Site and Query Scheduling Policies in Multicomputer Database Systems

Ophir Frieder, *Senior Member, IEEE*, and Chaitanya K. Baru, *Senior Member, IEEE*

***Abstract*— We study run-time issues, such as site allocation and query scheduling policies, in executing read-only queries in a hierarchical, distributed memory, multicomputer system. The particular architecture considered here is based on the hypercube interconnection. The data are stored in a *base cube*, which is controlled by a *control cube* and *host node* hierarchy. Input query trees are transformed into *operation sequence* trees, and the operation sequences become the units of scheduling. These sequences are scheduled dynamically at run-time. Algorithms for dynamic site allocation are provided. Several query scheduling policies that support interquery concurrency are also studied. Average query completion times and initiation delays are obtained for the various policies using simulations.**

***Index Terms*—Database systems, multicomputer systems, parallel database systems, query processing, scheduling, relational joins**

I. Introduction

DATABASE processing applications can benefit from the use of parallel processing techniques such as data parallel algorithms and intra- and interquery concurrency. The use of data parallel algorithms improves the execution times of individual database operations. Query response times can be reduced by supporting intra- and interquery concurrency or parallelism. A variety of special purpose database machines have been proposed to exploit the various levels of parallelism available in database applications. A survey of such machines is available in [37]. However, the growth in the number and variety of general purpose multiprocessor/multicomputer systems provides a justification to study the use of such systems for database processing. Various issues that need to be addressed in using multicomputer systems for database processing include the use of data parallel algorithms and parallel query execution strategies, data distribution/declustering techniques, query coordination and result collection, single-input, multiple-data (SIMD) versus multiple-input, multiple-data (MIMD) mode of operation, e.g., concurrency control, and logging and recovery.

Several parallel database system issues have been addressed by earlier database machine projects, such as DIRECT [11], GRACE [24], RDBM [1], BUBBA [8], and GAMMA [12]. DIRECT is a shared memory multiprocessor database machine in which the data from disks are staged into shared memory modules and the processors access the memory modules via a crossbar network. GRACE is also a multiprocessor database machine with a shared architecture. Data from disks are staged into several memory modules via a ring interconnection and reach the processors via a second ring interconnection. RDBM employs functionally specialized processors. Data are stored in a common disk subsystem, and the functionally specialized processors receive data from this subsystem. The processors communicate via shared memory. GAMMA and BUBBA are classified as *shared-nothing* multicomputers [38] where the system consists of several independent computers, each having its own memory, disks, and software. Thus, there is no sharing of either memory or disks and processors communicate via messages.

Manuscript received August 10, 1990; revised September 14, 1993. The work of O. Frieder was supported in part by the National Science Foundation under Grant CCR–9109804; the work of C. Baru was supported in part by the National Science Foundation under Grant IRI-8710855.

O. Frieder is with the Department of Computer Science, George Mason University, Fairfax, VA 22030 USA.

C. Baru is with IBM Toronto Laboratories, IBM Canada Ltd., North York, ON, Canada.

IEEE Log Number 9403084.

A. Database Processing on Hypercubes

Recently, several researchers have been studying the use of hypercube-based architectures for database applications [2]–[5], [9], [12], [13], [15], [17], [29], [31]–[33], [36], [39], [40]. An initial study of data redistribution and the nested-loop join algorithm in a hypercube system is reported in [2]. A comparison of the *nested-loop, global-sort,* and *global-hash*-based join algorithms is provided in [3]. Algorithms for *theta-join* computation are presented in [13], [15]. Index and hash-based join algorithms were studied in [31], [32], and the implementation of a variety of join algorithms on a JPL Mark III hypercube is reported by Upchurch *et al.* in [40]. An implementation of semijoin algorithms on a 16-processor hypercube is discussed in [33]. Implementation of join algorithms on an experimental 16-node hypercube and comparisons based on the Wisconsin benchmark are described in [9]. Information retrieval algorithms for a library database are discussed in [5]. DeWitt *et al.* report the implementation of GAMMA Version 2.0 on an Intel iPSC/2 containing 32 nodes [12]. Frieder *et al.* [17] experimentally evaluated the effects of data volume and distribution on the performance of a hypercube database engine. Other relevant work includes that of Pfaltz *et al.* [36], where a formal notation is derived for expressing database operators in a parallel environment and Topkar *et al.* [39], where several duplicate removal algorithms are presented and evaluated. Lakshmi and Yu [25] evaluate the overall effectiveness of parallel joins in the presence of data skew is presented. An approach to database processing based on logical E-R schema graphs is explored in [4], where lower bounds are derived for the *squashed* embedding of

E–R schemas in hypercubes. Several studies have also been conducted on file systems and input–output (I–O) architecture for hypercube computers [10], [19], [21], [42].

B. Site Allocation and Query Scheduling

Most of the work cited above is related to the study of individual relational database operations. However, in a parallel database system, it is also neccesary to devise strategies for the parallel execution of the various operations in a query and concurrent execution of several queries. We present the architecture of a run-time system that incorporates site allocation and query scheduling algorithms to support inter- and intraquery parallelism in a highly parallel, distributed memory multicomputer system. The run-time system architecture is based on simple, dynamic schemes that can operate effectively in the presence of frequent changes in the events and state of the computer system. For example, the site allocation and query scheduling strategies use only current system information rather than employing *a priori* optimization. This decision is based on the realization that it is difficult to accurately predict the times of future events in a highly parallel system. The effect of this strategy on different query scheduling policies is studied using simulated data sets and queries in a hypercube multicomputer system. The general site allocation and query scheduling methods described here are applicable to a wide range of distributed memory multicomputer systems. However, the system model and cost functions used here assume a hypercube multicomputer system.

Martin, Lam, and Russell [28] examined the site allocation problem. They defined queries by trees where nodes in the query tree represented base and intermediate relations. Using any of four suggested heuristic algorithms (branch and bound, greedy, simulated annealing, and local search), they identified the *best allocation* of the tree nodes onto the execution sites of the multiprocessor system. Liu and Chang [27] focused on the selection of a copy of a relation from within a set of replicated copies in a distributed environment. By enumerating all possible strategies, their algorithm selected the best copy in terms of communication and processing costs. Finally, Yu *et al.* [43] developed a heuristic approach to select the copy of choice within a replicated distributed database environment without the enumeration of all possible choices. Heuristic algorithms that approximate optimal solutions for a problem related to the mapping problem [6] are found in [6], [7], [16], [26].

Prior query scheduling schemes are limited and mainly deal with intraquery parallelism. Ganguly *et al.* [18] augment an annotated join tree to handle query execution in a parallel environment. Ioannidis and Cha Kang [23], to reduce scheduling complexity, schedule query fragments instead of individual join operators. Finally, Pramanik and Vineyard [34] optimize join queries in distributed environments by translating join graphs into semijoin graphs and focusing on the various components that comprise the execution of the query.

The remainder of this paper is organized as follows. Basic definitions and background material are provided in Section II. Section III discusses various issues in site allocation and the different cases that need to be considered for dynamic site allocation. Issues in query scheduling are discussed in Section IV. Four query scheduling policies are examined and simulation results are presented. Section V provides a conclusion.

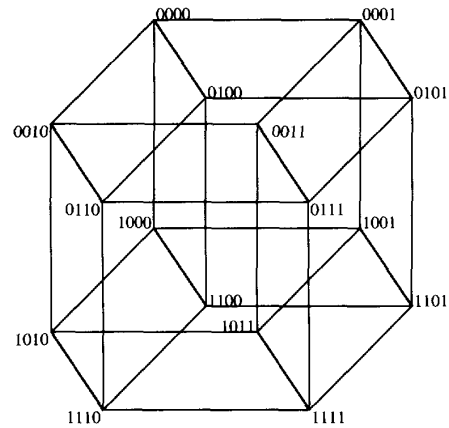


Fig. 1. A 4-D hypercube.

II. Background

A. System Architecture

The issues discussed herein are generally applicable to any distributed memory multicomputer system. Our results, however, are based on a hypercube multicomputer system. The architecture model and other details of the hypercube system are described here.

A hypercube system employs an n-dimensional Boolean cube interconnection scheme. The n-dimensional Boolean cube or Boolean n-cube, Q_n, is defined as a cross-product of the graph K_2 and the $(n-1)$-dimensional Boolean cube Q_{n-1}, with $Q_1 = K_2$. The graph Q_n contains $p = 2^n$ nodes, each of which is uniquely identified by an n-bit label or node address. Each node is connected (or adjacent) to n neighbors, such that the addresses of the node and each neighbor differ in a single bit. For example, in a 4-D cube, Q_4, node 0000 is adjacent to nodes 0001, 0010, 0100, and 1000. Fig. 1 shows a 4-D cube containing 16 nodes. Commercially available hypercube systems include Intel's *i*PSC/i860 and NCUBE's NCUBE/10 [20].

The multicomputer system is assumed to have a hierarchical architecture, consisting of a *base cube*, Q_b, *a control cube*, Q_c, and a *host node*, H, as shown in Fig. 2. Each node in this system is a complete computer system containing a central processing unit (CPU), memory, an intelligent disk controller, and communications processors (with buffers) associated with each communication link. The disk controller has direct memory access (DMA) capability to directly store data either in local memory or in one of the communication buffers. Within the base and control cubes, data transfer and communication among nodes can be achieved by using *packet-switching* or *cut-through* routing techniques. The internode communication

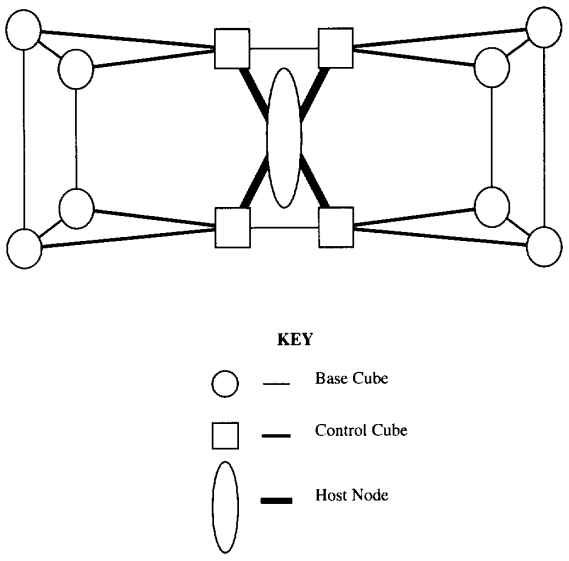


Fig. 2. Hierarchical cube multicomputer system.

time, t_{comm}, consists of a fixed startup overhead plus an incremental data transfer time per byte, i.e., $t_{\text{comm}} = \alpha + \beta x$, where α is the startup overhead, β is the incremental transfer time per byte, and x is the message size in bytes. Typical values for α and β are in the ranges 0.2–0.5 ms and 0.4–2.6 μs, respectively.

B. Initial Data Distribution

The database is stored in the base cube, Q_b, containing $p = 2^b$ nodes. Nodes in the base cube are logically partitioned into uniformly sized subcubes called *initial storage subcubes* (ISS's), of dimension $a < b$. Thus, there are 2^{b-a} ISS's in the base cube. Database relations are declustered (i.e., data in one relation are distributed over several nodes) across all the nodes of an ISS. The appropriate ISS size is selected based on parameters such as the I-O, CPU, and primary and secondary memory capabilites of each node, the size of relations in the database, and the number of relations in the database. The communication links at each node are divided into intra- and inter-ISS links. Relations are transferred in their declustered form across ISS's using the inter-ISS links.

All the nodes in an ISS are connected to a single *output collection node* (OCN) via a bus. All OCN's are in turn interconnected in the form of a hypercube, called the *control cube*, Q_c. The OCN's send commands and collect output to and from the ISS nodes. Finally, all OCN's are connected via a separate bus to the host node, H. The buses between the different levels are used for transfer of commands and results, not for executing the database operations. Thus, these buses allow for the overlapping of output collection with result computation.

In general, it is possible to employ ISS's of nonuniform sizes. In this case, data transfer algorithms need to be devised for transferring data between subcubes of unequal size. A similar situation also arises in a system with uniform ISS sizes if relations are allowed to be declustered across a *subset* of the nodes of the ISS. (In a hypercube system, the subset should form a subcube.) The problems of ISS size selection, initial data placement, and data transfer between subcubes of unequal size are not discussed here. Henceforth, each base relation is assumed to be contained entirely within a single ISS, with the tuples evenly distributed among the ISS nodes. This form of data distribution is referred to as *partial declustering* with *uniform degree of declustering*; i.e., each relation is spread across some, but not all, the nodes in the system (partial declustering), and every relation is distributed across the same *number* of nodes (uniform degree).

C. Parallel Algorithms

Let Q_a denote an ISS. As mentioned above, there are 2^{b-a} ISS's (or Q_a's) in Q_b. Since relations are declustered across the nodes of an ISS, one can employ data parallel algorithms within each ISS for the various database operations such as select, project, join, and scalar aggregation. Implementations of some of these operations in a hypercube system are described in [2], [3], [15]. Join algorithms that employ a *cycling* (or global nested-loop) algorithm and use either nested-loop or sort-merge algorithms locally, at each node, are introduced in [2]. Several other hypercube join algorithms have since been suggested in the literature, including schemes that employ broadcast communication, semijoins, indices, and hashing [31], [33], [40]. *Global* algorithms, based on *global-sort* (Hyperquicksort [41]) and *global-hash* schemes are presented in [3]. In the cycling schemes, the tuples of the smaller (outer) relation (R1) are sent to all processors containing tuples of the larger (inner) relation (R2) by forming a ring of processors within the ISS and sending the data of the smaller relation around the ring. In contrast, the global algorithms partition the tuples of both R1 and R2 across the set of processors, based on the join attribute values, using *recursive halving* schemes. The performance of the cycling and global algorithms is compared in [3], [15]. In general, the global algorithms were found to outperform the cycling algorithms. A review of various implementations of the join operation is provided in [30].

III. Dynamic Site Allocation

Query execution is initiated by the arrival of a compiled and optimized query at the host node, H. The output from the optimization step is a query tree containing parallel database operations at the nodes of the tree. This query tree is transformed into an *operation sequence tree* (OS-*tree*) as described below. The host node directs the necessary disk I-O commands to the relevant OCN's, which queue disk requests and broadcast them on a first-come-first-served basis to the ISS nodes. The disk I-O requests are accompanied by the qualifiers required for any selections that need to be performed on the base relations. Further operations in the query are scheduled at the nearest idle ISS. If k represents the number of inter-ISS links to be traversed from a given ISS to any other ISS, then the *nearest* ISS is the one for which k is a minimum. An *idle* ISS is one in which none of the node CPU's is executing a parallel database operation.

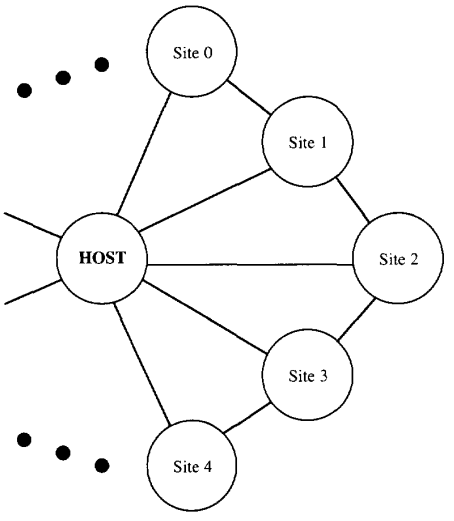


Fig. 3. The site-based model of a multicomputer system.

A. The System Model

For the purpose of site allocation and query scheduling, a hierarchical multicomputer system is modeled as a set of independent "sites" where each site has processing and I-O capabilities. Fig. 3 illustrates this site-based model of the system. The sites are connected together by a point-to-point ring interconnection, and each site is also connected directly to the central host, H, which makes the run-time scheduling decisions. A site may represent any combination of processing nodes and disks. In the hypercube multicomputer system, each ISS in the base cube corresponds to a site in Fig. 3, and the inter-ISS links correspond to the ring interconnection. The control cube and host node together correspond to the host node of Fig. 3. This site model is sufficiently general to accommodate a variety of other multicomputer systems as well.

The main memory at each site is assumed to be large enough to accommodate local segments of two input relations (i.e., the local segments obtained after applying any applicable search predicates), partial results being produced at the site, and buffer space for several disk blocks to allow for prefetching. The assumptions on the site memory size are justifiable, given the current rate of growth of main memory capacities. For example, the second-generation Intel hypercube, the *i*PSC/2, allows up to 16 megabytes of memory per node and 128 nodes (i.e., 2-gigabyte system memory). Thus, if the system was designed to have 16-node ISS's, then each ISS would have a main memory capacity of 256 megabytes.

B. Operation Sequences

An optimized query tree, in which each node represents a relational operator, is transformed into an OS-tree to aid in the scheduling of queries. The operation sequence is the smallest schedulable unit in a query and consists of an ordered list of one or more consecutive operations in a query tree where all the operations, except possibly the first, are unary operations. The first operation in the sequence may be a unary or binary operation. Operations at a lower level in the query tree appear first in the sequence. The length of an operation sequence is defined in terms of the number of operations in the sequence. Operation sequences can be of any one of three types, namely, *selection, unary,* and *join.* A selection operation performed on a base relation represents a *selection sequence.* The length of this sequence is always equal to 1. Since a selection sequence always operates on base relations, it is generally executed at the ISS that stores the corresponding base relation. A *unary sequence* is any sequence of unary database operations immediately following a selection sequence. The length of a unary sequence is greater than or equal to 1. Finally, a *join sequence* is a sequence in which the first operation is a relational join operation and the remaining operations, if any, are unary operations. Thus, a join sequence always has two inputs. The length of a join sequence is greater than or equal to 1. A join sequence in which at least one of the inputs is a selection sequence is called a *base join sequence.* Base join sequences are important in determining the number of resources (ISS's) required by a query, as discussed in Section IV. Both, unary and join sequences operate on intermediate relations.

Fig. 4(a) shows a query with three select operations, σ_1, σ_2, and σ_3; one projection operation, Π_1; two join operations, $\bowtie_1$ and $\bowtie_2$; an aggregation operation, α_1; and the "output" operation, O_1. Fig. 4(b) shows the corresponding OS tree. The three select operations are transformed into the three selection sequences, S_1, S_2, and S_3, respectively. The aggregation operation, α_1, is transformed into the unary sequence, U_1. The join operation, $\bowtie_1$, is combined with the subsequent project operation, Π_1 to form the join sequence, J_1; and the join operation, $\bowtie_2$, is combined with the output operation, O_1, to form the join sequence, J_2. Sequence J_1 is an example of a base join sequence.

Scheduling operation sequences instead of individual operations allows for the standard optimizations that can be obtained by combining consecutive unary operations in a query. It also reduces the amount of data movement during query execution and reduces the number of scheduling decisions that need to be made.

C. System Catalogs

The run-time system in the host node is required to maintain several data structures to handle intra- and interquery parallelism in the system. A sequence directory (SD) that stores all the information contained in an OS tree for each active (already scheduled) and inactive (to be scheduled) query in the system. Table I shows the typical entries in such a directory for three queries, Q1, Q2, and Q3. Query Q1 contains two selection sequences and one join sequence and operates on base tables R_3 and R_9, which are stored in ISS_1 and ISS_3, respectively. Queries Q2 and Q3 contain a selection sequence and a simple sequence. The inputs to these queries are R_7 and R_4, respectively.

Each entry in the SD catalog table has the following information. A unique sequence number (SEQ#); sequence type

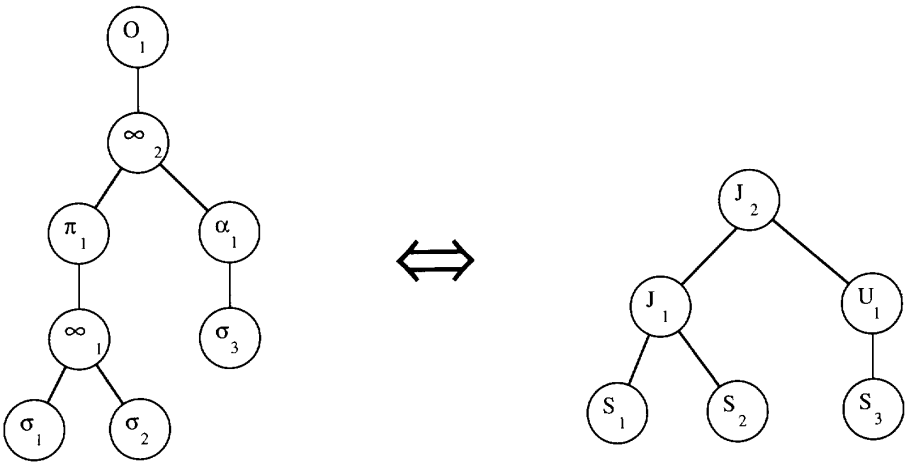


Fig. 4. (a) Example query tree. (b) Equivalent operation sequence tree.

TABLE I
THE SEQUENCE DIRECTORY

SEQ#	SEQ_TYPE	INIT_TIME	INPUT_1	INPUT_2	SUCC	ISS	ACTIVE
Q1-1	selection	102	R_3		Q1-3	1	1
Q1-2	selection	102	R_9		Q1-3	3	1
Q1-3	join	102	Q1-1	Q1-2	Host	3	1
Q2-1	selection	125	R_7		Q2-2	2	1
Q2-2	simple	125	Q2-1		Host	2	1
Q3-1	selection	915	R_4		Q3-2	-	0
Q3-2	simple	915	Q3-1		Host	-	0

(SEQ_TYPE); query submission/initiation time (INIT_TIME); input operand(s) required (INPUT_1, INPUT_2); the sequence number of the successor sequence (SUCC) or the entry "Host," if this is the last sequence in the query; the ISS on which the sequence was scheduled (ISS), if this is an active query; and an indication whether the query is active or inactive (ACTIVE). A 0 in the ACTIVE column indicates that a query that has entered the system has not yet been scheduled, whereas a 1 indicates that the query has been scheduled and is currently being executed. In Table I, query Q3 is yet to be scheduled.

The host node also maintains an ISS Activity Table, which provides the availability information for each ISS. Whenever an ISS becomes free, the next sequence that is ready to execute, if any, is scheduled from the sequence directory. A selection sequence is executable if the I-O controllers of the ISS containing the required base relation are free. A simple or join sequence is executable when the required input(s) becomes available. If there are no ready sequences and there are sufficient ISS's to initiate the next query in sequence, as determined by the current query scheduling policy, then the new query is activated. Once a query terminates, the corresponding row entries are logged and deleted from the SD.

D. Site Allocation Policies

Site allocation is performed at the time of query execution. This section describes the various possibilities that arise, at run-time, in allocating a site to perform the next operation sequence in a given OS tree.

Notation: The notation, **SS**, is used to denote a selection sequence, **US**, to denote a simple (or unary) sequence, and **JS**, to denote a join sequence. Let X, Y denote generic operation sequences, regardless of their type. For a given operation sequence, say, X, T_X represents its start time and Δ_X represents its estimated duration, i.e., estimated execution time. The variable s represents a processing node in the multicomputer system, and $\mathcal{S}$ represents the corresponding processing *site* (e.g., an ISS), which is a set of processing nodes.

The function *ExecSite*(X), returns the address of the execution site of X, and *BaseSite*(R) returns the address of the site at which the base table R is stored. For a unary OS, *Rel*(X) returns the base relation accessed by the OS. Similarly, *Rel1*(X) and *Rel2*(X) return the relations accessed by a binary OS. Function Z = *Slower*(X,Y) computes the quantity, MAX$(T_X+\Delta_X, T_Y+\Delta_Y)$, and returns the sequence,

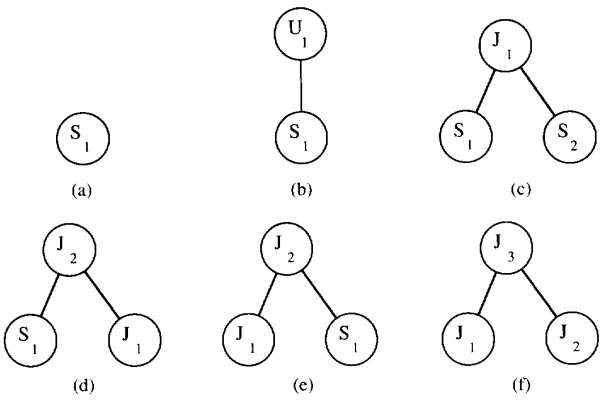


Fig. 5. (a)–(f) Six possible site scheduling cases.

X or Y, that terminates last. *Block*(X) returns true the first time a node, s, where $s \in$ *Site*(X), blocks because of buffers in s filling up. *Terminated*(X) returns true when *all* nodes in *Site*(X) have completed executing all the operations in sequence, X. *Scheduled*(X), returns true if the sequence X has already been scheduled at some site. T = *Nearest*($\mathcal{S}$) returns an idle site that is "nearest" to site, $\mathcal{S}$, in terms of communication distance (measured in terms of number of hops in a point-to-point communication network). The value returned by Nearest($\mathcal{S}$) could be $\mathcal{S}$ itself.

Six distinct types of nodes can be identified in a legal OS tree. Since a node in the OS tree represents a schedulable unit, these node types represent the six cases for which OS scheduling needs to be considered. Fig. 5(a) to 5(f) illustrate the various cases. Fig. 5(d) and 5(e) are different orientations of the same OS tree. The tree orientation is significant because the following discussion assumes that the left subtree of a node is always the first one to block or terminate.

When executing operation sequences, it is assumed that whenever possible, pipelined parallelism is employed between operations sequences and between database operations within a sequence. Thus, the individual steps within each algorithm, such as sorting, data redistribution, and hash table creation, are executed as pipelined steps. The OS scheduling policies attempt to maximize pipelined parallelism and the overlap between computation and communication. For example, let Z denote a join sequence. Let the two input operation sequences to Z be X and Y, and let X be scheduled on ISS_x and Y on ISS_y. If the result of X becomes available first, then Z is scheduled on an ISS, say, ISS_z, which is closer to ISS_y, and transfer of data is initiated from ISS_x to ISS_z. Also, the output data being generated at ISS_y is transferred to ISS_z. Choosing ISS_z close to ISS_y minimizes the impact on execution time of the data transfer between Y and Z once Y terminates. Although the transfer from X to Z may take longer, this time is overlapped with the processing of relation Y. Similarly, if the result of Y becomes available first, then Z is scheduled closer to ISS_x.

The join operation is initiated only after the data of both input operands have arrived at ISS_z. However, to support pipelined parallelism, the appropriate preprocessing operation such as data redistribution, sorting, and hash-table creation, is initiated as soon as the *first* unit of data of either input arrives at ISS_z. Data redistribution can be employed to reduce the join processing time in certain situations [3]. Local sorting can be started in the case of global sort-based join algorithms. Similarly, local hash table creation can be initiated in advance in the case of the global hash-based join algorithm. Initiating preprocessing steps in this manner allows for overlap between operation sequences and provides for pipelined parallelism. Site allocation for each of the six cases of OS scheduling is discussed in the following subsections.

Site Allocation—Case 1: Fig. 5(a) shows Case 1 of site allocation, where the operation sequence is an **SS** node. The **SS** node is always scheduled at the site of the base table. If S_1 denotes the **SS** node, then ExecSite(S_1) = BaseSite(Rel(S_1)).

Site Allocation—Case 2: The operation sequence in this case is a **US** node. Let U_1 be the label of the US node, and let S_1 be the label of its child SS node, if any. The following scheduling policy is employed.

if ((Block(S_1) or Terminate(S_1)) **then**
begin
Site(U_1) = Nearest(Site(S_1));
Route Output(S_1) to Site(U_1);
Start preprocessing at Site(U_1);
end

Site Allocation—Case 3: Fig. 5(c) shows an OS tree containing a JS node, J_1, whose children are two SS nodes, S_1 and S_2. The left subtree is assumed to block or terminate first. If the right subtree blocks or terminates first, S_1 can be replaced by S_2, and vice versa, in the following. The **JS** node is scheduled as follows.

if (Block(S_1)) **then**
  Site(J_1) = Nearest(Site(Slower(S_1, S_2)))
else if (Terminate(S_1)) **then**
begin
  Site(J_1) = Nearest(Site(S_2));
  Route Output(S_1) and Output(S_2) to Site(J_1);
  Start preprocessing at Site(J_1);
end

Site Allocation—Case 4: Fig. 5(d) shows an OS tree containing a JS node, J_2, whose left child is an SS node, S_1, and whose right child is a JS node, J_1. As before, the left subtree is assumed to block or terminate first. Node J_2 is scheduled as follows.

if ((Block(S_1) **or** Terminate(S_1)) **and** (**not** Scheduled(J_1)))
then
    Site(J_2) = Nearest(Site(J_1));
else if (Block(S_1) **and** Scheduled(J_1)) **then**
Site(J_2) = Nearest(Site(Slower(S_1, J_1)));
else if (Terminate(S_1) **and** Scheduled(J_1)) **then**
    Site(J_2) = Nearest(Site(S_1));
Route Output(S_1) and Output(J_1) to Site(J_2);
Start preprocessing at Site(J_2);

Site Allocation—Case 5: Fig. 5(e) shows an OS tree containing a JS node, J_2, whose left child is a JS node, J_1,

and whose right child is an SS node, S_1. The sequence J_1 is assumed to block or terminate first. Node J_2 is scheduled as follows.

if (Block(J_1)) **then**
begin
Create space in buffers of blocked node s, where $s \in$ Site(J_1)
(e.g., using redistribution and writing to disk);
Resume J_1;
end
else if (Terminate(J_1)) **then**
begin
Site(J_2) = Site(J_1);
Route Output(S_1) and Output(J_1) to Site(J_2);
Start preprocessing at Site(J_2);
end

Site Allocation—Case 6: Fig. 5(f) shows an OS tree containing a JS node, J_3, with two child nodes, J_1 and J_2, which are both JS nodes. Node J_1 is assumed to block or terminate first. If J_2 terminates or blocks first, J_1 can be replaced with J_2, and vice versa, in the following. Node J_3 is scheduled as follows.

if (Block(J_1)) **then**
begin
Create space in buffers of blocked node s, where $s \in$ Site(J_1)
(e.g., using redistribution and writing to disk);
Resume J_1;
end
else if (Terminate(J_1)) **then**
begin
Site(J_3) = Site(J_2);
preprocess Output(J_1) at Site(J_1);
When J_2 terminates:
co-begin
Preprocess Output(J_2);
Transfer preprocessed data of J_1 to Site(J_3);
co-end
end

A Dynamic Site Allocation Example: Site allocation for operation sequence execution is explained here via an example. As seen from the previous discussion, site allocation decisions are dynamic, except in the case of selection sequences. In general, specific sites are not reserved in advance when a query is activated.

Consider the OS tree of Fig. 4(b). The tree contains the sequences shown in Table II. The selection sequences, S_1, S_2, and S_3 operate on relations R_1, R_2, and R_3, respectively. Assume that the base cube of the multicomputer system contains four ISS's, numbered 0 to 3, as shown in Fig. 6. Suppose relations R_1 and R_3 reside on ISS_1, and R_2 resides on ISS_2. Thus, S_1 and S_3 are scheduled at ISS_1, and S_2 is scheduled at ISS_2.

In the example, assume that S_2 completes first (Case 3). The join sequence, J_1, is then scheduled near S_1. Assume that J_1 is scheduled on ISS_1 itself. The output of S_2 is then routed to ISS_1 from ISS_2. The available tuples of R_4 and R_5 are preprocessed at ISS_1, and J_1 is initiated when S_1 completes. S_3 is also initiated as soon as S_1 completes. After S_3 terminates, U_1 is scheduled at, say, ISS_3 (assume that ISS_1 is still busy). Fig. 6 shows a summary of the order in which the various operations sequences are performed at each ISS. The join sequence, J_2, is scheduled as indicated in Case 6. Suppose the first input sequence to terminate is U_1. The termination results in a message to the host node and the initiation of preprocessing operations at ISS_1 (= Site(U_1)). Now, when J_1 terminates, the preprocessed output of S_1 is sent from ISS_3 to ISS_1, and the join sequence J_2 is initiated at ISS_1. The final output is collected via the OCN associated with ISS_1.

TABLE II
EXAMPLE OF OPERATION SEQUENCES IN A QUERY
(to be determined by the site allocation policies (TBD))

Sequence	Type	Operator List	Inputs	Executed At
S_1	Selection	$\{\sigma_1\}$	R_1	ISS_1
S_2	Selection	$\{\sigma_2\}$	R_2	ISS_2
S_3	Selection	$\{\sigma_3\}$	R_3	ISS_1
J_1	Join	$\{\bowtie_1, \alpha_1\}$	R_4, R_5	TBD
U_1	Unary	$\{\Pi_2\}$	R_6	TBD
J_2	Join, Output	$\{\bowtie_2, O_1\}$	R_7, R_8	TBD

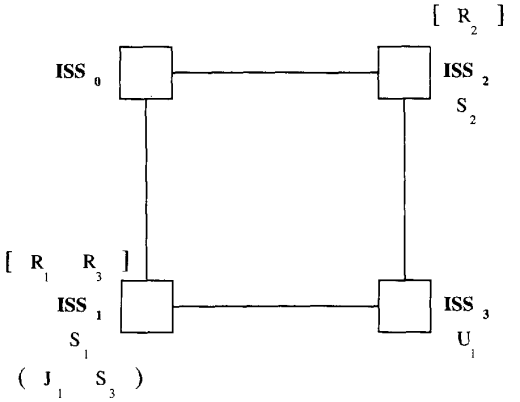


Fig. 6. Scheduling example using a hierachical multicomputer system with four ISS's.

In the above example, if S_1 had terminated first instead of S_2, then J_1 would have been scheduled at Nearest(Site(S_2)), say, at ISS_2 itself. Preprocessing of R_4 and the available tuples of R_5 would have begun at ISS_2 and S_3, and U_1 would have been executed at ISS_1. In this case, when J_1 terminates, data would be tranferred from ISS_1 to ISS_2, and J_2 would be executed at ISS_2. In general, since it is difficult to compute the exact completion times of each OS, it is also difficult to statically determined the best site for executing each operation sequence in a given query. The problem becomes even more difficult when we allow interquery concurrency. Thus, the site allocation heuristics dynamically choose the best available site at the given point in time.

E. Determining the Resource Requirements of a Query

As mentioned earlier, each OS in the OS tree represents a minimum schedulable unit. An OS is associated with the

CPU, I–O, and communication costs required to execute the individual database operations in that sequence. Thus, each OS is associated with certain "resource requirements." To ensure that a query is executed to completion without resource deadlocks, the total resource requirements of the corresponding OS tree must be determined. Next the query scheduling policies must ensure that the system can satisfy the resource requirements of the query to be scheduled, given the resource allocation policies. Query resource requirements are determined as described in the following section.

The Active Sequence Count (ASC): The resource requirements of a query are determined by the *active sequence count (ASC)* of each query. The ASC is defined as follows.

Definition 3.1: The ASC of a query is given by the sum of the number of base join sequences in the query plus the number of unary sequences in the query.

Definition 3.2: A base join sequence is a join sequence in which at least one of the inputs is a selection sequence.

The dynamic site allocation policies described in Section III-D provide an upper bound on the number of sites required to process a query that is equal to the ASC of the query. This result can be obtained by considering each of the six types of OS tree nodes mentioned in Section III-D.

Lemma 4.1: A selection sequence does not contribute to the ISS resource demands of a query.

Proof: In optimized queries, selection sequences are assumed to be executed at the node I-O processors. If there are several selection sequences to be executed at a node, the requests are queued using a standard queueing discipline, e.g., first-in, first-out. If a node gets *blocked* during selection, i.e., if the buffers become full, then an ISS is found for the next operation, and data are routed via the communications processors and inter-ISS communications links. Thus, the node CPU's are not employed in processing this sequence. Since the node CPU and intra-ISS communications resources are not employed in processing this OS, the number of sites needed for this OS is defined to be equal to 0.

Fig. 5(a) shows an OS tree containing an SS node, From Definition 3.1, the ASC of this tree is equal to 0.

Lemma 4.2: A unary sequence contributes one ISS to the ISS demands of a query.

Proof: First, unary sequences can have only selection sequences as inputs. If they had a join sequence as input, the operations in the sequence would be combined with those of the preceding join sequence to obtain a new join sequence. As mentioned above, the selection sequence is scheduled on the I–O processors of the appropriate site. When, this sequence blocks or terminates (Case 2 in site allocation), it is necessary to assign a site for the US node. Thus, the number of sites required for unary sequences is 1.

Fig. 5(b) shows an OS tree containing an SS node and a US node. By Definition 3.1, the ASC of this tree is 1.

Lemma 4.3: A join sequence contributes one ISS to the ISS demands of a query only if it is a base join sequence.

Proof: Consider the four site allocation cases, namely, Cases 3–6, in Section III-D. Cases 3 and 4 require an ISS to be identified to execute the upcoming join sequence, as soon as the left-hand input blocks or terminates. However, Case 5 does not require a new ISS for executing the join sequence, because the join sequence is scheduled at the site of the left-hand input itself. However, note that Cases 4 and 5 represent the same type of join sequence, namely, one that has a simple/join sequence as one input and a selection sequence as the other input. Thus, in terms of ISS requirements, Case 4 represents the worst-case situation for such a join sequence. Finally, Case 6 does not require a new ISS to be allocated for the upcoming join. Thus, only Cases 3 and 4 require a site to be available in order to schedule the subsequent join sequence. Since the join sequences in Cases 3 and 4 represent base join sequences, the above lemma is proved.

By Definition 3.2, the OS tree in Fig. 5(c), 5(d), and 5(e) will be assigned an ASC equal to 1, whereas the OS tree in Fig. 5(f) will be assigned an ASC equal to 0. From Lemmas 4.1 to 4.3, it is clear that Definition 3.1 correctly specifies the resource requirements of a query, except in the case of Case 5 of Section III-D, where the definition results in an overestimation.

IV. Query Scheduling Policies

When executing multiple queries concurrently, the host computer employs a policy by which to choose the next query to be scheduled. Queries are selected for execution based on their resource requirements, which is indicated by their ASC's. The sum of all the ASC's of all currently active queries is the total active sequence count (TASC). Thus, the number of available ISS's in the system at any time is, AVAIL equals the total number of ISS's in the system, or the TASC. A query is selected for execution only if its ASC $\leq$ AVAIL. All queries are assumed to have the same level of priority.

A. Indefinite Wait and Deadlock Prevention Mechanisms

The query selection policies must ensure that both system deadlock and indefinite wait conditions are avoided. Indefinite waits are prevented by time-stamping queries with arrival times. A query is forced to the head of the query queue if its waiting time exceeds a given limit. Once the resources required to execute this query become available, the aged query is initiated.

System deadlock is avoided by limiting the number of active queries in the system. For example, a deadlock may arise when processing, say, an OS tree that has the structure of a degenerate binary tree, if all later join sequences are scheduled on the available ISS's and the first join sequence is unable to execute because of the unavailability of ISS's. Such deadlocks are prevented by maintaining a running TASC count and ensuring that the TASC never exceeds the number of ISS's in the system. The following TASC update policy is used in the control node.

TASC Incrementation: The TASC is incremented by the ASC count whenever a new query is scheduled; i.e., TASC := TASC + (ASC of the new, active query).

TASC Decrementation: The TASC is decremented by one either (1) when a query containing at least one simple or join sequence terminates, or (2) whenever the second input to a join sequence terminates, provided that it is not a selection sequence.

B. Simulation Study

The performance of the various query scheduling policies was studied via simulations. The simulation program incorporates the various site allocation policies described in Section III. The input to the simulation is a set of base relations and queries, and the output consists of the *global* and *average query completion* times as well as the *average query initiation delay* for all the queries in each query mix. The *global query completion* time is the time interval from the arrival of the first query until the completion of the last query. The *average query completion* time is the average of the individual completion times of all the queries in the set, and the *average initiation delay* is the average of the time each query in the set must wait from its arrival time to the time it is actually scheduled.

The simulation assumes a multicomputer system consisting of a 1024-node base cube divided into 64 ISS's of 16 nodes each. The parameter values chosen were similar to those of the NCUBE hypercube system [20]. For example, each node contains a 2-MIPS CPU with internode communication handled by communications processors using 20-Mbit/s communication links. A maximum limitation of 64 kilobytes is imposed on the packet size. A total of 100 different base relations are generated, with each relation consisting of between 2000 and 10 000 tuples. All tuples are of 128 bytes, and the actual number of tuples in each of the 100 relations is randomly generated. The relations are distributed across the 64 ISS's such that each of the 36 lower-numbered (0–35) ISS's store two relations, and the remaining 28 ISS's store only one relation each. Within each ISS, tuples of relations are uniformly distributed across all the nodes.

Generating an appropriate or representative query mix is important to the outcome of the experiments. The objective of the simulation here is not to tune the system performance for a particular type or class of workload. Rather, the intention is to study the behavior of the different query scheduling policies for the same set of random queries. A query mix containing eight types of queries of varying complexity is chosen. The base relations accessed by a query are determined by generating random numbers that link queries to relations. The query mix consists of 200 queries distributed among the eight different types as shown in Table III. For each query type, the frequency of its occurrence in the mix, the height of the query tree, h (root is at level 1), and the active sequence count, ASC, are shown. The following four query scheduling policies are investigated:

1) The maximum active sequence count (MASC) policy, which selects the query with the greatest ASC first;
2) The least active sequence count (LASC) policy, which selects the query with the least ASC first;
3) The tallest query tree first (TQTF) policy, which selects the query with the greatest height first; and
4) The shortest query tree first (SQTF) policy, which selects the query tree with the smallest height first.

Results from running the simulation on 10 different data sets, i.e., the same set of queries running against different sets of base relations, are presented in Table IV.

TABLE III
QUERY MIX USED IN SIMULATION

Type	Query Tree Description	Frequency	Height, h	ASC
1	Single select operation	60	1	0
2	Single join operation	50	1	1
3	Linear join tree with two joins	35	2	2
4	Linear join tree with three joins	15	3	3
5	Linear join tree with four joins	12	4	4
6	Linear join tree with six joins	8	6	6
7	Full binary tree with three joins	12	2	2
8	Full binary tree with seven joins	8	3	4

TABLE IV
SIMULATION RESULTS
(all times in ms)
GQCT=global query completion time
AQCT=average query completion time
AQID=average query initiation delay

Data Set		MASC	TQTF	LASC
Data Set 1	GQCT	12571	12571	14075
	AQCT	2803	2712	1157
	AQID	1705	1924	156
Data Set 2	GQCT	6346	6346	7561
	AQCT	2551	2526	1070
	AQID	1596	1814	160
Data Set 3	GQCT	14840	14840	16501
	AQCT	3241	3135	1289
	AQID	1961	2210	154
Data Set 4	GQCT	6520	6520	8930
	AQCT	2330	2288	1000
	AQID	1442	1630	148
Data Set 5	GQCT	4957	4957	7031
	AQCT	2298	1709	995
	AQID	1524	1709	149
Data Set 6	GQCT	6956	6956	7909
	AQCT	2497	2467	1010
	AQID	1638	1819	138
Data Set 7	GQCT	9746	9746	11546
	AQCT	2924	2834	1209
	AQID	1792	2020	147
Data Set 8	GQCT	10529	10529	11721
	AQCT	2885	2840	1196
	AQID	1786	2021	168
Data Set 9	GQCT	5704	5704	6468
	AQCT	2586	2533	1021
	AQID	1633	1885	151
Data Set 10	GQCT	7337	7337	8178
	AQCT	2204	2151	1054
	AQID	1334	1509	151

Since queries are generated randomly, each data set represents different base relation access requirements. For the above mix of queries, the LASC and SQTF policies provide identical results for reasons explained below. Thus, the table shows only the results for the MASC, LASC, and TQTF policies. The MASC and TQTF policies behave identically for linear OS tree queries, because in this case, the queries with the maximum ASC are also the ones with the greatest height. The global query completion time, average individual query completion times, and the average query initiation delay are computed for each data set. The 10 data sets cannot be compared against one another, because they represent different base relation access requirements; but certain characteristic patterns are common to all. For example, LASC and SQTF always behave identically; the MASC/TQTF policies result in lower global completion times but higher average processing times and average initiation delays with respect to LASC; and though MASC and

TABLE V
COMPARISON OF LASC AND SQTF POLICIES
(all times are in ms)
AQCT=average query completion time
AQID=average query initiation delay

Data Set		LASC	SQTF
Data Set 1	AQCT	2302	2224
	AQID	418	401
Data Set 2	AQCT	1800	1795
	AQID	344	348
Data Set 3	AQCT	2836	2715
	AQID	402	365
Data Set 4	AQCT	1847	1903
	AQID	335	331

TQTF have the same global query completion times, the TQTF policy has lower average individual completion times, but higher average initiation delays.

The reasons for the similar performance of LASC and SQTF are as follows. For relatively simple queries, i.e., queries with few operations, both policies execute the queries in about the same order. Since a majority of queries in the above mix are simple queries, both LASC and SQTF provide the same results. Table V shows the results for these two policies for a query mix containing only complex queries, namely, Types 5, 6, and 8 queries. In this case, there is a difference between the two policies.

The lower global query completion times of the MASC and TQTF policies are directly related to the availability of the system resources. At startup, or under low resource utilization, all system resources are available. Thus, it is possible to initiate many complex queries without incurring long initiation delays. Since the simple queries can generally be scheduled without much initiation delays, because of their limited resource requirements, scheduling them later does not add much to the overall query initiation delay. Thus, these policies generally result in reduced overall processing time. On the other hand, if the simple queries are scheduled ahead of the complex queries (using, say, LASC), then relatively long initiation delays will result when the complex queries are scheduled, thereby leading to greater overall completion times. The longer average completion time and initiation delays of the MASC and TQTF policies is the result of the long initiation delays experienced by the simple queries, which have to wait for the complex queries to complete execution.

The MASC policy results in higher average completion times, but lower average initiation delays than TQTF. Although both policies favor complex queries over simple ones, MASC executes the "more complex" Type 8 queries before the Types 3–6 queries. Thus, using the same reasoning as above, executing more complex queries first results in lower initation delays, but higher completion times.

V. CONCLUSION

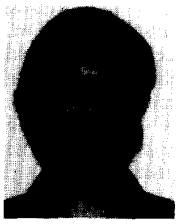
We addressed some run-time system policy issues for database processing in distributed memory multicomputer systems. Specifically, dynamic site allocation and query scheduling policies were examined for a hierarchically structured, hypercube-based multicomputer system. The site allocation policies do not perform *a priori* optimizations. Instead, sites are selected as required during the execution of a query. The paper analyzed and studied the use of such policies in parallel computer systems, where it is difficult to precisely predict the time of occurrence of future events. A simulation was carried out using randomly generated data sets and queries to study a few query scheduling policies, assuming a run-time system that implements the given site allocation policies. For the query mix considered, the results showed that policies that gave preference to complex queries had higher average query completion times, but lower average initiation delays. The results obtained are applicable in general to a variety of multicomputer systems that satisfy the "site model" mentioned in Section III.

Many interesting issues still remain to be studied. Initial data distribution is an important consideration. We assumed partial declustering of data with uniform degree of declustering. Other schemes can also be examined. Several variations of the site allocation policies can be studied, including those that account for the sizes of intermediate relations. The simulation can also be carried out for different types of workloads to study whether certain allocation and scheduling policies are more suitable for particular workloads.

REFERENCES

[1] H. Auer *et al.*, "RDBM: A relational database machine," *Inform. Syst.*, vol. 6, no. 2, pp. 91–100, 1981.
[2] C. K. Baru and O. Frieder, "Database operations in a cube-connected multicomputer system," *IEEE Trans. Comput.*, vol. 38, pp. 920–927, June 1989.
[3] C. K. Baru and S. Padmanabhan, "Join and data redistribution algorithms for hypercubes," *IEEE Trans. Knowl. Data Eng.*, vol. 5, pp. 161–168, Feb. 1993.
[4] C. K. Baru and P. Goel, "Squashed embedding of E-R schema graphs in hypercubes," *J. Parallel Distrib. Computing*, vol. 8, pp. 340–348, Apr. 1990.
[5] A. Bataineh, F. Ozguner, and A. Sarwal, "Parallel Boolean operations for information retrieval," *Inform. Processing Lett.*, vol. 39, pp. 99–108, 1991.
[6] S. H. Bokhari, "On the mapping problem," *IEEE Trans. Comput.* , vol. 30, no. 3, pp. 207–214, Mar. 1981.
[7] S. W. Bollinger and S. F. Midkiff, "Processor and link assignment in multicomputers using simulated annealing," *Proc. Int. Conf. Parallel Processing*, 1988, pp. 1–7.
[8] H. Boral *et al.*, "Prototyping Bubba, a highly parallel database system," *IEEE Trans. Knowl. Data Eng.*, vol. 2, pp. 4–24, Mar. 1990.
[9] K. Bratbergsengen, "The development of the parallel database computer HC16-186," *Proc. 4th Conf. Hypercubes, Concurrent Comput., and Applic.*, 1989, pp. 173–180.
[10] D. K. Bradley and D. A. Reed, "Performance of the Intel iPSC/2 input/output system," *Proc. 4th Conf. Hypercubes, Concurrent Comput., Applics.*, 1989, pp. 141–144.
[11] D. J. DeWitt, "DIRECT: A multiprocessor organization for supporting relational database management systems," *IEEE Trans. Comput.*, vol. C-28, no. 6, pp. 395–408, June 1979.
[12] DeWitt *et al.*, "The gamma database machine project," *IEEE Trans. Knowl. Data Eng.*, vol. 2, pp. 44–62, Mar. 1990.
[13] DeWitt *et al.*, "An evaluation of non-equijoin algorithms," *Proc. 17th Conf. on Very Large Data Bases*, 1991, pp. 443–452.
[14] T. H. Dunigan, "Performance of a second generation hypercube," Tech. Rep. ORNL/TM-10881, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 1988.
[15] O. Frieder, "Multiprocessor algorithms for relational-database operators on hypercube systems," *IEEE Comput.*, vol. 23, pp. 13–28, Nov. 1990.
[16] O. Frieder and H. T. Siegelmann, "On the allocation of documents in information retrieval systems," *Proc. 14th ACM SIGIR*, 1991, pp. 230–239.

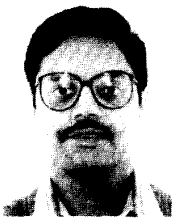
[17] O. Frieder, V. A. Topkar, R. K. Karne, and A. K. Sood, "Experimentation with hypercube database engines," *IEEE Micro*, vol. 12, pp. 42–56, Feb. 1992.
[18] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query optimization for parallel execution," *Procs. ACM SIGMOD*, 1992, pp. 9–18.
[19] H. Hadimioglu and R. J. Flynn, "The architectural design of a tightly-coupled distributed hypercube file system," *Proc. 4th Conf. Hypercubes, Concurrent Comput., Applics.*, 1989, pp. 147–150.
[20] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "Architecture of a hypercube supercomputer," *IEEE Micro*, vol. 6, pp. 653–660, Aug. 1986.
[21] J. P. Hong *et al.*, "A hypercube project and a simulator for a hypercube of computers," *Proc. 2nd Conf. Hypercube Multiprocessors*, SIAM, Sept. 1986.
[22] Intel *i*PSC Data Sheet, Order 280101-001, 1985.
[23] Y. Ioannidis and Y. Cha Kang, "Randomized algorithms for optimizing large join queries," *Proc. ACM SIGMOD*, 1990, pp. 312–321.
[24] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Architecture and performance of the relational algebra machine, GRACE," *Proc. Int. Conf. Parallel Processing*, Aug. 1984.
[25] M. Lakshmi and P. Yu, "Effectiveness of parallel joins," *IEEE Trans. Knowl. Data Eng.*, vol. 2, pp. 410–424, Dec. 1990.
[26] S.-Y. Lee and J. K. Aggarwal, "A mapping strategy for parallel processing," *IEEE Trans. Comput.* , vol. 36, no. 4, pp. 433–442, Apr. 1987.
[27] A. Liu and S. Chang, "Site selection in distributed query processing," *Proc. 3rd Conf. Distrib. Computing Syst.*, 1982, pp. 7–12.
[28] T. Martin, K. Lam, and J. Russell, "An evaluation of site selection algorithms for distributed query processing," *Comput. J.*, vol. 33, pp. 61–70, 1990.
[29] B. L. Menezes, K. Thadani, A. G. Dale, and R. Jenevien, "Design of a HyperKYKLOS-based multiprocessor architecture for high-performance join operations," *Proc. 5th Int. Workshop Database Machines*, 1987, pp. 88–101.
[30] P. Mishra and M. H. Eich, "Join processing in relational databases," *ACM Computing Surv.*, vol. 24, pp. 63–113, Mar. 1992.
[31] E. R. Omiecinski and E. T. Lin, "Hash-based and index-based join algorithms for cube and ring connected multicomputers," *IEEE Trans. Knowl. Data Eng.*, vol. 1, pp. 329–343, Sept. 1989.
[32] E. R. Omiecinski and E. T. Lin, "The adaptive-hash join algorithms for a hypercube multicomputer," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 334–349, May 1992.
[33] R. P. Pargas, J. C. Peck, and A. L. Pugh, "Use of semi-join programs for join queries on a hypercube," *Proc. 4th Conf. on Hypercubes. Concurrent Comput., and Applics.*, 1989, pp. 457–462.
[34] S. Pramanik and D. Vineyard, "Optimizing join queries in distributed databases," *IEEE Trans. Software Eng.*, vol. 14, Sept. 1988.
[35] J. C. Peterson, J. O. Tuazon, D. Lieberman, and M. Pniel, "The MARK III hypercube-ensemble concurrent computer," *Proc. Int. Conf. Parallel Processing*, 1985, pp. 71–73.
[36] J. L. Pfaltz, J. C. French, and S. H. Son, "Parallel set operators," *Proc. 4th Conf. Hypercubes, Concurrent Computers, and Applics.*, 1989, pp. 481–486.
[37] S. Y. W. Su, *Database Machines: Concepts and Techniques*. New York: McGraw-Hill, 1988.
[38] M. Stonebraker, "The case for shared nothing," *Data Eng.*, vol. 9, Mar. 1986.
[39] V. A. Topkar, O. Frieder, and A. K. Sood, "Duplicate removal on hypercube engines: An experimental analysis," *Parallel Computing*, vol. 17, pp. 845–971, Oct. 1991.
[40] E. Upchurch *et al.*, "Parallel joins on the Mark III hypercube," *Proc. 4th Conf. Hypercubes, Concurrent Comput., Applics.*, 1989, pp. 453–456.
[41] B. Wagar, "Hyperquicksort: A fast sorting algorithm for hypercubes," *Proc. 2nd Conf. Hypercube Multiprocessors*, Sept. 1986.
[42] A. Witkowski, K. Chandrakumar, and G. Macchio, "Concurrent I-O system for the hypercube multiprocessor," *Proc. 3rd Conf. on Hypercube Concurrent Comput. and Applics.*, SIAM, Jan. 1988.
[43] C. Yu, C. Chang, D. Templeton, and E. Lund, "On the design of a distributed query processing algorithm," *Proc. ACM SIGMOD*, 1983, pp. 30–39.



O. Frieder (SM'93) received the Ph.D. degree from the University of Michigan in 1987.

From 1987 to 1990, he was a Member of Technical Staff in the Applied Research Area of Bell Communications Research. In 1990, he joined the Department of Computer Science, George Mason University, where he is now an Associate Professor. While at George Mason University, he has also served as a Staff Consultant at the Federal Bureau of Investigations from 1991 to 1993, and at the Institute for Defense Analysis from 1992 to 1993. Since 1993, he has been a Staff Consultant to IBM FSC (now Loral Federal Systems). His research interests include parallel and distributed database and information retrieval systems and biological and medical data processing architectures.

Dr. Frieder has published more than 50 refereed papers, has been granted two patents, and has received research support from several government and industrial organizations. In 1993, he received the National Science Foundation's National Young Investigator Award. He is the Area Editor for databases of *IEEE Computer*. He is a member of Phi Beta Kappa.



C. K. Baru (S'84–M'85–SM'94) received the Ph.D. degree in electrical engineering from the University of Florida, Gainesville, FL, USA, in 1985.

He is currently one of the team leaders in the IBM DB2 Parallel Database Project at the IBM Toronto Laboratories,, North York, ON, Canada. He has research and development experience in the areas of parallel and object-oriented database systems. Prior to joining IBM, he was an Assistant Professor of Computer Science and Engineering at the University of Michigan, Ann Arbor, USA, where his research was funded by the National Science Foundation and the AT&T Foundation.

Dr. Baru has published several articles in his areas of research interest, and he has also presented industry tutorials on these topics. He is a member of ACM.