Multiprocessor Hash-Based Join Algorithms

David J. DeWitt
Robert Gerber

Computer Sciences Department
University of Wisconsin

This research was partially supported by the Department of Energy under contract #DE-AC02-81ER10920 and the National Science Foundation under grant MCS82-01870.

ABSTRACT

This paper extends earlier research on hash-join algorithms to a multiprocessor architecture. Implementations of a number of centralized join algorithms are described and measured. Evaluation of these algorithms served to verify earlier analytical results. In addition, they demonstrate that bit vector filtering provides dramatic improvement in the performance of all algorithms including the sort merge join algorithm. Multiprocessor configurations of the centralized Grace and Hybrid hash-join algorithms are also presented. Both algorithms are shown to provide linear increases in throughput with corresponding increases in processor and disk resources.

1. Introduction

After the publication of the classic join algorithm paper in 1977 by Blasgen and Eswaran [BLAS77], the topic was virtually abandoned as a research area. Everybody "knew" that a nested-loops algorithm provided acceptable performance on small relations or large relations when a suitable index existed and that sort-merge was the algorithm of choice for ad-hoc[1] queries. Last year two papers [DEWI84a, BRAT84] took another look at join algorithms for centralized relational database systems. In particular, both papers compared the performance of the more traditional join algorithms with a variety of algorithms based on hashing. The two papers reached the same conclusion: that while sort-merge is the commonly accepted algorithm for ad-hoc joins, it is, in fact, not nearly as fast as several join algorithms based on hashing. In retrospect, it is interesting to observe that a simple, but very good algorithm has been virtually ignored[2] simply because System R [ASTR76] did not support hashing as an access method.

The motivation for the research described in this paper was twofold. First, since [DEWI84a] and [BRAT84] were both analytical evaluations, we wanted to implement and measure the algorithms proposed in these papers in a common framework in order to verify the performance of the hash-based join algorithms. Second, we wanted to see if the results for a single processor could be extended to multiple processors. The hash-based join algorithms described in [DEWI84a], and in particular the Hybrid algorithm, made very effective use of main memory to minimize disk traffic. It seemed that since multiprocessor joins require that data be moved between processors, that the multiprocessor hash-based join algorithms might minimize the amount of data moved in the process of executing a join algorithm. Hash-based multiprocessor join algorithms for multiprocessors are not new. They were first suggested in [GOOD81], next adopted by the Grace database machine project [KITS83], and evaluated in [VALD84]. While each of these papers made important contributions to understanding multiprocessor hash-based join algorithms, a number of questions remain. First, in [GOOD81], it is hard to factor out the influence of the X-tree architecture and the parallel readout disks on the results obtained. [KITS83], on the other hand, concentrates on the speed of the sort-engine and not the overall performance of the Grace hash-join algorithm. Finally, the algorithm presented in [VALD84] exploits hashing only during the partitioning process and resorts to a pure nested loops algorithm aided by bit vector filtering during the join phase. The goal of our research was to examine the

1 By "ad-hoc" we mean a join for which no suitable index exists.

2 While INGRES [STON76] uses hashing for ad-hoc queries, the limited address space of the PDP 11 on which INGRES was first implemented made it impossible to exploit the use of large amounts of memory effectively. Consequently, the algorithm never received the recognition it deserves.

multiprocessor hash-join algorithms in a multiprocessor environment that enabled us to identify CPU, communications, and I/O bandwidth design parameters.

In Section 2, we review the join algorithms and analytical results presented in [DEWI84a]. As a first step toward developing a multiprocessor version of the hash based join algorithms, we implemented the join algorithms described in [DEWI84a] on top of the Wisconsin Storage System (WiSS). The results presented in Section 3 verify the analytical results presented in [DEWI84a]. Based on these results, we feel that all relational database systems should provide a hash-based join algorithm in order to effectively exploit main memory as it becomes increasingly inexpensive. The algorithms described in Section 3 were also used to gather some "real" numbers for use in a simulation of the multiprocessor join algorithms. In Section 4, we describe two multiprocessor hash join algorithms. We also present the results of a simulation study of these algorithms. The results are extremely exciting as they indicate that both algorithms provide very close to a linear speedup in performance with corresponding increases in resources. In Section 5, our conclusions and our plans for a new database machine based on these multiprocessor join algorithms are described.

2. An Overview of Hash-Partitioned Join Operations

In [DEWI84a], the performance of three hashed-based join algorithms (termed Simple, Grace [KITS83], and Hybrid) were compared with that of the more traditional sort merge algorithm. In the following discussion of the hash-partitioned join algorithms, the two source relations will be named R and S. R is assumed to be smaller (in pages) than S. All hash-join algorithms begin by partitioning R and S into disjoint subsets called **buckets** [GOOD81, KITS83]. These partitions have the important characteristic that all tuples with the same join attribute value will share the same bucket. The term bucket should not be confused with the overflow buckets of a hash table. The partitioned buckets are merely disjoint subsets of the original relations. Tuples are assigned to buckets based upon the value of a hash function that is applied to a tuple's join attribute value. Assuming that the potential range of hash values is partitioned into the subsets X_1, ..., X_n, then every tuple of R whose hashed join attribute value falls into the range of values associated with X_i will be put into the bucket R_i. Similarly, a tuple of S that hashes to the partition X_i will be put into the bucket S_i. Since the same hash function and partitioning ranges are used with both relations, the tuples in bucket R_i will only have to be joined with those tuples in S_i. It will be the case that tuples from bucket R_i will never have join attribute values equal to those of tuples in S_j where $i \neq j$. The potential power of this partitioning lies in the fact that a join of two large relations has been reduced to the separate joins of many smaller relation buckets.

The hash-join algorithms have two distinct phases. In the first phase, relations R and S are partitioned into buckets. In a centralized environment, this partitioning might be done by allocating a page frame to buffer the tuples being assigned to the particular buckets. As a page buffer is filled, it is flushed to a file on disk that represents a particular bucket. Each relation is scanned and partitioned in turn. At the end of the partitioning phase, relations R and S are represented by equal numbers of bucket files that have been written to disk. This partitioning phase must create a suitable number of buckets such that each bucket of relation R will be small enough to fit into main memory. The size of the buckets of S can be ignored because, at most, only a single page of relation S needs to be resident in memory at a time during the join phase.

The second phase of the hash-join algorithms effects the actual search for tuples from relations R and S that have matching join values. Any of the traditional join methods could be used in this phase to realize the final join result. However, as relation R has been partitioned into buckets that will fit into memory, it seems most appropriate to use a hash-based algorithm to process the search for matching join tuples. This second phase will be referred to as the join phase. In the first step of the join phase, a bucket R_i is used to build a hash table in main memory. Then bucket S_i is read and each tuple of S_i is used to probe the hash table for matches.

2.1. Problems with Hash Join Algorithms

The partitioning phase must ensure that the size of the buckets created from relation R do not exceed the size of main memory. Guaranteeing that a chosen partitioning of hash values will result in buckets of relation R that will fit in memory is not necessarily trivial. The problem of buckets growing unacceptably large is termed bucket overflow. The choice of an appropriate hash function will tend to randomize the distribution of tuples across buckets and, as such, will minimize the occurrence of bucket overflow. If the chosen hash function fails to distribute the tuples uniformly and bucket overflow occurs, a number of remedies are available. The relations could be partitioned again with another hash function. This solution is almost always too expensive. A better alternative is to apply the partitioning process recursively to the oversized buckets [DEWI84a, BRAT84]. The net effect of this solution is to split an oversized bucket into two or more smaller buckets. If relation R is partitioned before relation S, then this method only requires rescanning the particular bucket that overflowed. The range of values governing the partitioning of relation S can be adjusted to reflect the final partitioning of R after bucket overflow has been handled. This method could fail in the case that the combined sizes of tuples having identical join values exceeds the size of available memory. In such a case, a hash-based variation of the nested loops join algorithm can be applied. The performance of such an algorithm is analyzed in Section 3 of this paper. The solutions used to handle bucket overflow can also be applied to the overflow of a hash table.

2.2. Simple Hash-Join

The Simple hash-join processes one bucket at a time while doing a minimal amount of partitioning. In fact, the partitioning and join phases are executed simultaneously. Two files are associated with relations R and S. There are files R_input (S_input) which contain tuples that are waiting to be processed by the current phase of the algorithm. The files R_output (S_output) contain tuples that have been passed over by the current phase of the algorithm. At the start of the algorithm, R_input and S_input are set to equal the relations R and S. R_output and S_output are initially empty.

A partitioning basis consisting of a number and range of of hash values is chosen at the start. There will be as many stages to the algorithm as there are buckets of relation R. The buckets of R are sequentially used to build hash tables in main memory. One hash table is built at the start of each stage. Each stage begins with a scan of R_input. As each tuple is considered, if it belongs to the targeted memory bucket R_i the tuple is added to the hash table. Otherwise, the tuple is written to R_output. R_output contains all the remaining buckets that are not of current interest. Then S_input is scanned sequentially. If a tuple of S_input hashes to bucket S_i, then it is used to probe the hash table built from bucket R_i. If a match is found, the tuples are joined and output. Otherwise, (ie. the tuple does not belong to bucket S_i) it is written to S_output.

At the end of each stage of the Simple hash-join, the R_output (S_output) file becomes the R_input (S_input) file that will be used by the next stage. As the algorithm progresses, the R_output (S_output) file becomes progressively smaller as the buckets of interest are consumed. The algorithm finishes when either R_output or S_output are empty following a processing stage.

2.3. Grace Hash-Join

The Grace hash join algorithm [GOOD81, KITS83] is characterized by a complete separation of the partitioning and joining phases. The partitioning of relations R and S is completed prior to the start of the join phase. Ordinarily, the partitioning phase creates only as many buckets from relation R as are necessary to insure that the hash table for each bucket R_i will fit into memory. Since only a single page frame is needed as an output buffer for a bucket, it is possible that memory pages will remain unused after the requisite number of bucket buffers have been allocated. In the Grace algorithm, these extra pages can be used to increase the number of buckets that are generated by the partitioning phase. Following the partitioning phase, these smaller buckets can be logically integrated into larger buckets that are of optimal size for building the in-memory hash tables. This strategy is termed **bucket tuning** [KITS83]. Bucket tuning is a useful method for avoiding bucket overflow.

2.4. Hybrid Hash Join

The Hybrid hash join was first described in [DEWI84a]. All partitioning is finished in the first stage of the algorithm in a fashion similar to the Grace algorithm. However, whereas the Grace algorithm uses any additional available memory during the partitioning phase to partition the relations into a large number of buckets, Hybrid uses additional memory to begin the joining process. Hybrid creates the minimum number of buckets such that each bucket can be reasonably expected to fit into memory. Allocating one page frame to be used as an output buffer for each bucket, the Hybrid algorithm utilizes any remaining pages frames to build a hash table from R_0. The partitioning range is adjusted to create N equal-sized buckets, R_1, ..., R_N, that are written to disk and one independently sized bucket, R_0 , that is used to build the hash table. The same partitioning range is used for relation S. Tuples of S that hash into bucket S_0 are immediately used to probe the hash table for matches. When the partitioning phase completes, the Hybrid hash-join has already completed processing part of the join phase. Thus, the tuples that are immediately processed do not have to be written to and retrieved from the disk between the partitioning and join phases. These savings become significant as the amount of memory increases.

2.5. Sort-Merge Join Algorithm

The standard sort-merge [BLAS77] algorithm begins by producing sorted runs of tuples that are, on the average, twice as long as the number of tuples that can fit into a priority queue in memory [KNUT73]. This requires one pass over each relation. During the second phase, the runs are merged using an n-way merge, where n is as large as possible. If n is less than the number of runs produced by the first phase, more than two phases will be needed. In the final phase, the sorted source relations are sequentially scanned and matching tuples are joined and output.

2.6. Comparison of the Four Join Algorithms

Figure 1 displays the relative performance of the four join algorithms using the analysis and parameter settings presented in [DEWI84a]. The vertical axis is execution time in seconds. The horizontal axis is the ratio of $\frac{|M|}{|R|*F}$ where |M| and |R| are, respectively, the sizes of main memory and the R relation in pages and F equals 1.2 (F is a fudge factor used to account for the fact that even if |R| = |M|, a hash table for R will occupy more than |R| pages in main memory). For all the algorithms, R and S are assumed to be resident on mass storage when the algorithm begins execution. These results clearly indicate the advantage of using a hash based join algorithm over the more traditional sort merge algorithm. In retrospect, the results are not too surprising, as sorting creates a total ordering of the records in both files, while hashing simply groups related records together in the same bucket.

3. Evaluation of Centralized Hash Partitioned Join Algorithms

To verify the analysis presented in [DEWI84a] and to gather information on CPU and I/O utilizations during the partitioning and joining phases of the three hashing algorithms, we implemented the Simple, Grace, and Hybrid algorithms on a VAX 11/750 running 4.2 Berkeley UNIX. In addition to the three hash-partitioned join algorithms, two other popular join algorithms were studied. These algorithms, a sort-merge algorithm and a hash-based nested loops algorithm, provide a context for comparing the performance of the hash-partitioned join algorithms. All the algorithms were implemented using the Wisconsin Storage System (WiSS) [CHOU83].

3.1. An Overview of WiSS

The WiSS project was begun approximately 3 years ago when we recognized the need for a flexible data storage system that could serve as the basis for constructing experimental database management systems. While originally conceived as a replacement for the UNIX file system (WiSS can run on top of a raw disk under UNIX), WiSS has also been ported to run on the Crystal multicomputer [DEWI84b]. The services provided by WiSS include structured sequential files, byte-stream files as in UNIX, B^+ indices, stretch data items, a sort utility, and a scan mechanism. A sequential file is a sequence of records. Records may vary in length (up to one page in length), and may be inserted and deleted at arbitrary locations within a sequential file. Optionally, each sequential file may have one or more associated indices. The index maps key values to the records of the sequential file that contain a matching value. The indexing mechanism is also used to construct UNIX-style byte-stream files (the pages of the index correspond to the inode components of a UNIX file). A stretch item is a sequence of bytes, very similar to a file under UNIX. However, insertion and deletion at arbitrary locations is supported. Associated with each stretch item (and each record) is a unique identifier (RID). By including the RID of a stretch item in a record, one can construct records of arbitrary length. As demonstrated in [CHOU83], WiSS's performance is comparable to that of commercially available database systems.

3.2. Summary of Algorithms Evaluated

Centralized versions of the Grace, Simple and Hybrid hash-partitioned join algorithms were implemented in the manner described in Section 2. A modified version of the nested loops algorithm, termed Hashed Loops, was also implemented [BRAT84]. The Hashed Loops algorithms is so named because it uses hashing as means of effecting the internal join of tuples in main memory. It is similar to the algorithm used by the university version of INGRES [STON76]. For each phase of the Hashed Loops algorithm, a hash table is constructed from those pages of R that have been staged into memory. Tuples from S are used as probes into the hash table. Constructing such a hash table avoids exhaustively scanning all of the R tuples in memory for each tuple in S as is done with the simpler form of the nested loops algorithm. The last algorithm, the Sort Merge join, employed the sort utilities provided by WiSS.

All the algorithms were allocated identical amounts of main memory for buffering pages of the relation. Similarly, all the algorithms accessed relations on disk a page at a time, blocking until disk operations completed.

3.3. Presentation of Performance Results

The join algorithms were compared using queries and data from the Wisconsin Benchmark Database [BITT83]. As in Figure 1, the execution time of each join algorithm is shown as a function of the amount of available memory relative to the size of the smaller relation. The relative amount of memory is defined to be the number of pages of main memory, $|M|$, divided by the size in pages of the smaller relation, $|R|$. The elapsed times for all join algorithms include the time required to write the final result relation to disk. All tests were run in single user mode. The test machine had 8 megabytes of memory so no paging occurred. Bucket overflow did not occur in any of the tests of the hash-partitioned algorithms.

The results of joining two 10,000 tuple relations using each of the join algorithms is presented in Figure 2. The join produces 10,000 result tuples. The join attribute is a randomly ordered, two byte integer. Every tuple from both relations participates in the result relation produced by this join query. Whereas Figure 1 presented performance results that were calculated from analytical models, Figure 2 presents the measured performance of actual implementations of the algorithms. We find the similarity of Figures 1 and 2 both reassuring and encouraging.

In Figure 2, the performance of the Grace hash-join algorithm is constant for the given range of available memory. This results from the total separation of the partitioning and join phases in the Grace algorithm. From a performance viewpoint, the Grace algorithm only uses memory optimally during the joining phase. Excess memory during the partitioning phase is used as a means of creating a large number of buckets for the bucket tuning process. In contrast, the performance of the Simple hash-join algorithm is significantly affected by the amount of available memory and performs well only when the smaller relation is less than twice the size of available memory. The performance of the Hybrid algorithm reflects the fact that it combines the best performance features of the Grace and Simple[3] hash-join algorithms. Since Hybrid completes all partitioning in a single pass through both source relations, it's performance is always as least as good as that of the Grace algorithm. The Hybrid algorithm increasingly outperforms Grace as the amount of relative memory increases because the additional memory is used for immediately joining tuples from one bucket of each source relation. Such immediate joining eliminates the cost of writing and reading these tuples to disk between the partitioning and joining phases. The performance of all of the hash-partitioned algorithms remains unchanged once the smaller relation fits in memory. This point occurs at a relative memory value of 1.2 and not when available memory exactly equals the size of the smaller relation. This results from the fact that the hash-join algorithms use some of the available memory during the join phase for the structure of the hash table itself. Also, it must be realized that partitioning is a predictive process and as such, prudence requires that additional memory be used to accommodate fluctuations in the size of the hash tables that are constructed from buckets.

The performance of the Sort-Merge join algorithm is constant over a wide range of available memory in Figure 2. Until a source relation fits into memory, the sorting process completely reads and writes the relation at least twice, once when the sorted runs are produced and a second time when the sorted runs are merged. The Sort-Merge join algorithm then reads the source relations a third time to effect the final joining of tuples. An optimization is possible. Given sufficient memory, the sorted runs of both relations can be merged and joined simultane-

[3] The Simple hash-join algorithm should have performance equal to that of the Hybrid algorithm for relative memory values in excess of approximately 0.5. The difference in the performance of the Hybrid and Simple hash-join algorithms for relative memory values between 0.5 and 1.2 is an artifact of the implementation.

ously. In this case, the performance of the Sort-Merge algorithm could be expected to be similar to the Grace algorithm as each algorithm would access every page of each source relation three times (two reads and a write).

Perhaps, surprisingly, the Hashed Loops algorithm has quite good performance over a wide range of available memory in Figure 2. Due to the existence of the hash table, the cost of of probing for matches with tuples from relation S is a relatively inexpensive operation. The algorithm performs especially well when the size of the smaller relation is less than twice the size of available memory. As this is exactly the situation one would expect in the case of bucket overflow, the hash-based nested loops algorithm is an attractive remedy for handling bucket overflow.

The performance of the join algorithms for a join of a 1,000 tuple relation with a 10,000 tuple relation is shown in Figure 3. The result relation contains 1,000 tuples. The Hybrid algorithm continues to dominate all the other join algorithms over a wide range of relative memory values. The stepwise performance transitions of the Sort-Merge and nested loops algorithms become more obvious in the environment of this query.

Figure 4 reflects the performance of the join algorithms on the same query used for Figure 3. The difference is that in Figure 4 all the algorithms use bit vector filtering techniques [BABB79, BRAT84, VALD84]. The notable performance improvements demonstrated are the result of eliminating, at an earlier stage of processing, those tuples that will not produce any result tuples. The bit vector filtering technique used by the hash-partitioning and Sort-Merge algorithms are very similar.[4] Prior to the initial scan of relation R, a bit vector is initialized by setting all bits to 0. As each R tuple's join attribute is hashed, the hashed value is used to set a bit in the bit vector. Then as relation S is scanned, the appropriate bit in the bit vector is checked. If the bit is not set, then the tuple from S can be safely discarded. Applying the bit vector from relation R against relation S approximates a semijoin of relation S by relation R. The net impact of this process depends on the semijoin selectivity factor of relation S by R which is defined to be the ratio of tuples resulting from the semijoin of S by R relative to the cardinality of S. In the example query of Figure 4, the semijoin of relation S by R has a semijoin selectivity factor of 0.1. The net effect is that approximately 90% of the tuples of relation S can be eliminated at a very early stage of processing by the hash-partitioned and Sort-Merge algorithms. Significant I/O savings accrue from the fact that these non-participating tuples do not have to be stored on disk between the partitioning and joining phases of the hash-partitioning algorithms. Two disk accesses are saved for every page of tuples that can be eliminated by the bit

[4] The bit vector filtering technique used by the hash-partitioned and Sort-Merge algorithms is directly extendible to the case of Hashed Loops if the names of the relations in the discussion are reversed.

vector filtering of relation S.

Since, the Hashed Loops algorithm does not complete a scan of relation R until the end of the query, it must instead use bit vector filtering to approximate a semijoin of relation R by S. In Figure 4 the semijoin selectivity factor for a semijoin of R by S is 1.0. Therefore, in this instance, the Hashed Loops algorithm doesn't derive any benefit from applying bit vector filtering.

Collisions that occur in the process of accessing bit vectors may result in non-qualified (phantom) tuples being propagated along to the final joining process. The phantom tuples will, however, be eliminated by the final joining process. The number of phantom tuples can be reduced by increasing the size of the bit vector or by splitting the vector into a number of smaller vectors [BABB79]. A separate hash function would be associated with each of the smaller bit vectors. The costs associated with bit vector filtering are modest. For the given test, a single bit vector of length 4K bytes was used. Since the hash-partitioning algorithms already compute the hashed value of each tuple's join attribute, the only additional cost of bit vector filtering for these algorithms is the amount of space required for the bit vector itself.

4. Multiprocessor Hash-based Join Algorithms

Multiprocessor versions of the Hybrid and Grace algorithms are attractive for a number of reasons. First, the ability of these algorithms to cluster related tuples together in buckets provides a natural opportunity for exploiting parallelism. In addition, the number of buckets produced during the partitioning phase (or activated in the joining phase) of each algorithm can be adjusted to produce the level of parallelism desired during the joining phase. Second, the use of buckets by multiprocessor versions of the two algorithms should minimize communications overhead. Furthermore, just as the centralized form of the Hybrid algorithm made very effective use of main memory in order to minimize disk traffic, one would expect that a multiprocessor version of the Hybrid hash join algorithm should be able to use memory to minimize both disk and communications traffic. Finally, it appears that control of these algorithms can also be decentralized in a straightforward manner.

4.1. Horizontal Partitioning of Relations

All relations are assumed to be horizontally partitioned [RIES78] across all disk drives in the system. From the view point of raw bandwidth, this approach has the same aggregate bandwidth as the disk striping strategies [GARC84, KIM85, BROW85] given an equal number of disk drives. The difference is that in our approach once the data has been read, it can be processed directly rather than being transmitted first through some interconnection network to a processor.

There are at least two obvious strategies for distributing tuples across the disks drives in the system. One approach is to apply a randomizing function to each tuple (or the key attribute of the tuple) to select a disk for storing the tuple. Each processor maintains an independent index on the tuples stored on its disk. The advantage of this approach is that as additions are made to the file, the number of tuples on each disk should remain relatively well balanced. The second approach is to cluster tuples by key value and then distribute them across the disk drives. In this case the disks, and their associated processors, can be viewed as nodes in a primary, clustered index. A controlling processor acts, in effect, as the root page of the index. We intend to investigate whether traditional tree balancing algorithms provide acceptable performance in such an environment. This approach is similar to, but much simpler than, the clustering approach employed by MDBS [HE83]. In MDBS [HE83], each backend processor must examine every query as the clustering mechanism is implemented by the backends, not the controlling processor.

The real advantage of the second approach comes when processing queries. With the first distribution strategy (ie. tuples distributed randomly), except in the case of exact match queries on the attribute used to distribute tuples, all processors must execute every query. With the second distribution strategy, the controlling processor (which maintains the root page of each index) can direct each query to the appropriate processors. While there is certainly some overhead in performing this function, it is certainly less than the cost of sending the query to all the processors. Furthermore, for even a fairly large database, the root pages of all indices should fit in the controlling processor's main memory. While the two distribution strategies should provide approximately the same response time in single user benchmarks, we expect that system throughput would be significantly higher with the second distribution strategy in a multiuser environment. When no suitable index is available, all processors are available to perform the query.

4.2. Description of Multiprocessor Hybrid and Grace Algorithms

The algorithms described in this section assume that the relations of the database have been horizontally partitioned across multiple disk drives in the manner described above. Each disk drive has a processor associated with it. (**Note**, the converse is not necessarily true.) For the purposes of the performance evaluation presented below, we have assumed that the processors are interconnected with an 80 Mbit/second token ring. As we will demonstrate, such an interconnection topology provides adequate performance even when 50 processors are being used.

4.2.1. A Multiprocessor Version of the Grace Hash-Join Algorithm

There appear to be a number of alternative strategies for parallelizing the Grace hash-join algorithm. The approach we have selected and evaluated assumes that a different set of processors is used for the joining and partitioning phases of the algorithms. Furthermore, while the "partitioning" processors are assumed to have disk drives associated with them, the "joining" processors are assumed to be diskless. One reason that we find this strategy attractive is that diskless nodes are cheaper than nodes with disks and we are interested in exploring whether such processors can be effectively utilized.

The algorithm proceeds as follows. Each node with a disk partitions the smaller relation into buckets that are written across the network to the nodes without disks. These nodes perform the join phase of the algorithm. Each joining node will contain a single hash table that has been built from the tuples of a single bucket of the smaller source relation. After the hash tables have been completely built, the larger relation is partitioned and the buckets are sent to the joining nodes. Corresponding buckets of both source relations are guaranteed to be sent to the same joining node. Tuples from the larger relation are used to probe the join node hash tables for matches. If the size of the smaller relation exceeds the aggregate memory capacity of the joining nodes, multiple phases are necessary and unused buckets will be temporarily saved on the disks attached to the partitioning nodes.

The multiprocessor Grace algorithm can allocate varying numbers of joining nodes. The Grace algorithms have been named according to the ratio of partitioning nodes to joining nodes. The Grace 1:1 design allocates one partitioning node for each joining node. There is one partitioning node for every two joining nodes in the Grace 1:2 design. Finally, the Grace 2:1 design allocates two partitioning nodes for each joining node. While these design combinations proved optimal for the execution of single join queries, it may very well be the case that more varied combinations of processors may prove optimal for more complex queries.

4.2.2. A Multiprocessor Version of the Hybrid Hash-Join Algorithm

While the multiprocessor Grace algorithm employs a combination of processors with and without disks, the multiprocessor Hybrid algorithm requires that each processor has a disk drive. The multiprocessor Hybrid hash-join algorithm performs the partitioning and joining phases on the same nodes. Each processor partitions the source relations in a fashion similar to the Grace algorithms. However, each node allocates excess memory during the partitioning phase to a hash table for one bucket of tuples. As the source relations are partitioned on a local Hybrid processor, most tuples are written across the net to the appropriate join node. Tuples belonging to the bucket associated with a partitioning processor are instead immediately used to either build or probe the local hash table. Because some of the tuples can be processed locally, the Hybrid hash join algorithm generates a relatively lighter network load than the Grace algorithm. For a given level of resources, a Hybrid multiprocessor algorithm will use more disks and fewer processors than a Grace multiprocessor algorithm.

4.3. Discussion of Simulation Model

To evaluate the performance of the distributed Hybrid and Grace hash-join algorithms a simulation model of the proposed multiprocessor architecture was constructed. The hardware components that are represented in the model are intended to be examples of current, commercially available components. The capabilities of the various components can be varied to test the effects of various combinations of resources. While the distributed hash-partitioned algorithms could be implemented in many different kinds of network environments, the processors in the current simulation are loosely coupled via a token ring network.

4.3.1. Hardware

The model allows us to simulate 1, 2, and 3 MIP processors. The disk drives were modeled after the Fujistu Eagle drive and are assumed to support a transfer rate of 1.8 Mbytes/second. The combined positioning and latency times have been modeled as a normal distribution with a mean value of 26 milliseconds and a standard deviation of 4 milliseconds. The processor’s network interface is assumed to have a single, output buffer of 2 Kbytes. A similar input buffer is assumed. The effective DMA bandwidth at which these buffers can be filled or flushed to the main memory of a processor is assumed to be either 4 Mbits/second or 20 Mbits/second. The 4 Mbits/second number is derived from measurements made on a VAX 11/750 with a Proteon ProNet interface [PROT83] attached to the Unibus. The 20 Mbits/second is an estimate of the DMA rate if the device were attached to the internal bus of a VAX 11/750. The token ring is assumed to have a bandwidth of either 10 Mbits/second or 80 Mbits/second. The 10 Mbits/second value is representative of currently available local network interfaces such as the ProNet token ring. The 80 Mbits/second interface with a 20 Mbits/second DMA rate is representative of the CI interface from Digital Equipment Corporation.

Since the multiprocessor version of the Hybrid algorithm requires that each processor has a disk drive, while the Grace algorithm employs processors with and without disks, a method for computing the "cost" of a particular configuration of processors and disk was needed. The approach we adopted was to assume that a 1 MIP processor cost the same as a disk drive and controller. The relative cost of the 2 and 3 MIP processors was computed using Grosch's law [GROS53] which relates the cost of a processor to the performance (speed) of the processor:

$$\text{Performance} = \text{Technology\_Constant} * \text{Processor\_Cost}^{g}$$

The technology constant and cost exponent were assigned, respectively, values of 1 and 1.5.[5] The cost of a particular configuration is calculated by computing the aggregate cost of all processors[6] and disks.

So far, we have not incorporated memory or communications costs in our cost model. It might, for example, be more cost effective to use more memory and a lower speed communication device.

Using this cost model, calculating the cost of a particular configuration of processors and disks is straightforward. The reverse transformation, is not, however, always obvious. Assume, for example, that all processors are 1 MIP processors. Then a Hybrid join configuration with a cost of 10 will consist of 5 processors and 5 disks. A 2:1 Grace configuration (2 partitioning processors for every join processor), with a cost of 10 will consist of 4 partitioning processors, 4 disks, and 2 joining processors. No 1:2 or 1:1 Grace configuration will have exactly a cost of 10. For example, a 1:2 configuration with 2 partitioning nodes, 2 disks, and 4 joining nodes has a cost of 8 while the next 1:2 configuration (3,3,6) has a cost of 12. Likewise, the 1:1 Grace configuration with a cost closest to 10 will have 3 partitioning processors, 3 disks, and 3 join processors. To facilitate interpretation of the results presented below, we have summarized in Table 1 the resource costs of 6 alternative hardware configurations (assuming 1 MIP processors) for each of the four algorithms (Hybrid, Grace: 1:1, 2:1, and 1:2). The same 6 configurations were also used when we evaluated 2 and 3 MIP processors. While the cost of each configuration changes for these cases, the table can still be used to determine the hardware configuration associated with each data point.

4.3.2. Software

The operation of each simulated processor is controlled by a simple operating system kernel that provides a preemptive scheduler. Processes have associated priorities that are used to resolve contention for system resources.

---

[5] The price/performance relationship of the IBM System/370 series correlates well with a value of 1.6 for the cost exponent [SIEW82]).

[6] Note that in the case of the Grace algorithm, different performance processors might be used for the partitioning and joining nodes.

Resource Cost	Hybrid		Grace 1:1			Grace 2:1			Grace 1:2		
	#P	#D	#PP	#D	#JP	#PP	#D	#JP	#PP	#D	#JP
2	1	1									
3			1	1	1						
4	2	2							1	1	2
5						2	2	1			
6			2	2	2						
8									2	2	4
9			3	3	3						
10	5	5				4	4	2			
12									3	3	6
15			5	5	5	6	6	3			
16									4	4	8
20	10	10				8	8	4	5	5	10
21			7	7	7						
24	12	12							6	6	12
25						10	10	5			
30	15	15	10	10	10	12	12	6			

#P	- number of Processors
#D	- number of Disks
#PP	- number of Partitioning Processors
#JP	- number of Joining Processors

Table 1
Resource Costs for Hash-Join Configurations
(1 MIP Processors)

To minimize overhead, all disk transfers are done a track (28 Kbytes) at a time [GARC84]. In addition, a double buffer is associated with each open file so that while a process is processing track i, track i+1 can be read. The maximum packet size supported by the network is assumed to be 2K bytes.

The proposed multiprocessor join algorithms require that large blocks of data be transferred across the communications device. To this end, a model has been built of a modified, sliding window protocol that insures the reliable delivery of large blocks of data while enhancing the effective throughput of the network for such transfers. To help control contention for receivers, a higher-level, connection-based communications protocol has also been incorporated in the simulation model.

We have, so far, ignored the issue of memory size and bucket overflow. The preliminary results presented below assume that the smaller of the two relations being joined always fits in the aggregate memory of the processors used for joining. While this assumption will be true in a number of cases, one would not want to base the design of a machine on such an assumption. Assume, for example, that 8 processors are used for partitioning. With 64K RAMs, 1 megabyte memory boards are common. As 256K RAMS become available, a typical memory board will hold 4 megabytes of data. Thus with just 2 memory boards, an aggregate of 64 megabytes will be available for holding buckets. If one assumes that the smaller relation will be produced by applying a selection operation first, 64 megabytes might be enough to hold most temporary relations. We have also not addressed the issue of bucket overflow (the size of a bucket is larger than the memory of the joining processor to which the bucket is assigned) which can occur even in the case that the size of the smaller relation is less than the total available memory.

4.4. Preliminary Results

To evaluate the performance of the different algorithms, two 10,000 tuple relations were joined with each other. The result relation contained 10,000 tuples. For 1, 2, and 3 MIP processors, the network bandwidth and DMA rate were held constant while varying the resources available to the multiprocessor Hybrid join algorithm and the 3 configurations of the multiprocessor Grace algorithm: 1:1, 1:2, and 2:1. Throughput, measured in terms of queries per minute, was used as the performance metric. While we have conducted a wide range of tests, we have included only the results obtained using a network bandwidth of 80 Mbits/second and a DMA rate of 20 Mbits/second. A summary of the results obtained with other configurations is contained in Section 4.6. Since the cost of displaying or saving the result relation is the same for each configuration it has been ignored in the results displayed below.

The performance obtained by the multiprocessor Hybrid join algorithm and the 3 configurations of the multiprocessor Grace algorithm are displayed in Figures 5, 6, and 7 for 1, 2, and 3 MIP processors, respectively. We think these results are very exciting and represent a breakthrough in designing an architecturally simple, but very high speed database machine. For each type of processor, almost linear speedups are obtained with increasing level of resources.

With 1 MIP processors, the 1:1 Grace configuration provides a higher throughput rate over a wide range of available resources. The principal reason is that with 1 MIP processors, the average CPU utilization for the Hybrid design is almost 100%, whereas the 1:1 Grace design, which uses a larger number of processors per resource cost level, is not CPU bound. For example, when the total resource cost is equal to 6, the Grace design is using 2 partitioning nodes, 2 join nodes, and 2 disks. By comparison, for a resource cost of 6 the Hybrid design is using 3 processors and 3 disks. The average disk utilization was approximately 20% for the Hybrid design and 30% for the 1:1 Grace design. Thus, Hybrid's advantage of having the source relations partitioned over a larger number of disks for a given resource cost level was not a significant factor.

Figure 6 presents the throughput results for the case where all processors are assumed to be 2 MIP processors. In this test, the Hybrid processors are no longer CPU bound and the Hybrid algorithm outperforms all the Grace design combinations. The balanced nature of the processing requirements of this query favor the Hybrid and Grace 1:1 designs that allocate balanced processor resources. The Grace 2:1 and 1:2 designs perform less well because of lower processor utilizations resulting from the mismatch of processor resources. Figure 7 presents the throughput results when 3 MIP processors are used. The increased processor performance favors the Hybrid design which processes a bucket of each relation on the local processor. The Grace designs are not able to utilize the increased processor performance to the same magnitude as the network data transfers[7] become an impediment to increased performance.

4.5. A Look at Resource Utilizations with 2 MIP Processors

The performance of the multiprocessor hash-join algorithms necessarily depend on how well the algorithms utilize hardware resources. The Hybrid algorithm has the intrinsic advantage of sending a relatively smaller number of tuples across the communications network. On the other hand, the Hybrid algorithm imposes a greater load on each of the processors.

Figure 8 presents the resource utilization levels for the multiprocessor Hybrid algorithm with 2 MIP processors. The high CPU utilization levels reflect the fact that each processor in the Hybrid algorithm is used for both the partitioning and joining phases of the algorithm. The initial increase in CPU utilization is caused by the transition of the algorithm from using a single processor to using two processors. Whereas the single processor Hybrid design did not utilize the network at all, the two processor Hybrid design must expend a substantial amount of processing effort transferring buckets of tuples between processors. As additional processors are added to the Hybrid algorithm, the CPU utilization of the processors begins to decline. This decline corresponds to an increased level of contention for the network. As the level of contention for the network increases, processors are more frequently

[7] For a given query, the Grace designs must transfer a larger amount of data across the network than the Hybrid design.

blocked waiting to transfer blocks of data across the network. The increased levels of network contention also result in an increase in the total utilization of the network. The relatively low disk utilizations result from the fact that data is read from the disk a track at a time. With that disk I/O blocking factor, the disk is frequently idle while the previously read tuples are being partitioned. [8]

Figure 9 presents the resource utilizations for the Grace 1:1 multiprocessor algorithm design with 2 MIP processors. The relative CPU utilizations for the partitioning nodes and joining nodes reflect the fact that the partitioning phase is normally the most computationally expensive phase of the hash-join algorithm. The CPU utilizations of both the partitioning nodes and joining nodes decrease as the levels of network contention increase. The CPU utilizations of the Grace processors are relatively lower than the CPU utilizations presented for the Hybrid algorithm. This is due to the fact that for a given resource level, the Grace algorithm uses a greater number of processors than does the Hybrid algorithm. Conversely, the fact that the Grace algorithm uses fewer disks than the Hybrid algorithm for a given resource level leads to the relatively higher disk utilizations that are seen for the Grace algorithm.

4.6. Other Tests

Similar results were obtained when we varied the network bandwidth and the DMA rate. With a network bandwidth of 10 Mbits/second and a DMA rate of 4 Mbits/second (the slowest configuration tested), almost linear speedups were obtained up to approximately a resource cost of 20.[9] After this point, the network tended to become completely utilized and throughput remained constant.

We have, so far, chosen the same type of processors for the partitioning and joining nodes for the three alternative Grace designs. Join queries with varied distributions of join attribute values may provide the possibility of altering the balance of performance between the processing and joining nodes. We plan on investigating this alternative.

[8] Disk I/O blocking factors have been reduced to as low as 8 Kbytes without significantly altering the performance of the algorithms.

[9] The actual point varied with the MIP rate of the processors.

5. Conclusions and Future Research

In this paper, the hash-join algorithms presented in [DEWI84a] were extended to a multiprocessor architecture. As a first step, the algorithms described in [DEWI84a] were implemented using WiSS [CHOU83] running on a VAX 11/750 running 4.2 Berkeley UNIX. In addition to providing CPU and I/O utilization figures for use in the simulation of the multiprocessor algorithms, these centralized experiments provided two interesting results. First, the measured performance of the algorithms was very similar to that predicted analytically in [DEWI84a]. Second, bit vector filtering [BABB79] was shown to provide a dramatic reduction in the execution time of all algorithms including the sort merge join algorithm. In fact, for the one query tested, with bit-vector filtering all algorithms had virtually the same execution time.

We also extended the centralized Grace and Hybrid hash-join algorithms to a common multiprocessor configuration. These two centralized algorithms were chosen as they each provide a natural point for separating the joining and partitioning phases of the algorithm. The multiprocessor Hybrid algorithm uses a multiprocessor configuration consisting entirely of nodes having an associated disk drive. The nodes are used for both the partitioning and join phases of the algorithm. Three configurations of the multiprocessor Grace algorithm were evaluated: Grace 1:1, Grace 2:1, and Grace 1:2. In the 1:1 design one diskless joining processor is allocated for each partitioning processor. The 2:1 design allocates two partitioning nodes for each diskless joining node. The 1:2 design has one partitioning node for every two diskless joining nodes. The results from the simulation experiments of these algorithms is very encouraging as both algorithms provide linear increases in throughput with corresponding increases in processor and disk resources.

There are two interesting extensions to this research that we are currently exploring. This first is what we term **adjustable join parallelism**. By adjusting the partitioning algorithms, the number of buckets produced can be adjusted.[10] This in turn, effects how much parallelism can be used during the joining phase.[11] For example, if the partitioning phase produces just two buckets, than at most 2 processors can be used during the joining phase. There are a number of cases when such a technique might be useful:

(1) load balancing under heavy loads

[10] Constrained by the requirement that each bucket be reasonably be expected to fit into the memory of the joining processor.

[11] Equivalently, multiple buckets can be assigned to the same join processor. Since only one bucket will be active at any given time, the level of parallelism is controlled.

(2) low priority queries
(3) joins of small relations - Too much parallelism doesn't make sense if the relations being joined are small.

A second promising area is the use of **bit filtering** in multiprocessor hash join algorithms. There are a number of ways bit filtering [BABB79, KITS83] can be exploited by the multiprocessor hashing algorithms. For example, each joining node can build a bit vector simultaneously with the construction of a hash table. When completed, the bit vectors would be distributed to the partitioning processors. The partitioning processors could maintain the bit vectors on a per bucket basis. Alternately, the partitioning nodes might merge the per bucket bit vectors into a single bit vector. The bit vector(s) would then be applied during the partitioning of relation S. This strategy, plus a number of other bit vector filtering strategies, look promising.

Finally, we intend to use these algorithms as part of the Gamma Project. Gamma is a new database machine project that was begun recently. Gamma will provide a test vehicle for validating our multiprocessor hash-join results. Gamma will be built using the Crystal multicomputer [DEWI84b] and WiSS [CHOU83] as a basis. The Crystal Multicomputer project was funded as part of the National Science Foundation's Coordinate Experimental Research Program. Crystal is a network of bare VAX 11/750 processors (currently twenty, eventually forty) serving as nodes, connected by a 10 Mbit/second token ring from Proteon Associates [PROT83]. This ring is currently being upgraded to an 80 Mbit/second ring. Nine node machines have attached disks. File and database services are provided to Crystal "users" using WiSS. Crystal software provides a simple operating system (NOSE) with multiple, lightweight processes with shared memory and reliable connections to NOSE processes on other node machines and UNIX processes on the host machines (Vax's running 4.2 Unix). WiSS runs on top on NOSE. Crystal, NOSE, and WiSS are all operational and in production use.

6. References

[ASTR76] Astrahan, M., et. al., System R: Relational Approach to Database Management, ACM Transactions on Data Systems, Volume 1, No. 2, (June 1976), pp. 119-120.

[BABB79] Babb, E., Implementing a Relational Database by Means of Specialized Hardware, ACM Transactions on Database Systems, Volume 4, No. 1, 1979.

[BITT83] Bitton D., D.J. DeWitt, and C. Turbyfill, Benchmarking Database Systems - A Systematic Approach, Proceedings of the 1983 Very Large Database Conference, October, 1983.

[BLAS77] Blasgen, M. W., and K. P. Eswaran, Storage and Access in Relational Data Bases, IBM Systems Journal, No. 4, 1977.

[BRAT84] Bratbergsengen, Kjell, Hashing Methods and Relational Algebra Operations, Proceedings of the 1984 Very Large Database Conference, August, 1984.

[BROW85] Browne, J. C., Dale, A. G., Leung, C. and R. Jenevein, A Parallel Multi-Stage I/O Architecture with Self-Managing Disk Cache for Database Management Applications, Proceedings of the 4th International Workshop on Database Machines, March, 1985.

[CHOU83] Chou, H-T, DeWitt, D. J., Katz, R., and T. Klug, Design and Implementation of the Wisconsin Storage System (WiSS) to appear, Software Practice and Experience, also Computer Sciences Department, University of Wisconsin, Technical Report #524, November 1983.

[DEWI84a] DeWitt, D., Katz, R., Olken, F., Shapiro, D., Stonebraker, M. and D. Wood, Implementation Techniques for Main Memory Database Systems, Proceedings of the 1984 SIGMOD Conference, Boston, MA, June, 1984.

[DEWI84b] DeWitt, D. J., Finkel, R., and M. Solomon, The CRYSTAL Multicomputer: Design and Implementation Experience, submitted for publication IEEE Transactions on Software Engineering, also, Computer Sciences Department Technical Report #553, University of Wisconsin, September, 1984.

[GARC84] Garcia-Molina, H. and Kenneth Salem, Disk Striping, to appear Proceedings of the 1985 SIGMOD Conference, also Dept. of Electrical Engineering and Computer Science Technical Report #332, December, 1982.

[GOOD81] Goodman, J. R., An Investigation of Multiprocessor Structures and Algorithms for Database Management, University of California at Berkeley, Technical Report UCB/ERL, M81/33, May, 1981.

[GROS53] Grosch, H. R. J., High Speed Arithmetic: The Digital Computer as a Research Tool, Journal of the Optical Society of America, Volume 4, No. 4, April, 1953.

[HE83] He, X. et. al. The Implementation of a Multibackend Database Systems (MDBS), in Advanced Database Machine Architecture, edited by David Hsiao, Prentice-Hall, 1983.

[KIM85] Kim, M. Y, Parallel Operation of Magnetic Disk Storage Devices, Proceedings of the 4th International Workshop on Database Machines, March, 1985.

[KITS83a] Kitsuregawa, M., Tanaka, H., and T. Moto-oka, Application of Hash to Data Base Machine and Its Architecture, New Generation Computing, Volume 1, No. 1, 1983.

[KITS83b] Kitsuregawa, M., Tanaka, H., and T. Moto-oka, Relational Algebra Machine Grace, RIMS Symposia on Software Science and Engineering, 1982, Lecture Notes in Computer Science, Springer Verlag, 1983.

[KITS83c] Kitsuregawa, M., Tanaka, H., and T. Moto-oka, Architecture and Performance of Relational Algebra Machine Grace, University of Tokyo, Technical Report, 1983.

[KNUT73] Knuth, The Art of Computer Programming: Sorting and Searching, Volume III, 1973.

[PROT83] Proteon Associates, Operation and Maintenance Manual for the ProNet Model p1000 Unibus, Waltham, Mass, 1983.

[RIES78] Ries, D. and R. Epstein, Evaluation of Distribution Criteria for Distributed Database Systems, UCB/ERL Technical Report M78/22, UC Berkeley, May, 1978.

[SIEW82] Siewiorek, D. P., Bell, C. G., and A. Newell, Computer Structures: Principles and Examples, McGraw Hill, 1982.

[STON76] Stonebraker, Michael, Eugene Wong, and Peter Kreps, The Design and Implementation of INGRES, ACM Transactions on Database Systems, Volume 1, No. 3, September, 1976.

[VALD84] Valduriez, P., and G. Gardarin, Join and Semi-Join Algorithms for a Multiprocessor Database Machine, ACM Transactions on Database Systems, Volume 9, No. 1, March, 1984.