JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 23, 382–398 (1994)

Load-Balanced Join Processing in Shared-Nothing Systems

HONGJUN LU[1] AND KIAN-LEE TAN

Department of Information Systems and Computer Science, National University of Singapore, Lower Kent Ridge Road, Singapore 0511

In a shared-nothing parallel database system, a join operation is split into a set of tasks that are allocated to the nodes in the system to be executed concurrently and independently. While parallel processing could greatly reduce the completion time of a join operation, the system performance may degrade because of load imbalance across the nodes caused by data skewness in the relations. Load-balanced join processing uses various techniques to evenly distribute the load among nodes in a system and hence improves the overall system performance. In this paper, the basic issues in designing load-balanced parallel join algorithms are identified. From the solutions to those issues, a large set of load-balanced join algorithms can be constructed. Performance of four representative algorithms—two *dynamic* load-balancing algorithms proposed in this paper and two *static* load-balancing algorithms adapted from similar algorithms in the literature—is studied and compared with that of a parallel join algorithm that does not balance the join load. The results of our study clearly show the benefits of load-balancing. This study also demonstrates that the dynamic load-balancing techniques proposed in this paper not only are feasible but also provide good system performance. © 1994 Academic Press, Inc.

1. INTRODUCTION

A *shared-nothing* system comprises a set of processing nodes interconnected through a communication network. Each node has its own local memory and disk drives. A processor at one node has no direct access to memory or disks of other nodes. Data sharing among processors is realized by some message passing mechanisms. To facilitate parallel processing, data, or relations, are typically partitioned horizontally across a subset of the nodes. As such architecture can be scaled up to hundreds and probably thousands of processing nodes, it becomes one of the major parallel database system architectures [1]. Compared with other parallel database architectures, such as shared-memory systems, one major disadvantage of shared-nothing architecture is that load-balancing among the processing nodes becomes more important but more difficult. Examples of shared-nothing systems include GAMMA [2], NonStop SQL [4] and DBC/1012 [21].

Consider the parallel processing of the expensive relational *equi-join* operation that has received much attention in recent years [3, 6, 9, 18, 24, 25]. Most of the proposed algorithms adopt the three-phased *task-oriented* approach: (1) *task generation*, (2) *task allocation*, and (3) *task execution*. In the task generation phase, the join is split into a set of independent tasks[2] which are then allocated to the nodes in the task allocation phase. Finally, in the task execution phase, all the processors execute the allocated tasks concurrently and independently. This task-oriented approach is most effective when the processing loads allocated to the nodes are approximately the same. However, in real database applications, the load across the nodes will not be the same because of factors such as data skew. This unbalanced load across the nodes in the system may negate the benefits that can be gained by parallelism and degrade the system performance [12, 23].

In this paper, we revisit the problem of balancing the load for an equi-join operation in shared-nothing systems. This work is different from previous work in two ways. First, based on our understanding of the problem and the algorithms proposed in the literature, we develop a framework for designing task-oriented parallel join algorithms, from which a large number of feasible load balanced join algorithms can be constructed. Second, as case studies, we design two parallel join algorithms that apply *dynamic* load-balancing techniques. The performance of the new algorithms is studied and compared with that of an algorithm without load-balancing and two other algorithms using *static* load-balancing.

Several observations motivated this work. First, while many join algorithms proposed in the literature are different, most of them are but variants of one another. We would like to develop a framework where we may identify the similarities and differences. We believe that one way to do so is to identify the issues that are involved in the development of a join algorithm. Second, studies on load-balanced join algorithms have shown that no single join algorithm is superior in all cases [3, 6]. When the data skew is low, the simple parallel GRACE hash join algorithm [18] which does not attempt to balance the load, performs well enough. On the other hand, static load-balanced join algorithms (such as the extended adaptive load balancing join algorithm (ABJ$^+$) [6] and the virtual processor range partitioning algorithm [3]) are superior with high data skew. What is needed then is an

[1] E-mail: luhj@iscs.nus.s§. Fax: 65-779-4580.

[2] The term "task" here refers to the join of two corresponding buckets from the two relations.

0743-7315/94 $6.00

algorithm that is robust to data skew. In particular, we want to keep the advantage of the GRACE join algorithm at low skew, as skew conditions are typically mild for many applications [6], and avoid its poor performance at high skew. We would like to study whether it is feasible to balance the join load dynamically and how effective it is to do so. Third, introducing dynamism is important when statistics are not available or are inaccurate. Moreover, in multiuser environments, even if the join load is evenly distributed across the nodes, the workload at each node may vary drastically, resulting in high completion time of the join. Last, communication cost is getting cheaper in shared-nothing systems and parallel processing algorithms may be designed to take advantage of this.

In the next section, the issues related to load-balanced join algorithms are discussed. As a result, a large set of join algorithms may be derived. Section 3 describes four algorithms selected from those presented in Section 2. In particular, we present two static algorithms and propose two dynamic algorithms. In Section 4, we study the relative performance of the four algorithms. The GRACE hash join is also used as a reference to study the benefits of load-balancing. Finally, we conclude and discuss some promising future work in Section 5.

2. TASK ORIENTED PARALLEL JOIN ALGORITHMS

The task oriented parallel join processing usually consists of three phases: *task generation, task allocation,* and *task execution.*[3] For each phase, there are a number of important issues involved and decisions should be made when a join processing algorithm is to be designed. These issues and possible solutions are summarized in Fig. 1[4] and the details are discussed in this section.

2.1. Task Generation

In the task generation phase, a join operation is split into a set of tasks. Each task is a join of subrelations of the source and target relations that can be executed at a node independently but concurrently with other tasks. The results of all such tasks can be easily assembled into the final result which is the same as that from the operation without decomposition. Four basic issues in this phase include:

- the decomposition of the relations into subrelations,
- the number of tasks to be generated,
- the formation of task after decomposition, and
- the kind of statistics to be collected and maintained.

[3] These three phases form a logical view of the task oriented parallel join processing. The actual implementation may interleave these phases.

[4] Some cases listed in the taxonomy can be viewed as special cases of others. For example, the *fragmentation and replication* approach in decomposing a join into tasks is a special case of either the *full fragmentation* or the *full replication* approach. However, we would like to separate them, as the finer classification makes it easier to classify and refer to the existing algorithms.

2.1.1. Decomposition of the Relations

The basic criterion of decomposition is that the union of the results obtained from joining the subrelations (tasks) should be the same as the results of joining the original relations. There are several ways to decompose the relations to form tasks and ensure the correctness at the same time:

- *Full fragmentation.* In this category, each tuple will participate in only one task; that is, a tuple will not be involved in more than one join operation. Tasks are generated by partitioning the relations into buckets. Thus, only corresponding buckets from both relations need to be joined. Two techniques that have been employed to partition a relation are *range partitioning* and *hash partitioning*. For the former, the range of possible values of the joining attribute is divided into several subranges, one for each bucket. The latter employs a hashing function such that tuples that hash to the same value are grouped into the same partition. Examples of algorithms that employ this method are the Hybrid hash join, the GRACE hash join, and their variants [3, 6, 9, 18].
- *Fragmentation and replication.* While there is no replication in the full fragmentation decomposition technique, the *fragmentation and replication* decomposition technique allows partial replication of data. It employs the fragment-and-replicate strategy to generate tasks. The larger relation is split into buckets while the entire smaller relation is replicated into the same number of buckets; that is, each task comprises a bucket of the larger relation and the entire smaller relation. This ap-

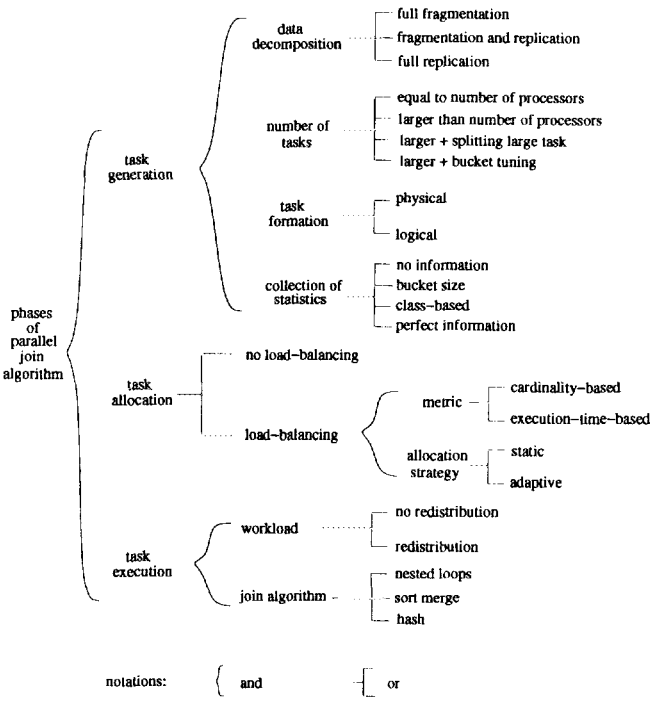

FIG. 1. Three phases of risk-oriented parallel join algorithms.

proach is effective when the smaller relation fits in memory. The nested-block join [22] is such an example.

• *Full replication.* In *full replication*, both relations are replicated in the following manner: the source relation is split into m buckets and the target relation into n buckets to produce $m \times n$ tasks. That is, each of the m buckets will be joined with each of the n buckets. Join algorithms that employ this method appear in [8, 17, 19].

While the *full fragmentation* category seems to be the most commonly used, it is not effective when the data are highly skewed. In such cases, the other two approaches come in handy.

2.1.2. Number of Tasks to Be Generated

The second issue in task generation is the optimal number of tasks to be generated. The number of tasks to be generated is limited by the decomposibility of the tasks. It is also limited by the amount of memory at each processor since space is needed to keep the information about tasks. The optimal number of tasks to be generated is closely associated with the size of the data that an operation work on. A large number of tasks may be generated such that an operation works on only a few tuples of data. This may result in poor performance because of the overhead incurred in splitting the data and duplicating the operation and the additional I/Os as a result of fragmented pages. On the other hand, a small number of tasks may also not be favorable, since this will limit the flexibility of allocating tasks to the processors and result in uneven loads among the processors.

The four basic approaches that have been adopted in the literature for a p-node system are:

• To generate p tasks. Examples of algorithms that employ this strategy are GRACE hash join, hybrid hash join, and simple hash join [18].

• To generate a large number of tasks ($>p$) and allocate these tasks to the nodes to satisfy some balancing criterion (to be discussed in the next section). The algorithms in [6] use this approach.

• To generate a large number of tasks ($>p$) and increase the number of tasks prior to allocating the tasks. One approach to achieve this is by splitting large buckets (which exceeds the size of the memory) into smaller ones that fit in the memory [13, 16, 24, 25].

• To generate a large number of tasks ($>p$) and reduce the number of buckets prior to allocating them to the nodes. The number of buckets may be reduced by "tuning the bucket size" so that several small buckets are combined to form a larger bucket as long as the larger bucket fits in memory. GRACE hash join [10] is such an example.

2.1.3. Task Formation

The third issue in task generation is whether the tasks generated are *physically* or *logically* formed. For a physically formed task, the data associated with the task (that is, the two buckets that correspond to the task) are collected and located at a single node. On the other hand, the data that correspond to a logically formed task may be declustered across several nodes. While a physical task may be processed at the node where it resides, data associated with a logical task must be assembled before they can be processed. However, logical tasks are more amiable for load-balancing. Both approaches have been explored in the literature. For the former see [6, 18] and the latter can be found in [6, 24, 25].

2.1.4. Statistics Collection

Another related issue in task generation is the statistics to be collected and kept during task generation. These statistics can then be used by the next phase to determine how the tasks are to be allocated. At one extreme, we do not need any information; that is, the tasks are allocated "blindly". At the other extreme, we have perfect information; that is, the bucket sizes and result sizes are known. An in-between is to keep track of the bucket sizes only [6]. Another possibility is to organize the number of distinct values into "equivalence" classes; the information maintained is (1) number of distinct classes (e), (2) for each class i, number of distinct attributes (d_i) (if all classes have the same number, then one value is kept), and (3) for each class i, number of tuples from both the source and target relations (r_i/s_i). Thus, we can estimate the result size as follows:[5]

$$\sum_{i=1}^{e} \frac{r_i \cdot s_i}{d_i}. \tag{1}$$

With the bucket and result size information, the execution time of the join may be estimated [24, 25].

2.2. Task Allocation/Load Balancing Phase

After the source and target relations have been partitioned, we have a pool of tasks (logical or physical) to be allocated to the nodes for concurrent processing. There are basically two strategies to allocate these tasks to different nodes in the system:

2.2.1. Allocate Tasks without Load Balancing

The straightforward way to allocate the tasks is to predetermine which node will process which task. When the tasks are generated, they are sent to the appropriate nodes. Some of the earlier work adopts this approach and

[5] The ideal case (perfect information): The number of distinct attributes per class should be 1, i.e., $d_i = 1\ \forall_i$, so that the number of distinct attributes is exactly the same as the number of classes. In this case, the statistics will provide an exact result size and hence allows a good estimate of the processing cost of the bucket at a node. In [20], this case is assumed. However, such detailed information consumes memory and is feasible only for large amount of memory.

have been shown to be effective when the load across the nodes are approximately the same [18]. However, the performance could be very poor when the load is unequally distributed as a result of factors such as data skew [12, 23].

2.2.2. Allocate Tasks with Load Balancing

With a large number of tasks, tasks can be allocated to the nodes so that the load across the nodes is approximately the same. The most commonly used heuristics to allocate the tasks are the best-fit-like and first-fit-like strategies which generally work in two phases: (1) the tasks are sorted according to some criterion; (2) the tasks are allocated in descending order in the following greedy manner: the next task to be allocated is assigned to the node that is expected to finish first. This leads to several issues that concern the load metrics and allocation strategies.

Load Metrics. The first issue is the notion of "balanced system"—"What is the criterion used to ensure that the system is balanced?". Two commonly used criteria, or *load metrics*, are the *cardinality of the buckets* and the *estimated execution time*:

- *Cardinality.* With this criterion, the objective is to balance the load across the nodes based on the size of the partition; that is, each node should process approximately the same number of tuples [6, 9]. The inherent assumption is that the processing time is proportional to the number of tuples processed.
- *Estimated execution time.* The time to process each task is first estimated using some cost model. The tasks are then allocated so that each node will complete about the same time [24, 25]. This approach will perform well if statistics that can provide an accurate estimation of the execution time of each task are available.

Allocation Strategy. The second issue concerns the allocation strategy, that is, how each task is to be allocated. This may be categorized as *static* and *adaptive* methods:

- *Static.* In static approaches, all the tasks generated are allocated prior to the execution of the tasks. Thus each node knows exactly the tasks that it is to process. Most of the published work on load balanced join algorithms are static [6, 9, 24, 25].
- *Adaptive.* The main feature of the adaptive approach is that task allocation is *demand-driven*. Each node will process a task and will *acquire* the next task only when the current task is processed. In this way, each node knows only the task that it is processing and has no idea of which will be the next task. To the best of our knowledge, there is no published report on shared-nothing environments that employs this approach. However, the approach has been studied in shared-disk environments [13].

2.3. Task Execution

In the task execution phase, each node independently performs the join of the tasks allocated to it. The main issues here concern the redistribution of workload and the choice of the local join algorithms for each task at each node.

2.3.1. Workload

There are two ways to handle the existing workload:

- *No redistribution of workload.* In this case, once the execution phase begins, there is no redistribution of workload allocated to a node. If the system is unbalanced, it will remain so. Most of the static load balanced join algorithms fall in this category.
- *Redistribution of workload.* In this case, to avoid an unbalanced system, workload may be redistributed. In other words, tasks may migrate during the execution phase from an overloaded node to an underloaded node. This may be achieved by introducing a task redistribution strategy to redistribute the workload at overloaded nodes to the idle/underloaded nodes. That is x overloaded nodes may redistribute their load to y underloaded nodes. The granularity of the workload to be redistributed may be restricted to a whole task as originally generated during the task generation phase or may be a subtask generated from a task for the purpose of distribution. In [13], this strategy is used but the redistribution is restricted to only two nodes—a donor and an idle node.

2.3.2. Local Join Algorithms

Any uniprocessor join methods may be used for each local join. The three basic uniprocessor join methods are: (1) nested-loops join, (2) sort–merge join, and (3) hash join. Studies have shown that none of the methods is superior in all cases. In general, nested-loops join performs well when the relations are small and fit in memory or when an index exists on the inner relation; sort-merge is superior when both relations are already sorted; and hash join performs best when enough main memory is available and when the join attribute values are uniformly distributed.

Based on the various methods to process each phase of the algorithm, we can derive a large set of load-balanced join algorithms which is equivalent to the product of the number of methods in each phase. Table I summarizes some of the existing join algorithms in shared-nothing systems based on the issues discussed above.

3. SELECTED LOAD-BALANCED JOIN ALGORITHMS

In this section, we describe four hash-based parallel join algorithms designed under the framework proposed in the last section. Among the four algorithms, two dynamic load-balanced join algorithms—*dynamic load-balanced join algorithm* (DBJ) and *extended dynamic load-*

TABLE I
Summary of Some Existing Join Algorithms in Shared-Nothing Systems

Algorithm	Task generation	Load-balancing	Task execution
Simple hash [18]	Full fragmentation, #processors, physical, no information	No load-balancing	No workload redistribution, hash
GRACE hash [18]		Same as simple hash join	
Hybrid hash [18]		Same as simple hash join	
ABJ [6]	Full fragmentation, large, physical, bucket size	Load-balancing, cardinality, static	No workload redistribution, hash
ABJ$^+$ [6]	Full fragmentation, large, logical, bucket size	Load-balancing, cardinality, static	No workload redistribution hash
Scheduling hash [25]	Full fragmentation, large + split, logical, class-based	Load-balancing, execution time, static	No workload redistribution, hash
DBJ (proposed)	Full fragmentation, large, physical, class-based	Load-balancing, execution time, static	Workload redistribution, hash

balanced join algorithm (EDBJ)—are new algorithms designed in this study, whose details are given in the following subsections. The other two algorithms—*static load-balanced join algorithm* (SBJ) and *adaptive static load-balanced join algorithm* (ASBJ)—are static algorithms adapted from similar ones that have appeared in the literature. As their detailed descriptions can be found in the relevant papers, only brief descriptions are given here to avoid repetition.

3.1. Static Load-Balanced Join Algorithm (SBJ)

Algorithm SBJ is similar to algorithm ABJ in [6]. The three phases for the join of relations R and S are as follow:

• A *large number of physical* tasks are generated using *full fragmentation data decomposition*. The tasks are assigned to the nodes statically. The buckets of R are generated by partitioning R as follows: Each node i, where $1 \le i \le p$, allocates one output buffer for each of the B buckets, where B is the number of desired buckets. Each node independently scans the portion of relation R stored therein and hashes the tuples to the corresponding output buffers based on the hash results. When a buffer is full, it is sent to the node to which it is assigned through the interconnection network. Once R is partitioned, the same hashing function is used to split S. During the partitioning process, each node collects *class-based* statistics for the buckets that are allocated to it. Once the partitioning is done, each node estimates the time to process each of the allocated tasks and reports the information to the coordinator. The coordinator computes the average completion time and broadcasts this information to all the nodes.

• The load-balancing phase is performed using static allocation strategy with the execution time as the load metric. Based on the statistical information, tasks from overloaded nodes will be relocated to underloaded nodes. The following two steps are used: First, using a best-fit strategy, each node retains k tasks such that the aggregated time to complete these tasks is greater than or equal to the average completion time; that is, k is the smallest number of tasks that satisfies the conditions

$$\sum_{j=1}^{k} Cost_j \le Cost_{avg} \quad \text{and} \quad \sum_{j=1}^{k+1} Cost_j > Cost_{avg},$$

where $Cost_j$ and $Cost_{avg}$ are the time to process the j^{th} task and the average completion time, respectively. Next, the cost of the remaining (*excess*) tasks and the new completion time (excluding the excess tasks) are reported to the coordinator, which then uses a best-fit algorithm to assign these excess tasks to the nodes. The excess tasks are then physically collected to the newly assigned nodes.

• In the task execution phase, each node performs a local join on each of the tasks assigned to it. There is no redistribution of workload once execution begins. For simplicity, we use the uniprocessor *hash-based nested-loops join* algorithm in our study. This algorithm is shown to be superior to the other uniprocessor algorithms when we handle medium-sized relations (the size of the build-

ing relation is no more than five times the size of the memory [15]). The algorithm comprises two phases: (1) Several pages of the building relation, which is determined by the available memory, are read and the hash table is built: (2) The whole of the probing relation is read a page at a time and each tuple is probed for joinability with the partially staged hash table. These two phases are repeated until all the pages of the building relation are read.

3.2. Adaptive Static Load-Balanced Join Algorithm (ASBJ)

Algorithm SBJ has two major shortcomings. First, when data are highly skewed, a large number of tuples will be allocated to one same node (the skewed node) resulting in high I/O cost at the node and communication hot spot. Second, the balancing is coarse in the sense that the granularity of load balancing is a task, that is the task as produced from the task generation phase. Thus, when the data is seriously skewed, one or two tasks may degrade the performance of the whole system. Algorithm ASBJ remedies the first problem by forming logical tasks and handles the second problem by splitting expensive tasks into smaller (and less expensive) tasks. The three phases of the algorithm to perform the join of relation R and S are:

- A large set of tasks are generated using the full fragmentation data decomposition. Here, each node partitions the fragments of relations R and S into *logical tasks*; that is, local subbuckets are stored back into the local disk. At the same time, statistics (cardinalities of the buckets) are collected. Once the partitioning is done, the coordinator assembles the statistics and sorts the information in nonascending order of the smaller size of the two buckets of the task. Next, for every task whose smaller bucket do not fit in the memory, the task is "split" into k subtasks using the *fragment-and-replicate* algorithm by dividing the smaller bucket into approximately k equal-sized buckets (whose hash tables fit in memory) and duplicating the larger bucket. In this way, the set of tasks is increased by $k - 1$ tasks.[6] k is given as

$$k = \left\lceil \frac{b \cdot f}{m} \right\rceil,$$

where b and m are the size of the smaller bucket of the task to be split and the memory available, and $b \cdot f$ represents the size of the hash table for a bucket of size b. The cost to process each task is then estimated.

- In the load-balancing phase, the greedy LPT algorithm [5] is used to assign the tasks to the nodes to minimize the estimated completion time. The LPT algorithm achieves this by iteratively allocating the most expensive of the remaining tasks to the node with the least (estimated) processing time until no more tasks are left in the system. The tasks are then distributed to the nodes in the following manner: At each node, for each of the tasks that is split, the smaller bucket is distributed evenly (in round robin) and the larger bucket is broadcast to the proper nodes. An unsplit task is simply sent to the corresponding node. Clearly, all tasks are allocated prior to the execution phase; i.e., a static allocation strategy is adopted.
- Finally, each node executes the tasks allocated to it, and no redistribution of workload is allowed.

[6] It should be noted that the splitting is logical since the actual task is not physically split into smaller tasks.

3.3. Dynamic Load-Balanced Join Algorithm (DBJ)

Unlike algorithms SBJ and ASBJ, in which a task is always executed at the node to which it is allocated, algorithm DBJ is dynamic in the sense that a task, or a portion of a task, may be transferred from an overloaded node where it is originally allocated to an underloaded node. The three phases of the algorithm are as follow:

- The task generation phase is similar to that of algorithm SBJ. Each node partitions the fragments of the source and target relations (R and S) to produce a large number of tasks, each of which has been assigned to a node. As the tuples are partitioned, they are sent to the corresponding node through the interconnection network. At the same time, statistics (cardinalities of the buckets) are collected. Once the partitioning is done, each node estimates the time to process each of the allocated tasks.
- At each node, the allocated tasks are executed one at a time, in nonincreasing order of the estimated time. This is reasonable, since it is easier to balance small tasks. During execution, each node maintains certain information about the task that is being processed, including the sizes of data left (in pages) and the size of the result generated so far. Such information is necessary in order to make correct decisions when transferring load between nodes.
- When a node finishes processing all the allocated tasks, it requests for some load from a node that has not completed the execution of its allocated tasks. The appropriate candidate node(s) that is overloaded and the amount of load to be transferred are then determined. The transferring process is realized by shipping data buckets from the *donor* to the idle nodes. This can be summarized as follows:

1. The idle node sends a *load-request* message to the *coordinator* (e.g., node with smallest index) to ask for more work.
2. At the coordinator, requests are queued in a first-come-first-serve basis. The coordinator broadcasts a *load-information* message to all nodes.
3. Upon receiving the *load-information* message,

each node computes its current load and sends it to the coordinator.

4. The coordinator determines the donor and sends a *transfer-load* message to the donor, after which the coordinator will proceed to serve the next request.[7]

5. The donor determines the load to be transferred and sends the load to the idle node. Only when the load is determined can the donor compute its new processing load for another request.

This process of transferring load between a busy node and an idle node is repeated until some criterion, which indicates that the minimum completion time has been achieved, is satisfied.

As in all load-balancing algorithms, there are two important issues to be addressed: (1) which node should be the donor, and (2) how much of the load should be transferred. The next two subsections describe the general approach taken in this study, followed by a subsection showing how these issues are addressed when a hash-based nested-loops join algorithm is used for all local joins.

3.3.1. Determination of Donor

When an *idle* node requests work, any *busy* node can be a donor. Naturally, we hope that the donor is the most heavily loaded node, since this will determine the completion time of the join. However, it is impossible to provide an exact measure of the load of a node. We use the *estimated completion time* (ECT) as the measure of the load of a node. When *task transfer* is required, the ECTs of those nodes with uncompleted tasks are computed, and the node with the maximum ECT will be chosen by the coordinator as the donor (ties are arbitrary broken). That is, node i is the donor when (assuming p nodes in the systems)

$$ECT_i = \max_{j=1}^{p} (ECT_j).$$

When there are k unprocessed tasks at node i, ECT_i at node i is given by

$$ECT_i = \sum_{j=1}^{k} EET_{ij} + EFT_i.$$

The first component, $\sum_{j=1}^{k} EET_{ij}$, is the total *estimated execution time* of the tasks at node i that are yet to be processed. *EET* is estimated based on the statistics collected and is determined by the cost formulas of the local join, that is,

$$EET = JoinCost(|R|, |S|, |Result|),$$

where $|R|$, $|S|$ are the sizes (in pages) of the source and target buckets of the task and the result size, $|Result|$, is derived as given by Eq. (1).

The second component, EFT_i, is the *estimated finish time of the task that is currently being processed* at node i, that is, the time needed to complete the processing of the currently executed task. Given a task with bucket sizes $|R|$ and $|S|$ respectively, EFT is determined using the expression

$$JoinCost(|R|, |S|, |Result|) = JoinCost(r, s, res) + EFT,$$

where r and s are the number of pages (inclusive of the page that is currently processed) of R and S already processed and *res* is the number of result tuples produced using these tuples. Recall that r, s, and *res* can be derived from the processing information maintained.

3.3.2. Determination of the Number of Data to Be Transferred

We employ the following heuristics to determine the number of data to be transferred:

1. *Determine the task.* If there are unprocessed tasks at the node, the load is to be taken from the unprocessed task with the smallest *EET*; otherwise the load is to be taken from the task that is currently being executed. By transferring the task with the smallest *EET*, it is hoped that *thrashing* may be minimized.

2. *Determine the amount of load.* The amount of load to be transferred must satisfy the following two constraints:

(a) *The amount of load transferred should provide a gain in the completion time of the join operation*; that is, the *ECT* at the donor node after the transfer (ECT^{donor}_{after}) must be less than the *ECT* at the donor node prior to the transfer (ECT^{donor}_{before}),

$$ECT^{donor}_{after} = ECT^{donor}_{before} + T_{overhead} - T^{donor}_{join} < ECT^{donor}_{before}, \quad (2)$$

where $T_{overhead}$ is the overhead incurred in transferring the load and T^{donor}_{join} is the cost to perform the join of the transferred load if it had been performed at the donor node. We assume, for simplicity of analysis, total overlap in both I/O and communication. Thus, $T_{overhead} = \max(T_{io}, T_{comm})$, where T_{io} and T_{comm} are the I/O and communication cost incurred to transfer the load.

(b) *After the load is transferred, the ECT at the idle node* (ECT^{idle}_{after}) *should not exceed that of its donor*; that is,

$$ECT^{donor}_{after} \geq ECT^{idle}_{after} = T^{idle}_{join}, \quad (3)$$

where T^{idle}_{join} is the cost to perform the join of the transferred load at the idle node. The overhead of the transferred load at the idle node may be negligible if pipe-

[7] For simplicity, we have restricted to serving an idle node at a time. It could be generalized, for example, to select (at most) k busy nodes as the donors of the first k requests in the queue.

lining is feasible; that is, as the load arrives, it is processed immediately without the need to store the incoming stream to disks. In most cases, T_{join}^{idle} may need to include part of the overhead, $T_{overhead}$.

The first constraint described above essentially questions the worth of the transfer and the second aims to minimize thrashing.

3.3.3. Task Transfer Strategy for Hash-Based Nested-Loops Join

In this section, we demonstrate how the donor and amount of load to be transferred can be determined when the hash-based nested-loops join algorithm (HNL) is used for all local joins. The result can be easily generalized to other algorithms. We denote the time for the entire join as $T_{HNL}(|R|, |S|, |Res|)$, where $|R|$, $|S|$, and $|Res|$ represent the size (in pages) of the source, target, and result relations respectively. For the following discussion, the source relation is chosen to be the smaller of the two relations.

Determine donor for HNL. For a join of two buckets, T_1 and T_2, with estimated result size of $|Result|$, *EET* is simply $T_{HNL}(|T_1|, |T_2|, |Result|)$. *EFT* is a little more complex and depends on the relative size of the smaller bucket on which a hash table is built. If $|T_1| \cdot F \leq |M|$ (we assume a hash table for R occupies $|R| \cdot F$ pages), where $|M|$ is the amount of memory available, the hash table for T_1 is staged in memory, and the joining process is to scan T_2 once and probe for a match for each tuple of T_2 scanned. The load of the node at any time after building the hash table is determined by the unprocessed pages of T_2. However, when T_1 is too large for the available memory, that is, $|T_1| \cdot F > |M|$, T_2 needs to be scanned several times, each with a portion of T_1 that can be staged in memory. The load therefore is dependent on both the unprocessed pages of T_1 and T_2. With this in mind, we proceed to estimate the *EFT* for a node.

Case 1. $|T_1| \cdot F \leq |M|$. The time to perform the join of T_1 and T_2 with result *Result* is estimated by

$$T_{HNL}(|T_1|, |T_2|, |Result|) = T_{HNL}(|T_1|, n_1, |Res|) + EFT,$$

where n_1 denotes the number of pages of T_2 that have been processed, inclusive of the page that is processed at the time of the load-information request and $|Res|$ is the size of the resultant relation generated so far, using n_1 pages of T_2, inclusive of the page that is being generated. If the statistical information is only probablistic, that is the result size as generated by Eq. (1) is not exact, we estimate the size of *Result* by the following expression (assuming uniform distribution within a bucket):[8]

$$\frac{|Result|}{|Res|} = \frac{|T_2|}{n_1}. \tag{4}$$

Therefore,

$$EFT = T_{HNL}(|T_1|, |T_2|, |Result|) - T_{HNL}(|T_1|, n_1, |Res|). \tag{5}$$

Case 2. $|T_1| \cdot F > |M|$. The number of loops required to scan T_2 is $L_1 = |T_1| \cdot F/|M|$. Let n_1 denotes the number of pages of T_1 processed. If $n_1 = |T_1|$, then all the pages of T_1 have already been read and the current loop is the last loop. In this case, *EFT* is derived in the same way as the case when $|T_1| \cdot F\ |M|$, using Eq. (5) (It should be noted that the value of $|T_1|$ in Eq. (5) should be replaced by $|T_1| - (L_1 - 1).\ |M|/F$). Otherwise, the following computation is done. Let n_2 denote the number of pages of T_2 processed in the current loop. Then the number of loops processed, inclusive of the loop that is currently being processed, is $L_2 = n_1/|T_1| \cdot L_1$. The estimated time to perform the join is

$$T_{HNL}(|T_1|, |T_2|, |Result|) = T_{HNL}\left((L_2 - 1) \cdot \frac{|M|}{F}, |T_2|, |Res_1|\right) + T_{HNL}\left(\frac{|M|}{F}, n_2, |Res_2|\right) + EFT.$$

The first expression on the right-hand side represents the cost to process the first $(L_2 - 1)$ iterations (i.e., excluding the current loop). The second expression reflects the cost to process the current loop (until the current page). $|Res_1|$ and $|Res_2|$ are unknown and may be estimated using the two expressions[9]

$$\frac{|Res_1|/(L_2 - 1)}{|Res_2|} = \frac{|T_2|}{n_2} \quad \text{and} \quad |Res_1| + |Res_2| = |Res|,$$

where $|Res|$ is the size of results generated thus far. $|Result|$ is estimated as follows:

$$\frac{|Result|}{|Res_1|} = \frac{|T_1|}{(L_2 - 1) \cdot |M|/F}.$$

Therefore,

$$EFT = T_{HNL}(|T_1|, |T_2|, |Result|) - T_{HNL}\left((L_2 - 1).\ \frac{|M|}{F}, |T_2|, |Res_1|\right) - T_{HNL}\left(\frac{|M|}{F}, n_2, |Res_2|\right).$$

Determine Number of Data to Be Transferred for HNL. As in the case for the determination of donor,

[8] In the case when $|Res| = 0$, that is when no tuple has been generated yet, *Result* is estimated using Eq. (1).

[9] When $L_2 = 1$, then $|Res_1| = 0$, $|Res_2| = |Res|$, and $|Result|$ can be estimated using an expression similar to Eq. (4).

there are also two cases that we need to consider, based on whether the smaller bucket of the selected task can be staged in memory or not. We use the fragment-and-replicate algorithm to "chop up" a large task into two portions—one to be transferred to the idle node and the other to remain at the donor node. We use the same notations as above in the following discussion.

Case 1. $|T_1| \cdot F \leq |M|$. If $|T_1| \cdot F \leq |M|$, then the idle node needs the entire relation T_1 and portion of the unprocessed T_2. Let k be the number of pages of T_2 to be transferred. k should satisfy the following two expressions, which correspond to the two constraints (Eq. (2) and (3), respectively),[10]

$$T_{\text{overhead}} = \max(T_{\text{comm}}, T_{\text{io}}) \leq T_{\text{join}}^{\text{donor}}(|T_1|, k, Res_k) \quad (6)$$

and

$$ECT + T_{\text{overhead}} - T_{\text{join}}^{\text{donor}}(|T_1|, k, Res_k) \geq \begin{cases} T_{\text{join}}^{\text{idle}}(|T_1|, k, Res_k), & \text{if } k \geq |T_1| \\ T_{\text{join}}^{\text{idle}}(k, |T_1|, Res_k), & \text{otherwise,} \end{cases} \quad (7)$$

where T_{comm} is the communication cost incurred in transmitting $k + |T_1|$ pages of data to the idle node, and T_{io} is the I/O cost to read these pages at the donor. Res_k is the estimated result size produced by the k pages and is estimated as follows:[11]

$$\frac{|Res_k|}{|Res|} = \frac{k}{n_1}.$$

When the task is an unprocessed task, $|Res|$ is estimated using Eq. (1) and $n_1 = |T_2|$. When the task is the currently processed task, $|Res|$ is the current result generated using n_1 pages of T_2. Recall that these are processing information and are maintained.

When there are unprocessed tasks and $k = |T_2|$, this implies that the entire task must be transferred. Thus the join cost can be computed from the cost formulas for T_{HNL}. Otherwise, only part of the unprocessed task is to be transferred. In this case, had the transferred loan been processed at the donor node, T_1 would have already been staged in memory. Hence, the cost to stage T_1 is not incurred at the donor. Therefore, T_{join}^{donor} is estimated as

$$T_{join}^{\text{donor}}(T_1|, k, |Res|) = \begin{cases} T_{\text{HNL}}(|T_1|, k, |Res|) & \text{if } k = |T_2| \\ T_{\text{join}}^{1}(|T_1|, k, |Res|) & \text{otherwise,} \end{cases}$$

where $T_{\text{join}}^{1}(r, s, |res|)$ is the cost to join r pages of the smaller bucket (which are already staged in memory) and s pages of larger bucket are used to probe for a match to generate result *res*.

[10] We have ignored the transmission cost of messages related to request for load transfer, since their sizes are expected to be small.

[11] For $|Res| = 0$, we use $|Res_k|/|Result| = k/(|T_2| - n_1)$.

$T_{\text{join}}^{\text{idle}}$ is more complex, since pipelining may or may not be explored. It is bounded as follows:

$$T_{\text{overhead}} + T_{\text{HNL}}(|T_1|, k, |Res|) - T_{\text{io}} \leq T_{\text{join}}^{\text{idle}}(|T_1|, k, |Res|) \leq T_{\text{overhead}} + T_{\text{HNL}}(|T_1|, k, |Res|).$$

The lower bound says that complete pipelining is feasible; that is, as the load arrives, it is processed immediately without the need to store the incoming stream to disk. On the other hand, the upper bound implies that no pipelining is possible. In both cases, an overhead is incurred. This overhead equals T_{overhead} since we assume total overlap in I/O and communication. k is determined using a binary algorithm.

Case 2. $|T_1| \cdot F > |M|$. On the other hand, if $|T_1| \cdot F > |M|$, then we should transfer a portion of the unprocessed T_1 and the entire T_2. If the task is the currently processed task, when all pages of T_1 have already been read, the number of pages to be transferred is determined in the same way as in Case (1) using the number of pages processed in the last loop instead of T_1. Otherwise, the two equations that meet the constraints and are used to determine k are

$$T_{\text{overhead}} \leq T_{\text{join}}^{\text{donor}}(k, |T_2|, Res_k) \quad (8)$$

and

$$ECT + T_{\text{overhead}} - T_{\text{join}}^{\text{donor}}(k, |T_2|, Res_k) \geq T_{\text{join}}^{\text{idle}}(k, |T_2|, Res_k). \quad (9)$$

$|Res_k|$ may be estimated using

$$\frac{|Res_k|}{|Res_1|/(L_2 - 1)} = \frac{k}{|M|/F}.$$

Again, in the case when $L_2 = 1$, we can derive $|Res_k|$ using an expression similar to Eq. (4).

To derive $T_{\text{join}}^{\text{donor}}$, several scenarios are considered. If, as a result of the load transferred, the number of loops to scan T_2 at the donor node (after the transfer) is reduced, it implies that had these loads been processed at the donor, then $T_{\text{join}}^{\text{donor}} = T_{\text{HNL}}$. Otherwise, since the amount of load transferred still did not reduce the number of loop to scan T_2, $T_{\text{join}}^{\text{donor}} = T_{\text{join}}^{2}$, where T_{join}^{2} is the cost to perform the join of the transferred load excluding the scanning cost for T_2.

$T_{\text{join}}^{\text{idle}}$ is bounded in a similar way as that for case (1). When k is small enough so that it can be staged in memory (that is, $k \leq |M|/F$), pipelining is feasible; otherwise (i.e., when $k > |M|/F$), the transferred load have to be stored before being processed.

For the scenario when the load is taken from an unprocessed task, the load is also determined by Eq. (8) and (9). If $k = |T_1|$, that is, the whole task is to be transmitted, $T_{\text{join}}^{\text{donor}} = T_{\text{HNL}}$. This same expression also holds when k

pages to be transmitted reduce the number of scan of T_2 at the donor node. As in the above case, if the number of scans is not reduced, $T_{join}^{donor} = T_{join}^{2}$. The same expression is used for T_{join}^{idle} and similar expressions for result sizes can be derived easily (assuming uniform distribution of data within the bucket).

3.4. Extended Dynamic Load-Balanced Join Algorithm EDBJ

Algorithm DBJ has the same disadvantage as SBJ when the relations are highly skewed—the skewed node would become a bottleneck in the task generation phase. EDBJ is designed to be more intelligent after the source relation is partitioned. Based on the skewness of the source relation, it will decide how to partition the target relation and allocate the tasks. The algorithm is essentially a hybrid of ASBJ and DBJ.

The task generation phase and allocation phase are similar to those in either algorithm ASBJ or DBJ. This is so because the algorithm essentially creates logical or physical tasks depending on the data skew of the source relation. First, the source relation is partitioned as in DBJ, after which each node reports the statistics to the coordinator. Based on the statistics, the coordinator estimates the severity of the data skew. If the relation is lightly skewed on the join attribute, then the target relation is partitioned as in DBJ, thus producing a set of physical tasks already allocated to the nodes. Otherwise, logical tasks are created and allocated as in ASBJ. That is, the target relation is partitioned and written back to the local disk. Based on statistics from both relations, the coordinator determines where the tasks should be allocated. The tasks are distributed accordingly.

The execution phase is the same as that in algorithm DBJ: the tasks are executed at the allocated nodes, followed by transfer of the load when there is load imbalance across the system nodes.

4. A PERFORMANCE STUDY

In this section, we present results from a simulation model developed to study the performance of the load-balanced join algorithms presented in Section 3. We use the parallel GRACE hash join algorithm, which does not perform any load-balancing, as a reference to study the benefits of load-balancing. We also model two versions of ASBJ, one where estimations based on the class-based technique are used in computing the result sizes and the cost of each task (denoted $ASBJ_e$) and another where perfect information is assumed (denoted $ASBJ_p$). We describe the simulation model and our findings here. The cost functions for the various algorithms, which compute the completion time of the join operation, are presented in Appendix I.

For this study, we assume that the values of the join column follow a *Zipf-like distribution* [11]. For a relation R with a domain of D distinct values, the i^{th} distinct join column value, for $1 \le i \le D$, has the number of tuples given by the expression

$$\|D_i\| = \frac{\|R\|}{i^{\theta} \cdot \sum_{j=1}^{D} 1/j^{\theta}}, \tag{10}$$

where θ is the skew factor. When $\theta = 0$, the distribution becomes *uniform*. With $\theta = 1$, it corresponds to the highly skewed *pure* Zipf distribution [26]. Though the join column is skewed, we assume that the relations to be joined are, initially, evenly distributed among the nodes to facilitate full concurrent access to the relations. The effect of different correlations between the skew values in the two relations is modeled in two ways:

- *Ordered correlation.* In this case, the values in both the attributes have the same ranking sequences. For example, the highest ranked value in attribute R_A of relation R is also the highest ranked value in the corresponding attribute S_A of relation S, the second highest ranked value in attribute R_A is also the second highest ranked value in the corresponding attribute S_A, and so on.
- *Random correlation.* As the name implies, this approach randomly correlates the join attribute in relation R and S.

As in [7], we first simulate the join execution on IBM RISC/6000 to obtain the size information, that is, the *actual* number of tuples of the source, target, and result relations, and the exact number of tuples transferred. The data are then used in the cost formulas. The distribution of data for the source and target relations is generated using Eq. (10). To compute the result size, we introduce another parameter, α, that determines the number of distinct tuples in the source relation that have a matching value in the target relation. For example, $\alpha = 0.5$ means that for each distinct value in the source relation, there is a probability of 0.5 that it will find a match in the target relation. α is modeled using a uniform distribution $UD(0, 1)$. When the ordered correlation is used, for each distinct value of the source relation examined, if the value of $UD(0, 1)$ is in the range 0–α, then the corresponding distinct value in the target relation is a match and the result size is the product of the number of tuples for this value in the two relations. Otherwise, the result size is 0. For random correlation, as each distinct value in the source relation is examined, a random tuple is picked from the target relation and the result is computed in the same manner as that of the ordered correlation.

For purpose of illustrating the performance study here, the default test values, which are similar to those used in [6], are shown in Table II. α is set to 0.5. We also vary the skew factor, number of nodes, relation sizes, disk I/O bandwidth, and communication bandwidth. We present only the results when ordered correlation is used. The results for random correlation are similar.

TABLE II
Parameter Settings

Parameter	Description	Default
$\|\|R\|\|$	Number of tuples in R	1,000,000
D	Number of distinct join attributes	10,000
t	Size in butes of each tuple	200
N	Number of nodes	64
M	Memory capacity	2 Mbytes per node
μ	CPU processing rate	20 MIPS
ω_{io}	I/O bandwidth	4 Mbytes/sec per node
ω_{comm}	Communication bandwidth	4 Mbytes/sec per node
I_{cpu}	Instruction pathlength	1000 instructions

4.1. Experiment 1: Effect of Relation Skew

In this experiment, we study how the skew factor will affect the performance of the algorithms. There are two scenarios that we are interested in—when both relations have the same skew and when both relations have different skews. Figure 2 shows the results when both the relations have the same skew factors and Table III summarizes the results when the relations have different skew factors.

Before we explain the results in Fig. 2, we must note that our study differs from that presented in [6] in that we include the cost of generating result tuples and writing the results to disk. This turns out to be a very important factor, especially for high data skew, since the result size of two highly skewed relations is a quadratic function of the relation sizes and hence could dominate the join cost. We will see this effect in the results presented in this section.

When the data skew is low (0.0–0.3), algorithms GRACE, SBJ, DBJ, and EDBJ perform equally well. This is so since all tasks have approximately the same processing time. Thus, the load in the system is almost balanced and there is little opportunity for load-balancing. As expected, both $ASBJ_e$ and $ASBJ_p$ are inferior to all the other algorithms at low data skew because of the extra I/O cost incurred in writing the partitions back to local disks during the task generation phase.

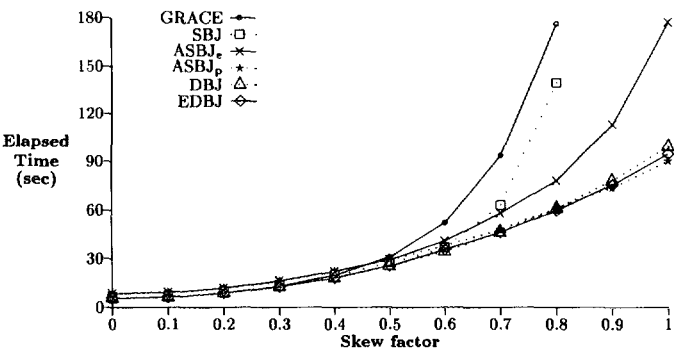

FIG. 2. Both relations have the same skew factor.

As the skew factor increases, some tasks are more time-consuming than others. Without balancing the load, as is done in GRACE, some processors will remain lightly loaded, while the heavily loaded ones will determine the completion time of the join resulting in high completion time. SBJ performs slightly better than GRACE since it exploits load-balancing. The gain is not significant because the balancing is coarse in that there is no transfer of partial load from a task. Moreover, any heavily loaded task, whether it be transferred or not, will also dominate performance. On the other hand, at high skew, both $ASBJ_e$ and $ASBJ_p$ improve the performance drastically—up to 70% improvement over GRACE. The gain is due to the splitting of an expensive task into smaller subtasks during the task generation phase, which resulted in an expensive task (which would have been allocated to a single node had there been no splitting) being distributed to several nodes.

We have also observed that $ASBJ_e$ performed significantly worse than $ASBJ_p$ when the skew factor is high. Recall that the difference between the two algorithms lie in the estimation of the join cost of each task, which is dependent on the bucket size of the task and its result size. While $ASBJ_p$ knows the bucket size and result size of each task, the result size is not known to $ASBJ_e$. $ASBJ_e$ estimates the result size using the class-based technique that keeps only statistics for a group of distinct join values. When both relations have low skew the estimation is not far from the actual size. On the other hand, the estimation error increases as the data skew increases. As a result, $ASBJ_p$ leads to better performance in general.

At high data skew, both algorithms DBJ and EDBJ outperform $ASBJ_e$. Several reasons account for this. First, while algorithm $ASBJ_e$ requires that a task be processed only at the allocated node, the dynamic algorithms allow a portion of a task to be transferred during join execution time. This allows the processing of a task to be shared by the donor and the idle nodes. In this way, the processing cost and the result are spread between the nodes. Spreading the result of a time-consuming task between the donor and the idle nodes also has the additional advantage of avoiding a single node from being the bottleneck (I/O and communication hotspot) when the results are to be merged. Second, at high data skew, time-con-

TABLE III
Both Relations Have Different Skew Factors

θ/θ	GRACE	SBJ	$ASBJ_e$	$ASBJ_p$	DBJ	EDBJ
Low/Medium	18.16	16.67	19.47	19.47	15.85	15.85
Low/High	141.85	123.16	120.43	120.32	37.99	37.79
Medium/Low	18.16	16.67	19.47	19.47	15.85	15.85
Medium/High	317.05	278.45	275.64	275.41	70.66	69.95
High/Low	141.85	123.16	120.43	120.32	37.99	33.02
High/Medium	317.05	278.45	275.64	275.41	70.66	65.39

suming tasks are not uncommon and hence the opportunities for task transfer increase. Third, the poor estimation of cost also accounts for the poor performance of $ASBJ_e$.

When perfect information are available, $ASBJ_p$ performs best at very high skew (skew factors of 0.9, 1.0). This is not unexpected since the accurate estimation allows a better task allocation for $ASBJ_p$. Moreover, the dynamic techniques incur additional overhead cost.

In Table III, for the column "θ/θ," the terms "Low," "Medium," and "High" refer to skew factors of 0.0, 0.5, and 1.0 respectively. Besides the same observations that we made above, we also noted the following from Table III. First, both versions of ASBJ perform poorly when only one of the relations is highly skewed. This is so because the algorithms split tasks only when the size of the smaller bucket is larger than the memory available. When one of the relations has low/medium skew, the splitting of tasks is reduced since most of the smaller bucket can be accommodated in memory. Second, unlike the case when both relations have the same skew factor (see Fig. 2 when the relations are highly skewed), the performance of $ASBJ_e$ is close to that of $ASBJ_p$ when the source and target relations have different skew factors. Third, for EDBJ, the choice of the source or target relation is important. A highly skewed target relation with a low/medium skew source relation is more expensive than when the order is interchanged. This is so since a low/medium skew source relation causes the highly skewed target relation to be partitioned as in GRACE, giving rise to the problem of GRACE—communication and I/O hotspot at the skewed node. We also note that both the dynamic algorithms are relatively close in performance. This implies that a straightforward implementation of dynamic load-balancing suffices to provide good performance. The added complexity of EDBJ, in particular the detection of the skewness of the source relation, does not provide any significant gain over DBJ.

We have also conducted some studies that reflect the cases where the join result is consumed by another operation such as aggregation. In such cases, the result is not written to disk. Our study shows that the relative performance of the algorithms is virtually unaffected. The relative difference between each pair of algorithms is within 5% of its counterpart when the result is written to disk.

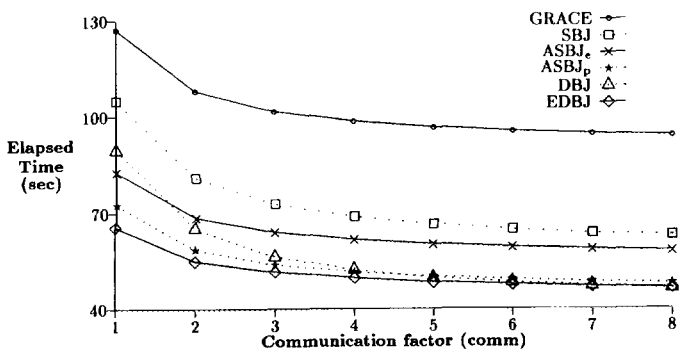

FIG. 3. Vary the communication channel bandwidth.

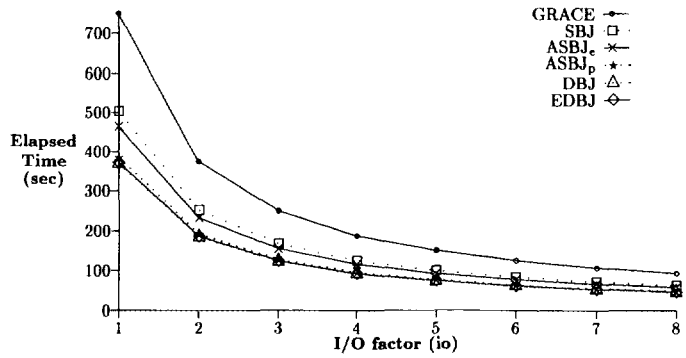

FIG. 4. Vary the I/O bandwidth.

4.2. Experiment 2: Effect of Communication Bandwidth

Here, we are interested in the effect of the communication bandwidth on the performance of the algorithms. The result is plotted in Fig. 3. The communication bandwidth varies according to the formulas $0.5 \cdot c$ Mbyte/s per node, where c is the communication factor. The skew factor is set to 0.7 in this study. The result shows that all algorithms perform better at high communication bandwidth. In particular, algorithm DBJ performs worse than $ASBJ_e$ and $ASBJ_p$ at low communication bandwidth. For DBJ, because of the high communication cost, lesser amounts of load are being transferred. Thus, the benefit of dynamically balancing the system load is smaller than the penalty in distributing the partitions of both relations during the partitioning phase. On the other hand, EDBJ is able to detect the skewness of the source relation and avoided the spreading of the target relation. In our study, we also noted that SBJ could perform worse than GRACE at low communication bandwidth. This is so since the load-balancing is done without considering the overhead incurred in transmitting the load.

4.3. Experiment 3: Effect of I/O Bandwidth

In this experiment, we vary the I/O bandwidth to study how different I/O systems may affect the algorithms. As in Experiment 2, we vary the I/O bandwidth from 0.5 to 4.0 Mbyte/s (i.e., I/O bandwidth = $io \cdot 0.5$ Mbyte/s, where io is the I/O factor) and the skew factor is 0.7. Figure 4 shows the results of this experiment. We observe that dynamic load-balancing provides a tremendous gain in performance at low I/O bandwidth (>50% over GRACE). This is so because the I/O cost for the results is spread across the donor node and the idle node. This implies that dynamic load-balancing is critical in systems with low I/O bandwidth.

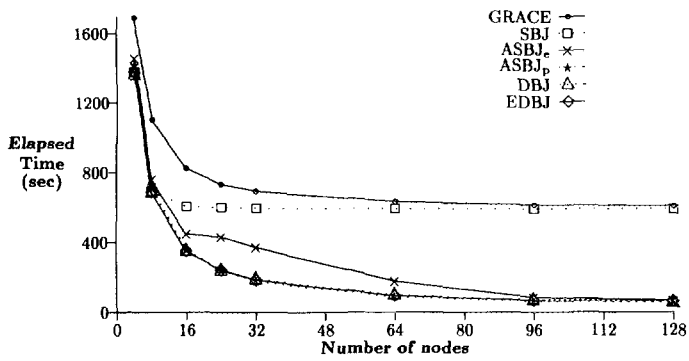

FIG. 5. Vary the number of nodes.

4.4. Experiment 4: Vary the Number of Nodes

In this experiment, we study the performance of the algorithms as the number of nodes varies. For purpose of illustration, we present here the results when both relations are highly skewed, i.e., $\theta = 1$. Similar behaviors were observed with other parameter settings. Figure 5 shows the result of the experiment. While the response time for all algorithms decreases as the number of nodes increases, the two dynamic algorithms (DBJ and EDBJ) clearly outperform the other methods. This is so because the two dynamic algorithms are able to redistribute the processing load at runtime. For algorithms GRACE, SBJ, and $ASBJ_e$, once a node completes the execution of all the tasks that are allocated to it, the node becomes idle. On the other hand, in algorithms DBJ and EDBJ, idle nodes will be allocated work from overloaded nodes. However, both DBJ and EDBJ are unable to achieve linear speedup. In particular, the gain in performance is significant with each additional node when the number of nodes is small (<16). Once the number of nodes reaches a certain value, the gain is not significant (>24). There are three possible reasons for this effect: First, there is an overhead in transferring the load. Second, less than half the load is transferred from the donor node to the idle node. Third, especially for high skew, when there is no further transfer possible, the processing load at the remaining active nodes may be time-consuming. Algorithm $ASBJ_p$ performs well as the number of nodes varies for high data skew. Like DBJ and EDBJ, it is not able to achieve linear speedup since the algorithm do not allow tasks to be migrated once they are allocated.

4.5. Experiment 5: Scaleup

We also study how the algorithms perform when the number of nodes in the system and the workload increases at the same time. Since only algorithms $ASBJ_e$, $ASBJ_p$, DBJ, and EDBJ are of interest, we present only the scaleup performance of these algorithms which is shown in Fig. 6. We vary the sizes of both relations from 25,000 tuples for a single node to 6,400,000 tuples for 128 nodes. In other words, with every 25,000 tuples increase in workload, a node is added to the system. The skew factor is set to 1.0.

From the figure, we see that the algorithms scale fairly well, especially for large numbers of nodes. The slight increase in elapsed time is due to the nonlinear increase in the size of the skewed buckets and the result. That is, while the sizes of the relations increases linearly, the sizes of the skewed buckets and the result increases faster than that. $ASBJ_e$ performs worse than the dynamic approaches for the same reasons as discussed in previous experiments.

5. CONCLUSION

In this paper, the problem of load-balancing for parallel join operations was addressed. Most parallel join algorithms adopt the task oriented approach. There are a number of design issues involved in each phase of the processing. We identified these issues and analyzed the possible solutions to derive a framework for designing load-balanced parallel join algorithms. The algorithms that appeared in the literature can be classified under this framework, and new algorithms can also be designed. Based on this, we proposed two new dynamic load-balanced join algorithms for shared-nothing systems. The unique feature of such dynamic load-balancing algorithms is that a task may be transferred, partially or as a whole, from the node to which it is originally allocated to a new node in the system. The performance of these two algorithms, together with that of another two algorithms that employ static load-balancing strategy, was studied. The GRACE hash join that performs no load-balancing was used as a reference to study the benefits of load-balancing. The results of this study showed that, although a sophisticated static load-balancing algorithm could handle the data skew problem quite well, inaccurate estimates could result in poor performance. On the other hand, dynamic load-balancing algorithms could improve the performance further, especially when data are highly skewed. We also see that a straightforward dynamic load-balancing algorithm such as DBJ will suffice to provide good performance. Adding more complexity

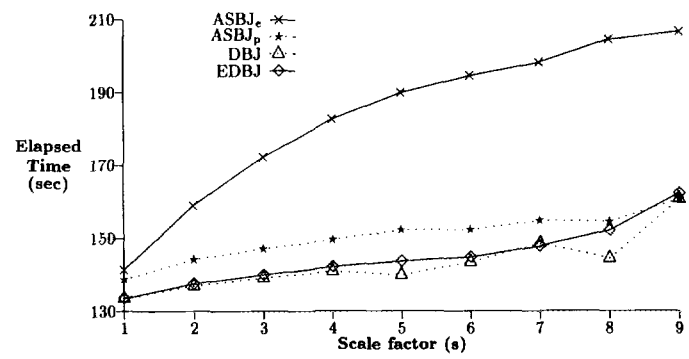

FIG. 6. Scaleup performance.

does not buy us much gain, as shown by EDBJ. We plan to extend this study to consider more complex queries. The overhead of serving the load request messages is also an issue to be addressed. Another extension is to explore how to serve multiple load request messages. We have also begun working on developing transfer strategies when the sort–merge join algorithm is used for the local joins.

APPENDIX

In this appendix, we present the cost functions for the join algorithms studied in this work. Recall that each of the join algorithms comprises several phases, each of which is made up of several steps. Since overlap can exist between and within the phases of the algorithms, we made the following assumptions, as in [6, 14]:

(a) There is total overlap within each step of each phase; i.e., the time to compute step i of phase j at node k is given by

$$T_{ij}^{k} = \max(t_{\text{cpu}}^{ijk}, t_{\text{io}}^{ijk}, t_{\text{comm}}^{ijk}),$$

where t_{cpu}^{ijk}, t_{io}^{ijk}, and t_{comm}^{ijk} are the CPU, I/O, and communication cost required for processing the ith step in phase j at node k. Thus for a *y-step* phase, the completion time for phase j at node k is given by

$$T_{j}^{k} = \sum_{i=1}^{y} T_{ij}^{k}.$$

(b) There is no overlap allowed between the steps of the phases and between the phases; i.e., step i must finish processing before step $i + 1$ begins processing, and phase j must finish processing before phase $j + 1$ begins processing.

Therefore, we can compute the elapsed time for an *n-phase* join algorithm running in a *p-node* system, $T_{\text{algorithm}}$, as

$$T_{\text{algorithm}} = \sum_{j=1}^{n} (\max_{k=1}^{p} T_{j}^{k}).$$

In the following analysis, we present only the cost functions for the various phases of an algorithm at an arbitrary node (since all nodes have the same cost function).[12] The notations used are shown in Table IV.

GRACE Hash Join

GRACE hash join consists of the following two distinct phases:

Phase 1. Task Generation Phase. In this phase, R and then S are scanned and distributed among the join nodes. The phase can be divided into two steps, corresponding to partitioning the two relations into buckets. The cost to partition R is given by

$$T_{\text{generate}}^{1a} = \max(T_{\text{cpu}}^{1a}, T_{\text{io}}^{1a}, T_{\text{comm}}^{1a}),$$

$$T_{\text{io}}^{1a} = \frac{\|R\|}{N} \cdot \frac{t}{\omega_{\text{io}}} + \sum_{j=1}^{\tau} \|B_j^r\| \cdot \frac{t}{\omega_{\text{io}}}$$

$$T_{\text{cpu}}^{1a} = \frac{\|R\|}{N} \cdot \frac{I_{\text{cpu}}}{\mu}$$

$$T_{\text{comm}}^{1a} = f_1 \cdot \frac{\|R\|}{N} \cdot \frac{t}{\omega_{\text{comm}}} + \left(\sum_{j=1}^{\tau} \|B_j^r\| - (1 - f_1) \cdot \frac{\|R\|}{N}\right) \cdot \frac{t}{\omega_{\text{comm}}}.$$

For the I/O cost, the first term is required to read R and the second is for writing the buckets of R that were allocated to the node. The CPU cost is simply the cost to initiate I/O, send and receive tuples, hash the tuples, and move them to the output buffers. The communication cost is required for the transmission of the tuples. Here the first term says that f_1 of the initial local fragment of R is transmitted, leaving behind $(1 - f_1)$ of the data. Since the final number of data of relation R at the node is $\sum_{j=1}^{\tau} \|B_j^r\|$, this implies that the difference must be the amount that is received by the node.

A similar expression can be derived for the cost to partition S. Denoting this cost as T_{generate}^{1b}, we have

$$T_{\text{generate}} = T_{\text{generate}}^{1a} + T_{\text{generate}}^{1b}.$$

TABLE IV
Notations Used

Notation	Meaning
$\|R\|$	Cardinality of a relation R
$\|Ex_i^{\text{send}}\|$	Cardinality of ith excess task sent
$\|Ex_i^{\text{rece}}\|$	Cardinality of ith excess task received
$\|B_i^r\|$	Cardinality of ith bucket of R
$\|B_i^s\|$	Cardinality of ith bucket of S
N	Total number of nodes in the system
t	Size of each tuple (in bytes)
M	Memory capacity for each node (in bytes)
f_1	Fraction of the initial local fragments of R that are transmitted
f_2	Fraction of the initial local fragments of S that are transmitted
μ	CPU processing rate in MIPS
ω_{io}	I/O bandwidth between a processor and its secondary storage
ω_{comm}	Effective communication channel bandwidth per node
I_{cpu}	Instruction pathlength
τ	Number of tasks at node
h	Fudge factor

[12] It should be noted that while several algorithms may share the same formulas in some phases, the values of the variables used in the cost formulas may be different.

Phase 2. Task Execution Phase. In this phase, all the nodes perform the joins of the tasks independently. Thus, there is no communication cost incurred, i.e., $T^2_{\text{comm}} = 0$. This phase comprises two steps. First, the hash table for a bucket of R is staged in memory. Second, the probing bucket is read and the join is performed. The cost for the first step is

$$T^{2a}_{\text{execute}} = \max(T^{2a}_{\text{cpu}}, T^{2a}_{\text{io}})$$

$$T^{2a}_{\text{io}} = \sum_{j=1}^{\tau} \|B^r_j\| \cdot \frac{t}{\omega_{\text{io}}}$$

$$T^{2a}_{\text{cpu}} = \sum_{j=1}^{\tau} \|B^r_j\| \cdot \frac{I_{\text{cpu}}}{\mu}.$$

The I/O cost is incurred to read the bucket in to build the hash table. The CPU cost is for building the hash table.

The cost for step 2 is

$$T^{2b}_{\text{execute}} = \max(T^{2b}_{\text{cpu}}, T^{2b}_{\text{io}})$$

$$T^{2b}_{\text{io}} = \left(\sum_{j=1}^{\tau} \left[h \cdot \|B^r_j\| \cdot \frac{t}{M}\right] \cdot \|B^s_j\| + \sum_{j=1}^{\tau} \|Res_j\|\right) \cdot \frac{t}{\omega_{\text{io}}}$$

$$T^{2b}_{\text{cpu}} = \left(\sum_{j=1}^{\tau} \left[h \cdot \|B^r_j\| \cdot \frac{t}{M}\right] \cdot \|B^s_j\| + \sum_{j=1}^{\tau} \|Res_j\|\right) \cdot \frac{I_{\text{cpu}}}{\mu}.$$

The I/O cost is for reading the probing bucket (as many times as needed to stage the corresponding R bucket) and to write the result to disk. The CPU cost is incurred for probing for a match and to generate the resulting tuples. Therefore,

$$T_{\text{execute}} = T^{2a}_{\text{execute}} + T^{2b}_{\text{execute}}.$$

Static Load Balanced Hash Join (SBJ)

For SBJ, the *task generation phase* and *task execution phase* have the same cost formulas as for GRACE hash join and so we will not repeat them here. We will only present the cost formulas for the *load balancing phase.*

Phase 2. Load Balancing Phase. Recall that it is the excess tasks that are transmitted—either sent from an overloaded node or received by an underloaded node. Note that each task is composed of two buckets. The I/O cost incurred would then be the cost to read the excess tasks to be transmitted and to write the excess tasks received. This is given as

$$T^2_{\text{io}} = \left(\sum_{j=1}^{\tau_1} \|Ex^{\text{send}}_j\| + \sum_{j=1}^{\tau_2} \|Ex^{\text{rece}}_j\|\right) \cdot \frac{t}{\omega_{\text{io}}},$$

where τ_1 and τ_2 represent the numbers of excess tasks that are transmitted and received, respectively. Based on the description of the algorithm, it should be clear that either $\tau_1 = 0$ or/and $\tau_2 = 0$.

We assume that the CPU time incurred is negligible; i.e.,

$$T^2_{\text{cpu}} = 0.$$

The corresponding communication time is

$$T^2_{\text{comm}} = \left(\sum_{j=1}^{\tau_1} \|Ex^{\text{send}}_j\| + \sum_{j=1}^{\tau_2} \|Ex^{\text{rece}}_j\|\right) \cdot \frac{t}{\omega_{\text{comm}}}.$$

Thus,

$$T_{\text{balance}} = \max(T^2_{\text{io}}, T^2_{\text{cpu}}, T^2_{\text{comm}}).$$

Adaptive Static Load Balanced Hash Join (ASBJ)

Based on the description of the algorithm ASBJ in Section 3, the *task execution phase* has the same cost formulas as GRACE hash join. We will only present the cost formulas for the *task generation phase* and the *load-balancing phase.*

Phase 1. Task Generation Phase. In this phase, the processing of each relation corresponds to a step. Since all processings are performed locally, there is no communication cost incurred. Therefore, the cost to perform this phase is

$$T_{\text{partition}} = \max(T^{1a}_{\text{io}}, T^{1a}_{\text{cpu}}) + \max(T^{1b}_{\text{io}}, T^{1a}_{\text{cpu}}).$$

The I/O times for the two steps are

$$T^{1a}_{\text{io}} = 2 \cdot \frac{\|R\|}{N} \cdot \frac{t}{\omega_{\text{io}}}$$

$$T^{1b}_{\text{io}} = 2 \cdot \frac{\|S\|}{N} \cdot \frac{t}{\omega_{\text{io}}}.$$

The corresponding CPU times for the steps are

$$T^{1a}_{\text{cpu}} = \frac{\|R\|}{N} \cdot \frac{I_{\text{cpu}}}{\mu}$$

$$T^{1b}_{\text{cpu}} = \frac{\|S\|}{N} \cdot \frac{I_{\text{cpu}}}{\mu}.$$

Phase 2. Load-Balancing Phase. In this phase, the tasks are distributed in two steps. First, buckets from relation R are distributed, followed by buckets from relation S. We assume that the CPU cost to redistribute the task in this phase is negligible. Thus, the cost for this phase is

$$T_{\text{balance}} = \max(T^{2a}_{\text{io}}, T^{2a}_{\text{comm}}) + \max(T^{2b}_{\text{io}}, T^{2b}_{\text{comm}}).$$

$$T^{2a}_{\text{io}} = \left\{f_1 \cdot \frac{\|R\|}{N} + \left(\sum_{j=1}^{\tau} \|B^r_j\| - (1 - f_1) \cdot \frac{\|R\|}{N}\right)\right\} \cdot \frac{t}{\omega_{\text{io}}}$$

$$T_{\text{io}}^{2b} = \left\{ f_2 \cdot \frac{\|S\|}{N} + \left(\sum_{j=1}^{\tau} \|B_j^s\| - (1 - f_2) \cdot \frac{\|S\|}{N} \right) \right\} \cdot \frac{t}{\omega_{\text{io}}}$$

$$T_{\text{comm}}^{2a} = \left\{ f_1 \cdot \frac{\|R\|}{N} + \left(\sum_{j=1}^{\tau} \|B_j^r\| - (1 - f_1) \cdot \frac{\|R\|}{N} \right) \right\} \cdot \frac{t}{\omega_{\text{comm}}}$$

$$T_{\text{comm}}^{2b} = \left\{ f_2 \cdot \frac{\|S\|}{N} + \left(\sum_{j=1}^{\tau} \|B_j^s\| - (1 - f_2) \cdot \frac{\|S\|}{N} \right) \right\} \cdot \frac{t}{\omega_{\text{comm}}}.$$

Dynamic Load Balanced Hash Join (DBJ)

From the description of the algorithm DBJ in Section 3, the *task generation phase* has the same cost formulas as GRACE hash join. However, since the algorithm, is dynamic at the *task execution phase*, the time for the phase is obtained only at runtime, depending on when the load is transferred and the amount of the load transferred.

Extended Dynamic Load Balanced Hash Join (EDBJ)

There are two cost functions for algorithm EDBJ depending on the skewness of the join attribute of the source relation.

Source Relation Has Low Data Skew. In this case, the cost formulas are derived in the same way as for algorithm DBJ.

Source Relation Has High Data Skew. The cost formulas for the algorithm are derived in the same way as for algorithm DBJ, except for the *task generation phase*, which is given by

$$\begin{aligned} T_{\text{generate}} = {} & \max(T_{\text{cpu}}^{1a}, T_{\text{io}}^{1a}, T_{\text{comm}}^{1a}) + \max(T_{\text{cpu}}^{1b}, T_{\text{io}}^{1b}, T_{\text{comm}}^{1b}) \\ & + \max(T_{\text{cpu}}^{1c}, T_{\text{io}}^{1c}, T_{\text{comm}}^{1c}) \\ & + \max(T_{\text{cpu}}^{1d}, T_{\text{io}}^{1d}, T_{\text{comm}}^{1d}). \end{aligned}$$

The I/O, CPU, and communication costs for step 1, which partitions relation R as in GRACE hash join algorithm, are

$$T_{\text{io}}^{1a} = \left(\frac{\|R\|}{N} + \sum_{j=1}^{\tau} \|B_j^r\| \right) \cdot \frac{t}{\omega_{\text{io}}}$$

$$T_{\text{cpu}}^{1a} = \frac{\|R\|}{N} \cdot \frac{I_{\text{cpu}}}{\mu}$$

$$T_{\text{comm}}^{1a} = \left\{ f_1 \cdot \frac{\|R\|}{N} + \left(\sum_{j=1}^{\tau} \|B_j^r\| - (1 - f_1) \cdot \frac{\|R\|}{N} \right) \right\} \cdot \frac{t}{\omega_{\text{comm}}}.$$

Step 2 partitions S locally, giving rise to the following I/O, CPU, and communication times:

$$T_{\text{io}}^{1b} = 2 \cdot \frac{\|S\|}{N} \cdot \frac{t}{\omega_{\text{io}}}$$

$$T_{\text{cpu}}^{1b} = \frac{\|S\|}{N} \cdot \frac{I_{\text{cpu}}}{\mu}$$

$$T_{\text{comm}}^{1b} = 0.$$

In step 3, R is redistributed with the times

$$T_{\text{io}}^{1c} = \left\{ f_3 \cdot \sum_{j=1}^{\tau} \|B_j^r\| + \left(\sum_{j=1}^{\tau} \|B_j^{r'}\| - (1 - f_3) \cdot \sum_{j=1}^{\tau} \|B_j^r\| \right) \right\} \cdot \frac{t}{\omega_{\text{io}}}$$

$$T_{\text{cpu}}^{1c} = 0$$

$$T_{\text{comm}}^{1c} = \left\{ f_3 \cdot \sum_{j=1}^{\tau} \|B_j^r\| + \left(\sum_{j=1}^{\tau} \|B_j^{r'}\| - (1 - f_3) \cdot \sum_{j=1}^{\tau} \|B_j^r\| \right) \right\} \cdot \frac{t}{\omega_{\text{comm}}},$$

where f_3 is the fraction of the buckets of R that needs to be redistributed at a node and $\sum_{j=1}^{\tau} \|B_j^{r'}\|$ is the sum of the sizes of the new sets of buckets of R after redistribution.

Finally, the I/O, CPU, and communication times to distribute S in step 4 are

$$T_{\text{io}}^{1d} = \left\{ f_2 \cdot \frac{\|S\|}{N} + \left(\sum_{j=1}^{\tau} \|B_j^s\| - (1 - f_2) \cdot \frac{\|S\|}{N} \right) \right\} \cdot \frac{t}{\omega_{\text{comm}}}$$

$$T_{\text{cpu}}^{1d} = 0$$

$$T_{\text{comm}}^{1d} = f_2 \cdot \frac{\|S\|}{N} \cdot \frac{t}{\omega_{\text{comm}}} + \left(\sum_{j=1}^{\tau} \|B_j^s\| - (1 - f_2) \cdot \frac{\|S\|}{N} \right) \cdot \frac{t}{\omega_{\text{comm}}}.$$

ACKNOWLEDGMENTS

We thank Guy Lohman, Eugene Shekita, John McPherson, and Honesty Young, all from IBM's Almaden Research Center, for inspiring this work. The anonymous referees have also provided comments and suggestions that greatly improved the presentation of this paper.

REFERENCES

1. DeWitt, D. J. Parallel database systems: The future of high performance database systems. *Comm. ACM* **35,** 6 (June 1992), 85–98.
2. DeWitt, D. J., Ghandeharizadeh, S., Scheneider, D. A., Bricker, A., Hsiao, H.-I. and Rasmussen, R. The gamma database machine project. *IEEE Trans. Knowledge Data Engrg.* **2,** 1 (Mar. 1990), 44–62.
3. DeWitt, D. J., Naughton, J. F., Schneider, D. A., and Seshadri, S. Practical skew handling in parallel joins. *Proc. of the 18th International Conference on Very Large Data Bases*, Vancouver, Canada, August 1992.
4. Englert, S., Gray, J., Kocher, T., and Shah, P. A benchmark of nonstop sql release 2 demonstrating near-linear speedup and scaleup on large databases. Technical Report Technical Report 89.4, Tandom Computer Inc., 1989.

5. Graham, R., Bounds on multiprocessing timing anomalies. *SIAM J. Comput.* **1,** 7 (1969), 416–429.
6. Hua, K. A., and Lee, C. Handling data skew in multiprocessor database computers using partition tuning. *Proc. of the 17th International Conference on Very Large Data Bases*. Barcelona, Sept. 1991, pp. 525–535.
7. Hua, K. A., Lo, Y. L., and Young, H. C. Including the load balancing issue in the optimization of multi-way join queries for shared-nothing database computers. *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*. San Diego, Jan. 1993.
8. Keller, A. M., and Roy, S. Adaptive parallel hash join in main-memory databases. *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. Miami Beach, Dec. 1991, pp. 58–67.
9. Kitsuregawa, M., and Ogawa, Y. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (scd). *Proc. of the 16th International Conference on Very Large Data Bases*. Brisbane, Aug. 1990, pp. 210–221.
10. Kitsuregawa, M., Tanaka, H., and Moto-oka, T. Application of hash to data base machine and its architecture. *New Generation Comput.* **1,** 1 (1983).
11. Knuth, D. E. *The Art of Programming, Vol. 3: Sorting and Searching*. Addison–Wesley, Reading, MA, 1973.
12. Lakshmi, M. S., and Yu, P. S. Effectiveness of parallel joins. *IEEE Trans. Knowledge Data Engrg.* **2,** 3 (Sept. 1990), 410–424.
13. Lu, H., and Tan, K. L. Dynamic and load-balanced task-oriented database query processing in parallel systems. *Proceedings of the Third International Conference on Extending Data Base Technology*. Vienna, Mar. 1992.
14. Lu, H., Tan, K. L., and Shan, M. C. Hash-based join algorithms for multiprocessor computers with shared memory. *Proc. of the 16th International Conference on Very Large Data Bases*. Brisbane, Aug. 1990, pp. 198–209.
15. Nakayama, M., and Kitsuregawa, M. Hash-partitioned join method using dynamic destaging strategy. *Proc. of the 14th International Conference on Very Large Data Bases*. Los Angeles, Aug. 1988, pp. 468–478.

Received October 27, 1992; revised September 29, 1993; accepted October 3, 1993

16. Omiecinski, E. Performance analysis of a load-balancing relational hash join algorithm for a shared-memory multiprocessor. *Proc. of the 17th International Conference on Very Large Data Bases*. Barcelona, Sept. 1991.
17. Richardson, J. P., Lu, H., and Mikkilineni, K. Design and evaluation of parallel pipelined join algorithms. *Proceedings of ACM International Conference on Management of Data*. San Francisco, May 1987, pp. 399–409.
18. Schneider, D. A., and DeWitt, D. J., A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *Proceedings of ACM International Conference on Management of Data*. Portland, June 1989, pp. 110–121.
19. Stamos, J. W., and Young, H. C. A symmetric fragment and replicate algorithm for distributed joins. Technical Report Research Report RJ 7188, IBM Almaden Research Center, 1989.
20. Swami, A., and Young, H. C. Online algorithms for handling skew in parallel joins. Technical Report Research Report RJ 8363, IBM Almaden Research Center, Sept. 1991.
21. Teradata Corporation. Dbc/1012 database computer concepts and facilities, rel. 3.1 edition, Teradata document c02-0001-05. Los Angeles, 1988.
22. Valduriez, P., and Gardarin, G. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Systems,* **9,** 1 (Mar. 1984), 133–161.
23. Walton, C. B., Dale, A. G., and Jenevein, R. M. A taxonomy and performance model of data skew effects in parallel joins. *Proc. of the 17th International Conference on Very Large Data Bases*. Barcelona, Sept. 1991, pp. 536–548.
24. Wolf, J. L., Dias, D. M., Yu, P. S., and Turek, J. J. An effective algorithm for parallelizing sort merge joins in the presence of data skew. *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*. Dublin, Dec. 4–6, 1990, pp. 103–115.
25. Wolf, J. L., Dias, D. M., Yu, P. S., and Turek, J. J. An effective algorithm for parallelizing hash joins in the presence of data skew. *Proceedings of the Seventh International Conference on Data Engineering*. Kobe, Apr. 1991, pp. 200–209.
26. Zipf, G. K. *Human Behavior and the Principle of Least Effort*. Addison–Wesley, Reading MA, 1949.