Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

REMOVING SKEW EFFECT IN JOIN OPERATION ON PARALLEL PROCESSORS

Ron-Chung Hu
Richard R. Muntz

June 1989
CSD-890027

Removing Skew Effect in Join Operation on Parallel Processors

Ron-Chung Hu † *Richard Muntz* ‡

Computer Science Department
University of California
Los Angeles, CA 90024-1596

ABSTRACT

A parallel processor architecture, with each relation horizontally fragmented, is expected to provide expandability along with linear performance improvement in database applications. However, the performance improvement of the relational join operation on parallel architectures may be severely limited by a phenomenon, referred to as data skew, in the join attributes, i.e. many tuples with a particular attribute value. This data skew phenomenon is very common in many real world databases, and its effect can be unbalanced load on processors and a less than linear speedup. It has been shown that a straightforward generalization of the conventional join algorithm is not adequate to handle the data skew problem in a parallel architecture with a large number of processors.

In this paper, we present two multiprocessor join algorithms which avoid unbalanced load among the processors caused by the data skew phenomenon. We show how our algorithms evenly distribute the load in join processing and effectively solve the skew problem. We further identify the domains in which each algorithm yields best join performance.

† Ron-Chung Hu was partially supported by Teradata Corporation.

‡ Richard Muntz was partially supported by DARPA contract F29601-87-C-0072.

1. INTRODUCTION

With the advent of low cost computer communication and microprocessor technologies, database systems using multiple microprocessors in parallel have become technically and economically feasible. In recent years, there have been several successful prototypes and commercial systems making use of parallel processing in database operations. Typical examples are MCC's Bubba system [Bora88], University of Wisconsin's Gamma machine [Dewi88], and Teradata's DBC/1012 [Tera88]. All these systems horizontally fragment a data relation across the processors based on the range of primary key value [Dewi88], hashing mechanisms [Dewi88] [Tera88], or access frequencies [Cope88]. Based on this mechanism, database operations, like select, project, and join, can be performed in parallel and their response times can be significantly reduced. The performance improvement on the parallel processor architecture is (ideally) proportional to the number of parallel processors involved in the operation when the data are evenly distributed across the processors.

In real world databases, some values for an attribute occur more frequently than the other values. Often the distribution is so skewed that even applying a very good hash function can not lead to a uniform distribution. When this kind of skewed distribution occurs in a join attribute on a parallel processor architecture, distributing tuples in the join operation can result in some processors being assigned a disproportionate number of tuples and a concomitantly disproportionate load. It has been shown that, using the current multiprocessor hash join algorithm, the skewed distribution present in the join attribute can lead to load skew and severely impair the join performance [Laks88] [Laks89]. In this paper, we first study the skew phenomenon and its effect on the join operation. We then introduce two multiprocessor hash join algorithms, which take full advantage of parallel processing by avoiding the load skew effect. Our algorithms can effectively remove the skew effect in the join operation on parallel processors.

We discuss the skew phenomenon in more detail in section 2. In section 3, we briefly describe the parallel multiprocessor architecture and outline the multiprocessor hash join algorithm for the architecture under consideration. In section 4, we give the parameters affecting the join performance and derive the join response time formula. From the response time formula, we examine the skew effect on the performance of the parallel processor join operation. In section 5, we present two multiprocessor hash join algorithms and show how they can effectively eliminate the skew problem. We discuss some implementation issues and identify the domains in which our

algorithms can provide good performance in section 6. Finally, the conclusions are drawn in section 7.

2. DATA SKEW

It is well known that the uniformity assumption is not realistic for describing the distribution of attribute values in a relation [Chri83]. Data bases often contain information describing populations of the real world, and many tuples can have the same attribute value. For instance, a relation storing all the UCLA students can lead to many tuples with the same value 'California' in the state attribute since more than 70% of UCLA's students are from California. This phenomenon is so common that it is evident in many databases. Lakshmi and Yu referred to this phenomenon as *data skew* [Laks88]. It should be noted that, once data skew exists in an attribute, applying a good hash function to the attribute cannot lead to an even distribution no matter how perfect the hash function is. This is because many tuples have the exactly same attribute value and they will have the same hash value. There are two data skew cases on join attributes: *single skew* means only one relation has a skewed distribution in the join attribute, while *double skew* means both relations have skewed distributions in their join attributes [Laks89].

To illustrate the data skew effect in a join operation, we consider the following class enrollment database in which there are three relations: Course, Student, and Teacher. The Course relation, which has attributes CourseId, course Name, and course Description, contains all the courses which are offered in a university. The Student relation contains the information as to which students are enrolled in which courses. Some courses, such as course 102, are a fundamental course and a requirement for all students. Some courses are more specialized and fewer students may have taken these courses. Clearly, there exists a data skew in the CourseId attribute of the Student relation. The Teacher relation contains the information as to which teacher can instruct which courses. Dependent on a teacher's background, some courses, like course 102, can be given by several teachers while some courses, like course 104, can be taught by only one teacher. Therefore there is also a data skew in the CourseId attribute of the Teacher relation. Suppose a course can be offered only if there is a student enrolling for it. If we want to list the names of all the courses to be offered, we have to perform a query by joining Course relation and Student relation on the CourseId attribute. This query will have single skew since one relation, Student relation, has a skewed distribution in the join attribute. If we want to retrieve all the teachers who

can instruct the offered courses, the query would join the Student relation and the Teacher relation on the CourseId attribute. We have an example of double skew in this case.

Course relation

CourseId	Name	Description
101	biocybernetics	- -
102	calculus	- -
103	data structure	- -
104	neural nets	- -
105	nuclear fusion	- -

Student relation

StudentName	CourseId	Credit
Bud Genius	102	- -
Don Duck	102	- -
Don Duck	104	- -
Bud Genius	103	- -
Holly Wood	102	- -
Sue Watt	102	- -

Teacher relation

TeacherName	CourseId	Time
Rex Carrs	102	- -
Buz Erk	102	- -
Matt Matix	102	- -
Fran Tastik	101	- -
Buz Erk	103	- -
Buz Erk	104	- -
Fran Tastik	102	- -

Figure 1. Class Enrollment database

In this paper, we first deal with the single skew case because of its simplicity. We then extend the solution to the double skew case.

3. MULTIPROCESSOR HASH JOIN

We first describe the architecture of the parallel processor system under consideration. The parallel processor architecture in this paper is essentially a shared-nothing multiprocessor system

[Ston86], in which there are two kinds of processors: Interface Processor (IP) and Access Processor (AP) [Nech84]. The IP's handle the interaction with users. They receive queries from users and decide the query access plans, which are broadcast to all the involved AP's. The AP is designed to have a shared-nothing structure: each AP has its own cpu, main memory, and disk. The tuples of each relation are distributed across all the AP's based on the hash value of a tuple's primary key [Tera88]. Each AP is in charge of the access to the tuples on its disk. The IP's and AP's communicate and coordinate with each other by passing messages through the interconnection network. Since the network physically connects a large number of processors, it must be very fast in order to handle a high volume of messages. The high-level architecture is illustrated in Figure 2.

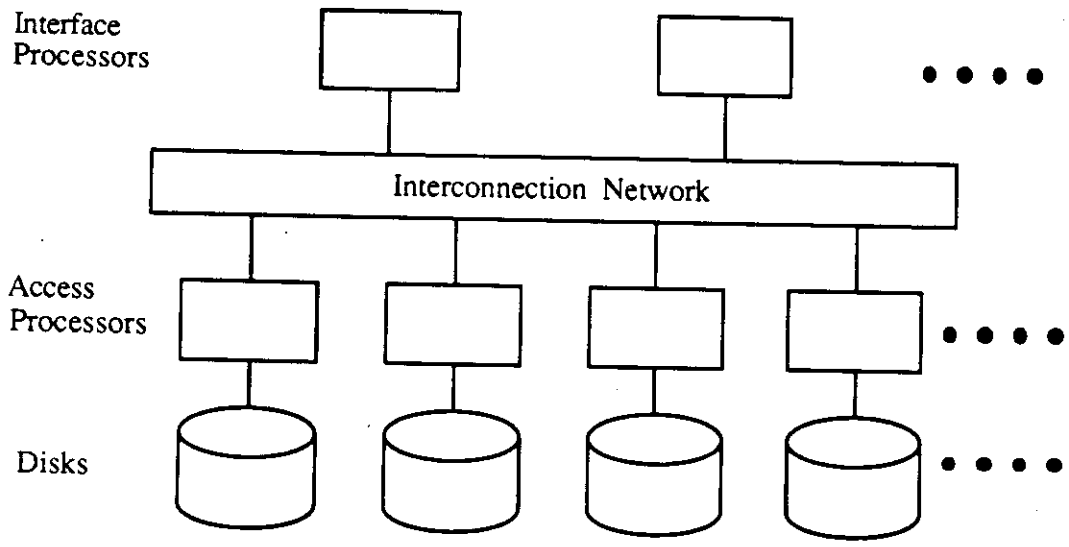

Figure 2. Parallel Processor Architecture

The join operation in this paper is meant to be the equijoin exclusively. To achieve high performance on a parallel processor architecture, the join operation must be performed in parallel with each Access Processor working on a portion of the join. Since the tuples of each relation have originally been distributed based on the primary key, the tuples of the join relations have to be distributed again if the join attribute is not the primary key. To make this distribution on join attribute distinct from the original distribution on primary key, we refer to this operation as *tuple redistribution*. Usually a hashing scheme is applied to the join attribute for attaining a more uniform distribution of processor workload. Because the same hashing function is used with both

join relations, the tuples after redistribution only have to be joined with those tuples on the same processor; hence making each processor perform a join on its disjoint subset in parallel. The results from each processor then are merged in the final processing step.

It has been shown that, with decreasing main memory costs, the hybrid hash join algorithm is the preferred strategy for joining large relations [Dewi85]. The multiprocessor version of hybrid hash join algorithm, a straightforward generalization from the single processor version, has been reported in [Dewi85] and [Laks89]. We outline this algorithm in the rest of this section.

Let R and S be the join relations, R_i and S_i be the subset of relation R and S respectively initially residing on access processor i. There are N access processors in the multiprocessor configuration. The multiprocessor hybrid hash join algorithm consists of two phases. In the first phase, also known as the redistribution phase, the tuples are redistributed among the access processors based on the hash value of the join attribute. In case the join attribute of relation R (S) happens to be the primary key, then there is no need to redistribute the tuples of that relation. In the second phase, also referred to as the join phase, each access processor executes in parallel the join operation on its disjoint subset. To distinguish this algorithm from a modified version in a later section, we term the algorithm shown in Figure 3 the *conventional* multiprocessor hash join algorithm. We want to point out that, in an efficient implementation, the two phases described in Figure 3 can be pipelined, i.e. redistribution phase does not have to finish before join phase starts. For clarity, we refer to the processing as though the phases are sequentially executed. This does not affect the analysis or conclusions.

4. PERFORMANCE MODEL

Various parameters affect the performance of the join operation. In our analysis, we use the following notation and parameters as defined in [Laks89]:

Notation

DB Related

R , S	The join relations.
R_i , S_i	Portion of R and S in processor i at the beginning of phase 1.
R_i' , S_i'	Portion of R and S in processor i at the beginning of phase 2, i.e. after tuple redistribution.

Phase 1 (Redistribution Phase) -- A predefined hash function $H1$ and the same number of hash buckets, N, are used in each access processor. Each access processor i, where $1 \leq i \leq N$, performs the following two steps:

1. Processor i allocates one output buffer for each of the N hash buckets. Then it reads its subset R_i of relation R from disk and hashes the join attribute of each tuple, which is moved to one of the output buffers based on the hash value. When an output buffer is full, it is transmitted through the interconnection network to the appropriate processor.

2. After relation R has been redistributed, processor i uses the same procedure to distribute its subset S_i of relation S.

At the end of phase 1, tuples belonging to the same hash bucket have been sent to the same processor. Let R_i' and S_i' be the subset of relation R and S respectively on processor i after tuple redistribution.

Phase 2 (Join Phase) -- The processors do not communicate with each other in this phase. On each processor i, we further partition its subrelation R_i' using a hash function $H2$. There are $(B + 1)$ hash buckets for the corresponding partitions R_{i0}', R_{i1}',.....,R_{iB}'. An output buffer is allocated for each of B partitions, R_{i1}',.....,R_{iB}'. The number of $(B + 1)$ partitions is chosen such that the hash table for each partition can individually fit in the available memory and the hash table of partition R_{i0}' can fit in memory in the presence of B output buffers for other partitions. Each processor i performs the following steps:

3. Subrelation R_i' is read and hash function $H2$ is applied to the join attribute of each tuple. If the tuple belongs to bucket R_{i0}', then it is placed into the the hash table in memory; otherwise it is moved to the corresponding output buffer. Each output buffer is individually flushed to disk when it is full. At the end, the hash table for R_{i0}' is in memory and the other partitions reside on disk.

4. Subrelation S_i' is read and similar procedure is applied to it. Tuples of S_i' that hash into bucket S_{i0}' are immediately used to probe the hash table for matches. At the end, tuples belonging to R_{i0}' and S_{i0}' have been joined; and partitions R_{i1}',.....,R_{iB}' and S_{i1}',.....,S_{iB}' reside on disk.

5. Now each of the remaining B partitions is joined one by one. An R_{ij}' partition is read from disk and its hash table is built in the memory. The corresponding S_{ij}' partition is read from disk and its tuples are hashed to probe against the hash table for join.

Figure 3. Conventional Multiprocessor Hybrid Hash Join Algorithm

$\|R\|$, $\|S\|$	Number of blocks in R and S.
$\{R\}$, $\{S\}$	Number of tuples in R and S.
$<R>$, $<S>$	Number of bytes in R and S.
Q	Measure of aggregate skew on the join attribute. Q equals the fraction of $R(S)$'s tuples which hash to one particular bucket.

Configuration Related

N	Number of Access Processors.

μ	MIPS ratings of the Access Processor.
M	Available memory space in a single processor.
β	Effective bandwidth of interconnection network.
f_i	Fraction of R_i' whose hash table can fit in memory M in the presence of output buffers for other partitions, that is, $f_i = R_{i0}'/R_i'$.

Processing Related

t_d	Disk IO access time.
I_p	Cpu pathlength for processing a tuple in any step of the two phases.
I_d	Cpu pathlength for initiating a disk IO.
I_c	Required pathlength for sending or receiving a message.

Assumptions

We assume we have perfect hash functions which can partition a set of distinct values equally into all the hash buckets. Suppose there is no data skew in the primary keys of either of the join relations. This is to say the tuples of both join relations are evenly distributed at the beginning of phase 1 since the perfect hash function can partition the relations uniformly across all the access processors. To simplify our analysis, we first consider the single skew case with skewed distribution in relation R's join attribute only. Further we assume there is only one value of the join attribute for which there are a significant number of tuples with this value. We call this value of the join attribute the *skew value*. Due to this skew value, the access processor corresponding to this value will receive the largest number of tuples during redistribution and will have to process the largest number of tuples in phase 2. We use parameter Q as the measure of aggregate skew on the access processor which receives the skew value. This means that one processor will process $Q\{R\}$ tuples and each of the other processors will process $(1-Q)\{R\}/(N-1)$ tuples. For relation S, each access processor will receive approximately $\{S\}/N$ tuples since there is no data skew in relation S's join attribute.

Response Time

We first investigate the response time in a single user environment. As mentioned earlier, both phases can be done in parallel among all the access processors. The overall response time of the join operation is determined by the processor which finishes last. We assume that all processors have the same processing power in terms of MIPS, main memory space, etc. Then, owing to the uneven tuple distribution in phase two, the processor receiving the tuples with the skew value has to handle the largest number of tuples and it will finish last; therefore determining the overall response time. There are three components in the response time $T0$: (1) cpu time $T0_{cpu}$ which can be further divided into three subcomponents: cpu time due to tuple processing $T0_{cpu-tuple}$,

cpu time for initiating disk IOs $T0_{cpu-initIO}$, cpu time due to communication protocol $T0_{cpu-comm}$; (2) physical IO access time $T0_{io}$; and (3) network delay due to tuple redistribution $T0_{net}$. Depending on how much overlap a system achieves, the join response time $T0$ can be either the sum of the three components for the worst case, or the maximum value of the three components for the best case, or somewhere in between. The detailed derivation appears in Appendix A; for brevity, only the final expression is given below:

$$T0 \ \varepsilon \ [\ max(T0_{cpu},\ T0_{io},\ T0_{net}),\ \ T0_{cpu} + T0_{io} + T0_{net}\ ]$$

where "ε" means $T0$ can be any value within the specified range, and

$$T0_{cpu} = T0_{cpu-tuple} + T0_{cpu-initIO} + T0_{cpu-comm}$$

$$T0_{cpu-tuple} = \left[\{R\} \left[\frac{1}{N} + (2-f_i)Q \right] + \frac{\{S\}}{N}(3-f_i) \right] \frac{I_p}{\mu}$$

$$T0_{cpu-initIO} = \left[|R| \left[\frac{1}{N} + (4-2f_i)Q \right] + \frac{|S|}{N}(5-2f_i) \right] \frac{I_d}{\mu}$$

$$T0_{cpu-comm} = \left[|R|(\frac{1}{N} + Q) + \frac{2|S|}{N} \right] \frac{(N-1)}{N} \ \frac{I_c}{\mu}$$

$$T0_{io} = \left[|R| \left[\frac{1}{N} + (4-2f_i)Q \right] + \frac{|S|}{N}(5-2f_i) \right] t_d$$

$$T0_{net} = \left[\frac{<R> + <S>}{\beta} \right] \frac{(N-1)}{N}$$

Using our response time formula, we can study the relationships among response time, configuration size, and degree of skew. In the following example, the multiprocessor system has 2 MIPS processors each with 4 megabytes of main memory, and the interconnection network has effective bandwidth at 12 megabytes per second. The processing related parameters are given the values: t_d = 20 milliseconds, I_p = 1000 instructions, I_d = 2000 instructions, I_c = 1000 instructions, The join relations R and S have 2 million tuples each. Moreover, each tuple is 200 bytes long; and there are 4K bytes per block. As illustrated in Figure 4, the join response time is plotted as a function of the number of access processors in the configuration. We observe that the response time initially decreases significantly as the system configuration increases. With 10% skew (Q = 0.10) in relation R's join attribute, the response time begins decreasing slowly when the number of processors N is around 60. Actually response time asymptotically approaches 1000 seconds no matter how many processors we employ in the system configuration. Without data skew, the

response time decreases as N increases and gradually approaches 100 seconds. This shows that, with data skew, the response time is highly restricted by the degree of skew. Further reduction in response time is impossible to achieve by increasing the number of processors.

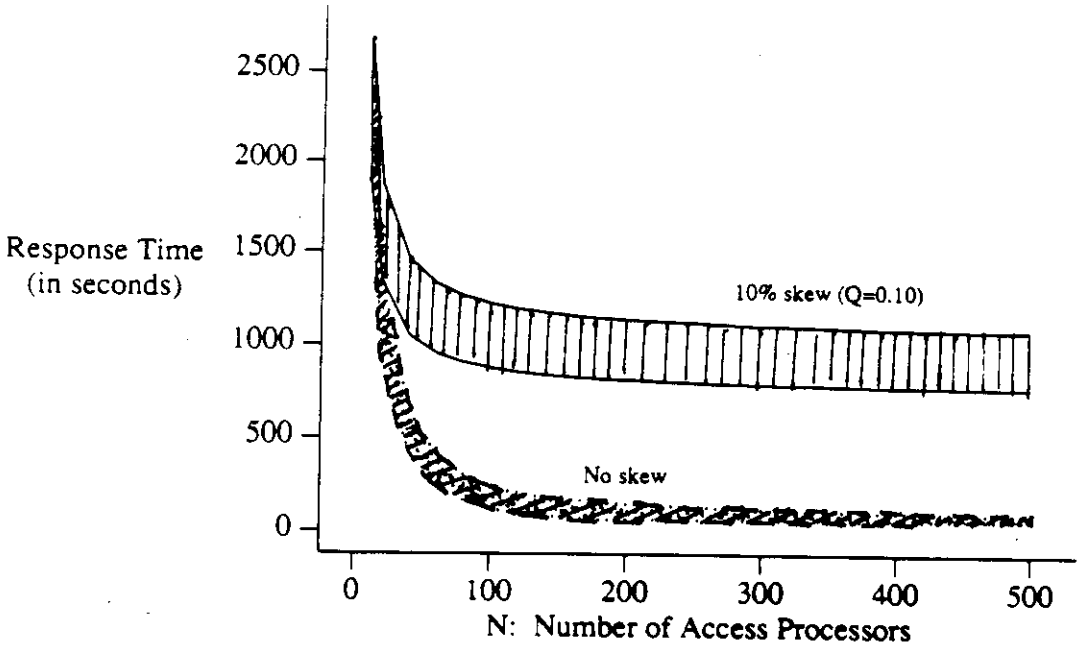

Figure 4. Join response time versus number of processors

The single user join response time versus the degree of skew on a system configuration of 100 access processors is shown in Figure 5. The join response time degrades (increases) in proportion to the degree of skew Q. When the degree of skew is very high, the actual response time may be worse since the hash table may not fit in the available memory and we have to handle overflow problems. Apparently, using the conventional multiprocessor hash join algorithm, the join performance of the parallel processor architecture is very sensitive and severely limited by the data skew phenomenon.

5. MODIFIED ALGORITHMS

Through tuple redistribution, data skew in the join attribute may lead to load skew and loss of parallelism when using the conventional multiprocessor hash join algorithm. In this section, we will discuss two algorithms which can minimize the impact of the skew effect.

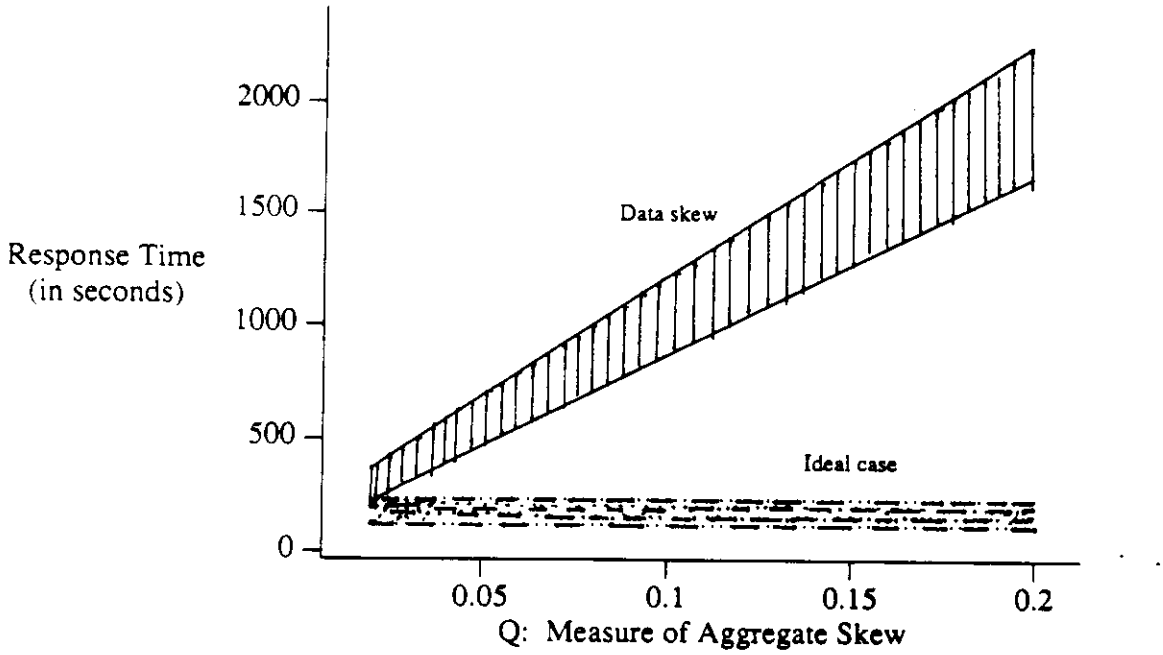

Figure 5. Join response time versus degree of skew ($N = 100$)

5.1. Tuple Duplication

In real database applications, a join operation frequently involves a small relation and a much larger relation. In this case, an alternative strategy is to make a full copy of the smaller relation on every access processor and not to redistribute the larger relation. When a tuple is redistributed to every processor, we refer to this action as *tuple duplication*. This strategy is similar to the fragment duplication strategy used in the distributed Ingres [Epst78] except that Ingres may duplicate a fragment of relation to a subset of the sites as opposed to duplication to every site. Although the tuple duplication strategy causes higher cpu and IO costs in processing the smaller relation, it avoids the expensive cpu and IO time incurred in redistributing the tuples of the larger relation. In addition, the tuple duplication strategy does not cause extra traffic in a broadcast-type interconnection network. The conditions under which this strategy is superior to the conventional algorithm are discussed in section 6. Assuming relation R is the smaller one of the two join relations, Figure 6 describes the modified multiprocessor hash join algorithm M1 using the tuple duplication strategy:

Phase 1 -- Each processor reads its subset, R_i, of relation R from disk. Without any computation, a block of tuples at a time is transmitted through the interconnection network and duplicated on every other processor.

At the end of phase 1, each processor i has a full copy of relation R and a portion S_i of relation S.

Phase 2 (Join Phase) -- The same steps as specified in the Conventional Multiprocessor Hybrid Hash Join algorithm can be applied here. The only difference is R and S_i, instead of R_i' and S_i', are joined.

Figure 6. Modified Multiprocessor Hash Join Algorithm M1

It should be pointed out that algorithm M1 using tuple duplication completely removes the skew effect no matter which relation (either R, or S, or both) has data skew. This is because each processor handles the same number of tuples, $\{R\} + (\{S\}/N)$, in phase 2. Suppose there is no stochastic phenomenon among the processors, each processor yields the same processing time. The join response time $T1$ using algorithm M1 can be obtained as:

$$T1 \ \varepsilon \ [\ max(T1_{cpu}, T1_{io}, T1_{net}),\ T1_{cpu} + T1_{io} + T1_{net}\ ]$$

where

$$T1_{cpu} = T1_{cpu-tuple} + T1_{cpu-initIO} + T1_{cpu-comm}$$

$$T1_{cpu-tuple} = \left[\{R\}(2-f_i) + \frac{\{S\}}{N}(2-f_i) \right] \frac{I_p}{\mu}$$

$$T1_{cpu-initIO} = \left[|R|(\frac{1}{N} + 4-2f_i) + \frac{|S|}{N}(3-2f_i) \right] \frac{I_d}{\mu}$$

$$T1_{cpu-comm} = (\frac{|R|}{N} + |R|)\ \frac{I_c}{\mu}$$

$$T1_{io} = \left[|R|(\frac{1}{N} + 4-2f_i) + \frac{|S|}{N}(3-2f_i) \right] t_d$$

$$T1_{net} = \frac{<R>}{\beta}$$

Due to the fact that a qualification predicate frequently occurs in a complex query, a good query strategy is to apply the predicate first to create a temporary relation, resulting in a much smaller relation before the join operation takes place. Also a relation may have a secondary index on the join attribute. A secondary index usually forms a relation just like the regular data relation [Tera87]. And the relatively small-sized secondary index relation, in many cases, can replace the data relation to take part in the join operation. For these reasons, the relations actually

participating in join operation are often of extreme sizes, hence we expect algorithm M1 to be more useful than one might expect at first glance.

5.2. Partial Duplication and Redistribution

As shown earlier, the Conventional Multiprocessor Hash Join algorithm is unable to deal with the data skew problem effectively. The key to solving this problem is to avoid load skew by dividing the work evenly across all the access processors. Instead of having only one processor receive the skew value, we can spread the tuples with the skew value to a group of designated processors. All the access processors are classified into two groups: the processors designated to handle the skew value are termed *reserved processors* and the rest of the processors are simply called *non-reserved processors*. For R tuples hashing to the same bucket as the skew value, we randomly select one of the reserved processors. For S tuples hashing to the skew value's bucket, they will match the corresponding R tuples which have been spread across the reserved processors. Therefore we have to duplicate this subset of S tuples to every reserved processor in order not to miss any valid join result. Because a subset of S tuples is duplicated on a subset of access processors, we refer to this action as *partial duplication*. Appendix C shows how to determine P, the number of reserved processors. We outline the modified multiprocessor hash join algorithm M2 using the partial duplication strategy in Figure 7:

Designate P access processors as reserved processors, and the other $(N-P)$ as non-reserved processors. And then broadcast this grouping to every access processor.

Phase 1 -- Tuple redistribution is applied to relation R with data skew and partial duplication is applied to relation S.

1. Each processor reads its subset R_i of relation R from disk. Using the predefined hash function $H1$, if a tuple hashes to the same bucket as the skew value, then a second hash based on the tuple ID or a simple round robin method is applied to decide which reserved processor it should be redistributed to; else it is redistributed to one of the non-reserved processors based on the hash value of a new hash function $H3$ on the join attribute.

2. Each processor works on its subset S_i of relation S. If a tuple hashes, using $H1$, to the same bucket as the skew value, then it is duplicated to every reserved processors; else it is redistributed to one of the non-reserved processors using hash function $H3$.

Phase 2 (Join Phase) -- same steps as specified in the Conventional Multiprocessor Hybrid Hash Join algorithm.

Figure 7. Modified Multiprocessor Hash Join Algorithm M2

In order to illustrate the difference between the conventional and M2 algorithms, we use an example as shown in Figure 8. Initially, both relations are evenly distributed across 4 access processors at the beginning of phase 1. The join attributes are the second attribute in both relations. A data skew with skew value "a" is present in relation R's join attribute. Using the conventional algorithm, half of relation R's tuples (6 out of 12) are redistributed to processor 1, causing a bad load skew and impairing the join performance. In this example, the degree of aggregate skew Q equals 0.50. If we use the balancing join output method in Appendix C, the M2 algorithm would designate P=2 ($P = Q*N = 0.5*4$) processors as the reserved processors to handle the skew problem. For R tuples hashing to the first bucket, we use a round robin method to spread these tuples among the reserved processors, which are processor 1 and processor 2. The other R tuples are redistributed, using hash function $H3$, among the non-reserved processors 3 and 4. For S tuples hashing to the first bucket, the partial duplication strategy is applied, i.e. tuples (s1,a) and (s3,b) are duplicated to both processors 1 and 2. The other S tuples are redistributed among processors 3 and 4. Note that the load on the bottleneck processor (processor 1) in the conventional algorithm is greatly decreased in the M2 algorithm.

Initially both relations are evenly distributed.

processor 1	processor 2	processor 3	processor 4
(r1, a)	(r4, a)	(r7, a)	(r10, a)
(r2, c)	(r5, b)	(r8, a)	(r11, f)
(r3, d)	(r6, e)	(r9, g)	(r12, h)
(s1, a)	(s3, b)	(s5, c)	(s7, d)
(s2, e)	(s4, f)	(s6, g)	(s8, h)

Tuple redistribution using the conventional algorithm.

processor 1	processor 2	processor 3	processor 4
(r1, a)	(r2, c)	(r6, e)	(r9, g)
(r4, a)	(r3, d)	(r11, f)	(r12, h)
(r5, b)			
(r7, a)			
(r8, a)			
(r10, a)			
(s1, a)	(s5, c)	(s2, e)	(s6, g)
(s3, b)	(s7, d)	(s4, f)	(s8, h)

Tuple redistribution using the M2 algorithm,
partial duplication is applied to relation S.

Group 1 (reserved proc.)		Group 2 (non-reserved proc.)	
processor 1	processor 2	processor 3	processor 4
(r1, a)	(r4, a)	(r2, c)	(r3, d)
(r5, b)	(r7, a)	(r6, e)	(r11, f)
(r8, a)	(r10, a)	(r9, g)	(r12, h)
(s1, a)	(s1, a)	(s5, c)	(s7, d)
(s3, b)	(s3, b)	(s2, e)	(s4, f)
		(s6, g)	(s8, h)

Figure 8. Tuples redistributed using Conventional and M2

In terms of workload in phase 2, each of the P reserved processors has to join $Q\{R\}/P$ tuples of relation R and $\{S\}/N$ tuples of relation S, while each of the $(N-P)$ non-reserved processors has to work on $(1-Q)\{R\}/(N-P)$ tuples of relation R and $(1-\frac{1}{N})\{S\}/(N-P)$ tuples of relation S. Exploiting the same procedure in deriving the response time formula $T0$, we obtain the join response time $T2$ for algorithm M2:

$$T2 \ \varepsilon \ [\ max(T2_{cpu}, T2_{io}, T2_{net}),\ T2_{cpu} + T2_{io} + T2_{net}\ ]$$

where

$$T2_{cpu} = T2_{cpu-tuple} + T2_{cpu-initIO} + T2_{cpu-comm}$$

$$T2_{cpu-tuple} = \left[\{R\} \left[\frac{1}{N} + (2-f_i)\frac{Q}{P} \right] + \frac{\{S\}}{N}(3-f_i) \right] \frac{I_p}{\mu}$$

$$T2_{cpu-initIO} = \left[|R| \left[\frac{1}{N} + (4-2f_i)\frac{Q}{P} \right] + \frac{|S|}{N}(5-2f_i) \right] \frac{I_d}{\mu}$$

$$T2_{cpu-comm} = \left[|R| (\frac{1}{N} + \frac{Q}{P}) + \frac{2|S|}{N} \right] \frac{I_c}{\mu}$$

$$T2_{io} = \left[|R| \left[\frac{1}{N} + (4-2f_i)\frac{Q}{P} \right] + \frac{|S|}{N}(5-2f_i) \right] t_d$$

$$T2_{net} = \frac{<R> + <S>}{\beta}$$

There is no question that algorithm M2 is more complex than the conventional algorithm. To execute the M2 algorithm, the multiprocessor system must be able to dynamically allocate the reserved processors to handle data skew. It also introduces some overhead in broadcasting the

selected grouping of the processors and for synchronization. On the other hand, algorithm M2 can significantly reduce the load on the bottleneck processor and decrease the overall join response time. We use the same example illustrated in Figure 4 and draw the response time curves of both the conventional and M2 algorithms in Figure 9.

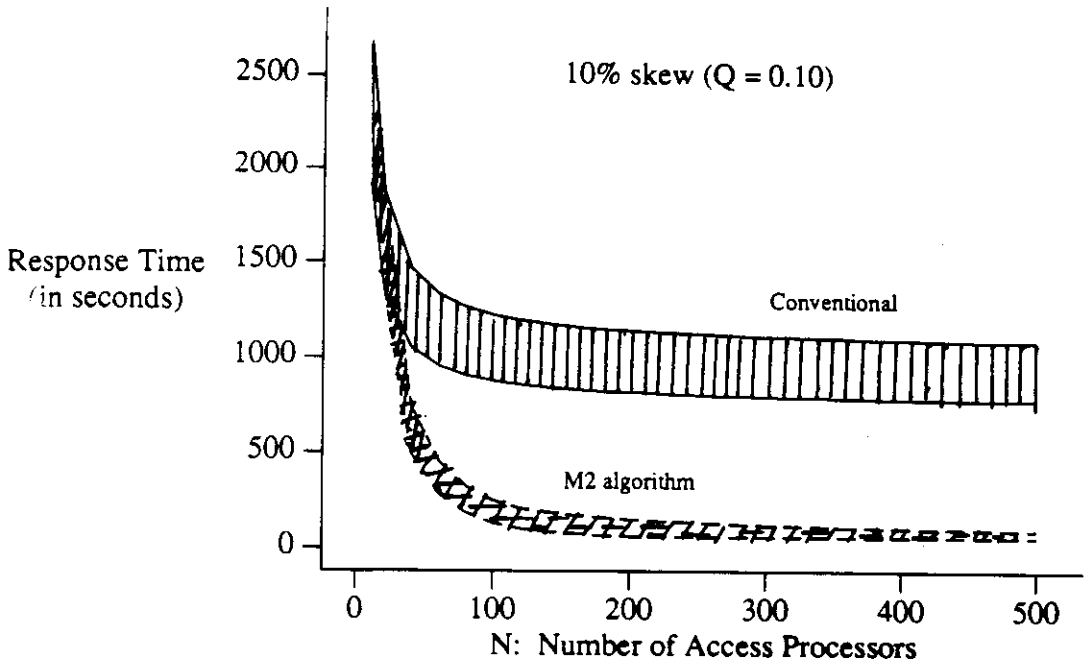

Figure 9. Comparison of response times of Conventional and M2

6. IMPLEMENTATION ISSUES

We introduced two modified multiprocessor hash join algorithms M1 and M2 in the previous section. Together with the conventional algorithm, there are three algorithms in performing the multiprocessor join operation. When a system decides which algorithm to use, a straightforward way is to use the response time formula to compute the timings for $T0$, $T1$, and $T2$, and then pick up the algorithm with the fastest join response time. To illustrate the domain each algorithm yields best performance, we now consider a simplified case in which the disk IO access time dominates the join response time.[1] Suppose, in the single skew case, the smaller relation R has data skew

[1] In the past decade, there is a clear trend that both cpu processing power and communication bandwidth have improved much faster than disk IO speed. Actually, with the parameter values specified in section 4, the IO access time is the dominant component in the response time.

problem, Figure 10 shows the domains in which the conventional, M1, and M2 algorithms can produce the best performance. As illustrated in the figure, the boundary lines, derived in Appendix B, are dependent on the number of access processors N in the system configuration. The larger configuration a system has, the smaller the domain in which the conventional algorithm can be useful. Obviously, in a parallel architecture with a large number of processors, the modified algorithm, either M1 or M2, can be employed in a very large domain of real world database applications in which data skew frequently occurs.

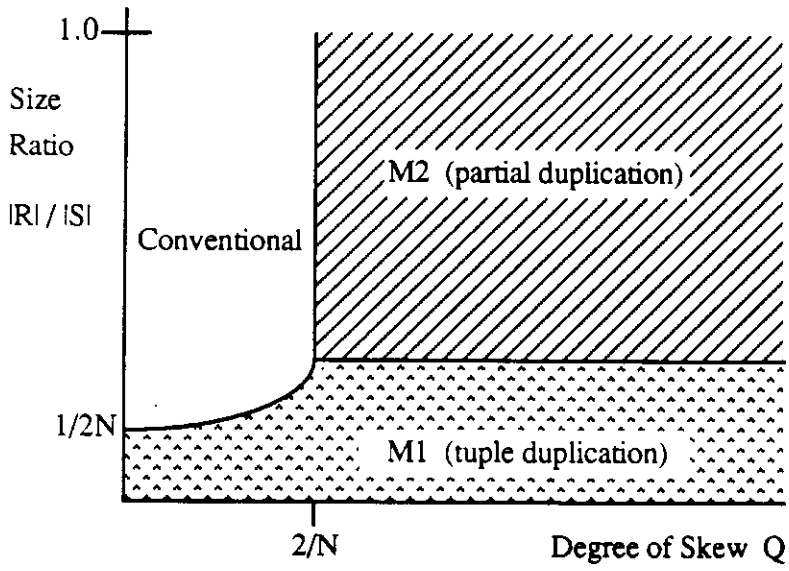

Figure 10. The domain in which the algorithms are the best

Many of today's relational data base management systems allow a user to control the computation of statistics for query optimization. An example is the COLLECT STATISTICS statement in Teradata's DBC/1012 [Tera87]. The frequency distributions of the attribute values are typically collected in this kind of statement. In our analysis, we use the aggregate distribution of the attribute after tuple redistribution to measure the degree of skew Q. This aggregate distribution on each processor can be obtained by expanding the function of the statistics collection statement. A new option can be added to instruct the statistics statement to use the tuple redistribution's hash function and find out the distribution in the hash buckets. Although collecting the aggregate distribution could be a time-consuming process, it is performed periodically as deemed necessary. Therefore, it should cause no impact on the join response time and query compilation time.

Also, in real database applications, usually there are more than one skew values, resulting in more

than one processor being heavily loaded after tuple redistribution. In the event algorithm M1 is the best performer and is used in the join operation, exactly the same M1 algorithm will remove the effect of other skew values since tuple duplication can make each processor handle equal number of tuples in the join phase, namely a full copy of the smaller relation and an un-redistributed equal portion of the larger relation as discussed in section 5.1. In the event algorithm M2 is used, we have to find all the aggregate skews $Q_1, Q_2, ..., Q_m$, which are greater than the the boundary value ($2/N$ as shown in Appendix B). Then all the access processors are classified into (m+1) groups, in which there are m groups of reserved processors and one group of non-reserved processors. Each group i, where $1 \leq i \leq m$, of the reserved processors is designated P_i access processors depending on its corresponding degree of aggregate skew Q_i. As illustrated in Appendix C, similar computations can be performed to obtain the P_i values.

As for the join operation involving the double skew case, the conventional algorithm has especially poor performance when the two join relations are skewed in the same direction since it has quadratic complexity [Laks89]. For this reason, the modified algorithms M1 and M2 are even more appealing in the double skew case. Again exactly the same M1 algorithm can be employed for the same reason previously described if M1 is the best performer for the query. In case the M2 algorithm is used, some detailed statistics information, such as the total size of the join result and the number of join tuples from the skew values, is needed in order to have effective groupings of the access processors. Although the double skew case is more complex than the single skew case, the same concept as outlined in the algorithm M2 can still be employed to effectively remove the skew effect and speed up the join operation.

7. CONCLUSIONS

In this paper, we analyzed the data skew phenomenon and its impact on join performance in a parallel processor architecture. The shared-nothing parallel architecture, with each relation horizontally fragmented, is expected to provide expandability along with linear performance improvement in data base application. However, using the conventional multiprocessor hash join algorithm, the data skew commonly present in the join attribute can lead to load skew and severely limits the linear speedup in join operation. The key to solving this problem is to avoid load skew by assigning the same amount of work to each processor. We introduced two modified multiprocessor hash join algorithms: namely the M1 algorithm using tuple duplication scheme

and the M2 algorithm using partial duplication scheme. These two algorithms can allocate an equal number of tuples for each processor to handle and effectively solve the skew problem. Further we identified the domains in which the conventional, M1, and M2 algorithms can give the best join performance.

Our research can lead to two interesting observations. First, although we assume that each relation is initially horizontally fragmented, our analysis does not demand this assumption and can be applied to an environment in which the relation is initially residing on a single processor. A good example is a set of Sun workstations interconnected through Ethernet with each single relation residing on a single workstation. It has been recently shown that the load sharing hash join algorithm, taking advantage of the cpu resources on the other idle workstations, can speed up the join operation [Wang88]. Clearly, the data skew phenomenon can lead to the same problem slowing down the join operation in this environment. Our modified algorithms can be similarly applied to eliminate the skew effect in the workstation environment.

The second observation is that the data skew problem may appear in other operations, e.g. the GROUP BY operation, on a multiprocessor system. For instance, suppose there is an Employee table with 3 attributes: Name, DepartmentNo, and Salary. And the tuples of the Employee table are initially distributed based on the primary key, the Name attribute. Then the SQL query

```
SELECT DepartmentNo, AVERAGE(Salary)
 FROM Employee
 GROUP BY DepartmentNo;
```

might be executed with one of two different strategies: (1) local group_by, followed by tuple redistribution, followed by global group_by; (2) tuple redistribution, followed by global group_by directly. Unaware of data skew phenomenon in the DepartmentNo attribute, the multiprocessor system may choose the second strategy, resulting in a bad load skew. We are currently working on the execution strategies for the GROUP BY operation on a multiprocessor system which takes data skew into account.

8. ACKNOWLEDGMENT

The authors would like to thank Jason Chen and Pekka Kostamaa of Teradata Corporation, and Cliff Leung, Tom Page, and Chung-Dak Shum of UCLA for interesting discussions and useful comments.

9. REFERENCE

[Bora88] Boral, H, "Parallelism in Bubba", Proc. International Symposium on Databases in Parallel and Distributed Systems, Austin, TX., 1988.

[Chri83] Christodoulakis, S., "Estimating Record Selectivities", Information systems, Volume 8, No. 2, 1983.

[Cope88] Copeland, G., Alexander, W., Boughter, E., and Keller, T., "Data Placement in Bubba", Proc. ACM SIGMOD conference, 1988.

[Dewi85] DeWitt, D.J., Gerber, R.H., "Multiprocessor Hashed-based Join Algorithms", Proc. 11th International Conference on Very Large Data Bases, 1985.

[Dewi88] DeWitt, D.J., Ghanderarizadeh, S., Schneider, D., "A Performance Analysis of the Gamma Database Machine", Proc. ACM SIGMOD Conference, 1988.

[Epst78] Epstein, R., Stonebraker, M., and Wong, E., "Distributed Query Processing in a Relational Database System", Proc. ACM SIGMOD conference, 1978.

[Laks88] Lakshmi, M.S. and Yu, P.S., "Effect of Skew on Join Performance in Parallel Architectures", Proc. International Symposium on Databases in Parallel and Distributed Systems, Austin, TX., 1988. (Also appears as IBM Research Report RC13370, T.J. Watson Research Center, Yorktown Heights, NY 10598)

[Laks89] Lakshmi, M.S. and Yu, P.S., "Limiting Factors of Join Performance on Parallel Processors", Proc. 5th International Conference on Data Engineering, 1989.

[Nech84] Neches, P., "Hardware Support for Advanced Data Management systems", IEEE Computer, Vol. 17, No. 11, 1984.

[Ston86] Stonebraker, M., "The Case for Shared Nothing", IEEE Database Engineering, Vol. 9, No. 1, 1986.

[Tera87] Teradata Corporation, "DBC/1012 Data Base Computer System Manual", Teradata document C10-0001-06, Los Angeles, CA., 1987.

[Tera88] Teradata Corporation, "DBC/1012 Data Base Computer Concepts and Facilities", Teradata document C02-0001-05, Los Angeles, CA., 1988.

[Wang88] Wang, X. and Luk, W.S., "Parallel Join Algorithms on a Network of Workstations", Proc. International Symposium on Databases in Parallel and Distributed Systems, Austin, TX., 1988.

Appendix A. Deriving Join Response Time $T0$

The join response time formula $T0$, using the conventional multiprocessor hybrid hash join algorithm, is derived here for the single user environment. This formula has been derived in [Laks88]. We assume processor i receives the largest number of tuples after tuple redistribution, hence determining the overall response time. As discussed in section 4, processor i has to work on $\{R\}/N + \{S\}/N$ tuples in phase 1 owing to even distribution at the beginning of phase 1. In phase 2, processor i has to join $Q\{R\}$ tuples of relation R and $\{S\}/N$ tuples of relation S since data skew occurs only in relation R's join attribute.

First we compute the number of disk reads/writes for each step as follows:

$$\text{step } 1 = |R_i| + |R_i'| = \frac{|R|}{N} + Q|R|$$

Note that $|R_i|$ is the number of disk reads and $|R_i'|$ is the number of disk writes required for redistributing relation R. And the other steps can be similarly computed.

$$\text{step } 2 = |S_i| + |S_i'| = \frac{2|S|}{N}$$

$$\text{step } 3 = |R_i'| + \sum_{j=1}^{B} |R_{ij}'| = |R_i'| + (1-f_i)|R_i'| = (2-f_i)Q|R|$$

$$\text{step } 4 = |S_i'| + \sum_{j=1}^{B} |S_{ij}'| = |S_i'| + (1-f_i)|S_i'| = (2-f_i)\frac{|S|}{N}$$

$$\text{step } 5 = \sum_{j=1}^{B} |R_{ij}'| + |S_{ij}'| = (1-f_i)\,(Q|R| + \frac{|S|}{N})$$

Summing up all the disk IOs, we have total number of disk IOs, Δ, as

$$\Delta = |R| \left[\frac{1}{N} + (4-2f_i)Q\right] + \frac{|S|}{N}(5-2f_i)$$

Then the physical IO access time is $T_{io} = \Delta * t_d$, and the cpu time for initiating disk IOs is $T_{cpu-initIO} = \Delta * I_d / \mu$.

Second, we derive the cpu time due to tuple processing, $T0_{cpu-tuple}$, as below:

$$phase\,1 = (\{R_i\} + \{S_i\})\,\frac{I_p}{\mu} = \left[\frac{\{R\}}{N} + \frac{\{S\}}{N}\right]\frac{I_p}{\mu}$$

and phase 2 consists of the following three steps,

$$step\,3 = \{R_i'\}\frac{I_p}{\mu} = Q\{R\}\frac{I_p}{\mu}$$

$$step\,4 = \{S_i'\}\frac{I_p}{\mu} = \frac{\{S\}}{N}\frac{I_p}{\mu}$$

$$step\,5 = \frac{I_p}{\mu}\sum_{j=1}^{B}(\{R_{ij}'\} + \{S_{ij}'\}) = \frac{I_p}{\mu}(\{R_i'\} - \{R_{i0}'\} + \{S_i'\} - \{S_{i0}'\})$$

$$= \frac{I_p}{\mu}(1-f_i)\,(\{R_i'\} + \{S_i'\} = (1-f_i)\,(Q\{R\} + \frac{\{S\}}{N})\frac{I_p}{\mu}$$

Hence,

$$TO_{cpu-tuple} = TO_{cpu-tuple}(phase\,1) + TO_{cpu-tuple}(phase\,2)$$

$$= \left(\{R\}\left[\frac{1}{N} + (2-f_i)Q\right] + \frac{\{S\}}{N}(3-f_i)\right)\frac{I_p}{\mu}$$

Third, we give the cpu time, $TO_{cpu-comm}$, due to communication protocol. We assume that the interconnection network is a broadcasting network and it has hardware logic to filter out the unsolicited messages for the processor. Further, a message buffer is assumed to have the same size as a disk block. Then, processor i has to send $|R_i|(N-1)/N$ messages and receive $|R_i'|(N-1)/N$ messages, and similar for relation S.

$$TO_{cpu-comm} = (\,|R_i|+|R_i'|+|S_i|+|S_i'|\,)\,\frac{(N-1)}{N}\,\frac{I_c}{\mu}$$

$$= \left(|R|(\frac{1}{N} + Q) + \frac{2|S|}{N}\right)\frac{(N-1)}{N}\,\frac{I_c}{\mu}$$

Last, we derive the network delay due to tuple redistribution. As discussed above, each processor has to transmit $(N-1)/N$ fraction of tuples to other processors. Thus,

$$TO_{net} = \left(\frac{<R> + <S>}{\beta}\right)\frac{(N-1)}{N}$$

Appendix B. Deriving The Boundary Conditions

In this appendix, we derive the boundary conditions among the conventional, M1, and M2 algorithms for the case in which the IO time dominates the join response time:

Boundary Condition Between Conventional and M1 --

We can set up the boundary condition as

$T1_{io} \leq T0_{io}$ which can be simplified to:

$$\left[|R|(\frac{1}{N} + 4-2f_i) + \frac{|S|}{N}(3-2f_i) \right] \leq \left[|R| \left[\frac{1}{N} + (4-2f_i)Q \right] + \frac{|S|}{N}(5-2f_i) \right]$$

$$|R|(1-Q)(4-2f_i) \leq \frac{2|S|}{N}$$

Taking the most restricting case $f_i \approx 0$, then the boundary condition is:

$$\frac{|R|}{|S|} \leq \frac{1}{2N(1-Q)}$$

Boundary Condition Between M1 and M2 --

$T1_{io} \leq T2_{io}$ which can be simplified to:

$$\left[|R|(\frac{1}{N} + 4-2f_i) + \frac{|S|}{N}(3-2f_i) \right] \leq \left[|R| \left[\frac{1}{N} + (4-2f_i)\frac{Q}{P} \right] + \frac{|S|}{N}(5-2f_i) \right]$$

$$|R|(1 - \frac{Q}{P})(4-2f_i) \leq \frac{2|S|}{N}$$

Taking the most restricting case $f_i \approx 0$ and the P value ($P=QN$) from the balancing join output method in Appendix C, then the boundary condition is:

$$\frac{|R|}{|S|} \leq \frac{1}{2(N-1)}$$

Boundary Condition Between Conventional and M2 --

If we look at the response time formulas alone, the M2 algorithm can outperform the conventional algorithm as long as there is a data skew. Since the M2 algorithm is the same as the conventional algorithm when the number of reserved processors is equal to 1. Hence, we can set the boundary condition as $P \geq 2$. Using the value $P=QN$ from the balancing join output method in Appendix C, then the boundary condition is:

$$Q \geq \frac{2}{N}$$

Appendix C. Determining The Number of Reserved Processors

In this appendix, we show how to determine P, the number of reserved processors, while using the M2 algorithm. There are two methods in assigning the same amount of work to each processor. If the join operation produces few tuples, we may want to have the same number of tuples on each processor in phase 2 immediately before join processing. We call this method as *balancing join input*. If the join operation produces many tuples or the join operation is just an intermediate step, i.e. the output of join operation will be used as input of other operations in the query

processing, then we want each processor to generate the same number of tuples as a result of the join operation. The second method is called *balancing join output.* We now determine the value P for these two methods separately:

Balancing Join Input --

As discussed in section 5.2, each reserved processor has to join $Q\{R\}/P$ tuples of relation R and $\{S\}/N$ tuples of relation S in phase 2, while each non-reserved processor has to work on $(1-Q)\{R\}/(N-P)$ tuples of relation R and $(1-\frac{1}{N})\{S\}/(N-P)$ tuples of relation S. To balance join input between the reserved and the non-reserved processors, we have

$$\frac{Q\{R\}}{P} + \frac{\{S\}}{N} = \frac{(1-Q)\{R\}}{(N-P)} + \frac{\{S\}(1-\frac{1}{N})}{(N-P)}$$

Solving this equation, we obtain

$$P = \left[1 - N\frac{\{R\}}{\{S\}} + \sqrt{(N\{R\}/\{S\} - 1)^2 + 4QN^2\{R\}/\{S\}}\right] / 2$$

Balancing Join Output --

Suppose there are D distinct values in the join attribute. To balance join output between the reserved and the non-reserved processors, we have

$$\frac{\frac{Q\{R\}}{P}\ \frac{\{S\}}{N}}{\frac{D}{N}} = \frac{\frac{(1-Q)\{R\}}{(N-P)}\ \frac{\{S\}(1-\frac{1}{N})}{(N-P)}}{\frac{D(1-\frac{1}{N})}{(N-P)}}$$

Solving this equation, we obtain $P = QN$