Database Operations in a Cube-Connected Multicomputer System

CHAITANYA K. BARU AND OPHIR FRIEDER

Abstract-Parallel architectures for database processing should provide parallel I/O as well as parallel CPU capability. Two issues that arise in such systems are data combination and nonuniform data distribution. Our recent interest has been in studying distributed memory architectures, specifically hypercubes, for parallel database processing. The cube interconnections support efficient data combination for the various database operations and nonuniform data distributions are handled by dynamically redistributing data utilizing these interconnections. Selection and scalar aggregation operations are easily supported. An algorithm for the join operation is discussed in some detail. A comparison of the cube and another multicomputer database machine, viz., SM3, is provided and the performance of the join operation in these systems is described. The join performance in a cube is comparable to that of SM3 even when the cube is assumed to have nonuniform data distribution.

I. INTRODUCTION

Early database machines (DBM's) were typically special purpose hardware aimed at relieving the well-known I/O and von Neumann bottlenecks present in "conventional" architectures (e.g., CASSM, DBC, and RAP 191, RELACS [3], etc.). More recent DBM's have been based on multiprocessorimulticomputer architectures, e.g., DIRECT 191, NON-VON [8], SM3[1], MIRDM [13], GRACE [II], GAMMA (41, and TERADATA [15]. While these systems use some

Manuscript received December 18, 1986; revised December 15, 1987. A preliminary version of this paper appeared in the IEEE COMPDEC 3rd International Conference of Data Engineering, Los Angeles, CA, February 2-6, 1987. This work was supported by the Rackham Faculty Grant, The University of Michigan, and the National Science Foundation under Grant IRI-8710855.

C. K. Baru is with the Advanced Computer Architecture Lab.. EECS Department, Ann Arbor. MI 48109.

0. Frieder is with Bell Communications Research, Morristown, NJ 07960.

IEEE Log Number 8926778.

special purpose hardware, they typically attempt to enhance database processing by employing parallel algorithms and general multiprocessing techniques. Parallel architecture for database processing should incorporate parallel CPU as well as parallel I/O capability. This gives rise to two important issues, data combination and nonuniform data distribution. First, an appropriate scheme is needed to combine data across parallel I/O paths. Second, relative immunity should be provided against the effects of uneven data distribution across paths. Different queries generate different patterns of distribution for intermediate results; **thus,** it is important to devise parallel algorithms which can perform well regardless of the distribution of data. This paper introduces strategies/algorithms that attempt to nullify the effects of intermediate data distribution by performing *dynamic* data redistribution.

A. Data Combination

The parallel IiO capability can be exploited by horizontally partitioning base relations into disjoint subsets and accessing them in parallel. Data from the parallel paths need to be combined in order to compute the final result for each of the relational and scalar aggregation (e.g., max, min. avg, sum, etc.) operations. A simple concatenation of tuples from parallel paths is sufficient in the select operation (assuming that output tuple ordering is unimportant). A similar data combination scheme would suffice for the join and project operations if there were no common join project attribute values across parallel paths. However, such a partitioning of data cannot always be guaranteed. Thus, for the join operation, data from each of N parallel paths need to **be** broadcast to the $N - 1$ other paths. For the project operation, data from the ith path, $i = N$ to 2, are broadcast to paths $i - 1$ to 1, where duplicate elimination is carried out. Scalar aggregation operations require data combination across all parallel paths.

B. Nonuniform Data Distribution

If the base relations are horizontally partitioned and distributed equally across the parallel I/O paths, then all paths take about the same amount of time to complete leaf-level operations (such as selection) in a query tree. For the remaining nonleaf operations, the distribution of data across parallel paths is dependent upon the previous operation(s) performed. As the query progresses, it becomes more difficult to predict and control these distributions. The concept of dynamic data redistribution, where intermediate data are redistributed on-the-fly. is introduced to handle this problem.

The rest of this paper is organized as follows. Section II discusses some recent trends in DBM design and also describes the assumed architecture of the cube system. The database operations select, scalar aggregate. join, and project along with the primitive operations tuple balancing and relation compaction/replication are described in Section III. Section IV provides the timing analysis and performance of the join operation in the cube system and SM3. Finally, a conclusion is provided in Section V. Part of the material presented in this paper has appeared in 121.

II. SOME RECENT DBM EFFORTS

Recent trends in DBM design have been toward the use of multiprocessorlmulticomputer architecture to support parallel database processing. Three recent machines that fit this description are GRACE [111, SM3 [1], and NON-VON [8]. GRACE is a multiprocessor DBM developed at the University of Tokyo which incorporates hardware sorters, hashing units, and other special purpose hardware to support database operations. The NON-VON is a multicomputer system designed at Columbia University. The system supports parallel 110 and can consist of as many as 1 million simple g-bit processing elements (SPE's) and up to one thousand large processing elements (LPE's).

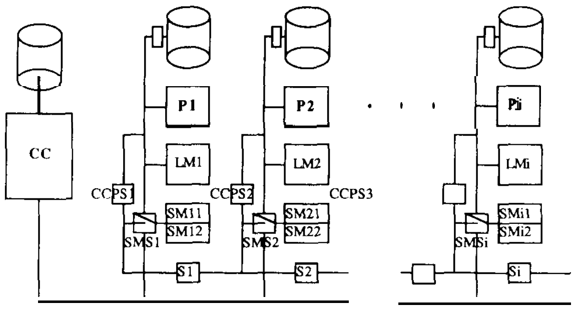

Fig. I. The switchable main memory modules system [1].

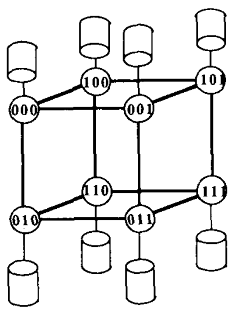

Fig. 2. Q_3 containing eight nodes with disk attached to each node.

A. The SM3 System

The Switchable Main Memory Modules (SM3) System is a dynamically partitionable multicomputer designed at the University of Florida. The SM3 system is included here since it provides parallel I/O capability as well as supports a variety of data combination schemes due to its partitionable bus. The SM3 consists of a number of independent computer systems connected by a physically partitionable common bus. The partitionable bus allows the system to be dynamically reconfigured into a number of *clusters* of adjacent computers thereby supporting MCMD (multiple code multiple data) processing. The SM3 processors (Fig. 1) are centrally controlled by a control computer (CC). The CC receives the queries, compiles them into command sequences, forms clusters based on the query requirements, and distributes commands accordingly. Each node contains a set of dual switchable memory modules which are used to facilitate fast data/command/message transfers among the processors and the clusters. Synchronization and interprocessor communication are supported via a number of special control lines. The partitionable common bus in SM3 can be used to support broadcast and tree-interconnection based algorithms in the system. The analysis in [1] assumes a 1 MIPS CPU and at least 4 x 13 kbytes of "switchable" memory at each node. The absolute parameter values are not important as long as the system performance is "balanced" (i.e., I/O, CPU, and data transfer rates are matched).

B. The Hypercube System

The cube system consists of N = 2" processors connected in the form of a Boolean n-cube. An n-cube or n-dimensional hypercube, Q_n, is defined recursively in terms of graphs as $K_2 X Q_{n-1}$, with $Q_1 = K_2$, where X denotes the Cartesian product of two graphs. Q_0 is a single node. Each of the N nodes is connected to n neighbors. The cube topology is attractive for two reasons. First, many interconnection topologies can be embedded in a Boolean n-cube. For example, embedding a tree aids in data combination for selection and scalar aggregation; embedding a ring is useful in the join operation; and embedding a chain is useful for the project operation. Second, multiprocessor systems based on the cube interconnection have been studied extensively for numeric computations. As numeric processing applications grow in size and nonnumeric (database) applications increase in complexity, such systems will need to support both numeric and nonnumeric processing requirements. Thus, it is interesting to study database processing on a cube. Several parallel computers based on the cube connection are currently in existence, including Caltech's Mark III [12], the Intel iPSC/II [10], the NCUBE/10 [7], the Floating Point Systems (FPS) T-series machines [5], etc.

The following assumptions are made regarding the architecture of the hypercube system. Each node in the system is assumed to have a CPU, local memory, secondary storage (disks), and n message/packet buffers, one for each neighbor. Fig. 2 shows a Boolean 3-cube consisting of N = 8 nodes with a disk attached to each node. Each node is also assumed to have two *tuple count registers,* TCR1 and TCR2, which are used to transfer small, fixed size fields between nodes. Very short, fixed size packets can achieve the same function in a general hypercube machine. However, assuming the existence of TCR's allows us to ignore the overheads associated with exchanging tuple counts in the analysis of Section III. Data are transferred among nodes via variable size packets with an upper bound placed on packet size. Any one of the cube nodes can be arbitrarily picked to be the host or controller node. However. most cube systems provide an external node as a controller and user interface node. A node has its own operating system, utilities, and programs. The nodes execute identical code but operate asynchronously and communicate with neighbors as dictated by the program. The horizontally partitioned base relations are stored as simple sequential files across the nodes of the cube (or subcube). No file access structures are assumed. The problem of optimal partitioning, distribution, and storage of data across the cube is part of our current work and is not addressed here. Also note that this paper discusses the performance of a single database operation in isolation and does not consider intra and interquery concurrencies.

III. Database Primitives

A. Selection

The select operation selects all tuples that satisfy some predicate. The command is broadcast from the single user or host node to all the other cube nodes in $n = \log_2$ N steps. Local selection at each node can be supported either in software or by using relatively inexpensive special purpose hardware (data filters). Collection of the local result is essentially a sequential operation and takes time proportional to the size of the output.

B. Scalar Aggregation

Scalar aggregation operations such as max, min, count, avg, etc., consist of a local and a global phase. In the local phase, each node computes the local aggregate value. In the global phase, the final aggregate value is computed by combining all the local values. The global aggregation phase takes n steps using recursive halving in the cube.

c. Join

Dynamic data redistribution is used in performing the join operation. A parallel version of the nested-loop algorithm is used here for the join. The sort-merge algorithm has also been studied and results obtained from a simulation study are summarized in Table II. In the simple, nested-loop algorithm, the tuples of the smaller (outer) relation are sent to all processors containing tuples of the larger (inner) relation. This is achieved by embedding a ring in the cube and transmitting the data of the smaller relation around the ring.

The best performance is achieved when both relations, R 1 and $R2$, are uniformly distributed across the nodes. While almost all past analyses have assumed a uniform data distribution, the algorithm presented here ensures such a distribution by performing *tuple balancing* as the first step of the algorithm. Once R 1 and $R2$ are balanced, the efficiency of the operation is increased by replicating one relation in multiple subcubes and embedding a ring in each to perform the same join. The relation replication is achieved in the second step of the algorithm using *relation compaction and replication* (RCR).

The smaller relation is compacted into as small a subcube as possible, based on the criterion explained in Section III-C-2, while it is simultaneously replicated across subcubes. Finally, in the third step of the algorithm, tuples of the smaller relation are transmitted around the ring embedded in each subcube and the local joins are performed in parallel at each node.

1) Tuple Balancing: A join typically operates on temporary relations (usually the outputs of a select, project, another join, etc.) whose data distribution across nodes is generally not known. Tuple balancing is employed to nullify the effects of such unknown distributions. For example, consider a 2-cube containing four nodes, as shown in Fig. 3(a). The nodes of the cube are associated with an n-bit address, as shown. Let $R1_i$ and $R2_i$ denote the number of tuples of $R1$ and $R2$ at node i, respectively ($0 \le i \le 3$). The tuple distribution of the two relations is as follows, $R1_0 = 5$, $R1_1 = 3$, $R1_2 = 2$, and $R1_3 = 1$ and $R2_0 = 3$, $R2_1 = 8$, $R2_2 = 9$, and $R2_3 = 7$. Tuple balancing proceeds in j stages ($1 \le j \le n$) where nodes that differ in address in the $(j - 1)$th bit balance tuples of Rl while, simultaneously, nodes that differ in address in the $k = (n \quad j)$th balance tuples of $R2$. Thus, each node simultaneously balances two relations. Assume that each fixed size packet can hold a maximum of six tuples. Fig. 3(a) shows the distribution of $R1$ and $R2$ before balancing. The $R1_i$ and $R2_i$ values are loaded into the TCR's of each node. Nodes exchange tuple counts and, for example, node 0 determines that it has to transfer 1 tuple of $R1$ to node 1 in order to make $R1_0 = R1_1 = 4$. Simultaneously, node 0 also determines that it should receive three tuples of $R2$ from node 2 to make $R2_0 = R2_2 = 6$. The distribution of tuples after the first stage ($j = 1$) is shown in Fig. 3(b) and the distribution after the second (and last) stage ($j = 2$) is shown in Fig. 3(c). The tuple balancing operation takes n steps. An additional step may be needed when n is odd and $j = k = (n + 1)/2$. The algorithm for balancing the tuple distribution of a relation $R1$ is shown in Fig. 4. The algorithm for balancing the second relation, $R2$, will be similar to this with R_1 being replaced by R_2 and $(i - 1)$ being replaced by $(n - i)$. If the input relations are already balanced, then the tuple balancing operation introduces a small overhead proportional to n.

After tuple balancing, if there is a clear difference in the sizes of the two relations, then each node can independently determine which of $R1$ and $R2$ is smaller. However, a problem may arise if $R1$ and $R2$ are nearly equal in size. Therefore, a software or hardware mechanism is used to select the smaller relation. In the software approach, two aggregation operations are performed to compute the total number of tuples of $R1$ and $R2$. The host node then knows which relation is smaller and broadcasts this information back in the cube. Alternatively, a global control line (AND or OR) can be used for this purpose. If an AND line is used, each node sets this line low to indicate that it has decided to choose, say, $R1$ as the smaller relation based on its local information.

2) Relation Compaction (RC) and Relation Compaction/Replication (RCR): One of the objectives of data redistribution is to balance CPU and communication times in the cube. The RC and RCR operations are used to aid in this. The RC operation compacts relations $R1$ and $R2$ so that the overall dimension of the cube used in the join is reduced. For example, let relation $R1$ contain t_1 tuples of size s_1 bytes each and relation $R2$ contain t_2 tuples of size s_2 bytes each. Let $t_1 \times s_1$ be less than $t_2 \times s_2$, i.e., $R1$ is smaller than $R2$. Assume that the nested-loop algorithm requires approximately five (VAX-like) instructions to determine whether a tuple of $R1$ joins with a tuple of $R2$. Let t_P represent the number of tuples of $R1$ per packet, t_L represent the number of tuples of $R2$ per node, and IPS represent the speed of the CPU in instructions per second. The time taken by each processor to join a packet-full of $R1$ tuples with its $R2$ tuples is then,

$$\text{CPU} = t_P \times t_L \times 5/\text{IPS}. \tag{1}$$

Let COM represent the transfer rate of the point-to-point interprocessor communication lines in bits per second and POvhd represent the overhead involved in packet assembly/disassembly. The time to

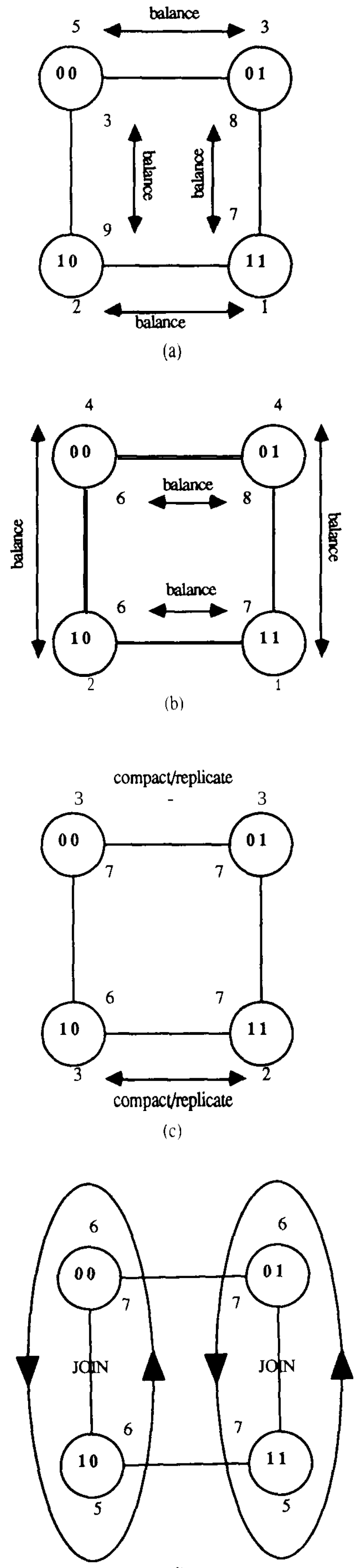

Fig. 3 An example of tuple balancing, RCR, and join of two relations.

Notation

n	dimension of the cube
P(R1)	maximum number of tuples of relation R1 that can be accomodated in a single packet
X	n-bit binary address of a node in the cube. Bits are numbered 0 to n- 1 from right (LSB) to left (MSB).
X^i	binary address which is identical to X except in bit i, $0 \le i < n$
R1(X) (or R 1 (Xi))	number of R1 tuples in node X (to be. read as "R1 of X")

(The following code runs on node X. The same code runs in parallel on all nodes of the cube.)

```
begin {tuple balancing}
    for i:= 1 to n do
        begin
          Read R1($X^{i-1}$); {receive tuple count of relation R1 from the appropriate neighbor}
          R1 avg := [R1($X^{i-1}$) + R1(X)] / 2;
          while (R1($X^{i-1}$) <> R1(X)) do
              begin
                  if (R1($X^{i-1}$) < R1(X)) then
                      begin
                          send-set := MIN ((R1_avg  R1($X^{i-1}$)), P(R1));
                                              [form set of tuples that need to be sent to neighbor]
                          send(send_set, $X^{i-1}$);
                                              { send send-set tuples to neighbor $X^{i-1}$ )
                          Mark and ignore the tuples sent for the rest of the operation;
                          R1(X) := R1(X) - | send-set |;  (decrement the local count of tuples)
                                                           (by the amount sent)
                      end
                  else
                      begin
                          receive  (receive-set, $X^{i-1}$);
                                              (receive receive-set tuples from neighbor $X^{i-1}$}
                          Add the tuples just received to local set of tuples;
                          R1(X) := R1(X) + | receive-set |; (increment the local count of tuples)
                                                           (by the amount received)
                      end
                  Read R1($X^{i-1}$);
              end {while}
        end {for}
end {tuple balancing}
```

Fig. 4. Pseudocode for tuple balancing operation for relation $R1$ in node X.

transfer a packet is then,

$$\text{Comm} = (t_P \times s_1 \times 8/\text{COM}) + \text{POvhd}. \tag{2}$$

The CPU and communication times will be balanced under the following condition:

$$t_P \times t_L \times 5/\text{IPS} = (t_P \times s_1 \times 8/\text{COM}) + \text{POvhd} \tag{3}$$

Ignoring the packet overhead for the moment,

$$t_L \approx (8 \times \text{IPS}/5 \times \text{COM}) \times s_1. \tag{4}$$

Both IPS and COM are hardware related parameters. The value 5 was obtained based on software/ hardware assumptions. Let

$$K = (8 \times \text{IPS}/5 \times \text{COM}) \tag{5}$$

then, $t_L \approx K \times s_1$. For example, if the CPU speed is 4 MIPS and the communication lines can transfer data at 8 Mbits/s, then $K = (8 \times 4 \times 10^6/5 \times 8 \times 10^6) = 0.8$ Since $t_L = t_2/N$ (tuplcs of $R2$ are equally distributed across all N processors after tuple balancing),

$$t_2 \approx N \times K \times s_1. \tag{6}$$

Equation (6) implies that if the number of tuples in the larger relation is less than $K \times N \times s_1$, then the CPU would be idling between packet arrivals. Conversely, if t_2 is greater than $K \times N \times s_1$, the CPU takes more time than communication. The optimal value for t_2 will be slightly more than $K \times N \times s_1$ due to the packet overhead that was ignored in (4). The RC operation can then be used, if necessary, to compact $R1$ and $R2$ into a subcube of the required size. The above equation for t_2 can be used to determine the size of the subcube needed for the join operation.

The RCR operation is applied only to the smaller relation, R 1. Relation compaction ensures that packets are as "full" as possible. Relation replication results in multiple subcubes containing a full copy of R 1. Therefore, a ring can be embedded in every subcube to increase the efficiency of the last and most costly step of the join operation (described in Section 111-C-3). The RC and RCR operations occur between pairs of nodes at a time. In the RC operation, **upon** completion, only one of the two nodes contains the multiset union of the tuples originally stored at each node. In the RCR operation, ***both*** the nodes contain the multiset union.

The RC and RCR operations take a maximum of n steps. In the jth step ($1 \le j \le n$), nodes that differ in address in the (j 1)th bit combine their tuples. The operation stops when either any pair of nodes cannot combine their tuples (i.e., the result of the multiset union is greater than the maximum packet size) or when $j = n$ (i.e., the entire relation has been compacted into a single node). For example, in Fig. 3(c), after tuple balancing each node decides that R 1 is the smaller relation. In the first RCR step, nodes are paired based on the LSB of the addresses, therefore, nodes 0 and 1 and nodes 2 and 3 form pairs. Each pair can combine tuples of R 1 into one packet (or less). Fig. 3(d) shows the tuple distribution after one step. Compaction paction must stop at this point since a packet can hold only six tuples in this example. The value of j at which the operation stops is denoted as k. In this example, k = 1. The justification for stopping the RCR operation is as follows. Assume that N nodes have one

```
Notation (see also figure 4)

AND(X)          Local AND line value in node X (to be read as "AND of X")
GLOBAL-AND      global AND line value across all the nodes of the cube

        (The following code runs on the node whose n-bit binary address is X.}
                (The same code runs in parallel on all nodes. }

begin {RCR operation)
  k:= 1;
  AND(X) := 1;                      {set local AND line in node X to 1}
  while GLOBAL-AND do
    begin
      Read R1(X^k)                  (receive the tuple count of relation R1 from neighbor)
        if (R1(X^k) + R1(X)) ≤ P(R1)
          then AND(X) := 1          (if multiset union of tuples fits in one }
        else AND(X) := 0            (packet, then send local tuples and receive }
                                    (remote tuples. Otherwise, set local AND line }
                                    (to 0 to indicate end of the RCR operation }

      if GLOBAL-AND then            (continue RCR operation)
        begin
          Send all R1(X) tuples to X^k;
          Receive tuples from X^k;
          Form the multiset union of local and received tuples;
          Adjust the value of R1(X)
        end
      k := k + 1
    end
end; {RCR operation)
```

Fig. 5. Pseudocode for the RCR operation.

packet each for a total of N packets. Compacting the relation further will halve the subcube size to N/2 and correspondingly double the number of packets per node to 2. Thus, there is no gain in communication time in the last step of the algorithm (since the number of packets sent = 1 x N = 2 x N/2). Also, since each node has finite memory, the RCR operation has to stop at some reasonable point unless the intermediate data are always written back to disk at each step. However, under this assumption, the size of memory at each node obviously affects performance.

A global AND line can be used to efficiently synchronize the termination of the RCR operation. The RCR operation proceeds as long as the AND line is high. The first pair of nodes that is unable to combine tuples sets this line low and the operation stops at this point. The RCR algorithm is shown in Fig. 5. For the RC operation, only one node in a pair sends data while the other node only receives data.

3) *The Join Step:* After the RC and RCR operations, the dimension n of the subcube containing $R2$ and the value k $(0 \leq k \leq n)$, where the RCR operation stopped for $R1$, are known. Subcubes are formed based on the $n - k$ least significant bits of the node addresses and a ring is embedded in each subcube. Thus, there are 2^k rings of 2^{n-k} nodes each, with each ring containing a full copy of $R1$. If no relation compaction was possible, then $k = 0$ and only a single ring containing all N nodes is formed. If $k = n$, then each node has all the tuples of $R1$ and maximum efficiency is achieved in the join step. A ring is formed by setting up a node sequence such that the addresses form a Gray code.

Since each node can send all its $R1$ tuples in one packet, it takes 2^{n-k} packet transfer time units (or, simply, time units) to circulate packets around the ring. If each node is able to join a packet of $R1$ tuples with its local segment of $R2$ in one time unit, then the join is performed in 2^{n-k} time units. For example, in Fig. 3(d), one step of compaction was performed. Nodes 0 and 1 combine their tuples resulting in each node having six tuples (one packet) of $R1$. Similarly, nodes 2 and 3 combine tuples resulting in five tuples in each node. Thus, in this example, $k = 1$ and $n - k = 2 - 1 = 1$. Therefore, in the join phase there are $2^k = 2$ rings of $2^{n-k} = 2$ nodes each, with each ring simultaneously performing the join. The RCR step takes one packet transfer and join takes $2^{n-k} = 2$ packet transfers, to give a total of three packet transfers. Without RCR, each node would have to send a packet in a ring of N nodes resulting in N = 4 packet transfers. The difference in performance obtained as a result of RCR is more dramatic in a cube of larger dimension. For example, in a 1024 node cube the corresponding numbers would be 513 (after one RCR step) versus 1024 packet transfers (without RCR), about a 50 percent improvement.

D. Project

A brief description of an algorithm to perform the project operation with duplicate elimination is provided here. This algorithm employs tuple balancing and relation compaction (RC).

```
PROJECT
1. Perform local Project (with duplicate elimination)
   at each node;
2. REPEAT
3.   WHILE (relation compaction is possible) DO
4.     Perform relation compaction and eliminate
       duplicates between local and incoming tuples;
5.   IF (subcube size > 1) THEN Perform tuple balancing
6. UNTIL (relation compaction is not possible);
7. IF subcube size > 1 THEN embed a chain in the subcube
     and perform a global duplicate elimination;
8. Results are available in the final subcube of size ≥ 1.
```

In the above algorithm, 1) if the subcube size is 1 then, by definition, relation compaction is not possible, 2) relation compaction is accompanied by duplicate elimination at each node, 3) the size of the cube containing the relation being projected reduces by half after every compaction step, and 4) the algorithm enters step 7 when no more compaction is possible.

IV. JOIN ALGORITHM ANALYSIS

This section analyzes the nested-loop join algorithm to highlight the merits/demerits of dynamic data redistribution. This, however, may not be the best algorithm to use for join. A simulation study showed better performance with the sort-merge algorithm, as indicated by the results tabulated in Table II which are discussed in Section IV-B. The nested-loop join algorithm example worked here

TABLE I
NESTED-LOOP JOIN TIME COMPARISON TO SM3

Relation R1: 3,000 tuples of 75 bytes each   Relation R2: 10,000 tuples of 100 bytes each
Output: 1,200 tuples of 175 bytes each

		SM3			CUBE		
	1 N	2 with data collection	3 without data collection	4 collection time	5 with collection	6 without collection	7 Balancing time
1	16	9460	8690	770	9615	9114	668
2	32	5971	4753	1219	5939	5359	835
3	64	4898	2784	2114	4308	3568	1002
4	128	5705	1799	3906	3842	2782	1169
5	256	8797	1629	7168	4139	2439	1336
6	512	-	-		5193	2633	1503
7	1024		-		7933	2813	1670

All times are in milliseconds

SM3 node: 1 MIPS CPU; at least 52 kbytes of switchable memory; IBM 3330 disk

Cube node: 1 MIPS CPU; at least 141 kbytes memory; 4 Mbits/sec lines; IBM 3330 disk

TABLE II(a)
CHARACTERIZATION OF INPUT DATA DISTRIBUTIONS
(b) SORT-MERGE JOIN TIMINGS IN A CUBE WITH AND WITHOUT TUPLE BALANCING

Data set Description	Maximum # of tuples in a node		Minimum # of tuples in a node	
	R1	R2	R1	R2
Random 1	284	1480	47	49
Random 2	401	1043	31	57
Random 3	347	1280	17	51
Random 4	459	1037	8	219
Random 5	426	1129	1	166
All in one node	3000	10000	0	0
All in first dimension	939	2395	0	0
All in last dimension	501	2058	0	0

(a)

Data set Description	Without Balancing ub	With Balancing b	% improvement (100*(ub-b/ub))
Random 1	293	252	14.6
Random 2	374	236	36.9
Random 3	358	247	31.0
Random 4	401	232	42.1
Random 5	382	239	37.4
All in one node	3950	637	83.9
All in first dimension	745	299	59.9
All in last dimension	505	272	46.1

Above timings are for a 16 node subcube with 64 kbytes maximum packet size, 4 MIPS CPU, and 20Mbits/sec communication lines

(b)

assumes a 1024 node system with 2 MIPS CPU's, 64 kbyte packets, and relations $R1$ and $R2$, each containing 64K tuples of 128 bytes each. It is assumed that an average of five instructions/tuple are required for the nested-loop join algorithm. From (3), the CPU and communication times are balanced if the point-to-point serial lines can transfer data at -6.8 Mbits/s.

Relation R 1 will be treated as the "smaller" relation. After balancing, each node has 64K/1024 = 64 tuples each of $R1$ and $R2$. This is already greater than the "optimum" for $R2$ (from (5), K = 0.47 and t_L = 0.47 x 128 $\approx$ 60) thus $R2$ need not be compacted any further. R 1, on the other hand, needs to be compacted thrice to give 512 R 1 tuples per node (t_p = 64 kbytes/128 bytes = 512). Thus, $R1$ is replicated in eight subcubes of 128 nodes each and the join is performed in parallel across all subcubes. The time to transfer a packet of 64 kbytes is 64K x 8/6.8 Mbits/s = 77.1 ms plus al assumed packet and synchronization overhead of 5 ms resulting in 82.1 ms (note that the 5 ms/packet overhead assumed here is relatively high). In the RCR operation, the packet sizes used in the first three steps are 8K (64 tuples), 16K, (128 tuples), and 32K bytes (256 tuples), respectively. The time taken in each step is 14.6, 24.3, and 43.6 ms, respectively, to give a total of 82.5 ms. Using (1), the CPU time to join a packet (5 12 tuples) of R 1 with 64 tuples of $R2$ is 512 x 64 x 5/2 x 10^6 = 81.9 ms. Therefore, the join performed in parallel in cubes of 128 nodes, takes 128 X 8.19 = 10 483 ms. The RCR time + join time = 82.54 + 10 483 $\approx$ 10.6 s. Tuple balancing is assumed to take a "worst case" $\log_2$ N = 10 steps with a full 64 kbyte packet being transferred in each step. Thus, the "worst case" tuple balancing time is 10 x 82.1 = 821 ms and the worst case join operation time is 0.821 + 10.6 $\approx$ 11.4 s.

A. Comparison to SM3

The timing equations that were derived for the SM3 system in [13] were used to compare the SM3 and cube algorithms. The two relations to be joined are R 1 and $R2$, where R 1 has 3000 tuples and $R2$ has 10 000 tuples. The values for SM3 are shown in columns 2, 3, and 4 of Table I. The values in column 2 assume no overlap between join processing and final result collection although, one can normally expect some overlap between these two steps. The CPU and data collection components are shown separately in columns 3 and 4, respectively. The values for SM3 are computed for N = 16, 32, 64, 128, and 256. Beyond N = 64, the data collection component determines the overall time.

Assuming the same query compilation overheads as in SM3, timings for the nested-loop join in a cube are computed and shown in columns 5, 6, and 7. The join operation time in column 5 is the sum of the time for (worst case) tuple balancing, RCR, local join, and output collection. Column 7 shows the worst case time for the tuple balancing operation. For 16 $\leq$ N $\leq$ 256, $R1$ is compacted into a four-node subcube ($\approx$ 752 tuplesinode) and the join is performed in 2^n, 2 $\leq n \leq$ 6, parallel subcubes, respectively. For N = 512 and 1024, $R2$ is compacted once and twice, respectively, and the join is performed as in the case when N = 256. Column 6 shows the join time excluding the output collection time. Collection is an important and time-consuming operation in database processing and is ultimately bound by the bandwidth of the final device (e.g., a user's terminal, disk, etc.) at which the result is collected. In SM3, the output is switched to the control computer (C) via the switchable memory modules and the operation is serialized at the CC.

The output collection time, in column 5, is computed based on the following assumptions. First, the result is collected at one of the processors involved in the final join step. While the join is being performed in parallel rings formed by the multiple subcubes, the result is collected along a single ring consisting of the nodes of all the subcubes. Second, every node produces the same amount of output. Thus, (N $-$ 1) packets need to arrive serially at the point of collection. Third, the collection step is not overlapped with processing. Ideally, some hardware support may be provided to overlap collection with processing. A better measure of relative system performance can be obtained by comparing the values in columns 3 and 6 which give the join times without data collection for SM3 and the cube, respectively. The cube times, including balancing are a little more than those of SM3. After subtracting of the balancing time component for the cube, the cube consistently performs better than SM3 due to the increased efficiency resulting from the multiple subcubes used in the last join step (note that the RCR operation time is included in this timing). The dynamically partitionable common bus structure of SM3 can be utilized to support dynamic redistribution. This problem is currently under study. A similar comparison as above has also been done for the NON-VON system and is available in [6].

B. Simulation

The effect of tuple balancing on operation performance was measured via simulation. However, the simulation employed a sort-merge join algorithm where R 1 and $R2$ are sorted after tuple balancing and the RCR operation preserves the sorted order. This results in a much faster join operation at each node. A 16 node subcube was assumed with 4 MIPS CPU's and 20 Mbit/s communication lines. These parameter values were chosen because they are closer to currently available technology. Eight different input data distributions were used, five random and three selected to represent highly skewed initial distribution. The relations $R1$ and $R2$ contain 3000 and 10 000 tuples, respectively. Table II(a) characterizes each of the eight different data distributions showing the maximum and minimum number of tuples at a node for Rl and $R2$, before balancing. Table II(b) shows the sort-merge join times with and without balancing and the corresponding improvement (computed as (unbalanced balanced)/unbalanced x 100). Note that when all the data are in a single node, SM3 should perform as well in dynamically redistributing the data.

V. Conclusion

This study was initiated to study issues in database processing in distributed memory multicomputers with a disk attached to each node. The particular architecture considered is a cube-connected multicomputer system. Nonuniform data distribution across parallel paths was mentioned as a source of inefficiency in such systems and some simple data redistribution schemes, tuple balancing, relation–compaction, and relation-compaction-replication, were suggested to handle this. Simple timing equations were used to obtain execution times for the join operation using the nested-loop algorithm and these were compared to the times in SM3. Results from a simulation to study the effects of tuple balancing were also reported. Many issues remain to be studied in this system. There is interest in studying hash-based join algorithms and data collection methods in the hypercube database system. Study of dynamic data redistribution in SM3 is of interest also. Issues related to intra- and interquery concurrency are currently being studied along with a further study of the effects of initial and intermediate data distributions.

Acknowledgment

The authors wish to thank the referees for their valuable comments and suggestions.

References

[1] C. K. Baru and S. Y. W. Su, "The architecture of SM3: A dynamically/partitionable multicomputer with switchable memory," *IEEE Trans.Comput.*, vol. C-35, Sept. 1986.

[2] C. K. Baru and 0. Frieder, "Implementing relational database operations in a cube-connected multicomputer," in *Proc. IEEE COMPDEC 3rd Int. Conf. Data Eng.*, Feb. 1987, Los Angeles, CA.

[3] P. B. Berra and E. Oliver, "The role of associative array processors in data base machine architecture," *IEEE Computer*, vol. 12. Mar. 1979.

[4] D. J. DeWitt *et al.*, "GAMMA-A high performance dataflow

database machine," in *Proc. Int. Conf. Very Large Data Bases,* Aug. 19086, Kyoto, Japan.
[5] K. A. Frenkel, "Evaluating two massively parallel machines," *Commun. ACM, vol. 29,* pp. 752-758, Aug. 1986.
[6] 0. Frieder, "Database processing on a cube-connected multicomputer," Ph.D. dissertation, Dep. EECS, Univ. of Michigan, Ann Arbor, 48109, Dec. 1987.
[7] J. P. Hayes *et al.*, "Architecture of a hypercube supercomputer," *IEEE Micro,* Aug. 1986.
[8] B. Hillyer, D. E. Shaw, and A. Nigam, "NON-VON's performance on certain database benchmarks," *IEEE Trans. Software Eng., vol.* SE-12, Apr. 1986.
[9] Special issue on Database Machines, *IEEE Trans. Comput.*, vol. C-28, June 1979.
[IO] Intel iPSC Data Sheet, Order No. 280101-001, 1985.
[1 1] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Architecture and performance of relational algebra machine GRACE," in *Proc. Int. Conf. Parallel Processing,* Aug. 1984.
[12] J. C. Peterson, J. 0. Tuazon, D. Lieberman, and M. Pniel. "The MARK III hypercube-ensemble concurrent computer," in *Proc. Int. Conf. Parallel Processing,* Aug. 1985.
[13] G. Z. Oadah and K. B. Irani, "A database machine for very large relational databases," *IEEE Trans. Comput., vol. C-34, Nov.* 1985.
[14] S. Y. W. Su and C. K. Baru, "Dynamically partitionable multicomputers with switchable memory," *J. Parallel Distribut. Comput.*, vol. 1, no. 2, 1984.
[15] Teradata. DBC/1012 Data Base Computer Concepts and Facilities, Release 1.3, June 1985.

Correction to "Line (Block) Size Choice for CPU Cache Memories"

ALAN JAY SMITH

The formula on page 1069 of the paper' should read as follows:

$$e\left(1+\frac{a}{b\quad 1+f^{*}\log \text{Csize}\quad \text{loglinesize}}\right)$$

$$.\ (1+c^{*}(\text{loglinesize}-1)\text{“}).$$

Also, a line should be inserted into the top (unified cache) section of Table IV which reads as follows:

4096 0.57 0.63 0.70 0.77 0.86.

My thanks to H. Stone of IBM, who discovered the error in the formula. The error was due to a transcription error, compounded by a typesetting error.

Manuscript received January 10, 1989.
The author is with the Computer Science Division, EECS Department, University of California, Berkeley, CA 94720.
IEEE Log Number 8926781.
A. J. Smith, *IEEE Trans. Comput., vol. C-36,* pp. 1063-1075, Sept. 1987.