This is to certify that the

dissertation entitled

PARALLEL HASH JOIN WITH SKEW
HANDLING ON MULTIPROCESSOR SYSTEMS

presented by

Walid Rifaat Tout

has been accepted towards fulfillment
of the requirements for

PhD degree in Computer Science

Major professor

Date 9/8/94

PARALLEL HASH JOIN WITH SKEW HANDLING ON MULTIPROCESSOR SYSTEMS

By

Walid R. Tout

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1993

ABSTRACT

PARALLEL HASH JOIN WITH SKEW HANDLING ON MULTIPROCESSOR SYSTEMS

By

Walid R. Tout

The sizes of databases have been growing at an exceedingly fast rate in the past years and this trend is expected to continue for years to come. Much research has focused on applying parallel processing to these increasingly large databases. Of all database operations, the relational join is considered as the most time consuming operation. Hence, it has been used in the research community as the standard measure for parallel database systems performance.

The parallel database architectures that have been extensively studied are the Shared-Everything, where processors share all memory and disk and the Shared-Nothing, where each processor has its own memory and disk and all inter-processor communication is done by message passing. Shared-Everything systems have the ability to efficiently perform load balancing but were traditionally limited to twenty or thirty nodes. Shared-Nothing systems can have hundreds of nodes but perform poorly with uneven loads.

Multiprocessor systems based on Non-Uniform Memory Architecture (NUMA) feature up to hundreds of nodes and provide a shared global memory. We investigate this architecture for the join operation and propose two main memory join algorithms that exploit well this shared architecture. We then introduce a dynamic load balancing scheme for the join algorithms to deal with uneven loads. We model both join algorithms and the load balancing scheme analytically and perform experimentations on commercially available NUMA multiprocessors.

We investigate a multiprocessor system based on NUMA architecture where nodes represent clusters of processors. We present a join algorithm with load balancing for this system and show that the architecture is well suited for parallel join processing. The performance of the system is evaluated by analytical modeling, simulation and actual experimentation. With this in hand, we study the effects of key system parameters, such as network and I/O bandwidths, processor speed and cluster size on performance. We show that the network bandwidth is a key limiting factor to increasing the system size, especially with small cluster sizes. This shows that systems based on this architecture perform better than Shared-Nothing systems. The reason is mainly because the clusters maintain a high level of locality and that load balancing is facilitated by the presence of shared memory.

In the name of God, most Merciful, most Beneficient

ACKNOWLEDGMENTS

First and foremost, my thanks go to my parents Rifaat Tout and Hafsa Hassan for all the love and patience they have shown through my long years of study. I just hope that I will be able to repay a small fraction of what I owe them. I owe a great deal to my wife, Nada Hoblos for all what she put up with during the course of my study and also to her family for their understanding and patience and for the precious gift they honored me with. My thanks go also to the rest of my family, my brothers Hicham and Samir for being by my side when I needed them (and when I did not). My sisters Aman, Mouna and Zeina for their encouragement. And last but not least, to my baby girl *Wala* whose coming into this world may have finally given me the boost I needed to wrap things up.

I would not have been able to complete the Ph.D. program if it were not for the help and guidance of my Thesis Advisor, Dr. Sakti Pramanik. My sincere thanks go also to my co-chair Dr. Carl V. Page. I owe both of them for their time and patience. I also like to thank the members of the committee, Dr. Lionel M. Ni, Dr. Matt Mutka, Dr. Jacob Plotkin and Dr. Gerald Ludden for their valuable comments on the thesis.

Over the years, many friends, officemates and many others not listed here have all encouraged me and gave me advice on various issues. Thanks go to all of them.

Finally I would like to thank all my brothers at the Center for being there when I needed them. I would especially like to thank Omar Soubani, Amr Azim, Amin Maklai, Iyad Saad and Naji Al-Arfaj for their support and Mohamad Naja and Khaldoun Rayes for their friendship and advice.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The need for parallelism in database systems

As databases expand beyond the capabilities of most mainframe systems, the interest in parallel architectures and algorithms for supporting and processing the relational queries has risen sharply. The ability of these architectures to deliver much higher performance than uniprocessor–based mainframes has been validated in both research and commercial parallel database systems [1, 2, 3, 4]. Most recently, *Oracle*, a company providing commercial relational database systems, demonstrated up to 2000 Transactions Per Second (TPS) on a new *HP* multiprocessor system [4]. According to *Oracle*, this is far beyond the capability of transaction processing of any mainframe system.

There is evidence of high demand for the processing capabilities of parallel systems to deal with the increasingly large amounts of information available currently and in the near future. In preparation for the coming of the *Information Highway*, Oracle is also experimenting with a 16–processor nCUBE machine to provide the Video on Demand service [5]. The nCUBE machine has 15 disks that can hold up to 40 digitized movies and can provide close to 42 simultaneous streams of video. A number of other companies such as *Time Warner Inc.* are looking to provide similar services and are thus investigating the use of parallel machines from other vendors such as from *Silicon Graphics* and *Digital Equipment Corporation* [5].

Recent statistics about current large databases indicate sizes ranging from hundreds of Megabytes to few Terabytes. The Customer Information and Package Tracking database of United Parcel Service is estimated currently at 700 Gigabytes and is expected to reach 1.2 Terabytes by mid 1995. The TRW, Equifax and Trans Union credit agencies have compiled over 500 million records on more than 160 million people throughout the United States [6]. With an average record size of only 5K per person, this amounts to more than 800 Gigabytes of data.

In a large number of applications, queries may require an examination of all tuples in a database relation. Given the sizes of the input relations in millions or even billions of tuples, this processing can take excessively long times. In relational database systems, queries consist of uniform operations applied to uniform streams of data. By partitioning the input data among multiple processors, a relational operator can be split into a number of independent operators with each working on a part of the data in parallel. With the shear volumes of data mentioned above, relational queries seem ideally suited for parallel execution.

The relational join is considered to be one of the most time consuming operations in relational database systems because it requires the scanning and processing of all the tuples of the corresponding relations. Most of the work in parallel database research has focused on the study of this operation. Consequently, the join has been used, by most researchers, as the standard performance measure for database systems. Most of the early work on parallel join was done in the context of database machines where the design of the parallel processing system relied heavily on specialized hardware components. Later and more recent work have shifted to general–purpose multiprocessor systems. Next, we start with a brief introduction of the join operator and present the time complexity for various algorithms implementing the join. We then give a general outline of parallel join algorithms.

Suppliers

S#	SNAME	CITY
S1	Smith	London
S2	Jones	Paris
S3	Blake	Chicago
S4	Clark	London
S5	Adams	Detroit

Parts

P#	PNAME	COLOR	CITY
P1	Nut	Red	London
P2	Bolt	Green	Paris
P3	Screw	Blue	Rome
P4	Screw	Red	London
P5	Cam	Blue	Paris
P6	Cog	Red	Chicago

The join of Suppliers and Parts where Suppliers.CITY = Parts.CITY

S#	SNAME	CITY	P#	PNAME	COLOR
S1	Smith	London	P1	Nut	Red
S1	Smith	London	P4	Screw	Red
S2	Jones	Paris	P2	Bolt	Green
S2	Jones	Paris	P5	Cam	Blue
S4	Clark	London	P1	Nut	Red
S4	Clark	London	P4	Screw	Red
S3	Blake	Chicago	P6	Cog	Red

Figure 1.1. A simple example of the relational join operator.

1.2 The relational join operation

The relational join operator, applied to the two relations R and S, is closely related to the Cartesian product of these relations. The Cartesian product concatenates each tuple from R with every tuple from S, while the join will combine only those pairs with a specified relationship. Formally, the relational join operator combines two relations, R and S, to produce a third relation, J, containing all tuple pairs from R and S with matching attribute values. Figure 1.1 provides an example of a join on two relations *Suppliers* and *Parts*.

1.2.1 Nested-Loops Join Algorithm

The simplest algorithm to compute the join is called the *nested–loops* join. With one of the relations designated as the *inner relation* and the other as the *outer relation*, the algorithm can be described as follows. For each tuple of the outer relation, all tuples of the inner relation are read and compared. Whenever the join condition is satisfied

between the two tuples, they are concatenated to form a new tuple in the output relation J. Therefore, the implementation of this algorithm requires $O(|R| \times |S|)$ execution time, where $|R|$ and $|S|$ represent the total number of tuples in the relations R and S respectively. Note that this is the same order of complexity as the Cartesian product.

1.2.2 Sort–Merge Join Algorithm

Another way to compute the join is to sort both relations, R and S, based on the join attribute values and then compare the sorted relations on matching join attribute values to form the output relation. This algorithm is known as *sort–merge* join. Since the merge phase has a linear time, the total execution time for this algorithms is dominated by the time of the sort phase. Hence the execution time of sort–merge has an order of $O(|R|\, log(|R|) + |S|\, log(|S|))$.

1.2.3 Hash–based Join Algorithm

Hash–join is an alternative method that offers a linear execution cost instead of the $O(|R| \times |S|)$ cost of nested–loops join or the $O(|R|\, log(|R|) + |S|\, log(|S|))$ cost of sort–merge join. In this method, for each tuple of relation R (S), a hash value will be computed by applying a hash function H_1 to the value of the join attributes of that tuple. The hash value determines the bucket where the tuple should be stored. This is referred to as hash–partitioning the relation R where the tuples in different buckets are totally disjoint with respect to join relationships. The relation S is hashed into a different set of buckets using the same hash function H_1. After both relations have been partitioned, corresponding buckets of R and S may be joined independently.

This method breaks a large join into a number of smaller joins since only corresponding buckets need to be checked. In order to join a pair of buckets, the R bucket

is read and tuples are organized into an in-memory hash table, using a second hash function, H_2. Next, the corresponding bucket of S is scanned, each tuple is compared against the hash table of R and matching tuples form the output (join) relation. In this case, R is called the building relations, referring to building the in-memory hash table and S is termed the probing relation.

As presented earlier, the sizes of individual relations in current and future databases may be in Megabytes and even Gigabytes. Even the most efficient uniprocessor implementations of the best join algorithms can not keep up with these extremely large amounts of data. By partitioning the data among multiple processors, the join can often be split into many independent joins, each operating on a single partition in parallel. This can greatly improve the execution time of the algorithm. Next, we present the general scheme for parallel join algorithms.

1.2.4 General Scheme for Parallel Join

Most of the work on parallel join algorithms has concentrated on hash-based joins [7, 8, 9, 10]. This section presents a general scheme that illustrates the basic workings of parallel joins. Chapter 2 will provide detailed description and analysis of a number of parallel join schemes proposed in the literature.

In parallel join algorithms, each relation is assumed to be initially partitioned among the N processors available in the system, i.e., each processor has a portion of both relations. This partitioning, which is based on a set of attributes, may be done using a number of schemes, such as hashing or range partitioning [11, 12]. This is also known as declustering the given relation. Parallel join aims to break up a large join of R and S into smaller joins that can be performed by different processors in parallel. To achieve this, the partitions of R and S, local to each processor, have to be totally disjoint from all other partitions, based on the join attributes. This is true if the join attributes of a relation are the same as the attributes used in the initial

partitioning of the relation. This may not be always the case, however. Hence, in general, the join algorithm requires the redistribution of R and/or S to guarantee that all partitions are disjoint.

To distribute the tuples of processor P_i, a destination processor P_j $(1 \leq j \leq N)$ is computed for each tuple based on its join attributes. The computation of P_j may be done using either hashing or range–partitioning. If $P_j = P_i$ then the tuple belongs to the local processor and consequently, it stays locally. Otherwise, the tuple is sent to the destination processor P_j. Once redistribution is complete, each processor will have a set of tuples consisting of the tuples that were kept locally and those received from all other processors. More specifically, node N_i will contain all the tuples of R and S whose destination node evaluated to N_i. Next, the join can be computed directly on the sets of tuples in each processor. The local join at each processor may be done using any of the schemes presented earlier in Section 1.2.

1.3 Data skew types and effects

In multiprocessor systems, relations are horizontally partitioned and distributed across all nodes. In performing the join, these partitioned data are processed in parallel by all the nodes in the system. If all nodes handle equal amounts of data, then maximum performance improvement can be achieved. When the distribution of loads on the various processors is unbalanced, it is likely that most processors in the system will stay idle waiting for few overloaded processors to finish. This load imbalance results in poor overall performance.

It has been shown that the major source of load imbalance in parallel processing of database applications is data skew [13, 14, 15, 16]. Data skew is defined as the imbalance in the distribution of tuples to processors. This skewness in data distribution may be attributed to the following reasons:

- Horizontal partitioning or how the data is initially partitioned among the processors is usually based on the values of the join attributes in the given relation. The values of these join attributes may not distribute uniformly over the tuples of the relation.

- For most queries involving join, a general query optimization is to perform all *select* operations before the join. The results of the selectivity are likely to vary at different processors. Thus, even if the processors start with evenly distributed tuples, the results produced by the selection operations are likely to be skewed.

- In hash–join algorithms, tuples from each relation are first hashed and distributed among all processors and then the join is performed. Lakshmi and Yu [15, 17] stated that in data sets belonging to real–life databases, the join values are inherently skewed. For example, the field *part_origin* for the *part* 'VCR' is likely to contain the value 'Japan' more than any other value.

  In the presence of such skew, an unbiased partitioning scheme, such as hashing, will result in unequal loads on the various processors in the system. Even within the same processor, the sizes of the various buckets can be quite different. Depending on the rate of skew, few buckets may overflow while others contain only a small number of tuples.

Since relation partitioning has been the primary method of distributing the load for the join operation, the effects of skew on the performance of the join may be severe. The nonuniformity in the sizes of the partitions, where some partitions may be significantly larger than others, means that the site with the larger load dictates the performance of the join as it becomes the main performance bottleneck.

1.4 Dissertation Outline

The rest of the dissertation is organized as follows:

Chapter 2 provides essential background on the evolution of parallel database systems and join algorithms. It outlines some of the early work on database machines and traces the transition of the research to parallel database systems for general-purpose multiprocessors. The chapter also presents a review of the recent work on load balancing schemes to deal with data skew in parallel join algorithms. A summary of the major advantages and disadvantages of each scheme is presented toward the end of the chapter. We elaborate further on this in Chapter 4 where we introduce our load balancing scheme for parallel hash join.

Chapter 3 introduces a new main-memory join algorithm that is suited for NUMA architectures. In some cases, where the initial *selection* step may greatly reduce the size of one of the input relations, the distribution of the larger relation may no longer be justified. Hence, we investigate the performance of the join algorithm where the smaller relation is fully replicated to all processors in the system. We analytically model both join algorithms and compare the analytical results to actual experimentations.

A dynamic load balancing scheme for main-memory multiprocessor systems that effectively deals with data skew is presented in Chapter 4. The load balancing scheme is applied to the Distributed Hash Join introduced in Chapter 3 and an analytical model is developed. The quantitative performance of the balancing scheme is investigated both analytically and experimentally. Experiments were conducted on both the BBN GP1000 and the BBN TC2000 multiprocessor systems.

It is suggested that parallel database systems on architectures with hierarchical network organization exhibit better performance for executing joins on very large relations [18]. Chapter 5 investigates this idea for dealing with data skew in parallel

join. The analysis is based on various system parameters such as CPU speed and network and I/O bandwidth. We extend our load balancing join algorithm for the new architecture and develop an analytical model to investigate the performance of the system. We develop a simulator for the system using the CSIM process–oriented simulation language. We also perform actual experimentation on the KSR1 and validate both the analytical model and the simulator. The architecture of the KSR1 is modeled by adjusting the values of certain system parameters. We project on the performance of the system, using both the analytical model and the simulator, by varying a number of key system parameters.

The dissertation is concluded in Chapter 6. This chapter also reiterates the contributions of the research and outlines possible future research directions.

CHAPTER 2

PREVIOUS WORK ON PARALLEL JOIN ALGORITHMS

Parallel processing of relational join has been a very active area of research and a large number of algorithms have been proposed [7, 19, 9, 20, 21]. These algorithms range widely from those highly dependent on specialized hardware [10] to algorithms designed for general–purpose multiprocessor systems. Next, we briefly describe some of the early works on parallel database systems. We then present some of the recent works that are more closely related to this thesis.

2.1 Early work

During the late 1970 through early 1980, database machines were the main focus of research on parallel database systems. Thus, most of the early work on parallel join algorithms was done mainly in the context of these machines. Database machines consist of the hardware and software designed to operate on very large database systems efficiently. These machines are most notable for their use of specialized hardware components to efficiently implement, totally or partially, a number of database operations. Parallelism was generally achieved by one of two approaches: *Reduction* and *Pipelining* [22]. Many of the complex computational tasks may be decomposed into a number of parallel subtasks with approximately equal computational complexity. Thus, a complex task can be reduced to a number of subtasks that are assigned to dif-

ferent processors and executed in parallel. This approach, namely reduction, exploits the horizontal parallelism inherent in most database operations.

For example, a complex database query may be decomposed into a tree structure of primitive database operations such as Selection, Projection and Join. The primitive operations at the same (horizontal) level of the tree can be executed in parallel. In addition, the data for a single primitive operation may be horizontally partitioned across a number of processors where each processor operates on its partition in parallel.

The second approach, namely pipelining, uses a linear array of specialized processors to perform a number of different functions required to complete a complex database query. A portion of the data required by the task is given to the first processor which transforms it in some way and passes the result to the next processor in the pipeline. While the first processor is working on the next portion of the data, the processor next to it, may work on the data portion it received from the first processor.

While reduction may be completely implemented in software [23, 24, 25, 26], pipelining, as conceived by the early research, relied heavily on specialized hardware, such as hardware filters and sorters [23, 27, 28, 29]. More recent work on parallel databases implements the pipelining of operations between the different levels of the query tree and does not rely on specialized hardware support for the various operations. Instead, intermediate results that are produced by a set of processors may be routed to another set, or even the same set of processors, in order to perform the next level of primitive operations within the query tree [1, 3].

We now describe some of the specialized hardware components used in the design of a number of database machines and note the systems that used them. Disk cache was used to accelerate the effective bandwidth of disk storage devices as in DIRECT [30, 31, 32], CADAM [33], IDM [34, 35], DDM [36, 37], GRACE [27, 28, 38, 39], SABRE [40, 41, 42, 43] and DELTA [44, 45, 46]. Today, disk cache is provided with

most disk drives and is currently considered an integral part of the drives' subsystem. GRACE and EDC [47] used magnetic bubble memories as secondary storage devices. SM3 [48, 49, 50, 51] used switchable memory modules to facilitate the passing of data, status and messages among a number of processors. Specialized hardware sorters were employed in DDM, DBC/1012 [23, 24], GRACE and RDBM [52, 29]. Hardware filters were used in GRACE, CADAM [33], DBC, SABRE and RDBM. Sorters are still used today in the DBC/1012 database system which is one of the very few database machines that survived to this date and is currently available commercially. MICRONET [53, 54, 55, 56] and SM3 relied on hardware control lines for fast inter-processor synchronization and communication. In terms of memory and storage, content-addressing hardware was used in DBC, CADAM, HYPERTREE [57, 58] and DDM and cellular logic in the form of associative secondary storage was used in EDC and DBC [59, 60].

These specialized hardware components formed an essential part of the design of database machines. Next, we present a number of such machines and highlight their dependence on the specialized components.

2.1.1 DIRECT

DIRECT was designed and prototyped at the University of Wisconsin in the late 1970s [30, 31, 32] to support highly parallel processing of a set of relational queries. The system had an MIMD architecture and consisted of a number of query processors (QP), a set of charge-coupled device (CCD) memory modules used as cache memory and a set of mass storage devices, all interconnected by a novel crossbar switch. Each CCD is connected to all query processors via special data lines. The CCDs continuously broadcast data through the data lines connected to all QPs. This design exhibits a number of desirable features:

1. Broadcasting of relation pages, since all QPs in the system can read the target page from the same CCD by simply switching to that CCD (using the crossbar switch).

2. Parallel accesses to relation pages, as individual QPs can simultaneously access different CCDs.

3. Fine–grain accesses to tuples at both page and tuple boundaries, since QPs can switch between CCDs at any given interval of time.

The weak point in the architecture of DIRECT is that relation pages must be moved very frequently between the disk, the CCDs and the QPs. The overhead of such constant data movement degrades the performance sharply and limits the scalability of the design [61]. Memory contention is also possible when moving data between the disk and the CCDs and between the CCDs and the QPs. Another problem with this design is its reliance on the specialized crossbar switch. The crossbar switch becomes prohibitively expensive when the total number of nodes in the system increases. This reliance severely limits the scalability of the system.

2.1.2 MICRONET

MICRONET is a multiprocessor system designed and prototyped at the University of Florida [53, 54, 55, 56]. The system consists of a number of microcomputers interconnected via a custom–built bus called the MICRONET. One of the microcomputers is designated as the *control* computer while all others are considered to be *data* computers. The MICRONET bus provides simple hardware facilities such as:

1. Interprocess communication, synchronization and control,

2. Resolving bus access contention,

3. Broadcasting data and commands.

The MICRONET system achieves a number of desirable features such as reliability and scalability. However, its performance and capability to implement most of the database operations, such as sorting and data filtering, rely heavily on the custom-built hardware.

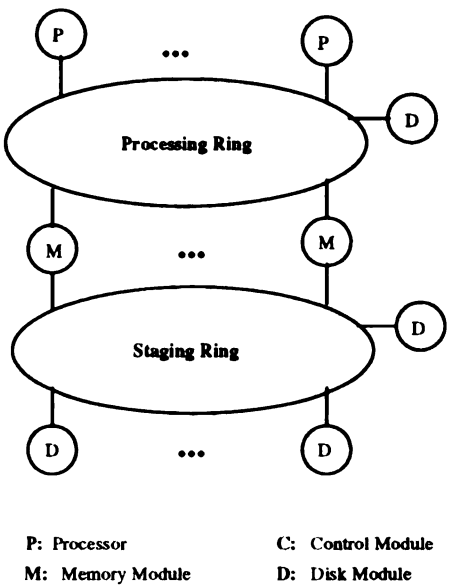

Figure 2.1. Architecture of GRACE.

2.1.3 GRACE

GRACE [28, 62, 63, 39] is one of the few database systems actually implemented and was part of the Japanese Fifth Generation Computer Initiative. The system was designed and implemented at the University of Tokyo, Japan. This database machine was one of the first multiprocessor database systems to apply hashing techniques to the join operation. The system is organized into two rings: the *staging* ring and the *processing* ring. The disks filter and hash the data into the appropriate memory modules on the staging ring. Next, the data is partitioned into buckets by using

dynamic hash clustering and the buckets are distributed to the memory modules. Each processor is assigned a bucket and the extra buckets (in case the number of buckets exceeds the number of processors) are processed serially. When tuples arrive into a processing node, they are sorted in a pipeline merge sort unit. The join executes in $O\left(\frac{(|R|+|S|)}{m}\right)$ time, where $|R|$ and $|S|$ are the sizes of the first and second relations and m is the number of disks involved, respectively.

2.1.4 Teradata's DBC/1012

The DBC/1012 [24] database machine was developed by Teradata Corporation and consists of a set of Interface Processors (IFP), Access Module Processors (AMP) and Disk Storage Units (DSU). The IFPs send and receive results from the AMPs which perform the storage and retrieval of data on the DSUs. All relations are partially declustered across multiple AMPs, that is each relation is horizontally partitioned across a number of the AMPs available in the system. The IFPs and AMPs are connected via a redundant, tree–structured network, the *Ynet* which is an active bus providing logic circuitry to perform selection and sorting on the data as it travels. Requests are broadcast to the corresponding AMPs and results are are sorted when they return up the Ynet and then broadcast to the requesting IFP.

The DBC/1012 provides alternative ways to execute the join in order to facilitate performing the join for different cases. The most common alternative, however, involves hashing and distributing the source relations to the involved AMPs. Tuples are then sorted in each AMP and the join is performed using the sort–merge join algorithm.

Teradata has installed many systems containing over 100 processors and several hundred disks. These systems have demonstrated near–linear speedup and scaleup on relational queries and have been able to efficiently maintain and process very large (Terabyte) databases.

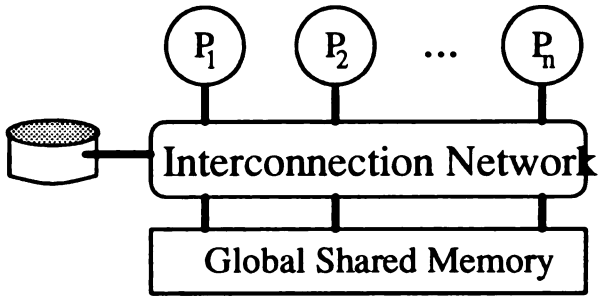

Figure 2.2. A typical Shared–Everything Multiprocessor system. Each node has access to all other nodes in the system. The access includes memory and disks.

The major disadvantage of database machines is their reliance on specialized hardware components. This reliance may render a database machine useless in a short period of time as new general purpose systems are able to provide the same functionality and performance for a much lesser price. Specialized hardware can benefit large database systems in terms of performance gains for special operations, as evidenced by DBC/1012 and GRACE. The DBC/1012 is still available commercially and its Y–net has been recently upgraded [5]. GRACE is still under investigation as part of the Super Database Computer project at the University of Tokyo [39, 10, 64]. However, in most recent works, this hardware reliance has been abandoned and most of the functionalities were implemented in software. Database machines have been replaced by parallel database systems running mostly on top of general–purpose multiprocessor systems. In the next section, we present a representative set of these parallel database systems and discuss critical design issues relevant to the work of this thesis.

2.2 Recent work on parallel database systems

The two paradigms that have dominated parallel database research in the recent past are the Shared–Nothing (SN) and the Shared–Everything (SE) architectures. In SE systems (Figure 2.2), processors share all memory and disks. Data is mainly stored

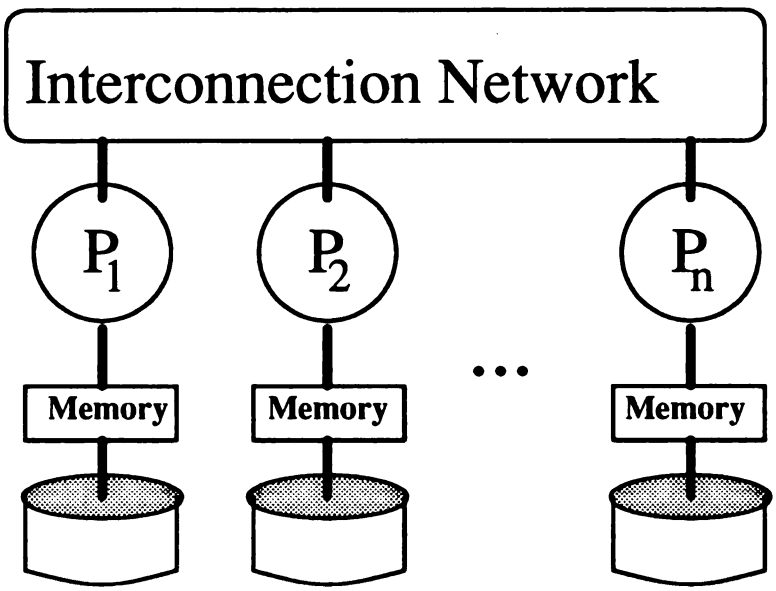

Figure 2.3. The Shared-Nothing architecture. Here, nodes do not share memory nor disk. All interactions between the nodes are done via message passing.

on disk and transferred to the common memory in fixed size pages. This paradigm is the basis for the XPRS [65] database system. In the *SN* architecture, each processor has its own main memory and disks and all interactions between processors are done via message passing (see Figure 2.3). An example of this architecture is the GAMMA database system [66]. These two paradigms are defined more in terms of inter-processors communications and sharing of resources than in terms of the underlying hardware. For example, the BUBBA database prototype [3] is considered a shared-nothing system even though it was implemented on top of a FLEX/32 multiprocessor system, a shared-memory machine.

Next, we present a number of parallel database systems developed on top of general-purpose multiprocessors and classify the into their corresponding paradigm.

2.2.1 XPRS

The XPRS system, developed at the University of Berkeley is a high performance multiprocessor database system. The goal is to demonstrate that high performance for transaction processing and complex ad–hoc queries can be provided by a next generation database system running on top of a general–purpose multiprocessing system. XPRS is based on a shared–everything architecture. The system features a query optimizer that can choose a good access plan based on the available buffer space and available processors. In addition, the access plan generated by the optimizer must be amenable to parallelism.

2.2.2 Volcano

The Volcano system [67] is being developed at the University of Colorado. Volcano is a dataflow query processing system that is extensible and incorporates parallelism. The system provides an operator model of parallelism, more specifically, it has an *exchange* operator that parallelizes all other operators. The encapsulation of parallelism in Volcano allows for new query processing algorithms to be coded for single–process execution but run in a parallel environment without modifications. Volcano allows processors to operate on different data subsets in parallel and allows pipelining between different operations.

Some experimental results were reported concerning the overhead of the exchange operator when executed on a shared memory multiprocessor system [67]. The results show that the operator is very fast. The granularity of data exchange between various processors show that small packet sizes produced a severe performance penalty.

2.2.3 BUBBA

The BUBBA prototype was implemented using a 40–node FLEX/32 multiprocessing system with 40 disks [68, 3]. Although the Flex/32 is a shared–memory machine, BUBBA was designed as a shared–nothing system. The shared–memory was only used for message passing. The system contained three groups of processors: Interface Processors (IP) for communicating with external host processors and coordinating query execution; Intelligent Repositories (IR) for data storage and query execution; and Checkpoint/Logging Repositories (CLR). Relations were partitioned among the various IRs using either hashing or range partitioning. Redistribution was applied based on collected statistics about the usage of each relation. A unique feature of BUBBA was its use FAD instead of SQL as the interface language. FAD is an extended-relational persistent programming language. It provides support for complex objects via several type constructors. The FAD compiler is responsible for detecting operations that can be executed in parallel. This is based on the way the data objects being accessed are partitioned. It was noted that the task of compiling and parallelizing a FAD program is significantly more difficult than parallelizing a relational query.

2.2.4 GAMMA

GAMMA [1, 2, 19] is a fully operational prototype of a shared–nothing database system developed at the University of Wisconsin, Madison. GAMMA is based on the experience from the earlier multiprocessor database machine prototype, DIRECT. It originally consisted of 17 VAX 11/750 processors, each with two Megabytes of main memory. The processors were connected together, with another VAX serving as the host machine, by a token ring. Only eight of the processors had disk drives attached to them. More recently, GAMMA has been ported to a 32 node iPSC/2

Intel hypercube [19]. Each node is configured with an Intel 386 CPU, 8 Megabytes of memory, and a 330 megabyte MAXTOR 4380 disk drive. The nodes are connected to form a hypercube using custom VLSI routine modules. Each module supports eight full–duplex, serial, reliable communication channels operating at 2.8 Megabytes/sec.

All relations in GAMMA are fully declustered, *i.e.* horizontally partitioned across all disks in the system. A number of declustering strategies are provided by the system to maximize the performance of different queries in the system. The Hybrid–Range partitioning scheme was introduced by Ghandeharizadeh and DeWitt [12] and was shown to provide support for small relations and for relations with skewed distributions. The same authors [11] also investigated the performance of various declustering strategies based on round–robin, hash and range partitioning. Given T tuples and N nodes, round–robin partitions a relation according to the following: place tuple number i in node $mod(i, N)$, where $mod(i, N)$ represents the modulus of i and N. Hash partitioning, decides the location of a tuple based on the value of a hash function applied to a set of attributes of that tuple. And finally, in range partitioning, the total range of tuples is divided into N subranges, and each subrange is assigned to the corresponding processor. The study concluded that each of these strategies outperform the others for certain types of queries. This work is based on the work by Kim and Pramanik in [69] where optimal declustering was studied for various sets of queries.

The major difference between GRACE, DBC/1012 and other SN systems is the reliance of these two systems on special–purpose hardware, *e.g.* the Ynet in DBC/1012 and the Omega network in GRACE. As mentioned earlier, the majority of SN systems are implemented on top of general–purpose multiprocessors systems. In their article on future database systems, DeWitt and Gray [70] note that while GRACE and DBC/1012 have demonstrated good performance, only time will tell whether special–purpose components can offer better price/performance than SN systems that use

conventional hardware

Most of the previously mentioned algorithms were designed with the assumption of uniform distribution of tuples. In general, this assumption is not valid and data skew is common in real-life database systems. The effect of data skew on four join algorithms was explored in [19] on a 8 processor version of the GAMMA database system. The study concluded that the parallel hash-based join algorithms, namely Hybrid, Grace and Simple hash, are sensitive to data skew especially when the skew rate is high. This study, however, did not consider any additions or modifications to the join algorithms that would allow them to deal with the problem of skew. The reason may be related to the fact that load balancing, on SN architectures, causes a large overhead as the number of control message increases with increasing number of nodes [18]. The next section surveys some of the recent work on dealing with data skew in parallel join algorithms.

2.3 Parallel join and load balancing

Ideally, nodes participating in the join operation would handle approximately equal amounts of data. This, however, is not generally the case and most real-life data exhibit some degree of skewness, as noted in Section 1.3. More recent work on parallel joins have developed various schemes to deal with the effects of data skew. Next, we briefly present and discuss a number of these schemes. Table 2.3 presents a quick overview of the main highlights of each scheme. The third column of this table summarizes what may be considered as the major disadvantages and potential problems of each scheme.

2.3.1 Bucket–converging and bucket–spreading join

Kitsuregawa and Ogawa [10] describe two algorithms, *bucket–converging parallel hash–join* and *bucket–spreading parallel hash join.* The first algorithm is a parallelization of the GRACE join algorithm and works as follows. All subrelations of R are read from disk in parallel and are partitioned into p buckets (where p is much larger than N, the number of nodes in the system). The size of each bucket is examined and, if necessary, enough buckets are redistributed so that the sums of the buckets' sizes at each processor are balanced. Next, S is processed similarly and in the last phase, all respective buckets of R and S in each node are joined locally.

The first phase of this algorithm is very susceptible to data skew that may occur during the distribution of tuples. The *bucket–spreading parallel hash join* algorithm is designed to deal with this problem. In this algorithm, both relations are partitioned into p buckets, but instead of statically assigning buckets to nodes, each bucket is horizontally partitioned across all nodes during the first phase. During the second phase of the algorithm, the buckets are redistributed onto the nodes using a very sophisticated network, the Omega network. This network contains logic to balance the load during the bucket redistribution. Simulation results show the algorithm to be effective in limiting the effect of distribution skew.

Note that both algorithms require redistributing the relations twice in order to produce a balanced load across all nodes. Also, the bucket–spreading algorithm relies heavily on the built–in balancing capabilities of the Omega network hardware.

2.3.2 Adaptive load balancing join

Hua and Lee [71] proposed three parallel join algorithms for dealing with skewed data. The first algorithm, *tuple interleaving parallel hash join* is mainly based on the *bucket–spreading hash–join* [10]. The major difference between the two algorithms is that

the tuple–interleaving algorithm does not require the *Omega* network that is needed by the bucket–spreading hash–join algorithm. The second algorithm, *Adaptive Load Balancing parallel hash–join*, is basically identical to the bucket–converging algorithm of [10].

The third algorithm, *Extended Adaptive Load Balancing* parallel hash–join, is designed mainly for the severe cases of date skew. After the relations R and S are partitioned and distributed, each nodes reports the sizes of its local R and S buckets to a central coordinator. Based on the reported information, the coordinator will decide on the allocation of buckets to nodes. The decision is broadcast to all the nodes in the system and the buckets are redistributed accordingly.

All three join algorithms are compared using analytical models. The basic results indicate that the *tuple–interleaved* and *extended adaptive* load balancing algorithms are unaffected by skew in the size of the partitions. Such skew does however affect the performance of the adaptive load balancing algorithm which eventually becomes much worse as the skew increases.

2.3.3 Scheduling–hash join

Wolf *et al.* [9], propose the *scheduling–hash* join algorithm for dealing with the cases of severe data skew. The relations R and S are hashed into local buckets and statistics are gathered while building each bucket. Next, a scheduling phase is run where a central coordinator collects all the statistics and computes an allocation strategy of buckets to nodes. This allocation strategy is broadcast to all nodes and the relations are accordingly redistributed over the network.

A number of heuristics are proposed for the computation of the allocation strategy including, *longest processing time first, first fit decreasing,* and *skew.* The different strategies were compared analytically and were shown to be highly effective in balancing the load especially as the number of odes becomes larger. Like Hua and Lee's

schemes, these strategies require extra scanning steps to be added to the join algorithm for collecting statistics about the various buckets of R and S. These extra steps increase the actual computation time for the algorithms and exhibit a severe effect especially with a moderate to low skew rate.

Walton *et al.* [72] presented a taxonomy of data skew for parallel database systems and a modified version of the the Scheduling Hash–join, to deal with the data skew. The algorithm used gathered statistics to schedule parts of the join on different processors in order to balance the load. An analytical model was used to compare the performance of Scheduling Hash–join with that of Hybrid Hash–join. It was found that Scheduling Hash handles skewed data effectively while Hybrid Hash degrades and becomes eventually worse than Scheduling Hash. However, unless the data skew rate is significant, Hybrid Hash stays significantly better than Scheduling Hash. This is because of the overhead needed by the latter to gather the needed statistics used in making the load balancing decisions.

2.3.4 Load balancing hash join

Omiecinski [20] proposed a load balancing hash–join algorithm for shared–memory multiprocessor systems. The algorithm is based mainly on the bucket–spreading algorithm of Kitsuregawa and Ogawa [10]. The major difference is that, like Hua and Lee's algorithms, it does not require any special hardware support for the redistribution of tuples. Analytical and limited experimental results on a 10–processor Sequent Balance multiprocessor were presented. It was shown that the algorithm is effective in limiting the effect of data skew. The algorithm does, however, suffer from the extra statistics collection steps needed for the redistribution in the cases of moderate and low skew rates.

Most of the above mentioned schemes attempt to balance the load by scanning the relations to derive statistical information. Both relations may have to be com-

pletely scanned before the join begins. While this may succeed in minimizing skew, performance may suffer greatly when the data is not skewed. The time to perform the parallel hash join is usually a small multiple of the time required to scan the two relations. Thus, the preprocessing of relations results in a substantial overhead that may only be justified for the cases of extreme data skew. There is little evidence, however, that such extreme levels of skew occur commonly in practice [21] and consequently, it may not be justified to penalize the normal case in order to benefit a few extreme cases.

2.3.5 Load balancing in Shared Virtual Memory

Shatdal and Naughton investigated the use of Shared Virtual Memory (*SVM*) [73] on top of shared-nothing to deal mainly with the problem of data skew. Shared virtual memory is commonly used on top of physically distributed memory to provide a single virtual address space shared by all processors in the system. This is accomplished by using memory mapping managers responsible for providing the processors with access to all available memory. These memory managers are also responsible for keeping the system-wide memory coherent. The protocols for such access and coherency control are beyond the scope of this thesis and can be found in more details in [74].

A new join algorithm, *SVM-join*, is introduced where the shared virtual memory is used to efficiently balance the load across all processors in the system. The algorithm was carefully designed to avoid any network or disk thrashing caused by multiple nodes updating shared memory pages or by too many pages being sent to the same node, respectively. In contrast to previous algorithms, such as those described earlier in this section, the *SVM-join* does not incur any overhead in the case of small or no skew. Its performance is virtually identical to that of hybrid hash under these conditions. The performed simulations show the join algorithm to be effective in dealing with data skew and to improve on *SN* schemes. Their model, however,

assumes the network to have infinite bandwidth. Such assumption oversimplifies the model since the bandwidth of the interconnection network is considered to be one of the primary limiting factors of scalability in parallel systems.

2.3.6 A hybrid system

More recently, Hua and Lee [18] proposed a hybrid system where Shared–Everything multiprocessors are connected by a network to form a Shared–Nothing system. The authors cited an example system consisting of multiple IBM 3090/600 multiprocessors linked together with an interconnection network. These multiprocessors can have up to six nodes and provide shared memory. The performance of this hybrid system was studied using the join operator as the workload model. The model assumes that only the relation S needs to be redistributed during the first phase of the join and that this distribution will result in data skew on the receiving nodes. The skew was modeled as a fraction θ of the total relation size being assigned to one node while the rest of the relation is distributed evenly among all other nodes.

The join algorithm used in the model is a modified version of the work by DeWitt and Gerber [7]. Assuming N to be the total number of nodes in the system, the algorithm works as follows:

1. *Distribution phase:* Each node i reads its subrelation R_i from disk. A hash function H_1 is applied to each tuple of R_i and the result, p ($1 \leq p \leq N$), becomes the destination node for that tuple. Tuples are sent to their destination nodes using the local bus, for nodes that belong to the same multiprocessor, or using the interconnection network for all other nodes. All received tuples are written to disk. Repeat the same procedure for S_i.

2. *Join phase:* Let R'_i and S'_i be, respectively, the subsets of relations R and S on nodes i after the distribution phase. Each node reads R'_i from disk and hash it

using hash function H_2 to produce B buckets, $R'_{i1}, ..., R'_{iB}$. At the end of this step, all resulting buckets are on disk and processing of S'_i starts similarly.

In order to complete the join, each node reads a pair of corresponding buckets R'_{ij} and S'_{ij} and join them separately.

Assuming there is no overlap between the two phases of the join algorithm, a number of cost evaluation functions are constructed using a set of workload parameters. The cost model includes the individual costs for the CPU, the memory, the network and the I/O. Each cost is computed separately, for each phase, and the maximum cost is taken as the cost of the corresponding phase. Using the cost functions, the system is investigated for different values of CPU speed and network, memory and I/O bandwidth. With the stated assumptions, the I/O bandwidth per cluster is determined to have the most effect on performance. Based on this conclusion, an optimal cluster size of 7 nodes is computed.

Note that, in the description of the algorithm, incoming tuples are written to disk immediately after they are received. These tuples are hashed into buckets during the join phase of the algorithm. This requires an extra pair of read and write per tuple which can explain the heavy contribution of I/O to the overall cost. Also, the role of the network is limited to only the distribution phase, since during the join phase, no tuples are exchanged between nodes. However, since the clusters are connected to form a SN architecture, the problem of balancing the load across clusters still exists. If load balancing is to be applied during the join phase, the network cost will contribute more to the total time of the algorithm. We show later, in Chapter 5, that the network bandwidth has a strong impact on the total number of clusters. Another problem is that the cost of memory accesses was computed separately from the processing cost, thus ignoring the effect of memory access time on processing cost. This computation is not accurate for most systems as a processor normally waits for

requested data to arrive from either main or cache memory into local registers before it resumes execution.

Algorithm	Description	Comments
Bucket Converging	- Partition relations into p buckets. - Examine each bucket and balance by redistributing enough buckets.	- Relations are distributed twice - Suseptible to skew
Bucket Spreading	-Parition relations into p buckets but partition each bucket across all nodes. - Redistribute using Omega Network.	- Relations are distributed twice - Relies on Omega Network
Tuple Interleaving	- Similar to bucket–spreading but redistribution is done in software.	- Relations are distributed twice
Adaptive Load Balancing	- Similar to bucket–converging.	- Relations are distributed twice
Extended Adaptive Load Balancing	- Designed for severe data skew. - Buckets sizes are reported to central coordinator (CC). - CC computes a bucket allocation plan and broadcast it to all nodes.	- Centralized control - Reliability of allocation plan
Scheduling Hash Join	- Gather buckets statistics. - A CC computes and broadcast a bucket allocation plan. - The *Longest processing time first*, *first fit decreasing* and *skew* hueristics are used for plan computation.	- Cost of bucket statistics - Centralized control
Load Balancing Hash Join	- Similar to Bucket–Spreading. - Does not rely on hardware.	- Cost of statistics collection
Load Balancing in Shared Virtual Memory	- Partition and distribute relations. - Nodes that are done find busy nodes to help. - Busy nodes decide, for each tuple, whether to join it locally or send it to the helping node(s)	- Can generate high network traffic. - If busy node is processing then helping node idles
Hybrid System	- System consists of clusters of SE connected to form a SN system - Perform load balancing inside each cluster	-Efficient balancing inside clusters - No cross clusters balancing

Table 2.1. Summary of previously proposed join algorithms with load balancing.

CHAPTER 3

MAIN MEMORY HASH–BASED JOIN ALGORITHMS FOR MULTIPROCESSORS WITH *NUMA* ARCHITECTURE.

3.1 Introduction

In some cases, the size of the database is limited or is growing at a slower rate than current trends in memory capacities [75, 76]. For a number of real time applications, such as telecommunications or radar tracking, the data must be memory resident in order to meet real-time constraints. In such cases, the database is necessarily smaller than the amount of available main memory [75]. Examples of main memory database systems are IBM's QBE database project [77, 78, 79], the MM-DBMS system at the University of Wisconsin [80], the MARS MMDB at Southern Methodist University [81, 82], and IBM's IMS/VS Fast Path [83], which is commercially available.

New Shared–Memory multiprocessor systems, such as the commercially available KSR1 [84] or the Stanford DASH research prototype [85, 86], are scalable far beyond the capabilities of earlier shared–memory systems. These multiprocessors provide a large number of nodes and very large main memories and as such, offer a suitable architecture for parallel database systems.

Next, we present the current trends in shared–memory multiprocessors and briefly discuss the issues relevant to our work on parallel database systems. We then proceed

to introduce the parallel join algorithms for NUMA architecture.

3.2 The Multiprocessor Architecture Model

Shared memory multiprocessor systems can be divided into the following three groups:

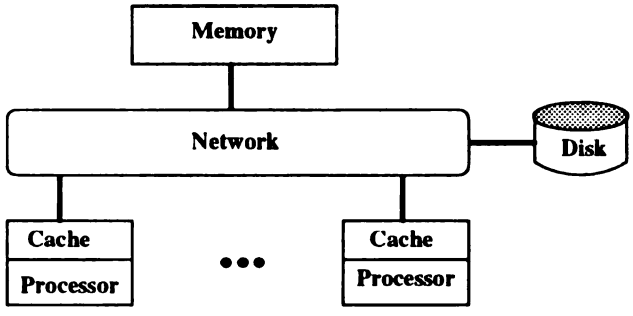

Figure 3.1. A typical Multiprocessor with UMA architecture. Nodes in the system access the physically shared memory through the network, typically a bus.

1. **Uniform Memory Architecture (UMA)**: Systems in this group are usually based on a single bus connecting a number of processors, each having a local cache, to a large physically shared memory. The UMA architecture is presented in Figure 3.1. The bus provides uniform access time from all processors to the shared memory. Contentions for the common memory and the common bus limit the scalability of these systems. Multiprocessors made by Encore and Sequent are based on UMA and can provide up to 30 processors.

2. **Nonuniform Memory Architecture (NUMA)**: These systems consist of a number of nodes, each with a local memory, linked together by an interconnection network. Nodes may contain local caches and have shared access to all memories in the system through the network. Figure 3.2 shows a typical NUMA multiprocessor system. As the shared memory in these systems is distributed

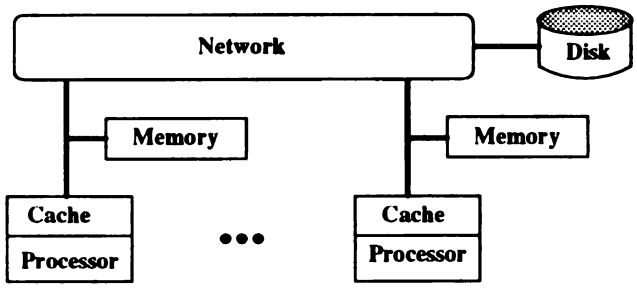

Figure 3.2. A typical Multiprocessor with NUMA architecture. Each node in the system has a processor and local main memory. The interconnection network provides each node with shared access to the physically distributed memory.

across all nodes, access times to different parts of the memory can vary and are thus, nonuniform. The BBN GP1000, TC2000 [87] and the IBM RP3 represent commercial NUMA multiprocessors where each node contains a single processor and the network consists of a Multistage Interconnection Network. The total number of processors in these systems can be much higher than in UMA based systems*[87].

Example research prototypes based on NUMA architecture are the Stanford Directory Architecture for SHared memory (DASH) [85, 86] and Paradigm [88] projects and the Encore GigaMax. In these prototypes, nodes are clusters consisting of a local memory and a number of processors, each with a local cache, connected by a bus. The clusters are connected by an network (which may also be a bus) that provides shared access to the distributed memories.

3. **Cache–Only Memory Architecture (COMA)**: The memory organization of COMA systems is similar to that of NUMA in that each node has a portion of the total memory space. However, there is no notion of main memory as all local memories are organized like large (second level) caches. A typical system based on COMA architecture is shown in Figure 3.3. The KSR1 [84] from Kendall

*The BBN GP1000 has up to 250 nodes and the TC2000 can support up to 504 nodes

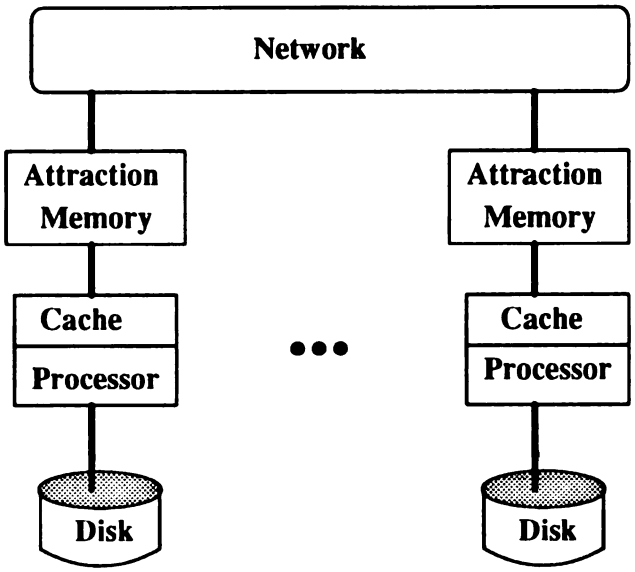

Figure 3.3. A Multiprocessor with COMA architecture. Each cluster–node can have a number of processors, each with a local cache (subcache) and a large second–level cache (main memory).

Square Research is the first commercial system based on COMA architecture. This currently available system supports up to 34 cluster–nodes, each having 32 processors for a total of 1088 processors.

The Swedish DDM [89] is a COMA–based research prototype developed at the Swedish Institute of Computer Science. In DDM, each node represents a single–bus multiprocessor system with physically distributed memories. These memories form a second–level cache called, Attraction Memories. The DDM features a hierarchical design where a number of nodes can be connected via a bus to form a cluster. A number of these clusters can be connected by a bus and grouped into a larger cluster and so on.

Most of the previous research on parallel join algorithms for shared memory multiprocessors has focused mainly on UMA–based systems [90, 91, 92]. As noted earlier, contention for the shared bus and memory as well as cache coherency problems limit the scalability of these systems. This restriction on scalability is one of the major

arguments by DeWitt and Gray in [70] to select Shared–Nothing as the architecture of choice for parallel database systems. NUMA multiprocessor systems, however, provide a large number of nodes and a very large, shared main memory [87, 86]. With new cache coherence schemes such as the Directory based coherency or partial snooping schemes, these systems can have a large number of processors with little overhead.

In NUMA–based systems, the globally shared memory is the sum of the memories local to all processors. Thus, the size of available main memory increases with increasing number of nodes in the system. The BBN TC2000 system can have up to 504 processing nodes with each having up to 32 MBytes of local memory. This sums up to a maximum of 16 Gigabytes of available memory for a fully configured system [87]. Thus, multiprocessors based on NUMA architecture seem well suited for parallel main memory database processing.

To design efficient algorithms for NUMA–based systems, certain architectural issues have to be taken into consideration. Since access times from a given processor to different memories can vary, locality of reference becomes important. Algorithms should be designed to maintain a high degree of locality in order to minimize both the access times and the network load. Another important issue is regarding the use of globally shared locks for synchronization purposes. Locking may cause a high rate of memory access conflicts due to the *Hot–Spot effect* [93, 94]. Thus, the use of the globally shared locks should be carefully designed to minimize the adverse effects of hot–spots.

3.3 Parallel Join on *NUMA* Architecture

In this section we present two parallel algorithms for implementing the relational join operation on NUMA architecture. The first algorithm, Distributed Hash Join

(DHJ), is designed specifically for main memory database systems on NUMA-based multiprocessors. The second algorithm, Full Replication Join (FRJ), is designed for the specific cases when one of the relations is much smaller than the other. FRJ is based on the replication of all the tuples of the smaller relation.

In previous works [91, 90], a global hash table of buckets is built in the shared memory and then probed by all processors in parallel during the hashing and the joining phases. Access synchronization is provided by using globally shared locks. While this may be acceptable for UMA–based architectures (where the number of nodes is moderate), performance in the presence of locks on NUMA–based machines degrades severely with increasing number of nodes.

We provide distributed data structures to minimize the access conflicts and hot–spots associated with processing the hash buckets. The approach taken here is to build the buckets locally in each node. This has the effect of localizing the accesses to the buckets and thus, eliminating the centralized structure (the global hash table) and the locks associated with it. We also distribute and localize the processors synchronization between the different phases of the join algorithm. This minimizes the number of remote accesses performed by the various processors in order to test for synchronization.

In order to analytically predict the performance of the join algorithm, we used modified multinomial distribution to model the behavior of the hash function in distributing the tuples across the nodes. This allows us to accurately compute the loads at different processors in the system. We derive analytical models for both join algorithms and compare their results to actual experimentations performed on a BBN multiprocessor system. Provided some parameters values are known apriori, the analytical models can be used as a basis for an optimizer to decide which algorithm should be used.

For the proposed join algorithm and in the rest of this thesis, we assume that,

unless otherwise noted, the input relations are initially distributed uniformly across all nodes.

3.3.1 Distributed Hash Join on *NUMA* Architecture

This section presents the Distributed Hash Join (*DHJ*) algorithm for implementing relational join on NUMA architecture. In this algorithm, we provide distributed data structures and special mechanisms to minimize the impact of locking and synchronization and to maximize the locality of reference within the nodes.

The DHJ algorithm proceeds as follows. Each node hashes its local tuples based on the join attributes to determine their destination nodes. Tuples that hash into the local node are further hashed and organized into local buckets. These local buckets are used for probing during the second phase of the algorithm. All other tuples, i.e., those that hashed to remote destination nodes, are grouped in order to be sent to these nodes. When a node has completed distributing R, processing of S proceeds similarly. Tuples are hashed and sent to the corresponding nodes where they are joined with the tuples of R.

In order to improve the performance of the algorithm, tuples that are found to belong to a remote node are not sent directly to that node since that would cause a large number of remote accesses and possibly a large number of conflicts. Instead, they are marked with that node's number (N_i). Later, a list of each node's tuples will be constructed and bulk-transferred to that node. The algorithm is presented in figure 3.4.

The DHJ algorithm was designed to avoid locking of the remote access structures when transferring tuples. Each node, N_i, maintains a list, P_i, of $N-1$ (the number of remote nodes) transfer structures. The structure $P_i[j]$ contains a variable indicating the total number of tuples sent to node N_i by node N_j as well as a pointer

For each node (in parallel) *do*

Step 1: Hash each tuple of R to determine its destination node, N_i.
If N_i = local_node then
Organize tuple locally
else
Mark tuple with the destination node number, N_i.
Bulk–transfer tuples to their destination nodes.
Increment *Counter1*.
If (*Counter1* $\geq$ N) then
Set *Done1* to TRUE in all nodes.

Step 2: Hash each tuple of S and determine N_i.
If $N_i \neq$ local_node then
Mark tuple with the destination node number, N_i.
Bulk–transfer tuples to their destination nodes.
Increment *Counter2*.
If (*Counter2* $\geq$ N) then
Set *Done2* to TRUE in all nodes.

Step 3: Do
Hash R's tuples received from remote nodes and
organize them locally.
Until (local *Done1* is TRUE)
Perform one more check for additional R tuples
received from remote nodes and organize them locally.

Step 4: Hash and join local tuples of S.
Do
Hash S's tuples received from remote nodes and
join them locally.
Until (local *Done2* is TRUE)
Perform one more check for additional S tuples
received from remote nodes and join them locally.
Done

Figure 3.4. *Distributed Hash Join* algorithm for main memory databases on NUMA architecture.

to where these tuples have been stored on node N_i by node N_j. As each node has exclusive access to the structures indexed by its node number, no locks are required to synchronize the accesses.

Note that before starting the join phase, nodes have to wait for all the tuples of R to be organized in the local buckets. This is achieved in *Step* 1 of Figure 3.4. The same is true for *Step* 2 where it is required that all nodes be done before any node can finish execution. Typically, this waiting is done using barrier synchronization where nodes update a shared variable, the barrier, and then continuously check that barrier to reach a predetermined value. For NUMA architecture, however, this type of synchronization can cause severe performance degradation as a large number of nodes need to frequently and concurrently check the current value of the barrier. The solution employed by DHJ is to distribute the synchronization process and localize the checking within each node. This is done by each node having a "Done1" and "Done2" flags that are initially set to $FALSE$. *Done*1 and *Done*2 correspond to the synchronizations for *Step* 1 and *Step* 2 respectively. We only describe the synchronization process for *Step* 1. The process for *Step* 2 is done similarly.

When a node is done executing *Step* 1, it increments a global counter, *Counter1*, that is initially set to zero. Next, the node compares the value of *Counter1* to N, the total number of nodes in the system. If the value of *Counter1* is greater than or equal to N then, the node is the last node to finish *Step* 1. Consequently, this node will set the *Done*1 flags in all the nodes (including its own flag) to TRUE and then proceed to *Step* 2. If the value of the counter is less than N, the node proceeds immediately to *Step* 2. The cost of this type of synchronization is just N sequential remote accesses to the shared global counter during the entire execution of the algorithm.

Next, we motivate and describe the Full Replication Join algorithm.

3.3.2 Full Replication Join

In a typical *Selection-Projection-Join* (*SPJ*) type query, the *Selection-Projection* operations may reduce the size of some of the relations significantly. In such cases, *Full Replication Join* (*FRJ*) fully replicates the smaller relation to all nodes so as to avoid all the remote processing associated with processing the tuples of the larger relation.

In *FRJ* the smaller relation, R, is first replicated to all nodes, hashed into a local bucket and then each node hashes its part of S to join the tuples locally. An advantage of full replication is that, for any given node, the execution of the two steps of the algorithm is totally independent of all other nodes in the system. When full replication of the first relation is done, the processing of the second relation, within each node, can start immediately. The replication saves the cost of remotely accessing the tuples of the larger relation, S, at the expense of replicating and hashing all of the smaller relation, R. The algorithm is shown in Figure 3.5.

For each node (in parallel) *do*
Step 1: Bulk-transfer all tuples of R from all the other nodes.
Step 2: Hash the tuples of R and organize them locally.
Step 3: Hash tuples of S and perform the join.
Done

Figure 3.5. *Full Replication Join* for main memory database systems on NUMA architecture.

Note that in this algorithm all hashing and joining are executed locally and no remote accesses are needed, except for the initial replication.

3.4 Analytical Models

In this section we develop analytical models for the two join algorithms. Performance results derived from these analytical models are then compared with those of the experiments in the next section. First, we present a number of simplifying assumptions made in developing the analytical models for *DHJ* and *FRJ* . We then develop the models for analysis.

- Both relations R and S are horizontally partitioned and their tuples are uniformly distributed across all nodes in the system. In multiprocessor systems based on NUMA architecture, the amount of main memory increases linearly with the number of nodes. Hence, $|R|$ and $|S|$, the total sizes of the relations R and S respectively, are taken to be proportional to the number of nodes in the system. Therefore, $|R| = N \times N_R$ and $|S| = N \times N_S$, where N_R and N_S denote the average number of tuples of R and S per node respectively. For example, in a system with hundred processing nodes and $N_R = 2,500$ tuples, the size of the relation R is $|R| = 250,000$ tuples.

- The average number of comparisons needed to probe a bucket for a given tuple is F.

- All local and remote memory allocations needed for distribution, organization and joining of tuples, are in blocks of size S_b.

Table 3.1 contains a glossary of the terms and parameters used in formulating the cost functions of the analytical models. The table provides a brief description of each parameter as well as the corresponding value used in the computations. The values of the system parameters were measured on a BBN TC2000 multiprocessor by executing the corresponding operations inside tight loops for a large number of iterations. The

results were then averaged over the number of iterations while accounting for *loop* and *system* overheads.

Symbol	Meaning	Value
F	Access Average	1.2
S_b	Block Size (in number of tuples)	50
T_h	Time to hash	.003
T_c	Time to compare two keys	.003
T_l	Time to access local	.005
T_r	Time to access remote	.013
T_{ml}	Time to move a block locally	.23
T_{mr}	Time to move a block remotely	.27
T_{al}	Time to allocate a local block	.05
T_{ar}	Time to allocate a remote block	.09
T_{aj}	Time to actually join two tuples	$2 \times T_{ml} + \frac{1}{S_b} \times T_{al}$
T_i	Time to insert a tuple	$(F \times T_c + T_l)$
T_j	Time to join 2 tuples	$(F \times T_c + T_{aj})$

Table 3.1. Notations and parameters values. All values that pertain to the specific architecture were measured on the BBN TC2000 multiprocessor system.

The network and memory bandwidths, for the NUMA system discussed in this chapter, increase linearly with increasing number of nodes. While concurrent accesses to the same memory location must be synchronized, it has been shown that, in the abscence of hot–spots, networks such as the Multistage Interconnection Network (MIN) of the BBN multiprocessor systen, maintain a close to linear bandwidth with increasing number of nodes [95, 96, 87]. The DHJ algorithm distributes all data uniformely and presents a synchronization scheme that minimizes locking. Hence, network conflicts are minimized through this uniform data distribution and accesses.

Conflicts may occur during the different phases of *DHJ* when a number of nodes try to concurrently access the same target node. To maintain data consistency all

such accesses must be serialized. If a given node N_i attempts to access a target node N_j, N_i may have to wait for the other nodes that are already involved in accessing N_j. Thus, in order to compute the time for accesses with conflicts, we need to determine the number of nodes trying to access N_j. With N nodes active in the system, the time to concurrently access the target node is $\xi_{(R,N)} \times T_a$, where T_a is the time to access the node without any conflicts, $|R|$ is the number of data items and $\xi_{(R,N)}$ is the expected number of nodes involved in the access.

To determine the expected number of concurrent accesses, ξ, we formulate the problem as follows:

Given $|R|$ tuples and N nodes, ($N << |R|$), if N tuples ($N \leq |R| - \frac{|R|}{N}$) are randomly selected from the $|R|$ tuples, find the expected number of nodes with at least one tuple to be written to.

This is the same as the problem of characterizing the number of granules (blocks) accessed by a transaction as determined by Yao [97, 91]. The solution to the above problem is given by Yao's theorem [97] which states that the expected number of blocks hit is given by

$$\phi \;=\; N \times \left[1 \;-\; \Pi_{i=1}^{N} \left(\frac{|R| \times N \times D - i + 1}{|R| \times N - i + 1} \right) \right], \qquad (1)$$

where $|R|$ is the number of tuples in the relation R and

$$D \;=\; 1 - \frac{1}{N}.$$

Therefore, the expected number of nodes involved in the access is $\xi_{(R,N)} \;=\; \frac{N}{\phi}$.

In this model, conflicts are assumed and studied at each node's access path to the network and we assume a single path at each node. If multiple (x) physical or

virtual paths exist for each node, then x accesses may be executed at the same time and consequently, the conflict formulation should be updated accordingly.

3.4.1 Modeling of processors' loads

As noted in Section 3.3.1, before a node can enter *Step* 4 of the algorithm in Figure 3.4, all nodes must have finished executing *Step* 1. Thus the node that handles the largest number of tuples of R, will dominate the time for *Steps* 1 and 3. Same is true for *Steps* 2 and 4 with regard to S. Thus, the analytical model for *DHJ* determines the performance based on the time for the node that handles the maximum number of tuples M_R and M_S for both R and S, respectively.

If the hash functions randomly distribute the tuples across the nodes, some nodes will receive more tuples than others during the distribution phases of R and S. These nodes, having to process more tuples, tend to dominate the performance of the algorithm in the corresponding phases. Thus, a node that receives the maximum number of R tuples will dominate both *Steps* 1 and 3 of Figure 3.4. In order to accurately measure the time for each phase, we need to determine the maximum number of tuples M_R and M_S (for R and S respectively) that a node may receive.

We first define the term *stochastic increase* and then conjecture about the behavior of the value of M_R.

Definition 1 *Given two variables X and Y, X is said to be stochastically greater or equal to Y ($X \geq_S Y$) $\iff \forall\, r$ ($r \in Domain(X,Y)$),*

$$P(X \geq r) \geq P(Y \geq r),$$

where $P(A)$ is the probability of event A.

Conjecturing on the behavior of M_R

Assuming a uniform hash function and the tuples to be uniformly distributed in the tuple space, we can formulate our model as follows. Let x be the number of balls (tuples) and y be the number of urns (nodes) where $x = y \times \mu$ for some constant factor μ (μ represents the average number of tuples per node, N_R). In this model, both balls and urns are indistinguishable. This is to reflect the fact that, from processing point of vew, all processors are equivalent and only the number of tuples is important in contrast to their individual identities. For example, if $x = 12$ and $y = 4$, then $(8,3,1,0) \equiv (8,1,3,0) \equiv (8,0,1,3)$ and so on. Since the urns are also indistinguishable, all the combinations in the above example reduce to just one. For notational convenience, we use the combination with the numbers in descending order, e.g., for the above example we use $(8,3,1,0)$.

Given x balls and x urns and allowing for empty urns, the number of possible combinations is equivalent to the number of ways to partition an integer into different sums of integers [98, 99, 100]. Integer partitioning is a well known problem in *Number Theory*. For example, for $x = 4$ we have 4, $3 + 1$ and $2 + 2$, $2 + 1 + 1$, $1 + 1 + 1 + 1$. To view this example with 4 urns, we have $(4,0,0,0)$, $(3,1,0,0)$, $(2,2,0,0)$, $(2,1,1,0)$, $(1,1,1,1)$. Now let us consider P_x^y, the number of partitions of x into y parts. For $x = 4, y = 2$ in the above example, the combinations are $\{(4,0,0,0)\}$ and $\{(3,1,0,0),\ (2,2,0,0)\}$ and $P_4^2 = 1 + 2 = 3$.

It is easy to see that the total number of combinations of x balls into y urns, W_x^y, is equivalent to the number of partitions of x into a maximum of y integers. Hence, W_x^y can be computed as the sum of the number of ways to parition x into y or less numbers:

$$W_x^y = \sum_{i=1}^{y} P_x^i.$$

It has been shown in *Number Theory* that the numbers P_x^y satisfy the following

recurrence relation [98, 101, 102]:

$$P_x^1 + P_x^2 + \ldots + P_x^k = P_{x+k}^k,$$

with $P_x^1 = P_x^x = 1$. This relationship enables a recursive computation of the numbers P_x^y row by row as shown in Table 3.2.

y / x	1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0	0	0
3	1	1	1	0	0	0	0	0	0	0	0	0
4	1	2	1	1	0	0	0	0	0	0	0	0
5	1	2	2	1	1	0	0	0	0	0	0	0
6	1	3	3	2	1	1	0	0	0	0	0	0
7	1	3	4	3	2	1	1	0	0	0	0	0
8	1	4	5	5	3	2	1	1	0	0	0	0
9	1	4	7	6	5	3	2	1	1	0	0	0
10	1	5	8	9	7	5	3	2	1	1	0	0
11	1	5	10	11	10	7	5	3	2	1	1	0
12	1	6	12	15	13	11	7	5	3	2	1	1

Table 3.2. Number of partitions of x into y parts.

We now proceed with the conjecture that the value of M_R increases stochastically with increasing values of N. Note that the probability $P(X \geq r)$ mentioned in *Definition 1* can be computed as the number of combinations with the rightmost number, the maximum number in the combination, greater or equal to r divided by the total number of combinations W_x^y.

Given y urns, we let M_r^y be the number of combinations having r as maximum. First, we provide an example to illustrate and simplify the presentation of the var-

ious formulations. Let $\mu = 3$, Figure 3.3 presents the possible combinations for $(x, y) = (6, 2)$, $(9, 3)$ and $(12, 4)$ respectively. We note that when $r \geq \frac{x}{2}$,

$$M_r^y = W_s^{y-1}$$

where $s = x - r$. The reason is that, the number of combinations with y urns and r as maximum is equivalent to the number of combinations of $x - r$ balls with $y - 1$ urns. For example, in Table 3.3 with $x = 12$, $M_7^4 = 5 = W_5^3$ and $M_8^4 = 4 = W_4^3$ but $M_5^4 = 6 \neq W_7^3 = 8$.

(x,y)	Combinations						
(6,2)	(6,0)	(5,1)	(4,2)	(3,3)			
(9,3)	(9,0,0) (8,0,1) (7,0,2) (6,3,0) (5,4,0) (4,4,1) (3,3,3)	(7,1,1) (6,2,1) (5,3,1) (4,3,2)	(5,2,2)				
(12,4)	(12,0,0,0) (11,1,0,0) (10,2,0,0) (9,3,0,0) (8,4,0,0) (7,5,0,0) (6,6,0,0) (5,5,2,0) (4,4,4,0) (3,3,3,3)	(10,1,1,0) (9,2,1,0) (8,3,1,0) (7,4,1,0) (6,5,1,0) (5,5,1,1) (4,4,3,1)	(9,1,1,1) (8,2,2,0) (7,3,2,0) (6,4,2,0) (5,4,3,0) (4,4,2,2)	(8,2,1,1) (7,3,1,1) (6,4,1,1) (5,4,2,1) (4,3,3,2)	(7,2,2,1) (6,3,3,0) (5,3,3,1)	(6,3,2,1) (5,3,2,2)	(6,2,2,2)

Table 3.3. Example of various combinations for $\mu = 3$.

When $r < \frac{x}{2}$, s becomes larger than r. But, since r is the maximum number,

all combinations starting with r and containing a number larger than r have to be dropped. These combinations are exactly the ones containing $s, s-1, \ldots, r+1$ as the maximums. Hence,

$$M_r^y = W_s^{y-1} - \sum_{j=0}^{s-r-1} P_{j+y-2}^{y-2}.$$

For example,

$$M_5^4 = W_7^3 - \sum_{j=0}^{1} P_{j+2}^2 = 8 - (1+1) = 6$$

and

$$M_4^4 = W_8^3 - \sum_{j=0}^{3} P_{j+2}^2 = 10 - (1+1+2+2) = 4.$$

Hence, the number of combinations with $r \geq k$ for some variable k can be expressed as

$$T_k^y = \sum_{r=k}^{x} \left[W_{x-r}^{y-1} - \sum_{j=0}^{x-2r-1} P_{j+y-2}^{y-2} \right].$$

The probability of obtaining a combination with $r \geq k$ is

$$P(k,y) = \frac{T_k^y}{W_x^y}.$$

That is to say, the probability is the number of combinations with k as maximum over the total number of combinations with x balls and y urns.

From the example in Figure 3.3, we have $P(5,3) = \frac{9}{12} \leq P(5,4) = \frac{29}{34}$ and $P(6,3) = \frac{6}{12} \leq P(6,4) = \frac{23}{34}$. We can see from the above example (and we have verified using numerous other examples) that

$$\forall\, k \leq x,\ P(k,y) \leq P(k,y+1).$$

Thus, we conjecture that the value of the maximum (M_R) increases stochastically since the corresponding probability is increasing.

It is important to note that the above states only that the value of M_R increases

stochastically with increasing number of nodes. That is to say,

$$\forall\ K,\ P(M_R \geq K\ s.t.\ N = X + 1)\ \geq\ P(M_R \geq K\ s.t.\ N = X).$$

This does not show however, that M_R increases monotonically as the value of M_R can actually decrease as shown in Figure 3.7.

Computing the value of M_R

In this section, we compute the actual value for the maximum based on a modified *Multinomial* probabilistic distribution. Assuming the hashing function to randomly distribute the tuples among all nodes, the number of tuples per node is a random variable with *Multinomial* distribution. When n_N is large compared to N, we have

$$\mu\ =\ \frac{n_N}{N}$$

and

$$VAR(X)\ =\ \sigma\ =\ n_N \times \left(\frac{1}{N}\right) \times \left(1 - \frac{1}{N}\right)$$

and Normal distribution may be used for approximation.

Let $\vec{Z}\ =\ (Z_1, Z_2, \ldots, Z_N)$ be a vector of N Independent and Identically Distributed (IID) random variables. Let $\vec{Y}\ =\ \sigma \times (\vec{Z} - \bar{Z}) + \mu$, where $\bar{Z}$ is the mean of $\vec{Z}$. Then the maximum number of tuples a node may receive for the parameter values n_N and N is $M_{N,n_N}\ =\ max(\vec{Y})$.

In order to determine M_{N,n_N}, we compute

$$Q_N\ =\ \frac{M_{N,n_N} - \mu}{\sqrt{n_N}}\ =\ max(\vec{Z}_i - \bar{Z}),\quad for\ i = 1, \ldots, N$$

then solve for M_{N,n_N}:

$$M_{N,n_N} = Q_N \times \sqrt{n_N} + \mu.$$

Given N and N_R, the number of nodes and average number of tuples per node respectively, the computation of M_R proceeds by computing N *Normal* variables with a mean of 0 and a standard deviation of 1 [103, 104, 105, 106]. Next, we compute the average of these variables ($\bar{Z}$) and then the value of $max(\vec{Z}_i - \bar{Z})$ for $i = 1, \ldots, N$. Now, the value of M_R can be computed as $\left[max(\vec{Z}_i - \bar{Z}) \times \sqrt{N \times N_R} + N_R\right]$. Figure 3.6 provides pseudo–code for the actual algorithm used to compute the value of M_R.

```
/* Compute N Normal Variables */
For i = 1 to ceiling(N/2) do
    Repeat
      Generate u1 and u2 (Uniformly over [0,1])
      Let v1 = 2 × u1 − 1 and v2 = 2 × u2 − 1
      Let s = v1^2 + v2^2
    Until (s < 1)
    Let t = sqrt(−2 × ln(s)/s) and
    Let Z2i = t × v1 and Z2i+1 = t × v2.
Done

/* Compute value of MR */
Let Z̄ = average of the Z variables and
Let Y = max(Zi − Z̄)  i = 1, ..., N
MR = Y × sqrt(N × NR) + NR
```

Figure 3.6. Pseudo–code of the computation of M_R using the Normal variables to approximate the Multinomial distribution.

Next, we derive the cost functions for the DHJ algorithm using the derived values

of M_R and M_S.

3.4.2 Cost Formulation for *DHJ*

Given the maximums derived above, the cost for the Distributed Hash Join algorithm is derived as follows:

- *Step* 1 : Hash each tuple of R and determine its destination node, N_i. With the assumption of uniform distribution, $\frac{1}{N}$ of the tuples will hash to the local node while the other $\frac{N-1}{N}$ of the tuples will hash to remote nodes. Consequently, the cost of organizing the local tuples into buckets is $\frac{1}{N} \times (T_h + T_i)$ and that of grouping the other tuples according to their remote destination nodes is $\frac{N-1}{N} \times (\frac{1}{S_b} \times T_{al} + T_l)$. Thus, the total cost of this is:

$$M_R \times \left[T_h + \frac{1}{N} \times (T_h + T_i) + \frac{(N-1)}{N} \times \left(\frac{1}{S_b} \times T_{al} + T_l \right) \right] \times \xi_{(R,N)} \times \xi_{(S,N)}$$

  Next, for each remote destination node, we allocate space on the corresponding nodes and move the lists. The cost is:

$$M_R \times \frac{(N-1)}{N} \times \left[\frac{1}{S_b} \times T_{ar} + T_{mr} \right] \times \xi_{(R,N)} \times \xi_{(S,N)}$$

- *Step* 2 : The cost for this step is similar to that of *Step* 1. Each tuple of S is hashed to determine its destination node, N_i. The tuple is added to the corresponding list if N_i is not the current node. The cost for this is:

$$M_S \times \left[T_h + \frac{(N-1)}{N} \times \left(\frac{1}{S_b} \times T_{al} + T_l \right) \right] \times \xi_{(S,N)} \times \xi_{(R,N)}$$

Next, allocate space on the corresponding nodes and move the lists. The cost is:

$$M_S \times \frac{(N-1)}{N} \times \left[\frac{1}{S_b} \times T_{ar} + T_{mr}\right] \times \xi_{(S,N)} \times \xi_{(R,N)}$$

- *Step* 3 : Hash all the tuples of R received from the remote nodes in *Step 1* and organize them locally. As in *Step* 1, with the assumption of uniform distribution, the fraction of tuples sent to the local node from remote nodes is $\frac{N-1}{N}$. Thus, the cost of this step is:

$$M_R \times \frac{(N-1)}{N} \times [T_h + T_i] \times \xi_{(S,N)} \times \xi_{(R,N)}$$

- *Step* 4 : Finally, the cost of joining the local and remotely received tuples of S is:

$$M_S \times [T_h + T_j] \times \xi_{(S,N)}$$

3.4.3 Cost Formulation for *FRJ*

The cost for *FRJ* is analyzed as follows:

- *Step* 1 : Copy all tuples of R from all other nodes (using bulk-transfer). The cost is:

$$(N-1) \times T_b(N_R, N) \times \xi_{(R,N)}$$

- *Step* 2 : Hash the tuples of R and organize them locally. Cost is

$$N_R \times N \times \left[T_h + T_i \times \xi_{(R,N)}\right]$$

- *Step* 3 : Hash tuples of S and perform the join. The cost is:

$$N_S \times \left[T_h + T_j \times \xi_{(R,N)}\right]$$

A quantitative performance evaluation of these models is presented in the next section along with actual experimentation on a commercial multiprocessor system.

3.5 Performance Evaluation

In this section we study the performance of both join algorithms using the cost formulas developed in the previous section. We present actual experimental results obtained from implementing these algorithms on a BBN TC2000 multiprocessor system and show that these results do validate our analytical models with varying degrees of accuracy. Measurements for the experimental results were taken for varying relations sizes and the total cost of the algorithm was computed as the maximum time over all nodes in the system. Each plotted value represents the average of a number of runs with the same parameter values. A brief explanation concerning the actual relations used in the experiments follows.

3.5.1 Benchmarking relations

The relations used in the experimentations are based on the standard Wisconsin Benchmark [107]. Each relation consists of two integer identifiers and three strings each of length 40 bytes. The initial partitioning of the relations is based on the first integer attribute. Each string is 40 letters long with three letters (the first, middle and last) being varied, and two separating substrings that contain only the letter x. The three significant letters are chosen in the range (A,B,...,V) to allow up to $10,648$ different string combinations. If a larger relation size is desired, as is the case in most of this thesis, more significant letters are added within the separating substrings to create additional combinations.

In generating the tuples for the experimental runs, only the first integer attribute and one of the strings in each tuple were used. Other attributes in the benchmark

relations were designed to test operators other than join, such as the projection and aggregate operations and thus, the values they contained were not used in our experiments.

3.5.2 Performance comparison

When the sizes, in number of tuples, of the relations increase then more tuples need to be transferred and processed by the remote nodes. In Figure 3.7, N_R was fixed at 500 tuples and N_S was set to 1,000 and 2,500 tuples, while N varied from 1 to 25 nodes.

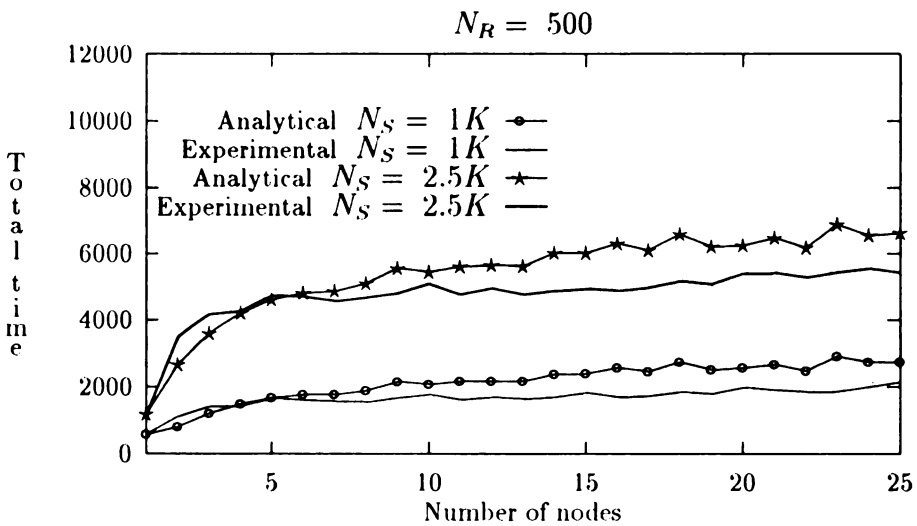

Figure 3.7. Comparison of the analytical and experimental results for *DHJ*. The experimental runs were conducted on the BBN TC2000 with up to 25 nodes - the maximum number of public nodes available on the system we had access to.

The figure shows an initial increase in cost when the number of nodes increases from 1 to 5. This is due to the increase, in this range, of the number of conflicts

generated by concurrent accesses (as presented in the discussion about conflicts in section 3.4). When the number of nodes and consequently the total number of tuples increase, the maximum number of tuples a node may receive also increases. Note however, that this maximum number of tuples increases stochastically and not monotonically. This explains the fact that, for certain number of nodes in the figure, this value does actually decrease as evident in our experimental results. The value of the maximum was approximated using the formulation in section 3.4.1. This increase will contribute to more delay in the processing of each phase and consequently, cause more conflicts. This is shown in Figure 3.7 where the cost of the algorithm does increase slightly with increasing number of nodes.

Nodes that receive more tuples and thus lag behind in *Step* 1, are subjected to remote accesses by those processing tuples in *Step* 2. On average, this will only occur for the period of time needed to process the extra tuples. The analytical model handles this by including the conflict rate $\xi_{(S,N)}$ in the cost formula for *Step* 1. While this helps in accounting for the conflicts generated by the faster nodes (those in *Step* 2), it overestimates the effects of these conflicts. The same occurs when $\xi_{(R,N)}$ is used in *Steps* 2 and 3. However, because the model computes the cost of the algorithm based on the node with the maximum number of tuples, it closely approximates the actual experimental results. It can be noted from Figure 3.7 that the analytical model does slightly overestimate the cost of the algorithm but that for the most part, it closely models the experimental results.

Figure 3.8 illustrates both *DHJ* and *FRJ* with a small ratio of R to S (R is 100 and S is 2,500 tuples) and shows that for up to 23 nodes, in this particular case, *FRJ* outperforms *DHJ*. The reason is that, as mentioned earlier, while *DHJ* has to transfer lists of tuples from remote nodes, all tuples accesses in *FRJ*, following the replication step, are local. This locality of references is the main idea behind the *FRJ* algorithm. When the size of S increases, the cost of remotely accessing the tuples, in *DHJ*,

increases much more rapidly than the cost of accessing them locally in *FRJ*. Hence, larger sizes of S favor *FRJ* over *DHJ*. On the other hand, the costs of replicating and locally hashing all of R increase very rapidly with the size of R. When the size of R increases, the benefits of the locality of references in *FRJ* decrease against the rapidly growing costs of replication and local hashing. Thus, *FRJ* outperforms *DHJ* when the size of R is very small compared to that of S. In Figure 3.8, the cost of *FRJ* increases beyond that of *DHJ* when the size of R is approximately 2,300 tuples (at 23 nodes).

Given the accuracy of the analytical models in predicting the performance of both join algorithms, the models can be used in order to decide which algorithm to apply. The information needed to make such a decision is the total number of nodes in the system and the sizes of both relations. This information can be readily available when the execution of the join operation is attempted.

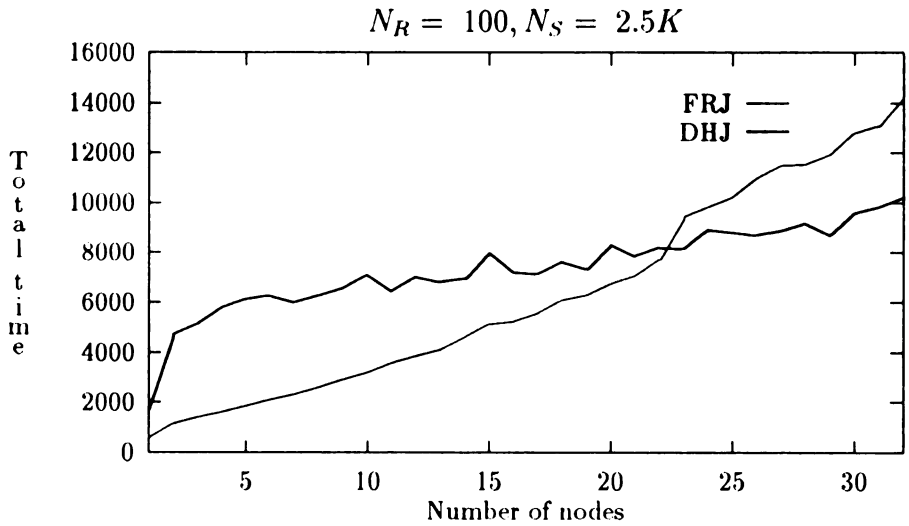

Figure 3.8. Comparison between the Distributed Hash Join and the Full Replication Join algorithms.

Figure 3.9 shows the performance of *FRJ* when N_R= 50 and N_S= 1,000. Note that as the number of nodes increases the number of tuples to be replicated and locally organized increases linearly, and so the costs of replication and local organization increase linearly (but with different rates, see Figure 3.10). When the size of R increases, these costs dominate the performance of *FRJ* . The figure compares the analytical and simulation results produced with the same parameters' values. It can be noted that for the most part the analytical model does reasonably approximate the simulation results.

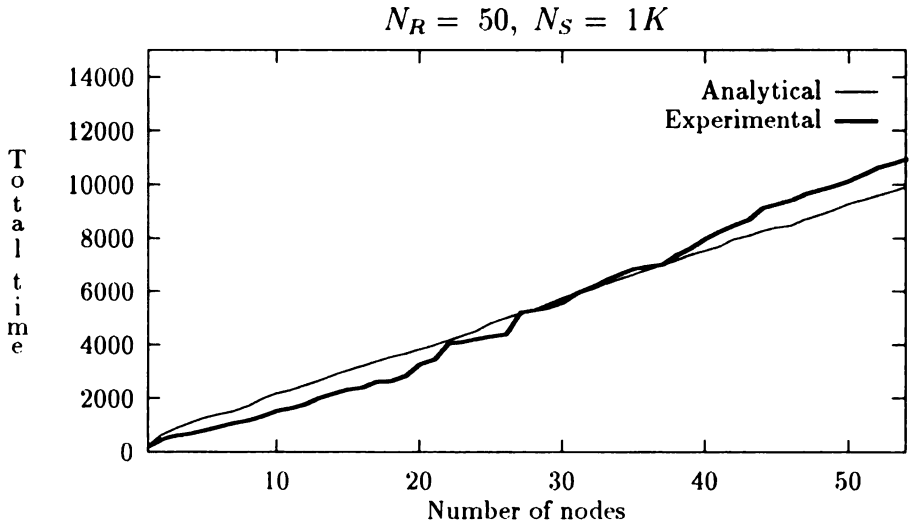

Figure 3.9. Comparing analytical and experimental results for Full Replication Join algorithm.

Figure 3.10 shows the individual costs of the replication, local hash and join steps in *FRJ* when R = 25 and S = 1,000 tuples. As noted earlier, *FRJ* eliminates all remote processing of S, thus improving the performance when S is large, at the expense of fully replicating and locally organizing R . When the number of nodes

increases, the cost of joining the local tuples of S increases slightly because the size of the local hash table for R gets larger. The local organization of R increases linearly with the number of nodes but since the size of R is very small for this algorithm, the effect of this step on the overall performance is minimal. The major factor affecting performance is the cost of replication which increases more rapidly than the cost of joining and local hashing. This is shown in Figure 3.10.

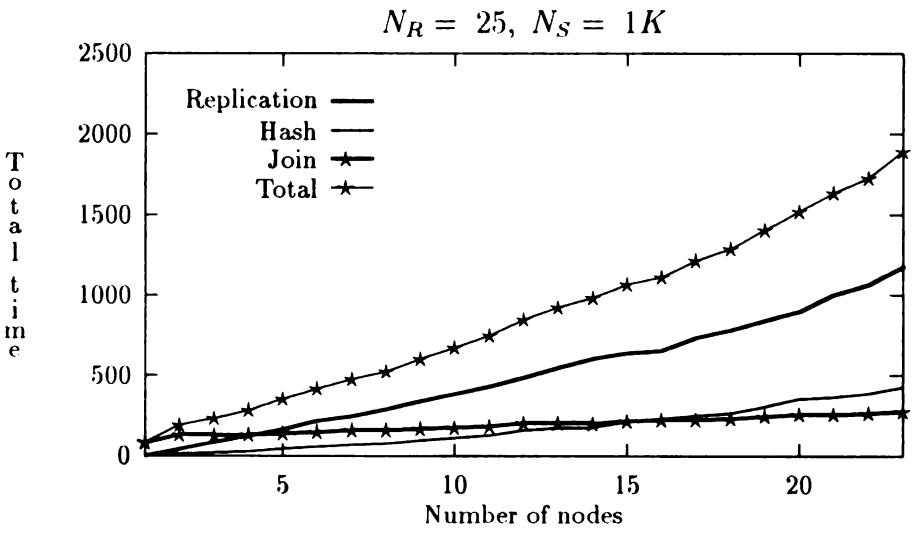

Figure 3.10. Execution times for individual steps of the Full Replication join algorithm.

3.6 Data Skew

When the distribution of tuples is not uniform, some processors may get a much larger share of work than others. This data skewness can severely affect the performance of join algorithms [17, 72]. We next present the Zipfian distribution used to model data

skew and then examine the effects of skew on the performance of DHJ.

3.6.1 Zipfian Distribution

In order to model data skew in the input relations, the uniform distribution of tuples in the tuple space is often replaced with the parameterized Zipfian distribution [108, 109]. Zipfian distribution, applied to the input relations' tuple space, states that a large number of tuples, tunable by a given parameter, are concentrated within a small region of space. In the context of parallel database systems, it means that given an unbiased partitioning strategy, such as hashing, a small number of nodes will get most of the tuples. As noted in Chapter 1, the findings of Lakshmi and Yu state that many real-life data sets exhibit this kind of data skew [17]. We next introduce the formulation for Zipfian Distribution in the context of parallel database systems.

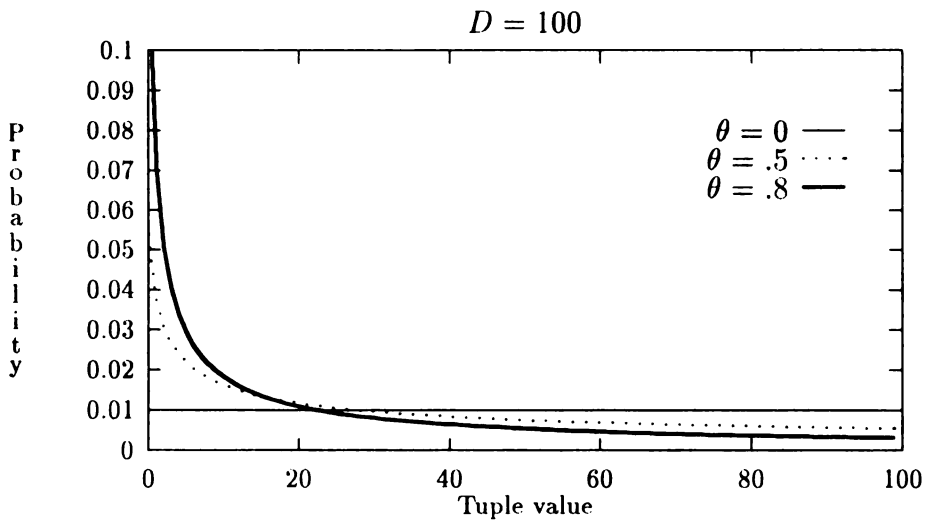

Figure 3.11. Zipf-like distributions with various θ values.

Assume the domain of the join attribute to have D distinct values. The probability that the join attribute of a given tuple takes on the ith value in the domain is

$$p_i = \frac{c}{i^{(1-Z)}}$$

, where $1 \leq i \leq D$ and

$$c = \frac{1}{\sum_{i=1}^{D} \frac{1}{i^{(1-Z)}}}$$

is a normalization constant. The value of Z ranges between 0, the pure Zipfian which is highly skewed and 1 which corresponds to the Uniform distribution. In order to make the value of the skew parameter more intuitive, i.e., a higher value means more skew, we opted to use $\theta = 1 - Z$ in our discussions regarding data skew. Figure 3.11 presents a typical Zipfian distribution where the tuple domain is of size 100 and the Zipfian parameter is set to 0, 0.5 and 0.8 respectively.

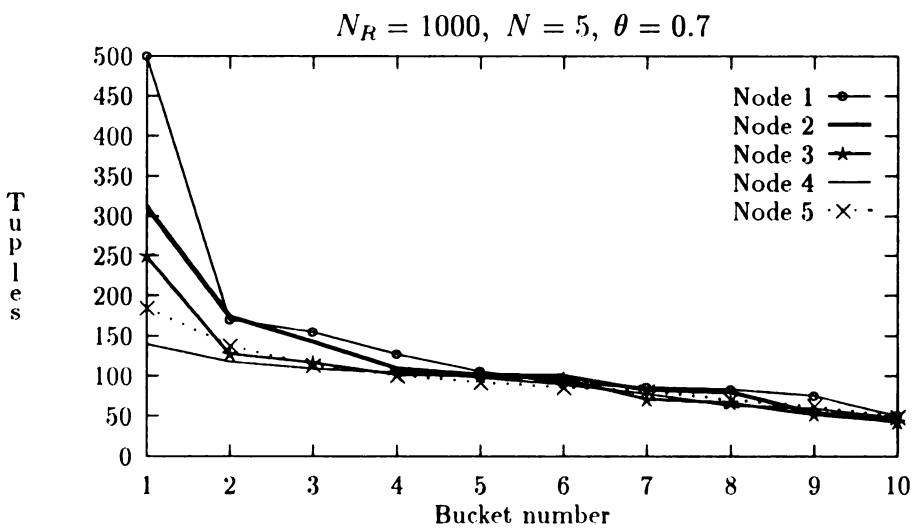

Figure 3.12. Typical distribution of tuples to buckets under skew.

Figure 3.12 shows a typical distribution of tuples, in the presence of skew, to buckets within the various nodes. Note the effect of skew on the sizes of the buckets where few buckets are quite large due to highly skewed input values while others are quite small. The results presented in this figure were computed experimentally where a total of 5,000 tuples were distributed among 5 nodes each with 10 buckets. The skew rate was fixed at $\theta = 0.7$. The buckets in each node were sorted by size to simplify the presentation of the results in the figure.

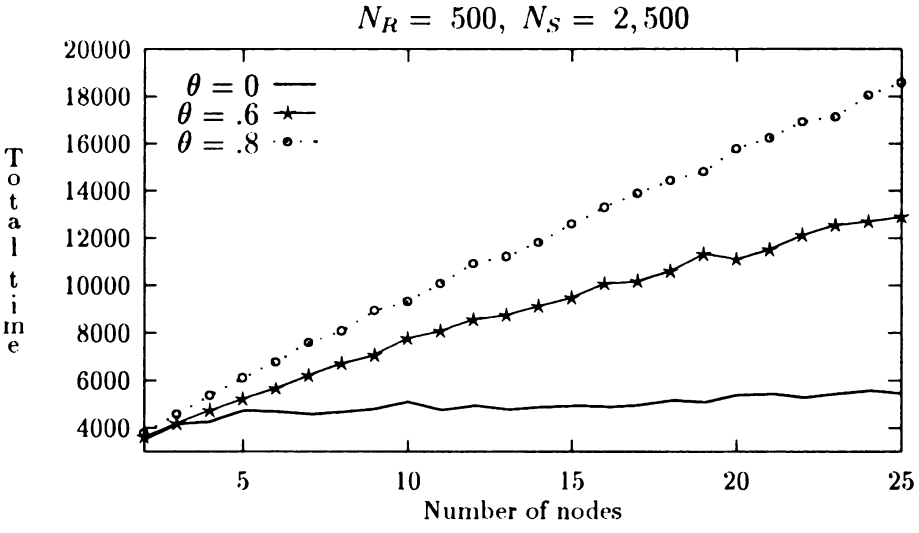

Figure 3.13. Performance of Distributed Hash Join with skewed input relations.

3.6.2 Effects of Skew on the Performance of DHJ

Figure 3.13 compares the performance of DHJ with various degrees of data skew. The figure shows that with increasing values of θ, i.e. higher degrees of data skew, the total execution time for DHJ increases. This is because the few nodes, overloaded

with a large number of tuples, will take longer to process these tuples. Other, more lightly loaded nodes, may be idle during this time and thus resource utilization can be very poor in these cases. The next chapter investigates load balancing for the join algorithm in order to effectively deal with skewed data.

CHAPTER 4

DISTRIBUTED LOAD BALANCING FOR PARALLEL MAIN MEMORY HASH JOIN

4.1 Introduction

In Selection–Projection–Join (*SPJ*) type queries, the selection and projection steps may largely alter the distribution of tuples between nodes. Consequently, the join operation will have to deal with an unbalanced load. Hashing of the tuples of R and S, during the initial distribution phase of the join algorithm, may also result in data skew, as explained in Chapter 1 and shown in [17, 72].

In Chapter 2, we introduced a number of proposed schemes to deal with the problem of load balancing. Based on our examination of these schemes according to both, the balancing decisions and the actual data transfers, we can classify them into different categories. The first category uses a centralized scheduler to make the balancing decisions. Algorithms that fall into this category were mainly designed for UMA–based systems and do not scale well with increasing number of nodes. Algorithms in the second category add a separate phase for collecting statistical distribution information about the tuples. This information is later used to make the decisions concerning load balancing. The major problem with the algorithms in this category comes from the added cost incurred by the information gathering phase. Thus, the performance of the load balancing scheme suffers in the cases of moderate to low skew

rates as the statistics collection phase has to be executed anyhow.

In this chapter, we propose a distributed dynamic load balancing scheme [110]. This scheme does not rely on prescanning the input relations and thus, it avoids the extra overhead associated with collecting statistics about the distribution of tuples.

The balancing scheme is completely dynamic where each helping node, i.e. a processor that is done processing its local data, decides which other nodes to help. In order to make the helping decision, a helping node checks the status of the other nodes and selects an overloaded node to help. The helping node then gets a block of tuples from the overloaded node and processes these tuples locally. In a *NUMA* multiprocessor, such interactions between the nodes may cause conflicts.

We show that random probing for overloaded nodes eliminates the hot-spot effects associated with sequential probing. We develop an analytical model that accurately predicts the performance of the balancing scheme and compare its results to those obtained from actual experiments on a BBN multiprocessor system. We show the load balancing scheme to be effective in dealing with various degrees of data skew. The next section presents the load balancing scheme in details and addresses the problem of conflicts.

4.2 Proposed Load Balancing Scheme

When load imbalance arises, each helping node will transfer an appropriately sized block of tuples from a selected overloaded node and process the block locally. The selection of the overloaded node is made by each helping node independently in order to prevent any bottlenecks that may result from a centralized scheduling scheme.

The general scheme for distributed load balancing is shown in Figure 4.1. The termination check in *Step* 2 of the figure is similar to that of DHJ in Section 3.3.1. Each node in the system contains a local *Done* flag that is initially set to *FALSE*. A

node that is done with *Step* 1 increments a global counter, initially set to 0. The node then determines whether it is the last node to finish *Step* 1. This is done by simply comparing the current value of the global counter to N, the total number of nodes in the system. If the values are different, then the node can go into the helping mode. Otherwise, if the values are equal, this node will set the *Done* flags to *TRUE* in all the nodes, thus signaling the end of execution. While locking is necessary in updating the global counter, there are only N such updates throughout the execution of the algorithm. All helping nodes will check for termination of execution using their local flags. Thus, termination check will not cause any additional network contention.

Each node P_k in the system, maintains an index (I_k) to its data and a lock (L_k) to synchronize simultaneous accesses to I_k. A node, wishing to copy a block from node P_k, has to lock L_k, update I_k with the size of the block, unlock L_k and then transfer the block. The size of the block to be transferred from a given node is taken as the minimum of S_b and the number of tuples left at that node. This means that all nodes must use the locks in order to process their own tuples. On NUMA architecture, locks can be very costly if not handled properly. The next two sections discuss strategies for selecting overloaded nodes while considering the effects of locking.

4.2.1 Scheduling by Sequential Probing

Probing, to find an overloaded node, can be done either sequentially or randomly. In sequential probing, helping nodes sequentially check the status of other nodes and select the first overloaded node to help. Since the helping nodes are searching for overloaded nodes sequentially, all the helping nodes between two consecutive overloaded nodes will concurrently try to help one of these overloaded nodes. These concurrent accesses have to be serialized and thus, will form long chains at the overloaded nodes. This phenomenon may cause an uneven distribution of helping to helped nodes and can result in hot spots.

For each node (in parallel) *do*
Step 1: Process local data.
Increment Global Counter.
If (Global Counter = N) then
Set the *Done* flags to
TRUE in all nodes.
Step 2: Repeat
A. Select an overloaded node.
B. Transfer a block of size
Minimum(Un–processed data items, S_b)
C. Process the block locally.
Until (local *Done* flag = TRUE).
Done

Figure 4.1. Distributed load balancing scheme

To illustrate this phenomenon, let us assume that the system has 16 nodes and at time t, all nodes have become helping nodes except for nodes number 13 and 16 that are still overloaded. With sequential probing, node number 13 may be probed and helped by all the 12 nodes before it, while node number 16, which may have many more tuples than number 13, will only get help from nodes 14 and 15.

Figure 4.2 displays the percentages of large intervals when the number of nodes in the system is $N = 100$. We define an interval as the number of helping nodes between two successive overloaded nodes and we let n_1 be the number of overloaded nodes in the system. The figure shows that the percentage of large intervals increases sharply with the skew rate. For example, less than 6 % of the intervals are more than 40 when $n_1 = 6$. This percentage increases sharply to about 20 % when n_1 is decreased to 4. Thus, the probability of forming long chains increases (which results in hot spots) with decreasing number of overloaded nodes and consequently with higher skew rates. According to Lakshmi and Yu [17], only very few nodes will have

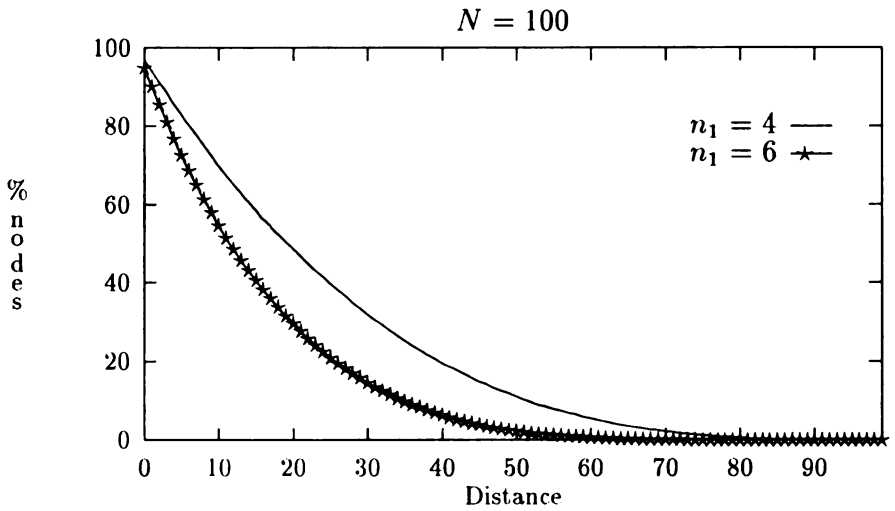

Figure 4.2. Percentage of overloaded nodes vs interval size.

most of the tuples. Hence, sequential probing is likely to degrade the performance of the balancing scheme in the presence of data skew.

The data in Figure 4.2 was obtained experimentally where 4 and 6 nodes were selected randomly to be overloaded nodes and the intervals between them were measured. A count of each interval size was kept and the experiment was run for 10,000 iterations. The final results were then averaged and their cumulative probabilities were computed and plotted.

4.2.2 Scheduling by Random Probing

As the number of nodes increases, the chain effect in sequential probing causes more performance degradation. In order to avoid this problem, helping nodes probe randomly for overloaded nodes. A node is selected at random and its status is checked. If it is overloaded, the helping node will get a block of data from this node, process it locally and then randomly select another overloaded node.

Random selection was shown in [111] to be as good as other more complex methods for implementing load sharing in a homogeneous distributed environment. Kumar *et al.* [112, 113] studied the scalability of various load balancing schemes including a scheme called random polling. Under this scheme, an idle node randomly polls other nodes for work and sends requests to these nodes. Upon receiving a work request, a node will reject the request if it is an idle node, or grant it and thus, transfer an appropriately sized block to the requesting node. A background process monitors the state of each node and will terminate the whole program when all nodes become idle. This random scheme was compared with a large number of other balancing schemes that relied on statistics collection and/or redistribution of loads and was shown to be the most scalable [112].

In our scheme, idle nodes probe randomly and perform the transfers directly and thus, do not require any extra processing or lookups on the part of the overloaded nodes. When a node is done joining its local buckets, it becomes idle and increments a globally shared variable. The program may terminate when the value of this variable reaches N. Accessing the shared variable is efficiently handled by the scheme presented earlier in this section.

The effects of random probing on block transfers were tested experimentally on a BBN multiprocessor system. A pair of processors, P_i and P_j were designated as source and destination nodes respectively. During execution, processor P_j transfers a block of tuples from P_i and processes the block locally. This transfer process from P_i to P_j is repeated 1000*times while all other nodes in the system are engaged in random probing. A probing node would repeatedly execute a number of local operations and then randomly probe a remote node. The number of local operations, Y determines the frequency of random probing. The effects of varying the value of Y are shown in

*It is important to have a large enough number of transfer to allow all processors enough time to probe and ensure the measurements' accuracy.

Figure 4.3 and discussed next.

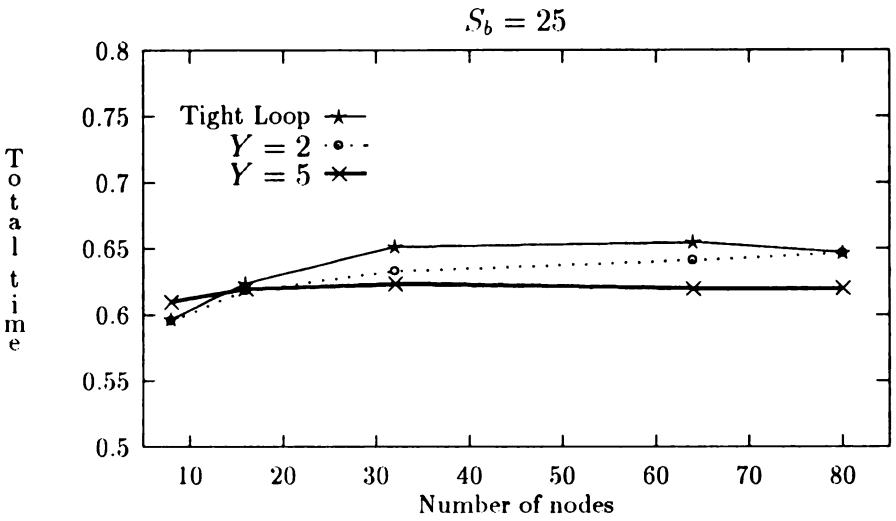

Figure 4.3. Effect of random probing on network traffic.

The results of the experiments are shown in Figure 4.3. The figure illustrates the effects of random probing on the general network traffic and, in particular, on the block transfer operations. Note the effects of the tight loop, i.e. no local operations, where an increase in transfer time can be seen. This is because the processors are constantly generating remote accesses as fast as they could. Despite being very small, single data–item accesses, the effects of this very high frequency of requests on the transfer cost are evident with increasing number of nodes in the system. In our load balancing algorithm, a helping node repeatedly executes a number of local operations (such as checking the local *Done* flag, a loop index or various local variables) and then probes for an overloaded node. The number of local operations executed, Y is equal to five. The figure shows that for this case, the effect of random probing on the

cost of transfers is minimal even for large values of N (up to 80 nodes).

4.3 Analytical Model for Load Balancing based on Random Probing

In this section we develop the analytical model for the load balancing scheme. The cost of processing all the tuples, without load balancing, can be computed as the total time taken by the processor that finishes last, i.e. the processor with the maximum number of data items. However, with the load balancing scheme, it is not the node with the maximum number of tuples but rather the helping node, that gets the last block of tuples, that will finish last. In the rest of this chapter, this node will be referred to as the *last helping node*. The cost formulas for the total computation time, derived in this section, correspond to this node.

To obtain the total time for load balancing, we will add up the processing costs of each block processed by the last helping node. Thus, if $T(i)$ is the time to process the ith block by the last helping node, then the total time for load balancing is

$$T = \sum_{i=1}^{L} (T(i))$$

where L is the total number of blocks processed by the last helping node.

Let $n_0(i)$ and $n_1(i)$ be respectively, the average number of helping nodes and overloaded nodes during the time interval needed to process the *ith* block. The cost $T(i)$ can be computed as the sum of the following components:

- For a selected overloaded node, N_k, lock L_k, determine the block size, update I_k and then unlock L_k. The contention for the locks are modeled using the same formulation as in Section 3.4. Hence, the cost is $T_{lock} \times \xi_{(M,N)} + T_{unlock}$. The costs of locking and unlocking are system dependent and the costs of determin-

ing the block size and updating I_k may be ignored (as these are local operations and they are only executed once for each block of tuples).

- Transfer a block to be processed locally, only if this is a helping node. With $n_0(i)$ helping nodes scanning among all N nodes in the system, the probability of transferring a block is given by $\frac{n_0(i)}{N}$. Thus, the total cost of this step is:

$$\frac{n_0(i)}{N} \times T_{bt}(S_b, N),$$

where $T_{bt}(S_b, N)$ is the cost of transferring a block of size S_b when N nodes are active. This parameter is system dependent.

- Process the data block locally. The cost for this is $T_b = S_b \times T_p$, where T_p is the time to process a single data item. The parameter T_p is application dependent and for the join algorithm, it corresponds to the cost of organizing the tuples of R into the local buckets, or the cost of joining the tuples of S.

While random probing eliminates the chain effect, conflicts may still occur when more than one node randomly select and access the same overloaded node. Using the same conflict formulation described in section 3.4, the number of nodes involved in these accesses is $\xi_{(M,N)}$, where M is the number of data blocks. It is important to note that remote accesses through the network are being done only by the helping nodes. Hence, a node being accessed, i.e. an overload node can only be processing locally and will not conflict with incoming requests.

During any time step, the number of concurrent trasnfers through the network is necessarily smaller than the minimum of $n_0(i)$ and $n_1(i)$. This is because only helping nodes perform transfers and these transfers are synchronized at the overloaded nodes. When the helping and overloaded nodes are almost equal in number, it is very likely that only a small fraction of the helping nodes will be attempting transfers through

the network. The main reason being that the helping nodes are probing randomly for overloaded nodes and the probability of each helping node locating a distinct overloaded node is very small.

We showed earlier that random probing by the helping nodes does not cause network conflicts. It is also clear, from the above discussion, that the number of concurrent transfers is almost always small compared to the network bandwidth (which increases with the number of nodes). Hence, the major conflicts that occur are those caused by an overloaded node being selected by more than one helping node. These conflicts are modeled using the ξ formulation as mentioned above.

Therefore, the cost of reserving and processing a block of data is $T(i) =$

$$\left[T_b + (\frac{n_0(i)}{N} \times T_{bt}(S_b, N) + T_{lock}) \times \xi_{(M,N)} + T_{unlock})\right]$$

Initially, the system starts with all nodes processing their local tuples and thus, $n_0(0) = 0$ and $n_1(0) = N$. Computing the value of L in the cost formula for T is not necessary. We evaluate T iteratively as follows: At each step of the cost formula for T, the initial number of blocks in each node will be compared to i. If the number of blocks for a node is greater or equal to i then that node becomes a helping node. Hence, $n_0(i)$ is incremented and $n_1(i)$ is decremented accordingly. When $n_0(i)$ reaches N, this signals the end of processing. Thus, the summation of the cost formula for T is evaluated iteratively until $n_0(i)$ is equal to N.

Next, we present and discuss the analytical and experimental results.

4.4 Performance Evaluation

In this section we compare the performance of the original Distributed Hash Join (DHJ) of Chapter 3 and that of *DHJ* with load balancing (DHJLB). The section also compares the results of the analytical model with those of actual experiments. For

each plotted value representing experimental results in the figures, a number of runs were made with the same parameters' settings. The time for each run is measured as the maximum time over all the nodes in the system. The results from these runs would then be averaged, disregarding the first three runs to eliminate the overhead of system startup. The experiments were run on a BBN TC2000 with up to 25 nodes available for use. The analytical model, however, preserves the same trends shown in this section for an even larger number of nodes.

As mentioned in the previous chapter, the total size of available main memory is proportional to the number of nodes and so is the sizes of the input relations. For the performance figures in the rest of this section, N_R and N_S are set to 500 and 2500 respectively unless specified otherwise. In order to analyze the performance of the load balancing scheme, the input data was skewed using parameterized Zipfian distribution introduced in section 3.6.1.

To compare the performance of the analytical model to that of the experiments, the values of the system dependent parameters of Section 4.3 were taken from Table 3.1 of Chapter 3.

Figure 4.4 shows the total processing times for both DHJ and DHJLB algorithms. By transferring blocks of tuples from overloaded nodes and processing them locally, the helping nodes improve the performance of *DHJLB* over that of *DHJ* as shown in the figure. As the number of nodes increases, the total number of tuples in each relation also increases and so does the skew rate. Consequently, DHJLB has to deal with more load imbalance and thus, the balancing cost will increase. Note, however, that *DHJLB* maintains a near–constant improvement rate (i.e. over the performance of *DHJ*) as we increase the number of nodes in the system. For example, with a skew rate of $\theta = 0.8$ and for the whole range of nodes, the performance of *DHJ* is about 50% worse than that of *DHJLB*.

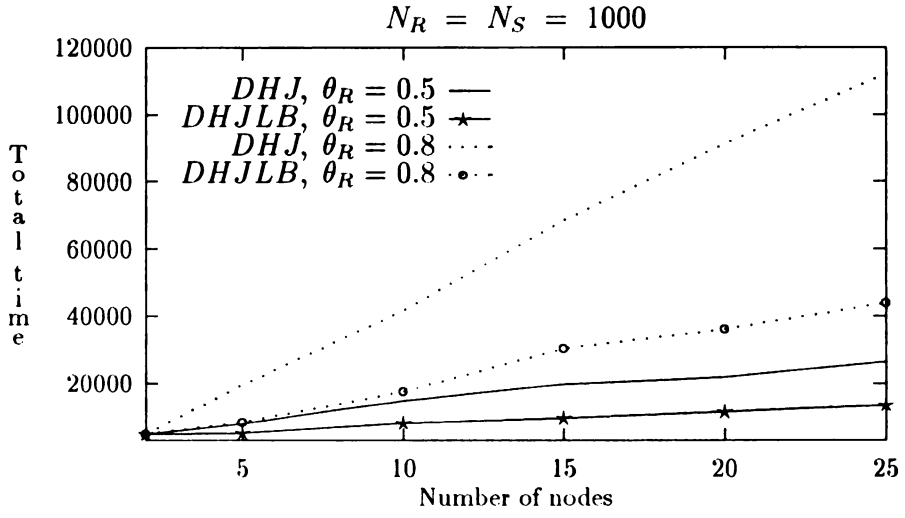

Figure 4.4. Total time for the join algorithm with and without load balancing.

4.4.1 Results of Varying Data Skew

Figure 4.5 shows the performance with no skew in the building relation, R, while varying the skew in the probing relation, S. In the range of skew (0–.2), all nodes will finish processing their tuples almost at the same time and thus, only minimal or no helping will be done. The figure shows that, in this range, both algorithms exhibit almost exact performance. This supports our claim that, in the case of no or low skew, *DHJLB* always performs as well as the original *DHJ* algorithm and thus outperforms most load balancing schemes which rely on collected statistics. The reason is that the balancing scheme does not incur any overhead since it does not require any prescanning or presampling of the input relations.

At high skew rates, few buckets within the nodes will have a large number of tuples. While the load balancing scheme is able to help with the other buckets, processing these large buckets takes longer times. The figure shows that the balancing scheme

is able to maintain good performance with increasing skew rates.

The effect of varying the skew rate in the building relation is presented in Figure 4.6. The figure shows the load balancing scheme to be effective in dealing with the problem of skew, even for large skew rates. The figure also shows the size, in number of tuples, of the largest bucket, i.e., the bucket that corresponds to the attribute with the highest skew value in Figure 3.11. This skew value increases sharply with increasing skew rate and at high skew, e.g. $\theta = 0.8$, the size of the corresponding bucket may become even larger than N_R.

Note that the effect of skewing the building relation on the performance of the join algorithm is more profound than that of skewing the probing relation. This is because tuples in the building relation have to be distributed among the local buckets and hashed and organized within each bucket. When the skew rate of R is high, these steps will take even longer to execute.

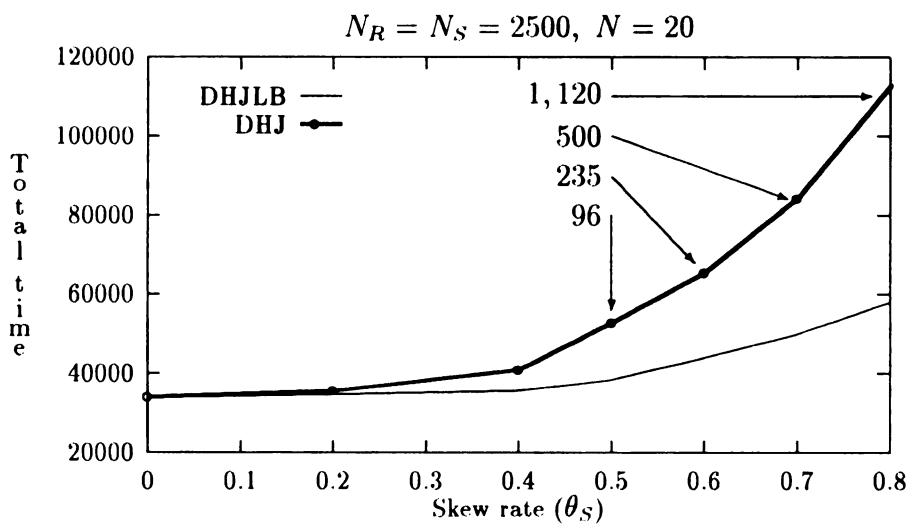

Figure 4.5. Total time with skew in the probing relation S.

Figure 4.7 presents the effect of varying the skew in both relations. From the figure, it is clear that computing the join with this kind of skew becomes very expensive. The main reason for this is that when both relations are skewed on the same attribute, the result which is quadratic in nature becomes excessively large with higher skew rates.

Consider, for example, the case where $\theta = 0.8$. We know that the sizes of the largest buckets are approximately 1000 tuples each (see Figure 4.6). The join of only these two buckets will generate close to $1,000,000$ tuples. Such joins take an excessive amount of time to compute and as noted in [21], are not practical. The figure shows that the load balancing scheme improves the performance of the join with these cases of double skew, even for high skew rates ($\theta = 0.7$). Results for skew rates higher than $\theta = 0.7$ were not reported since the amount of space needed for the results as well as the computation time were excessive.

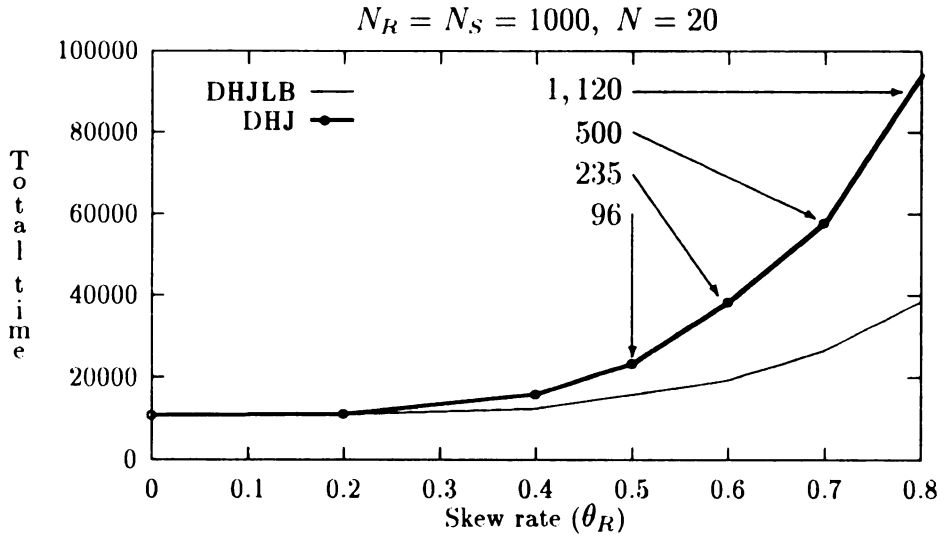

Figure 4.6. Total time with skew in the building relation R.

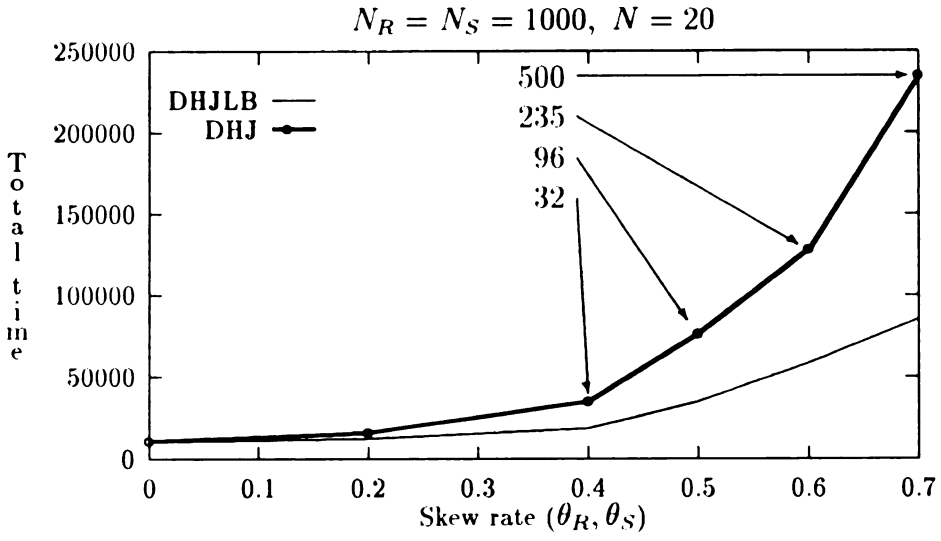

Figure 4.7. Total time with skew in both relations.

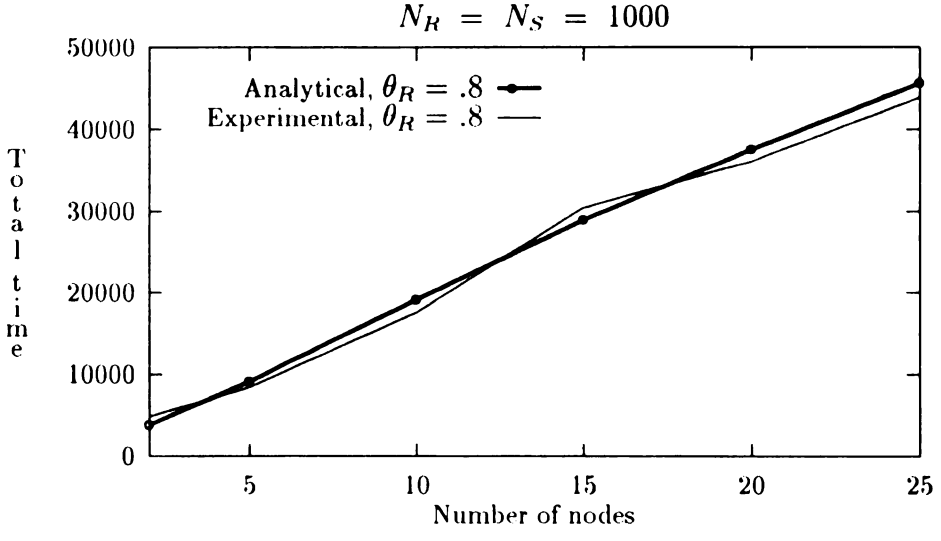

Figure 4.8. Comparison of analytical and experimental results.

4.4.2 Analytical vs. Experimental Models

The performance of the analytical model is compared with the experimental results in Figure 4.8. Since the total number of tuples is $N \times (N_R + N_S)$, when the number of nodes increases, the number of tuples to be processed increases and the data becomes more skewed. The figure shows this trend as the number of nodes is increased up to 25. It should be noted from the figure that the analytical results closely approximate those of the experiments. The maximum difference between the model and the experimental results is 9%. This slight difference can be explained by the same reasoning used to explain the differences in Figure 3.7.

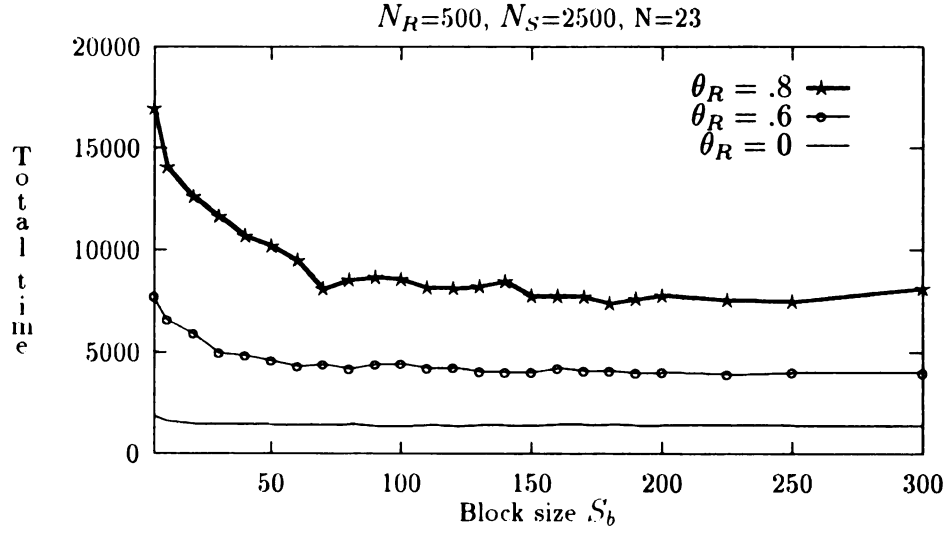

Figure 4.9. Effect of block size S_b on load balancing.

Before transferring a block of tuples for local processing, a helping node has to lock L_k, update I_k (as defined in Section 4.3) and then unlock L_k. Given a fixed number of tuples (during a specified run of the algorithm), a small block size means a larger

number of blocks and consequently more frequent lock accesses. As mentioned earlier, lock operations are expensive on NUMA systems and a large number of simultaneous lock operations can degrade the performance severely. The effects of different block sizes are shown in Figure 4.9 where it can be seen that for small block sizes, the increased lock conflicts degrade the performance severely. The figure also shows that the performance degradation for small block sizes is more profound for larger values of θ, i.e. higher skew rates. This is because higher skew rates mean more blocks will be transferred by the helping nodes during load balancing and thus more lock conflicts. Note that, as expected, varying the block size does not have much effect on the uniform distribution case ($\theta = 0$) since the number of blocks to move in this case is either zero or insignificantly small.

CHAPTER 5

THE NUMA WITH CLUSTERS OF PROCESSORS ARCHITECTURE

5.1 Introduction

Most of the proposed parallel database systems have been designed following either the Shared-Nothing or Shared-Everything architectures. While SN systems are supposed to scale up to hundreds of nodes [70] in the absence of data skew, their scalability becomes limited if moderate or severe data skew is present. Shared-Everything systems are able to deal efficiently with data skew but have been mostly limited to a small number of nodes *

As mentioned in Chapter 2, Hua and Lee [18] have proposed combining the two architectures into a hybrid system where clusters of SE systems are interconnected to form a SN system. According to their analysis, the network does not have a significant impact on the overall system performance. They show that I/O is the major factor and based on that, compute an optimal cluster size. However, their join algorithm for the hybrid system incurs extra I/O steps for writing the tuples to disk during the distribution phase and reading and writing them again during the first part of the join phase. By constructing the buckets as soon as the tuples are received, these extra I/O costs can be eliminated. In addition, load balancing across clusters

*Only up to 30 nodes in most systems.

was not considered and perfect balancing within each cluster was assumed.

Load balancing in SN systems can cause high network contentions and may consequently lead to performance degradations [91]. Parallel database systems with a large number of nodes and the ability to efficiently handle skewed data are necessary. Current NUMA–based multiprocessor systems can combine the high scalability of the Shared–Nothing architecture with the low–cost load balancing of Shared–Everything systems. These multiprocessors are able to provide very large I/O bandwidth by connecting more than one disk to a single processor or to a cluster of processors. With such features, these systems seem very desirable for parallel processing of database systems.

In this chapter, we investigate a NUMA–based multiprocessor system where each node is a cluster containing a number of processors. We term this architecture as NUMA with Clusters Of Processors or NUCOP. We present a parallel hash join algorithm for the NUCOP architecture that provides load balancing between all the nodes in the system. In previous work on SN systems [66, 73], the I/O was considered the main bottleneck in the system and the network was assumed to have infinite bandwidth.

We demonstrate a larger cluster size than that determined by Hua and Lee in [18] for the same set of system parameter values. Our results show that the network bandwidth is an important parameter that affects the system performance, especially when the number of nodes or clusters is large. By fixing the total number of nodes in the system and varying the sizes of the clusters, we show that NUCOP systems perform better than SN systems and can support a larger number of nodes that SE systems.

We develop an analytical model for the algorithm and validate it using both simulation and actual experimentation. The simulator developed for the NUCOP architecture has a modular design that allows it to model a variety of systems. This

can be accomplished by replacing some specific component, e.g. the global network, with a new one that may possess different properties such as topology and speed.

Our results show that the NUCOP systems are able to deal efficiently with skewed data while providing enough resources to process very large relations. An illustrative example of this is given in the last section of this chapter.

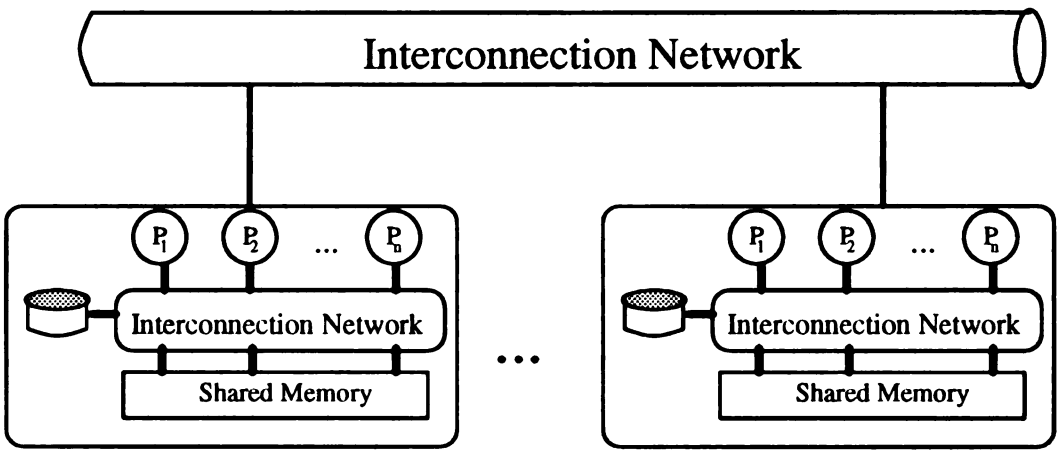

Figure 5.1. Architecture of NUCOP.

5.1.1 The NUCOP Architecture

This section presents our model of the NUCOP architecture. The system is shown in Figure 5.1. NUCOP consists of clusters of processors connected by an interconnection network. Each cluster contains a number of processors, a physically or logically shared main memory and a number of disks. All processors within the same cluster share the memory and disks. The network provides a globally shared memory by allowing processors in any cluster to access the memories in all other clusters. Clusters in NUCOP maintain a higher degree of locality than that provided by shared-nothing systems. This is because many of the remote references may be satisfied within

the cluster itself resulting in a lighter overall network load. This is essential for maintaining good performance with increasing number of nodes.

With a high degree of data skew in Hua and Lee's hybrid architecture [18], many clusters can be idle while the overloaded cluster is still processing its data. Since the clusters are connected to form a shared–nothing system, load balancing across clusters will limit the total number of clusters [91]. The main difference between our model and the hybrid architecture is that we use a shared–memory NUMA multiprocessor system. The globally shared memory allows for more efficient load balancing between all nodes in the system [73]. Another key difference is that the NUCOP system does not require the individual clusters to be UMA–based multiprocessors. This allows for a large number of nodes within each cluster.

In the NUCOP architecture, we assume the system to have parallel CPU, network and I/O capabilities. That is to say, the only CPU cost for executing a network or I/O request is the cost of initiating such a request. Commercial multiprocessors already provide various forms of these parallel capabilities. The KSR1 multiprocessor [84], for example, has separate I/O co–processors and provides parallel I/O operations through asynchronous system calls. Parallel data transfers over the interconnection network are also provided through the pre–fetch and post–store operations [84, 114]. These operations will be described in more details later in this chapter.

We next describe the join algorithm designed for this architecture and outline some of the key issues in its design.

5.1.2 The Proposed Join Algorithm

In order to describe the join algorithm, let m be the number of clusters in the system and n be the number of nodes per cluster. We assume that the attributes used to make the initial partitioning of both relations R and S are different from the join attributes. This means that both relations have to be distributed before the join can

be performed.

The join algorithm for the NUCOP architecture may now be described, at a high level, by the following two phases:

- **Distribution Phase:**

  Each node N_i reads and scans the local part of R. Each tuple is hashed and a destination node, N_j is computed. Tuples are sent to their destination nodes in blocks of size S_b. When a node receives a tuple, it hashes it using a second hashing function to determine the local bucket for that tuple. Tuples that fall into bucket zero are organized into an in-memory hash table that will be joined with bucket zero of S. All other buckets are written to disk.

- **Join Phase**

  When all nodes are done with the distribution of R, the processing of S starts. The process is similar to the distribution phase for R, except that when a tuple hashes into bucket zero of S, it is immediately joined with the matching tuples in bucket zero of R. All other buckets are written to disk.

  Once processing of bucket zero is done, processing of the other buckets starts. Each node will read the corresponding pair of R and S buckets from disk and perform their join.

We use the load balancing scheme of Chapter 4 in extending this hash join algorithm to deal with data skew. This extension works as follows:

- When a node is done performing the join on all its buckets, it checks for overloaded nodes within its cluster to help balance the load. Load balancing within each cluster can be done efficiently since the clusters provide shared access to the memory. This balancing phase is restricted to the current cluster and thus, does not generate any traffic on the global network connecting the various clusters.

- Once all tuples within a cluster have been processed, the nodes of this cluster probe other clusters and help balance the load across clusters. The selection of an overloaded node uses random probing as in Chapter 4. Probing remote nodes directly, as provided by the shared memory, allows efficient load balancing between the various nodes in the system [73].

5.2 Analytical Model

In this section, we develop the analytical model for the hash-based join algorithm presented for the NUCOP architecture. In order to model the data skew, we assume that the distribution of S will result in a skew on the receiving nodes. The amount of skew, θ, is defined as a fraction of the total relation size that is assigned to a single node. Based on published measures of partition skew, Walton et al [16, 72] state that typical values of θ range between .1 and .3. The rest of the relation is divided evenly among all the other nodes in the system. According to DeWitt *et al.* [21], this skew distribution captures the spirit of the Zipfian distribution (see Section 3.6.1) while being much simpler to manipulate analytically.

To compute the execution time of the algorithm, we assume that there is no overlap between the two phases of the join. This is necessary to ensure that all tuples of R that hashed to a certain node, have been actually received by that node before proceeding with the join phase. Hence, $C_{total} = C_1 + C_2$, where C_1 and C_2 are the costs of the first and second phases respectively. The synchronization between the two phases of the algorithm is similar to that described in Section 3.3.1.

Since the NUCOP system provides parallel CPU, network and I/O capabilities, I/O and network data transfers may be overlapped with CPU processing. Hence, the total cost for each of the phases may be computed as the maximum of the following:

$$C_1 = MAX(C_{CPU\_1}, C_{IO\_1}, C_{Net\_1})$$

and

$$C_2 = MAX(C_{CPU\_2}, C_{IO\_2}, C_{Net\_2}),$$

where C_{CPU_i}, C_{IO_i} *and* C_{Net_i} are respectively, the CPU, I/O and *Network* costs for phase i.

Note that we do not include a separate cost function for memory accesses as in [18]. Memory access is not usually executed in parallel with other computations. Hence, we feel the approach should be to include the costs of memory accesses with the CPU costs to account for the various read/write accesses to memory.

The performance evaluation of the dual-paradigm join algorithm in [73] did not consider the network cost, but rather assumed the network bandwidth to be always large enough based on previous experimentations. The network, however, is a resource that is shared by all clusters in the system and thus can form a performance bottleneck. We do consider the network cost in the analytical model and our results show that the network bandwidth has serious impact on the performance of the system.

During the processing of the join algorithm, conflicts may occur on the network as a number of clusters attempt to access other clusters. Conflicts can also occur within a cluster, as the various nodes within that cluster access each other or compete for I/O. These conflicts are modeled using the same formulation as in Section 3.4 with the appropriate parameters. A brief explanation of the various system and algorithm parameters used in the model along with their default values are presented in Appendix *A*. Next, we derive the costs for both join phases by formulating the individual costs mentioned above.

In the distribution phase:

- *I/O*: R_i is initially read from disk then, all tuples received from other nodes are written to disk except those in bucket zero. Thus, the number of tuples involved is $N_{IO\_1} = (R_i + R_i \times \frac{(B-1)}{B})$, where B is the number of buckets. The cost is

$$C_{IO\_1} = T_{IO} \times N_{IO\_1} \times \xi\left(\frac{(R_i \times n)}{B_s},\ n\right)$$

- *Network:* The tuples of R that hash to remote clusters are sent over the network. Assuming a uniform hashing function, each cluster will send $N_{Net\_1} = \frac{(m-1)}{m} \times R_i$ tuples to remote destinations. The total number of tuples to be sent over the network is then $N_{Net\_1} \times m$. Hence,

$$C_{Net\_1} = T_{Net} \times m \times N_{Net\_1} \times \xi\left(\frac{N_{Net\_1}}{B_s},\ N-n\right)$$

- *CPU:* Tuples of R are hashed to determine their destination nodes. Incoming tuples are also hashed into buckets and those that hash to bucket zero are further hashed and organized into an in memory hash–table. The cost of this is $T_{CPU} \times I_{hash} \times R_i \times (2 + \frac{1}{B})$. Each cluster will write $(N_{IO\_1} - R_i)$ tuples to disk and send $\frac{N_{Net\_1}}{m}$ tuples over the network. Thus, the CPU cost is:

$$C_{CPU\_1} = T_{CPU} \times I_{hash} \times R_i \times (2 + \frac{1}{B}) +$$

$$(N_{IO\_1} - R_i) \times \frac{I_{IO}}{(n \times W_{CPU} \times B_s)} + N_{Net\_1} \times \frac{I_{Net}}{(n \times W_{CPU} \times B_s)}$$

For the join phase, all cost computations that involve S will use K_s instead of S_i, where K_s represents the number of tuples received by the overloaded cluster. The costs are as follows:

- *I/O:* Read S_i from disk, store all received S tuples except those that fall in bucket zero. Join bucket zero of S with that of R and store the result to disk. Next, read each pair of R and S buckets, join them and store the results.

$$C_{IO\_2} = T_{IO} \times \left[S_i + \frac{(B-1)}{B} \times K_s + \frac{(B-1)}{B} \times (K_s + R_i)\right] \times \xi_{(\frac{K_s}{B_s},\, n)} +$$

$$T_{IO} \times \left[R_i \times (S_i + \frac{E_s}{m}) \times J_s\right] \times \xi_{(\frac{K_s}{B_s},\, n)}$$

- *Network:* Each cluster will initially send $\frac{(m-1)}{m} \times S_i$ tuples. During load balancing, the network will carry at most $\frac{(m-1)}{m} \times E_s$ tuples to be processed in the non–overloaded (helping) clusters.

$$C_{Net\_2} = T_{net} \times \left[(m-1) \times S_i + \frac{(m-1)}{m} \times (E_s + R_i)\right] \times \xi_{(\frac{(m-1)}{m} \times E_s \times \frac{1}{B_s},\, N-n)}$$

- *CPU:* Read S_i from disk, hash tuples to decide whether to keep them locally or send them to remote nodes. Tuples that stay locally have to be hashed again and organized into buckets. Receive incoming K_s tuples, hash them into buckets and except for bucket zero, write all buckets to disk. Join tuples that hash to bucket zero of S with those in bucket zero of R and write the result to disk. Read each pair of buckets, join them and write the result to disk. During load balancing, transfer a couple of buckets, join them locally and write the result to disk.

$$C_{CPU\_2} = T_{CPU} \times \left(\left[S_i + K_s + \frac{K_s}{B}\right] \times I_{hash} + \left[\frac{K_s}{B} \times \frac{R_i}{B} \times J_s\right] \times I_{join}\right) +$$

$$\left[S_i + K_s \times \frac{(B-1)}{B}\right] \times \frac{I_{IO}}{[n \times W_{CPU} \times B_s]} +$$

$$\left[S_i \times \frac{(m-1)}{m}\right] \times \frac{I_{Net}}{[n \times W_{CPU} \times B_s]} +$$

$$\frac{(B-1)}{B} \times \left[n \times S_{ir} + R_i + \frac{E_s}{m}\right] + \frac{E_s}{m} \times R_i \times J_s$$

5.3 Model Validation by Simulation

In order to verify and validate the analytical model, we developed a simulator for the NUCOP architecture. The simulator is written in CSIM which is a process-oriented simulation language [115]. Figure 5.2 presents a high level diagram of the simulator. The main components of the system are defined as separate objects. At the lower level are the *processor* objects, the *I/O* objects and the *local network* (*lNet*) objects. The *cluster* object includes one *lNet* object and a number of I/O and processor objects. The *lNet* object serves to connect the various processors and I/O objects within a cluster. Another main component of the simulator is the *global interconnection network* (*gNet*) object. The *gNet* object is used to connect the various cluster objects in the system.

The modular design of the simulator will allow it to model different system configurations with various types of networks, I/O and CPU components. By defining a new object with different characteristics such as configuration and speed, the simulator should be capable of modeling completely new systems. The components, currently available with the simulator, model only the KSR architecture. For example, the *lNet* object models a uni-directional slotted ring bus. It should not be hard to add new objects designed for different network topologies and have the simulator correctly model the new systems.

The relations, R and S, in the simulator are assumed to be initially partitioned across all nodes according to a uniform distribution. In order to model data skew, the relations are drawn from a distribution approximating that of Zipfian as discussed

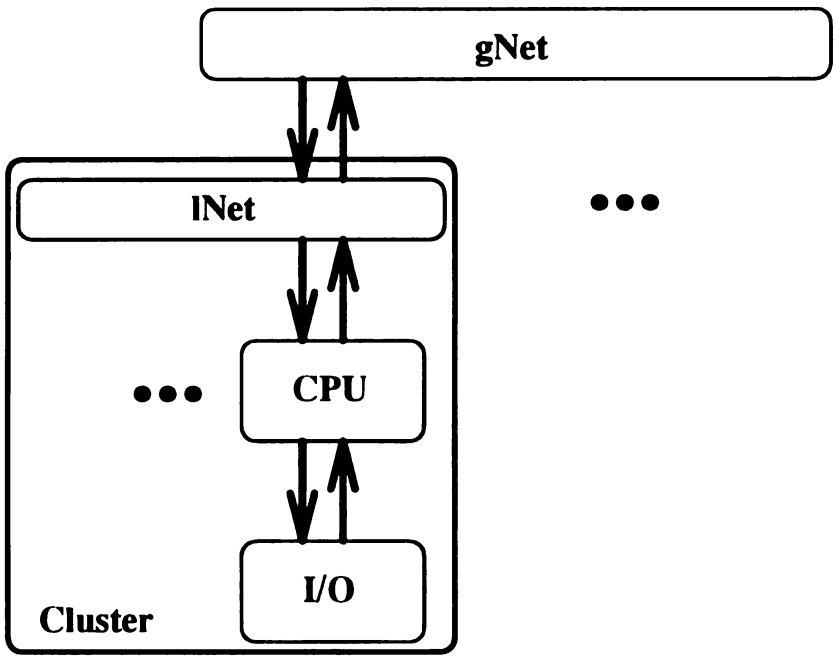

Figure 5.2. Diagram of the NUCOP simulator.

earlier in Section 5.2. All relations come from our synthetic database which is designed according to the Wisconsin Benchmark guidelines [107] (see Section 3.5.1).

In the simulator, tuples are represented by a pair of numbers, the tuple's key attribute and the size of the tuple in bytes. After hashing a tuple and deciding that it should be sent to another node, a counter representing the number of tuples that have so far hashed to that node is incremented. This counter is compared with the value of S_b, the maximum number of tuples per block. If the counter exceeds the value of S_b, then the block of tuples will be sent to the corresponding node and the counter is reset to zero. Otherwise, processing of the next tuple starts.

The actual sending of a block of tuples is simulated as follows. The block of tuples is divided into a number of packets (this number depends on the size of the packet, the size of the tuple and the value of S_b). The CPU object initiates a network request to attempt the delivery of these packets to the lNet object. The cost of initiating a network request is added to the CPU time of the current node for each packet that

```
; Comment lines start with the character ';'.
cpu     25
io      40
gnet    150
n       24
m       4
nr      8000
ns      8000
```

Figure 5.3. An example of an input file for the NUCOP simulator.

the CPU delivers to lNet.

Next, the lNet object processes the received packets and determines their destination nodes. The cost for lNet is updated accordingly. If the destination node belongs to another cluster, then the lNet delivers the packets to the gNet object. Each packet is then routed by the gNet to the correct cluster where it is delivered to the corresponding lNet. This lNet finally routes the packet to its destination node.

5.3.1 Simulator Input

The simulator accepts command line parameters but can also accept values included in an input file that is specified on the command line. The input syntax of this file is to include a single parameter per line with the parameter name, as defined in the glossary (Appendix B), followed by the desired value. This format is shown via an example in Figure 5.3. System parameters that are not specified neither in the input file nor on the command line will default to the values specified in Appendix B. These default values are built into the simulator by defining them in the program's header files. Since the input file is read last, parameter values that are specified in the file will

override any other values built into the program itself or supplied on the command line.

5.3.2 Simulator Output

The main output of the simulator are the timings for the various components, e.g., the total I/O, CPU and network time for each node. Optional debugging and tracing information for many of the CSIM objects are provided by invoking the program with the $-l$ option. The user is warned however, that this can and actually does generate excessively large output files. For example, during some of the debugging sessions of this simulator, output files of sizes up to 20 or 30 MB were not uncommon.

In order to compute the total timings for the join, the values were extracted from the columns corresponding to the CPU, I/O and network costs. The maximum value was then obtained for each stage of the join algorithm and the total time was computed as the sum of these values.

Appendix B presents more details concerning the implementation and current limitations of the NUCOP simulator. The appendix also contains a glossary of the various parameters used in the simulator along with the default values for these parameters. Performance figures obtained using the simulator are compared with those of the KSR1 experiments and the analytical model in Sections 5.4.2 and 5.5.

5.4 Model Validation on the KSR1

The KSR1 multiprocessor from Kendall Square Research is used for the purpose of testing and validating our analytical model. The KSR1 is a shared-memory multiprocessor that can have up to hundreds of nodes† Next we briefly describe the architecture of the KSR1, discuss the experimental results and compare them to the

†The current commercial version has 128 nodes but the architecture supports up to 1088 nodes.

results obtained from the analytical and simulation models.

5.4.1 KSR1 Architecture

The KSR1 is the first multiprocessor in a family of shared–memory systems designed by Kendall Square Research. It has up to 128 nodes with 32 MBytes of memory (which is actually a cache memory) and .5 megabyte sub–cache[‡] per node. Up to 32 nodes are connected to slotted, pipelined rings called Ring:0. Larger systems may be obtained by connecting a number of rings of type Ring:0 to a larger ring called Ring:1. The current system[§]consists of a Ring:1 connecting 4 rings of type Ring:0 and thus, has a total of:

- 128 nodes, 32 each in a ring of type Ring:0,
- 4 Gigabytes of cache, 32 MBytes local to each node and
- 50 Gigabytes of disk space.

The amount of disk space allowed for use was much more modest. That was the main reason behind running the experiments with only 8,000 tuples per node for each relation.

What is novel about the KSR1 is the notion of ALLCACHE memory. The main memory local to each node is treated as a large local cache and is considered a part of the single cashed address space available in the system. Data items referenced by a cell migrate automatically to its local cache. Thus, when a block of memory is sequentially accessed, only the first reference is considered remote while subsequent references are local.

As support for the parallel I/O and network capabilities in our model, the KSR1 provides asynchronous read/write operations to disk and post–store and pre–fetch

[‡]The sub–cache is divided equally into instruction and data sub–caches with $256KBytes$ each.
[§]The KSR1 we used for the experiments resided at the Cornell Theory Center.

operations over the network. Up to 3 concurrent network operations may be in progress for each node. During these operations, the processor at the requesting node is not stalled and can continue execution. Hence, the CPU cost for such operations is only the actual time needed to initiate them.

In order to make the experimental runs with up to 128 nodes on the KSR1 and have full, uninterrupted access to the whole system, the machine had to be set to *single–user* mode (where only one user is allowed to login during a certain period of time). The KSR1 at the University of Michigan runs the NQS queueing system. Users would submit their jobs to NQS and expect the results the next day. The system is reserved overnight for batch operations and the jobs are executed in single–user mode. However, this system is limited to only 64 nodes and that is why most of the initial small runs were executed here. The KSR1 at Cornell also ran NQS but the largest group of nodes was also 64. The full 128–node system was only available in single–user mode for two–hour slots every Thursday morning and only by reservation. The staff at the Theory Center generously allocated us five of these slots, over a period of two months, during which all the reported runs were made.

Next, we briefly discuss the implementation and compare the results of the experiments to those of the analytical and simulation models.

5.4.2 Comparison of Results

The join algorithm presented in Section 5.1.2 was implemented on the KSR1 multiprocessor system. The results of the analytical model presented in this section were obtained by using parameter values suitable for the KSR1 architecture. These parameter values were measured individually using a number of facilities on the KSR1 including the *pmon* monitoring tool to get information about the program at the hardware level. These values were also corroborated by other researchers who were also using the KSR1 [116] and in a number of published works [117, 114]. We first list

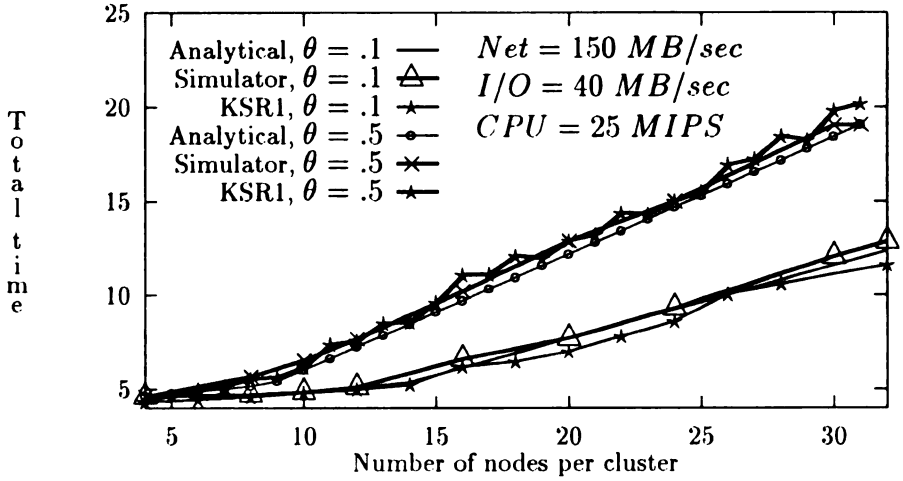

Figure 5.4. Validation of Analytical and Simulation models with Experimental results. Data points in the experimental results (obtained on the KSR1) represent the total execution time averaged over 3 runs.

these parameters, discuss the experimental runs and then compare the performances results.

The main system parameters for the KSR1 multiprocessor are:

- I/O bandwidth = 40 $MBytes/sec$ per cluster,
- Network bandwidth = 150 $MBytes/sec$,
- CPU speed = 25 $MIPS$.

To run the experiments on the KSR1, the relations were partitioned across all disks in the system and were generated to cause a distribution skew according to the value of the specified parameter, θ. Performance results were obtained while increasing the number of nodes per Ring:0 from 4 to 32, for a total of 16 to 128 nodes. As presented in most figures, the total size of the data (number of tuples in each relation) increases

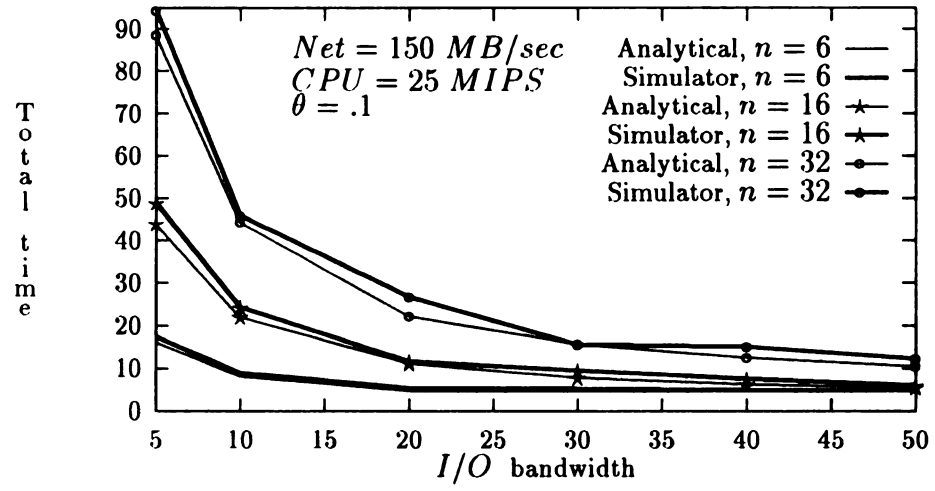

Figure 5.5. Effect of IO bandwidth.

linearly with the number of nodes. Each plotted data point in Figure 5.4 represents the total execution time averaged over 3 runs.

Figure 5.4 compares the analytical, simulation and experimental results for $N_R = N_S = 8K$ tuples per node. When the number of nodes per cluster increases, there are more I/O requests and transfers and the I/O system starts to saturate. The same is true for the global network connecting the various clusters and the local network connecting the individual processors within each cluster. However, because of the high network bandwidth on the KSR1, the IO cost dominates the performance. With higher skew rates, the amount of data (K_s) that has to be accessed from disk, in the skewed cluster, increases sharply with increasing number of nodes. For the KSR1 configuration tested in this work, the performance starts to degrade around 15 nodes per cluster for $\theta = .1$, and 9 nodes for $\theta = .5$.

By increasing the I/O bandwidth per cluster, the performance can be improved up to the limitation of the network bandwidth. This could not be tested experimentally

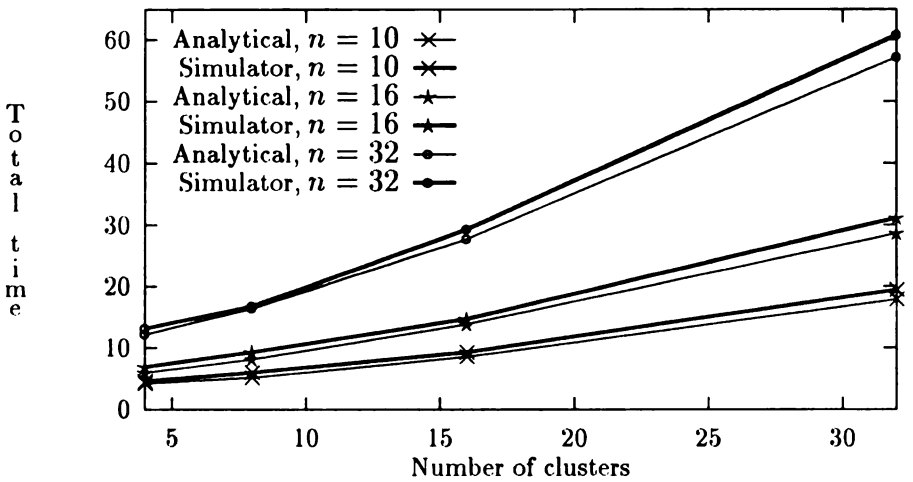

Figure 5.6. Effect of network bandwidth with various number of clusters.

since we do not have any control over the static configuration of the KSR1. The next section, however, experiments with changing the values of the parameters and studies the performance of the system using both the analytical and simulation models. Note that both the analytical and simulation models closely approximate the experimental results with a maximum difference of 8%.

5.5 Projections for Variant Architectures

In this section we study the performance of the algorithm by evaluating the cost functions under different models. We project on the performance of the system by varying the values of key system parameters, such as the I/O bandwidth, CPU speed or cluster size. The figures shown in this section illustrate the impact of these parameters on the performance of the overall system. We show that NUCOP provides good performance and can accommodate a large number of nodes. Unless noted

otherwise, all figures in this section were generated using the parameter values given in Appendix *A*. These values were used by both the simulator and the analytical model.

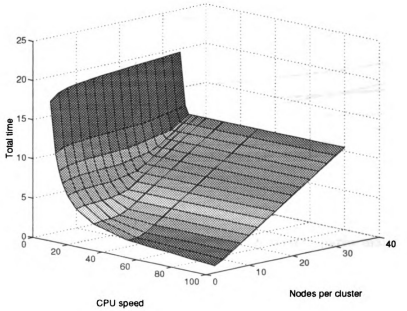

Figure 5.7. Effect of CPU speed.

5.5.1 Effect of I/O bandwidth

Figure 5.5 shows the performance of the system with increasing I/O bandwidth. The various curves represent different values for n, the number of nodes per cluster. As shown in the figure, when n increases, C_{IO} increases accordingly as the amount of data per cluster, to be accessed on disk, increases. Results from the analytical model and the simulation show that increasing the I/O bandwidth results in significant performance improvements in the lower bandwidth range, i.e. 5 to 20 MB/sec. The figure shows, however, that while larger I/O bandwidths do improve the overall

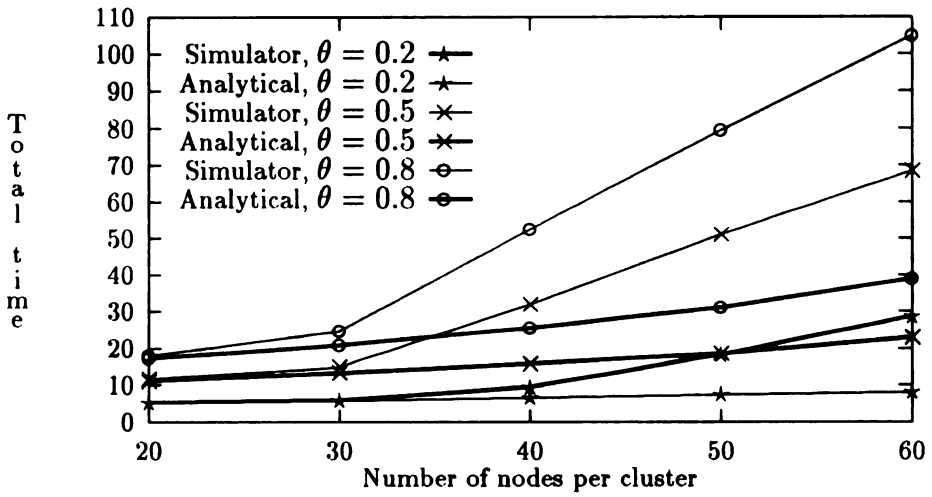

Figure 5.8. Effect of the cluster size on system performance.

performance, further increases in I/O speed do not yield similar improvements. For example, the bandwidth increase from 40 to 50 results in a very small improvement compared to that from 10 to 20.

5.5.2 Effect of network bandwidth

The network connecting the clusters is a shared resource that can form a performance bottleneck. Figure 5.6 shows the effect of increasing the number of clusters while keeping the network speed constant at $60MB/sec$. With increasing values of m, the total sizes of R and S, in number of tuples, increase. Consequently, the CPU, I/O and network costs also increase. However, what is important to note here is that while C_{IO} increases with the higher volume of data per cluster, C_{Net} is more severely affected because the network is shared among all nodes and not localized to individual clusters as is the case for C_{IO}. Note that on average, the amount of data to be sent over the network is proportional to $\frac{(m-1)}{m}$. Hence, increasing the value of m has a

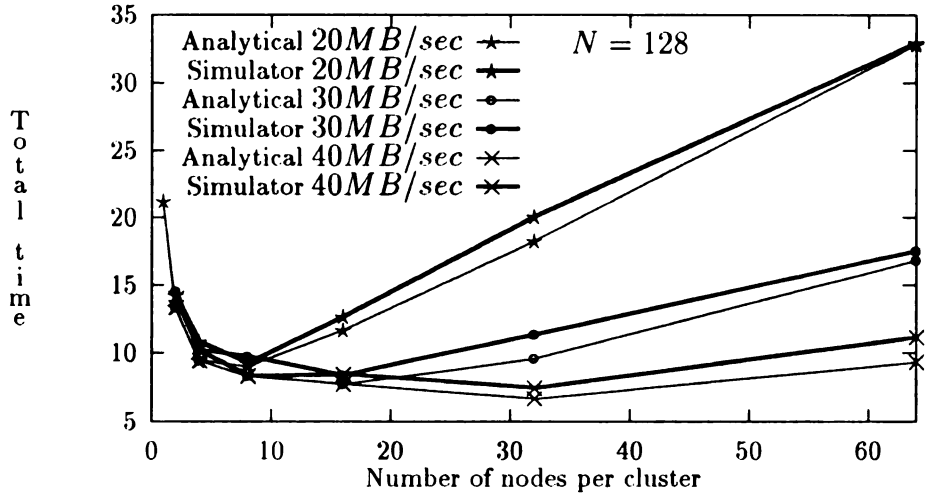

Figure 5.9. Total cost with constant N but varying n and m.

much more severe effect on C_{Net} than on C_{IO}.

The figure also presents the performance for different values of n. Increasing the value of n results in more nodes and consequently, larger relations. This, in turn, increases the load on the network and results in higher network costs.

It may seem, from Figure 5.6, that large cluster sizes (large n) degrade the system's performance by adding to the total load on the network. Section 5.5.6 focuses more on the effects of varying the cluster size and provides further details regarding the corresponding effects on performance. The section also compares the *NUCOP* architecture to previously proposed systems.

5.5.3 Effect of CPU speed

CPU speed is an important parameter for good performance but as Figure 5.7 shows, it is not as critical to the overall system performance as network and *I/O* bandwidths. With small *CPU* speeds, *e.g.* 5 and 10 *MIPS*, C_{CPU} dominates the performance of

the algorithm up to large values of n (28 nodes for $10MIPS$). However, the figure indicates that, for moderate CPU speeds, *e.g.* 20 *MIPS*, the performance is mostly dominated by either network or I/O costs. The technological trend indicates that CPU speeds have and continue to improve at a faster rate than network and I/O bandwidths. With currently available CPU speeds of 50 $MIPS$ and above, the contribution of C_{CPU} becomes almost negligible even for very small values of n, as shown in the figure.

5.5.4 Effect of large cluster sizes

Figure 5.5 showed that the I/O bandwidth poses a limitation on the total number of processors per cluster. Another important system parameter is the bandwidth of the local network (or bus) connecting the nodes within each cluster. This should also pose limitations on the total number of nodes that can be effectively used within the clusters. However, the KSR1 multiprocessor system has a maximum of 32 nodes per cluster and the bandwidth of Ring:0 is 1 GB/sec. With such configuration [84], the bandwidth of Ring:0 does not pose any performance limitations [118]. Thus, we could not conduct any experiments to check the performance effects of the local network bandwidth with various cluster sizes.

Here, we rely primarily on the simulator to investigate this aspect of the system's performance. The results are shown in Figure 5.8. To concentrate on the effects of the local network bandwidth, we changed that bandwidth to 50 MB/sec and also changed the I/O bandwidth to 100 MB/sec. The figure shows that, given such large I/O bandwidth per cluster, increasing the number of processors within the cluster, beyond a certain limit, will sharply degrade the performance of the local network.

The performance is shown with 3 different skew rates. Note that with higher skew rates, the performance degrades sharply. This is because the local network in the overloaded cluster has to transport the skewed data to the local nodes as well

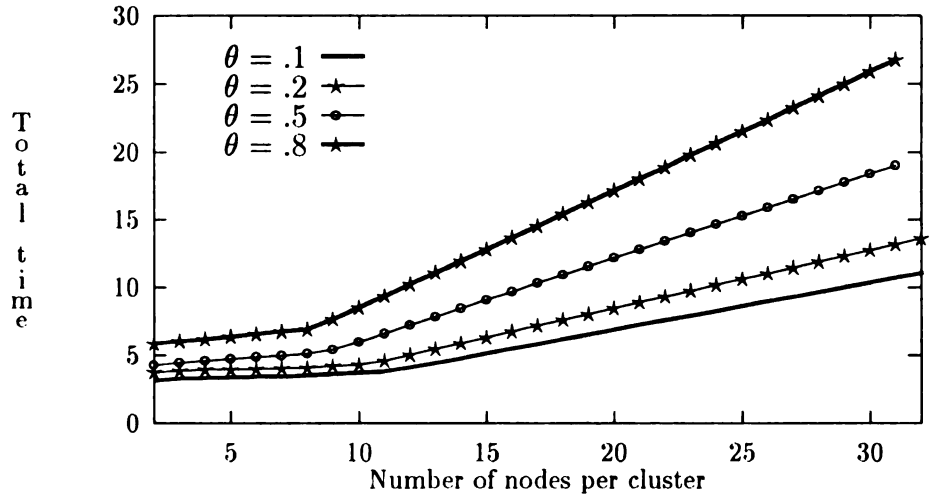

Figure 5.10. Total join cost for various degrees of data skew.

as to node within other clusters. Hence, the amount of data transported becomes too large for this local network and the performance starts to degrade. However, with reasonable values for the the different components, such as the values used in Figure 5.4 for the KSR1, we note that the I/O is the more restrictive component. Note that the analytical model does not exhibit this effect and only reflects a larger I/O load as the sizes of the clusters increase. This is because the bandwidth of the local network could not be incorporated in the formulation of the analytical model.

5.5.5 Effect of skew rate

Data skew is a very important factor that can severely affect the performance of parallel hash-based join algorithms [17]. Figure 5.10 shows the total cost of the join for various degrees of data skew. The figure shows that increasing the degree of skew increases the total cost accordingly. Note that for moderate skew rates, between 0.1 and 0.25, the cost function slightly increases. A larger skew rate means a larger value

for K_S which translates into:

- Higher local network cost at the (overloaded) cluster that received K_S,
- Higher I/O cost for writing and then reading K_S during the second phase of the algorithm and
- Slightly more CPU processing since there are more tuples to process in K_S.

With severely skewed data (*i.e.* $\theta = .8$) the I/O cost, in the skewed cluster, for storing and reading this very large number of tuples totally dominates the overall performance as shown in the figure.

5.5.6 Comparison to other work

Figure 5.9 shows the performance of the algorithm when the total number of nodes, N, and consequently the sizes of R and S are kept constant while the size of the clusters, n, is varied from 2 to 64. The figure attempts to include the spectrum of values for m and n when N is fixed at 128 nodes. The case of $m = N$ ($n = 1$) corresponds to the Shared–Nothing with Shared Virtual Memory system introduced by Shatdal and Naughton [73]. However, higher values of n correspond to larger SE systems within each cluster. The case of $n = 8$ approximates the performance of the system studied in [18].¶

Shatdal and Naughton [73] assumed the network to be of infinite bandwidth. This was based on results obtained from runs on the GAMMA database machine with a small number of nodes (typically 4 to 16), where the network bandwidth did not form a bottleneck. Note that since $N = m \times n$, increasing the cluster size means decreasing the number of clusters in the system. With a large number of clusters, the amount of data to be read from disk in each cluster is small and may be easily

¶Using 7, as in [18], instead of 8 would make m a fractional number.

handled by the available I/O bandwidth. However, the amount of data sent across the network by each cluster (C_{Net}) and the number of network requests generated by each node during distribution and load balancing are large. This is especially true in the presence of large data skew.

Figure 5.9 suggests that with a large number of nodes (or clusters of very small size) in the system, the network does form a bottleneck and the cost of network accesses may completely dominate the performance of the algorithm. However, an important conclusion here is that while network bandwidth does place a limitation on the scalability of the system in terms of total number of clusters, as is the case in pure shared–nothing systems, the NUCOP system has the potential of providing more nodes since each individual cluster can have a large number of nodes.

As the figure shows, by grouping a number of nodes into clusters and hence decreasing the number of clusters, the amount of data and requests to be sent over the network is minimized and the performance improves up to $n = 16$ (for $I/O = 30$ MBytes/sec). This shows that the NUCOP system is preferable to that introduced in [73]. Another major advantage is that load balancing within a cluster is handled locally inside the cluster and hence, does not interfere with across–clusters processing, thus reducing the total network load.

In the NUCOP system, load balancing across clusters improves the overall performance of the join algorithm as shown earlier in this section. However, as the cluster sizes increase, the I/O bandwidth can saturate, as more data is accessed per cluster, and start to dominate the performance as shown in Figure 5.9. It was shown earlier (Section 5.5.1) that I/O bandwidth is a limiting factor for the sizes of clusters in NUCOP systems. However, the figure suggests a larger optimal cluster size, 16 nodes, than what was previously determined in [18] for the same set of system parameters.

5.6 Concluding Remarks

In this chapter, we introduced a parallel join algorithm with load balancing for the NUCOP architecture. The analytical model, developed for this architecture, was validated with both simulation and actual experimentation on the KSR1 multiprocessor. The model and the simulator were also used to project on the performance of the system with various parameter settings. We investigated the effects of various key parameters on the performance of the system. We showed that the network bandwidth becomes a limiting factor when the number of clusters increases. We demonstrated a large cluster size and consequently showed that NUCOP systems can have a larger number of nodes than both SN and SE systems.

Given the large number of processors and the I/O and network bandwidths provided by the NUCOP architecture, systems based on this architecture are able to efficiently handle very large relations. Consider, for example, the processing of a database system that belongs to one of the credit agencies mentioned in Chapter 1. Assume that the database holds approximately 200 million records and has a total of five relations of equal sizes. This yields around 40 million tuples per relation. With 40 MB/sec and 160 MB/sec as the values of I/O and network bandwidths respectively, we can have $m = 16$ and $n = 14$ for a total of 224 nodes in the system. Assuming uniform distribution initially (using an appropriate partitioning strategy), each node will have around $174K$ tuples for each of the relations. By extrapolating on the results in Figures 5.5 and 5.6, the join can be computed in approximately six minutes.

CHAPTER 6

CONCLUSION AND FUTURE RESEARCH

We conclude that the proposed parallel hash join algorithms perform well on NUMA-based multiprocessor systems. These systems are more suited for load balancing than systems based on shared-nothing architecture. We also conclude that the NUCOP architecture can successfully meet the demands of current and future database systems based on the following facts:

- The proposed system scales better than previously proposed parallel database systems for performing relational joins.
- Very large relations are processed efficiently in terms of both processing time and *I/O* speed.
- Even with very large number of processors in the system, load balancing can be performed efficiently.

The contributions of this research can be summarized as follows:

- We introduced and studied two parallel hash join algorithms for main memory database systems on NUMA architecture. The first algorithm, Distributed Hash Join, uses distributed data structures where separate buckets are built local to each node to minimize access conflicts. Processor synchronization is also partially localized where processors increment the global counter and then wait till a local flag is set by the last processor to reach the barrier. This eliminates

the remote accesses that would otherwise be needed to test the current value of the global counter. By minimizing the access conflicts and localizing the processing of buckets, the DHJ algorithm was shown to exhibit linear performance improvements with increasing number of nodes and relation sizes.

We showed that Full Replication Join outperforms DHJ when the size of one of the relations is very small compared to the other. The analytical models of DHJ and FRJ can be used to determine the algorithm that performs better given the total number of nodes and the sizes of both relations. This can be used as a basis for a query optimizer in deciding which algorithm to use.

- Given the number of processors and the relations sizes (in number of tuples) and assuming the hash function to uniformly distribute the results, a probabilistic model was developed to determine the loads at the various processors. The model uses multinomial distribution to determine the maximum number of tuples received by the various nodes in the system. This allowed us to accurately predict the performance of the Distributed Hash Join algorithm using the analytical formulations.

- A distributed load balancing scheme was introduced in Chapter 4 to deal with the problem of data skew. The scheme was shown to perform well with increasing number of nodes and to be robust in dealing with various degrees of data skew. Random probing was shown to minimize the severe conflicts associated with probing sequentially and to have only a minimal effect on the activities of other helping nodes, e.g., transferring of blocks from overloaded nodes. An analytical model was derived that accurately captures the performance of the balancing scheme.

- The join algorithm proposed in Chapter 5 was implemented on the KSR1 multiprocessor system. Section 5.4.2 showed that with the KSR1 configuration

and for certain skew rates, only up to 15 nodes per cluster can be used effectively. By varying the system parameters, such as I/O and/or network speed, the analytical model for NUCOP architecture is able to accurately predict the performance of the system. The model was shown to be able to expose the scalability limitations of the given architecture for parallel database processing.

- The network bandwidth was shown to restrict the total effective number of processors when the cluster size is small. With larger cluster sizes, a larger proportion of the data is processed within the cluster itself and hence, the amount of data sent over the network decreases. This shows that for a given network bandwidth the total number of effective processors for NUCOP systems is larger than that of systems based on shared–nothing architecture.

- Simulation studies allowed us to evaluate the performance of the NUCOP architecture across a wide range of parameter values. The simulator was used to verify the analytical results and its output closely matched that of the KSR1. The simulator was designed in a modular fashion and has the ability to work with various configurations and different system parameters values. Because of the integration of functionalities within each component, the interactions between the workings of different components are kept down to a minimum and that allows for easier replacements. The main components of the system, i.e., the CPU, local and global networks and I/O can be replaced by new components with different topologies and speeds.

Several issues mentioned in this dissertation, but not studied thoroughly, provide the basis for future research. These include extending the simulator with new network and I/O configurations in order to model new systems with different configurations. Also of interest is the impact of multiple joins as well as multi–join operations on the load balancing scheme and on the overall performance of the system.

APPENDICES

APPENDIX A

SYSTEM PARAMETERS FOR THE NUCOP MODEL

The following table (A.1) presents the system and algorithm parameters used in the formulation of the analytical model for the NUCOP architecture in Chapter 5. The table presents each parameter along with a brief explanation and a default value. Unless otherwise noted, this is the value used in evaluating the various cost functions of the analytical model.

Parameter	Default Value	Explanation
n	32	Number of nodes per cluster
m	4	Number of clusters
N	$n \times m$	Total number of nodes in the system
N_R	8000	Number of R tuples per node
N_S	8000	Number of S tuples per node
T_R	$N_R \times N$	Total number of tuples in R
T_S	$N_S \times N$	Total number of tuples in S
Js	.00001	Join Selectivity
θ	.15	Skew Rate
t	128	Tuple size in number of bytes
b_s	300000	Bucket size in bytes
b_c	131072	Bucket size that fits in cache (128K)
B	25	Number of buckets
B_s	128	Block size in number of tuples
W_{io}	40	IO bandwidth (in MB/sec)
W_{cpu}	25	CPU speed (in MIPS)
W_{net}	100	Network bandwidth (in MB/sec)
I_{join}	1000	Cost of join (in number of instructions)
I_{io}	500	Cost of initiating an IO request
I_{net}	500	Cost of initiating a Network request
I_{hash}	300	Cost of hashing a tuple
R_i	$\frac{T_R}{n}$	Initial portion of R in cluster i
S_i	$\frac{T_S}{n}$	Initial portion of S in cluster i
S_{ir}	$\frac{(1-\theta)\times T_S}{N-1}$	Portion of S received by each node (except Skewed)
K_s	$\theta \times T_S + S_{ir} \times (n-1)$	Number of tuples in skewed cluster
E_s	$\theta \times T_S - S_{ir}$	Extra tuples in skewed cluster
T_{io}	$\frac{t}{W_{IO}}$	Cost of IO
T_{net}	$\frac{t}{W_{Net}}$	Cost of Network
T_{cpu}	$\frac{1}{n \times W_{CPU}}$	Cost of CPU processing

Table A.1. Various system parameters with their default values.

APPENDIX B

THE NUCOP SIMULATOR

In the following, we provide a detailed description of the simulator for the NUCOP architecture. The simulator is written in CSIM which is a process–oriented discrete–event simulation package for use with the *C* programming language. CSIM is implemented as a library of routines that the user may call from a regular *C* program. In CSIM, a system is modeled as a collection of predefined CSIM structures. Processes, which represent the active part of the system, interact with each other by visiting the CSIM structures. The structure provided by CSIM are:

1. Facility: These are servers that are either reserved or currently in use by some process.

2. Storage: A resource that can be partially allocated and deallocated by a process.

3. Event: This is the main resource used for synchronization.

4. Mailbox: A resource for interprocess communication.

5. Table: Resources to collect detailed information and statistics about the runs. There are a number of table types and each one is for collecting a different type of information.

Following are the definitions of the main components of the simulator:

1. **CPU**: An array of N facilities, where N is the number of processors.

2. **IO**: An array of m facilities, where m is the number of clusters. This represents a single I/O source per cluster.

3. **lNet**: An array of m multiple–server facilities. Each multiple–server facility simulates a slotted–ring bus with the specified number of slots.

4. **gNet**: A single facility through which all communication between the various lNets takes place.

5. **nReq**: An array of m multiple–server facilities. This is not a component of the simulator but it had to be incorporated for more accurate results. On the KSR1, each processor can have up to three pending network requests only. **nReq** is used to enforce such restriction.

In simulating the networks in NUCOP, a simplifying assumption was made concerning the locations of the empty slots on the ring. More precisely, it is assumed that if there is an empty slot on the ring, it can be used immediately regardless of its current position on the ring. While this assumption simplifies the implementation, our aim is not to extensively model the slotted ring in every aspect. Instead, the objective is to understand the effect of the network bandwidth on the performance of the system. By comparing the results from a number of simulator runs with some actual experimental data, we found that the assumption does not seem to have a great effect on the performance of the network.

In order to simulate actual processing of tuples by a given processor, the corresponding CPU facility will be *held* for the corresponding amount of time. Holding a facility is done by calling the function *use(f,t)*, where f is the facility and t is the amount of time required to process the current request. While a facility is in use by some process, all other processes have to wait till the current process releases the facility before they can use it.

Sending a block of tuples over the network involves first reserving the CPU for an amount of time equivalent to that of issuing a network request. Next, we check the *nReq* facility to see if there are already 3 pending requests and if there are, then we wait until one of the requests is done. This waiting time is added to the CPU time since the network request has thus far been unsuccessful. When the request is successful, the data reaches the local network *lNet*. Here, the first step is to calculate how many packets are needed and check if the destination node, *dn*, is within the same cluster as the source node, *sn*. Next, the local *lNet* is reserved for each packet to simulate the sending. If *sn* and *dn* belong to the same cluster, we are done sending the data. Otherwise, the *lNet* has to send the packets to the cluster of *dn* over *gNet*. Thus, *gNet* is reserved for each of these packets and then the *lNet* corresponding to the cluster of *dn* is reserved for the duration of the receive.

Since the network is modeling a slotted–ring bus, reserving the network for a packet results in reserving only one slot. All other slots may still be available and can be used simultaneously by other processors.

The overall design of the simulator is modular enough to allow adding new network topologies and I/O configurations. Time limitations did not, however, allow us to investigate other configurations than that of the KSR1 family of multiprocessor systems. Table B.1 presents the parameters used in the simulation along with brief description of each parameter.

B.1 Simulator Validation

In addition to being validated against the analytical and experimental results, the simulator was also validated with the following static test. The system consists of only two clusters with two nodes each. All parameters are assigned their default values (see Table B.1) except the sizes of R and S which were made smaller, $N_R = N_S = 500$.

We measured the number of requests submitted to gNet and to each lNet and I/O object and compared them against the expected values which were computed by hand. This is to make sure that all requests are being handled correctly by the different components of the system.

The main reason for choosing such a small system is to make the hand computations feasible. Even for a small system of this scale and modest values for N_R and N_S, the hand computations proved to be challenging.

Parameter	Default Value	Description
n	32	Number of processors per cluster
m	4	Number of clusters
N	$n \times m$	Total number of nodes in the system
N_R	8000	Average number of R tuples per processor
N_S	8000	Average number of S tuples per processor
J_s	.00001	Join Selectivity
θ	.15	Skew Rate
t	128	Tuple size in number of bytes
S_{pkt}	128	Packet size in bytes
B	100	Number of buckets
S_b	128	Block size in number of tuples
S_{page}	16384	I/O page size (16K bytes)
B_{IO}	40	IO bandwidth (in MB/sec)
B_{cpu}	25	CPU speed (in MIPS)
B_{lNet}	100	Local Network bandwidth (in MB/sec)
B_{gNet}	100	Global Network bandwidth (in MB/sec)
I_{io}	500	Cost of IO request (number of instructions)
I_{net}	500	Cost of Network request (number of ins.)
I_{hash}	300	Cost of hashing a tuple (number of ins.)
I_{join}	1000	Cost of join (number of ins.)

Table B.1. The major system parameters used to control the behavior of the NUCOP simulator.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, "GAMMA - a high performance dataflow database machine," in *Proceedings of the 12th International Conference on Very Large Data Bases*, pp. 228–237, 1986.

[2] D. J. DeWitt, S. Ghandeharizadeh, and D. Scheider, "A performance analysis of the GAMMA database machine," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 350–360, 1988.

[3] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping BUBBA: A highly parallel database system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, pp. 4–24, Mar. 1990.

[4] S. Friedman, "New HP models break performance barrier," *Open Systems Today*, p. 54, Nov. 1993.

[5] "Oracle predicts: Video-On-Demand key selling point," *Dow Jones/News Retrieval*, Nov. 1993.

[6] M. James, "Dangerous things strangers know about you," *McCall*, pp. 90–91, Jan. 1994.

[7] D. J. DeWitt and R. Gerber, "Multiprocessor hash-based join algorithms," in *Proceedings of the 11th International Conference on Very Large Data Bases*, pp. 151–164, 1985.

[8] E. Omiecinski and E. Tien, "A hash-based join algorithm for a cube-connected parallel computer," *Information Processing Letters*, vol. 30, pp. 269–275, Mar. 1989.

[9] J. L. Wolf, D. M. Dias, P. S. Yu, and J. J. Turek, "An effective algorithm for parallelizing hash joins in the presence of data skew," tech. rep., IBM T. J. Watson Research Center, RC 15510, 1990.

[10] M. Kitsuregawa and Y. Ogawa, "Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the Super Database Computer (SDC)," in *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 210–221, Aug. 1990.

[11] S. Ghandeharizadeh and D. J. DeWitt, "A multiuser performance analysis of alternative declustering strategies," in *Proceedings of the IEEE Data Engineering Conference*, pp. 466–475, 1990.

[12] S. Ghandeharizadeh and D. J. DeWitt, "Hybrid–range partitioning strategy: a new declustering strategy for multiprocessor database machines," in *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 484–492, 1990.

[13] C. A. Lynch, "Selectivity estimation and query optimization in large databases with highly skewed distributions of column values," in *Proceedings of the 14th International Conference on Very Large Data Bases*, pp. 240–250, 1988.

[14] M. S. Lakshmi and P. S. Yu, "Effect of skew on join performance in parallel architectures," in *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pp. 107–117, 1988.

[15] M. S. Lakshmi and P. S. Yu, "Limiting factors of join performance on parallel processors," *IEEE Transactions on Knowledge and Data Engineering*, pp. 488–496, Feb. 1989.

[16] C. B. Walton, "Four types of data skew and their effect on parallel join performance," tech. rep., Dept. of Computer Science, Univ. of Texas at Austin, TR–90–12, 1990.

[17] M. S. Lakshmi and P. S. Yu, "Effectiveness of parallel joins," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, pp. 410–424, Dec. 1990.

[18] K. A. Hua, C. Lee, and J. K. Peir, "Interconnecting shared–everything systems for efficient parallel query processing," *Proceedings of the 18th International Conference on Very Large Data Bases*, pp. 262–270, 1992.

[19] D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared–nothing multiprocessor environment," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 110–121, June 1989.

[20] E. Omiecinski, "Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor," *Proceedings of the 17th International Conference on Very Large Data Bases*, pp. 375–385, Sept. 1991.

[21] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *Proceedings of the 18th International Conference on Very Large Data Bases*, pp. 27–40, 1992.

[22] S. Y. W. Su, *Database Computers: Principles, architectures, and techniques.* McGraw-Hill Book Company, 1988.

[23] Teradata, *DBC/1012 Data Base Computer Concepts and Facilities.* Teradata Corporation, Release No. 1.1, C02-0001-01, 1984.

[24] Teradata, *DBC/1012 Data Base Computer Concepts and Facilities.* Teradata Corporation, Doc. No. C02-0001-05, 1988.

[25] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood, "Implementation techniques for main memory database systems," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 1–8, June 1984.

[26] D. J. DeWitt and D. A. Schneider, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proceedings of the International Conference on Parallel Processing*, pp. 110–121, 1989.

[27] M. Kitsuregawa, S. Suzuki, H. Tanaka, and T. Moto-Oka, "Relational algebra machine based on hash and sort," *IECE Japan Technical Group Meeting*, vol. EC-81, no. 35, 1981.

[28] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Relational algebra machine GRACE," *RIMS Symposia on Software Science and Engineering*, pp. 191–212, 1983.

[29] H. Schweppe, H. C. Zeidler, W. Hell, H. O. Leilich, G. Stiege, and W. Teich, *Advanced Database Machine Architecture*, ch. RDBM - A dedicated multiprocessor system for database management, pp. 36–86. Englewood Cliffs, NJ: Prentice Hall, Mar. 1984.

[30] D. J. DeWitt, "DIRECT - a multiprocessor organization for supporting relational database management systems," *IEEE Transactions on Computers*, vol. C-28, pp. 395–406, June 1979.

[31] D. J. DeWitt, "Query execution in DIRECT," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 13–22, May 1979.

[32] H. Boral, D. J. DeWitt, D. Friedland, and N. D. Jarrell, "Implementation of the database machine DIRECT," *IEEE Transactions on Software Engineering*, vol. SE–8, pp. 533–543, Sept. 1982.

[33] S. Hikita, H. Yamazaki, H. Hasegawa, and Y. Mitsushita, "Optimization of the file access method in content–addressable database machine (CADAM)," in *Proceedings AFIPS*, pp. 507–513, 1981.

[34] "IDM 400: Intelligent database machine product description," tech. rep., Britton Lee, Inc., 1981.

[35] "IDL tutorial," tech. rep., Britton Lee, Inc., 1981.

[36] Y. Tanaka, Y. Noxaka, and A. Masuyama, "Pipeline searching and sorting modules as components of a data flow database computer," *Information Processing Letters*, pp. 427–432, 1980.

[37] Y. Tanaka, "A data–stream database machine with large capacity," in *Proceedings of the International Workshop on Database Machines*, Sept. 1982.

[38] M. Kitsuregawa, M. Fushimi, H. Tanaka, and T. Moto-Oka, "Memory management algorithms in pipeline merger sorter," in *Proceedings of the Fourth International Workshop on Database Machines*, pp. 208–232, 1985.

[39] S. Fushimi, Kitsuregawa, and H. Tanaka, "An overview of a parallel relational database machine GRACE," in *Proceedings of the 12th International Conference on Very Large Data Bases*, pp. 209–219, 1986.

[40] G. Gardarin, "An introduction to SABRE: A multi–microprocessor database machine," *Sixth Workshop on Computer Architecture for Non–Numeric Processing*, June 1981.

[41] P. Valduriez and G. Gardarin, "Multiprocessor join algorithms of relations," *Second International Conference on Data Bases: Improving Usability and Responsivness*, pp. 219–237, June 1982.

[42] P. Valduriez, "Semi–join algorithms for multiprocessor systems," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 225–233, June 1982.

[43] G. Gardarin, P. Bernadat, N. Temmerman, P. Valduriez, and Y. Viemont, "SABRE: A relational database system for a multi-microprocessor machine," *Advanced Database Machine Architecture*, pp. 19–35, 1983.

[44] N. Miyazaki, T. Kakuta, S. Shibayama, H. Yokota, and K. Murakami, "An overview of relational database machine Delta," in *Proceedings of the Advanced Database Symposium*, pp. 11–20, Dec. 1984.

[45] H. Sakai, K. Iwata, S. Kamiya, M. Abe, A. Tanaka, S. Shibayama, and K. Murakami, "Design and implementation of the relational database engine," tech. rep., Institute for New Generation Computer Technology, Tokyo, Japan, Apr. 1984.

[46] K. Iwata, S. Kamiya, H. Sakai, S. Matsuda, S. Shibayama, and K. Murakami, "Design and implementation of a two–way merge–sorter and its application to relational database processing," tech. rep., Institue for New Generation Computer Technology, Tokyo, Japan, May 1984.

[47] S. Uemura, T. Yuba, A. Kokubu, R. Ooomote, and Y. Sugawara, "The design and implementation of a magnetic–bubble database machine," *Information Processing Letters*, pp. 433–438, 1980.

[48] S. Y. W. Su and C. K. Baru, "Dynamically partitionable multicomputers with switchable memory," *Journal of Parallel and Distributed Computing*, vol. 1, pp. 152–184, Nov. 1984.

[49] C. K. Baru and S. Y. W. Su, "Performance of statistical aggregation operations in the SM3 system," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 77–89, June 1984.

[50] C. K. Baru and S. Y. W. Su, "The architecture of SM3: A dynamically partitionable multicomputer system," *IEEE Transactions on Computers*, vol. C–35, pp. 790–802, Sept. 1986.

[51] A. K. Thakaore and S. Y. W. Su, "Matrix inversion and LU decomposition on a multicomputer system with dynamic control," in *Proceedings of the Second International Conference on Supercomputing*, vol. 1, pp. 291–300, 1987.

[52] H. Auer, W. Hell, H. O. Leilich, H. Schweppe, G. Stiege, S. Seehusen, J. Lie, H. Zeidler, and W. Teich, "RDBM – a relational database machine," *Information Systems*, vol. 6, no. 2, pp. 91–100, 1981.

[53] J. D. Brownsmith and S. Y. W. Su, "Performance analysis of the EQUI-JOIN operation in the MICRONET computer system," in *Proceedings of the ICCC Conference*, pp. 264–268, Oct. 1980.

[54] T. B. Genduso and S. Y. W. Su, "An analytical model of the MICRONET distributed database management system," in *Proceedings of the Third International Conference on Distributed Computing Systems*, pp. 232–239, Oct. 1982.

[55] S. Y. W. Su and K. P. Mikkilineni, "Parallel algorithms and their implementation on MICRONET," in *Proceedings of the 8th International Conference on Very Large Data Bases*, 1982.

[56] S. Y. W. Su, "A microcomputer network system for distributed relational databases: Design, implementation, and analysis," *Journal of Telecommunication Networks*, vol. 2, no. 3, pp. 307–334, 1983.

[57] J. R. Goodman, "An investigation of multiprocessor structures and algorithms for database management," tech. rep., University of California at Berkeley, 1980.

[58] J. R. Goodman and A. M. Despain, "A study of interconnection of multiple processors in a database environment," in *Proceedings of the International Conference on Parallel Processing*, pp. 269–278, Aug. 1980.

[59] D. K. Hsiao, *Collected readings on a database computer (DBC).* The Ohio State University Press, 1980.

[60] J. Banerjee, D. K. Hsiao, and K. Kannan, "DBC – a database computer for very large databases," *IEEE Transactions on Computers*, vol. C–28, pp. 414–429, June 1979.

[61] R. K. Shultz and R. J. Zingg, "Response time analysis of multiprocessor computers for database support," *ACM Transactions On Database Systems*, pp. 14–17, 1984.

[62] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to database machine and its architecture," *New Generation Computing*, vol. 1, pp. 63–74, 1983.

[63] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Architecture and performance of relational algebra machine GRACE," in *Proceedings of the International Conference on Parallel Processing*, pp. 241–250, Aug. 1984.

[64] M. Kitsuregawa, S. Tsudaka, and M. Nakano, "Parallel GRACE hash join on shared–everything multiprocessor: Implementation and performance evaluation on Symmetry S81," *IEEE Transactions on Knowledge and Data Engineering*, pp. 256–264, 1992.

[65] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The design of XPRS," in *Proceedings of the 14th International Conference on Very Large Data Bases*, pp. 318–330, 1988.

[66] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The GAMMA database machine project," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 44–62, 1990.

[67] G. Graefe, "Encapsulation of parallelism in the Volcano query processing system," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 102–111, 1990.

[68] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in BUBBA," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 99–108, May 1988.

[69] M. H. Kim and S. Pramanik, "Optimal file distribution for partial match retrieval," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 173–182, June 1988.

[70] D. DeWitt and J. Gray, "Parallel database systems: The future of high performance database systems," *Communications of the ACM*, vol. 35, pp. 85–98, June 1992.

[71] K. A. Hua and C. Lee, "Handling data skew in multiprocessor database computers using partition tuning," in *Proceedings of the 17th International Conference on Very Large Data Bases*, pp. 525–535, Sept. 1991.

[72] C. D. Walton, A. G. Dale, and R. M. Jenevin, "A taxonomy and performance model of data skew effects in parallel joins," *Proceedings of the 17th International Conference on Very Large Data Bases*, pp. 537–548, 1991.

[73] A. Shatdal and J. F. Naughton, "Using shared virtual memory for parallel join processing," in *Proceedings of the ACM Special Interest Group on Management Of Data*, pp. 119–128, 1993.

[74] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *Transactions on Computer Systems*, vol. 7, pp. 321–359, Nov. 1989.

[75] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, pp. 509–516, Dec. 1992.

[76] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut, "PRISMA/DB: A parallel, main memory relational DBMS," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, pp. 541–554, Dec. 1992.

[77] A. C. Ammann, M. B. Hanrahan, and R. Krishnamurth, "Design of a memory resident DBMS," in *Proceedings of the 1985 IEEE COMPCOM Conference*, pp. 54–57, 1985.

[78] D. Bitton, M. B. Hanrahan, and C. Turbyfill, "Performance of complex queries in main memory database systems," in *Proceedings of the IEEE Data Engineering Conference*, pp. 72–81, Feb. 1987.

[79] K.-Y. Whang and R. Krishnamurthy, "Query optimization in a memory–resident domain relational calculus system," *ACM Transactions On Database Systems*, vol. 15, pp. 67–95, Mar. 1990.

[80] T. J. Lehman and M. J. Carey, "Query processing in main memory database management systems," in *Proceedings of the ACM Special Interest Group on Management Of Data*, (Washington, DC), May 1986.

[81] M. H. Eich, "A classification and comparison of main memory database recovery techniques," in *Proceedings of the International Conference on Data Engineering*, pp. 332–339, Feb. 1987.

[82] L. Gruenwald and M. H. Eich, "MMDB reload algorithms," in *Proceedings of the ACM Special Interest Group on Management Of Data*, (Denver, Colorado), pp. 397–405, May 1991.

[83] D. Gawlick and D. Kinkade, "Varieties of concurrency control in IMS/VS Fast Path," *Data Engineering Bulletin*, vol. 8, pp. 3–10, June 1985.

[84] "KSR1 principles of operations, rev. 6.0," tech. rep., Kendall Square Research, Waltham, MA, 10 1992.

[85] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessey, "The directory-based cache coherence protocol for the DASH multiprocessor," *In 17th International Symposium on Computer Architecture*, pp. 148–159, May 1990.

[86] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, pp. 63–79, Mar. 1992.

[87] BBN, "Inside the Butterfly TC2000," tech. rep., BBN Advanced Computers, Cambridge, MA, Feb. 1990.

[88] D. R. Cheriton, H. A. Goosen, and P. D. Boyle, "Paradigm: A highly scalable shared-memory multicomputer architecture," *IEEE Computer*, vol. 24, pp. 33–46, Feb. 1991.

[89] E. Hagersten, A. Landing, and S. Haridi, "DDM – A cache-only memory architecture," *IEEE Computer*, pp. 44–54, 1992.

[90] M. C. Taylor, "Parallel multi-join algorithms for main memory databases," in *Proceedings of the International Conference on Parallel Processing*, 1989.

[91] H. Lu, K. Tan, and M. Shan, "Hash-based join algorithms for multiprocessor computers with shared memory," *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 198–208, 1990.

[92] V. Deshpande and P. A. Larson, "The design and implementation of a parallel join algorithm for nested relations on shared-memory multiprocessors," *IEEE Transactions on Knowledge and Data Engineering*, pp. 68–77, 1992.

[93] W. T.-Y. Hsu and P.-C. Yew, "An effective synchronization network for hot-spot accesses," *Transactions on Computer Systems*, vol. 10, pp. 167–190, Aug. 1992.

[94] S. P. Dandamudi and D. L. Eager, "Hot-spot contention in binary hypercude networks," vol. 4, pp. 239–245, Feb. 1992.

[95] C. Severance, 1989.

[96] E. F. Gehringer, J. Abullarade, and M. H. Gulyn, "A survey of commercial parallel processors," *IEEE Computer*, vol. 22, pp. 75–107, Aug. 1989.

[97] S. B. Yao, "Approximating block accesses in database organizations," *Communications of the ACM*, vol. 20, pp. 260–261, Apr. 1977.

[98] C. Berge, *Principles of Combinatorics.* Academic Press, 1971.

[99] G. Berman, *Introduction to Combinatorics.* Academic Press, 1972.

[100] A. B. Slomson, *An Introduction to Combinatorics.* Chapman and Hall, 1991.

[101] I. Tomescu, *Introduction to Combinatorics.* Collet's Publishers LTD, 1975.

[102] J. H. Lint, *A Course in Combinatorics.* Cambridge Universal Press, 1992.

[103] D. Knuth, *The art of programming*, vol. 3. 1972.

[104] G. Dahlquist, *Numerical Methods.* Prentice Hall, 1974.

[105] G. E. Forsythe, *Computational methods for Mathematics.* Prentice Hall, 1977.

[106] *Numerical Recepies.* Prentice Hall, 1994.

[107] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking database systems – A systematic approach," in *Proceedings of the 9th International Conference on Very Large Data Bases*, pp. 8–19, Oct. 1983.

[108] G. Zipf, "Human behavior and the principle of least effort: An Introduction to Human Ecology," *Addison–Wesley*, 1949.

[109] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek, "An efficient algorithm for parallelizing hash joins in presence of data skew," in *Proceedings of the International Conference on Data Engineering*, pp. 200–209, Apr. 1991.

[110] W. Tout and S. Pramanik, "A distributed load balancing scheme for data parallel applications," in *Proceedings of the International Conference on Parallel Processing*, pp. II:213–II:216, Aug. 1993.

[111] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, vol. 12, pp. 662–675, May 1986.

[112] G. Ananth, A. Gupta, and V. Kumar, "Isoefficiency function: A scalability metric for parallel algorithms and architectures," tech. rep., Department of Computer Science, University of Minnesota, 1992.

[113] A. Gupta and V. Kumar, "Analyzing performance of large scale parallel systems," in *Proceedings of the 26th Hawaii International Conference on System Sciences*, 1993.

[114] E. Rosti, E. Smirni, T. D. Wagner, A. W. Apon, and L. W. Dowdy, "The KSR1: Experimentation and modeling of Poststore," tech. rep., Department of Computer Science, Vanderbilt University, Nashville, TN, 1993.

[115] H. Schwetman, "CSIM users' guide," tech. rep., Micro Electronics and Computer Technology Corp., MCC tech report ACT–126–90, Mar. 1990.

[116] C. Larsen, "KSR1 technical information." Kendall Square Research, 1993. Personal contact through email.

[117] R. H. Saavedra, R. S. Gaines, and M. J. Carlton, "Micro benchmark analysis of the KSR1," *Transactions on Computer Systems*, pp. 202–213, 1993.

[118] T. H. Dunigan, "Kendall square multiprocessor: Early experiences and performance," *Kendall Square Research: Technical Notes*, Aug. 1992.