Parallel Algorithms for the Execution of Relational Database Operations

DINA BITTON, HARAN BORAL, DAVID J. DEWITT, and W. KEVIN WILKINSON
University of Wisconsin

This paper presents and analyzes algorithms for parallel processing of relational database operations in a general multiprocessor framework. To analyze alternative algorithms, we introduce an analysis methodology which incorporates I/O, CPU, and message costs and which can be adjusted to fit different multiprocessor architectures. Algorithms are presented and analyzed for sorting, projection, and join operations. While some of these algorithms have been presented and analyzed previously, we have generalized each in order to handle the case where the number of pages is significantly larger than the number of processors. In addition, we present and analyze algorithms for the parallel execution of update and aggregate operations.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*query processing*; H.2.6 [**Database Management**]: Database Machines

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Sorting, projection operator, join operation, aggregate operations, database machines, parallel processing

1. INTRODUCTION

Research on algorithms for database machines which support massive parallelism in tightly coupled multiprocessor systems has, for the most part, been "architecture directed." That is, database machine designers usually begin by designing what they consider to be a good architecture and only afterward develop the algorithms to support database operations using the basic primitives of their architecture. As an example consider associative disks (or logic-per-track devices) [14] from which RAP [13], RARES [12], CASSM [16], and to some extent, DBC [2] are derived. The basic design goal of the associative disk design was the efficient execution of the selection operation to select records which satisfy a certain criterion. Given this building block, other relational database operators such as join, project, and update can be implemented with varying degrees of success (see [7]). In general, this is done by combining the processing capabilities of the host with those of the back-end database machine. The

This research was supported in part by the National Science Foundation under grant MCS-78-01721 and in part by the United States Army under contracts DAAG29-79-C-0165 and DAAG29-75-C-0024.
Authors' address: Department of Computer Science, 1210 W. Dayton St., University of Wisconsin, Madison, WI 53706.

designers of RAP recognized the limitations of the pure associative disk design and added interconnections between the processing elements to facilitate processing of certain interrelation operations such as join. On the other hand, the designers of the DBC started with the recognition that an entire database could never be stored on logic-per-track devices in a cost-effective manner. Consequently, they concentrated on designing a machine to facilitate the use of indices so that moving head disks with a processor per head instead of a processor per track could be utilized efficiently.

It is our thesis that for a database machine design to be "successful," the following design procedure must be followed. First, a thorough study of algorithms for all the operations to be supported by the machine must be undertaken. Next, the algorithms must be analyzed in terms of primitive operations, such as read a block of data, send a message, and sort a block of data. Finally, various hardware organizations must be examined to determine their suitability for the implementation of the algorithms for *all* the operations. It will most likely be necessary to repeat this process so that the final machine organization can be implemented within certain cost boundaries.

It is certainly the case that this procedure cannot be used by every computer system designer. It may be the case that a priori information about the makeup of programs to be executed is not available, or that the information is of such breadth as to render it useless. However, storage structures used in relational databases and the relational operators are well understood, thereby enabling relational database machine designers to follow this proposed course of action.

How are the different algorithms to be evaluated? There are two possible approaches that one could adopt. The first is to use a general-purpose machine with capabilities to support any conceivable algorithm. This would enable the comparison of the various algorithms using a uniform set of assumptions. Alternatively, one could specify the "ideal" machine organization for each algorithm. In this approach each algorithm's execution would attain its optimal performance. However, comparing the performance of two algorithms for the same operator would be considerably more difficult than with the general-purpose machine approach.

In either case the evaluation of the algorithms must be sensitive to the following two points:

(1) costs of performing all aspects of the computation, including processing, communication, and I/O costs;
(2) performance of the algorithms when the number of available processors is smaller than the desirable number of processors and the amount of "main memory" available is less than the size of the relations being processed; examination of performance means that the algorithms must be external.

In this paper we describe and analyze algorithms for the relational operators using the general-purpose machine approach. We chose to follow this route for two reasons. First, by comparing the algorithms in the general-purpose machine approach we felt that we would develop a better understanding of the strengths and weaknesses of the various algorithms. Second, since the machine model we selected is very similar to DIRECT [6], this approach gave us an opportunity to compare the performance of the algorithms used by DIRECT with alternative algorithms for each of the relational operators without going to the trouble of actually implementing these algorithms in DIRECT.

While we feel that this paper makes a number of contributions to the database machine literature, it does, however, have several limitations. First, the use of indices as a tool for implementing algorithms for complex relational operators has not been explored. While it seems feasible to use indices in a parallel algorithm for execution of the selection operation (something which we failed to realize in the design and implementation of DIRECT), their use in parallel algorithms for complex operations appears to be a very difficult problem. Consider, for example, a complex relational query comprised of several selection operations and a join operation. If indices are to be used to process the join, these indices must be created by the multiple processors being used to execute each of the selection operations. The problem of synchronizing access to the index without completely serializing the actions of the processors executing in parallel is a very difficult problem, one that we have not been able to solve.

A second limitation is that we have ignored the impact of concurrency control and recovery on the performance of the parallel update algorithms described in Section 4. This is an area of research we are currently pursuing.

In Section 2 we describe the properties of the multiprocessor organization used for the evaluation of the algorithms. In Section 3 we introduce the analysis techniques and assumptions that we use to evaluate the different parallel algorithms. Parallel algorithms for update, sorting, projection, join, and aggregate operations are presented and evaluated in Section 4. Our conclusions and areas for future research are discussed in Section 5.

2. A GENERAL MULTIPROCESSOR ORGANIZATION

The organization of the multiprocessor used for the evaluation of our parallel algorithms consists of the following components:

(1) a set of general-purpose processors,
(2) a number of mass storage devices,
(3) an interconnection device connecting the processors to the mass storage devices via a high-speed cache.

Such an organization is shown in Figure 1. The processors are responsible for executing relational database operations and operate independently. Therefore, the processors form a multiple instruction stream, multiple data stream (MIMD) machine. Since the multiprocessor organization is intended to serve as a back-end database machine, one of the processors is chosen to act as an interface to a host processor (the processor with which a user interacts). It is the responsibility of this processor to also act as controller to coordinate the activities of the other processors. After a user submits a query for execution, the host will compile the query and send it to the controller for execution on the database machine.

The memory hierarchy consists of three components. The top level consists of the internal memories of all the processors. Each processor's local memory is assumed to be large enough to hold both a compiled query and three pages of data. At the bottom level of the memory hierarchy are the mass storage devices

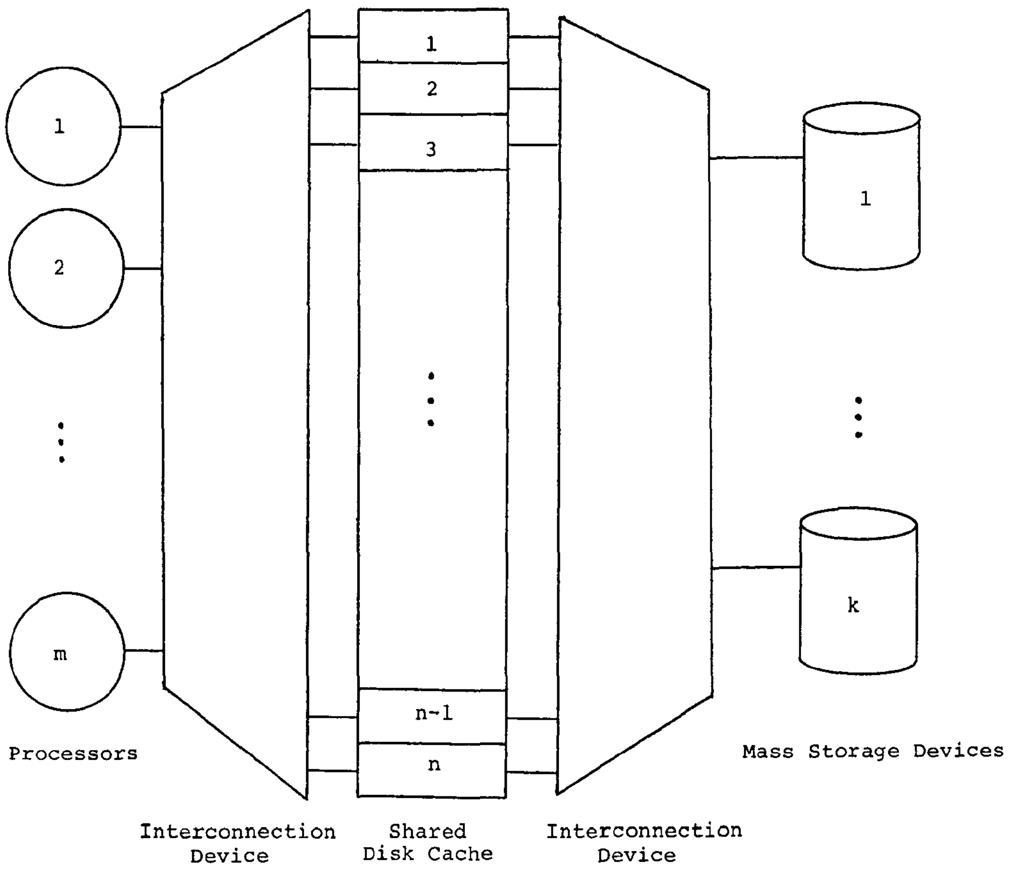

Fig. 1. Organization of a generalized multiprocessor.

used to hold the relations in the database. The middle level of the hierarchy is a disk cache which is addressable in page units. A page of a relation is the unit of transfer between all levels of the memory hierarchy.

The bottom two levels of the memory hierarchy are connected together in a way that allows for data transfer between each mass storage device and any page frame in the disk cache. The top two levels of the hierarchy are connected together by an interconnection device with the following two properties. The first is that several processors can read or write a different page of the disk cache simultaneously. The second is the ability to broadcast the contents of a page frame of the disk cache to any number of processors. Note that such an organization may not be cost effective and we are not advocating it in this paper. Our purpose is merely to use it to describe and evaluate algorithms.

A final and very important point is that both the total memory of the processors and the size of the disk cache are generally not large enough to contain a whole relation. Therefore we cannot assume that a whole relation can be read from mass storage to either the processor's local memory or the disk cache before processing begins. Accordingly, I/O costs become a significant factor in the evaluation of the various algorithms. One consequence of this (very realistic) assumption is that all the algorithms employed must be external.

3. ANALYSIS PARAMETERS

In this section we describe the parameters used in the analyses of the various algorithms. We have chosen to represent fixed costs by capital letters. Other parameters (for example, the number of pages to be read) are represented by lowercase letters. We assume that data are moved and processed by page units. A full page contains k tuples; C is the cost of a simple operation such as comparing two attributes or performing an addition; and the cost of moving a tuple inside a page is V time units. The basic tasks used in evaluating the performance of our algorithms are

(1) *Communication Cost.* In this architecture transfer of pages are considered I/O operations. In order to read or write a page, a processor must request a page frame number in the cache from the controller. We term messages exchanged for this purpose as "I/O related messages." The cost of such messages is denoted by C_{msg}. The remaining communication cost of an algorithm can be measured by the number of control messages sent. Examples of control messages are messages necessary to allocate processors to an operation, synchronization messages indicating the end of a phase, and the initiation of a new phase during the execution of an algorithm. Since we feel that the number of control messages is small compared with the number of I/O related messages and since these messages are short (they contain only a few words of information), we are neglecting them when we compare the cost of several algorithms. While this may not be a reasonable assumption, the cost of controlling cooperating processors is an unexplored (and difficult) area requiring additional research.

(2) *I/O Cost.* A read request moves a page into a processor's memory from the cache (fetching it first from mass storage if necessary). A write request always moves a page residing in a processor's local memory to the cache. When a processor wants to read or write a page, it sends a request message to the controller specifying the relation name and the page number. The controller replies by sending to the processor a cache frame number. We denote the cost of a mass storage to cache transfer by R_m and the cost of a cache to processor transfer by R_c. An upper bound for the read cost is achieved by assuming that all read operations are from the mass storage device (i.e., the cost of any read is $R_m + R_c$). A lower bound results from assuming that all read operations are from the cache, in which case a read cost is R_c. To simplify our analysis we assume a certain hit ratio for the cache, denoted by H. Since the entire relation is to be referenced in processing a query (recall that there are no indices), the reference string is known and pages can be prefetched from the mass storage devices to the disk cache. This can result in a high value for H. Given the values for R_c, R_m, H, and C_{msg}, we can calculate C_r, the average cost of a read, by a processor:

$$C_r = HR_c + (1 - H)(R_c + R_m) + 2C_{msg}.$$

Similarly, in order to calculate the average cost to write a page, we assume that H' is a fraction describing the amount of time a free page frame will be available in the cache during a write operation. Thus C_w, the average cost of writing a page, is

$$C_w = H'R_c + (1 - H')(R_c + R_m) + 2C_{msg}.$$

In order to analyze the algorithms in Section 4 we were forced to assume that the cache and underlying I/O system have sufficient bandwidth to permit p read or write operations to proceed simultaneously. For some number of processors p, this assumption probably becomes invalid. An interesting extension to our analysis would be to develop a more detailed model of the cache and I/O system which would more accurately reflect their impact on the performance of the various algorithms. We have performed such an analysis for parallel sorting algorithms in [9].

(3) *Scan Cost.* If a page is to be scanned, the scan is sequential. The number of tuples in the page is assumed to be k. Thus the scan cost C_{sc} is computed as

$$C_{sc} = kC.$$

(4) *Merge Cost.* If two sorted pages are to be merged, the number of tuples in each page is assumed to be k. Since all our operations require internally sorted pages (see Section 4.1), both pages will already be sorted and thus the worst case number of comparisons required to perform the merge of two sorted lists of length k is $2k$ [11]. The number of tuples to be moved is the same. Thus C_m, the cost of merging two pages, is computed as

$$C_m = 2k(C + V).$$

(5) *Page Reorganization Cost.* There are two cases when a page must be reorganized to keep the tuples in sorted order. The first case occurs after the application of an update operation which modifies the attribute on which the page is sorted. We assume that the reorganization consists of both tuple comparisons and movements and expect that, on the average, half of the tuples in the page will be affected. As before, a page is assumed to have k tuples. We compute C_o, the reorganization cost, as follows:

$$C_o = \tfrac{1}{2}k(C + V).$$

The second case occurs when a buffer containing new tuples (e.g., the result of a projection or a page of an intermediate relation) is to be used in a subsequent operation. Since all our operations require internally sorted pages, the page must be sorted before it is written to disk. We assume that the new page has k tuples (though in some cases this number may be smaller) and that, on the average, internal sorting of a page would require $k \log k$ comparisons and moves.[1] Thus C_{so}, the cost to internally sort a page, is

$$C_{so} = (k \log k)(C + V).$$

For our analysis of project, sort, and join algorithms we found it convenient to group some of the above parameters and to define the following "2-page operation:"

$$C_p^2 = 2C_r + C_m + 2C_w$$

[1] Throughout this paper we assume all logs are to the base 2.

is the cost of a 2-page operation and consists of reading 2 sorted pages, merging them, and writing the resulting sorted block of 2 pages.

4. PARALLEL ALGORITHMS FOR DATABASE OPERATIONS

In this section we present and evaluate parallel algorithms for update operations, sorting, projection, join, and aggregate operations using the analysis techniques described in the previous section. *Each algorithm presented is intended to handle the general case where the number of pages to be processed is significantly larger than the number of processors available.* This implies that the operand relation or relations cannot be brought entirely into the processors' memories for processing. Several passes over the data are necessary. Intermediate results not used in a pass must be written out to the cache (and possibly the disk). We begin with a presentation of a set of update algorithms which maintain each page in sorted order. Since sorting will be used as a basic step in the project, join, and aggregate operations, it is presented second. Finally, the project, join, and aggregate operations are presented.

4.1 Update Algorithms

Many of the retrieval algorithms presented in the following sections rely on the property that each page is sorted on some attribute or group of attributes. Permanent relation pages are sorted on the relation key. It follows then that any update algorithm must keep the pages sorted. A second property that must be preserved is that no duplicates are introduced as a result of an update. We show that our algorithms do indeed preserve these properties. An analysis of one algorithm's complexity is presented in Appendix A.

We consider three update operations: delete, append, and modify. Each operation specifies a relation to be updated and a qualification clause specifying which tuples of the relation are to be affected. For example,

Delete emp where emp.eno $<$ 153.

However, there may be cases where the selection criteria for an update operation is more complex than a simple selection. For example,

Delete emp where emp.eno $<$ 153 and emp.dno $=$ dept.dno and dept.name $\neq$ "toy".

Here we have to restrict both the employee and department relations according to the selection criteria, perform the join, and then apply the delete operation to the employee relation using the values produced by the join as the deletion criteria.

We term these two kinds of qualification clauses simple and complex. A simple qualification is one that may be applied in a single scan of the relation. A complex qualification is one which requires us to perform some interrelation operation or operations (e.g., join) in order to determine the tuples to be updated. The algorithms presented below handle both simple and complex updates.

For consistency, we assume that updates are atomic operations; that is, an update either successfully terminates, or in the event of a crash or abort, does not affect the stored database. One reason for aborting update operations is the introduction of duplicates into a relation.

4.1.1 *Delete.* A deletion operation is, in effect, the negation of a selection. If the qualification is simple, each processor executing the deletion will examine a unique subset of source relation pages. Tuples satisfying the deletion criterion are removed from the page and the page is compressed and written to the cache. The controller is informed of the size of the new page and stores it as a new page of the relation.

Complex deletes require a preprocessing step to determine the set of tuples to be removed. The set produced by this step is a list of database keys henceforth referred to as Q, which is produced by executing the qualification clause of the update operation. This clause typically consists of a number of retrieval operations such as selection and projection. The same processors assigned to execute the deletion, perhaps augmented with other processors, first execute the qualification clause. Once Q is produced it must be distributed to the processors which perform the deletion. One possibility is to attach Q to the code segment as a data structure. This approach is feasible if the size of Q is small (a few database keys). Otherwise, Q can be broadcast to all the processors that have pages of the source relation. Each processor would then perform a modified merge of its source page with every page in Q. The modified merge would consist of *deleting* a tuple from the *source* relation page if a key value in Q matches the tuple's key. As in simple deletes, modified pages are written out as new pages of the relation replacing the corresponding source page.

4.1.2 *Append.* A simple append is one in which a small number of tuples are to be appended to a relation. The simple append begins with the controller deciding where to add the additional tuples, based on the density of the pages in the relation. The processors first search for duplicates of those tuples to be appended. If duplicates are found by any of the processors, the controller is informed, the operation aborted, and the relation restored to its pre-append state. If no duplicates are found, tuples are then added to the pages designated by the controller. A page chosen for appending will have to undergo reorganization to preserve its sort order.

Complex appends are executed in a similar manner to complex deletes. After the list of tuples to be appended has been generated, the processors search for duplicates using the modified merge described above. If the number of new tuples is small, they are added to designated pages. Otherwise, the new pages are added to the relation's page table at the end of the operation.

4.1.3 *Modify.* There are two cases to consider for the modify operation. In the case that the modified attribute or attributes does not contain the relation key (or part of it), we are assured that no duplicate tuples will result from the modify. In this case each processor executes the same code as the simple delete, applying the modification to matching tuples rather than deleting them. The same analogy holds for a complex, nonkey modify. Note that no page reorganization is required since the page is sorted on the relation key which is unaffected.

In cases where the query modifies some part of the key, the algorithm must check for duplicates. To do this we must have a list of the new key values and check the source relation for duplicates using this list before we apply the update. Our algorithm works in a similar manner to the algorithm for nonkey modifies with one exception. When a tuple to be modified is found, the processor deletes that tuple from the page and writes the modified tuple into a separate buffer. After all the pages of the relation have been scanned, each page containing modified tuples is sorted on the relation key. The new pages are then broadcast to all processors that contain source relation pages to check for duplicates. As in the other update operations, if duplicates are found the operation is aborted. Otherwise, the new pages are added to the source relation page table.

As the update algorithms are all quite similar, we provide a performance analysis of only one of them. We chose to analyze the simple key modify since it is one of the more complicated algorithms and since it has elements that appear in all the others. This analysis is presented in Appendix A. We conclude this section by observing that all the update algorithms (except the key modify) operate in linear time. That is, given p processors, each algorithm would be executed by the p processors in n/p "basic" time units. (Note that the basic time unit used in the algorithm for one operation may differ from that used by the algorithm for another operator.)

4.2 Parallel Sorting Algorithms

In this section we present two parallel sorting algorithms and analyze the performance of each. The algorithms, the "parallel binary merge" sort and the "block bitonic" sort, were only two of a number examined. Our analysis has shown that the performance of the second algorithm is generally better.

Unlike other analyses of parallel sorting algorithms [4, 17], we do not assume that the relation to be sorted initially resides in the processor's main memory, or that the algorithm may terminate when the sorted relation can be obtained by gathering, in a specific order, the blocks of data from these memories. We assume that the number of processors allocated to the sorting operation p will, in general, be much less than the number of pages in the relation n and that n is larger than the total memory of the processors and the size of the disk cache.[2] Therefore, we only consider *external* parallel sorting algorithms (i.e., algorithms where the relation is read in successive blocks and sorting is done in a number of phases, each of which terminates with its output in temporary buckets).

The relation to be sorted is stored as a set of pages, each of which is individually sorted with respect to a prespecified key (see Section 4.1). Generally the relation resides on one or more mass storage devices when the sort is initiated. However, portions of it may be in the disk cache at that time owing to the relation's use in another, concurrent, operation. Similarly, when the algorithm terminates, the relation is returned to the mass storage device. During intermediate phases of the algorithm, temporary relations are created, and pages of these relations are transferred to the processors under the controller's supervision.

Each processor merges two ordered sequences (termed runs) of i pages each into a run of $2i$ pages. Since we assume that the size of each processor's main memory is only 3 pages, this operation requires that the processor must execute an external merge. For this case, the controller must maintain control tables which enable it to transfer entire runs, 1 page at a time, to a processor in the order necessary for a 2-way merge of 2 runs. The controller supervises and

[2] To simplify the analyses of both algorithms, we have assumed that n and p are both powers of 2.

coordinates the reading and the writing of single pages by the processors. Thus, at any time, a processor merges 2 pages residing in its 2 input buffers into a single page output buffer. When one of the input buffers has been completely scanned, the processor reads into this same buffer the next page of the appropriate run. When the output buffer fills up, the processor requests from the controller a "new page" and transfers the contents of the output buffer to the cache. The new page is an appropriately numbered page of a temporary relation. This page will serve either as an input for the next phase of the sort or as a page of the result (sorted) relation. It follows from the above argument that a processor can merge sort 2 runs of i pages each in $i\,C_{\mathrm{p}}^{2}$ operations (using the notation defined in Section 3).

4.2.1 *Parallel Binary Merge Sort*

Description. In this section we describe a merge-sort algorithm which utilizes both parallelism during each phase and pipelining between the phases to enhance performance. In [5], a binary merge sort without pipelining of the phases was analyzed. The parallel binary sort algorithm presented below represents a significant improvement.

Execution of this algorithm is divided into three stages, as shown in Figure 2. We assume that there are at least twice as many pages as processors. The algorithm begins execution in a suboptimal stage in which sorting is done by successively merging pairs of longer and longer runs until the number of runs is equal to twice the number of processors. During the suboptimal stage the processors operate in parallel, but on separate data. First, each of the p processors reads 2 pages and merges them into a sorted run of 2 pages. This step is repeated until all single pages have been read. If the number of runs of 2 pages is greater than $2p$, each of the p processors proceeds to the second phase of the suboptimal stage in which it repeatedly merges 2 runs of 2 pages into sorted runs of 4 pages until all runs of 2 pages have been processed. This process continues with longer and longer runs until the number of runs equals $2p$.

When the number of runs equals $2p$, each processor will merge exactly 2 runs of length $n/2p$. We term this phase the optimal stage. At the beginning of the postoptimal stage the controller releases one processor and logically arranges the remainder as a binary tree (see Figure 2). During the postoptimal stage parallelism is employed in two ways. First, all processors at the same level of the tree (Figure 2) execute concurrently. Second, pipelining is used between levels. By pipelining data between levels of the tree, a parent is able to start its execution a single time unit after both of its children (i.e., as soon as its children have produced 1 output page). Therefore, the cost of the postoptimal stage will be a 2-page operation for each level of the three plus the cost of the root processor to merge 2 runs of length $n/2$.

Analysis. If $p = n/2$, there is no suboptimal stage and the processor at the top of the binary tree waits $\log(n/2)$ units of time before it starts merging 2 runs of size $n/2$. Therefore, the algorithm terminates in $\log(n/2) + (n/2)\,C_{\mathrm{p}}^{2}$ operations.

If $p < n/2$, then during each of the $\log(n/2p)$ phases of the suboptimal stage each processor executes a total of n/p page operations (i.e., $(n/2p)\,C_{\mathrm{p}}^{2}$ operations). In phase i the runs are one-half the size of the runs of phase $i + 1$, but each of the p processors performs twice as many merge operations in order to exhaust the

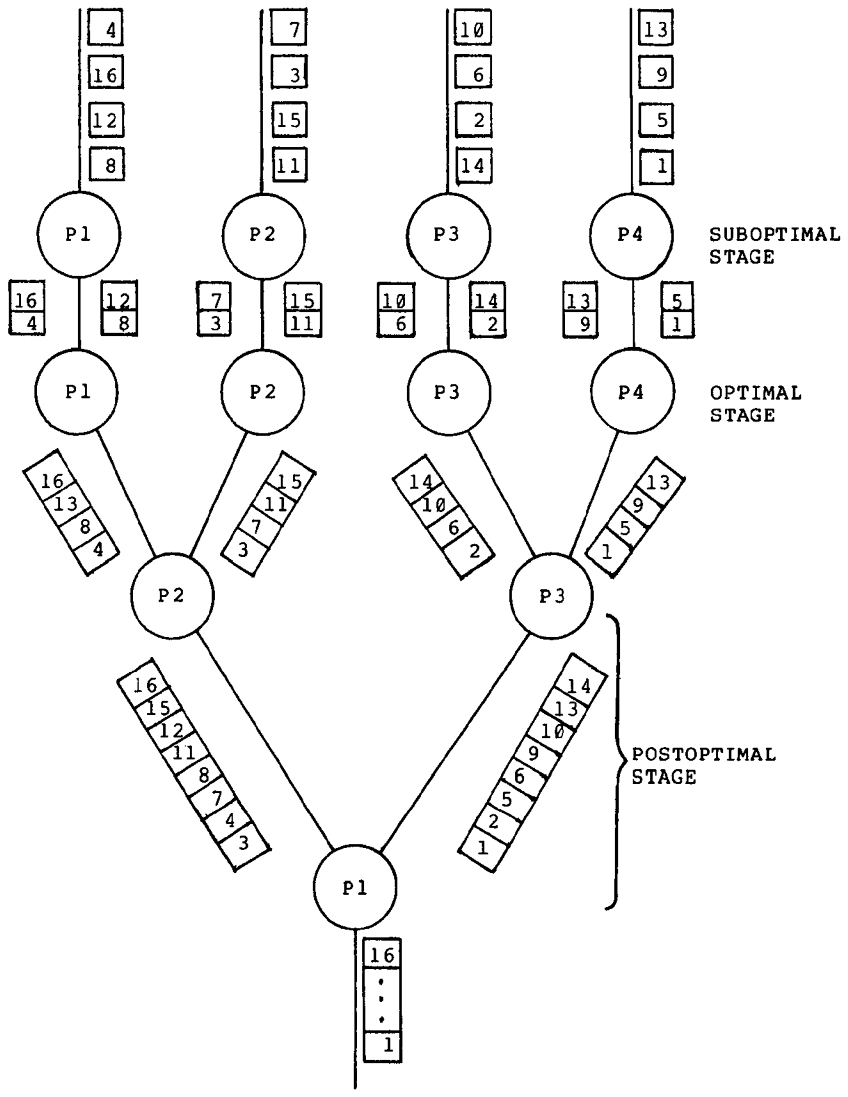

Fig. 2. Parallel binary merge with 4 processors and 16 pages.

runs. During the optimal stage, each of the p processors reads 2 runs of length $n/2p$. Therefore, there are $n/2p$ parallel 2-page operations. Finally, for the postoptimal phases, the number of 2-page operations is equal to $(\log p - 1) + n/2$ where $(\log p - 1)$ represents the time for the first page of both runs to reach the top processor. After this point the top processor must merge 2 runs of length $n/2$. Therefore, the total execution time of the algorithm expressed in C_p^2 units is

$$\underbrace{\frac{n}{2p}\log\left(\frac{n}{2p}\right)}_{\text{suboptimal}} + \underbrace{\frac{n}{2p}}_{\text{optimal}} + \underbrace{\log p - 1 + \frac{n}{2}}_{\text{postoptimal}}$$

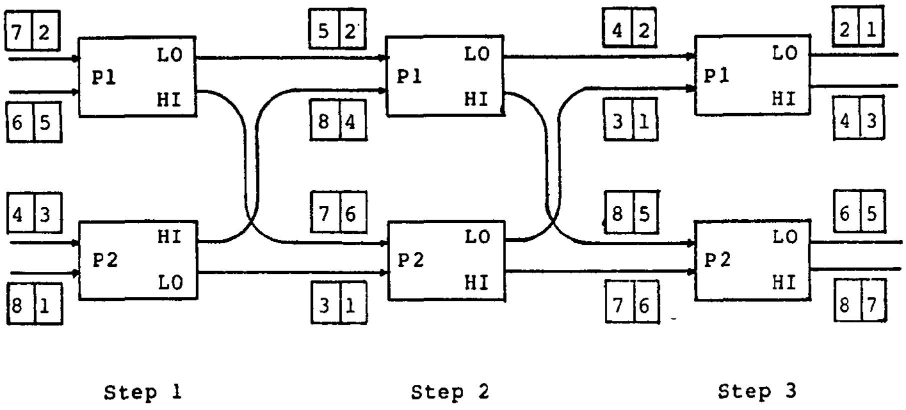

Fig. 3. Block bitonic sort with 2 processors and 4 runs of 2 pages each.

which can be expressed as

$$\frac{n \log n}{2p} + \frac{n}{2} - \left(\frac{n}{2p} - 1\right)(\log p) - 1.$$

4.2.2 *Block Bitonic Sort*

Description. Batcher's bitonic sort algorithm sorts n numbers with $n/2$ comparator modules in $\frac{1}{2} \log n(\log n + 1)$ steps [3]. Each step consists of a parallel comparison-exchange and a transfer. Execution of this algorithm requires that the comparison-exchange units be interconnected with a perfect shuffle interconnection scheme [15].

As first suggested in [4], if a comparator module is replaced with a processor which can merge 2 pages of data and then separately output the "lower" and the "higher" pages of the sorted 2-page block, then we have a block parallel algorithm which can sort n pages with $n/2$ processors in $\frac{1}{2} \log n(\log n + 1)$ 2-page operations. Execution of this algorithm using two processors is illustrated in Figure 3. We further generalize this idea to obtain an "external bitonic sort" scheme. The basic instruction performed by a processor is an external 2-way merge sort of 2 sorted streams each of size $n/2p$.

Because the block bitonic algorithm can process at most $2p$ blocks (runs) with p processors, a preprocessing stage is necessary when the number of pages to be sorted exceeds $2p$. The function of this preprocessing stage is to produce $2p$ sorted blocks of size $n/2p$ pages each. We have identified two ways of performing this preprocessing stage. The first is to use a parallel 2-way merge sort to create $2p$ sorted blocks (runs) of $n/2p$ pages each. The second is to execute a bitonic sort in several phases with blocks of size 1, $2p$, $(2p)^2, \ldots,$ until blocks of size $n/2p$ pages are produced. We have analyzed both approaches and have discovered that the first approach is approximately twice as fast as the second for large n and relatively small p. Therefore, we present below only an analysis of the first.

Analysis. The first part of the algorithm is identical to the suboptimal phase of the parallel binary merge and completes in $(n/2p)\log(n/2p)C_p^2$ time units. Then, an external bitonic sort algorithm is applied to the $2p$ blocks of size $n/2p$.

This step requires

$$\frac{n}{2p}\frac{\log 2p}{2}(\log 2p + 1)C_{\mathrm{p}}^2 \quad \text{operations.}$$

The total cost is thus

$$\frac{n}{2p}\left(\log n + \frac{\log^2 2p - \log 2p}{2}\right)C_{\mathrm{p}}^2.$$

4.2.3 *Performance Comparison of the* 2 *Sorting Algorithms.* Since both algorithms presented in this section execute essentially in $n\ \log(n/2p)C_{\mathrm{p}}^2$ time units, when $O(p) < O(\log n)$ each achieves the optimal speedup of p over a uniprocessor external merge sort. Indeed, when $O(p) < O(\log n)$, the other factors in the formulas established for the algorithms (Sections 4.2.1 and 4.2.2) are linear in n. In Figure 4 we have plotted the performance of both algorithms for a fixed number of processors and a varying number of pages to be sorted. As established by these graphs, the block bitonic sort outperforms the parallel binary merge (this fact can be proven analytically by comparing the formulas).

4.3 The Project Operation

The projection of a relation with domains $d1, d2, \ldots, dn$ on a subset of domains $di, dj, \ldots, dm$ requires the execution of two distinct operations. First the source relation must be reduced to a "vertical" subrelation by discarding all domains other than $di, dj, \ldots, dm$. Since discarding attributes may introduce duplicate tuples, the duplicates must be removed in order to produce a proper relation.

While the first operation can be performed very efficiently, the second is much more complex and requires nonlinear (with respect to the number of tuples) time on a single processor. One could argue that, if the result of the projection is going to only be used in a subsequent operation and not become a permanent relation in the database, it is unnecessary to perform the duplicate removal. However, if there are a large number of duplicates in the result relation (e.g., if the relation is projected on a nonkey attribute), the execution time of the complete query could be considerably slower (possibly orders of magnitude slower) without removal of the duplicates.

On a single processor, the complexity of eliminating duplicates is essentially the same as the complexity of sorting the relation. However, in a multiprocessor organization, we may either sort or make use of parallelism to eliminate duplicates without sorting. Since parallel sorting was considered in the previous section, in this section we present and analyze a method to eliminate duplicates which does not require sorting. The method relies on a hardware broadcast facility.

We assume that pages have already been reduced to a vertical form by the previous operation and that there are no intrapage duplicates. Each processor reads one page. Let a processor be labeled by the number of the page it has read (that is, the processor that has read page i is known as P_i). For $i = p, p - 1, p - 2, \ldots, 2$, page number i is broadcast to processors $P_{i-1}, P_{i-2}, \ldots, P_1$. When page i is broadcast, P_i is released. When processor P_j ($j < i$) receives page i, it compares page i and page j and eliminates any duplicates from *its* page. Note

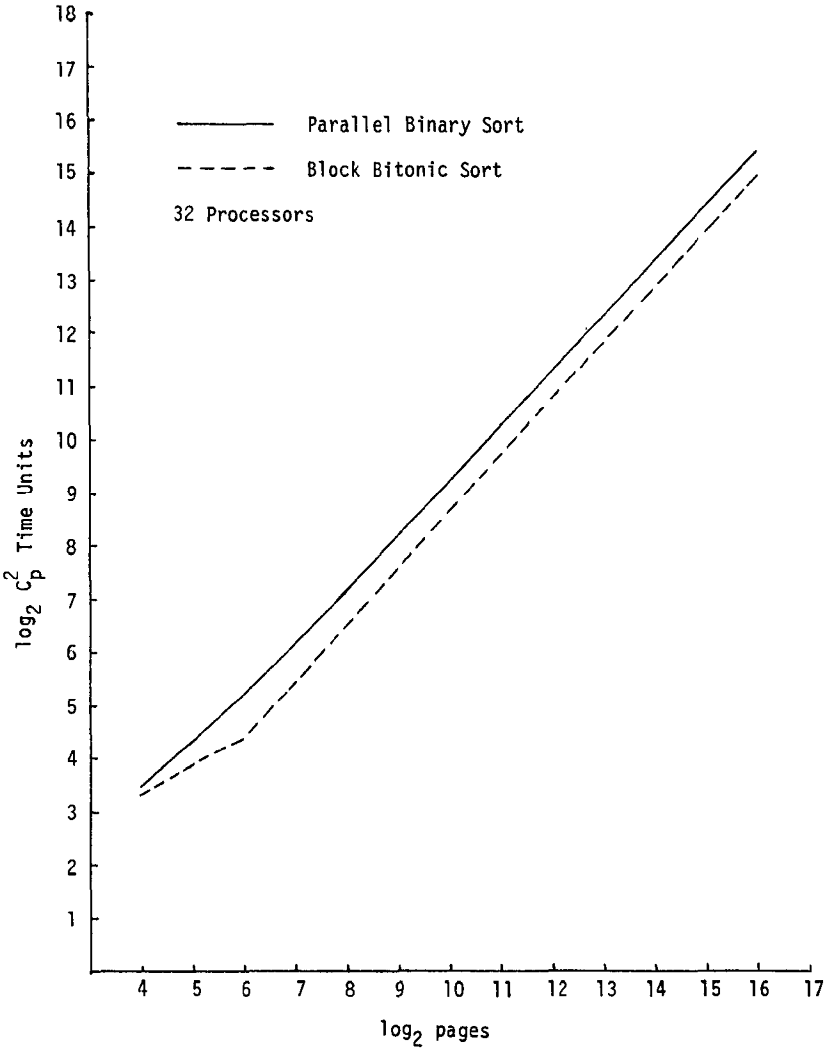

Fig. 4. Comparison of the two sorting algorithms.

that P_j will not see page i if $i < j$. Consequently it is guaranteed that only one copy of each tuple will remain in the relation (that copy will reside in the highest numbered page of all the pages that had a copy of it). The broadcast step is shown in Figure 5.

In the general case when p, the number of processors, is smaller than n, the number of pages, the algorithm works in a number of distinct phases. Each phase produces p projected pages and sees p fewer pages than the previous phase. In phase i there are $(i - 1)p$ pages that have already been projected, p pages in the processors' memories, and $n - (ip)$ nonprojected pages. The phase begins by broadcasting the $n - (ip)$ nonprojected pages to the p processors for duplicate removal. After this step has completed, P_p broadcasts its page and exits. The

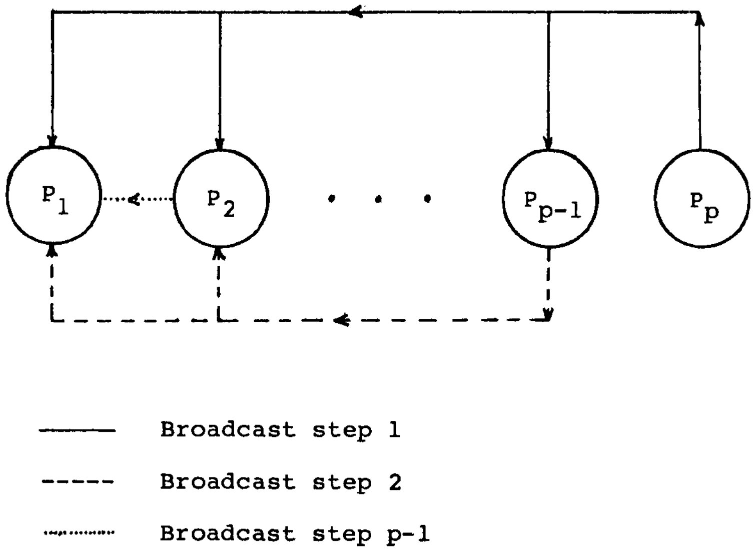

Fig. 5. Projection by broadcast.

remaining processors follow suit. The cost of phase i is thus

$$C_r + (n - ip)(C_r + C_m) + (p - 1)(C_r + C_m + C_w) + C_w.$$

If $n = pm$, there are m phases and the total cost of the algorithm is

$$mC_r + \frac{m(m-1)p}{2}(C_r + C_m) + m(p - 1)(C_r + C_m + C_w) + mC_w.$$

This may be rewritten as

$$\left(\frac{n^2}{2p} + \frac{n}{2}\right)(C_r + C_m) - \frac{n}{p}C_m + nC_w$$

which is of the order of $n^2/2p$ page operations. Note that if n is not an exact multiple of p, the last phase would use only $n \bmod p$ processors and thus terminate faster.

One may think of reducing the number of pages before starting the broadcast steps. For this modified version of the algorithm, each processor reads as many pages as it can, eliminating duplicates as it goes along. This modification may considerably improve the performance of the algorithm in the case of a high duplication factor. For example, if a tuple is duplicated 10 times on the average and the duplicates are uniformly distributed among the pages, up to 10 pages may be merged by each processor before the sequential broadcast algorithm is initiated. A second improvement to the algorithm would be to perform such a compression, at least once, of all the source relation pages before the broadcast step is initiated. Thus, when the number of duplicates is expected to be large, a broadcast method with a priori compression would perform much more efficiently than the analytical upper bound of $O(n^2/2p)$ page operations. On the other hand, if the number of duplicates is expected to be small, and if $n \gg p$, it is probably more efficient to use one of the sorting algorithms which perform in $O(n \log n/2p)$ page operations rather than $O(n^2/2p)$.

A comparison of the performance of these two algorithms is unfortunately beyond the scope of this paper. An accurate evaluation of these two algorithms requires the application of statistical tools since the distribution of the duplicates will have a significant effect both on the number of compression steps of the modified broadcast algorithm and the lengths of runs in the sorting algorithms. We are currently engaged in such an effort.

4.4 Join Algorithms

In this section we present two parallel algorithms for the relational join operation: a parallel "nested-loops" algorithm and a parallel "merge-sort" algorithm. The nested-loops join algorithm relies heavily on a broadcast facility, while the merge-sort algorithm requires sorting of the two source relations with respect to the join attribute. A third strategy based on hashing techniques has been recently investigated in [1, 10]. A performance comparison of the hashing strategy with either the nested-loops or the merge-sort joins is beyond the scope of this paper, but we plan to incorporate it in future work.

4.4.1 *The Parallel Nested-Loops Join Algorithm.* Given two relations R and T, the "smaller" relation (i.e., the one with fewer pages) is chosen as the inner relation, and the larger (say R) becomes the outer relation. The first step is for the processors to each read a different page of the outer relation. Next, all pages of the inner relation T are sequentially broadcast to the processors. As each page of T is received by a processor, it joins the page with its page from R. Clearly, this algorithm is a block parallel version of the most inefficient uniprocessor join algorithm since each tuple of relation R is compared with each tuple of relation T. However, since it achieves a high degree of parallelism for the duration of its execution (limited only by the number of pages in R), it may outperform more sophisticated join algorithms.

Let n and m be the sizes, in pages, of the relations R and R' and suppose $n \geq m$. Let p be the number of processors assigned to perform the join of R and R'. S is the join selectivity factor and indicates the average number of pages produced by the join of a single page of R with a single page of R'. If $p = n$, the execution time of this algorithm is

$$\begin{aligned} T_{\text{nested loops}} &= T(\text{read a page of } R) \\ &\quad + mT(\text{broadcast a page of } R') \\ &\quad + mT(\text{join 2 pages}). \end{aligned}$$

It is important to notice that joining 2 pages consists of the following operations. The 2 pages are joined by merging, then the output page is sorted on the join attribute of the subsequent join (if there is one), and finally the output page is written out. The number of output pages written depends on the join selectivity factor S defined by

$$S = \frac{\text{size}(R \text{ join } T)}{mn}.$$

If $p < n$, the same process must be repeated n/p times, yielding

$$T_{\text{nested loops}} = \frac{n}{p}\{C_r + m[C_r + C_m + S(C_{so} + C_w)]\}.$$

In the case that either the subsequent join is to use the same join attribute or the result of the join is to be displayed on a screen, the output pages need not be internally sorted.

4.4.2 *Merge-Sort.* This algorithm is performed by first doing a parallel sort on both relations to be joined (assuming that they are not both already sorted on the join attribute). After both relations have been sorted, they are joined, and the result relation pages are sorted on the attribute used by the subsequent operation. The merge operation is executed by a single processor. If the sort steps are performed using our block bitonic algorithm, the join cost is

$$T = \left[\frac{n}{2p}\log n + \frac{m}{2p}\log m + (\log^2 2p - \log 2p)\frac{m+m}{4p}\right]C_p^2$$
$$+ (n+m)C_r + \max(n,m)C_m + mnS(C_{so} + C_w).$$

One improvement to this algorithm is to leave the pages of the output relation unsorted. Then the p processors participating in the subsequent operation can perform the internal sort in parallel rather than the sequential sort done now.

It should be noted that by using a merge-sort algorithm to perform the join, we obtain a relation sorted with respect to the join attribute. This property is desirable if the output relation is the final result of a query or if it becomes the source relation for a subsequent operation using the same joining attribute.

4.4.3 *Comparison.* Using the formulas developed in the previous two sections, we have compared the performance of these two join algorithms. Our results are presented in Figures 6 and 7. Our assumptions about the processors' capabilities are specified in Appendix B.

Figure 6 presents the results for a selectivity factor of 0.001 with no sorting of output pages. We assumed that each page contained three hundred, 55-byte tuples, but we found very similar results for pages composed of one hundred, 165-byte tuples. The results indicate that when two relations of a similar size are joined, the merge-sort algorithm should be employed, unless the number of processors available is close to the larger relation size. However, if the ratio between the relation sizes is significantly different from 1, the nested-loops algorithm outperforms the merge-sort (except for a small numbers of processors). For lower selectivity factor values, the merge-sort algorithm performs better than the nested-loops algorithm because the merge step (handled by a single processor) has to output fewer pages. Since in the nested-loops algorithm the output relation is divided among all the processors, a reduction in its size has very little effect on the total execution time. Figure 7 shows a similar result for the same joins with a selectivity factor of 0.001 with sorting of output pages.

4.5. Aggregate Operations

In contrast to the relational operations join, project, select, etc., there is no commonly accepted set of aggregate operations among existing relational data-

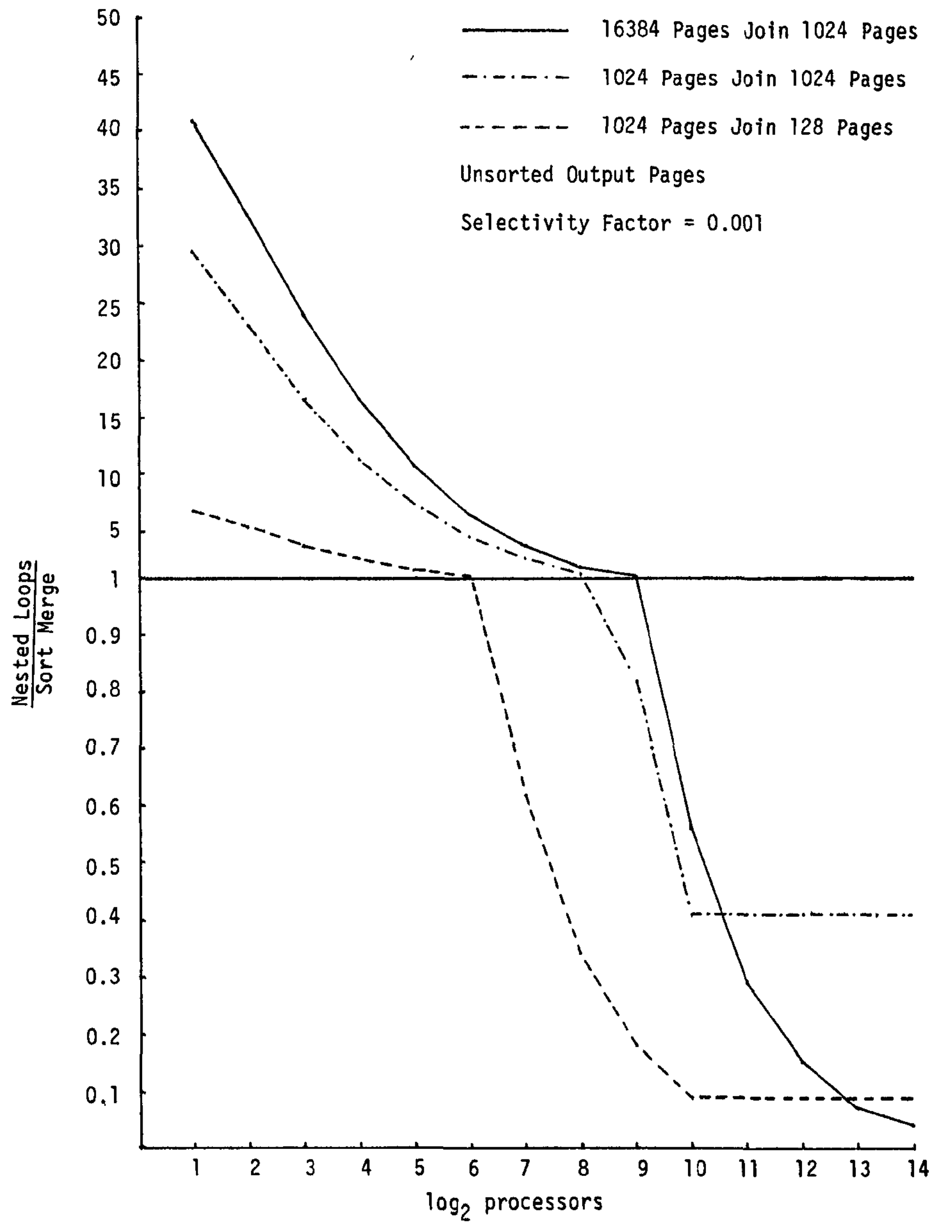

Fig. 6. Comparison of the two join algorithms.

base systems. For our purposes, we adopt the facilities provided by INGRES [18] as being representative and develop algorithms to process them (see [8] for a presentation of algorithms for processing aggregates in a uniprocessor environment). We distinguish "scalar" aggregates from aggregate "functions." Scalar aggregates are aggregations (average, max, etc.) over an entire relation. Aggregate functions first divide a relation into nonintersecting partitions (based on some attribute value, e.g., sex) and then compute scalar aggregates on the individual

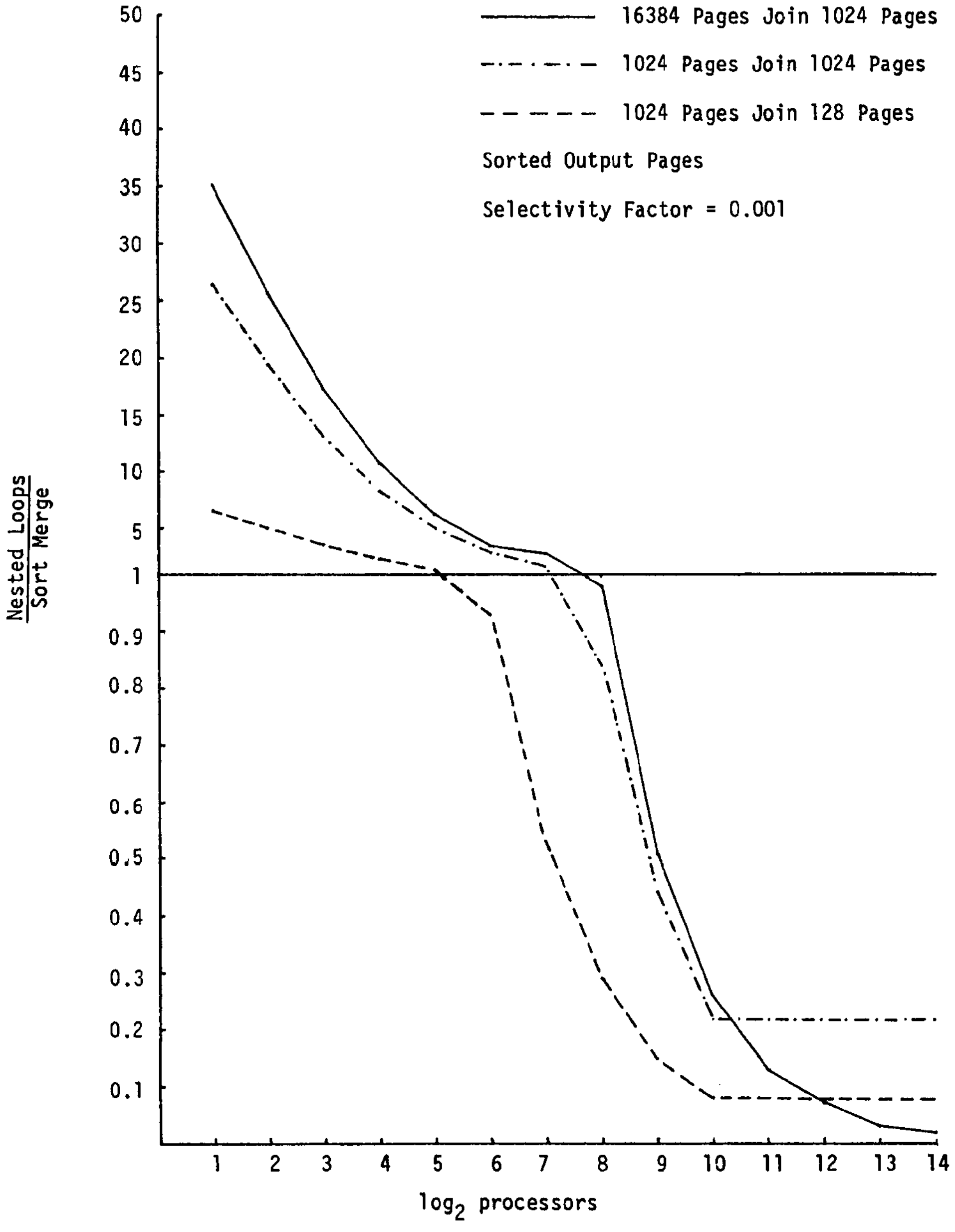

Fig. 7. Comparison of the two join algorithms.

partitions. Thus, given a source relation, scalar aggregates compute a single result, while aggregate functions produce a set of results (i.e., an output relation). The two types of aggregates have the following form:

scalar	agg_op (agg_att where qual)
function	agg_op (agg_att by_list where src_qual) where by_qual
by_list	by att-1 by att-2 by . . . by att-n
agg_op	sum, avg, count, max, min, sumu, avgu, countu

Employee Relation

Name	Department	Task	Salary	Manager
Smith	Toys	Clerk	300.00	Johnson
Miller	Shoes	Buyer	650.00	Bergman
Jones	Books	Acct	550.00	Harris
Brown	Shoes	Clerk	400.00	Conners

countu(Employee Department) = 3

Fig. 8. Example of a "unique" scalar aggregate.

The agg_att is the attribute over which the aggregate is being computed. The aggregate operators (agg_op above) are self-explanatory except for those with the "u" suffix. The u denotes "unique" and implies that duplicates (tuples which match on the agg_att) will be eliminated before the aggregate is computed (see Figure 8).

Qualifications may be added ("where qual") to compute an aggregate over a subset of tuples in a relation. For aggregate functions, the partitioning attributes are specified with the by_list. Note that relations may be partitioned on more than one attribute (e.g., partitioning employees by department and task within department). Also note that the result of an aggregate function may depend on qualifications outside the aggregate (by_qual) (this is discussed in more detail later). In contrast, scalar aggregates are "self-contained" and are not affected by the rest of the query. Finally, as with update operations, we distinguish "simple" qualifications from "complex" qualifications. Simple qualifications can be processed in a single scan of the relation and may be applied at the same time that the aggregate is being computed. Complex qualifications require interrelation operations so the relation must be preprocessed before computing the aggregate.

The compute a scalar aggregate, a processor maintains two fields: a count field, and the aggregate value itself. The count field specifies the number of tuples contributing to the aggregate value and is used in averaging and initialization. When processing aggregate functions, a third field is also required to identify the partition (since a processor may be accumulating aggregate values for more than one partition at the same time). The analyses of the algorithms described below are presented in Appendix C.

4.5.1 *Scalar Aggregates.* Scalar aggregates may be processed in a single pass over a relation. We consider only the obvious algorithm. The p processors request pages of the source relation from the controller and compute an aggregate value for the pages they see. When the pages are exhausted, we have p partial results, and a single processor must combine them to produce the final value. A simple qualification is applied at the same time the processors are accumulating their partial results. Complex qualifications require preprocessing of the source relation since interrelation operations are involved. If the aggregate operator is a "unique" operator, the source relation must be projected on the agg_att so that duplicate tuples are eliminated.

4.5.2 *Aggregate Functions.* In this section we describe two algorithms for processing aggregate functions. Recall that we must consider two types of qualifications. To see why, consider the following example:

count(emp.name by emp.mgr where emp.sal > 500)

This query requests, for each manager, a count of the number of employees earning more than $500. However, even if a manager does not have any employees making more than $500, he should not be excluded from the list and his count should be set to 0. If we applied the qualification first and then computed the aggregate function on the result, we would miss those managers since all his employees were removed by the qualification. As another example, consider

count(emp.name by emp.mgr where emp.sal > 500) where emp.mgr ≠ "Smith"

Clearly, in this case we want to include the count for all managers other than Smith. Thus, we need to distinguish restrictions on the source tuples from restrictions on the set of possible partitions. This is why we allow for two different types of qualifications in aggregate functions. Qualifications inside the aggregate (the "src_qual"), in addition to selecting a subset of the source relation, may have the undesirable side effect of removing desired partitions (e.g., manager Johnson in Figure 8). On the other hand, qualifications outside the aggregate (the "by_qual") are used to eliminate unwanted partitions. (e.g., manager Smith above).

When an aggregate function contains an src_qual, any algorithm for processing the aggregate must begin by determining the set of desired partitions so that any partitions which are removed by applying the src_qual (e.g., managers with zero counts, above) can be included in the result of the query. Determining the set of desired partitions occurs in one or two steps depending on whether the query contains a by_qual. If the query does contain a by_qual (whether simple or complex), it is applied to the source relation in order to eliminate "unwanted" partitions. Then, the resulting relation (or the source relation if the relation did not contain a by_qual) is projected on the by_list attributes to determine the "names" of the desired partitions.

Algorithm 1: Subqueries with a parallel merge. Our first algorithm is similar to the scalar aggregate algorithm and works best when the number of partitions is small. In the first stage, each processor reads its source relation pages, but instead of accumulating a single aggregate value, it produces one aggregate value for each partition it sees. This results in a number of pages containing partial results which must be combined. The second stage is a parallel "merge" of the pages produced in the first stage.

When qualifications are included in a query, several additional steps are needed to extract the correct partitions. First, the by_qual (if the query has one) must be applied to eliminate unwanted partitions. If the query has an src_qual, three additional steps must be performed. First, the set of desired partitions must be determined by projecting the source relation (or the relation produced by executing the by_qual) on the by_list. This step will produce a temporary result relation (denoted R') with the result and count values for each partition initialized to 0. Next, the src_qual must be applied. If the query has a simple src_qual, it may be processed at the same time the aggregate is computed; otherwise, it is performed as a separate operation before the aggregate is computed. The final stage required when an src_qual is specified is for one processor to merge R' with its run of t pages before the parallel merge is initiated.

Finally, note that unique aggregates require a separate preprocessing step in which the source relation is sorted on the by_list in order to eliminate duplicates.

Algorithm 2: Project by_list and broadcast source relation. This algorithm exploits the ability of our architecture to broadcast pages to multiple processors. The idea is to first project the source relation on the by_list domains to determine the partitions. This gives us a list of partitions which we will distribute among p processors. The pages of the source relation are then broadcast to all processors and each processor computes the aggregate value for its set of partitions. If the space occupied by the list of partitions exceeds the combined buffer space of the processors, then the source relation will have to be broadcast more than once. If the query has a simple src_qual, it may be processed at the same time the aggregate is computed. If a unique aggregate is specified, the source relation must be sorted on its by_list (as the major field) and the agg_att (as the minor field). Then, duplicates will be eliminated by the processors which will compare tuples with the previous tuple received for that partition.

Comparison. In order to compare the performance of these two algorithms, we selected two queries, one with an src_qual and one without an src_qual. In Figures 9 and 10, we have plotted the execution time for both algorithms for 32 processors, 1000 partitions, and varying relation sizes. (The assumptions made with regard to I/O costs and processor speeds are described in Appendix B; the curves are based on formulas derived in Appendix C.) Figure 9 shows that Algorithm 1 is significantly superior to Algorithm 2 (up to two orders of magnitude) when the query does not contain an src_qual. However, as shown in Figure 10, when the query contains an src_qual, Algorithm 2 is better except when the relation is very large. Furthermore, the performance of Algorithm 1 is sensitive to the number of partitions. Since both algorithms process by_qualifications in the same way, the results presented are representative whether or not the query contains a by_qual. Similar results were obtained with both different numbers of processors and processors of varying speeds.

In addition to the two parallel aggregate function algorithms which we have presented, we also developed and evaluated another algorithm which employed a parallel binary merge sort to divide the source relation into one subrelation for each partition. As each subrelation was produced by the sort, another processor immediately read the subrelation and computed its aggregate value. While this algorithm initially looked promising, our analysis showed that is was always inferior to the other two algorithms except when the relation was partitioned on a key (an unlikely event).

5. CONCLUSIONS AND FUTURE RESEARCH

This paper has presented and analyzed algorithms for parallel processing of relational database operations. We have concentrated on those operations which cannot be processed in a single pass over the relation. To analyze alternative algorithms, we have introduced an analysis methodology which incorporates I/O, CPU, and message costs.

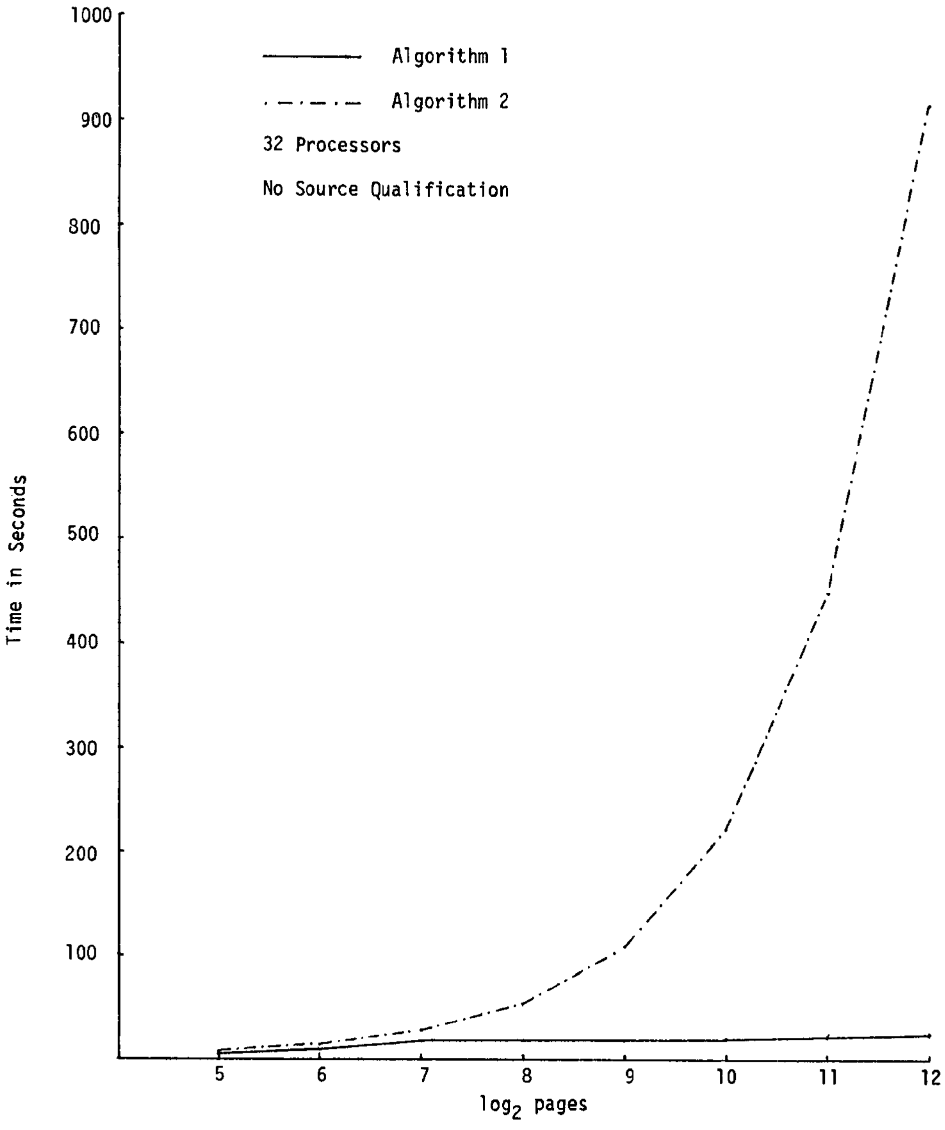

Fig. 9. Comparison of two aggregate algorithms.

Parallel sorting has been used as a basic building block in the design of algorithms for relational database operations. Although an extensive literature on parallel sorting exists, none of the algorithms appear feasible. We have discussed two algorithms for parallel sorting of large relations residing on mass storage devices. We did not rely on an asymptotic complexity analysis to derive the "optimal" algorithm because I/O time and communication overhead must be incorporated for a more accurate analysis. For example, despite the theoretical complexity of $O(n)$ and $O(\log^2 n)$, the parallel binary sort and parallel block bitonic sort show a comparable performance (see Figure 4).

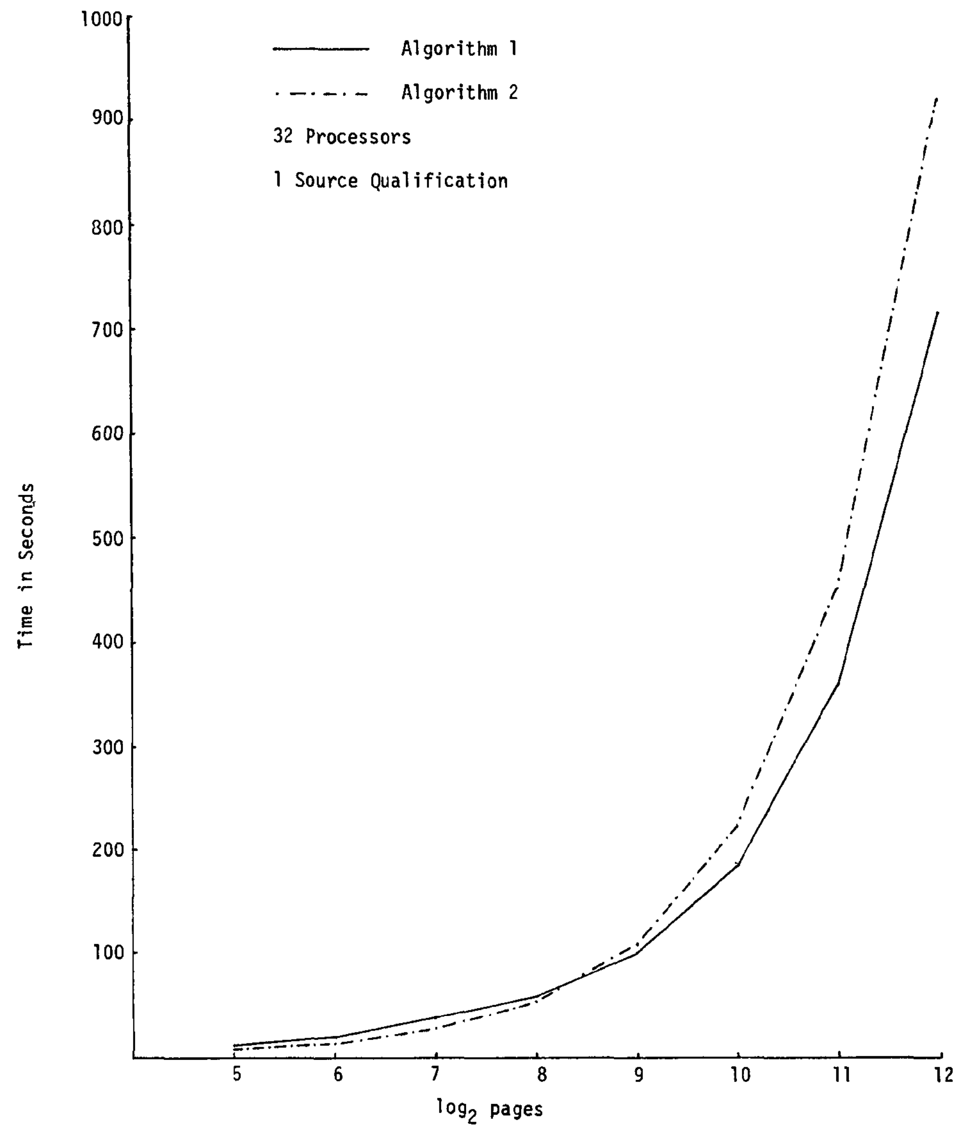

Fig. 10. Comparison of the two aggregate algorithms.

Our analysis of algorithms for parallel join operations indicates that when the sizes of the two relations to be joined are approximately the same, the parallel merge-sort algorithm is superior to the parallel nested-loops algorithm. However, when one relation is larger than the other (as is frequently the case when joining a relation describing an entity set with a relation describing a relationship), the parallel nested-loops algorithm is faster.

We have presented two algorithms for the project operation: one based on sorting and the other based on broadcasting. The results presented are inconclusive since the analyses do not incorporate the effect of duplicate tuples on the performance of the two algorithms. Our feeling is that elimination of duplicates through sorting is probably faster except when there is a high duplication factor. The extension of these analyses to handle the effect of duplicates is currently under investigation.

Our model of a shared memory multiprocessor architecture has enabled us to examine a wide range of algorithms without being constrained by the limitations of a specific interconnection scheme. On the other hand, it has forced us to make simplifying assumptions. In particular, we have not clearly differentiated between the I/O and interprocessor communication costs. Also, we have assumed that the processors were synchronized by their page requests to the controller. Despite these limitations, we believe this study to be a first important step in the design and analysis of parallel algorithms for database machines. A strong argument to be made in favor of this approach is that it has enabled us to proceed with the implementation of update, sorting, and aggregate operations on DIRECT.

This paper leaves open several other areas for future research. First the addition of "logic-per-track" devices to the multiprocessor organization would permit the development of additional algorithms for join, projection, and aggregate operations. A second area for future research is the design and analysis of parallel algorithms for database operations which employ indices. If algorithms can be developed that allow the efficient processing of indices in a multiprocessing environment, these algorithms could also be utilized to develop parallel algorithms for the selection operation whose performance can then be compared with the performance of "logic-per-head" devices. Another area that needs further exploration is the development of techniques for evaluating the cost of controlling multiple processors on complex algorithms (such as the parallel merge-sort join). While it may be the case that the control cost is dominated by the I/O cost (and hence the relative performance of the algorithms is unchanged), this topic merits further investigation. In addition, the performance of each of these algorithms in the context of a complete query should be analyzed since the choice of an algorithm for each operation may be affected by the other operations in the query. Finally, an interesting extension would be to consider a cost/performance evaluation of these algorithms in which architectural costs were incorporated.

APPENDIX A. ANALYSIS OF THE PARALLEL KEY MODIFY ALGORITHM

In this appendix we analyze the performance of the parallel key modify algorithm described in Section 4.1.3. The execution time of the simple key modify by p processors is given by the following formula:

$$T_p = \frac{n}{p}\,(\underset{\text{stage}_1}{[T_1^1]} + \underset{\text{stage}_2}{[T_1^2]})$$

where

$$T_1^1 = C_r + C_{sc} + C_o + C_w + \frac{j}{k}\,(C_{so} + C_w)$$

and

$$T_1^2 = C_r + z(C_r + C_m).$$

In stage$_1$ each processor examines n/p source relation pages, looking for tuples matching the qualification $(C_r + C_{sc})$. We assume that on the average j such tuples exist in each source relation page. Each page containing qualifying tuples needs to be reorganized (C_o) and written out (C_w) after the matching tuples have been moved to the buffer. Finally, the new tuples need to be sorted ($(j/k)C_{so}$) and written out ($(j/k)C_w$).

In stage$_2$ the processors search for the possible introduction of duplicates into the relation. Let z denote the number of pages containing modified source tuples. Then each processor reads a page of the source relation and all of the z pages. The processor performs the modified merge described above. Finally, if no duplicates are found, the z new pages are added to the source relation page table.

APPENDIX B. PROCESSOR CAPABILITIES ASSUMPTIONS

In this appendix we outline our assumptions about the capabilities of processors used in our evaluation of the join and aggregate algorithms (see Sections 4.4 and 4.5).

Page size is 16 kbytes.

C, the time to compare two attributes, is 10 μs.

V, the time to move a tuple, is based on the cost of 1.5 μs to move a single word. Thus, for a tuple length of 150 bytes, V is 225 μs.

R_m, the time to transfer a page between mass storage and the cache is 28 ms. This is based on a transfer time of 20 ms, a latency time of 8 ms, and a negligible track seek time.

R_c, the time to transfer a page from the cache to the processor's memory is 16 ms, based on a processor bus bandwidth of approximately 1 Mbyte per second.

The cache hit ratios H and H' were assigned the values 0.85 and 0.35, respectively.

The cost to process a message C_{msg}, including the sending, transfer time, and receiving, was picked to be 15 ms.

Finally, it should be noted that each experiment was subsequently performed with slower processor speeds (about half as fast), and that similar results were obtained.

APPENDIX C. ANALYSIS OF THE PARALLEL AGGREGATE ALGORITHMS

In this appendix we analyze the performance of the parallel aggregate algorithms described in Section 4.5. In the following discussion, we assume these parameters:

n	number of pages in source relation
p	number of processors to process aggregate
m	for aggregate functions, number of partitions
r	for aggregate functions, number of result tuples per page
q	number of operations to apply for a simple qualification (if query has one); else 0.

Scalar aggregates are as follows:

$$\begin{aligned} T\text{sc\_agg} = {} & T(\text{exec qual}) && \text{(if complex qual)} \\ & + T(\text{project}) && \text{(if unique agg\_op)} \\ & + T(\text{partial results}) \\ & + T(\text{combine } p \text{ partials}) \end{aligned}$$

We are concerned with the time needed to produce and combine the partial results since the time required to execute the qualification and to project the source relation have been covered by other sections of this paper.

$$T(\text{partial results}) = \frac{n}{p}[C_r + (q + 1)C_{sc}] + C_{msg}.$$

Each processor sees n/p pages. To process the page it must read it, apply a qualification to it (if simple), and update the partial result. Thus, each tuple requires a number of comparisons for the qualification plus an additional operation (e.g., add) to process the aggregate. The time to send the partial result is just the cost of a message. The processor which combines the partial results simply reads p messages and performs p arithmetic operations (note, the cost of the message is accounted for by the partial results formula). Thus, T(combine partials) $= pC$. The final formula is thus

$$\begin{aligned} T\text{sc\_agg} = {} & T(\text{exec qual}) && \text{(if qualification)} \\ & + T(\text{project}) && \text{(if unique aggregate)} \\ & + (n/p)[C_r + (q + 1)C_{sc}] + C_{msg} \\ & + pC \end{aligned}$$

Aggregate Functions: Algorithm 1

The cost of this algorithm (assuming no qualifications and a nonunique aggregate) may be computed as

$$T\text{alg1} = T(\text{produce partial result pages}) + T(\text{parallel merge}).$$

Each processor will read n/p source relation pages. Each tuple in the page must be placed in the correct partition and the aggregate value for that partition must be updated. If we assume $x = \min(m, r)$ partitions and use a binary search, then for each of the k tuples in a source page, $\log x$ comparisons are required to locate the correct partition. After the correct partition is located, the aggregate value must be updated. Thus, the cost to process the source relation pages is

$$\frac{n}{p}\{C_r + k[(\log x) + 1]C\}.$$

We need to estimate the number of output pages produced by one processor. An upper bound of $[(n/p)k]/r$ pages occurs when the relation is partitioned on a key. This is a pathological case. A lower bound is $\lceil (m/p)/r \rceil$ which occurs when the tuples from each partition are seen by only one processor (an equally unlikely event). We feel that $t = \lceil m/r \rceil$ is a plausible estimate of the number of output pages produced by each processor if one assumes that the partitions are uniformly distributed in the relation so that each processor sees all the partitions. Therefore, the cost for a processor to output its partial result pages is tC_w.

In addition to accounting for the cost of writing each of the t pages, we must also account for the cost of putting each page in sorted order (so that a binary search can be utilized). Each time a new by_list value is encountered (i.e., a new partition), the processor must create a new result tuple and add it to the sorted page. For each new partition this step requires, on the average, that one-half the result tuples be moved down. For $x = \min(m, r)$ partitions, $x(x + 1)/4$ tuple moves will be required. Thus, the cost to process each of the t pages produced by a processor is

$$t\left[\frac{Vx}{4}(x+1)+C_w\right].$$

The parallel "merge" we use in the second stage is not a true merge since 2 partial output pages are combined to form a single output page. First, each processor must form a sorted run of the t pages it has produced. Using a merge sort this step requires $(t/2)\log(t/2)\,C_p^2$ operations. Next a parallel binary merge (see Section 4.2.1) is used to combine the p runs of t pages into one run of t pages. The number of stages used is $\log p$. Each processor will read two runs of t pages, merge them, and write a run of length t. Let $C_{m'} = 2r(C + V)$ denote the cost of a merge of 2 output pages. Then the cost of the parallel merge is

$$(t+\log p)(2C_r + C_{m'} + C_w).$$

The total cost of the basic algorithm is then

$$Talg1 = \frac{n}{p}\{C_r + k[(\log x)+1]C\} + t\left[\frac{x(x+1)V}{4} + C_w\right]$$
$$+\frac{t}{2}\log\left(\frac{t}{2}\right)C_p^2 + (t+\log p)(2C_r + C_{m'} + C_w).$$

The final formula for Algorithm 1 is thus:

$Talg1 = T$(execute by_qual)	if by_qual
$+\ T$(project on by_list)	if src_qual
$+\ T$(execute src_qual)	if complex src_qual
$+\ T$(project)	if unique aggregate

process partitions:

$+\ (n/p)(C_r + k((\log x)+1)C)$	$x = \min(r, m)$
$+\ (n/p)(qC_{sc})$	if simple src_qual
$+\ t(x(x+1)(V/4)+C_w)$	$t = \lceil m/r \rceil$

perform parallel merge:

$+\ (t/2)\log(t/2)C_p^2$	
$+\ t\,(2C_r + C_{m'} + C_w)$	if src_qual
$+\ (t+\log p)(2C_r + C_{m'} + C_w)$	

Aggregate Functions: Algorithm 2:

The cost of this algorithm (assuming no qualifications and nonunique aggregates) may be summarized as

$$Talg2 = T(\text{project by\_list}) + T(\text{process partitions}).$$

A processor sees every page of the source relation (n pages). Each tuple must be placed in the correct partition (depending on the number of passes over the source relation, there are either m/p or r possible partitions), and we assume that the partitions are sorted so a binary search may be used. When the broadcast is complete, the processor must write its result. Let $b = \lceil (m/r)/p \rceil$ denote the number of complete broadcasts of the source relation. The cost to process partitions is

$$T(\text{process partitions}) = b\{n[C_r + (\log x)C_{sc}] + C_w\}$$
$$\text{where } x = \min(r, m/p).$$

Thus, the total cost for this algorithm is

$$\begin{array}{lll} Talg2 = & T(\text{exec by\_qual}) & \text{if by\_qual} \\ & + T(\text{project by\_list}) & \text{determine partitions} \\ & + T(\text{exec src\_qual}) & \text{if complex src\_qual} \\ & + T(\text{project source}) + bn(C_{sc}) & \text{if unique aggregate} \end{array}$$

process partitions:

$$\begin{array}{ll} + b(n(C_r + (\log x)C_{sc}) + C_w) & \\ + bnqC_{sc} & \text{if simple src\_qual} \end{array}$$

REFERENCES

1. Babb, E. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst. 4,* 1 (Mar. 1979), 1-29.
2. Banerjee, J., Baum, R.I., and Hsiao, D.K. Concepts and capabilities of a database computer. *ACM Trans. Database Syst. 3,* 4 (Dec. 1978), 347-384.
3. Batcher, K.E. Sorting networks and their applications. In *Proc. AFIPS 1968 Spring Jt. Computer Conf., vol. 32,* AFIPS Press, Arlington, Va.
4. Baudet, G., and Stevenson, D. Optimal sorting algorithms for parallel computers. *IEEE Trans. Comput. C-27,* 1 (Jan 1978).
5. Boral, H., and DeWitt, D.J. Design considerations for data-flow database machines. In *Proc. ACM SIGMOD 1980 Int. Conf. Management of Data,* (Santa Monica, Calif., May 14-16), ACM, New York, pp. 94-104.
6. DeWitt, D.J. DIRECT—A multiprocessor organization for supporting relational database management systems. *IEEE Trans. Comput. C-28,* 6 (June 1979).
7. DeWitt, D.J. and Hawthorn, P. A performance evaluation of database machine architectures. In *Proc. 7th Conf. Very Large Data Bases* (Sept. 1981).
8. Epstein, R. Techniques for processing of aggregates in relational database systems. Memo. UCB/ERL M79/8, Electronics Research Lab., College of Engineering, Univ. California, Berkeley, Feb. 1979.
9. Friedland, D. Design, analysis, and implementation of parallel external sorting algorithms. Ph.D. dissertation, Dept. Computer Sciences, Univ. Wisconsin, Madison, 1982.
10. Goodman, J.R. Personal communication.
11. Knuth, D.E. *The Art of Computer Programming—Sorting and Searching.* Addison-Wesley, Reading, Mass., 1975, p. 160.
12. Lin, C.S., Smith, D.C.P., and Smith, J.M. The design of a rotating associative memory for relational database applications. *ACM Trans. Database Syst. 1,* 1 (Mar. 1976), 53-65.
13. Ozkarahan, E.A., Schuster, S.A., and Smith K.C. RAP—An associative processor for database management. In *Proc. AFIPS 1975 Nat. Computer Conf., vol. 45,* AFIPS Press, Arlington, Va.
14. Slotnick, D.L. Logic per track device. In *Advances in Computers,* J. Tou (Ed.), vol. 10, Academic Press, New York, 1970.
15. Stone, H.S. Parallel processing with the perfect shuffle. *IEEE Trans. Comput., C-20,* 2 (Feb. 1971).

16. SU, S.Y.W., AND LIPOVSKI, G.J. CASSM: A cellular system for very large databases. In *Proc. Int. Conf. Very Large Data Bases*, (September 22-24, Framingham, Mass.) ACM, New York, 1975, pp. 456-472.
17. THOMPSON, C.D., AND KUNG, H.T. Sorting on a mesh-connected parallel computer. *Commun. ACM 20*, 4 (Apr. 1977), 263-271.
18. YOUSSEFI, K., ET AL. INGRES version 6.0 reference manual.

Received September 1980; revised July 1981; accepted July 1982