, larger memory size means lesser number of scans of the S relation. Another interesting fact is that when the number of processors is small, it is very effective to introduce more processors to reduce the elapsed time. But after the number of processors reaches some point, the performance gain from allocating more processors will not be so much.

The total processing times show similar behaviors with the exception of the Modified Hybrid Hash-Join where the total processing time does not decrease a lot when the number of processors increases. This is because the total processing time of MHHJ will only be affected by the size of R_0. However, this increase is limited. For SHJ and HNLJ, the number of scans through S is the major portion of the total processing time. With the increase of memory, it will reduce the number of scans of S and hence improve the performance of the algorithms. For HNLJ, if the increase in memory is not large enough to reduce the number of scans of S, the performance will not be affected at all. The staircase shape of the HNLJ curves clearly indicates this. For example, both the elapsed time and the total processing time for the HNLJ does not change when the number of processors

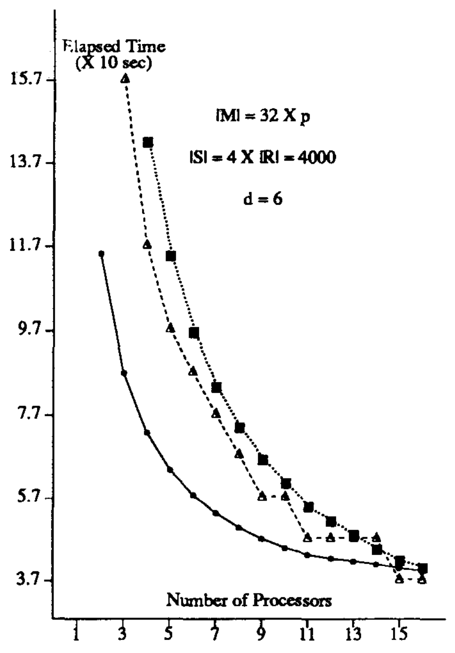
●——● Modified Hybrid Hash-Join ▲- - - -▲ Hash-based Nested-Loop Join ■·········■ Simple Hash-Join



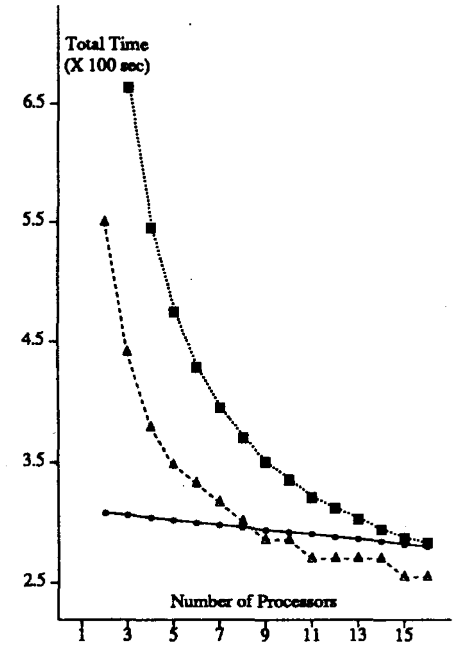
5.2a.



5.3a.

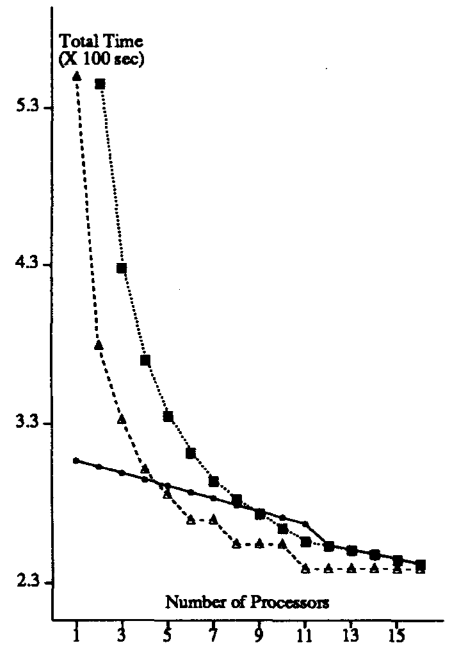


5.4a.



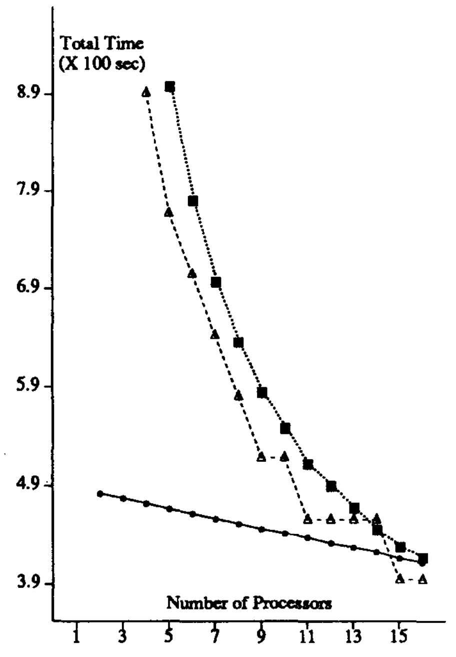
5.2b.

Figure 5.2. Base Experiment.



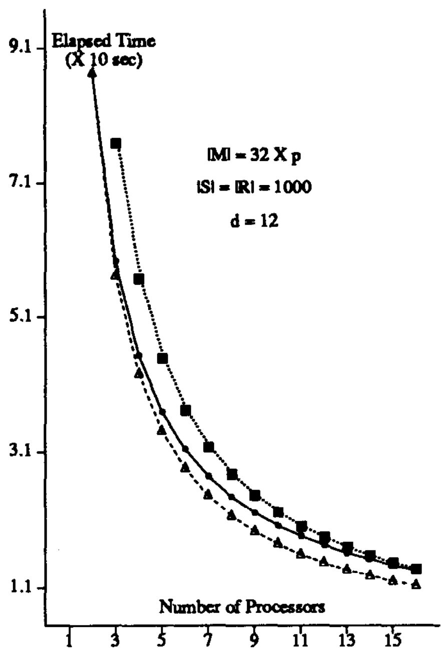
5.3b.

Figure 5.3. |M|/p = 64.



5.4b.

Figure 5.4. |S| = 4|R|.



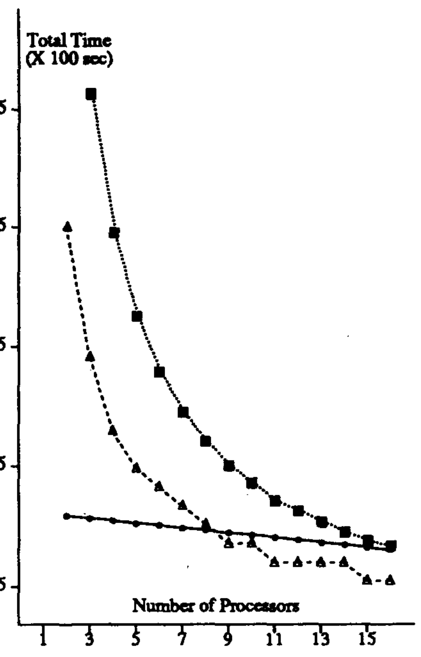


Figure 5.5. d = 12.

increases from 9 to 10 and from 11 to 14. During these two regions, the processing is I/O-bound. Thus increasing the number of processors will not decrease the elapsed time. At the same time, the increase of memory size is not large enough to reduce the number of times required to scan relation *S*. This also accounts for the better performance of SHJ than HNLJ when relation *S* is much larger than relation *R*.

While the performance of the Simple Hash Join is bounded by the other two algorithms, the Modified Hybrid Hash-Join and the Hash-based Nested Loops Join outperform each other depending on the sizes of the relations and the system configuration. We discuss the elapsed time first. The set of results presented indicates that HNLJ performs quite well with the exception of Figure 5.4. The reason is that HNLJ has better overlap in CPU and I/O processings when the number of processors is small. Most of the costs comes from the second phase of the algorithm where the CPU and I/O costs are both high. On the other hand, for MHHJ, with a small number of processors, the partitioning phase is I/O-bound while the joining phase is CPU-bound. The elapsed time is thus higher than HNLJ. When the number of processors increases, the amount of memory available increases, HNLJ performs well since the number of scannings of the entire *S* relation is reduced (as discussed above). Figure 5.4 shows that MHHJ performs much better than HNLJ when the size of relation *S* is four times the size of relation *R*. This is expected because HNLJ eliminates prior partitioning with the cost of repeated scannings of *S*. This cost is clearly shown in the results of the total processing time. Although HNLJ outperforms MHHJ with the elapsed time as the metric, it requires much more total processing time when the number of processors is small. If the total processing time is taken as the metric, then MHHJ is the best algorithm in all cases when the number of processors is limited.

In Figure 5.5, the number of disks in the system is increased to 12 from 6. This serves to represent the use of faster disks as well as slower processors. By comparing with Figure 5.2, we can see that the elapsed time in Figure 5.5 is smaller than that in Figure 5.2 for all the algorithms. However, the differences among them are smaller. This indicates that the algorithms, on the whole, are still slightly I/O bound with the testing parameters. There are some phases that are not I/O bound and hence the decrease of the elapsed time is not proportional to the increase of the number of disks.

4.3. Comparison to previous work

The performance of hash-based join algorithms have been discussed in several literatures, so we would like to compare the results of our study with two previous work — [DeWi85] and [Rich87]. Both work are chosen because overlap was considered in their studies. In [DeWi85], results of actual implementations of the uniprocessor version of the hash-based algorithms [DeWi84] were presented. In [Rich87], the analytical model considers overlap in disks, CPU and network transfers.

The results of [DeWi85] shows that the Hash Loops and the Hybrid Hash dominated the other algorithms and that the Hybrid Hash performs well in uniprocessor environment. This is also the case in our study (see Figure 5.3). Both results also show that when the size of the larger relation *S* is much larger than the size of the smaller relation R, the modified Hybrid Hash outperforms the Hash-based Nested Loops in most cases. In [DeWi85], there is no implementations of the parallel version of the Hashed Loops algorithm. However, our study shows that Hash-based Nested Loops can outperform Hybrid Hash. This is because, in a multiprocessor environment, the Hash-based Nested Loops is able to exploit the shared memory to parallelize its operation. Moreover, in our study, we assume the CREW (concurrent read exclusive write) model of a parallel system. Hence, there is no waiting when several processors read the same location in the hash table.

In [Rich87], the overlap in CPU, disk and network communications is considered. The Hybrid Hash-Join was one of the join algorithms proposed. We have shown that when CPU processing is not the bottleneck, increasing the number of processors does not reduce the elapsed times. In fact, in our study, the reduction in elapsed times of the Hybrid Hash algorithms are due to the increase in memory as a result of an increase in the number of processors. Similarly, when the disk I/O is the bottleneck, by increasing the number of disks will reduce the elapsed time. The same conclusions were arrived at in [Rich87]. However, in [Rich87], it was shown that there is not a big difference in elapsed time with different memory sizes once the memory was large enough for the algorithm to begin execution. This was so provided the first phase of the Hybrid Hash was not I/O-bound. In our study, memory size increases together with the number of processors and a large enough amount of memory is only available when the number of processors are large too. As a result, the disk I/O becomes a bottleneck. This results in the reduction in elapsed time as memory increases.

5. Conclusion

In this paper, we have exploited the shared-memory of a generalized multiprocessor system to parallelize the costly join operation in database query processing. Such a system comprises of conventional, commercially available components without the assistance of any special-purpose hardware components. The system allows concurrent read from but exclusive write to the shared-memory. Any conflict in writing the shared-memory is regulated by a locking mechanism. Moreover, each processor is allocated a fixed amount of shared-memory for each operation. Thus, the amount of memory increases as the number of processors increases.

The four parallel hash-based join algorithms designed to be executed on such an environment — Hybrid Hash, Modified Hybrid Hash, Hash-based Nested Loops and Simple Hash — were studied. The performance of the algorithms were modeled analytically, with two key features, to determine the elapsed times. First, we model the overlap between the CPU and I/O operations of each algorithm when analyzing the elapsed times. Second, we consider the contention when there is a write conflict.

Our study shows that the Hybrid Hash-Join, which outperforms other hash-based algorithms in uniprocessor environment, does not always performs the best. It is unable to exploit the memory as it is supposed to do, especially when the number of partitions is small. This is due to contention in such an environment. Our modified version — Modified Hybrid Hash — eliminates the contention by allocating one output buffer to each partition. However, this is done at the expense of higher elapsed time. The simpler Hash-based Nested Loops performs better in elapsed times when the sizes of both relations are similar. However, when the size difference between the two relations are widened, the Modified Hybrid Hash outperforms the other algorithms.

From the study, we may draw the following conclusions :

- Both the memory and the number of processors are important factors in a multiprocessor environment. More memory reduces disk I/Os in hash-based join algorithms while more processors facilitates parallelism. As such, a multiprocessor environment which increases the memory whenever the number of processors allocated to an operation increases is desirable.
- Proper management of memory may reduce memory contention. The two versions of the Hybrid Hash-Join proposed are two extreme strategies in allocating buffers during the partitioning phase — *one buffer page per partition* which may leads to high contention and *p buffer pages per partition* where there is no contention. We may explore for a balance between these two extremes.
- In a multiprocessor environment, the goal of improving the elapsed time and reducing the total processing time may conflict. An algorithm which performs well with respect to the elapsed time may do so at the expense of consuming more resources. It seems that, using the elapsed time or the total processing time as the only criteria of choosing an optimal join method may not be sufficient for multiprocessor computer systems. A more appropriate metric would need to combine the effect of both the elapsed time and the total processing time in order to simplify the query optimization process. Intuitively, if both the elapsed and total procession times of an algorithm are the smallest among all algorithms under consideration, then the algorithm should be selected. Similarly, when both the elapsed and total processing times are the largest, the algorithm should not be considered. However, when two algorithms conflict such that algorithm A may have a lower elapsed time but a higher total processing time than algorithm B, some metrics need to be defined to decide which one is better. One possible metric is the product of E and T:

$$\beta = E \times T = \frac{T}{(1/E)}$$

The β value can be roughly interpreted as the ratio of total processing cost and the system throughput ($1/E$) and it satisfy the first two points of the above intuition. In fact, if β is used as a metric, the Modified Hybrid Hash-Join outperforms the Hash-based Nested Loops in most instances.

The results of this paper can be extended in several directions. First, we may study the trade off between elapsed time and total processing time in order to arrive at a more appropriate performance metric for multiprocessor computer systems. Second, a uniform distribution of the join attribute values is assumed in our analysis. Some recent work [Laks88, Low90] has indicated that skew distributions of join attribute values may significantly affect the performance of join algorithms. One possible future work is to include this skew factor in our analysis. Third, as an analytical analysis, it is very difficult to model the effects of multiuser environment. Further study using simulation is planed to evaluate the performance of those algorithms in the multiuser environment. Furthermore, in addtion to hash-based join methods, we would like to further explore other join methods for multiprocessor computer systems, such as those methods based on index [Meno86].

References

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

[Brat84] Bratsbergsengen, K., "Hashing Methods and Relational Algebra Operations," *Proc. VLDB 84*, Singapore, Aug. 1984, pp. 323-333.

[DeWi84] DeWitt, D. J., *et al.*, "Implementation Techniques for Main Memory Database Systems," *Proc. SIGMOD 84*, Boston, June 1984, pp. 1-8.

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

[Ensl77] Enslow, P. H. Jr., "Multiprocessor Organization — A Survey," *ACM Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 103-129.

[Good81] Goodman, J. R., "*An Investigation of Multiprocessor Structures and Algorithms for Database Management*," University of California at Berkeley, Technical Report UCB/ERL, M81/33, May 1981.

[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.

[Laks88] Lakshmi, M. S. and Yu, P. S., "Effect of Skew on Join Performance in Parallel Architectures,", *Proc. Intl. Symp. on database in Parallel and Distributed Systems*, 1988, pp. 107-120.

[Lang82] Langer, A. M., and Shum, A.W., "The Distribution of Granule Accesses made by Database Transactions," *Comm. ACM*, Vol. 25, No. 11, Nov. 1982, pp. 831-832.

[Low90] Low, C. C., D'Souza, A. J., and Montgomery, A. Y., "The influence of Skew Distributions on Query Optimization," *Proc. australia Research Database, Conf.*, April 1990.

[Lu85] Lu, H., and Carey, M. J., "Some Experimental Results on Distributed Join Algorithms in a Local Network," *Proc. VLDB 85*, Stockholm, Aug. 1985, pp. 292-304.

[Lu90] Lu, H., Tan, K-L, and Shan, M-C, "*A Performance Study of Hash-Based Join Algorithms in Tightly-Coupled Multiprocessor Systems*," National University of Singapore, Department of Information Systems and Computer Science Technical Report, TRA6/90, June 1990.

[Meno86] Menon, J., "Sorting and Join Algorithms for Multiprocessor Database Machines," NATO ASI Series, Vol. F 24, Database Machines, Springer-Verlag, 1986.

[Qada88] Qadah, G. Z., and Irani, K. B., "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine," *IEEE Trans. Software Eng.* , vol. 14, no. 11, Nov. 1988, pp. 1668-1683.

[Rich87] Richardson, J. P., Lu, H., and Mikkilineni, K., "Design and Evaluation of Parallel Pipelined Join Algorithms," *Proc. SIGMOD 87*, San Francisco, May 1987, pp.399-409.

[Schn89] Schneider, D. A. and DeWitt, D. J., "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *Proc. SIGMOD 89*, Portland, Oregon, June 1989, pp. 110-121.

[Shap86] Shapiro, L. D., "Join Processing in Database Systems with Large Main Memories," *ACM Trans. Database Syst.*, vol. 11, No. 3, Sept. 1986, pp. 239-264.

[Tera83] Teradata Corporation, DBC/1012 Database Computer Concepts and Facilities, Inglewood, CA, April 1983.

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

[Yao77] Yao, S. B., "Approximating Block Accesses in Database Organizations," *Comm. ACM*, Vol. 20, No. 4, April 1977, pp. 260-261.