PARALLEL DATABASE SYSTEMS AND ITS ALGORITHMS

Amit A. Guled*

Dr. B.B. Meshram*

Abstract

Parallel database systems are becoming increasingly popular because of the throughput and scalability it provides. This paper describes the various ways to achieve parallelism in database. There are different parallel database architectures and shared memory and shared nothing architecture are most extensively employed in various applications. The execution of query can be controlled in either control flow way or data flow way. This paper describes the control flow approach and data flow approach. Large number operations that are performed in parallel database are sorting and joins. We illustrate various sorting and join algorithms and focus on hash based join algorithms which are most efficient. This paper also addresses the major issues such as overloaded processor and communication overhead. Finally we describe the declustering mechanism and performance tuning used to overcome the above issues.

***Keywords*— Parallel database, control flow, data flow, sorting, joins, declustering and performance tuning.**

* Department of Computer Technology, University of Mumbai

I. INTRODUCTION

Parallel database system [1,2] is extended implementation of parallel computers. Parallel database system exploits the parallelism in data management to deliver high performance and high availability database servers at a much lower price than equivalent mainframe computer.

Parallel processing exploits multiprocessor computers to run application programs by using several processors cooperatively, in order to improve performance. Parallel database combine the database management and parallel processing to increase performance and availability. The problem faced by conventional database management system is 'I/O bottleneck', induced by disk access time with respect to main memory access time. This resulted in poor performance of conventional DBMS. Thus the solution was to increase the 'I/O bandwidth through parallelism'. For example, if we store a database of size D on a single disk with throughput T, the system throughput is bounded by T. On the contrary if we partition the database across n disks, each with capacity D/n with same throughput, we get ideal throughput of n*T with n processors.

The objective of parallel database systems is to extend distributed technology, for example, by partitioning database across multiple small disks so that much inter- and intra-query parallelism can be obtained. This leads to improvement in response time and throughput. The research on parallel database was to support parallelization of query (such as SQL) execution in databases. Parallel database system can also be defined as DBMS implemented on a tightly coupled multiprocessor system.

Parallel database systems must have the following advantage over conventional database systems:

High-performance: Parallelism increases the throughput, using inter-query parallelism and decreases response time by intra-query parallelism several architectures for parallel database. But decreasing the response time of complex queries through parallelism can affect throughput. Therefore, it is necessary to optimize and parallelize queries in order to minimize the overhead of parallelism.

High-availability: Replicating data across multiple disks increases database availability. Probability of single disk failure is always in highly parallel database system. So, replication of data ensures that disk failure does not imbalance the load.

Extensibility: It is the ability of smooth expansion of the system by adding processing and storage power to the system. Parallel database system attempts to provide speed-up and scale-up to the conventional database system. Speed-up refers increasing number of processors and disks reduces the time to process a task whereas scale-up refers handling larger number of tasks by increasing the number of processor and disk. Extending system should require minimal reorganization of existing database.

II. RELATED WORK

A. Parallel database architecture:

There mainly three parallel database architectures.

1) Shared Memory:

All the processors share a common memory (Figure 1). This architecture is beneficial if fast communication among processors is required. This architecture is not scalable beyond 32/64 processor because adding more processor results in increase in memory contention.

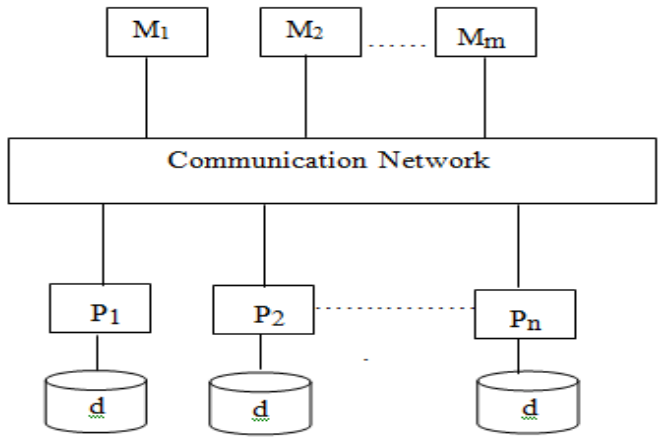

Figure 1. Shared Memory Architecture.

2) Shared Disk:

All the processors have private memory but share a common set of disks (Figure 2). Shared-disk systems are sometimes called clusters. It does not suffer from memory contention and is fault tolerant. But communication between processor is slower. Its scalability is slightly greater than shared memory architecture.

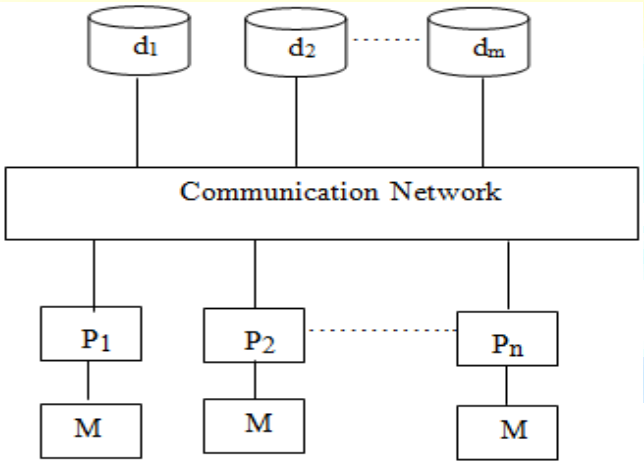

Figure 2. Shared Disk Architecture.

3) Shared Nothing:

The processors share neither a common memory nor common disk (Figure 3). Each processor have their own memory and disks. There is no memory contention. This architecture is fault tolerant and highly scalable. Only drawback is cost of communication and non local disk access is high. Gamma database [3] and prototyping Bubba[4] use this type of architecture.

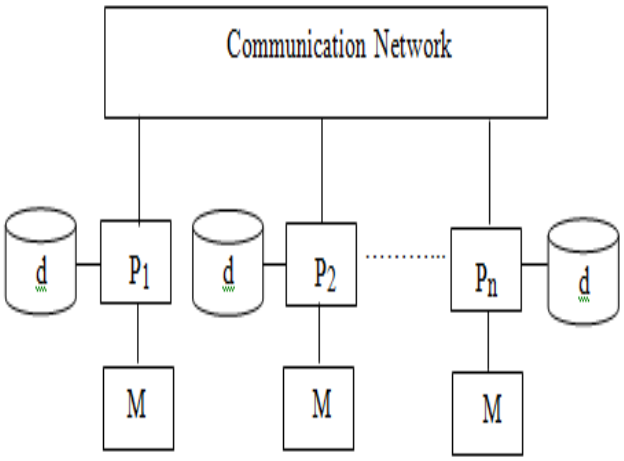

Figure 3. Shared Nothing Architecture.

B. Ways to achieve Parallelism:

There are two ways to achieve parallel: I/O parallelism and query parallelism.

1) I/O parallelism:

I/O parallelism [2] refers to reducing the time required to retrieve relation from disks by partitioning the relation on multiple disk. There are two ways of data partitioning: horizontal and vertical partitioning. In horizontal partitioning, tuples of the relation are divided among multiple disks such that each tuple resides on single disk. In vertical partitioning, columns of the relation are divided among multiple disk. In vertical partitioning, we need to take care that decomposition is lossless. To avoid this situation mostly horizontal partitioning is used.

Partitioning techniques:

1) Round Robin partitioning:

Every ith tuple is placed in disk $D_{i(mod)n}$ where n is number disks. Each disk have same number of tuple.

2) Hash partitioning:

Hash function is applied on chosen partitioning attribute(s). If hash function returns value i, then the tuple is send to disk D_i.

3) Range partitioning:

This strategy distributes contiguous attribute-value range to each disk. If tuple fall in attribute-value range assigned to disk D_i, then the tuple is send to disk D_i. Table I shows what type of queries are supported by each of partitioning techniques. The type of partitioning technique chosen affect other relational operations such as joins.

Table I. Comparison of Partitioning techniques.

Partitioning techniques	Sequential scan	Point Queries	Range Queries
Round Robin	Suited	Not	Not
Hash	Suited	Suited(on Partitioning attribute only)	Not
Range	Not	Suited	Suited

2) Query parallelism:

There are two types of query parallelism [2]:

Inter-query parallelism:

Different queries are executed in parallel with one another. This type of parallelism increases the throughput of the system. But this doesn't mean that response time decreases. Instead response time for a query executed in parallel is always greater than or equal to the response time when executed in isolation.

Intra-query parallelism:

There are two types of intra-query parallelism.

1) Intra-operation parallelism:

The execution of query is speed-up by parallelizing individual operation of the query, such as sort, select, project, ,join, etc.

2) Inter-operation parallelism:

The execution of query is speed-up by parallelizing the different operation of the query.

Both form of query-parallelism are responsible for improving the response time of the query.

C) Control flow vs Data flow:

The execution of query can be controlled in either control flow way or data flow way [5]. In first case, there is a central node (processor) responsible for entire execution of the query. In second case, each participating processors trigger each other to execute the query.

To illustrate these approaches consider a join query that joins two relation A and B. Relation A distributed over N_a nodes and relation B is distributed over N_b nodes and join is performed at N_c nodes. This approaches are illustrated in the context of shared nothing architecture.

1) Control flow approach:

If the example query is executed in a control flow way, a single node (the control node) is controlling the entire execution. It starts and synchronizes all processes on all nodes. Figure 4. shows the entire process.

1) A control message (start-msg) to start the selection is sent by the control node to the nodes that store relation A. The nodes reply with an acknowledgment message (start-ack), and send another acknowledgment message on having completed the operation (selection-ack). The former acknowledgement tells the control node that its message has been received and that processing has started. The latter acknowledgement is needed by the control node for synchronization purposes.

2) The same happens for relation B. This step can be executed concurrently with the previous one.

3) The nodes that store relation A are asked (distribute-msg) to distribute the selection result over the N_c, nodes the join has to be executed on. Again, acknowledgments are sent for receipt and completion. Each data message (data-msg) is answered with an acknowledgment for receipt as well.

4) The same happens for relation B. This step can be executed concurrently with the previous one.

5) After completion of the preceding steps, the control node asks the N_c, nodes to join the relation fragments they received (again using the same communication protocol).

6) Finally, after completion of the previous step, the control node asks the nodes that store relation C to execute the projection.

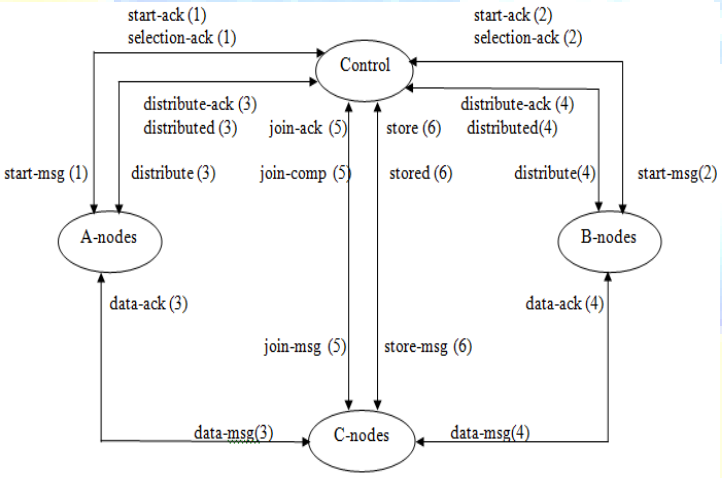

Figure 4. Query execution in Control flow way.

2) Data flow approach:

With a data flow execution strategy there is no central control node. The processes on the nodes wait for input messages to arrive, start execution if the input data are available, and send output messages to other processes at other nodes. Figure 5. shows the resulting messages for our example query.

1) The control node starts the selection on the nodes that store relation A. The nodes reply with an acknowledgment message for having received the control message.

2) The same happens for the nodes that store relation B.

3) After having completed the selection on relation A, the nodes distribute their result tuples over the nodes the join has to be executed on. Acknowledgments are sent for each data message.

4) The same happens for the nodes that store relation B.

5) As soon as the input relations are available, a join is performed on each of the N_c, nodes on which the result relation C will be generated. These nodes have to know what to do with a message when it arrives. We come to this later.

6) Finally, these N_c, nodes apply a projection on the join result and send a completion message to the control node.

D) Join Algorithms in parallel database:

In parallel database major operation performed are join operation. There are many join algorithms proposed in past but one of the most effective is the hash-based join algorithm [6]. There are three variants of hash based join algorithms.

1) Hashed-loop algorithm:

The algorithm is composed of two phases, which we call the build and join phases. The number of passes required in an execution of the algorithm is dependent upon the size of the outer relation R compared with the amount of available memory M. The number of passes can be approximated by the following:

A=ceiling(R/M).

A pseudocode description of the Hashed Loops algorithm is given in Figure. 5.

repeat A times do

fill up buffer space with R-tuples

BUILD:

for each R-tuple in the buffer space do

```
        hash join attribute of R-tuple to a hash table entry
        insert pointer to R-tuple into table entry
    end

JOIN:
    for each S-tuple in relation S do
        hash join attribute of S-tuple to a hash table entry
        probe that hash table entry
        if there is a match then do
            copy R-tuple and S-tuple to result buffer
        end
    end
end
```

Figure 6. Hashed Loop join algorithm.

During the build phase of the algorithm, read tasks read, in parallel, as much of R as will fit into the available buffer space. At the same time, a set of parallel build tasks create a hash table for this portion of R by taking the one data block at a time, hashing each tuple on its join attribute, and then inserting a pointer to that tuple into the appropriate location in the table.

In the second, or join, phase, the read tasks read all of S in parallel, into the available buffer space for S. In parallel, a set of join tasks take blocks of S as they become available, hash each tuple in the block on its join attribute, probe the hash table for matches, and then write any result tuples to an output buffer.

2) GRACE Algorithm:

The GRACE algorithm has four separate phases. The first two phases partition R and S into disjoint subsets, say, R_1 , R_2, . . . , R_B and S_1 , S_2 , . . . ,.S_B The value B is chosen such that each partition from R fits into main memory. The partitioning is done by hashing on the join attribute value, so that only corresponding partitions need to be considered for the join.

PARTITION:
for each R-tuple in R do
    hash join attribute of R-tuple to a partition Ri
    move R-tuple to Ri output buffer
end
for each S-tuple in S do
    hash join attribute of S-tuple to a partition Si
    move S-tuple to Si output buffer
end
for i = 1 to B do
    fill up buffer space with R_i-tuples
    perform BUILD for R_i,
    perform JOIN for Si
end

Figure 7. GRACE join algorithm.

After the partitioning phase, the Hashed-Loops algorithm is applied to the corresponding partitions. Some implementations of the algorithm proceed by creating many partitions, so that each partition will be small. Then at the next stage, as many partitions of R as will fit into memory are brought in for the hashed-loops stage. This technique is known as bucket tuning. A pseudocode description of the GRACE algorithm is given in Figure 7.

3) Hybrid Join Algorithm:

The Hybrid algorithm is a variation of the GRACE algorithm. During the partitioning phase, instead of using extra memory to increase the number of buckets, it introduces an R_0 bucket and builds a hash table for it. The relation R is therefore partitioned into B + 1 buckets, R_0, R_1,. . . , R_B. Bucket R_0 is kept in memory, and only partitions 1,. ..B are written out to disk. When S is partitioned, any tuples hashing to the corresponding S_0 are immediately joined with tuples in R_0 to create result tuples. The advantage gained is that tuples in the 0 buckets do not have to be written to disk and read in again for subsequent joining. After the partitioning phase is

over, a part of the join has already been computed. Depending on the amount of memory available and the size of the 0 bucket, the amount of I/O can be reduced substantially. As in GRACE, buckets 1,. . . , B are joined by using hashed-loops after the partitioning phase is over. Because joining begins from first phase itself, hybrid join algorithm is much faster than the other two algorithms. A pseudocode description of the Hybrid algorithm is given in Figure 8.

```
for each R-tuple in R do
        hash join attribute of R-tuple to a partition Ri
        if R-tuple belongs in R0 then do
            copy R-tuple to R0 buffer
             hash join attribute of R0-tuple to a hash table                entry
            insert pointer to R0-tuple into table entry
        end
        else do
            move R-tuple to Rii output buffer
        end
end
for each S-tuple in S do
        hash join attribute of S-tuple to a partition Si
        if S-tuple belongs in S0 then do
            hash join attribute of S0-tuple to a hash table        entry
            probe that hash table entry
            if there is a match then do
                copy R0-tuple and S0-tuple to result buffer
            end
            else do
                move S-tuple to Si output buffer
            end
        end
end
for i = 1 to B do
```

fill up buffer space with R_i,-tuples

perform BUILD for R_i

perform JOIN for S_i

end

Figure 8. Hybrid join algorithm.

E) Dynamic load balancing using partition tuning:

There are two major problems associated with hash-based parallel join algorithms:

1) Bucket Overflow:

In hash-based join algorithms, the size of each bucket should be smaller than the memory capacity. However, non-uniform distribution of the join attribute values occasionally generates bucket overflow, in which the sizes of the buckets exceed the memory capacity. The performance diminishes because it requires extra I/O to repartition the buckets into smaller fragments so that each will fit in the memory.

2) Skewed Tuple Distribution:

The performance of conventional parallel hash join algorithms relies on the randomizing hash function to redistribute the tuples of the join relations evenly across all PNs in the system. Their performance degrades when the join attribute values of the relations are non-uniformly distributed. That is, some processing nodes (PNs) have more tuples to process than the remaining PNs in the system. The concept of data skew is a phenomenon in which certain values for a given attribute occur more frequently than other values.

A partition is a set of hash buckets assigned to a PN. The term bucket in this refers to the tuples being hashed to the same bucket.

To overcome the bucket overflow problem, Hybrid Hash Join uses a second hash function, h2, to redistributes the overflow bucket between an in-memory hash table and overflow buckets on disk. GRACE hash join tries to avoid the bucket overflow problem by splitting the

relations into a large number of smaller buckets, and then these small buckets are combined into buckets to fit the memory capacity. Although these conventional parallel hash join algorithms can effectively resolve the bucket overflow problem, no mechanism is provided to handle the skew effect. The skew in tuple distribution reduces the degree of parallelism, resulting in degradation of the overall system performance.

The redistribution of the entire relations, when skewed distribution is resulted, is very costly. Alternative is to decluster the relation into smaller buckets using fine grain hash function so that these uneven hash buckets can be combined to form the balanced partitions for the PNs. This process is referred to as partition tuning in. A bestfit decreasing strategy can be used for partition tuning . In this scheme, the hash buckets are first sorted into decreasing order according to size. In each step, the currently largest bucket is assigned to the currently smallest partition (or PN). This process is repeated until all the buckets are allocated.

We know discuss three parallel hash join alogorithm that use partition tuning to balance the load.

1) Naive Load Balancing:

A naive approach to handle the skew in tuple distribution is to augment the conventional parallel hash join algorithms with an additional step to do load balancing. This algorithm work as follows:

1) Split phase: R and then S are hashed (partitioned) into a large number of buckets in parallel. Each bucket is statically allocated to a PN as in GRACE Hash Join algorithm. Each tuple in a bucket is collected to the corresponding PN through the interconnection network.

2) Partition tuning phase: This phase consists of three stages.

a) Bucket Sorting Stage: Each PN_i sorts its local bucket-pairs into descending order according to their sizes. The sorted bucket-pairs are then labeled as B_{i1}, B_{i2}, B_{i3}, ... That is,

$$B_{im} > B_{in} \text{ if } m < n.$$

b) Bucket Retaining Stage: Each PN_i retains ni of its larger bucket-pairs (i.e., having more tuples) such that

$\sum_{j=1}^{ni} |B_{ij}| < \frac{|R|+|S|}{N}$ and $\sum_{j=1}^{ni+1} |B_{ij}| > \frac{|R|+|S|}{N}$

The remaining buckets not retained in this stage are termed the excess buckets.

c) Bucket Relocating Stage: Each PN reports its current size and the sizes of the excess buckets to a designated coordinating PN. The coordinator then uses this information to reallocate the excess buckets to the undersize PNs using the best fit decreasing strategy illustrated. Once the destinations of the excess buckets have been determined, this information is broadcast to all the PNs, and the excess buckets are then physically collected by the undersize PNs accordingly.

3) Bucket Tuning Phase: Each PN combines the small buckets to form more optimal size join buckets.

4) Join Phase: Each PN performs the local joins of respectively matching buckets.

2) Tuple Interleaving Parallel Hash Join:

1) Split Phase: R and then S are hashed (partitioned) into a large number of join buckets in parallel. However, unlike GRACE, we decompose a join bucket, say R_i, into N sub buckets, R^1_i, ..., R^N_i (N is the number of PNs in the system), and each subbucket R^j_i is assigned to PN_j. During the partitioning process, each PN sends sends mth tuple of its R_i bucket to R^n_i where n = ((m-1)modN)+1. That is, the spreading is done by interleaving the consecutive tuples belonging to the same bucket among the PNs in the system. Since this spreading strategy guarantees that each bucket is spread evenly among PNs, the *N* sub-buckets of each bucket such derived should be uniform in size.

2) Bucket Tuning Phase: A predetermined coordinating PN can decide how to tune the size of buckets to fit the memory capacity based only on its local distribution of sub-buckets. The remaining PNs can then tune their sub-buckets accordingly as directed by the coordinator.

3) Partition Tuning Phase: The coordinating PN groups the matching bucket-pairs into N equal partitions using the best fit decreasing strategy. Each partition is then assigned to a distinct PN. The bucket-to-PN mapping information is then broadcast to all the PNs in the system. Each PN then forms its partition by gathering the remote sub-buckets as indicated in the mapping information.

4) Join Phase: Each PN performs the local joins of respectively matching buckets.

When the skew condition is mild, this strategy results in unnecessary communication and computation overhead. Therefore, to resolve this issue the algorithm is refined to obtain the following algorithm.

3) Adaptive Load Balancing Parallel Hash Join:

1) Split Phase: Each PN hashes its portion of each operand relation into considerably small sub-buckets. Each sub-bucket is stored back in the local disks.

2) Partition tuning: Each PN reports the sizes of its sub-buckets to a designated coordinating PN. For each hash bucket, the coordinator adds up the sizes of its sub-buckets to derive the size of the corresponding bucket. The coordinator then allocates the buckets to the PNs using the following strategy:

a) The matching bucket-pairs are sorted into descending order according to their sizes.

B) These Bucket-Pairs are then allocated to the PNs In the sorted order. For each bucket-pair, it is assigned to the PN With the Largest Matching Sub-Bucket Pair. That is, we retain the Largest Sub-Bucket-Pair at its resident PN, Say PN_i, and gather the smaller Sub-Bucket-Pairs from other PNs to form the corresponding bucket-pair at PN_i. The size of PN_i is then updated to reflect the addition of the new Bucket-Pair. When the size of some PN, satisfies the following condition:

$$\sum_{j=1}^{ni} |B_{ij}| < \frac{|R|+|S|}{N} \text{ And } \sum_{j=1}^{ni+1} |B_{ij}| > \frac{|R|+|S|}{N}$$

Where N, denotes the number of bucket-Pairs that has been assigned to PNr, It is disqualified from consideration for more bucket allocation. This iterative process continues until all the PNs become disqualified. At this time, the remaining buckets (if any) are assigned to the PNs using the Best Fit Decreasing Strategy. Once the assignment of the buckets to PNs is cmplete, the allocation information is broadcast to all PNs, and the sub-buckets are physically collected accordingly to their respective destination to form the corresponding local buckets.

3) Bucket Tuning Phase: Each PN combines the small buckets to form more optimal size join buckets.

4) Join Phase: Each PN performs the local joins of respectively matching buckets.

Conclusion:

In this paper we discussed different architecture for parallel database out of which shared nothing architecture provides the highest parallelism at some cost, i.e. each processor should have its own memory and disk. Then we discussed the various ways to achieve parallelism and noticed that to achieve small response time we need to parallelize the individual operations of the query. Operation that are mostly required in executing query are join operation. Among various join algorithm hash based join algorithm is best. Therefore, we discuss three hash based algorithm out of which hybrid join algorithm is most efficient. These algorithms do not solve the problem of skewed tuple distribution. To resolve this problem we saw partitioning tuning is the solution. Then we redefined the conventional join algorithms by augmenting it partitioning tuning phase and we obtain three algorithms: naive load balancing, tuple interleaving and adaptive load balancing parallel hash join algorithm. Among these algorithm is dynamic one because it redistribute the tuples only when the skew effect is above threshold.

REFERENCES

[1] 'Database System Concept' by Silberschatz-Korth-Sudarshan ,Fourth Edition .

[2] 'Principles Of Database System' by M. Tamer.Ozsu, Patrick Valduriez, Third Edition.

[3] D.J. DeWitt, *S.* Ghandehrinzadeh, D.A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, 'The Gamma database machine project," *IEEE Trans. Knowledge and Data Engineering*, vol. 2, no. 1, pp. 44-62, Feb. 1990.

[4] H. Boral, W. Alexander, L. Clay, G. Copeland, *S.* Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, a highly parallel database system," *IEEE Trans. Knowledge and Datu Engineering*, vol. 2, no. 1, pp. 4-24, Feb. 1990.

[5] 'Control Versus Data Flow in Parallel Database Machines' Wouter **B.** Teeuw and Henk M. Blanken, IEEE Transactions On Parallel And Distributed Systems, Vol. 4, No. 11, November 1993.

[6] Parallel Hash-Based Join Algorithms for **a** Shared-Everything Environment T. Patrick Martin, Per-&e Larson, and Vinay Deshpande, IEEE Transactions On Knowledge And Data Engineering, Vol. 6, No. 5, October 1994.

[7] Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning Kien **A.** Hua, ***Member,*** *IEEE,* Chiang Lee, and Chau M. Hua IEEE Transactions On Knowledge And Data Engineering, Vol. 7, No. 6, December 1995.

[8] 'A Scalable Sharing Architecture for a Parallel Database System' Vibby Gottemukkala Edward Omiecinski Umakishore Ramachandran College of Computing Georgia Institute of Technology, Atlanta, GA 30332. IEEE Transactions On Knowledge And Data Engineering, Vol. 8, No. 5, December 1991.

[9] An Adaptive Load Balancing Framework for Parallel Database Systems Based on Collaborative Agents.

[10] Hash-Based and Index-Based Join Algorithms for Cube and Ring Connected Multicomputers EDWARD R. OMIECINSKI AND EILEEN TIEN LIN. IEEE Transactions On Knowledge And Data Engineering, Vol. I , No. 3, September 1989.

[11] The Join Algorithms on a Shared-Memory Multiprocessor Database Machine. Ghassan Z.Qadah, IEEE Member and Keki B.Irani,Senior Member,IEEE. IEEE Transactions On Software Engineering, Vol. 14. No. I I , November 1988.

[12] Parallel Execution of Hash Joins in Parallel Databases Hui-I Hsiao, Ming-Syan Chen, Senior Member, IEEE, and Philip S. Yu, Fellow, IEEE. IEEE Transactions On Parallel And Distributed Systems, Vol. 8, No. 8, August 1997.