PERFORMANCE EVALUATION OF GRACE HASH-JOIN ALGORITHM ON THE KSR-1 MULTIPROCESSOR SYSTEM

By

XIAOHAI ZHANG

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1994

Dedicated to my
Parents

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for showing me the path of research with his enlightening advise and expert guidance, and for his continuous support and constant encouragement throughout the course of this research work.

I am extremely grateful to Dr. Andrew Francis Laine and Dr. Herman Lam for agreeing to serve on my committee and for their careful perusal of this thesis. Thanks also go to Dr. Haruo Yokota at Japan Advanced Institute of Science and Technology for his insightful suggestions and comments.

I would like to thank Randy Fischer for his various help and suggestions, and for providing me all the resources I requested on the KSR-1 system.

I would like to thank Mrs. Sharon Grant for maintaining a well administered research environment and making numerous copies of papers for me.

I will also take this opportunity to thank all the graduate students at this center for their help and friendship.

Last, but not the least, I thank my parents and brother for their endless love. Without their encouragement and endurance, this work would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Master of Science

PERFORMANCE EVALUATION OF GRACE HASH-JOIN ALGORITHM ON THE KSR-1 MULTIPROCESSOR SYSTEM

By

Xiaohai Zhang

December 1994

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Sciences

The performance of relational databases is being challenged by the increasing data size they have to deal with. Due to the limited capability of uniprocessor systems and the reduction of hardware cost, multiprocessor systems may provide a viable alternative for meeting the requirements of applications with very large databases. This thesis presents our preliminary work on the parallel database research within the COMA shared-everything multiprocessor environment. In the relational database systems, join is one of the most expensive but fundamental query operations. Among various join methods, the hash-based join algorithms show great potential because they lend themselves for parallelism. This thesis describes our implementation and performance evaluation of the GRACE hash-based join algorithm on the KSR-1 shared-everything multiprocessor system which has the COMA memory structure. Various join methods are analyzed and compared with respect to their efficiency and suitability for the multiprocessor environment. The following implementation issues are addressed: double buffering, data partition and distribution, potential parallelism, synchronization and hash function. Finally, the performance evaluation results and the corresponding analysis are presented. The results and analysis indicate that the

KSR-1 system provides a good environment for parallelizing the GRACE hash-based join algorithm. In addition, the work also shows that the architecture proposed by G. Graefe in 1992 possesses great potential for parallel database development.

CHAPTER 1
INTRODUCTION

In relational database management systems (RDBMS), join is one of the expensive but fundamental query operations. It is frequently used, computationally expensive and difficult to optimize. During the last decade, a lot of research work has been 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 [17].

1.1 Join Algorithms in Relational Database Systems

Nested-loops join comes directly from the definition of join operation. In the nested-loops join, the source relations are named as inner and outer relations. For each tuple of the outer relation, all tuples of inner relation are retrieved and compared with it. If the join condition is satisfied, the pair of tuples are concatenated and put into the result relation. 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 another tuple from the other relation according to the join condition, they are concatenated as a result tuple.

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 relation that match on join attributes will be concatenated and written into the result relation (Chapter 2 describes hash-based join algorithms in detail).

The advantage of nested-loops join is simple. However, since it compares the tuples from two relations exhaustively, it is not efficient under most circumstances. When joining two large relations and the selectivity factor is extremely low, most of comparisons are wasteful as they do not generate result tuples. If the cardinality of the two relations are m and n, respectively, the complexity of this algorithm is *O(mn)*. In contrast, sort-merge join outperforms the nested-loops join while the relations are presorted, since both relations need to be scanned only once to complete the join operation. In the case of low selectivity factor, the number of comparisons is significantly reduced. But if the relations are not sorted before they participate in the join, the complexity includes the cost to sort both relations, which is $O(n \log n) + O(m \log m)$, where m, n are the cardinalities of the source relations. Hash-based join also improves the performance via reducing the number of comparisons. During the probing phase, a tuple from the larger relation only needs to be compared with the tuples from the smaller relation which have the *same hash value*. The tuples which have different hash values do not match. The scope of comparison for each tuple from the larger relation is then reduced from the entire smaller relation to a small subset. The complexity of this algorithm is $O(n + m)$, since 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 [26, 15, 8, 19]. However, suprisingly, most of the existing database systems only use the nested-loops join and sort-merge join. Historically, the sort-merge join is considered as the most efficient join method [6]. The main reason for this is that System R did not measure the performance of hash-based join [2]. However, since the work [15, 8], hash-based join has received considerable attention: a large number of hash-based join algorithms have been proposed, implemented and evaluated. The emergence of interest in hash-based join is driven by the rapid hardware development. First, the technology makes large amounts of memory possible, which is not necessary but desirable for the hash-based join to achieve its best performance. Second, with the trends moving towards multiprocessor system, hash-based join shows great potential as it lends itself for parallelism.

1.2 Multiprocessor Systems and Database Design

Multiprocessor systems are being used nowadays, mostly because of their high performance and relatively low cost. In order to improve the performance of the relational databases by means of parallel computing, specialized hardware such as database machines were proposed and implemented in the early stages of parallel database systems. For example, Gamma at University of Wisconsin [13, 9], Bubba at MCC [7, 1], GRACE in Japan [18, 26], Volcano at University of Colorado at Boulder [20, 21] are some of the research prototypes, and NonStop SQL from Tandem [33], TBC/1012 from Teradata [12] are commercial products. Although these systems provide high performance, they use propriety hardware. The high cost to design and build such database machines prevent them from becoming popular in the real world. Thus, the trend has been gradually towards developing parallel database system using a general-purpose multiprocessor environment.

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 each disk is accessible to only one processor; and in a shared-disk system, every processor has its local memory but a disk may be accessed by more than one processors; in a shared-everything system, both memory and disks are shared among the processors.

Several researchers [32, 3, 4, 29] 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. Shared-everything architecture provides a more comfortable software development 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, the shared-everything systems provide automatic load balancing. Therefore, we believe that shared-everything multiprocessor systems are 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 [23]. 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 certain processor. All the caches constitutes a global pool of memory. Actually, the COMA model is a special case of NUMA model. The KSR-1 all-cache memory structure in chapter 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.

1.3 Motivation and Contribution

Hash-based join algorithms have been implemented and evaluated on both shared-nothing systems and shared-everything systems. Gamma project has carefully studied the performance of hash-based join on shared-nothing architecture [16, 14]. The behavior of hash-based join on shared-everything has been investigated [27, 30]. Actually, Lu et al. [30] evaluated the hash-based join based on simulation, where the lack of accuracy is unavoidable; and Kitsuregawa et al. [27] implemented and evaluated the hash-based join on Symmetry S81, which is classified as UMA multiprocessor system. The performance of hash-based join on COMA multiprocessor architecture is still under question. We feel it is very useful to understand the hash-based join algorithms in the COMA model, because the COMA systems have high scalability besides the advantages of shared-everything architecture (for example, the KSR-1 can be scaled up to 1096 processors). COMA model is one of the most suitable multiprocessor environment for databases as it Combines both the features of shared-nothing and shared-everything architecture, and in addition it has fast memory access. Graefe [22] proposed a general hierarchical architecture which is believed to have the advantages, such as efficient communication, sychroniztion, automatic load balancing, high scalability and reliability (Figure 1.1). These features are very essential for a parallel database system. The paper only gave a general description of this architecture and many details were left open. However, we believe that the architecture of COMA system KSR-1 is very similar to the proposed architecture (this issue will be discussed further in Chapter 2). It is interesting to verify the advantages of this architecture by actually implementing database algorithms on the KSR-1 system. These factors motivated our research on the parallelism of hash-based join within the COMA model and KSR-1 architecture.

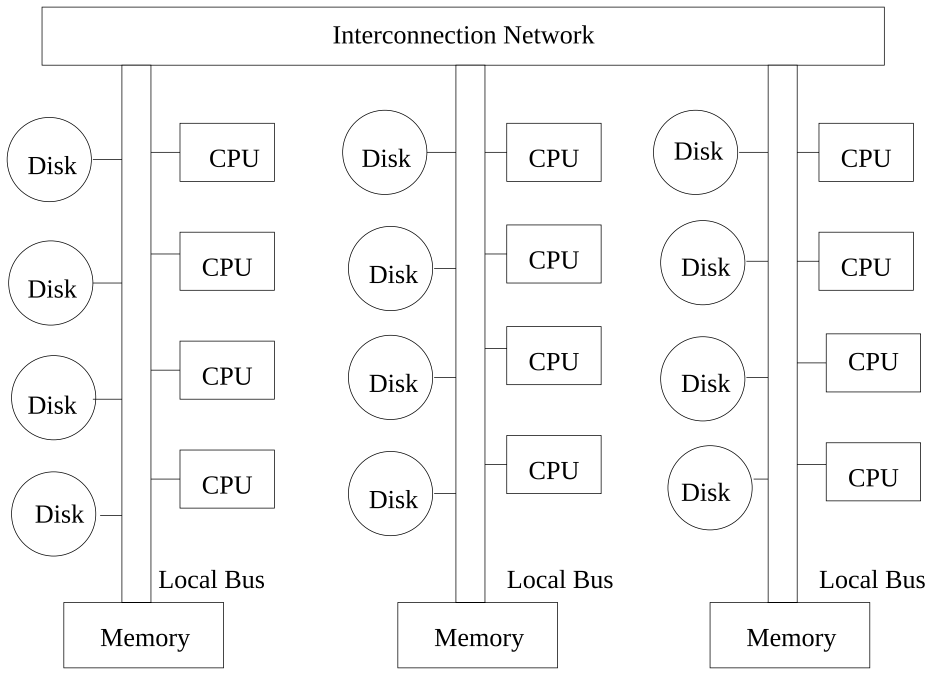


Figure 1.1. A General Hierarchical Architecture

There are three types of hash-based join algorithms: Simple hash-join, GRACE hash-join and Hybrid hash-join (their difference will be addressed in chapter 2). The GRACE hash-join is not only easy to parallelize, but also able to handles large relations efficiently. In addition, it can be easily modified to implement other hash-join algorithms. Therefore, we implemented the GRACE hash-join in our first stage of study. This paper addresses some implementation issues of parallelizing the GRACE hash-join on the KSR-1 multiprocessor system (which has COMA structure) and presents the results of our study about the GRACE hash-join in the COMA model. We evaluated the performance of GRACE hash-join under various conditions. The analysis based on the evaluation results is aiming to show how to optimize the performance of GRACE hash-join in the COMA model. The following techniques to parallelize GRACE hash-join are covered in this thesis: processor allocation and load balancing, data Partition and distribution, I/O overlapping and buffer management.

The rest of this thesis is organized as follows: Chapter 2 describes the hash-based join algorithms, the COMA architecture of the KSR-1 system, and gives an overview of related work. Chapter 3 addresses some implementation issues. Chapter 4 presents the evaluation results and analysis. Finally, Chapter 5 gives conclusion and discusses some future work.

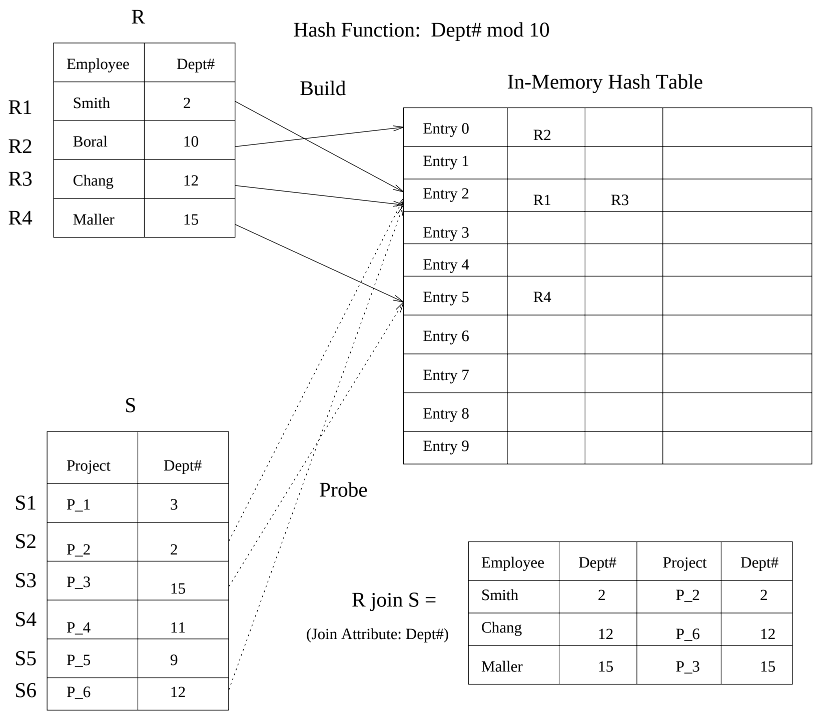
CHAPTER 2
SOME BACKGROUND: ALGORITHMS AND ARCHITECTURE

This chapter presents various hash-based join algorithms, the COMA architecture of the KSR-1 shared-everything multiprocessor system and some related work. Section 2.1 discusses the general strategy of hash-based join and describes three major hash-based join algorithms; section 2.2 presents the architecture of the KSR-1 system, with the focus on its all-cache memory structure and parallel I/O system; Section 2.3 provides an overview of related work.

2.1 Hash-Based Join Algorithms

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. An example of this basic hash-join is shown in figure 2.1.

As illustrated in figure 2.1, each tuple of R is put into the hash table according to the value obtained after applying the hash function to the join attribute, which is Dept# in this example. If more than one tuple have the same hash value, they are linked together in the same entry (R_1 and R_3 in entry 2). This is the procedure to build the hash table. After having built the hash-table, for each tuple of S, the same hash function is also applied to the join attribute. If the hash value indicates an empty entry in the hash-table, This tuple is dropped; otherwise, the tuple is compared with the tuples in the corresponding entry. If a match is found, the pair of



	Employee	Dept#
R1	Smith	2
R2	Boral	10
R3	Chang	12
R4	Maller	15

Entry 0	R2		
Entry 1			
Entry 2	R1	R3	
Entry 3			
Entry 4			
Entry 5	R4		
Entry 6			
Entry 7			
Entry 8			
Entry 9			

	Project	Dept#
S1	P_1	3
S2	P_2	2
S3	P_3	15
S4	P_4	11
S5	P_5	9
S6	P_6	12

Employee	Dept#	Project	Dept#
Smith	2	P_2	2
Chang	12	P_6	12
Maller	15	P_3	15

Figure 2.1. An Example of Basic Hash-Join Algorithm

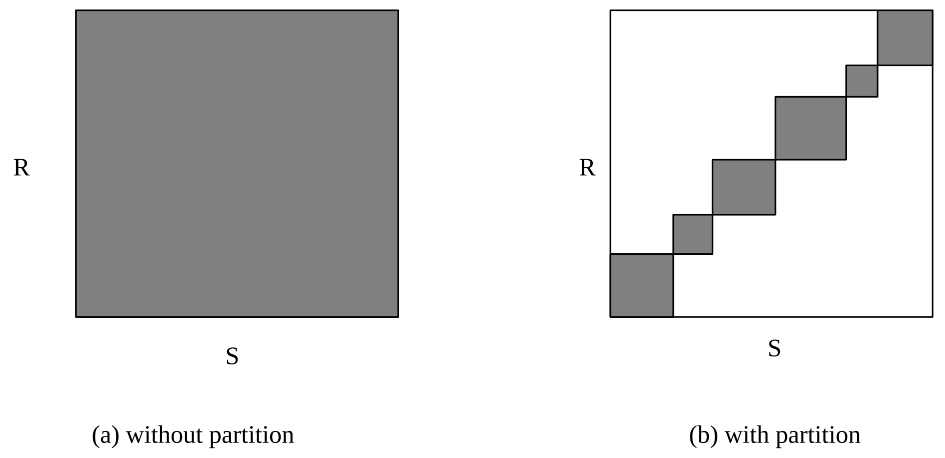
tuples are written into result relation. This procedure is termed as probing phase. It works as follows: while applying a hash function to two attributes, if the hash values are different, then the two attribute values can not be equal.

The reason for choosing the smaller relation to build the hash table is try to avoid building a large hash table in the available memory. But even the smaller relation can still produce a hash table which exceeds the memory. To solve this problem, all the enhanced hash-join algorithms partition the two relations into disjoint subsets, called buckets, and then join the corresponding buckets. The partition does not reduce the overall size of hash table. However, the hash table is divided into a set of small hash tables, and the small hash tables are built in the memory one by one to join each pair of the corresponding buckets. It is this “divide and conquer “strategy allows us to parallelize the hash-join algorithms. Usually, each hash-join algorithm is executed in two phases:

Partitioning phase: Apply a hash function *h(x)* to the join attributes of the tuples in both R and S. According to the hash value, each tuple is put into a corresponding bucket. Suppose the range of hash values is H, and $H_1, H_2, \ldots, H_n$ are disjoint subsets of H, where $H = H_1 \cup H_2 ... \cup H_n$, and $H_1 \cap H_2 \ldots \cap H_n = \phi$. Buckets $R_1, R_2, \ldots, R_n$ and $S_1, S_2, \ldots, S_n$ are corresponding buckets of R and S, such that $R = R_1 \cup R_2 ... \cup R_n$, $R_1 \cap R_2 \ldots \cap R_n = \phi$, and $S = S_1 \cup S_2 ... \cup S_n$, $S_1 \cap S_2 \ldots \cap S_n = \phi$. Suppose r is a tuple of R, if *h(r)* is in H_i, then r is put into R_i. Similarly, the tuples of S are put into S_i.

Joining phase: Use the basic hash-join algorithm to join R_i and $S_i (i = 1 \ldots n)$. We do not need to join R_i and S_j if $i \neq j$. If $r \in R_i$ and $s \in S_i$, we have *h(r)* $\neq$ *h(s)*, so that r and s do not match on join attributes.

The efficiency of this join operation comes from the reduction of work load, which is illustrated in figure 2.2.



Shaded parts represent the work load

Figure 2.2. The Reduction of Work Load in Hash-Join

Although the hash-based join algorithms are efficient under most conditions, there are some problems with this class of algorithms. First, it is not easy to choose a hash function that partitions the source relations into equal size buckets, as the distribution of join attribute values are not known. Furthermore, keeping track of the distribution of join attribute values may not be practical due to its high cost. When a hash function fails to partition the source relations uniformly, some buckets of R may be too large to fit into the memory. This is called bucket overflow. A compensating method for bucket overflow is to further divide the oversized buckets into smaller buckets by recursively applying the same hash function [15, 8]. Even when the bucket overflow does not occur, the unbalanced partitioning has impact on the load balancing in a multiprocessor environment. Second, the hash-join algorithms are only suitable for equijoin. For nonequijoins, the probing procedure is difficult to implement.

2.1.1 Simple Hash-Join Algorithm

The Simple Hash-Join algorithm creates only one bucket at a time during its execution, instead of creating all the buckets at the begining. Suppose the range of hash values is H. According to the size of available memory, H is partitioned into $H_0, H_2, \ldots, H_{n-1}$, so that the corresponding $R_0, R_2, \ldots, R_{n-1}$ can fit into memory. At first, the relation R is scanned and the hash function is applied to the join attributes of each tuple. If the hash value is in H_1, the tuple is put into the in-memory hash table, otherwise it is written into a temporary file *R_temp*. At the end of this stage, R is partitioned into two parts: the tuples whose hash values are in H_1 are used to build the hash-table, the other tuples are written in *R_temp* file. Next, the relation S is scanned, and the same hash function is also applied to the join attributes of each tuple. If the hash value is in H_1, this tuple is used to probe the hash table, otherwise the tuple is written into a temporary file *S_temp*. This process is repeated with *R_temp* and *S_temp* as its input files and $H_i (i = 2 \ldots n)$ as new range partition.

The process terminates whenever either the *R_temp* or *S_temp* is empty.

The pseudo code for Simple Hash-Join algorithm is as follows:

```
/* h is the hash function     */
/* H[0..n-1] is the range array */
i=0;
do
{        for (each tuple r in R){
                 if (h(r) in current_range)
                         insert r into hash-table;
                 else
                         write r into R_temp;
        }
        for (each tuple s in S){
                 if (h(s) in current_range){
                         use s to probe the hash-table;
                         if (any match is found)
                                 output the matching tuples;
                 }
                 else
                         write s into S_temp;
        }
        R = R_temp;
        S = S_temp;
        current_range = H[i+1];
}
while (R_temp is not empty and S_temp is not empty);
```

The disadvantage of this algorithm is that it introduces too many I/O operations when the memory is not large. In that case, the range of hash values has to be partitioned in many subsets; and each tuple may be frequently read and written. Under most conditions, this algorithm is not efficient due to the I/O overhead. However, if the memory is large enough to hold the entire hash table built from R, partitioning is not necessary and this algorithm can provide good performance.

2.1.2 GRACE Hash-Join Algorithm

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 chapter 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 avoides 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.

2.1.3 Hybrid Hash-Join Algorithm

The Hybrid hash-join algorithm combines both the features of Simple Hash-Join and Grace Hash Join. Similar to the GRACE hash-join, it consists of partitioning phase and joining phase. However, in the partitioning phase, a portion of relation R is used to build an in-memory hash table. When the memory is large enough, there may be additional memory available besides the memory used to partition relation R. Hybrid hash-join creates the hash table in the additional memory. R is partitioned into *n+1* parts: $R_0, R_1, \ldots, R_n$, where R_0 is an in-memory hash table, and $R_1, \ldots, R_n$ are written back into the disk as temporary files. The corresponding range sets of hash values are $H_0, H_1, \ldots, H_n$. Next, relation S is also partitioned using the same hash function. Logically, S is also partitioned into *n+1* parts: $S_0, S_1, \ldots, S_n$. However, instead of being written back into the disk, the tuples in S_0 are used to probe the hash table. Any matching tuples are output to the result relation. The other parts, which are $S_1, \ldots, S_n$, are written into the disk as temporary files. At the end of this phase, R and S are split into buckets and a portion of join work has been completed. The join phase is the same as in the GRACE hash-join.

The following is the description of this algorithm:

```
/* H[0..n] is the array of range sets of hash function */
/* R[1..n] and S[1..n] are buckets */

for (each tuple r in R)
{         if (the hash value of r is in H[0])
                    insert r into the in-memory hash table;
          else
                    put r into appropriate bucket R[i];
}

for (each tuple s in S)
{         if (the hash value of s is in H[0]){
                    use s to probe the hash table;
                    put any matching tuples into the result relation;
          }
          else
                    put s into 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;
          }
}
```

When the available memory is extremely large, the Hybrid hash-join algorithm outperforms the GRACE hash-join algorithm, since it optimizes the usage of memory. Actually, the size of hash table built in partitioning phase depends on the size of additional memory. If a large hash table can be created in the partitioning phase, a significant part of join phase is done in advance. Meanwhile, the tuples in R_0 and S_0 do not need to be written back into the disk and read in the joining phase. This reduces the number of I/O operations.

2.2 Architecture of the 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: large-scale numeric processing, transaction processing, database processing and their combinations. Its unique all-cache memory structure enables it to run many industry-standard software systems such as standard UNIX operating system, standard programming language (Fortran, C, Cobol) and standard networks. This industry-standard development environment makes it easy to port existing application softwares to the KSR-1 system [24]. 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 2.3 and figure 2.4:

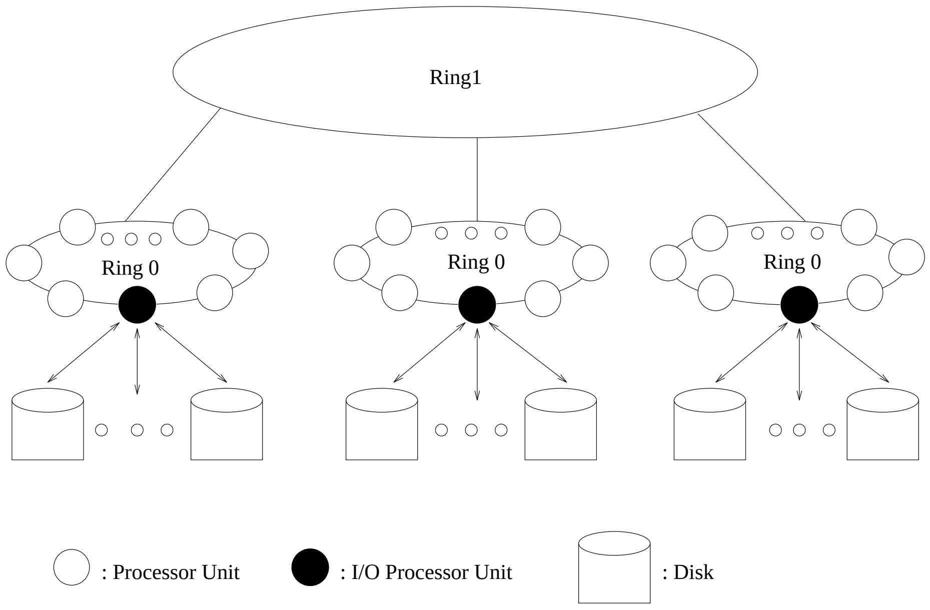


Figure 2.3. General Architecture of KSR-1 System

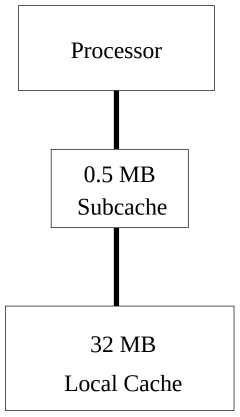


Figure 2.4. The Processor Unit in KSR-1 System

The system 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 a ring:0 is 8 million packets/second (1 GB/second). For a 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 are 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 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.

2.2.1 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 page and the unit of transfer/sharing is subpage. In the memory system, the ALLCACHE engine is responsible for locating and transferring subpages among the local caches, as described in figure 2.5.

In the figure 2.5, 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 [25].

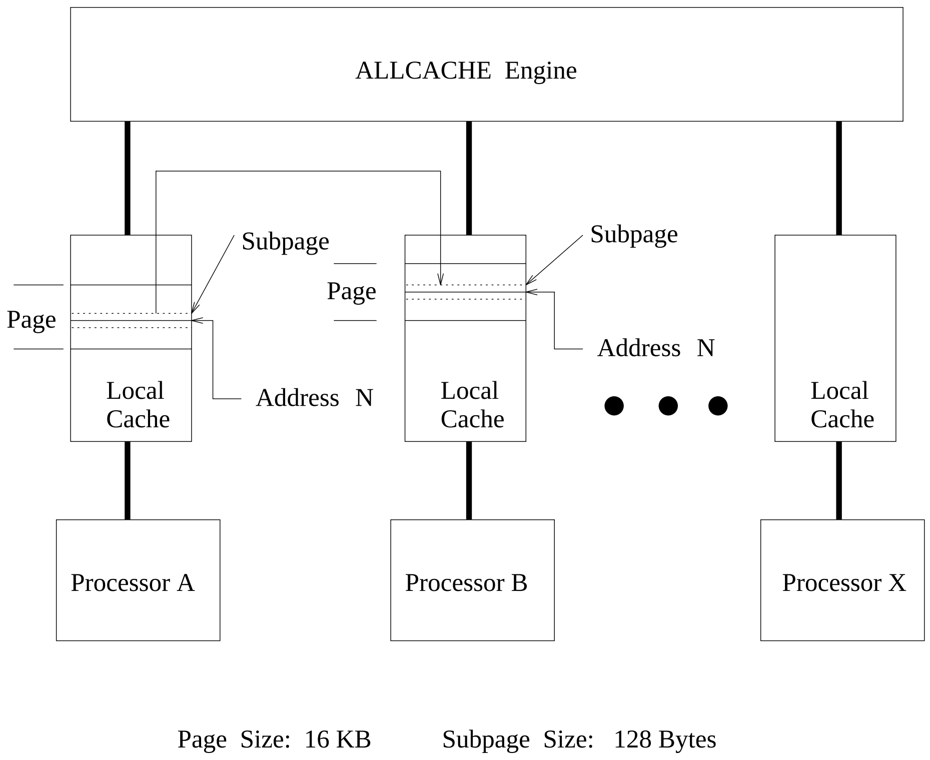


Figure 2.5. ON-Demand Data Movement in The All-Cache Memory

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 caches and processors within the same ring:0 shown in figure 2.3. The second level is ALLCACHE group:1, which consists of three ALLCACHE group:0. Figure 2.6 shows this hierarchical structure:

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

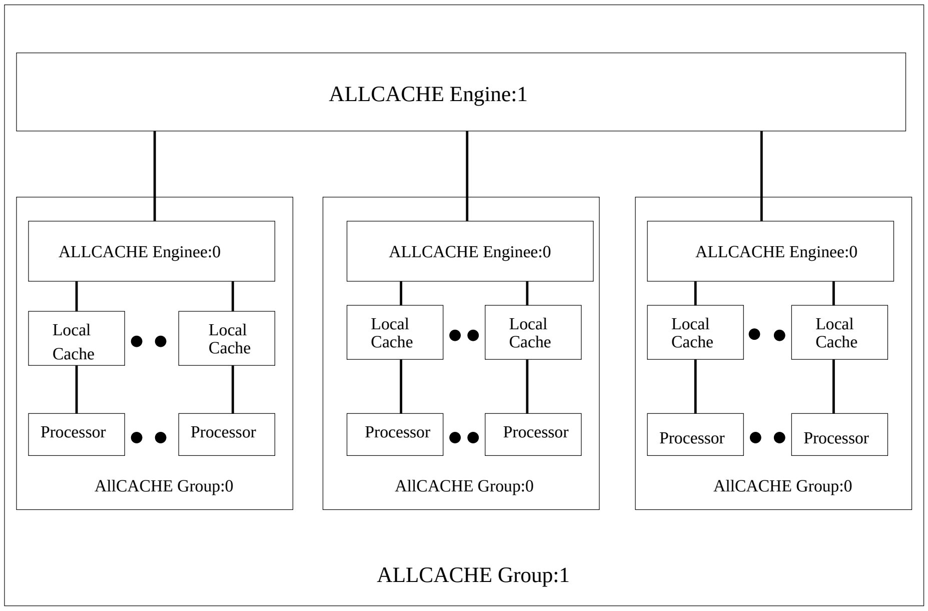


Figure 2.6. The Hierarchical Structure of All-Cache Memory System

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.

2.2.2 Parallel I/O System

As shown in figure 2.3, 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 an 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 has been finished, the I/O processor notifies the requesting processor the accomplishment. In the case that more than one processor need 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.

The architecture of KSR-1 system is similar to the proposed architecture in figure 1.1. First, ring:0 is equivalent to the local bus in the proposed architecture. In figure 1.1, all the processors access the shared memory through local bus; in the KSR-1 system, the processors send memory access requests to ALLCACHE engine:0, and the ALLCACHE engine searches and transfers the requested data through ring:0. Second, the interconnection network in figure 1.1 is implemented in the KSR-1 as the ring:1. ring:1 connects the processor sets located in separate ring:0. This is exactly what the interconnection network does in figure 1.1. Third, although the disks associated to each ring:0 are only connected to the I/O processor, they can be accessed in parallel. The architecture in figure 1.1 provides the same functionality but in a different way: every processor accesses any disks by way of local bus. Furthermore, both of them possess high degree of scalability because of their hierarchical architectures.

2.3 An Overview of Related Work

Hash-based join algorithms show great potential in multiprocessor systems because they can be easily parallelized. Many researchers have proposed parallel version of hash-join algorithms and studied their behavior and performance using simulation, implementation and analytical models [26, 15, 14, 16, 30, 31, 27].

Based on an analytical model, [15] 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. [14] 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) [10]. The result of performance evaluation not only verified the analytical conclusions in [15], but also showed that both GRACE hash-join and Hybrid hash-join algorithms provide linear increases in performance when resources increase. [16] 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-uniformly distribution of join attribute values has 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. [30] 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 the 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 the contention. [31] proposed a new version of parallel GRACE hash-join algorithm for a shared-everything environment. The modification on this algorithm was designed to improve the 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 a much better performance when the data is skewed. [27] implemented and evaluated 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.

CHAPTER 3
IMPLEMENTATION ISSUES

Since the KSR-1 is a general-purpose multiprocessor system, it provides great flexibility for software development, but does not have special hardware for supporting database operations such as sorting and indexing. Therefore, to parallelize the GRACE hash-join algorithm on KSR-1 system, it is very important to efficiently make use of the available resources such as parallel I/O system, automatic load balancing and all-cache memory structure.

We further divide the joining phase of the GRACE hash-based join algorithm into two parts: the phase of building the hash table and the phase for probing the hash table. This results in three phases for the GRACE hash-based join algorithm: partitioning phase, building phase and probing phase. Exploit of parallelism in each phase is important.

This chapter discusses several implementation techniques, each making use of a particular aspect of parallelism. The rest of this chapter is organized as follows: section 3.1 presents the technique for overlapping I/O operations and CPU processing; section 3.2 addresses data partitioning and distribution; section 3.3 describes the synchronization mechanism; section 3.4 details the parallelism obtained in each phase of GRACE hash-based join algorithm; section 3.5 discusses the hash function.

3.1 Double Buffering Technique

Most database operations spend a lot of time in accessing the secondary storage because the main memory is usually not large enough to hold the entire database. In the case of GRACE hash-based join algorithm, there are I/O operations involved in each phase: in the partitioning phase, the tuples in both R and S need to be read into the memory for applying the hash function for partitioning the relations, and then each partition is written back to one or more disks; in the building phase, the tuples in R are read into memory to build hash tables; in the probing phase, the tuples in S are read into memory to probe the hash tables. Hence, taking advantage of the parallel I/O system on the KSR-1 to reduce the I/O waiting time is a main issue for implementing the parallel GRACE hash-join algorithm.

In each phase, the data are read into memory page by page. It is not possible to reduce the time to read/write one page, but we can overlap the I/O operations and CPU processing. The idea is to use double buffering technique. Usually, each processor needs a local buffer to hold the data which is being processed. For example, in the partitioning phase, suppose processor P_0 is in charge of partitioning the data file R_0, and B_0 is the local buffer associated with P_0. The tuples in R_0 are first read into B_0, then P_0 begins to process the data in B_0. When P_0 has finished processing B_0, it sends request to the I/O system for getting more data into buffer B_0 from file R_0. The disadvantage of this procedure is obvious: whenever P_0 needs new data, it has to wait for the disk read operation.

Double buffering technique solves this problem by assigning two buffers for each processor. In order to implement double buffering, there need be two buffers, B_0 and B_1, associated with P_0 in the previous example. First, the processor P_0 reads data into B_0; then, it creates another thread to read data from R_0 into B_1. Thus, while P_0 is processing the data in B_0, new data are also being read into B_1. When P_0 finishes the processing of B_0, hopefully B_1 has been ready to be processed. Similarly, while processing B_1, new data are also being read into B_0. In this way, most of I/O waiting time is avoided.

The double buffering technique is suitable for the parallel I/O structure in the KSR-1 system. As we know, the disks are only connected to the I/O processor in each ring. The other processors only need to send I/O requests to the I/O processors, and the I/O processors will take charge of the actual I/O operations. In the case of double buffering, the thread created for reading new data takes little time of P_0, because the corresponding I/O processor accomplishes most of the I/O operation. Hence, P_0 can dedicate itself to performing join operation.

3.2 Data Partitioning and Distribution

To take full advantage of parallel I/O system, data can also be partitioned and distributed. In the absence of data distribution, multiple processors may have to access the same disk at the same time, and one I/O operation needs to wait for the end of another I/O operation. This situation causes the disk subsystem to be the bottleneck of performance.

If data is partitioned and distributed across multiple disks, simultaneous access to the same relation, either R or S, , or both, becomes feasible. Suppose the relation R is partitioned into five data files: R_0, R_1, R_2, R_3, R_4, and they are stored in five disks: D_0, D_1, D_2, D_3, D_4, respectively. Five processors, P_0, P_1, P_2, P_3, P_4, are assigned to process each data file. P_i can access file R_i $(i = 0, \ldots, 5)$ without any contention. If the number of processors is larger than the number of disks, it is also possible to allocate multiple processors for each data file. However, there is an optimal number of processors for each data file [27]. Let us denote it as M. If more than M processors are processing a data file, the performance may remain the same or even be degraded while increasing the number of processors. The reason is that the effect of too much I/O contention outweighs the acceleration of multiple processors.

Distribution of data for the parallel GRACE hash-join is shown in figure 3.1. First, the two source relations R and S are horizontally partitioned and distributed across a set of disks. These disks may be controlled by the same I/O processor but must be connected to different I/O channels, because two disks using the same I/O channel can not be accessed in parallel. Second, At the end of partitioning phase, the bucket files of the same relation are distributed across the involved disks, so that the join operation can be performed on different pairs of buckets concurrently. Finally, To minimize the case that different processors are trying to write into the same disk simultaneously, the result relation are distributed across the disks.

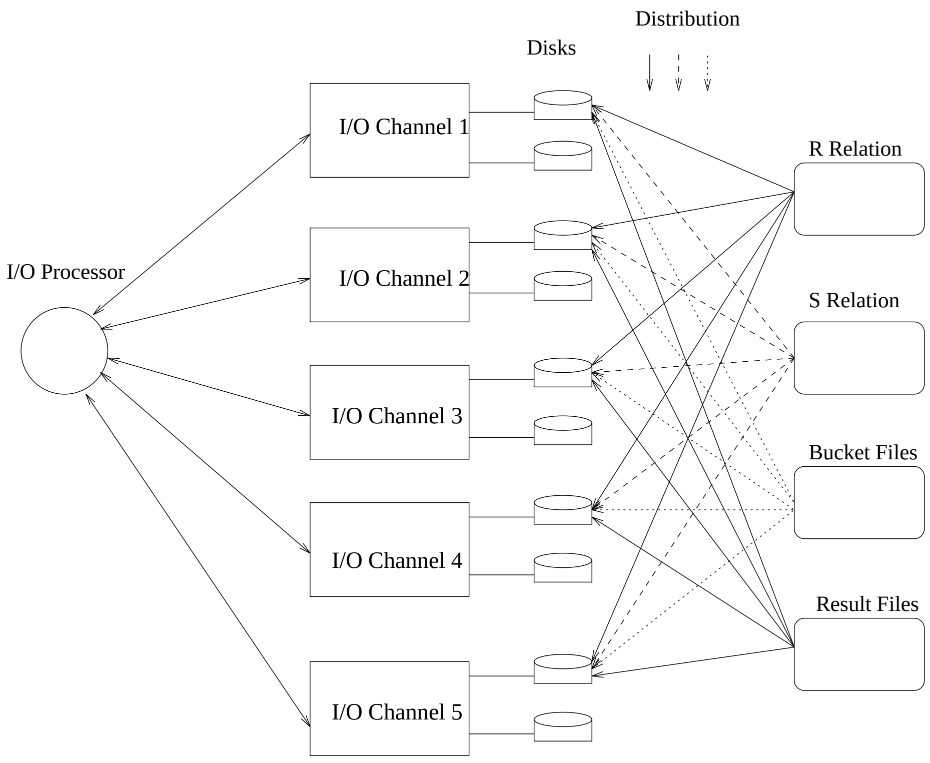


Figure 3.1. Data Distribution During Parallel GRACE Hash-Join

In our implementation, the two source relations, R and S are horizontally partitioned into data files of equal size. The data files of the same relation are stored in different disks, but one disk can hold the data files from both relations. For example, Suppose R_0 is one of the data files from R and S_0 is one of the data files from S, R_0 and S_0 can reside in the same disk. During the execution of parallel GRACE hash join algorithm, at any point of time, only one relation, either R or S needs to be accessed, unless the algorithm intends to partition them concurrently.

3.3 Parallelism in Each Phase

In the partitioning phase, suppose the relation R is being partitioned, and R consists of n data files $R_0, \ldots, R_{n-1}$, which reside in n disks $D_0, \ldots, D_{n-1}$, respectively. These data files are processed in parallel: for each data file, more than one processor can be allocated. Since each partition of R is a bucket file, a write buffer is allocated for each bucket at the beginning of this phase. Suppose P_i is one of the processors which are processing R_0. First, P_i allocates two local buffers for itself in order to realize the double buffering. Then, the tuples are read into the buffers page by page. P_i applies the partitioning hash function to the join attribute values of each tuple in its local buffer. According to the hash value, the tuples are moved to the appropriate buffers of buckets. When the buffer for a bucket gets full, it is written back into the disk. Each of the other processors works in the same way as P_i. After having partitioned relation R, The processors begin to partition relation S. This is the case that R and S are partitioned one by one. It is also possible to partition R and S concurrently, as long as the number of processors is large enough, and the I/O contention caused by different files residing in the same disk is not high. At the end of this phase, All the tuples are organized into buckets, and these buckets are distributed across the disks.

In the building phase, each bucket of R is used to build an in-memory hash table. Assuming that the bucket files: $RB_0, \ldots, RB_{n-1}$ are stored in the disks $D_0, \ldots, D_{n-1}$, the hash table of each bucket can be created in parallel: Each bucket file may be processed by a small processor set. The involved processors read data into their local buffers, apply the hash function to the join attributes of each tuple, and then insert the tuple into the corresponding hash table. At the end of this phase, the hash tables are built for some or all of the bucket files from R. The number of hash tables created in this phase depends on the size of available memory.

In the probing phase, the join results will be generated. To achieve the maximum parallelism, the result relation is also distributed across the disks. If the result consists of n data files, n write buffers need to be allocated for each data file at the beginning of this phase. Since the relation S is also distributed among the disks, they can be processed in parallel. A small processor set can be assigned to process each data file. Suppose processor P_i is in the processor set in charge of RS_j. P_i reads the tuples from RS_j into its local buffers, then applies the same hash function used in the building phase to the the join attribute of each tuple. If a match is found in the corresponding hash table, the two tuples are concatenated and moved to the appropriate write buffer of result data file. When the write buffers are full, they are written into the disks.

3.4 Synchronization Mechanism

Since the KSR-1 is a shared-everything multiprocessor system, the processors or threads communicate with each other by means of accessing the same memory. The KSR-1 provides a "barrier" mechanism for the sake of synchronization. A barrier is a software environment to enforce multiple threads to execute in parallel. There are two types of threads in a barrier: master thread and slave thread. Each barrier can have only one master thread, but may have many slave threads. All the slave threads are suspended until the master thread enters the barrier, and only after all the slave threads have terminated, the master thread can check out of the barrier. Thus, the parallel work begins with the check in of the master thread, ends with the check out of the master thread. The KSR-1 system also provides other software synchronization mechanisms. Since it is easy to measure the timing using a barrier because the barrier can guarantee that each thread starts at the same time, we select the barrier mechanism for our implementation.

The KSR-1 system also provides lock mechanism which is necessary for protecting some shared files and buffers. There are a couple of cases in which locks are needed to prohibit the concurrent access to shared data. For example, in the partitioning phase, if more than one processor is used to process the same data file, lock is used to prevent the processors from reading the same portion of data to their local buffers. The write buffer for each bucket also need to be protected by lock from concurrent writes, which may cause the loss of data. In the building phase, lock is used to prevent concurrent access to the hash table. In the probing phase, the write buffer for each result data file is protected by lock from being concurrently written.

3.5 Hash Function

A good hash function is very critical to the hash-based join algorithms. In the partitioning phase of GRACE hash-join algorithm, a proper hash function is needed to generate the buckets of equal or similar size; in the building phase, a good hash function means that few tuples should be hashed into the same entry. In other words, the chain in each entry should be kept short. Having short chains in the hash table accelerates the probing phase.

The criteria for a good hash function is to hash the tuples in a uniform way: each tuple has the same probability to hash to any entry. However, it is difficult to find such an optimal hash function in practice. As a guideline, if a hash function derives the hash value in a way which is independent of the patterns of hashed data, it is likely to perform well [11].

There are two hash functions in our implementation. The first hash function is used in the partitioning phase. In order to control the number of buckets easily, we choose the *MOD* function as the hash function. If the algorithm intends to partition the relations into n buckets, then the hash function is

$keyMODn.$

The range of hash values is $0, 1, \ldots, n-1$. If the hash value of a tuple is i, then it is put into bucket i.

The second hash function is used in the building phase and probing phase. Suppose k is the join attribute of tuples, the chosen hash function is: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$, where A is a constant in the range $0 < A < 1$, m is an integer. [28] suggests that when $A = (\sqrt{5}-1)/2 \approx 0.618034$, the hash function provides good performance. For example, if k = 5427, m = 1000, and A = 0.618034, then

$\mathrm{h(k)} = \lfloor 1000 * (5427 * 0.618034 - \lfloor 5427 * 0.618034 \rfloor) \rfloor$

$= \lfloor 1000 * (3354.0705 - 3354) \rfloor$

$= \lfloor 70.5 \rfloor$

$= 70$

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 range without influencing the performance.

CHAPTER 4
RESULTS AND ANALYSIS

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

We implemented the GRACE hash-join algorithm 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 (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 of 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 optimal 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. Consequently, the execution time of a parallel program may vary slightly from run to run. To obtain the most accurate results, we executed our GRACE hash-join algorithm with the same parameters for several times (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 time measuring 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 are similar to the one proposed in Wisconsin benchmark [5]. 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 joinability is 60%, which means that the result relation has 150,000 tuples.

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

4.1 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 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 are also a sort of 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.

4.1.1 Processing Buffer Size vs. Performance

Although the KSR-1 system provides a huge memory pool, either read buffer or write buffer can not be arbitrary large. On the other hand, large buffers do not necessarily mean good performance. We executed the algorithm with various sizes of buffers. Some other parameters are listed below:

Size of R relation:	250,000 tuples
Size of S relation:	250,000 tuples
Size of the result relation:	150,000 tuples
Number of processors to partition each data file:	5
Number of processors to build hash table from each bucket file:	5
Number of processors to probe hash table from each bucket file:	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
Range of hash values:	30000

The results is shown in figure 4.1.

Figure 4.1 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.

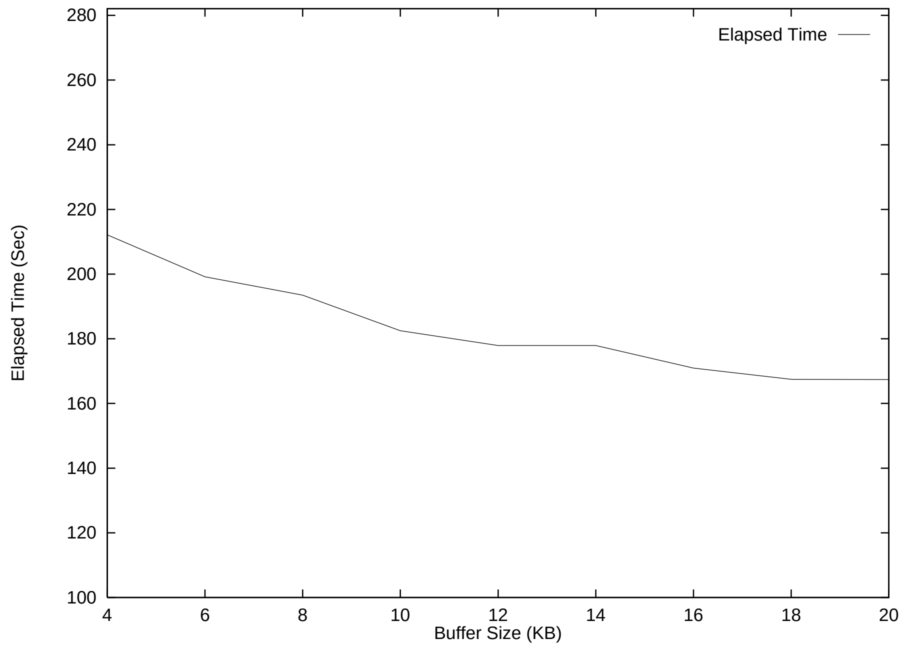


Figure 4.1. Processing Buffer Size vs. Elapsed Time

It is noticed that the page size of the all-cache memory of KSR-1 system is also 16 KB. We believe that this is somehow related the optimal buffer size in the GRACE hash-join algorithm. As described in chapter 2, the unit of allocation in the all-cache memory is 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 certain level with the increase in buffer size.

4.1.2 Range of Hash Values vs. Performance

The hash function we used in the implementation allows easy change of the range of hash values. Actually, the range of hash values implies the average number of tuples in each entry of 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 4.2 shows the change of probing time with the increase in hash range. The following is a list of parameters in this experiment:

Size of R relation:	250,000 tuples
Size of S relation:	250,000 tuples
Size of the result relation:	150,000 tuples
Number of processors to partition each data file:	5
Number of processors to build hash table from each bucket file:	5
Number of processors to probe hash table from each bucket file:	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

During the probing phase, if the hash value of a tuple from the buckets of 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 4.2, the probing time decreases with the increase in hash range while the hash range is smaller than 30 K. After the hash range is larger than 30 K, 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, 30 K in this case, almost each entry has only one tuple. The increase in hash range above this number

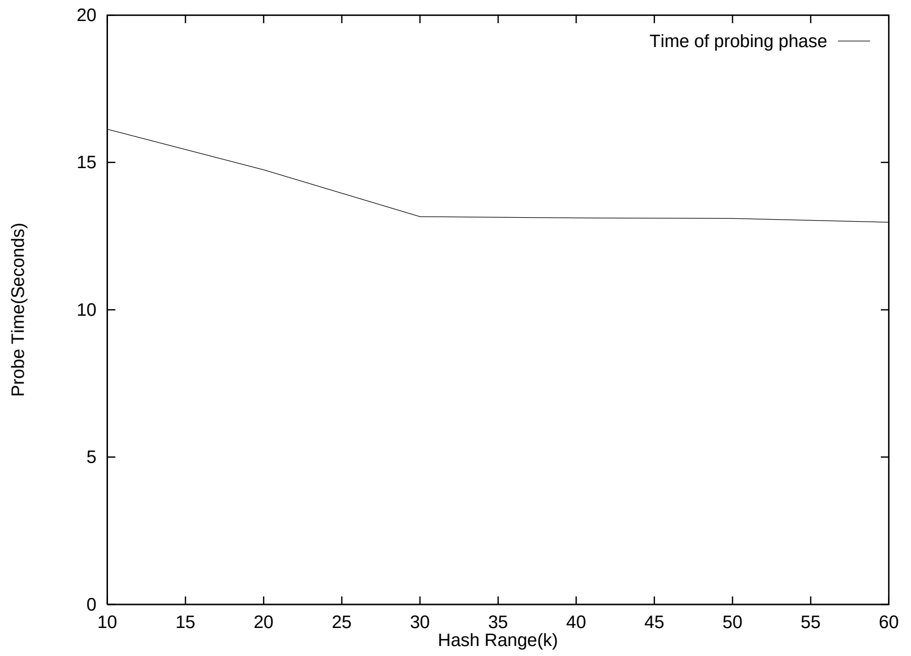


Figure 4.2. Range of Hash values vs. Probing Time

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.

4.2 Parallelizing I/O Operations

Usually, the main memory is not large enough to hold an entire relation. During each phase of the GRACE hash-join, data need to be moved back and forth between disks and memory. The efficiency of I/O operations has a great influence on the performance. In the centralized database systems, a major objective of query optimization is to reduce the I/O cost. This is also true in the 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 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.

4.2.1 Double Buffering

The double buffering technique is presented in chapter 3. It is a common technique to overlap the I/O operations and CPU computing. In the GRACE hash-join on the KSR-1 system, this technique is applied to each processor. Although the double buffering can reduce the I/O waiting time of each processor, there is also cost for this option. The KSR-1 system provides asynchronous I/O ability which allows the 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 certain data file. The behavior of 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 double buffering technique.

We executed the GRACE hash-join algorithm with and without double buffering, measuring the performance in each case. The following are some parameters:

Size of R relation:	250,000 tuples
Size of S relation:	250,000 tuples
Size of the result relation:	150,000 tuples
Number of processors to partition each data file:	5
Number of processors to build hash table from each bucket file:	5
Number of processors to probe hash table from each bucket file:	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
Range of hash values:	30000

The results are shown in figure 4.3.

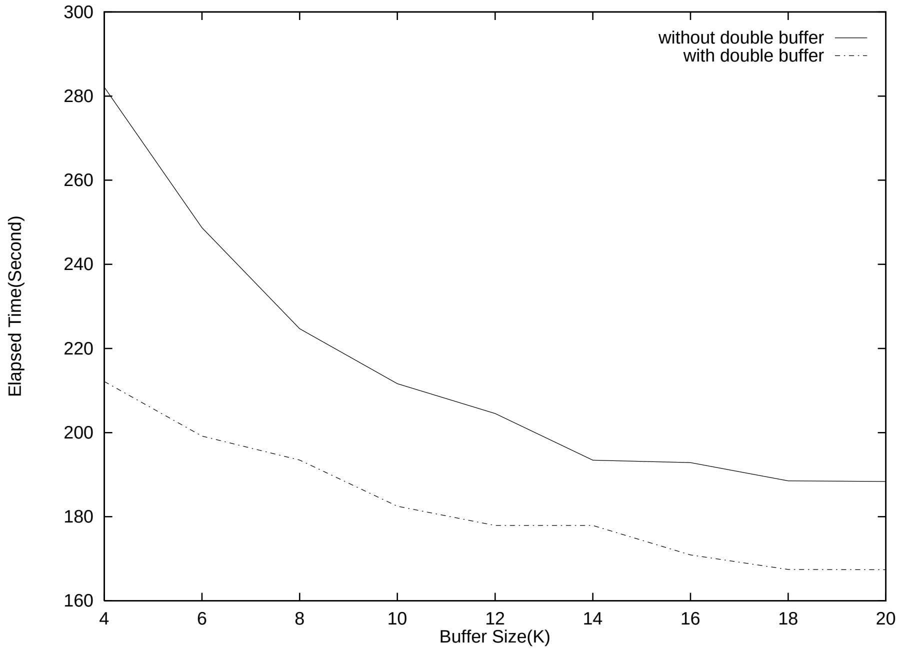


Figure 4.3. The Effect of Double Buffering

Figure 4.3 shows that the double buffering technique makes a great difference of performance. The algorithm have much shorter elapsed time when the double buffering is applied. The results indicate that the benefits of double buffering outweigh its overhead. We conclude that the double buffering is appropriate for the GRACE hash-join algorithm on the KSR-1 system.

4.2.2 Arranging I/O threads

Because of the implementation of double buffering, each I/O operation is carried on 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. Some parameters are listed below:

Buffer Size:	16 KB
Size of R relation:	250,000 tuples
Size of S relation:	250,000 tuples
Size of the result relation:	150,000 tuples
Number of processors to partition each data file:	5
Number of processors to build hash table from each bucket file:	5
Number of processors to probe hash table from each bucket file:	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
Range of hash values:	30000

The elapsed time of each case are compared in figure 4.4.

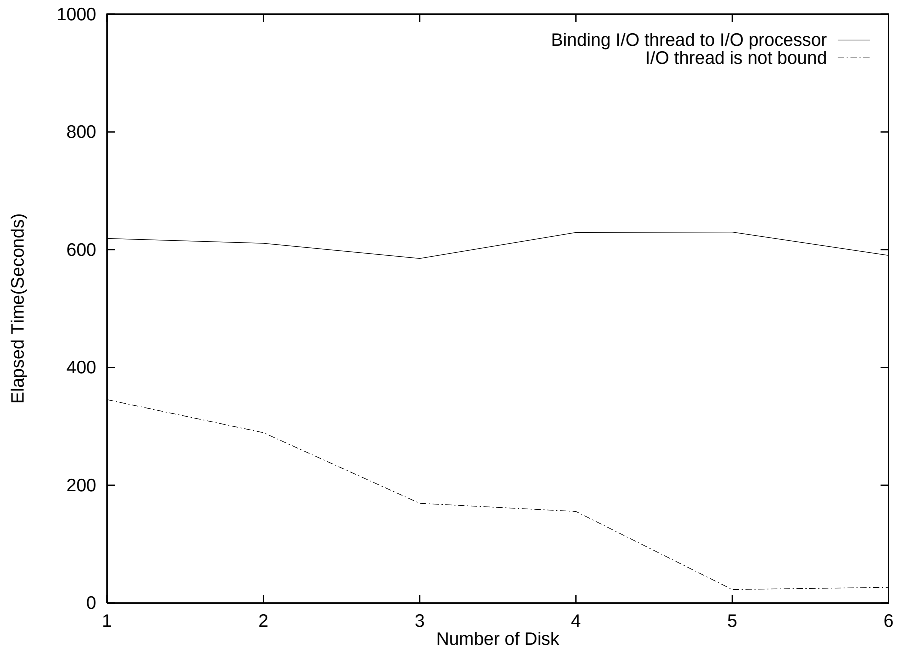


Figure 4.4. Binding methods of I/O threads vs. performance

Figure 4.4 shows surprising results. Instead of improving the performance, binding I/O threads keeps the elapsed time consistently large, regardless of the increase in the number of disks. In contrast, when the I/O threads are assigned to the processors automatically by the system, the elapsed time is much shorter and decreases with the increase in the number of disks. To find out the reason for this anomaly, we observed the CPU utilization when running the algorithm with binding the 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 is 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 I/O thread for each of its I/O operation. 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 is definitely overloaded. This experiment result suggests it is better to let the system handle the I/O threads automatically.

4.3 Processor Allocation and Load Balancing

The 96 processors are distributed in three separate rings in the KSR-1 system. Because the communication cost across rings is 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. It is interesting to find out what is a good processor allocation strategy among these available alternatives.

4.3.1 Different Processor Sets vs. Performance

When an algorithm runs on the KSR-1 system, the processors it can allocate 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 such restriction, all the processors are available for the algorithm, and each thread of the algorithm will be assigned to 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. Other parameters in this experiment are as follows:

Buffer size:	16 KB
Number of processors to partition each data file:	5
Number of processors to build hash table from each bucket file:	5
Number of processors to probe hash table from each bucket file:	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
Range of hash values:	30000

The result of this experiment is shown in figure 4.5.

From figure 4.5, we observed that the elapsed time is shorter when the processor set is restricted in 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 are lots

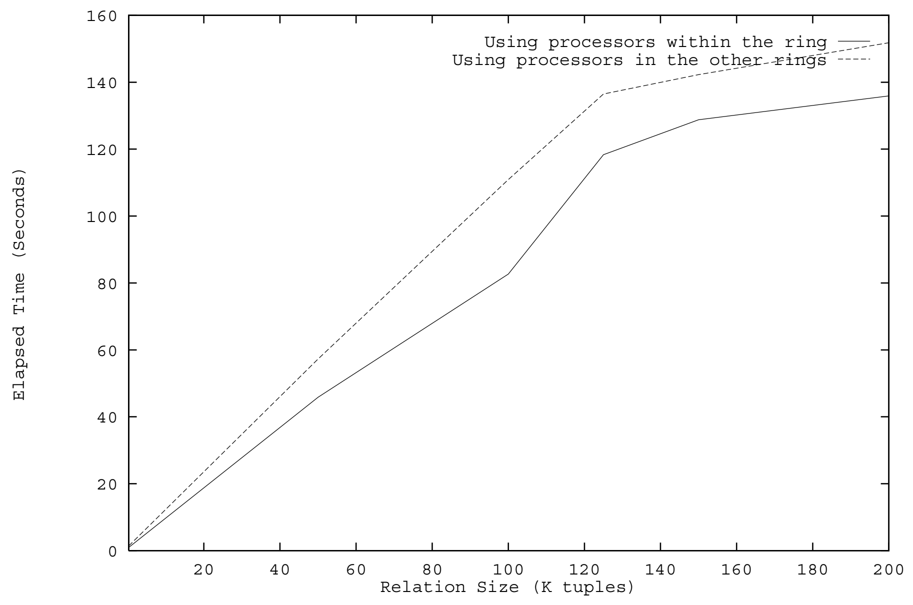


Figure 4.5. The Impact of Different Processor Sets

of 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 the communication are within this ring and has lower cost.

In figure 4.5, With the increase in relation size, the difference in elapsed time increases. Larger relation size implies 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 4.5 diverge.

Based on the above result and analysis, we conclude that the location of 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.

4.3.2 Load Balancing vs. Performance

A 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 the load balancing 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 migrations are prohibited. Each processor only handles the jobs which are assigned to it at the beginning of execution. "autobinding" is one of the functions to perform 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 find out a suitable load balancing strategy for the GRACE hash-join algorithm, we first executed the algorithm with automatic load balancing, and then executed it using "autobinding" to prevent the migrations of threads. Other parameters in this experiment are as follows:

Buffer size:	16 KB
Size of R relation:	250,000 tuples
Size of S relation:	250,000 tuples

Size of the result relation:	150,000 tuples
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
Range of hash values:	30000

Figure 4.6 shows the elapsed time for both executions.

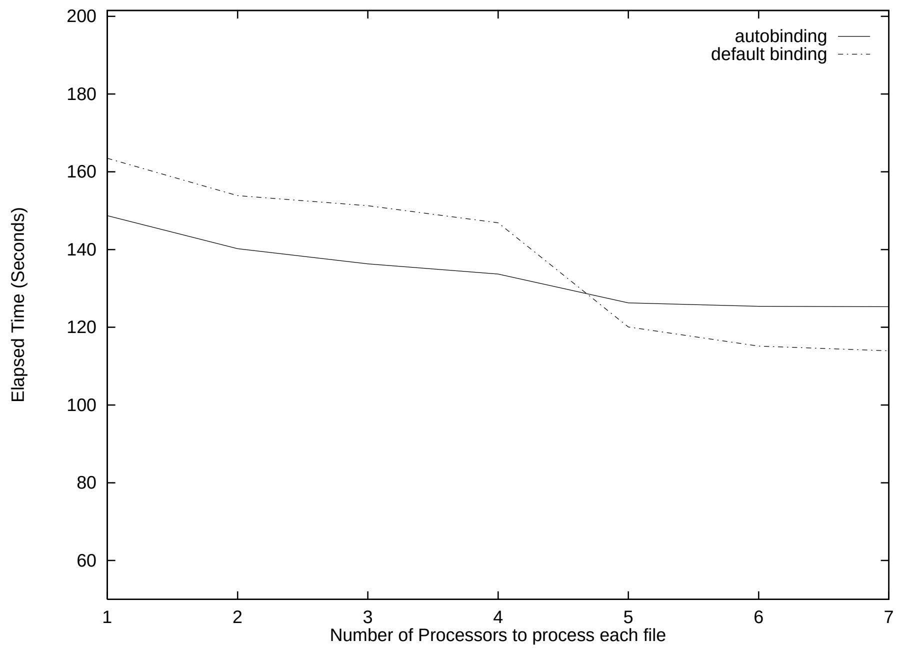


Figure 4.6. Load Balancing strategy vs. Elapsed Time

Figure 4.6 shows that autobinding case outperforms automatic load balancing case 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 the difference in elapsed time when only one processor is used. Second, automatic load balancing needs to migrate the threads among the processors. The migrations invoke additional communication which is absent in the autobinding case. When the processor set is small, the automatic load balancing mechanism does not have much chance 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 autobinding case. As a consequence, the overhead and additional communication cost are compensated by the effect of the 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 autobinding strategy when the number of processors is small. If the source relations are not partitioned in a uniform way, The autobinding method may probably lead to extremely unbalanced load for the processors. The elapsed time will be much larger since it is the execution time of the processor with the heaviest load. Hence, we can not conclude that the autobinding strategy is better than the automatic load balancing in the case of small processor set. The autobinding strategy is only suitable for the GRACE hash-join when the processor set is small and the partitioning phase has ideal results. However, it is not easy to divide a job into small jobs with the same size in practice. In conclusion, we believe that the automatic load balancing is a better and safer choice for the GRACE hash-join algorithm on the KSR-1 system.

4.3.3 Number of Processors vs. Performance

Another issue we concern is the number of processors used to process each data file. Since there are three phases in GRACE hash-join algorithm, we investigate this problem in each phase. Because the previous results and analysis show that the automatic load balancing outperforms the other load balancing strategy, we adopt automatic load balancing in the following experiments. Some parameters in these experiments are listed below:

Buffer size:	16 KB
Size of R relation:	250,000 tuples
Size of S relation:	250,000 tuples
Size of the result relation:	150,000 tuples
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
Range of hash values:	30000

Figure 4.7 shows the elapsed time of partitioning phase vs. the number of processors used. While the number of processors is less than five, the partitioning time decreased rapidly with the increase in processor number. But when the number of processor is more than five, there is no substantial change in the partitioning time. This result indicates that five processors is appropriate for catching up with the data flow from one disk.

Figure 4.8 describes the relationship between the number of processors and the time to build the hash table. It is similar to what is shown in figure 4.7. The elapsed time of building phase maintains certain level after the number of processors are greater than five. Five Processors are good enough to build the hash table from one data file.

Figure 4.9 gives the probing time when the number of processors increases. 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 probing phase. The situation implies that one processor is good enough for each data file during the probing phase.

In the above experiments, we observed that there is no linear speed-up when the number of processors increases. When the processor set is large enough, the

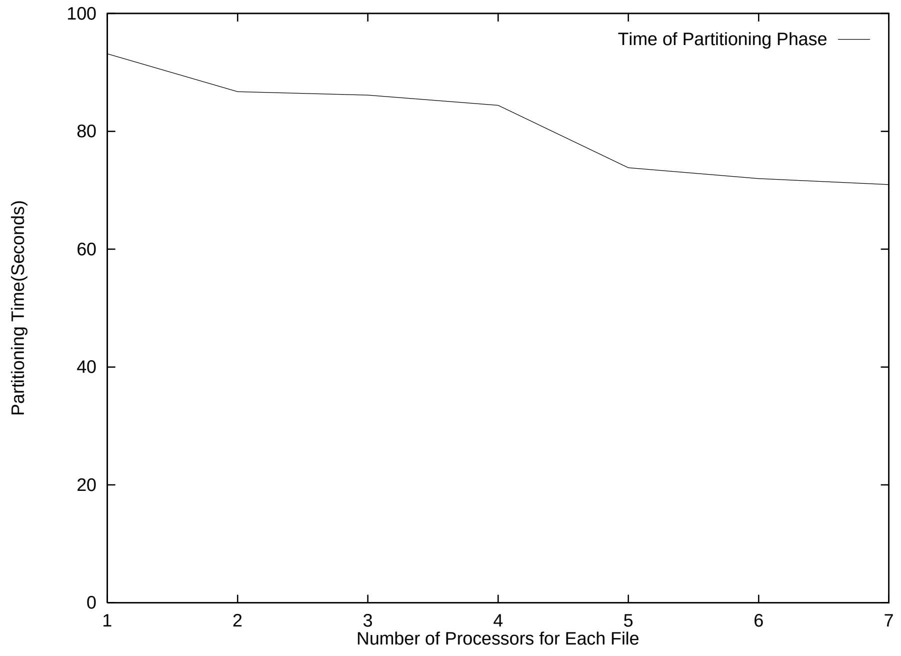


Figure 4.7. Number of Processors vs. Partitioning Time

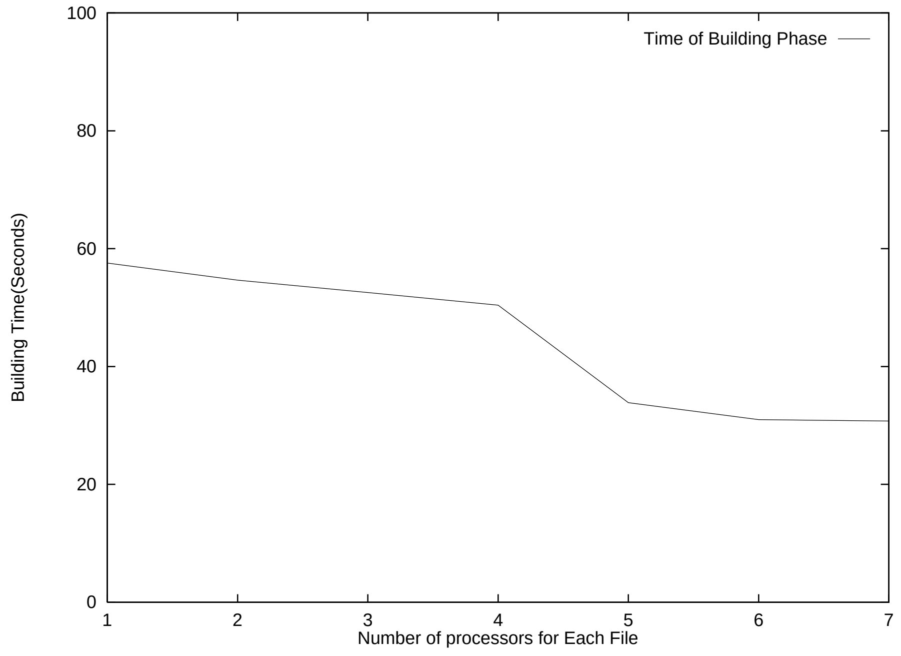


Figure 4.8. Number of Processors vs. Building Time

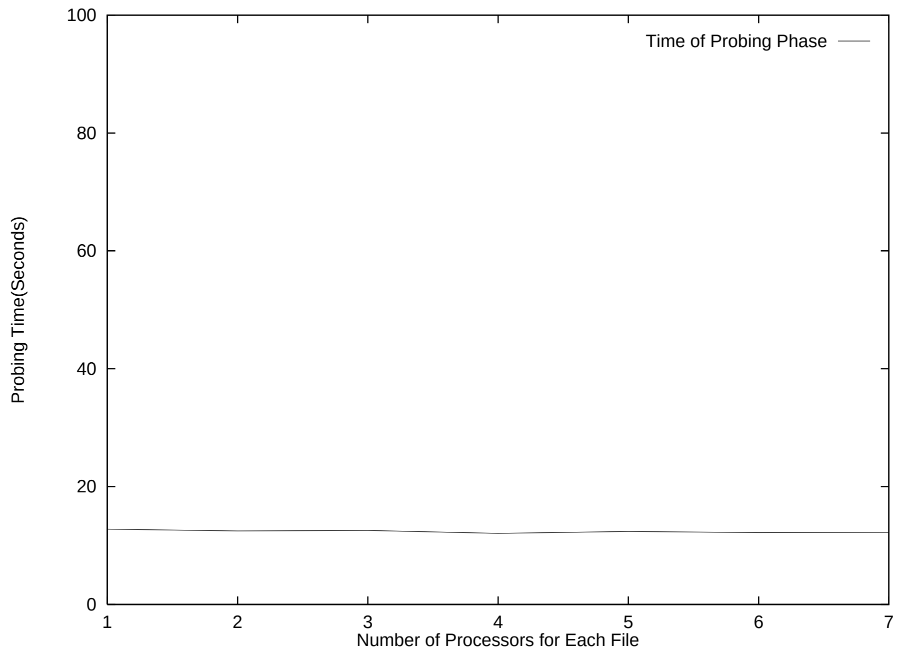


Figure 4.9. Number of Processors vs. Probing Time

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 contentions. 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 many contentions occur.

4.4 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 reside in 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 can not 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, number of data files for each relation expands when the number of disks increases.

4.4.1 Number of Disks vs. Performance

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 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 executed the algorithm with different numbers of disks and compared the elapsed time for each case. Other parameters in this experiment are as follows:

Buffer size:	16 KB
Size of R relation:	250,000 tuples
Size of S relation:	250,000 tuples
Size of the result relation:	150,000 tuples
Number of processors to partition each data file:	5
Number of processors to build hash table from each bucket file:	5
Number of processors to probe hash table from each bucket file:	1
Range of hash values:	30000

The result is presented in figure 4.10:

Figure 4.10 shows the decrease in elapsed time with the increase in the number of disks. It is observed that 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 result shows 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 dominates the elapsed time.

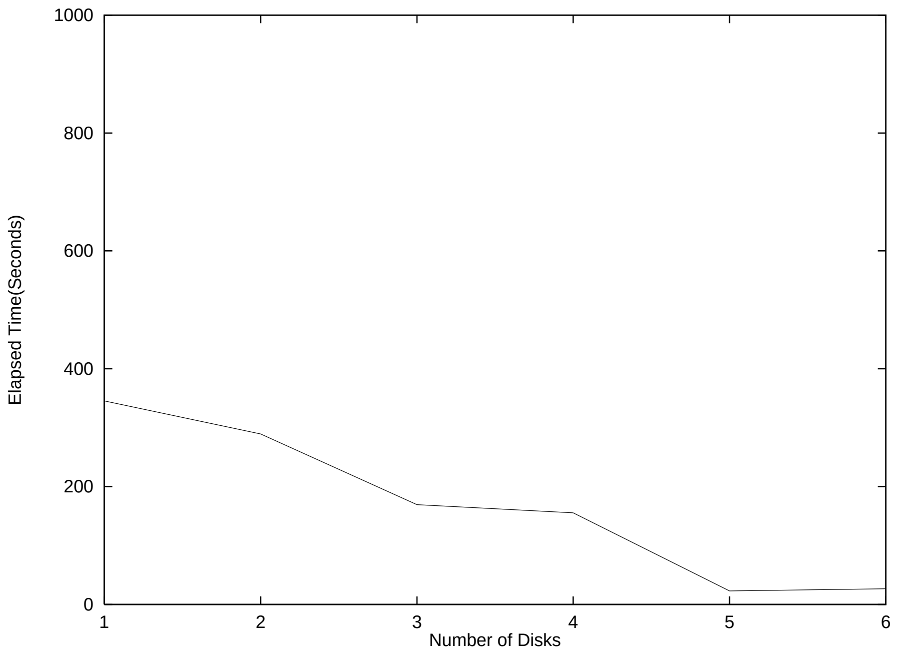


Figure 4.10. Number of Disks vs. Elapsed Time

The increase in disk number can not reduce this part of elapsed time. We also need to mention that the sixth disk shown in figure 4.10 is not connected to the same ring as the previous five disks. [1] To read/write the data files in this disk results in more communication cost. This additional cost also degrades the performance.

4.4.2 Data Distribution vs. Elapsed Time

As we mentioned in chapter 2, 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 be distributed within one disk group or across multiple disk groups during the execution of the parallel GRACE hash-based join. Will it make any difference in performance to distribute the data among different groups? In order to answer this question, 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

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

across three rings, across two rings and within one ring. Some other parameters are as follows:

Buffer size:	16 KB
Number of processors to partition each data file:	5
Number of processors to build hash table from each bucket file:	5
Number of processors to probe hash table from each bucket file:	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
Range of hash values:	30000

The results are shown in figure 4.11.

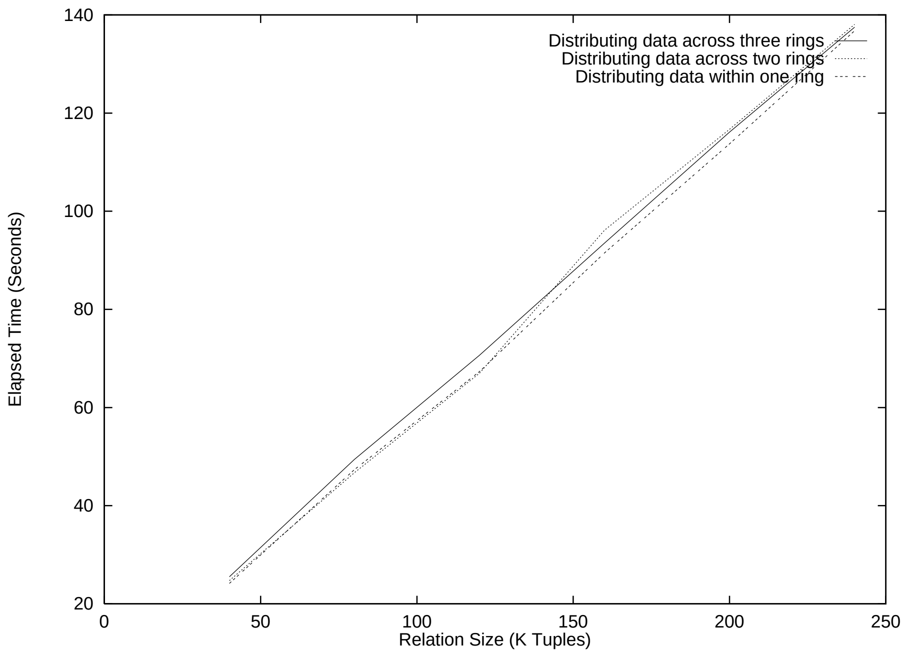


Figure 4.11. Data Distribution vs. Elapsed Time

Figure 4.11 describes the elapsed time in each case when the relation size increase. We noticed that there is no substantial difference among the elapsed time in each case. Although there is 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 that all the data files are distributed within one ring. Although the same amount I/O operations are processed by one I/O processor in the later case, there is no difference in performance. This indicates that the I/O processors are not usually overloaded by the 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 is constituted by all the processors in three rings. If the processor set is restricted within one ring, then it would better distribute the data only within the disks associated to that ring. The purpose is to avoid high communication cost across multiple rings. We concluded that if the processor set consists of the processors in rings $R_1, \ldots, R_n$, then the data are free to be distributed among $R_1, \ldots, R_n$.

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

This thesis presents our preliminary work on the parallel database research within the COMA model. It evaluates the performance of GRACE hash-join algorithm on the KSR-1 multiprocessor system under various conditions. The hash-based join algorithms are distinguished by their efficiency and amenability for parallel implementation. Three different kinds of hash-based join algorithms, Simple hash-join, GRACE hash-join and Hybrid hash-join are described and their suitability for the multiprocessor environment is analyzed.

KSR-1 is a shared-everything multiprocessor with COMA memory structure. The thesis provides an overview of its hardware architecture. The all-cache memory structure and parallel I/O system are described in detail. The similarities between the architecture of KSR-1 system and the general hierarchical architecture proposed in [22] are identified.

To investigate the behavior of hash-based join algorithms in the COMA model and verify the suitability of the proposed architecture in [22] for parallel database design, we implemented the GRACE hash-join algorithm on the KSR-1 system. The thesis addresses the following issues in our implementation: double buffering, data partition and distribution, potential parallelism, synchronization and hash function.

Finally, the performance of GRACE hash-based join algorithm is evaluated under different conditions (e.g., number of processors, relation size, number of disks, etc.) and the corresponding analysis is presented. The experiments are designed to explore the optimal utilization of various resources in the KSR-1 system. The analysis corresponding to each experiment reveals the underlying reasons for the results. A summary of conclusions drawn from the experimental results are listed below:

- The cost of communication across multiple rings is higher than the communication cost within each ring. The processors from a ring should be used to process the local data files (within the same ring) 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.

- In each phase of GRACE hash-join algorithm, there is an optimal number of processors for each data file. In the partitioning phase, five processors seem to be good enough to partition one data file; 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 automatic management of I/O threads in the KSR-1 system gives good performance.

- 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 30 K 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. Furthermore, the KSR-1 system provides a comfortable software development environment for the parallel programming. The standard parallel facilities such as barrier and lock mechanism make it easy to implement the parallel GRACE hash-join algorithm.

The results also suggest that the proposed architecture in [22] possess great potential for parallel database development. Since any multiprocessor systems derived from this architecture may have efficient communication, easy synchronization, automatic load balancing and high degree of scalability. All these facilities have been verified to be desirable for the parallel database design by our experiments on the KSR-1 system.

This is our first effort towards parallel database research within the COMA shared-everything architecture. There are still many open issues. In the shared-everything multiprocessor environment, lock mechanisms are frequently used to protect shared data. In our experiments, we noticed that the contention for various locks seriously affects the performance of algorithm. As part of future work, the effect of lock contention in each phase of GRACE hash-join algorithm needs to be investigated. In the partitioning phase, the hash function splits the source relations R and S into buckets of equal size, because we know the distribution of join attribute values in our experiments. However, this is not practical in a real database system. The performance of

GRACE hash-join algorithm needs to be studied when the partitioning hash function can not uniformly split the source relations. Another open issue is the relationship between the size of available memory and the performance. In the experiments, we assumed that there is enough memory during each phase of the GRACE hash-join, because the KSR-1 system provides a huge shared memory pool. However, for the join operations involving larger relations than in our experiments, the memory size is one of the key elements to determine the performance. When the memory is not large enough to hold all the buffers and hash tables concurrently, the bucket tuning technique may be adopted to solve the problem. It is very useful to find out how these conditions will influence the parallelism of hash-based join algorithms in the COMA shared-everything architecture.

We only implemented and evaluated the GRACE hash-based join algorithm on the KSR-1 system. The Simple hash-based join and Hybrid hash-based join need to be studied to fully understand the hash-based join algorithms in the COMA shared-everything multiprocessor environment. The multiprocessor systems such as the KSR-1 tend to have huge memory. The Hybrid hash-based join algorithm may probably outperforms the other hash-based join algorithms, because it supports better memory usage. This prediction needs to be verified by the future research. Furthermore, there are many other hash-based database operations such as hash-based selection, hash-based projection. The study of these hash-based operations is necessary to build the database system within the COMA shared-everything multiprocessor environment.

REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

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

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

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

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

[9] A. Bricker, D. J. DeWitt, S. Ghandeharizadeh, 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.

[10] 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.

[11] Thmos H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, Cambridge, MA, 1992.

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

[13] D. J. DeWitt. Gamma-a high performance dataflow database machine. In *Proceedings of International Conference on VLDB*, Kyoto, Japan, 1986.

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of ACM SIGMOD Conference*, Atlantic City, NJ, May 1990.

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

[22] Goetz Graefe. Query evaluation techniques for large databases. Technical report cu-cs-579-92, University of Colorado at Boulder, January 1992.

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

[24] Kendall Square Research, Boston, MA. *KSR Parallel Programming*, 1993.

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

[26] M. Kitsuregawa, T. Moto-oka, and H. Tanaka. Application to data base machine and its architecture. *New Generation Computing*, 1(1), 1983.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] M. Stonebraker. The case for shared-nothing. *IEEE Database Engineering*, March 1986.

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

BIOGRAPHICAL SKETCH

Xioahai Zhang was born on March 12, 1968, at Nanning, People's Republic of China. He recieved his bachelor's degree in computer sciences from Hangzhou University, China, in July 1987. He also received his master's degree in computer engineering from Shanghai Jiao Tong University, China, in December 1989. Thereafter, he worked in Hangzhou Instruments Co., China as a software engineer for one year, developing a database management information system. He will receive his Master of Science degree in computer and information sciences from the University of Florida, Gainesville, in December 1994.

His research interests include query processing, parallel databases and object-oriented databases.