Sequenced vs. Pipelined Parallel Multiple Joins in Paradata [1]

Liping Zhu, Arthur M. Keller[2] and Gio Wiederhold

Abstract

In this report we analyze and compare hash-join based parallel multi-join algorithms for sequenced and pipelined processing. The BBN Butterfly machine serves as the host for the performance analysis. The sequenced algorithm handles the multiple join operations in a conventional sequenced manner, except that it distributes the work load of each operation among all processors. The pipelined algorithms handle the different join operations in parallel, by dividing the processors into several groups, with the data flowing through these groups.

The detailed timing tests revealed the bus/memory contention that grows linearly with the number of processors. The existence of such a contention leads to an optimal region for the number of processors, given the join operands fixed. We present the analytical and experimental formulae for both algorithms, which incorporate this contention. We discuss the way of finding an optimal point, and give the heuristics for choosing the best processor's partition in pipelined processing.

The study shows that the pipelined algorithms produce the first joined result sooner than the sequenced algorithm and need less memory to store the intermediate result. The sequenced algorithm, on the other hand, takes less time to finish the whole join operations.

Key words: sequenced algorithm, pipelined algorithm, processor partition, intermediate buffer size, parallel computer, computation time, bus and memory contention.

[1]This work is supported by NSF grant IRI 88-13954 and BBN Advanced Computers.
[2]Dr. Keller's affiliation is Advanced Decision Systems and Stanford University

1 Introduction

The emergence of parallel computers in the late 1970s has been thought of as a breakthrough in solving the Von-Neuman bottleneck problem. Thus parallel computing tethniques have been developed over the past decade. Among various parallel computer applications, the most promising one is building parallel database systems. The two major pervasive problems in database systems that have bothered computer scientists, are slow transaction processing time and insufficient transaction throughput. The Paradata project at Stanford is investigating the first problem. The goals of this research are to develop database technology that will take advantage of medium grained parallelism, and to demonstrate a system on parallel computers using the algorithms we have developed.

1.1 Related Work

During the last ten years, much research has been done in two directions, the development of database machine and of new algorithms for parallel database operations. XRDB, a high-speed extended relational database engine, has been developed [?]; a hardware design for associative parallel join algorithm has been created [?]; and a query processor has been designed, which consists of four processing modules. Each module processes tuples of relations in a bit-serial, tuple-parallel manner for each of the primitive database operations that comprise a complex relational query [?]. The database machine GRACE adopts a novel relational algebraic processing algorithm based on hash and sort [?]. In addition, many other references on hardware architecture can be found in [?].

Recently, for investigating new algorithms for database operations, a pipelined 2-way merge sort algorithm for sorting has been developed, since, for efficient execution of relational algebraic operations, it is often advantageous to sort the object relations by key attributes in advance, reducing the range of comparison and simplifying downstream processing [?]. Some algorithms for the parallel processing of relational operations have been presented and analyzed, incorporating I/O, CPU, and message costs [?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. Some works reveal that distributing a database for parallel processing is NP-hard [?]. Other research has addressed parallel lock management [?], and parallel concurrency control [?].

Although much work has been done on join algorithms, relatively little attention has been given to parallel multiple join operations, let alone the pipelined processing of these join operations, which are quite common in database queries. This observation stimulate us to do some work in this area and we believe our study will be instrumental in furthering research in parallel database systems.

1.2 Algorithms, Tests, and Results

The algorithms we used are based on hash join algorithms [?, ?, ?]. The sequenced algorithm divides the work loads for each join operation evenly among the processors, and handles the multiple joins sequentially. The pipelined algorithms, including the single pipelined and the double pipelined algorithms, divide the processors into several groups, each of which is responsible for a separate join operation and its related hash tables. Data is made to flow through these hash tables in a pipeline, so that several joins take place in parallel, and each join implements a parallel hash join.

We believe that the Butterfly is a good choice for the experimental vehicle because it has a hierarchical shared memory architecture that is popular for parallel machines and is commercially available.

The experiments were designed to test the performance of each of the join algorithms under several different conditions. We assumed that all multiple join operations can proceed in main memory. For the purpose of the study, we discounted the input/output time, which varies significantly from machine to machine. First, we examined how the computation time is related to the change in the size of join relations, the increase in the number of processors, and the change in the partitioning of processors with differnt algorithms. Next, we analyzed the data we got, investigated the optimal point, the best partition, the optimal join sequence, with regards to the overall computation time. Our goal was to show how the performance of each algorithm is affected as the number of processors increases and the inter-processor communication and bus/memory contention become significant. Finally, we investigated the memory usage and its effect on the performance of different algorithms.

Our results showed the following:

- Bus/memory contention affects multi-join operations
- There is an optimal point for overall computation time and the time to get the first result
- There is an optimal join sequence for overall computation time in sequenced algorithm
- There are empirical best partitions for pipelined processing
- Pipelined algorithms produce the first result sooner.
- Pipelined algorithms allows one of the relations to be arbitrarily large.
- Pipelined algorithms need less memory to store the intermediate results.

The remainder of this report is organized as follows:

2 Overview of the BBN Butterfly machine

In this section, we present a brief overview of the Butterfly parallel machine. For a complete description of Butterfly see [?].

2.1 Hardware Configuration

Butterfly is a MIMD parallel computer. Each processor, along with an associated memory module, is connected through a high performance network-interconnect system. Featuring an efficient shared-memory architecture, the Butterfly computer can offer a gigabyte of shared memory in configurations containing as many as 256 processors. Each processor node consists of MC68020 microprocessor with MC68881 floating point coprocessor, MC68851 paged memory management unit (PMMU), and 4 megabytes of semiconductor dynamic random access memory (DRAM) for local and remote memory accesses, and so on. Typical memory reference instructions that access local memory take about one microsecond. Those accessing remote memory take about five microseconds. The memory bandwidth is 102-Megabytes-per-second.

2.2 Software Overview

The operating system in GP1000 is Mach 1000, which supports the standard UNIX 4.3BSD functions, then extends these functions to encompass a multiprocessing environment. Mach 1000 features high-performance virtual memory, efficient interprocessor communication via shared memory, atomic operations for fast multiprocessor-application performance, the ability to dedicate processors to parallel applications, and so on.

This architecture of the Butterfly system provides a program execution environment where tasks can be distributed among processors with little regard to the physical location of data associated with the tasks. The Uniform System is a software development environment which is effective for applications containing a few frequent repeated tasks. it provides higher-level memory and processor management facilities. The goal of storage management is to keep all the memory modules in the machine equally busy, thereby preventing the slowdown that occurs when many processors attempt to access a single memory module. The goal of processor management is to keep all the processors equally busy, thereby preventing the inefficiency that occurs when some processors are overloaded while others sit idle without work to do. The memory management is based on two principles:

- Use of a single large address space shared by all processes to simplify programming.
- Scattering application data uniformly across all memories of the machine to reduce possible memory contention.

The processor management requires identification of the parallel structure inherent in a chosen algorithm, and control of the processors to achieve the determined parallelism.

3 Parallel Multi-join Algorithms

We designed three parallel versions of multi-join algorithms: sequenced, single pipelined, and double pipelined for double-join queries. The common feature of the parallel versions of each of these algorithms is that there are two phases, namely, hash phase and join phase, in all three algorithms. Before the hash phase, the input relations are distributed in a round-robin manner among multiple processors' memories. During the hash phase, in parallel, each processor takes the data in its memory and hashes it, by using the same hash function, into the output buckets. The output buckets are distributed through the multiple processors' memories also. During the join phase, each processor can handle the join operation with the buckets that have already been filled by other processors during the hash phase. The first join needs to reapply the hash function to its result and redistribute the join results among other processors, to prepare for the next join operation.

The actual hash and join computation depend on the algorithm: in all three algorithms, hash tables for all join operands are built; in the sequenced algorithm, the two joins are done successively; in the single pipelined algorithm, the two joins are done in parallel, in the sense that the second join starts right after there are joined results from the first join; in the double pipelined algorithm, building the hash table for the largest relation is done in parallel with the two joins, which means the first join can start as soon as there are hashed tuples available for the largest relation. More details on each algorithm are presented in the following sections.

3.1 Sequenced algorithm

Our parallel version of the sequenced multi-join algorithm is a straightforward adaptation of the traditional single processor version of the algorithm. It consists of three parts: relation distribution, hash table construction, and join operation. The relations participating in the multi-join will be first distributed across the memory of the Butterfly processors. Then the actual hash process will take place followed by the join process. Diagram 1 depicts the sequenced processing of the multi-join operations.

From Diagram 1, we can see that in sequenced algorithm, each hash and join operation is handled by all processors one by one. From the three join operands, relations 1, 2, and 3, we build up corresponding hash tables 1, 2, and 3, in steps 1, 2, and 4. The first join operation can start as step 3 after step 1 and 2, and the results are rehashed to build up the hash table $1 * 2$. In step 5, the second join starts with the data from both hash table $1 * 2$ and hash table 3. The join results can be further hashed, if there are successive join operations in the query.

3.2 Single pipelined algorithm

In the single pipelined algorithm, the hash phase is exactly the same as that in the sequenced algorithm. In the join phase, however, we handle the two joins in parallel. After the hash phase, we divide the whole processors into two groups, with each group works on one join. By starting the two joins simultaneously, we can have the second join started whenever there is a result tuple from the first join operation. With the first join operation going on in the first group of processors, their results will flow continuely to the second group of processors. Diagram 2 depicts the single pipelined algorithm.

3.3 Double pipelined algorithm

In the double pipelined algorithm, we further improve the parallelization, by starting the join phase in parallel with part of the hash phase. Diagram 3 depicts this algorithm.

Diagram 1

Diagram 2

Diagram 3

From Diagram 3, we can see in the first step, we build up hash tables for relations 2 and 3 in parallel. After that, we start to build up hash table 1 for the relation 1, do the first and the second join, simultaneously. The whole processors are divided into three groups, with each group handles a specific operation, and with data flow through these groups.

4 The implementation of the algorithms

4.1 Data structures

When a relation is loaded or created on the Butterfly computer, it is distributed across all the shared memories in equal sized segments. Assuming the relation can be thought of as a continuous block of memory with concurrent access, we used the same method as

[?] to divide the relation up among the processors. Let p be the number of processors, f be the number of tuples in a relation, and b be the number of tuples to be worked on by each processor, then a good initial value for b would be

$$b = \lceil f/p \rceil.$$

This will allow every processor to work on at most $\lceil f/p \rceil$ tuples.

As discussed in [?], in order to minimize memory contention, we distribute the rows of the hash table across all of the memories and allow chains of buckets to run from processor to processor. The Butterfly has a mechanism for distributing rows of an array across all the memories of the machine. This mechanism distribute the $M \times N$ array by allocating the M rows to different processor's memory. Each row will be a vector of size N and can be accessed using standard C syntax (A[i][j]). This will reduce contention for each row by a factor of p, the number of processors. Also this will increase the maximum size of the file to be operated on by allowing the file to occupy all of the processors' memory. This scatter matrix is maintained by a vector of M pointers each of which points to the next row in the array. This vector is stored on one processor's memory thus causing a secondary place of contention, however, this contention can be removed by making a local copy of this vector at each processor that will be accessing the scatter matrix.

The five fields that make up each slot of the hash table are two pointers to the first and the last buckets (initialized to NULL), two integer fields to keep track of the positions for input and output in the chain and a lock field (all initialized to 0). The lock field will allow only one processor to manipulate a particular row at a time. Since each row of the hash table has its own lock, only those processors with tuples that have hashed to that row will have to wait. By choosing a large enough hash table, the number of requests for the same row will be kept to a minimum. Only the code involved with chains will be locked. This is kept to a minimum to prevent processor wait.

4.2 Implementation highlights

In the implementation of the three algorithms, we have paid more attention to the following issues:

• The code is flexible in dealing with different join operands' sizes, the number of processors available, and different partitions of these processors when using pipelined algorithms.

• There is a input file storing the processors' partition information. In the sequenced algorithm, the total number of processors participating in the query is stored. In the single pipelined algorithm, the partition between the first and the second join is stored. In the double pipelined algorithm, the partition between the first hash, the first and second join, the partition for the hash 2 and 3 are also stored. From the the information stored in the input file, the code can recognize which algorithm should be put to use.

• Hash tables are scattered among processors, and copies of all global variables and data structures' pointers exist in local memory of each processor to improve efficiency.

• We use the principle of the locality of reference to improve the access efficiency. In the implementation, each hashed tuple is put into the bucket located in certain processor's memory, where this tuple will be used later on by this processor. Allocating memories this way will guarantee that the input data needed by a process will resides in its local memory, the output will locate at remote memory, which is in fact, the local memory of processors that are going to use them as input.

• We release memory when it is no longer in use.

• In order to guarantee the atomicity of parallel operations, in case that different processors will access the same data structures at the same time, we use the synchronization facilities provided by the Uniform system. There are more synchronization overhead in pipelined algorithm than in the sequenced algorithm.

5 Analytical and experimental results

In this section, we present our analysis of the three proposed algorithms, based on the experimental conditions and assumptions. In our experiments, we chose that the join attributes are uniformly distributed over their domains, in all relations participating in the double join queries. In addition, the work load in both hashe and join phases is evenly distributed among processors. We present the relevant analytical work and the corresponding experimental results for the computation time, the time needed to get the first result, the optimal number of processors for a fixed job, and the best partition of processors in the pipelined algorithms. A comparison of the three algorithm, and some derivations from the analytical formulae are also discussed.

5.1 Analytical criteria and experimental design

The following notations are needed to evaluate the three proposed algorithms:

$F1$, $F2$, $F3$: names of the relations that participate in the join query.

$f1$, $f2$, $f3$: the number of tuples in the three relations $F1$, $F2$, $F3$ participating in the join query.

$e1$, $e2$, $e3$: the size of one tuple in relations $f1$, $f2$, $f3$, respectively.

$s1$, $s2$: the first and second join selectivity factors, which are defined by $s1 = card(F1 \bowtie F2)/(f1 \times f2)$ and $s2 = card(F1 \bowtie F2 \bowtie F3)/(s1 \times f1 \times f2 \times f3)$.

$t_{hashwait}$: average unit of waiting time caused by bus/memory contention in the hash phase.

$t_{pipehashwait}$: average unit of waiting time caused by the bus/memory contention in the hash phase of pipelined algorithm.

$t_{hashstore}$: average unit of time for hashing and storing one hashed tuple.

$t_{hashcommu}$: average unit of time for transferring one hashed tuple to a remote processor.

$t_{joinstore}$: average unit of time for comparing and storing one joined tuple.

$t_{joinwait}$: average unit of waiting time caused by the bus/memory contention in the first join.

$t_{joincommu}$: average unit of time for transferring one joined tuple to a remote processor.

p: the total number of processors that participate in the query processing.

Notice that we defined the $t_{joinwait}$ in the first join only. In our current algorithms, we consider only two successive joins, and there is no need for the second join to redistribute its results to other processors. This feature guaranteed that there will be no waiting time in the second join due to the communication with other processors. If multiple (N) joins are incorporated into the queries, then $t_{joinwait}$ will also apply to the first $N-1$ joins.

These parameters allow one to measure the CPU time, the bus/memory contention time, and the interprocessor communication time when executing a multiprocessor algorithm. The identified parameters depend on hardware capabilities.

The CPU time is used to perform several basic operations, namely, computing a hashing function, comparing two tuples, and storing hashed or joined tuples locally or remotely. Under our test conditions, because fewer tuples are required to be transferred to remote processors after hashing in the first join than in the hash phase, we use a different $t_{hashstore}$ and $t_{joinstore}$ to denote the time to hash and store one tuple in the hash phase, and the time to compare and store one tuple in the join phase.

The bus/memory contention is caused by the existence of "hot spots," a memory module that several processors want to access at the same time. This contention causes the access request to wait or retransmit. In our analysis, we chose a simple contention model specific to Butterfly memory interconnection networks. If there are p processors, each of which has the same probability to access any other processors, then the average contention time can be expressed as $t * (p-1)$. The coefficient t is the average waiting time caused by the bus/memory contention, if there are two processors accessing one memory module at the same time.

The communication time we considered is the "pure" time it takes to transfer data between processors, if there is no bus/memory contention. In our algorithm, the data communication occurs at the hash phase and the first join, where tuples have to be hashed and put into corresponding local or remote memories.

The benchmark relations are based on the standard Wisconsin Benchmark [?]. Each tuple in relation $F1$, $F2$, and $F3$ consists of ten, eight, and eight 4-byte integer values, and three, one, and one 52-byte string attributes, respectively. The tuple size is 190 in $F1$, and 84 in both $F2$, and $F3$. The typical tests are joins among 1000 tuple relation (approximately 200 Kbyte in size), with two 100 tuple relations (each of which approximately 8 Kbyte in size); 10,000 tuple relation (about 2 megabyte in size), with two 100 tuple relations; and 10000 tuple relation, with two 1000 tuple relations. The tests will produce 1000, 10,000, and 100,000 tuples after the first join, and 1000, 10,000, and 1,000,000 tuple relations after the second join, respectively. The largest of these relations which needs being stored in the memory, with 100,000 tuples, uses 20 megabytes, while the total memory available on the Butterfly is 320 megabytes. Because there is only 4 megabytes per processor, and we don't want to include the input/output in our tests, we could not conduct experiments on larger databases.

5.2 Formulae for overall computation time

As discussed above, we formulate the computation time for the sequenced operation as follows:

$$T_{sequ} = T_{hashes} + T_{joins}, \tag{1}$$

where

$$\begin{aligned} T_{hashes} &= T_{hash1} + T_{hash2} + T_{hash3} \\ &= (f1 + f2 + f3) * (t_{hashwait} * (p-1) \\ &\quad + t_{hashstore}/p + t_{hashcommu}), \end{aligned} \tag{2}$$

$$T_{joins} = T_{join1} + T_{join2}, \tag{3}$$

$$\begin{aligned} T_{join1} &= t_{joinstore} * f1 * f2/p + t_{joinwait} * s1 * f1 * f2 * (p-1) \\ &\quad + t_{joincommu} * s1 * f1 * f2, \end{aligned} \tag{4}$$

$$\begin{aligned} T_{join2} &= t_{joinstore} * s1 * f1 * f2 * f3/p \\ &\quad + t_{joincommu} * s1 * s2 * f1 * f2 * f3. \end{aligned} \tag{5}$$

Since in the sequenced algorithm, both hash and join phase are processed serially, with one hash or join operation at a time, the overall computation time is the addition of the time needed in each phase (Equation 1). In the hash phase, the hash operation on each tuple can be done in parallel on each processors. Once the hashed tuple has been sent to the corresponding memory, the storing work can be done in parallel with other processors. This portion of time is illustrated by term $t_{hashstore}/p$ in Equation 2. The term $t_{hashwait} * (p-1)$ in Equation 2 is corresponding to the bus/memory contention cost during the hashing phase. We came up with this simple contention model by the following considerations:

• If there is only one processor, there will be no bus/memory contention whatsoever.

• From the test results, we see clearly that when the number of processors participating in the whole computation is large enough, the overal computation time reveals a linear increase with the number of processors.

• We can assume that if only two processors access one memory module at the same time, one processor must wait $t_{hashwait}$ time for another processor. If there are p processors access one memory module at the same time, the longest waiting time on one processor will be $t_{hashwait}*(p-1)$. Considering all the processors, we can choose the average waiting time, which is expressed as $t_{hashwait} * (p-1)/2$. We then incorporate the denominator 2 into $t_{hashwait}$.

Based on these considerations, we give the term $t_{hashwait} * (p-1)$ in Equation 2. Notice that this contention model is a simplified one, under the current test conditions. The last term in Equation 2 is $t_{hashcommu}$, the "pure" communication time needed to transfer one tuple to remote processor.

With all parallel computation time, serial waiting time, and communication time being considered, and the total number of tuples in the three join operands is $f1+f2+f3$, finally we get the Equation 2, the time used in the hash phase in the sequenced algorithm.

In the join phase, since multiple joins will be handled one after another, we have Equation 3 for the time to finish two joins. In our algorithm, we do the joins locally with the data that are already put into the local memory from other processors in the hash phase. In addition, in the first join, after joining two matching tuples, we rehash it and redistribute it across the whole processors, in order to do the second join. This will cause the bus/memory contention, just as that in the hash phase. The term $t_{joinstore} * f1 * f2/p$ in Equation 4 represents the simple parallel computation in comparing and storing each matching tuple. The term $t_{joinwait} * s1 * f1 * f2 * (p-1)$ expresses the waiting time caused by redistributing those joined results from the first join $(s1 * f1 * f2)$ among all processors. The "pure" communication time needed is represented by $t_{joincommu} * s1 * f1 * f2$ in the third term in Equation 4. The summation of the three part results in Equation 4.

For the second join, the local join operation can be done in parallel with other processors, which is expressed by the term $t_{joinstore} * s1 * f1 * f2 * f3/p$ in Equation 5. For the constant term, although we use $t_{joincommu}$ as that in Equation 4, it represents the small amount of time needed for the overhead.

By investigating the multiple join operations, we are interested in the time to get the first result. The following formulae describe the timing for this. We express the elapsed time to get the first result with the sequenced algorithm as:

$$T1_{sequ} = T_{hashes} + T_{join1}. \tag{6}$$

In Equation 6, we see that the time to get the first result using the sequenced algorithm includes the time for hashing and the first join only. This is because with our tests conditions and assumptions, the first joined result will be produced instantly after finishing the first join and starting the second join.

Similarly, with the single and double pipelined algorithms, the elapsed time to get the first result:

$$\begin{aligned} T1_{singlepipe} &= T_{hashes} \\ &= (f1 + f2 + f3) * (t_{pipehashwait} * (p-1) \\ &\quad + t_{hashstore}/p + t_{hashcommu}). \end{aligned} \tag{7}$$

For the single pipelined algorithm, the hash phase is the same as that in the sequenced algorithm, in the sense that all three hashes are done one by one by all processors. The test results showed that it takes no time to produce the first join result when the first and the second join starts simultaneously. Therefore, the time to get the first join result will be approximately the time used in the hash phase.

$$\begin{aligned} T1_{doublepipe} &= max\{T_{hash1}, T_{hash2}\} \\ &= max\{f2 * (t_{pipehashwait} * (p1-1) + t_{hashstore}/p1 + t_{hashcommu}), \\ &\quad f3 * (t_{pipehashwait} * (p2-1) + t_{hashstore}/p2 + t_{hshcommu})\}, \end{aligned} \tag{8}$$

where $p1$, $p2$ is the partition of p that handle the hash phase over two relations, in parallel.

For the double pipelined algorithm, we have the similar observation as in the single pipelined algorithm. The first result comes out instantly after starting the first hash, the first and the second joins simultaneously. We then consider the time to get the first result as the time used in the hashing phase, which in this algorithm, is the maximum of the time used in doing hashes for relation 2 and 3.

We got the coefficients appearing in the above equations, by fitting them to our test results. The best fit yields the following relations:

$$\begin{aligned} t_{hashwait} &= 0.0205 \;\; millisecond, \\ t_{hashstore} &= 1.60 \;\; millisecond, \\ t_{hashcommu} &= 0.052 \;\; millisecond, \\ t_{joinstore} &= 0.0095 \;\; millisecond, \\ t_{joinwait} &= 0.002 \;\; millisecond, \\ t_{joincommu} &= 0.0068 \;\; millisecond, \\ t_{pipehashwait} &= 0.019 \;\; millisecond. \end{aligned}$$

on the Butterfly, under the experimental conditions. By examining the statistic of the fit between the theoretical results and the test results, we see a very good agreement between them. By using the standard deviation (σ) from the analytical and experimental results, we derived that the fluctuation (σ/(mean of the test data)) of the fit is within 10%.

Note that there is no bus/memory contention contribution in the second join, as predicted by Equation 5. This is the common conclusion got from much other research, where only single join operation is studied. Figure 1 shows results for the time used in the second join with different file sizes. The standard deviations of the difference between the analytical results from Equation 5 and experimental results in both tests showed a fluctuation within 5%.

The bus/memory contention effect clearly stands out in the hashes and the first join. In Figure 2 we present the time verses the number of processors in the first hash. The pretty good fit to the Equation 2 (fluctuation $\leq$ 10confirms our conjecture that the bus/memory contention is proportional to $(p-1)$.

Due to the bus/memory contention in the hash phase and in the first join, we can predicate that the overall computation time will also be affected by this contention. We show in Figure 3 the total computation time for the hash and join computations using the sequenced algorithm, with different processors and relation sizes.

An important issue in studying pipelined algorithms is to investigate the time to get the first result. Because of the parallelization among hash and join operations, we predicate that the pipelined algorithm will get the first joined result sooner than the sequenced algorithm. We measured the time needed to get the $T1_{sequenced}$, $T1_{singlepipe}$, and $T1_{doublepipe}$, and found that this always true. In Figure 4 we show the results of $T1$ by using the three algorithms. The order of the algorithms with regard to the time to get the first result, from least to most, consuming, is the double pipelined, the single pipelined, and the sequenced.

Figure 1: The computation time for the second join in the sequenced algorithm. a) $f1$=1,000, $f2$=100, $f3$=100; b) $f1$=10,000, $f2$=100, $f3$=100.

Figure 2: The computation time in building up the hash table for $F1$ in the sequenced algorithm. a) $f1$=1,000, $f2$=100, $f3$=100; b) $f1$=10,000, $f2$=100, $f3$=100.

Figure 3: The total computation time using the sequenced algorithm, for relations a) $f1$=1000, $f2$=100, $f3$=100, b) $f1$=10,000 $f2$=100, $f3$=100.

Figure 4: The time needed to get the first joined result versus the number of processor, for sequenced, single pipelined and double pipelined algorithms. $f1$=10,000, $f2$=100, $f3$=100.

5.3 The optimal point

From Equation 1–5, we can calculate the optimal processor ($Popt$), for the overall computation time, in the sequenced algorithm by the equation:

$$Popt = \sqrt{a/b} \tag{9}$$

where

$$\begin{aligned} a &= (f1 + f2 + f3) * t_{hashstore} \\ &\quad + f1 * f2 * t_{joinstore} + s1 * f1 * f2 * f3 * t_{joinstore} \qquad (10) \\ b &= (f1 + f2 + f3) * t_{hashwait} + s1 * f1 * f2 * t_{joinwait} \qquad (11) \end{aligned}$$

Equations 9–11 predict a weak $f1$-dependence for the $Popt$, if $f1 >> f2$ and $f3$. In this case, we can simplify Equations 10–11 by crossing out $f2$ and $f3$ in the first term, which will cause further simplification by crossing out the common factor $f1$ from both the numerator and the denominator.

We have conducted several groups of tests for the sensitivity analysis on the optimal point. We examined the relationship between the total computation time, the optimal point and the original relations' sizes. In our tests, we first chose $f1$=1000 as one of the join operands, then we varied the sizes of $f2$ and $f3$. There are four combinations we tested: a) $f1$=1000, $f2$=100, $f3$=100; b) $f1$=1000, $f2$=1000, $f3$=100; c) $f1$=1000, $f2$=100, $f3$=1000; and d) $f1$ =1000, $f2$=1000, $f3$=1000. The test results are presented in Tables 1–4. For the optimal point, from the tests results for b) and c), we can see that if $f1 >> f2$ or $f3$, and the hash and the first join phase use over 50% of total computation time, the optimal point is in a weak $f1$-dependence, both within p=16 to p=20. While if $f1$ is comparable with $f2$ and $f3$, and the second join dominate the whole computation, the optimal point will move towards larger processors. Similar tests for three combinations, a) $f1$=10,000, $f2$=100, $f3$=100; b) $f1$=10,000, $f2$=100, $f3$=1000; c) $f1$=10,000, $f2$=1,000, $f3$=1,000 also revealed the same conclusion. The test results can be found in Tables 5–7.

From Figure 3, we observe the similar behavior for the computation time for sequenced algorithm. The optimal p is almost identical for different $f1$ varying from $1,000$ to $10,000$, which indicates a very weak $f1$-dependence. Test results strongly support our previous analysis, qualitatively and quantitatively. The theoretical optimal point ($Popt = 10$) is in good agreement with the experimental optimal point ($p = 12$).

However, if $f1$ is comparable with $f2$, and $f3$, the minimum point will shift towards larger processors, which means that in this case the join, especially the second join operation, dominates the total computation time. In Figure 5, we show this phenomenon by giving the computation time for some larger relations.

In Figure 6, we show the sensitivity of the optimal number of processors in terms of the time to get the first result with respect to the file sizes, in single pipelined algorithm.

We noticed that there is little difference for the optimal p in these two cases, which is in agreement with what Equation 9 predicts. From Equations 7–8, we see that the choice of the optimal processor number, in term of getting the first result in pipelined

Figure 5: The total computation time with the increase in processors, in large databases. a) $f1$=10,000, $f2$=1,000, $f3$=1,000; b) $f1$=10,000, $f2$=5,000, $f3$=5,000.

algorithms should not depend on $f1$, $f2$, and $f3$, but rather on the intrinsic parameters $t_{pipehashwait}$ and $t_{hashstore}$. Theoretically, this value is given by the equation:

$$Popt = \sqrt{t_{hashstore}/t_{pipehashwait}}. \tag{12}$$

5.4 Join sequence

Having shown our experimental results and compared them with the analytical equations, we further discuss the optimal choice of the join sequence in the multi hash-join sequenced algorithm. As can be observed from Equation 4, the contribution of the bus/memory contention is proportional to $s1 * f1 * f2$. Thus the smaller the $s1 * f1 * f2$, the less the bus/memory contention affects the elapsed time of the sequenced operations.

For the purpose of testing the above optimization ideas, we conducted tests with different join sequences. The results can be found from Tables 5 and 8 and shown in Figure 7, from which we can clearly see that the smaller $s1 * f1 * f2$ choice takes less computation time.

However, we cannot get this improvement, if $f1$, $f2$, and $f3$ are comparable. Tables 9 and 10 record the timing results for $F3 \bowtie (F1 \bowtie F2)$ and $F1 \bowtie (F2 \bowtie F3)$. where $f1$=10,000, $f2$=5,000, $f3$=5,000. We can see that there are not much time saved by changing join sequence.

The same idea is also valid for sequenced multiple joins. Suppose we have $f1$, $f2$, ..., fN, and $s1$, $s2$, ..., $sN-1$, the best choice corresponds to join the largest $s * fi * fj$

at the end.

5.5 Best processors' partition

For the single and double pipelined algorithms, partitioning of processors is required. An improper partition of the processors leads to an inefficient coordination between the producer (first join) and the consumer (the second join) and frequently causes mutual waiting. This type of waiting consumes a large proportion of the total computation time (from 20% to 80%).

Through extensive timing tests for different partitions in both single and double pipelined algorithms, we have found that the time needed to get the first result is not affected by the way the processors are partitioned, while the computation time is quite sensitive to it. We assume that with the best or near best partition among processors, the extra mutual waiting time needed between different group of processors will be minimized, that is, the parallel processing part will dominate the whole operation. In pipelined algorithm, although there might be more complicated model for the interference between processors, the bus/memory contention, and communication model, the model for the parallel operation part is similar to that in the sequenced algorithm. By using the corresponding part in Equations 4–5, we should have the following reasonable

Figure 6: The time to get the first result using single pipelined algorithm. a) $f1$=1,000, $f2$=100, $f3$=100; b) $f1$=10,000, $f2$=100, $f3$=100.

Figure 7: The comparison between the two different join sequences. $s1 = s2 = 0.01$. a) join1=$F2 \bowtie F3$, join2= $F1 \bowtie (F2 \bowtie F3)$; b) join1=$F1 \bowtie F2$, join2=$F3 \bowtie (F1 \bowtie F2)$, where $f1 = 10,000$, $f2 = 100$, $f3 = 100$.

relation between the two joins for the single pipelined algorithm:

$$p1 : p2 = t_{joinstore} * f1 * f2 : t_{joinstore} * s1 * f1 * f2 * f3$$

from which we get the best partition in the single pipelined algorithm as follows:

$$p1 : p2 \quad = \quad 1 : s1 * f3 \tag{13}$$

where $p1 : p2$ is the processors' partition between the first and the second joins.

For the double pipelined algorithm, we should have the similar relation between the first hash, the first and the second join operations. The relation can be expressed as:

$$\begin{aligned} p1 : p2 : p3 \quad = \quad & t_{hashstore} * f1 : t_{joinstore} * f1 * f2 \\ & : t_{joinstore} * s1 * f1 * f2 * f3 \end{aligned}$$

which is then simplified as:

$$p1 : p2 : p3 \quad = \quad \beta : f2 : s1 * f2 * f3 \tag{14}$$

where $p1 : p2 : p3$ is the processors' partition between the first hash, the first join and the second join. Notice that we used β in Equation 14, instead of using an expression of $t_{hashstore}$ and $t_{hashcommu}$. This is because we want these best partition formulae to be empirical so that they may fit with the test result better. From our test result, we fit this β by 80.

Based on our empirical formulae, we checked our test results from the single pipelined algorithm for a) $f1$=1,000, $f2$=100, $f3$=100, where s1*f1=1; b) $f1$=1,000, $f2$=100, and $f3$=1,000, where $s1 * f3$=10. The test results showed that the empirical formula works well in this case, although usually around the theoretical best partition, there is a processors' partition range where the timing showed a very insensitive relation to the partition. The partition tests have been done exclusively for p=10, and non-exclusively for p=20, 40. The results are stored in Tables 14–21. The optimal p to get the first result for these relations is also tested, which is about 10, fitting with the theoretical result.

Figures 8 and 9 show examples of these best partition tests. With the single pipelined algorithm, the difference between the best partition tested and that from Equation 13 is less than 5%. The difference between the experimental result and that from Equation 14 with the double pipelined algorithm is less than 10%. It is worth mentioning that the difference in computation time between using the tested optimal partition and that predicted by Equations 13–14 is also quite small (less than 10%); The best partition from Equations 13–14 gives a good estimation in practice.

For double pipelined algorithm, we checked the results from a group of tests investigating the relationship between the computation time, processors' number, and the processors' partition (Table 25, Figure 9). Exclusive partition tests are on p=16 (Table 26). The empirical formula gives best partition about 4:5:5, and it is quite near the experimental best partition 4:6:6. Tests are also conducted on p=40 (Table 27), where several partitions around empirical optimal partition are tested. A few partitions were also tested with different number of processors. The minimum point for these relations are 10, exactly the same as predicted by the optimal point formula.

Figure 8: The total computation time versus different partitions in the single pipelined algorithm. $p = 10$, $s1 * f3 = 1$. $f1$=1,000, $f2$=100, $f3$=100.

Figure 9: The total computation time versus different partitions in the double pipelined algorithm. $p = 16$. Partition: 2:7:7, 4:6:6, 6:5:5, 8:4:4, 12:2:2. $f1$=10,000, $f2$=100, $f3$=100.

5.6 Sequenced vs. pipelined algorithms in getting the last joined result

It remains an open question whether the pipelined algorithms would be superior to the sequenced algorithm in terms of the elapsed time. In fact, one of the motivations of the present work is to check quantitatively the existence of such a possibility. We found that in all our tested cases, it always takes longer for the pipelined algorithm to get the last joined result. The order of the algorithm with regard to the total elapsed time to get the last joined result, from least to most, is sequenced, single pipelined, double pipelined, the order exactly opposite to that for getting the first joined result.

In the following paragraphs, we compare the total elapsed time used in the sequenced and pipelined algorithms under ideal conditions, that is, there is no waiting because of buffer limitation or improper partition, no synchronization overhead (lock, unlock), no bus/memory contention, and no communication time needed for transferring data to remote processors. Such an analysis is useful in understanding the fact that under the current experimental conditions, we can only get last result later with pipelined algorithms, and in understanding the Equations 13–14. We claim that in the ideal case, the time used to get the last joined result is the same with both sequenced and pipelined algorithms.

Let us consider the single pipelined algorithm. In this ideal case, since there is no time difference between the sequenced and single pipelined algorithm for hashes, we only need to consider the time for joins.

With the sequenced algorithm,

$$\begin{aligned} T_{lastsequ} &= T_{join1} + T_{join2} \\ &= f1 * f2 * t_{joinstore}/p + s1 * f1 * f2 * f3 * t_{joinstore}/p \\ &= (1 + s1 * f3) * f1 * f2 * t_{joinstore}/p. \end{aligned} \tag{15}$$

With the single pipelined algorithm,

$$\begin{aligned} T_{lastpipe} &= T_{join1} = T_{join2} \\ &= f1 * f2 * t_{joinstore}/p1 \\ &= s1 * f1 * f2 * f3 * t_{joinstore}/p2. \end{aligned} \tag{16}$$

Note that Equations 16 gives the partition relation $p1 : p2 = f1 * f2 * t_{joinstore} : s1 * f1 * f2 * t_{joinstore} = 1 : s1 * f3$, which is exactly the same as that in Equation 13. Since $p = p1 + p2 = p1 + p1 * s1 * f3 = p1 * (1 + s1 * f3)$, we see immediately that $T_{lastsequ} = T_{lastpipe}$, that is, in the ideal case the last result comes out at the same time in both algorithms.

Similar analysis can be done for the double pipelined algorithm. In the sequenced algorithm,

$$\begin{aligned} T_{lastsequ} &= T_{hash1} + T_{join1} + T_{join2} \\ &= f1 * t_{hashstore}/p + f1 * f2 * (1 + s1 * f3) * t_{joinstore}/p \\ &= (f1 * t_{hashstore} + f1 * f2 * (1 + s1 * f3) * t_{joinstore})/p. \end{aligned} \tag{17}$$

Figure 10: The comparison between three different algorithms in the total computation time for the multi-join operation versus the number of processors. $f1$=10,000, $f2$=100, $f3$=100.

In the pipelined case,

$$\begin{aligned} T_{lastpipe} &= T_{hash1} = T_{join1} = T_{join2} \\ &= f1 * t_{hashstore}/p1 = f1 * f2 * t_{joinstore}/p2 \\ &= s1 * f1 * f2 * f3 * t_{joinstore}/p3. \end{aligned} \tag{18}$$

Again we can see that Equation 18 gives the partition relation $p1 : p2 : p3 = t_{hashstore} : t_{joinstore} * f2 : t_{joinstore} * f2 * (1 + s1 * f3)$, which is similar to Equation 14. Since $p2 = p1 * f2 * t_{joinstore}/t_{hashstore}$, $p3 = p1 * s1 * f2 * f3 * t_{joinstore}/t_{hashstore}$, thus $p = p1 + p2 + p3 = p1 * (t_{hashstore} + f2 * (1 + s1 * f3) * t_{joinstore})/t_{hashstore}$. We then have $T_{lastsequ} = T_{lastpipe}$.

From the above discussion, we see that in the ideal case the computation time for getting the last joined result is the same with both sequenced and pipelined algorithms. In reality, because of improper partitioning of processors, high communication costs, and synchronization overhead, pipelined algorithms take more total computation time. The best partitioning rules Equations 13–14 turn out to be exact in this ideal case. Figure 10 compares the different algorithms in regard to getting the last result.

The agreement of Equations 13–14 with respect to predictions of the best partition in the ideal case indicates that the bus/memory contention, the synchronization between

Figure 11: The comparison between three different algorithms in efficiency. $f1$=10,000, $f2$=100, $f3$=100.

groups of processors, and so on, are not very significant in determining the best partition. Moreover, the β in Equation 14 can be written explicitly in the form $t_{hashstore}/t_{joinstore}$. The numerical calculation shows that they are indeed close. We use the experimental results to determine the β in our tests.

5.7 Memory usage

In this subsection we briefly discuss memory usage in regard to various algorithms. With the sequenced algorithm, all hashes and joins can be done only one at a time. This limitation requires enough storage for the intermediate results, for example, the first hash table, the results from the first join, and so on. Therefore

$$\begin{aligned} Memory(sequ) &= f1 * e1 + f2 * e2 + f3 * e3 + \\ &\quad s1 * f1 * f2 * (e1 + e2) + C \end{aligned} \tag{19}$$

where C is a constant amount of memory needed for the global data structures and variables.

With the single pipelined algorithm, although the same amount of memory is needed to store the hash tables, less memory is needed for the first join results. This is because of the first and the second join working in parallel. Only enough memory to coordinate

Figure 12: The comparison in the largest job size for the sequenced and pipelined algorithms, given the available memory fixed.

the two join operations is necessary. Thus, in the single pipelined algorithm,

$$Memory(single) \quad = \quad f1 * e1 + f2 * e2 + f3 * e3 + (e1 + e2) * V + C, \tag{20}$$

where $p \leq V \leq s1 * f1 * f2$. When $V = p$, there is only one memory slot on each processor to store the first join result. This saves memory, but at the same time, it introduces much waiting time in both first and second join, since they have a producer-consumer relationship.

We can analyze similarly the memory usage in the double pipelined algorithm. The amount of memory that can be saved, comparing to that with the single pipelined algorithm, comes from the part that stores the first hash table. Therefore,

$$Memory(double) \quad = \quad f2 * e2 + f3 * e3 + e1 * V1 + (e1 + e2) * V2 + C, \tag{21}$$

where $p \leq V1 \leq f1$, and $p \leq V2 \leq s1 * f1 * f2$. The pro and con in choosing $V1, V2$ are similar to the discussion in the single pipelined case.

6 The future work

Our future work in this direction includes the optimizer generation based on the analytical work presented in this report, the generalization of the analysis to non-uniformly distributed join attributes and non-evenly distributed work load among processors. The system throughput problem will also be of interest to us. From our test data, especially the results from the sequenced algorithm, we found that the marginal utility of processors is decreasing with the increase of the number of processors, and it becomes nearly zero around the minimum point. This observation suggests that if there are multiple jobs submitting to a system simultaneously, a good scheduling strategy will maximize the system throughput.

7 Summary

In this report, we have presented an application of the pipelined and sequenced parallel multi-join ideas, and analyzed the performance behavior for three hash based join algorithms.

• The bus/memory contention limits the processor's utilization (that is, the *Popt* can't be too large in some cases). This effect is proportional to $(p-1)$, and can't be ignored in multi-join algorithms.

• The formulae for the total computation time and the time to get the first result for the sequenced algorithm can be generalized to N-hash—$N-1$join.

• The pipelined algorithms get the first result sooner than the sequenced algorithm, with the same join sequence.

• The optimal processor number, with both the sequenced and pipelined algorithms, is in weak dependence on the relation sizes, if the first join operand is much greater than the other two operands. In this case, it depends mainly on the intrinsic parameters of the machine.

• The best partition formulae for both single and double pipelined algorithms are empirical, but work well when checking with the experimental results.

• The double pipelined formula for the best partition includes a parameter β, which may be estimated from $t_{hashstore}/t_{joinstore}$.

• The pipelined algorithms needs longer elapsed time than sequenced algorithm to finish the entire join operation.

• An optimal join sequence, when using the sequenced algorithm, joins the largest relation last.

• Memory usage with the pipelined algorithms can be less than that in the sequenced algorithm.

References

[Bitt] Bitton, D., D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems - A Systematic Approach". Proceedings of the 1983 Very Large Database Conference, October, 1983.

[Bitton] Bitton, D., et al., "Parallel Algorithms for the Execution of Relational Database Operations", ACM Transaction on Database Systems, Vol. 8, No. 3, September 1983, Pages 324-353

[Boral] Boral, H., DeWitt, D., et al., "Parallel Algorithms for the Execution of Relational Database Operations", Computer Sciences Technical Report # 402, Computer Sciences Department, University of Wisconsin-Madison

[Brat] Bratbergsengen, Kjell, "Hashing Methods and Relational Algebra Operations", Proceedings of the 1984 Very Large Database Conference, August, 1984.

[Butterfly] "Inside the Butterfly Plus", BBN advanced computers inc., 10 Fawcett Street, Cambridge, MA 02238.

[DeWitt1] DeWitt, D., et al., "Implementation Techniques for Main Memory Database Systems", Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.

[DeWitt2] DeWitt, D., et al., "Multiprocessor Hash-Based Join Algorithms", Proceedings of VLDB 85, Stockholm

[Du] Du, H.C., "Distributing a Database for Parallel Processing is NP-hard", Technical Report 83-9, Department of Computer Science, University of Minnesota

[Gajski] Gajski, D., Kim, Won, et al., Proceedings of Computer Architecture Conference, 1984, Pages 134-141

[Hurson] Jirspm. A.R., "VLSI TIME/SPACE Complexity of an Associative Parallel Join Module", Proceedings of the International Conference on Parallel Processing, 1986, Pages 379-386

[Itoh] Itoh, H., et al., "Parallel Control Techniques for Dedicated Relational Database Engines", ICOT research center, technical report, TR-182

[Jajodia] Jajodia, Sushil and Rosenau, Todd., "Implementing Relational Database Operations on Shared-Memory MIMD Computers", Naval Research Laboratory, Washington, D.C.

[Kitsu] Kitsuregawa, M., et al., "Application of Hash to Data Base Machine and its Architecture", New Generation Computing, Vol. 1, No. 1, 1983.

[Lin] Lin, Diana, Paradata, Department of Computer Science, Stanford University, 1989

[Murphy] Murphy, M., et al., “Effective Resource Utilization for Multiprocessor Join Execution”, Technical Report LBL-26601, Computer Science Research Department, Lawrence Berkeley Laboratory.

[Nakay] Nakay, T., et al., “Architecture and Algorithm for Parallel Execution of a Join Operation”, CSG Technical Report 83-19

[Qadah] Qadah, G., et al., “Evaluation of Performance of the equi-join Operation on the Michigan Relational Database Machine”, Proceedings of International Conference on Parallel Processing, 1984, Pages 260-265

[Richard] Richardson, J., et al., “Design and Evaluation of Parallel Pipelined Join Algorithms”, ACM 0-89791-236-5/87/0005/0399, 1987

[Shultz] Shultz, Roger, et al., “Memory Capacity in Multiple Processor Joins”, Technical Report 86-10, Department of Computer Science, The University of IOWA

[Stone] Stone, H., “Database Applications of the Fetch-and-Add Instruction”, Technical Report RC 9467 (# 41881) 7/16/82, IBM Thomas J. Watson Research Center

[Swami] Swami, Arun, “Optimization of Large Join Queries: Distributions of Query Plan Costs”, Computer Science Department, Stanford University, June, 1989, to appear in ACM(SOSP) 89

[Valdu] Valduriez, P., et al., “Join & Semijoin Algorithms for a Multiprocessor Database Machine”, ACM Transaction on Database Systems, Vol. 9, No. 1, March 1984, Pages 133-161

[Wieder] Wiederhold, Gio and Keller, Arthur, “Notes on Databases on Parallel Computers”, Department of Computer Science, Stanford University, 1988

[Tsitsik] Tsitsik, J., et al., “The Throughput of a Precedence-Based Queuing Discipline”, Report No. STAN-CS-84-1017, Department of Computer Science, Stanford University

[Yamane] Yamane, Y. et al., “Design and Evaluation of a High-Speed Extended Relational Database Engine”, XRDB, Fujitsu Laboratories Ltd.