University of Florida

Computer and Information Science and Engineering

Performance of Grace Hash Join Algorithm on the KSR-1 Multiprocessor: Evaluation and Analysis

S. Chakravarthy
X. Zhang
H. Yokota

EMAIL: sharma@cis.ufl.edu

WWW: http://www.cis.ufl.edu/~sharma

Tech. Report UF-CIS-TR-95-030
(Submitted for publication)

October 1995

(This work is partly supported by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory.)

Computer and Information Science and Engineering Department
E301 Computer Science and Engineering Building
University of Florida, PO Box 116120
Gainesville, Florida 32611-6120

Performance of Grace Hash Join Algorithm on the KSR-1 Multiprocessor: Evaluation and Analysis

Sharma Chakravarthy **Xiaohai Zhang** **Harou Yokota***

Database Systems Research and Development Center
Computer and Information Science and Engineering Department
University of Florida, Gainesville, FL 32611

Abstract

In relational database systems, the join is one of the most expensive but fundamental query operations. Among various join methods, the hash-based join algorithms show great potential as they lend themselves for parallelization. Although performance of the hash join algorithm has been evaluated for many architectures, to the best of our knowledge, it has not been evaluated for the COMA memory architecture. This paper evaluates and analyzes the performance of hash join on the KSR-1 shared-everything multiprocessor system which has the COMA memory structure.

The following performance related issues are presented in this paper for an implementation of the Grace hash join algorithm: double buffering, data partition and distribution, potential parallelism, and synchronization. The performance evaluation results for the above are presented and analyzed. The results and analysis indicate that the ALLCACHE memory organization of KSR-1 is beneficial for parallelizing the Grace hash join algorithm.

1 Introduction

In relational database management systems (RDBMS), the join is one of the expensive but fundamental query operations. It is frequently used, computationally expensive, and difficult to optimize. During the last decade, research work has focused on developing efficient join algorithms. Consequently, various join methods are available for current database systems: nested-loops join, sort-merge join, hash-based join [EM92].

Nested-loops join is directly based on the definition of the join operation. In the nested-loops join, the source relations are named as inner and outer relations. For each tuple of the outer relation, every tuple of the inner relation is retrieved and compared with it. If the join condition is satisfied, the pair of tuples are concatenated and added to the result relation. If the cardinality of the two relations are m and n, respectively, the complexity of this algorithm is $O(mn)$. Sort-merge join first sorts the two relations on the join attributes, then scans both relations on the join attributes. Whenever a tuple from one relation matches a tuple from the other relation according to the join condition, they are concatenated as a result tuple (complexity $O(n \log n) + O(m \log m)$, including the sort phases).

Hash-based join performs the operation in a more interesting way. Typically, it is executed in two phases. First, the smaller relation is used to build a hash table based on the values of applying a hash function to the join attributes. Second, the tuples of the other relation are used to probe the hash table by means of applying the same hash function to its join attributes. Tuples from each

*School of Information Science, Japan Advanced Institute of Science and Technology, Hokuriku Tatsunokuchi, Ishikawa 923-12, Japan. Part of this work was performed while the author was visiting the University of Florida.

relation that match on join attributes are concatenated and written into the result relation. The complexity of this algorithm is $O(n + m)$. For space optimization, the smaller relation is scanned once to build the in-memory hash table, and the larger relation is also scanned once to probe the hash table.

Hash-based join has been proved to be more efficient than other join algorithms in most cases [DKO+84, Bra84, Ger86]. Historically, the sort-merge join was considered as the most efficient join method [BE77] because System R did not measure the performance of hash-based join [ABCE76]. However, since the work of [DKO+84, Bra84], hash-based join has received considerable attention: a large number of hash-based join algorithms have been proposed, implemented and evaluated. The technology makes large amounts of memory possible, which is not necessary but desirable for the hash-based join to achieve its best performance. Also, with the advent of multiprocessor systems, hash-based join shows great potential as it lends itself for parallelization.

1.1 Multiprocessor Systems

Based on their architecture, general-purpose multiprocessor systems can be categorized into shared-nothing, shared-disk and shared-everything systems. In a shared-nothing multiprocessor system, every processor has its own local memory and a set of disks that are accessible only to that processor; in a shared-disk system, every processor has its local memory but a disk may be accessed by more than one processor; and in a shared-everything system, both memory and disks are shared among the processors. Several researchers [Bhi88, BS88, LMR87] have investigated the suitability of each architecture from the view point of DBMS design. Although the shared-nothing architecture can provide high scalability, the processors can only communicate with each other through message passing, which is typically much slower than the shared-everything case. The development of software is also more difficult in a shared-nothing system because of the lack of flexibility and compatibility with conventional programming. In contrast, shared-everything architecture provides a software development environment that is very close to a uniprocessor environment. Each processor can easily communicate with the other processors via the shared memory. The synchronization of parallel operations can be achieved with little effort. Besides, most of the shared-everything systems provide automatic load balancing. These characteristics make shared-everything multiprocessor systems good candidates for parallelizing database systems.

Shared-everything systems can be further divided into three categories according to their memory structure. The first is Uniform Memory Access model (UMA), in which every processor can access each memory unit in a uniform way, no matter where the memory unit is. All processors have equal access time to all memory units [Hwa93]. The second is Nonuniform Memory Access model (NUMA), which means that the way a processor accesses a memory unit depends on the location of the particular memory unit. For example, different interconnect networks may be traversed. The third is Cache Only Memory Architecture model (COMA), in which every part of the memory is also the local cache of a certain processor. All the cache constitutes a global pool of memory. Actually, the COMA model is a special case of NUMA model. The KSR-1 all-cache memory structure described in section 2 will illustrate the COMA model. Symmetry S-81 is an example of UMA model, BBN TC-2000 Butterfly is an example of NUMA model and KSR-1 is a typical COMA machine.

There is a large body of work aimed at improving the performance of relational databases by means of parallel computing using specialized/proprietary hardware such as database machines. For example, Gamma [D+86, BDG+90], Bubba [Bor88, ABC+90], Grace [FKT86], Volcano [Gra92]

are some of the research prototypes. NonStop SQL [Tan88], TBC/1012 [Cor83] are commercial products.

1.2 Related Work

Hash-based join algorithms show great potential in multiprocessor systems because they can be easily parallelized. Many researchers have proposed parallel algorithms for hash join and studied their behavior and performance using simulation, implementation and analytical models [DKO+84, DG85, DS89, LST90, Omi91, KNiT92].

Based on an analytical model, [DKO+84] compared various query processing algorithms in a centralized database system with large memory. The results showed that hash-based algorithms outperform all the other algorithms when the size of available memory is larger than the square root of the size of involved relations. [DG85] extended the hash-based join algorithms to a multiprocessor environment. They implemented and evaluated Simple hash join, Grace hash join, Hybrid hash join and sort-merge join using the Wisconsin Storage System(Wiss) [CD+83]. The results of the performance evaluation not only verified the analytical conclusions in [DKO+84], but also showed that both Grace hash join and Hybrid hash join algorithms provide linear increases in performance when resources increase. [DS89] studied the performance of Simple hash join, Grace hash join, Hybrid hash join and sort-merge join algorithms within a shared-nothing architecture. They found that non-uniform distribution of join attribute values has a great impact on the performance of hash-based join algorithms. The Hybrid hash join algorithm was shown to dominate the others unless the join attribute values are non-uniformly distributed and the memory is relatively small. [LST90] analyzed the hash-based join algorithms in a shared-memory environment. Their analytical model considered two key features to optimize the performance: the overlap between CPU processing and I/O operations and the contention of writing to the same memory. The study concluded that the Hybrid hash join algorithm does not always outperform the other algorithms because of the contention. The authors also proposed a modified Hybrid hash join algorithm to reduce contention.

Omiecinski [Omi91] proposed a new version of parallel Grace hash join algorithm for a shared-everything environment. The modification of this algorithm was designed to improve load balancing in the presence of data skew. They implemented the modified Grace hash join algorithm on a 10 node Sequent Symmetry multiprocessor system. Both the theoretical analysis and implementation results showed that the modified algorithm provides much better performance when the data is skewed. [KNiT92] implemented and evaluated the parallel Grace hash join algorithm on a shared-everything multiprocessor system which is Sequent Symmetry S81. They exploited the parallelism with respect to I/O page size, parallel disk access, number of processors and number of buckets. The work concluded that such a shared-everything multiprocessor system has potential for building parallel database systems.

The rest of this paper is organized as follows: Section 2 describes motivation and the COMA architecture of the KSR-1 system. Section 3 presents our implementation, results obtained and analysis. Finally, section 4 gives conclusions.

2 Motivation

This section first presents the motivation for our work (see [Zha94] for more details) and the COMA architecture of the KSR-1 shared-everything multiprocessor system.

As summarized in the previous section, hash-based join algorithms have been implemented and evaluated on both shared-nothing systems, shared-disk systems, and some shared-everything systems. However, to the best of our knowledge, the performance of hash-based join on COMA multiprocessor architecture has not been evaluated. Implementation of DBMSs on multiprocessors require a good understanding of the performance of different secondary storage access methods. Hence, we believe that it is very important to understand the hash-based join algorithms in the COMA model, especially because COMA systems have high scalability (the KSR-1 can be scaled up to 1096 processors) as well as the advantages of shared-everything architecture. COMA model is one of the most suitable multiprocessor environments for databases as it combines both the features of shared-nothing and shared-everything architecture, and in addition it has fast memory access.

There are three types of hash-based join algorithms: simple hash join, Grace hash join and hybrid hash join. The Grace hash join is not only easy to parallelize, but can also handle large relations efficiently. In addition, it can be easily modified to implement other hash join algorithms. For the above reasons, we chose to implement the Grace hash join first. This paper addresses the implementation issues of parallelizing the Grace hash join on the KSR-1 multiprocessor system and presents the results of our study about the Grace hash join in the COMA model. We evaluate the performance of Grace hash join under various conditions. The analysis based on the evaluation results shows that the performance of Grace hash join can be optimized in the COMA model. The following techniques to parallelize Grace hash join are covered in this paper : processor allocation and load balancing, data Partition and distribution, I/O overlapping and buffer management.

2.1 Architecture of the KSR-1 System

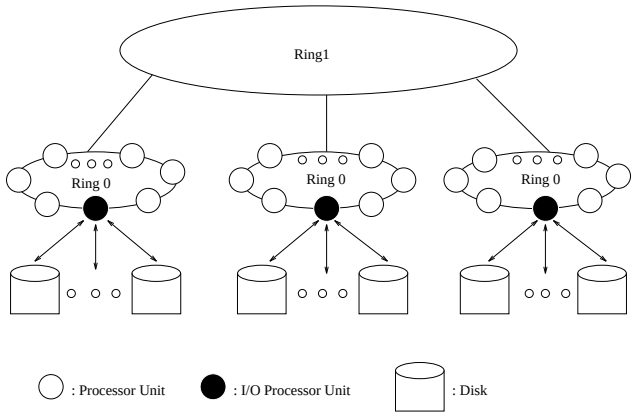

Figure 1: General Architecture of KSR-1 System

The KSR-1 is a typical shared-everything multiprocessor system with COMA structure. Combining the shared-memory architecture with high scalability, the system provides a suitable software development environment for various types of applications. Its unique all-cache memory structure enables it to run many industry-standard software systems such as standard UNIX operating system, standard programming languages (Fortran, C, Cobol) and standard networks. The system also provides automatic load balancing, which is absent in the shared-nothing systems.

The general architecture of the KSR-1 system is illustrated in Figure 1. The system (at UF) has 96 processors, which are divided into three sets. These processor sets are further organized into a ring structure. The processors in the same set are connected by one of the rings at level 0 (ring:0),

so that they can communicate with each other directly through the ring. The three rings at level 0 are further connected by a ring at level 1 (ring:1). There are communication ports between the ring at level 1 and each of the rings at level 0. The processors in different rings must communicate with each other via ring:1. The packet-passing rate of ring:0 is 8 million packets/second (1 GB/second). For ring:1, the rate can be configured as 8, 16 or 32 million packets/second (1, 2 or 4 GB/second).

In each ring at level 0, there is one processor working as the I/O processor which is capable of accessing five I/O channels in parallel, and each I/O channel may be connected up to two disks. Therefore, each I/O processor can control as many as 10 disks. The speed of the I/O channel is 30 MB/second. If other processors want to access these disks, they must communicate with the corresponding I/O processor.

Each processor has 0.5 Mbytes subcache plus 32 Mbytes local cache; all the local cache memories are managed by the all-cache engine as a huge pool of shared-memory.

2.1.1 On-Demand Data Movement

The local cache associated to each processor is divided into 2^{11} pages of 16 KB, and each page is divided into 128 subpages of 128 bytes. When a processor references an address, it first checks with the subcache, then searches the local cache if necessary. If the address is not found in the local cache, on-demand data movement occurs: The memory system allocates one page of space in the local cache, and then copies the subpage containing the referenced address into the local cache. In other words, the unit of allocation is a page and the unit of transfer/sharing is a subpage. In the memory system, the ALLCACHE engine is responsible for locating and transferring subpages among the local cache, as described in Figure 2.

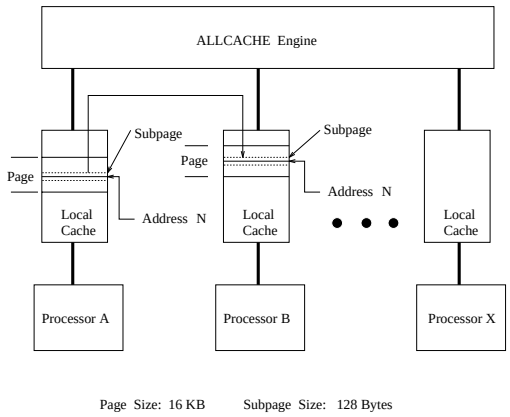

Figure 2: On-Demand Data Movement in The ALLCACHE Memory

In Figure 2, suppose processor B wants to read address N, but N is not in the local cache of processor B. Then, the ALLCACHE engine is informed to find address N within the global memory. It is assumed that N is found in the local cache of processor A. A page is allocated in the local cache of processor B, and the subpage containing address N is copied from A's local cache to B's local cache [Ken93].

If processor B needs to write address N, the ALLCACHE engine invalidates all the copies of address N. Therefore, after processor B updates the address N, the copy of N in processor B's local cache is the only valid copy. All the other processors should get a copy of N from processor B's

local cache if they need to access address N.

The all-cache memory system in KSR-1 has a hierarchical structure. The first level of the hierarchy is ALLCACHE group:0, which is comprised of ALLCACHE engine:0, local cache and processors within the same ring:0 shown in Figure 1. The second level is ALLCACHE group:1, which consists of three ALLCACHE group:0.

Although the KSR-1 system only implements two levels of this hierarchy, the structure is natural for scalability: higher level ALLCACHE group can be formed by the lower level groups. During the memory access procedure, if the referenced address is located in the local cache in the same ALLCACHE group:0 as the requesting processor, then only ALLCACHE engine:0 needs involved and data is only transferred through ring:0. If the referenced address is located in a local cache in a different ALLCACHE group:0, the ALLCACHE engine:1 will be requested to search and transfer data via ring:1.

2.1.2 Parallel I/O System

As shown in Figure 1, the disks are only connected to the I/O processors in each ring. Every other processor can access the disks but must do this via the I/O processors. At first, the accessing processor sends a I/O request to the I/O processor; upon receiving the request, the I/O processor invokes the corresponding I/O channel which will handle the I/O operation without the assistance of the I/O processor. When the I/O operation finishes, the I/O processor notifies the requesting processor. In the case of more than one processor needing to access the disks connected to the same I/O processor simultaneously, the I/O processor *serializes* the I/O requests and issues I/O commands to the corresponding channels one by one. Since the issuing of I/O commands takes very short time compared with the actual disk accessing time, the disks can almost be accessed in parallel. However, the disks bound to the same I/O channel can only be accessed serially.

3 Implementation, Results, and Analysis

In a multiprocessor environment, the performance of hash join algorithms is affected by many parameters. It is not easy to find the optimal value for each parameter. Although theoretical analysis may indicate the trends of performance with respect to changes in parameters, the actual evaluation of performance under controlled conditions is one of the best ways to understand the behavior of a hash-based join algorithm.

We implemented the Grace hash join algorithm[1] in such a way that most of the parameters can be supplied as a data file making it easy to perform experiments. These parameters include: number of processors to process each data file[2] (may be different in each phase), buffer size, range of hash values, number of buckets, number of disks, etc. Some of these parameters, such as buffer size and range of hash values are very critical to the performance of the algorithm. A naive choice

[1] The algorithm is given in the appendix for clarity but will not be included in the final version. For the partitioning phase, the hash function used is $key\ MOD\ n$, where n is the number of buckets. The second hash function used in both the building and probing phases is: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$, where k is the join attribute(s), A is a constant in the range $0 < A < 1$, and m is an integer. [Knu73] suggests that when $A = (\sqrt{5} - 1)/2 \approx 0.618034$, the hash function provides good performance. The advantage of this hash function is that the value of m is independent of the performance. Thus, we can change m to get different hash value ranges without influencing the performance.

[2] a fragment of a relation, a partition of a relation, a bucket – is a data file

of some of the parameters may hide the optimal value of other parameters. For example, if the buffer size is very small, the performance may not increase with the increase in the number of processors. Therefore, we decided to evaluate the effects of these parameters first, and conduct other experiments using the best observed values of these parameters.

Although a parallel program should provide the same final results when it runs each time, its internal execution procedure may be different from run to run. For instance, the threads created during the execution may be completed in different order. To obtain accurate results, we executed our Grace hash join algorithm with the same parameters at least 5 times, and took the average values as the final results.

The KSR-1 system provides a system function `pmon_delta` to collect performance data on a per-thread basis. This function is used for measuring time in one experiment. In this function, there is a wall_clock counter to monitor the number of elapsed clock cycles during the execution of a parallel program. This number of clock cycles is converted to obtain the elapsed time.

Typically, a parallel database is used to manage large data. It is much more meaningful to investigate the Grace hash join algorithm with relations of large size rather than small relations. The size of relations and tuples used in our experiments are similar to the one proposed in the Wisconsin benchmark [BDT83]. The join attribute is a 4-byte integer; each tuple is 208-bytes long; The source relations R and S both consist of 250,000 tuples. The join selectivity is 60%, which means that the result relation has 150,000 tuples. We use uniformly distributed data for all the experiments.

The following sections present the experimental results and their analysis, which are categorized into: buffer management, parallelism of I/O system, processor allocation, data partition and distribution.

3.1 Buffer Management

During the execution of parallel Grace hash join algorithm, each processor allocates local buffers for itself. The data stored in the disks are read into the buffers page by page. These are used as read buffers to store the data being processed. There are also write buffers when data need to be written back into the disks. For example, in the partitioning phase, each bucket has corresponding write buffer; in the probing phase, each partition file of the result relation has corresponding write buffer. The data are first stored into these write buffers. When the write buffers get full, the data are written into the disks page by page. In general, these buffers are used to avoid frequent I/O operations. Without these buffers, the I/O cost will be very high because the process of each tuple may invoke I/O operations.

In the building phase, a hash table is created for each bucket from relation R. Therefore, multiple hash tables exist at the end of building phase. These hash tables also act as buffers. The range of hash values has a great impact on the size and efficiency of these buffers. It is useful to find out an optimal range.

3.1.1 Buffer Size vs. Performance

Although the KSR-1 system provides a huge memory pool, either the read buffer or the write buffer cannot be arbitrarily large. On the other hand, large buffers do not necessarily yield good

performance. We executed the algorithm with various sizes of buffers. The parameters used for this experiment are listed below[3]:

Number of processors to partition each fragment of R and S:	5
Number of processors for each bucket file during the build phase :	5
Number of processors for each bucket file during the probe phase: (based on Figure 8)	1
Number of the disks among which R is distributed:	5
Number of the disks among which S is distributed:	5
Number of the disks among which the bucket files are distributed:	5
Number of the disks among which the result relation is distributed:	5

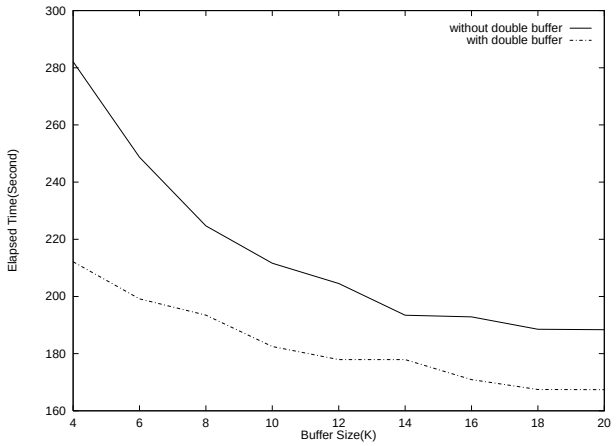

Figure 3: The Effect of Double Buffering

Figure 3 describes the relationship between elapsed time and buffer size. When the buffer size is less than 16 KB, the elapsed time decreases with the increase in buffer size. However, after the buffer size is larger than 16 KB, the elapsed time does not change too much when the buffer size increases. The curve indicates that 16 KB is an optimal size for the read/write buffer.

It is interesting that the page size of the all-cache memory of KSR-1 system is also 16 KB. We believe that this is related to the optimal buffer size in the Grace hash join algorithm. As described in section 2, the unit of allocation in the all-cache memory is a page. If the buffer size is less than 16 KB, when a processor allocates a local buffer, one page of space is allocated in its local cache regardless of the actual size of buffer. Since the buffer is smaller than one page, there is an unused portion of this page. With the increase in buffer size, the unused portion becomes smaller but there is no additional cost of allocation. The elapsed time keeps decreasing when the buffer size is smaller than the page size. When the buffer size is larger than the page size, at least two pages of space need to be allocated in the local cache. The cost of allocation is higher than allocating one page. Therefore, the elapsed time remains at a certain level with the increase in buffer size.

[3]We omit the sizes of relations R and S as they are always 250,000 tuples. The size of the resulting relation, which is always 150,000 tuples, is also omitted from the parameters in the rest of the paper. Also, the hash range used is 30,000 for all the experiments as derived from Figure 4 and hence is not shown henceforth. Each fragment (and bucket) of the same relation is on a different disk. However, fragments of R and S occupy the same disk. The same set of processors can be used in different phases. Number of processors used for each bucket of S during the probe phase is 1; the reason for this is explained in section 3.2.3

3.1.2 Range of Hash Values vs. Performance

The hash function used in the implementation allows easy changes to the range of hash values. Actually, the range of hash values implies the average number of tuples in each entry of the hash table. The larger the range, the more entries in the hash table and the less number of tuples in each entry.

The number of tuples in each entry is especially related to the elapsed time of probing phase, because the hash table is accessed only in this phase. Figure 4 shows the change of probing time with the increase in hash range. The parameters used in this experiment were the same as those shown in section 3.1.1. During the probing phase, if the hash value of a tuple from the buckets of

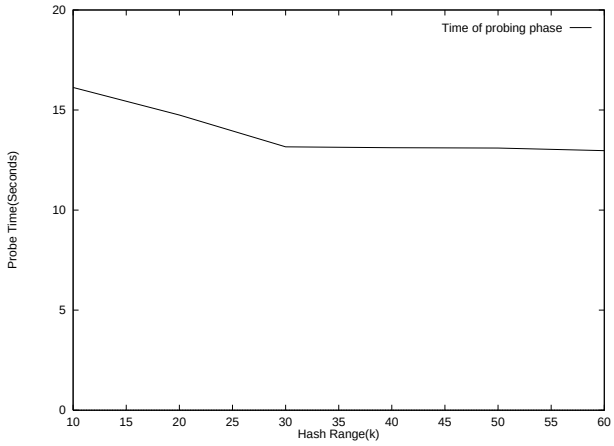

Figure 4: Range of Hash values vs. Probing Time

S maps to a non-empty entry, the tuple will be compared with the tuples in this entry to find a match. Hence, smaller number of tuples in each entry means faster search in the entries. In Figure 4, the probing time decreases with the increase in hash range while the hash range is smaller than 30K. After the hash range is larger than 30K, the probing time does not decrease with the increase in hash range. The reason is obvious. When the range of hash values reaches certain number, 30K in this case, almost all entries have only one tuple. The increase in hash range above this number only introduces more empty entries. The length of link in non-empty entry remains 1. Therefore, the search time in each entry does not change.

3.1.3 Parallelizing I/O Operations

Usually, main memory is not large enough to hold an entire relation. During each phase of the Grace hash join, data needs to be moved back and forth between disks and memory. The efficiency of I/O operations has a great influence on performance. In centralized database systems, a major objective of query optimization is to reduce I/O cost. This is also true of parallel database systems. The elapsed time of each single I/O operation is bound by the hardware design, and there is no way to reduce the latency of each I/O operation. However, there are ways to improve the performance of the I/O system as a whole, in a multiprocessor environment. First, the number of I/O operations can be minimized by appropriate algorithms. Second, it is possible to overlap the I/O operations and CPU computing. Third, I/O operations can be performed concurrently. The following experiments

are designed to investigate the proper techniques to make use of the parallel I/O system of the KSR-1.

Double Buffering: Double buffering is a common technique to overlap I/O operations and CPU computing. In the Grace hash join on the KSR-1 system, this technique is applied to each processor. Although double buffering can reduce the I/O waiting time of each processor, there is an associated cost for this option. The KSR-1 system provides asynchronous I/O ability which allows application programs to read/write data from the disks asynchronously. After issuing the asynchronous I/O command, the program can continue without waiting for the completeness of I/O operation. This function seems to be suitable for the implementation of double buffering. However, this function only works when a single processor asynchronously accesses a certain data file. The behavior of the asynchronous I/O function is uncertain when multiple processors are accessing the same file. Unfortunately, in the Grace hash join algorithm, multiple processors may be processing a particular file. Hence, the algorithm has to create a thread for each I/O operation, so that the I/O operation can be performed while the processor is processing the data in the current buffer. The initialization of each I/O thread forms the overhead of the double buffering technique. Figure 3 shows that the double buffering technique makes a great difference. The elapsed time is much shorter when double buffering is applied. The results indicate that the benefits of double buffering outweigh its overhead.

We executed the Grace hash join algorithm with and without double buffering, measuring the performance in each case. The parameters used in this experiment were the same as those shown in section 3.1.1.

Binding I/O threads: Because of the implementation of double buffering, each I/O operation is carried out by a thread. These threads run concurrently with the processing threads. At first, we thought that binding these I/O threads to the I/O processors is a good idea, since each I/O operation is accomplished by the I/O processors. However, the experiment results does not support this argument. We executed the algorithm with and without binding the I/O threads to the I/O processors. The parameters used in this experiment were the same as those shown in section 3.1.1. Figure 5 showed surprising results. Instead of improving performance, binding I/O threads keeps

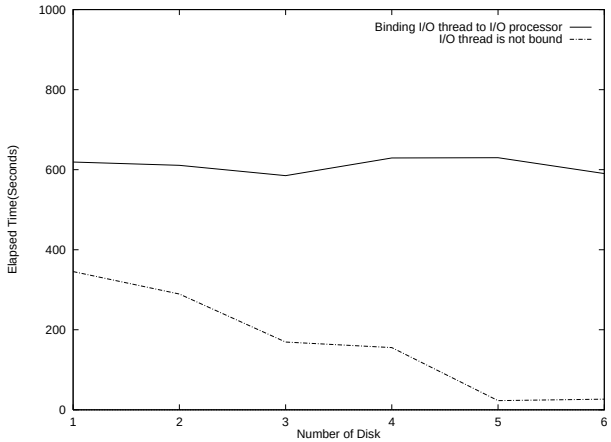

Figure 5: Binding methods of I/O threads vs. performance

the elapsed time consistently large, regardless of the increase in the number of disks. In contrast, when the I/O threads are automatically assigned to processors by the system, the elapsed time is

substantially smaller and decreases with the increase in the number of disks. To ascertain the reason for this anomaly, we observed CPU utilization when running the algorithm with bound I/O threads. It was noticed that only the I/O processor was kept busy while the other processors remained idle for most of the time. From this observation, we believe that the I/O processor was overloaded and became the bottleneck. This seems to be the major reason for degradation of performance when the I/O threads are bound to the I/O processor.

As we know, there may be up to 10 disks controlled by one I/O processor (there are five disks used in our implementation). When the algorithm is in execution, each processor creates an I/O thread for each of its I/O operations. Suppose five processors are allocated to process each data file, then there are $5 \times 5 = 25$ processors continuously creating I/O threads. When all of the I/O threads are bound to the I/O processor, the I/O processor seems to be definitely overloaded. This experiment result suggests that it is better to let the system handle the I/O threads to balance the load.

3.2 Processor Allocation and Load Balancing

The 96 processors are distributed in three separate rings in the KSR-1 system. Because communication costs across rings are much higher than the cost within one ring, choice of processor sets will lead to different communication costs. Furthermore, the KSR-1 system provides flexibility for processor allocation: automatic load balancing and various processor binding methods. The following experiments were designed to understand the processor allocation strategy among these available alternatives.

3.2.1 Processor Sets vs. Performance

When an algorithm is executed on the KSR-1 system, the processors that are allocated for that execution may be restricted to a subset, rather than all the processors in the system. The subset may be within one ring or across multiple rings. Without this restriction, all the processors are available for the algorithm, and each thread of the algorithm is assigned to a specific processor automatically.

Suppose the three rings in the system are denoted as R_a, R_b, R_c, and the source relations R and S reside in the disks which are controlled by the I/O processor in R_b. In order to find out the impact of different processor sets on the performance of Grace hash join algorithm, we conducted the following experiment: First, we restricted the processor set to R_b and executed the algorithm; then we restricted the processor set to R_a and R_c and executed the algorithm. The parameters used in this experiment were the same as those shown in section 3.1.1. From Figure 6, we observe that the elapsed time is smaller when the processor set is restricted to R_b. That means, using the processors in R_b achieves better performance. An obvious reason for this result is the difference in communication cost. When the processor set consists of the processors in both R_a and R_c, there is data exchange between R_a and R_b, or between R_c and R_b. Since the relations are stored in the disks connected to R_b, the processors in R_a and R_c need to communicate with the I/O controller in R_b when they want to access the data files. The communication is across multiple rings and involves ring:1. Thus, the communication cost is relatively high. In contrast, when the processor set is restricted in R_b, although the processors still need to communicate with the I/O controller to access the disks, all communication is within this ring and has lower cost.

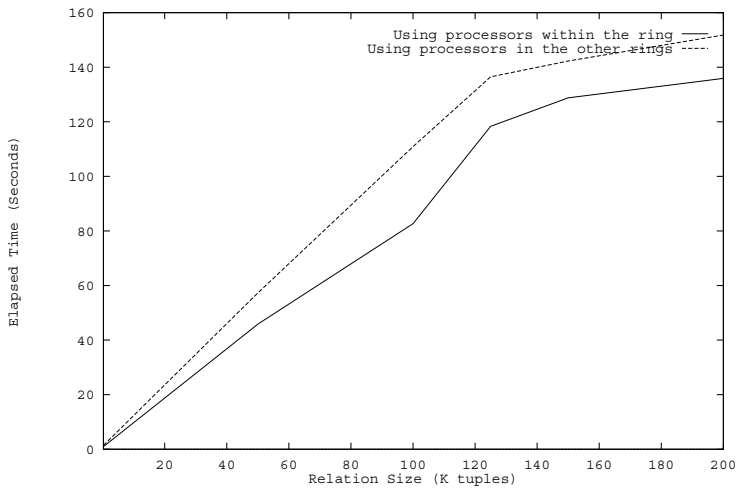

Figure 6: The Impact of Different Processor Sets

In Figure 6, with the increase in relation size, the difference in elapsed time increases. Larger relation sizes imply more I/O operations should be performed, and each I/O operation involves the communication between the processing processor and I/O processor. However, as we know, the processors in both R_a and R_c have to communicate with the I/O processor in R_b via ring:1 while the processors in R_b can communicate with the I/O processor directly. When the relation size increases, the cumulative difference of communication cost gets larger. This is why the curves in Figure 6 diverge.

Based on the above result and analysis, we can conclude that the location of the processor set is critical to the performance of Grace hash join on the KSR-1 system. If the data is stored in the disks connected to a particular ring, it is more efficient to use the processors within that ring. When data is distributed across multiple rings, the communication cost is still a major issue in processor allocation.

3.2.2 Load Balancing vs. Performance

One feature of the KSR-1 system is its automatic load balancing, which is achieved in a dynamic way. During the execution of a parallel algorithm, the threads can migrate from one processor to another in order to maintain a balanced load among the processors. The operating system is responsible for migrating threads. By default, parallel programs are executed on the KSR-1 with automatic load balancing. However, it is also possible to bind a thread to a specific processor. When a thread is bound to a processor, its migration is prohibited. Each processor only handles the jobs which are assigned to it at the beginning of execution. "fixed-binding" is one of the functions which performs processor binding. It binds the calling thread to a processor which is chosen by the system. With this function, the users may have their own load balancing strategy.

To understand the load balancing strategy for the Grace hash join algorithm, we first executed the algorithm with automatic load balancing, and then executed it using "fixed-binding" to prevent the migration of threads. The parameters used in this experiment were the same as those shown in section 3.1.1.

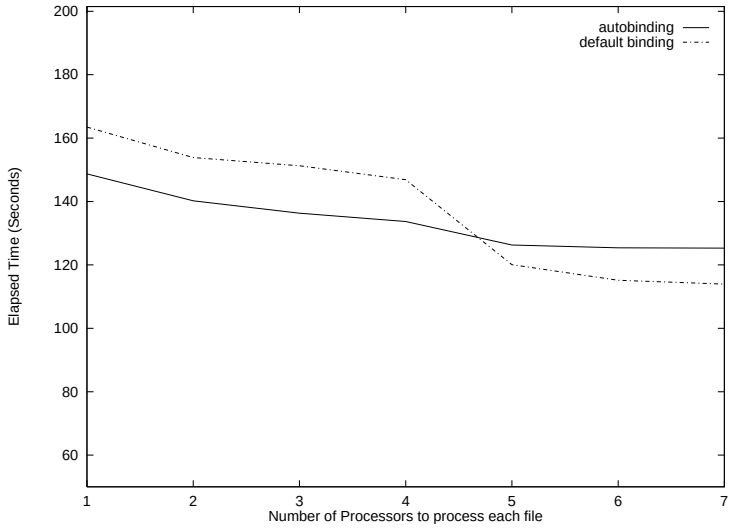

Figure 7: Load Balancing strategy vs. Elapsed Time

Figure 7 shows that autobinding (that is fixed-binding of a thread to a processor) outperforms automatic load balancing (termed default binding) when the number of processors is less than or equal to 4, but vice versa when the number of processors is larger than 4. The reason for this phenomenon is twofold. First, there is overhead caused by the automatic load balancing. This overhead contributes to the difference in elapsed time when only one processor is used. Second, automatic load balancing needs to migrate the threads among processors. Migration requires additional communication which is absent in the fixed-binding case. When the processor set is small, the automatic load balancing mechanism does not have much opportunity to improve the performance, because there are few alternatives available. Therefore, the overhead and additional communication cost outweigh the benefits of automatic load balancing. In contrast, while the processor set is large enough, automatic load balancing can take full advantage of run time information about resource utilization, especially the status of processors. Usually, a much better processor allocation plan can be found than in the fixed-binding case. As a consequence, the overhead and additional communication costs are compensated by the effect of good load balancing.

In our Grace hash join algorithm, the source relations R and S are divided into buckets of almost equal size during the partitioning phase. This is a major reason for the good performance of fixed-binding strategy when the number of processors is small. If the source relations are not partitioned in a uniform way, the fixed-binding method may probably lead to extremely unbalanced load for the processors. Elapsed time will be much larger since it is the execution time of the processor with the heaviest load. Hence, we cannot conclude that the fixed-binding strategy is better than automatic load balancing in the case of a small processor set. The fixed-binding strategy is only suitable for Grace hash join when the processor set is small and the partitioning phase has ideal results. However, in practice, it is not easy to divide a job into small jobs with the same size. In conclusion, we believe that automatic load balancing is a better and safer choice for the Grace hash join algorithm on the KSR-1 system.

3.2.3 Number of Processors vs. Performance

Another issue of concern for this algorithm is the number of processors used to process each data file. Since there are three phases in the Grace hash join algorithm, we investigate this for each phase. Because the previous results and analysis show that automatic load balancing outperforms the other load balancing strategy, we adopt automatic load balancing in the following experiments. The parameters used in this experiment were the same as those shown in section 3.1.1.

Figure 8 shows the elapsed time of the partitioning phase vs. the number of processors used. While the number of processors is less than five, partitioning time decreased rapidly with the increase in processor numbers. But when the number of processors is greater than five, there is no substantial change in partitioning time. This result indicates that five processors is appropriate for catching up with the data flow from one disk. Figure 8 also shows the relationship between the

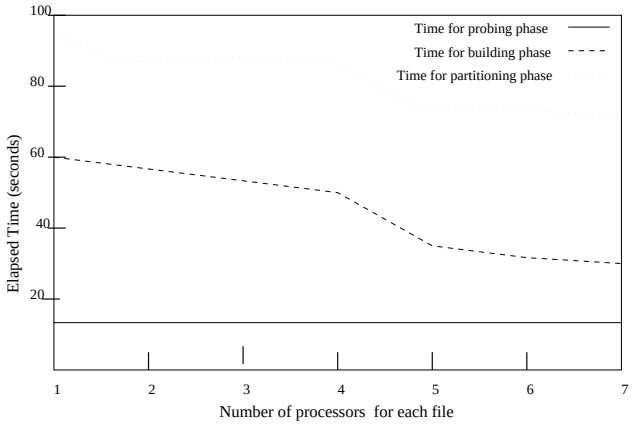

Figure 8: Number of Processors vs. Time for individual phases

number of processors and the time to build the hash table. The elapsed time of the building phase remains almost the same after the number of processors are greater than five. Five processors seem to be good enough to build the hash table from one data file. The probing time is also shown in Figure 8 along with the other two. The result is not the same as that of partitioning time and building time. The number of processors has little impact on the elapsed time of the probing phase. This implies that one processor is enough for each data file during the probing phase.

In the above experiments, we observed that there is no linear speed-up as the number of processors increases. When the processor set is large enough, the performance remains almost the same regardless of the number of processors. Many factors may contribute to this scenario. First, more processors indicate that more threads need to be created, and the cost to initialize a thread is high. Second, more processors also cause more lock contention. During the execution of the Grace hash join algorithm, there are locks for data files, hash tables and write buffers. These critical sections may become bottlenecks when too much contention occurs.

3.3 Data Partition and Distribution

Data partition and distribution is necessary for parallel I/O operations in a multiprocessor system. During each phase of the Grace hash join algorithm, the relations are divided into multiple data files which are written to different disks. Even before the execution of the algorithm, the source relations

R and S need to be partitioned and distributed into multiple disks. Otherwise, the partitioning phase of Grace hash join cannot be performed in parallel. The partitioning phase generates buckets which are separate data files. The buckets can be regarded as a new partition of R and S. After the probing phase, the result relation is also generated as multiple files. In the following experiments, the number of data files for each relation expands when the number of disks increases.

3.3.1 Number of Disks vs. Performance

The number of disks is a very important parameter for the parallel Grace hash join algorithm. Since I/O operations are much slower than CPU operations, the algorithm spends most of its time waiting for I/O operations. More disks will increase the parallelism of I/O operations, which is critical for the algorithm to achieve better performance.

We experimented with different numbers of disks and compared the elapsed time for each case. The parameters used in this experiment were the same as those shown in section 3.1.1. Figure 9

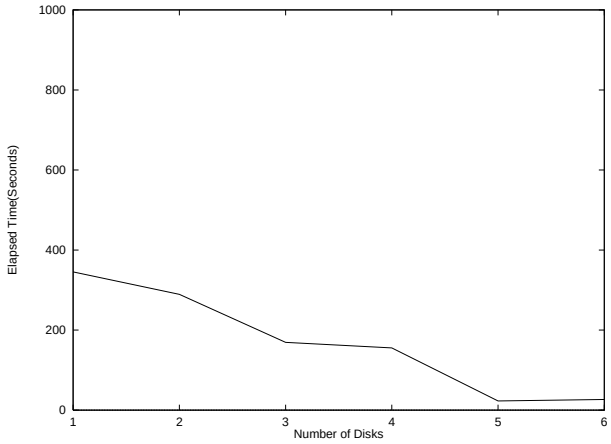

Figure 9: Number of Disks vs. Elapsed Time

shows the decrease in elapsed time with the increase in the number of disks. The elapsed time decreases rapidly when the number of disks is less than five. However, when the number of disks is larger than five, the elapsed time does not change so much with the increase in disk number. The results show that I/O operations are not the bottleneck when the files are distributed into five disks. In this case, the data files are relatively small, so that most of the I/O operations are performed concurrently in each disk. The time spent on other operations such as lock contention and CPU calculation seems to dominate the elapsed time. The increase in disk numbers cannot reduce this part of elapsed time. We should point out that the sixth disk shown in Figure 9 is not connected to the same ring as the previous five disks[4]. To read/write the data files in this disk results in more communication costs. This additional cost also degrades performance.

3.3.2 Data Distribution vs. Elapsed Time

As described earlier, the disks of the KSR-1 system are connected to the I/O processors within different rings. The disks associated to each ring constitute a disk group. The data files may

[4]There are only five disks mounted in this ring currently.

be distributed within one disk group or across multiple disk groups during the execution of the parallel Grace hash-based join. In order to understand the effect of distributing data among disks in different groups (rings), we conducted the following experiment: Let the processor set consist of all of the available processors in the system, and executed the algorithm with different data distribution strategies: distributing data across three rings, across two rings and within one ring. The parameters used are[5]:

Buffer size:	16 KB
Number of processors to partition each data file:	5
Number of processors for each bucket file during the build phase :	5
Number of processors for each bucket file during the probe phase:	1
Number of the disks among which R is distributed:	4
Number of the disks among which S is distributed:	4
Number of the disks among which the bucket files are distributed:	4
Number of the disks among which the result relation is distributed:	4

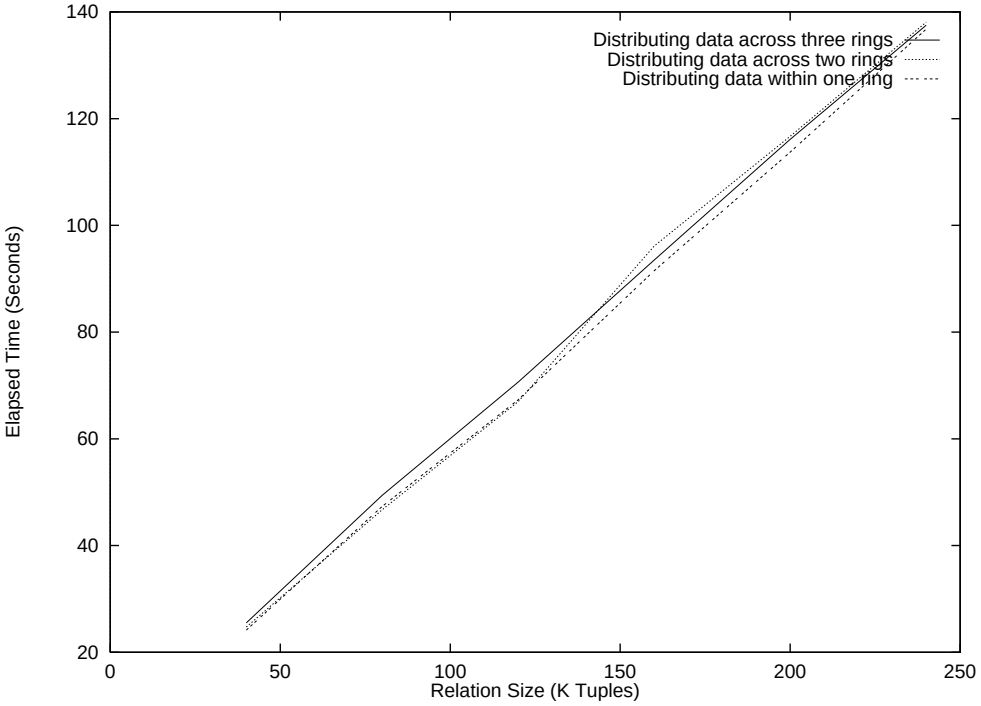

Figure 10: Data Distribution vs. Elapsed Time

Figure 10 shows the elapsed time in each case when the relation size increases. We noticed that there is no substantial difference among the elapsed time in each case. Although there is a slight difference, it is caused by the internal difference of each run of the parallel algorithm. The results reveal the strong ability of the parallel I/O system in the KSR-1. When the data files are distributed among three rings, the I/O operations are carried on by three I/O processors concurrently. The load of each I/O processor is relatively low compared with the case when all the data files are

[5]The reason for using 4 disks instead of 5 in this experiment is that only 2 disks can be accessed concurrently from within each ring. If we used 5 disks, it would be across all three rings preventing us to evaluate the performance of distribution of data across 2 rings.

distributed within one ring. Although the same amount of I/O operations are processed by one I/O processor in the latter case, there is no difference in performance. This indicates that I/O processors are not usually overloaded by simultaneous I/O requests. The parallel I/O system in the KSR-1 provides good performance in each case.

In the above experiment, the processor set consists of all the processors in the three rings. If the processor set is restricted within one ring, then it would be better to distribute the data only within the disks associated with that ring. The idea is to avoid high communication costs across multiple rings. We perhaps can conclude that if the processor set consists of the processors in rings $R_1, \ldots, R_n$, then data can be freely distributed among $R_1, \ldots, R_n$.

4 Conclusions

This paper presents the performance evaluation and its analysis for the first time for the COMA model and for KSR-1 multiprocessor system. The analysis corresponding to each experiment reveals the underlying reasons for the results. The following conclusions can be drawn from the experimental results and their analysis presented in this paper:

- The cost of communication across multiple rings (in KSR-1) is higher than the communication cost within each ring. The processors from within the ring should be used to process the local data files in order to reduce the communication cost.
- The automatic load balancing mechanism of the KSR-1 system is suitable for the parallel Grace hash join algorithm. The automatic management of I/O threads in the KSR-1 system gives good performance.
- In each phase of Grace hash join algorithm, there is an optimal number of processors for each data file (a data file is either a fragment or a partition or a bucket). In the partitioning phase, five processors seem to be good enough to partition one fragment of a a relation; in the building phase, five processors is suitable for building the hash table from each data file; in the probing phase, one processor is sufficient to probe each hash table.
- To process two relations which have the size of 250,000 tuples each, five disks are sufficient to take full advantage of the parallel I/O system of the KSR-1.
- The data can be distributed among the rings from which the processor set is constituted.
- Double buffering technique does improve the performance of parallel Grace hash join substantially on the KSR-1 system.
- The page size of all-cache memory of the KSR-1 system is also the optimal processing buffer size of parallel Grace hash join.
- The optimal range of hash values is 30K in the experiments.

These conclusions indicate that the KSR-1 system provides a good environment for parallelizing Grace hash-based join algorithm. In most cases, the internal system mechanisms directly support the parallelism of the Grace hash join. For instance, the ring structure provides the locality and scalability; the all-cache engine guarantees the quick reference of memory and the easy implementation of double buffering; the automatic load balancing mechanism optimizes the allocation of processors; the parallel I/O system can handle the I/O operations efficiently.

References

[ABC+90] W. Alexander, H. Boral, L. Clay, G. Copel, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Transaction on Knowledge and Data Engineering*, pages 4–24, March 1990.

[ABCE76] M. Astrhan, M. W. Blasgen, D. D. Chamberlin, and P. Eswaran. System r: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):119–120, June 1976.

[BDG+90] A. Bricker, D. J. DeWitt, S. Ghandeharizaeh, H. I. Hsiao, and D. A. Schneider. The gamma database machine project. *IEEE Transaction on Knowledge and Data Engineering*, 2(1):44, March 1990.

[BDT83] D. Bitton, D. DeWitt, and C. Tubyfill. Benchmarking database systems - a systematic approach. In *Proceedings of International Conference on VLDB*, Florence, Italy, 1983.

[BE77] M. W. Blasgen and K. P. Eswaran. Storage and access in relational databases. *IBM System Journal*, 16(4):21–33, 1977.

[Bhi88] A. Bhide. An analysis of three transaction processing architectures. In *Proceedings of International Conference on VLDB*, page 339, Long Beach, CA, August 1988.

[Bor88] H. Boral. Parallelism in bubba. In *Proceedings of International Symposium on Database in Parallel and Distributed Systems*, Austin, TX, December 1988.

[Bra84] Kjell Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of International Conference on VLDB*, Singapore, August 1984.

[BS88] A. Bhide and M. Stonebraker. A performance comparison of two architectures for fast transaction processing. In *Proceedings IEEE Conference on Data Engineering*, page 536, Los Angeles, CA, February 1988.

[CD+83] H-T Chou, , D. J. DeWitt, R. Katz, and T. Klug. Design and implementation of the wisconsin storage system. Technical Report 524, University of Wisconsin, November 1983.

[Cor83] Teradata Corp. Dbc/1012 data base computer concepts & facilities. Technical Report C02-0001-00, Teradata Corp., 1983.

[D+86] D. DeWitt et al. GAMMA – A High Performance Backend Database Machine. In *Proceedings 12th International Conference on Very Large Data Bases*, Aug. 1986.

[DG85] David J. DeWitt and Robert Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of International Conference on VLDB*, Stockholm, 1985.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *Proceedings of ACM SIGMOD Conference*, Boston, MA, 1984.

[DS89] David J. DeWitt and Donovan A. Schneider. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of ACM SIGMOD Conference*, Portland, OR, June 1989.

[EM92] Margaret H. Eich and Priti Mishra. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63, March 1992.

[FKT86] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine grace. In *Proceedings of International Conference on VLDB*, Kyoto, Japan, August 1986.

[Ger86] R. J. Gerber. Dataflow query processing using multiprocessor hash-partitioned algorithms. Technical Report 672, University of Wisconsin,, Madison, WI, 1986.

[Gra92] G. Graefe. Volcano, an extensible and parallel dataflow query processing system. *IEEE Transaction on Knowledge and Data Engineering*, pages 14–21, June 1992.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., Reading, MA, 1993.

[Ken93] Kendall Square Research, Boston, MA. *KSR1 Principles of Operation*, 1993.

[KNiT92] Masaru Kitsuregawa, Miyuki Nakano, and Shin ichiro Tsudaka. Parallel grace hash join on shared-everything multiprocessor: Implementation and performance evaluation on symmetry s81. *IEEE 8th International Conference on Data Engineering*, 1992.

[Knu73] Donald E. Knuth. *Sorting and Searching, volume 3 of The Art of Computer Programming.* Addison-Wesley, 1973.

[LMR87] H. Lu, K. Mikkilineni, and J . P. Richardson. Design and evaluation of parallel pipelined join algorithms. In *Proceedings of ACM SIGMOD Conference*, San Francisco, CA, May 1987.

[LST90] Hongjun Lu, Ming-Chien Shan, and Kian-Lee Tan. Hash-based join algorithms for multiprocessor computers with shared memory. In *Proceedings of International Conference on VLDB*, Brisbane, Australia, 1990.

[Omi91] Edward Omiecinski. Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. In *Proceedings of International Conference on VLDB*, Barcelona, September 1991.

[Tan88] Performance Group Tandem. A benchmark of non-stop sql on the debit credit transaction. In *Proceedings of ACM SIGMOD Conference*, Chicago, IL, June 1988.

[Zha94] X. Zhang. Performance evaluation and analysis of grace hash-join algorithm on a ksr-1 multiprocessor system. Master's thesis, University of Florida, July 1994.

A Grace Hash Join Algorithm

Let R and S be two relations participating in the join, and the size of R be smaller than that of S. The most straightforward hash join algorithm works as follows: applying a hash function to R's join attributes, build an in-memory hash table from R; then use each tuple of S to probe the hash table. Whenever a match occurs, the matching tuples are output. Usually, each hash join algorithm is executed in two phases:

Simple hash Join algorithm combines the partitioning work and probing work into each iteration of the loop. In contrast, the Grace hash Join algorithm executes the partitioning phase and joining phase separately. In the partitioning phase, Both R and S are partitioned into an equal number of buckets; in the joining phase, each pair of corresponding buckets are joined and the result relation is formed by concatenating the results of each separate join. These two phases are very similar to the two phases presented at the beginning of this section as the general description of hash-based join algorithms. The Grace hash Join algorithm can be described as follows:

```
/* R[i](i=1..n) and S[i](i=1..n) are buckets */
for (each tuple r in R relation){
          apply hash function to the join attributes of r;
          put r into the appropriate bucket R[i];
}
for (each tuple s in S relation){
          apply hash function to the join attributes of s;
          put r into the appropriate bucket S[i];
}
for (i=1;i<=n;i++){
          build the hash table from R[i];
          for (each tuple s in S[i]){
                    apply hash function to the join attributes of s;
                    use s to probe the hash table;
                    output any matches to the result relation;
          }
}
```

In this algorithm, all the tuples only need to be written back into disk once. When the memory is not large, the I/O overhead is greatly reduced compared with the Simple hash Join algorithm. Therefore, this algorithm performs much better than the Simple hash join algorithm under most circumstances. From the above description we can also see that the partitioning phase and joining phase are completely disjoint in the Grace hash join. This feature avoids bucket overflow: in the partitioning phase, increase the number of buckets to guarantee that each bucket fits into the available memory; in the joining phase, integrate multiple buckets into a set of larger buckets which have the maximum size to fit into the memory. This techniques is termed bucket tuning. Another advantage is that during the partitioning phase, R and S can be partitioned concurrently. These features make it easy to split the join into many smaller operations. These operations can be assigned to different processors with little data dependence in the multiprocessor environment.