PARALLEL ALGORITHMS FOR THE EXECUTION OF RELATIONAL DATABASE OPERATIONS

by

Haran Boral
David J. DeWitt
Dina Friedland
W. Kevin Wilkinson

Computer Sciences Technical Report #402

October 1980

Parallel Algorithms for the Execution of Relational Database Operations

Haran Boral
David J. DeWitt
Dina Friedland
W. Kevin Wilkinson

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin

This research was partially supported by the National Science Foundation under grant MCS78-Ø1721 and the United States Army under contracts #DAAG29-79-C-Ø165 and #DAAG29-75-C-ØØ24.

ABSTRACT

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 which are presented and analyzed have been suggested 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 (for the first time) and analyze algorithms for the parallel execution of update and aggregate operations.

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 afterwards 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) [Slot70] from which RAP [Ozka75], RARES [Lin76], CASSM [Su75], and to some extent, DBC [Bane78] 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 [Hawt80]). In general, this is done by combining the processing capabilities of the host with those of the back-end database machine. The designers of RAP recognized the limitations of the pure associative disk design and added interconnections between the processing elements to facilitate processing of certain inter-relation 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.

In this paper, we attempt to back up from the design of any particular database machine (including DIRECT [DeWi79b]) and take a fresh look, from a more general perspective, at algorithms for relational algebra operations which can be executed by multiple processors in parallel. For each operation (e.g. project) we will present several algorithms and analyze the performance of each in terms of general parameters. Since the different algorithms may have different architectural requirements for an efficient execution, we hope that our results can be used to guide the design of future database machines. We will not, however, attempt to present such an ultimate design in this paper. Indeed, the best algorithm for the project operation may require a different architecture than the best algorithm for performing update operations. The designer of future database machines may very well have to weigh the importance of each of the different operators before choosing an appropriate architecture.

A further intent of this paper is to introduce an analysis methodology for parallel algorithms. We feel that despite their theoretical importance, past complexity analyses of parallel algorithms [Mull75], [Prep78], [Hirs78] have been unrealistic as they concentrated on a particular aspect of execution (e.g. number of record comparisons performed in a sorting algorithm). In this paper, we attempt to provide a realistic analysis of the algorithms for database operations which takes into account I/O, CPU, and communications costs.

As the reader will notice, the results presented in this paper concentrate on efficient parallel algorithms for "complex" database operations including projection, updates (appends, modifies, and deletions), sorting, joins, and aggregate operations (both scalar aggregate and aggregate functions). Our results are incomplete for a number of reasons. First, no parallel algorithms which employ indices to enhance performance are presented. Up to this point, we have avoided relying on indices for improving the performance of database operations for two reasons. The first is the overhead of maintaining these indices, which can be non-trivial in a database machine environment. (This is true even in DBC where special hardware is used [Hsia78]). The second reason is that, even when parallel algorithms which utilize indices are developed, occasionally a user will perform an operation for which none of the available indices are useful (it is certainly unrealistic to maintain a secondary index for each attribute of each relation). We have thus concentrated on using other techniques to develop efficient algorithms which can be used regardless of the query.

A second area not explored by this paper is that of efficient parallel algorithms for the selection operator. Recent research [Hawt8Ø] has indicated that the best (in terms of performance) way of doing selections is to have some sort of processor-per-track ([Slot7Ø], [Ozka75], [Su75]) or processor-per-head ([Bane78], [Leil78], [Banc8Ø]) device. However, [Hawt8Ø] also demonstrates that a conventional database system which uses indices to support efficient processing of selections sometimes performs as well as a selection-oriented database machine. In our opinion it is still an open question whether a parallel algorithm employing a combination of indices and general purpose processors or logic-per-head devices is the fastest and most cost-effective solution for processing selection operations. In fact, just the problems of maintenance and use of indices in a parallel processor environment is an important area of research.

It is impossible to completely divorce the execution of a parallel algorithm for a multiprocessor from any architectural assumptions. Therefore, in Section 2, we describe the properties of the general multiprocessor organization on which our algorithms are based. We have made our architectural assumptions as general as possible in an attempt to avoid "architecture directed" algorithms. In Section 3, we introduce the analysis techniques and assumptions which we will use to evaluate the different parallel algorithms. Section 4 presents and evaluates parallel algorithms for updates, sorting, projection, join, and aggregate operations. Our conclusions and areas for future research are discussed in Section 5.

2. A GENERAL MULTIPROCESSOR ORGANIZATION

The multiprocessor organization on which our parallel algorithms are based consists of the following components:

1. A set of general purpose processors
2. A multi-level memory hierarchy
3. An interconnection device connecting the processors with the multi-level memory hierarchy.

The processors are responsible for executing user queries and operate independently. Therefore, the processors form an MIMD (multiple instruction stream, multiple data stream) 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. (An alternative organization would be for each processor to have its own interface to the host processor. In this case, the host acts as the controller as well as dealing with users). 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 we assume 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 blocks (or pages) of data. At the bottom level of the memory hierarchy are the mass storage devices used to hold the relations in the database. The middle level of the hierarchy is a disk cache which is addressable by pages. A page of a relation is the unit of transfer between all levels of the memory hierarchy. The page size is chosen so that a page will constitute both a convenient processing unit and an efficient unit of data transfer. The larger a page the more efficient communications will be (since larger message size implies less overhead for control information). On the other hand, the page size must remain small enough so that enough processors, each examining one page, can participate in the operation (i.e. increased parallelism). In addition, the page size must be small enough so that a processor can internally merge sort 2 pages.

The bottom two levels of the memory hierarchy are connected together in a way that allows for data transfers 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.

Several proposed interconnection schemes seem to satisfy these requirements. The first is the banyan [Goke73] or folded banyan [Upch8Ø] switch. While not originally designed to provide broadcast capabilities, it has recently been shown that this can be done [Lipo8Ø]. A second suitable interconnection scheme which can provide these properties is a cross-point switch, connecting every page of the cache to every processor. This switch has been traditionally avoided because of its $O(n^2)$ complexity. However, recent research [Fran8Ø] has indicated that for VLSI implementations the cross-point switch may be superior to the banyan switch due to its regularity. [DeWi79b] discusses a method that achieves broadcast capability using a cross-point switch.

A third alternative is a very high speed ring connecting together each processor, each page frame of the cache, and each mass storage device. To be feasible the bandwidth of the ring must at least equal the total bandwidth of all processors and mass storage devices. While probably not feasible with today's technology, rings constructed with light-pipe technology may provide the necessary bandwidth.

There are several remaining points which should be mentioned. First, the disk cache described above is an integral and important part of the architecture. The need for such a cache has been demonstrated by [Hawt80] where it was shown that caching systems outperform logic-per-track and logic-per-head systems for "complex" non-linear time operations such as sorting and joins. It should be noted, however, that the effect of a cache can be achieved by cacheless systems with an adequate processor-to-processor interconnection (e.g. XTREE [Desp78]). In this sort of an organization both requirements (parallel transfers of data and a broadcast facility) can be achieved (broadcasting might have to be emulated using a store and forward approach). Therefore, our algorithms appear also to be applicable for these architectures. In fact, since the disk cache can emulate arbitrary interconnection schemes such as the binary tree interconnection, a linear interconnection, or a perfect shuffle interconnection, we feel that our results can be generalized to any such architecture. It should be noted, though, that in the analysis of the algorithms for architectures with direct processor-to-processor interconnections, the parameters reflecting the cost of writing and reading a page should be replaced by the cost of transferring a page between two processors.

A final, 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 processors' local memory or the disk cache before processing begins. Consequently 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 sorting schemes employed must be external sorting schemes.

3. ANALYSIS CRITERIA AND METHODS

After several preliminary attempts to analyze the performance of our algorithms, we felt the need for a rigorous definition of some basic performance parameters. These parameters must measure the I/O cost, the processing cost, and the communication cost for executing an algorithm on a given multiprocessor architecture. We have identified a number of basic tasks common to all our algorithms (e.g. reading a page) and have associated a cost with each. The execution time expression for each algorithm will be expressed in terms of the costs of these basic steps. For different architectures, the parameters may have different values and may relate differently to each other; for example, the I/O cost may be more significant than the processing cost for some architectures, but not for others. Our first basic assumption is that data is moved and processed by page units. We assume that 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. We have chosen to represent fixed costs by capital letters. Other parameters (for example, the number of pages to be read) are represented by lower case letters. The basic tasks used in evaluating the performance of our algorithms are:

(i) I/O cost: A read request moves a page into a processor's memory from either the cache or one of the mass storage units. A write request always moves a page residing in a processor's local memory to the cache. 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. To achieve a good hit ratio, the cache manager should use an appropriate replacement algorithm and a prefetching strategy. It should be noted that one cannot expect a hit ratio as high as for a main memory cache (.95 for many known implementations) since this is achieved on the basis of the program locality principle. However, 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. Given the values for R_c, R_m, and H we can calculate C_r, the average cost of a read by a processor:

$$C_r = H * R_c + (1-H) * (R_c + R_m)$$

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:

$$C_w = H' * R_c + (1-H') * (R_c + R_m)$$

(ii). 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:

$$C_{sc} = k * C$$

(iii). 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 [Knut75]. The number of tuples to be moved is the same. Thus, C_m, the cost of merging two pages is computed as:

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

(iv). 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:

$$C_o = (k * (C + V)) / 2$$

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. 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":

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

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

(v). Communication cost: Since transfers of pages are considered as I/O operations, the cost of communication includes only the page request and reply messages plus the control messages between the controlling processor and the other processors. When a processor wants to read or to 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 shall include the cost of the request and reply messages in our definition of page read and write operations, i.e we shall replace C_r by C_r+C(request message)+C(reply message). Therefore, the remaining communication cost of an algorithm can be measured by the number of control messages sent required by an algorithm. 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 the number of control messages is small compared to the number of I/O 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.

Another important evaluation measure of a parallel algorithm is its efficiency. By efficiency we mean a measure of the effective processor utilization. For example, an algorithm may request p processors at initiation time but some of these processors may remain idle until the algorithm has reached a certain stage. An algorithm may also require that a processor that has been active for some time become idle for a short period. While total execution time is one measure of the cost of a parallel algorithm, we are also concerned with the total amount of processing resources consumed by the algorithm. The efficiency of an algorithm is then defined as the ratio of the time the processors are busy over the time the processors are reserved (i.e. busy plus idle time). A high degree of efficiency is especially significant for parallel algorithms for MIMD organizations since idle processors can be used either by operations which do not have their optimal allocation of processors or to initiate new operations.

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. 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. We shall also present an analysis of one algorithm's complexity.

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, suppose we wanted to delete all employees whose employee number is less than 153 and the department in which they work is not the toy department. The query would be expressed as:

```
Delete emp where emp.eno < 153 and
   emp.dno = dept.dno and dept.name != "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 inter-relation operation(s), (e.g. join) in order to determine the tuples to be updated. The algorithms presented below handle both simple and complex updates.

For consistency reasons, 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, no pre-processing is required. Each processor executing the deletion will request pages of the source relation from the controller for examination. Tuples satisfying the deletion criterion are removed from the page and the page is compressed and flushed out to the buffer memory. 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 pre-processing step to determine the set of tuples to be removed. The set produced is a list of database keys (henceforth referred to as Q) which must be distributed to the processors which perform the deletion. One possibility is to include Q with the compiled code for the deletion. The controller could attach Q to the code segment as a data structure. This approach would be feasible if the size of Q is small (a page or less). If Q is large its pages can be broadcast to all the processors that have pages of the source relation. Each processor would 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-operation 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(s) 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 would execute the same code as the simple delete, applying the modification to matching tuples rather than deleting them. The same analogy holds for a complex, non-key modify. Note that no page reorganization is required since the page is sorted on the relation key which does not include the modified attribute(s).

In the case that 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 non-key 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 shall 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 it has elements that appear in all the others. The execution time of the simple key modify by p processors is given by the following formula:

$$T_p = (n/p) * (\underset{stage_1}{[T_1^1]} + \underset{stage_2}{[T_1^2]})$$

where:

$$T_1^1 = C_r + C_{sc} + C_o + C_w + (j/k) * (C_{so} + C_w)$$

and

$$T_1^2 = C_r + l' * (C_r + C_m)$$

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 l' denote the number of pages containing modified source tuples. Then each processor reads a page of the source relation and all of the l' pages. The processor performs the modified merge described above. Finally, if no duplicates are found, the l' new pages are added to the source relation page table.

We conclude this section by observing that all the update algorithms 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 three parallel sorting algorithms and analyze the performance of each. The algorithms, the "pipelined merge" sort, the "parallel binary merge" sort, and the "block bitonic" sort, were only three of a number examined. Our analysis has shown that the performance of the last algorithm is generally best. Although we will demonstrate that the performance of the first algorithm is significantly inferior to the other two, it is included because it has properties which make it useful in the context of the execution of an entire query.

Unlike other analyses of parallel sorting algorithms

[Baud78], [Thom77], we do not assume that the relation to be sorted initially resides in the processors' main memory, nor 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[1]. Therefore, we only consider external 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 terminate with their 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 due 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.

For some of the sorting algorithms, sorted "runs" of several pages are formed and a processor has to merge 2 runs of i pages each and output a sorted run of 2i pages. Since we assume that

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

the size of each processor's main memory is only three pages, this operation requires that for runs larger than one page ($i>1$) that the processor must execute an external merge. For this case, the controller must maintain control tables which enable it to transfer entire runs, one page at a time, to a processor in the order necessary for a 2-way merge of 2 runs. The controller supervises and coordinates the reading and the writing of single pages by the processors. Thus, at any time, a processor merges two pages residing in its two 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 relation 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 $i*C_p^2$ operations (using the notation defined in Section 3).

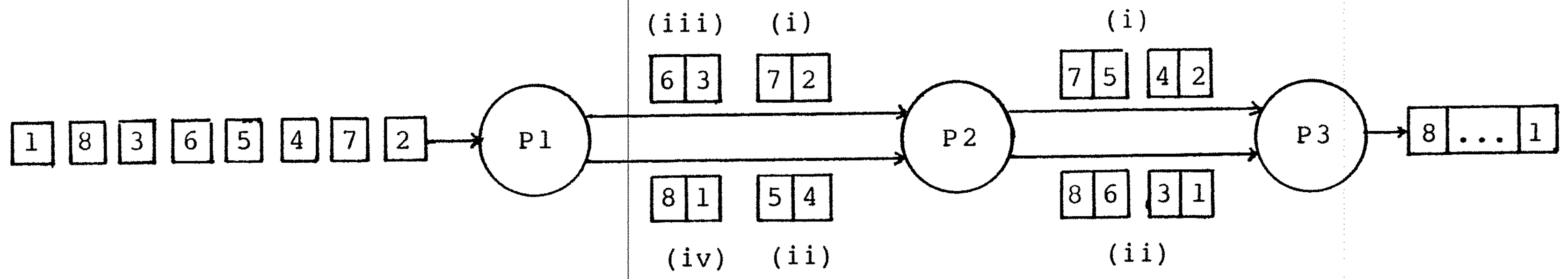
4.2.1. Pipelined Merge Sort

Description:

The processors assigned to the sort operation are labeled P_1, P_2, ..., P_p, and are logically organized as a linear pipeline. Each processor performs a 2-way merge operation of pairs of sorted runs produced by its predecessor in the pipeline as shown in Figure 1. During the first pass through the pipeline, processor P_{i+1} merges pairs of runs of size 2^i pages produced by P_i into runs of size 2^{i+1}. Therefore, if $2^p=n$ the relation can be sorted in one pass through the pipeline. Otherwise, (i.e. if $p<\log n$), additional phases are needed: each phase (except perhaps the last) requires a pass through the entire pipeline and increases the size of the sorted runs by a factor of 2^p. In order to achieve a maximum degree of overlap between stages of the pipeline, processor P_{i+1} can begin execution as soon as processor P_i has written a first run and is ready to write the first page of a second run. At this point, the writing of pages from the second run by processor P_i can be overlapped with their reading by processor P_{i+1} since the disk cache is used by all processors. This ability to overlap writing and reading of the same run is what distinguishes our algorithm from Even's parallel tape sorting algorithm [Even74]. Note that each processor, at any stage in the pipeline, does the same amount of work: $n/2\ C_p^2$ operations. What changes from one processor to another (and from one phase to another) is the time at which a processor can begin execution. This algorithm also has the property that it can accommodate a dynamic allocation of processors by the controller: at the start of a new phase, additional processors can be assigned to speed up termination of the sort, if they have become available.

<u>Analysis</u>:

In the optimal case, i.e $p=\log n$, the last processor on the pipeline merges two runs of length n/2 pages. Processor P_{i+1}



$\boxed{b \mid a}$ represents a run of 2 pages. The first page of the run contains the value 'a' and the second page contains the value 'b'.

Figure 1

Pipelined Merge Sort
with
3 Processors and 8 pages

starts processing after P_i has written a whole run and is ready to write the first page of a second run. Thus, P_{i+1} begins $2^{i-1}+1/2\ C_p^2$ time units after P_i (getting a page ready for output requires approximately one half a 2_page operation). This implies that the last processor P_p starts processing after time:

$$1+2+2^2+\ldots+2^{p-2}+ p/2 = 2^{p-1}-1+(p-1)/2$$

and then it has to merge the last 2 runs in n/2 time units. Thus, for p = logn, we conclude that execution time of the algorithm is:

$$n+(\log n-1)/2 \quad C_p^2 \text{ operations}$$

For the case $p<\log n$, we divide the execution into $\lceil(\log n)/p\rceil$ phases. In phase j, processor P_i mimics the action of processor $P_{(j-1)p+i}$ if logn processors were available. An important point to realize is that P_1 will be able to start execution of phase j+1 as soon as it has finished the execution of phase j, regardless of the values of p and n. We prove this statement for j=1 as follows. For any j, P_j begins execution at time $2^{j-1}-1+(j-1)/2$. Suppose we had p+1 rather than p processors, then P_{p+1} would begin its execution at time $2^p-1+p/2$. Each processor, in particular P_1, takes n/2 time units to execute its share in each phase. Thus, P_1 finishes execution of its share in phase 1 at time n/2. Since $p<\log n$, P_1 would have to mimic P_{p+1} in an optimal pipeline, that is P_1 should read the runs of size 2^p produced by P_p, and merge sort them by pairs. The first page of the second run produced by P_p appears at time $2^p-1+p/2$, but P_1 is not ready for it because $n/2>2^p-1+p/2$ for $p<\log n-2$. Therefore the second phase starts n/2 time units later than the first phase. Our argument can easily be extended to any phase, to show that phase (i+1) starts n/2 time units after phase i has started. Thus, we see that the cost of the algorithm with less than an optimal allocation of processors is:

$$((\log n/p)-1)*(n/2) + \text{Cost of last phase}$$

Let $k=\log n/p$.[2] Then the size of the first run in the last phase is $2^{(k-1)p}$ and the cost of the last phase is:

$$2^{(k-1)p}+2^{(k-1)p+1}+\ldots+2^{(k-1)p+(p-2)} + (p-1)/2 + n/2$$

which reduces to:

$$n + (p-1)/2 - n/(2^p)$$

Thus, the total cost of the algorithm is:

$$(n\log n)/2p + n/2 - n/(2^p) + (p-1)/2 \quad C_p^2 \text{ operations}$$

4.2.2. 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 [Bora80a], 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

[2] We assume that $n=2^{kp}$ for some integer k. Otherwise, the last phase may not require the use of all the processors.

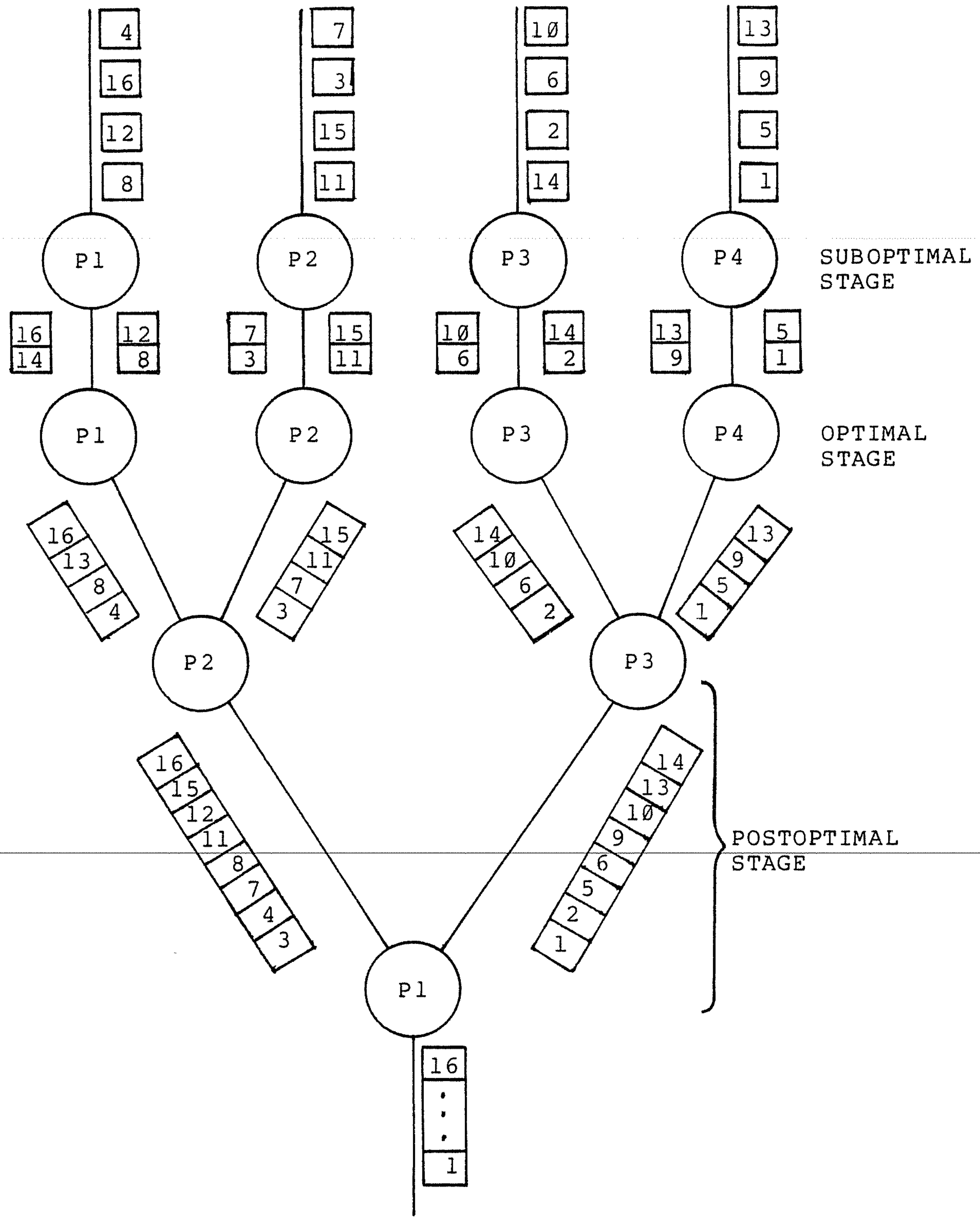


Figure 2

Parallel Binary Merge
with
4 Processors and 16 pages

pairs of longer and longer runs until the number of runs is equal to twice the number of processors. 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 2*p, 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 2*p.

When the number of runs equals 2*p each processor will merge exactly two runs of length n/2p. This phase is called 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 in a manner similar to the pipelined merge sort (described in the previous section) except that each processor outputs a single run rather than two or more. By pipelining data between levels of the tree, a parent is able to start its execution a single time unit after both its children (i.e. as soon as its children have produced one page). Therefore, the cost of the postoptimal stage will be a 2-page operation for each level of the tree plus the cost for the root processor to merge two 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 $\log(n/2) + n/2 \; C_p^2$ 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. $n/2p \; C_p^2$ 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 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 process two runs of length $n/2$. Therefore, the total execution time of the algorithm expressed in C_p^2 units is:

$$\underbrace{(n/2p)*\log(n/2p)}_{\text{suboptimal}} + \underbrace{n/2p}_{\text{optimal}} + \underbrace{\log p - 1 + n/2}_{\text{postoptimal}}$$

which can be expressed as:

$$(n\log n)/2p + n/2 - (n/2p - 1)*(\log p) - 1$$

4.2.3. Block Bitonic Sort

Description:

Batcher's bitonic sort algorithm sorts n numbers with n/2 comparator modules in 1/2 logn(logn + 1) steps [Batc68]. 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 [Ston71].

As first suggested in [Baud78], 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 1/2 logn(logn + 1) 2-page operations. Execution of this algorithm using two processors is illustrated in Figure 3.

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 produced 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 binary merge 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, $(2p)^2$, ... 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

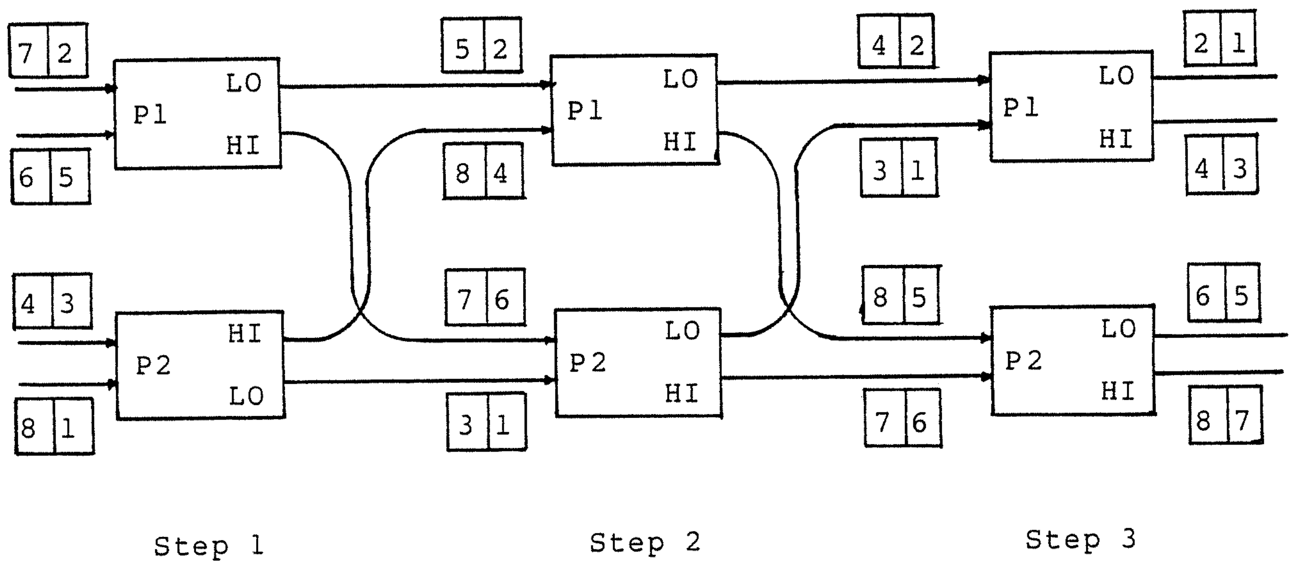


Figure 3

Block Bitonic Sort
with
2 Processors and 4 Runs of 2 Pages each

suboptimal phase of the parallel binary merge and completes in $(n/2p)*\log(n/2p)*C_p^2$ time units. Then, the bitonic sort algorithm is applied to the 2p blocks of size n/2p. This step requires:

$$(n/2p) * (\log 2p)/2 * (\log 2p + 1) \quad C_p^2 \text{ operations}$$

The total cost is thus:

$$n/2p \; [\; \log n + 1/2 \; (\log^2 2p - \log 2p) \;] \; C_p^2$$

4.2.4. Performance Comparison of the 3 Sorting Algorithms

Since all three algorithms presented in this section execute essentially in $n\log n/2p \;\; C_p^2$ 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 formulae established for the algorithms (4.2.1, 4.2.2, 4.2.3) are linear in n. In Figure 4 we have plotted the performance of each algorithm for a fixed number of processors and a varying number of pages to be sorted. As established by these graphs, the factors which are linear in n are such that the pipelined merge sort does not perform as well as either the parallel binary merge or the block bitonic sort. Also, our results show that when there are more than 16 processors the block bitonic sort outperforms the parallel binary merge (this fact can be proven analytically by comparing formula 4.2.2 to formula 4.2.3).

4.3. The Project Operation

The projection of a relation with domains d1,d2,...,dn on a subset of domains di,dj,...,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

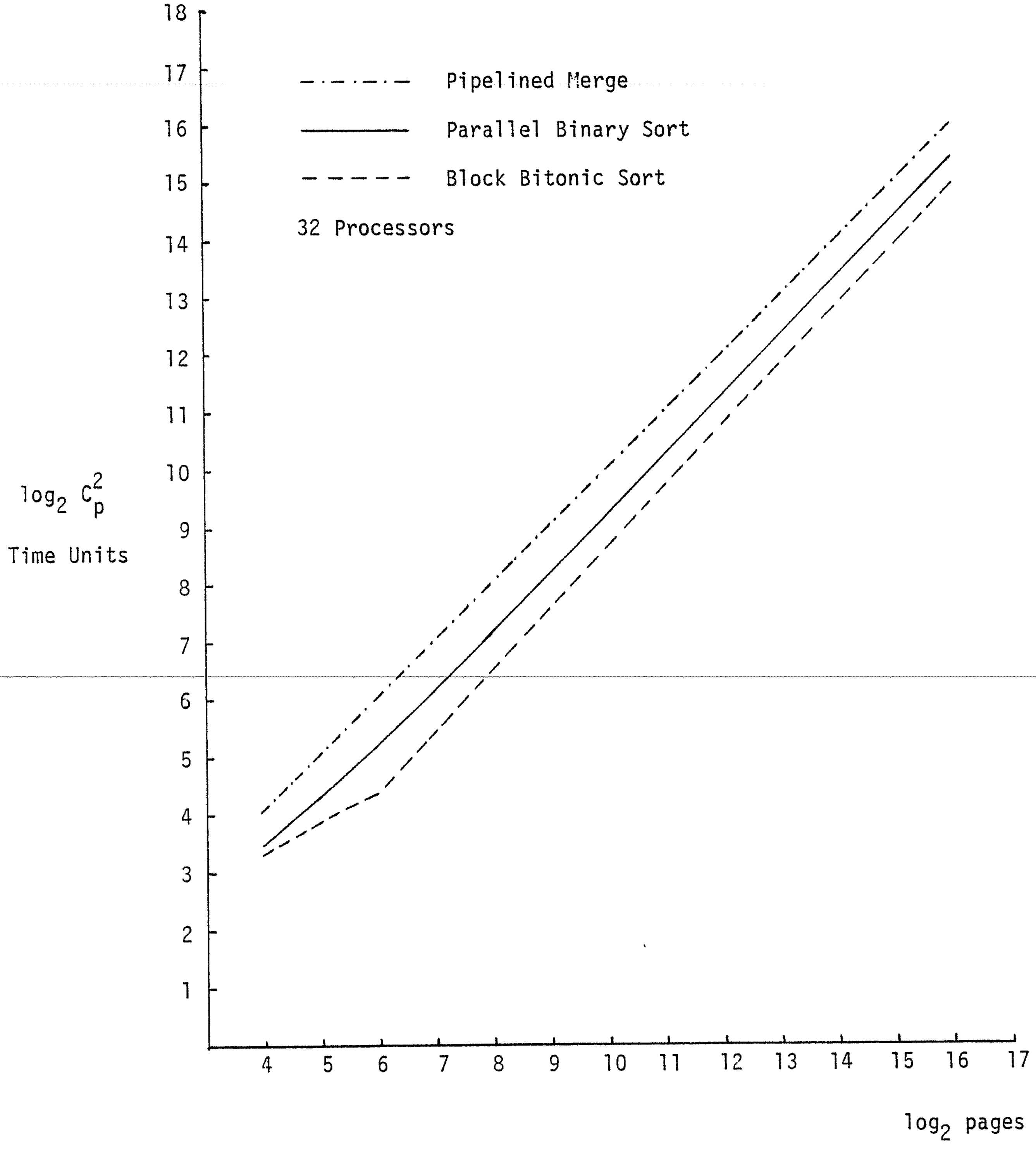


Figure 4

Comparison of the 3 Sorting Algorithms

di,dj,...,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 on an associative-disk type database machine, the second is much more complex and requires nonlinear (with respect to the number of tuples) time. 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 non-key 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 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 heavily on a hardware broadcast facility.

We assume that pages have already been reduced to a vertical form by the previous operation and there are no intra-page duplicates. Each processor reads one page. Let a processor be labeled according to the page number of the page it read (that is, the processor that read page i is known as P_i). Starting with P_p, and continuing with $P_{p-1}, \ldots, P_2$, each processor, in turn, broadcasts its page and then exits. If processor P_j receives page i, then $j<i$. P_j compares the two pages and eliminates any duplicates found from its page. Note 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 or pages, our algorithm works in a number of distinct phases. Each phase produces p projected pages and sees p less pages than the previous phase. In phase i there are (i-1)*p pages that already been projected, p pages in the processors' memories, and n-(i*p) nonprojected pages. The phase begins by broadcasting the n-(i*p) nonprojected pages to the p processors for duplicate removal. After this step has completed, P_p broadcasts its page and exits. The remaining processors follow suit. The cost of phase i is thus:

$$C_r + (n-i*p)*(C_r+C_m) + (p-1)*(C_r+C_m+C_w) + C_w$$

If n = p*m, there are m phases and the total cost of the algorithm is:

$$m*C_r + m(m-1)p/2 * (C_r+C_m) + m(p-1)*(C_r+C_m+C_w) + mC_w$$

This may be rewritten as:

$$(n^2/2p+n/2)*(C_r+C_m) - (n/p)*C_m + nC_w$$

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 mod p processors and thus terminate faster.

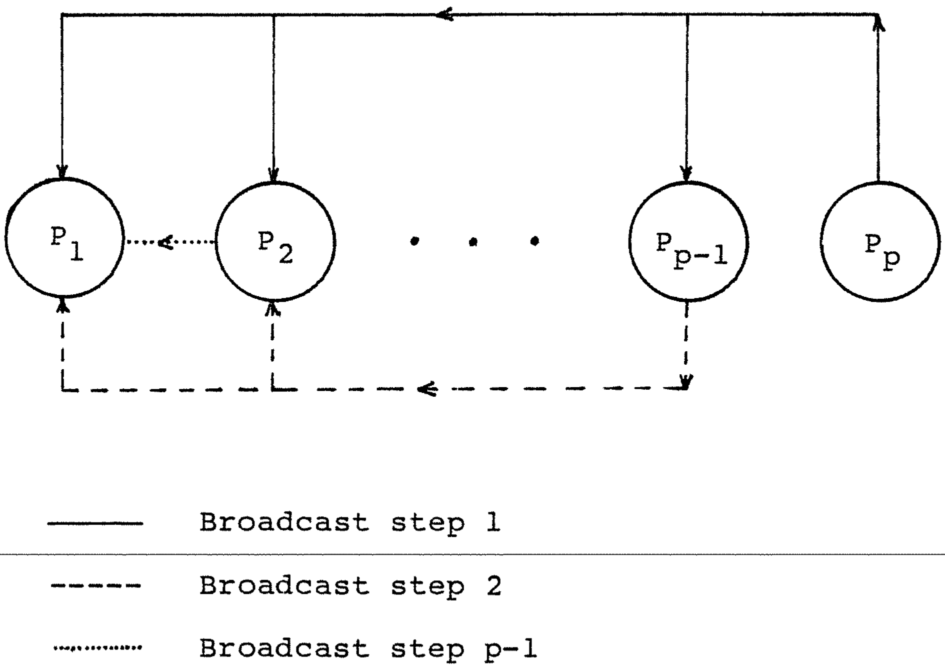


Figure 5

Projection by Broadcast

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 1Ø times on the average and the duplicates are uniformly distributed among the pages, up to 1Ø 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 $n >> p$, 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.

Several elegant methods for the removal of duplicates appear in a recent publication [Good8Øa]. However, each of these methods requires a prespecified architecture and assumes that the relation fits into the processors' memory. Furthermore, the analysis presented only includes the case where there are no duplicates and the case in which all tuples are identical. Our broadcast algorithm, for the case that p=n, is similar to the method described in [Good8Øa] for the common bus architecture.

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 "sort-merge" algorithm. The "nested loops" join algorithm relies heavily on a broadcast facility, while the "sort-merge" 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 by [Babb79], [Good8Øb]. A performance comparison of the hashing strategy with either the nested loops or the sort-merge joins is beyond the scope of this paper, but we plan to incorporate it in future work. Also, as mentioned in the introduction, we have not yet examined parallel join algorithms which use indices.

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 to 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 T, and suppose $n \geq m$. Let p be the number of processors assigned to perform the join of R and T. 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 T. If $p = n$, the execution time of this algorithm is:

$$\begin{aligned} T_{\text{nested loops}} &= T(\text{read a page of R}) \\ &+ m*T(\text{broadcast a page of T}) \\ &+ m*T(\text{join 2 pages}) \end{aligned}$$

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

$$S = \text{size}(R \text{ join } T)/(m*n)$$

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

$$T_{\text{nested loops}} = n/p(C_r + m*(C_r + C_m + S*(C_{so} + C_w)))$$

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

4.4.2. Sort-Merge Join

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 subsequent join's join attribute. The merge and the sort operations are executed by a single processor. Since the relations have been sorted, the complexity of the join step is the cost of merging two sorted files and sequentially sorting the result pages. The time to perform the join is equal to:

$$\begin{aligned} T = {} & T(\text{sort } R) + T(\text{sort } T) \\ & + T(\text{merge 2 sorted files}) \\ & + T(\text{sequential sort of pages of result file})) \\ = {} & T_{sort}(n) + T_{sort}(m) \\ & + (n+m)*C_r + \max(n,m)*C_m \\ & + m*n*S*(C_{so} + C_w) \end{aligned}$$

If the sort steps are performed using our block bitonic algorithm, the join cost is:

$$\begin{aligned} T = {} & [(n/2p)\log n + (m/2p)\log m + 1/4p(\log^2 p - \log 2p)(n+m)]C_p^2 \\ & + (n+m)*C_r + \max(n,m)*C_m \\ & + m*n*S*(C_{so} + C_w) \end{aligned}$$

We have identified a number of ways in which the execution time of this algorithm can be improved. The first is to somehow overlap sorting of the two files. This can significantly improve the performance of the join, if a "pipelined type" algorithm is employed. To illustrate this assumption, let us consider the simplified case where R and T both contain n pages and the number of processors is also n. In this case half of the time to sort R using the pipeline merge algorithm is for propagating the last page of R through the pipeline; now, as soon as the first processor has processed the nth page of R, it can start reading and processing pages of T. By reusing the pipeline, we are able to replace $2T_{sort}(n)$ by $3/2T_{sort}(n)$ in the execution time for the join. Furthermore, since pages of T emerge one at a time from the pipeline, we may begin merging the 2 sorted relations as soon as the first page of the sorted T relation is produced.

A second improvement is to leave the pages of the result 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 might be desirable if the result relation is the final result of a query, or if it becomes the source relation for a subsequent join 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 to 9. Each figure contains three curves, each showing the ratio of the nested loops execution time to the sort merge execution time for three different joins. In order to illustrate the variation in the relative performance of the algorithms we divided the Y-axis in each figure into two parts, each using a different scale. The bottom part of the axis shows the region in which the nested loops algorithm outperforms the sort merge (ratio < 1) and the higher part shows the other case. For reasons of clarity the X-axis shows the logarithm (base 2) of the number of processors used rather than the actual number. Our assumptions about the processors' capabilities are specified in appendix A.

Figures 6 and 7 present the results for selectivity factors of Ø.Ø1 and Ø.ØØ1 with no sorting of result pages. We assumed that each page contained three hundred, 55 byte tuples but 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 sort merge 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 sort merge (except for small numbers of processors). It should be noted that for lower selectivity factor values the sort merge algorithm performs better than the nested loops. This is because the merge step (handled by a single processor) has to output less pages. Since in the nested loops algorithm, the result relation is divided among all the processors, a reduction in its size has very little effect on the total execution time.

Figure 8 shows a similar result for the same joins with selectivity factor of Ø.ØØ1 with sorting of result pages (this is the case analyzed above). Finally, Figure 9 (surprisingly) shows

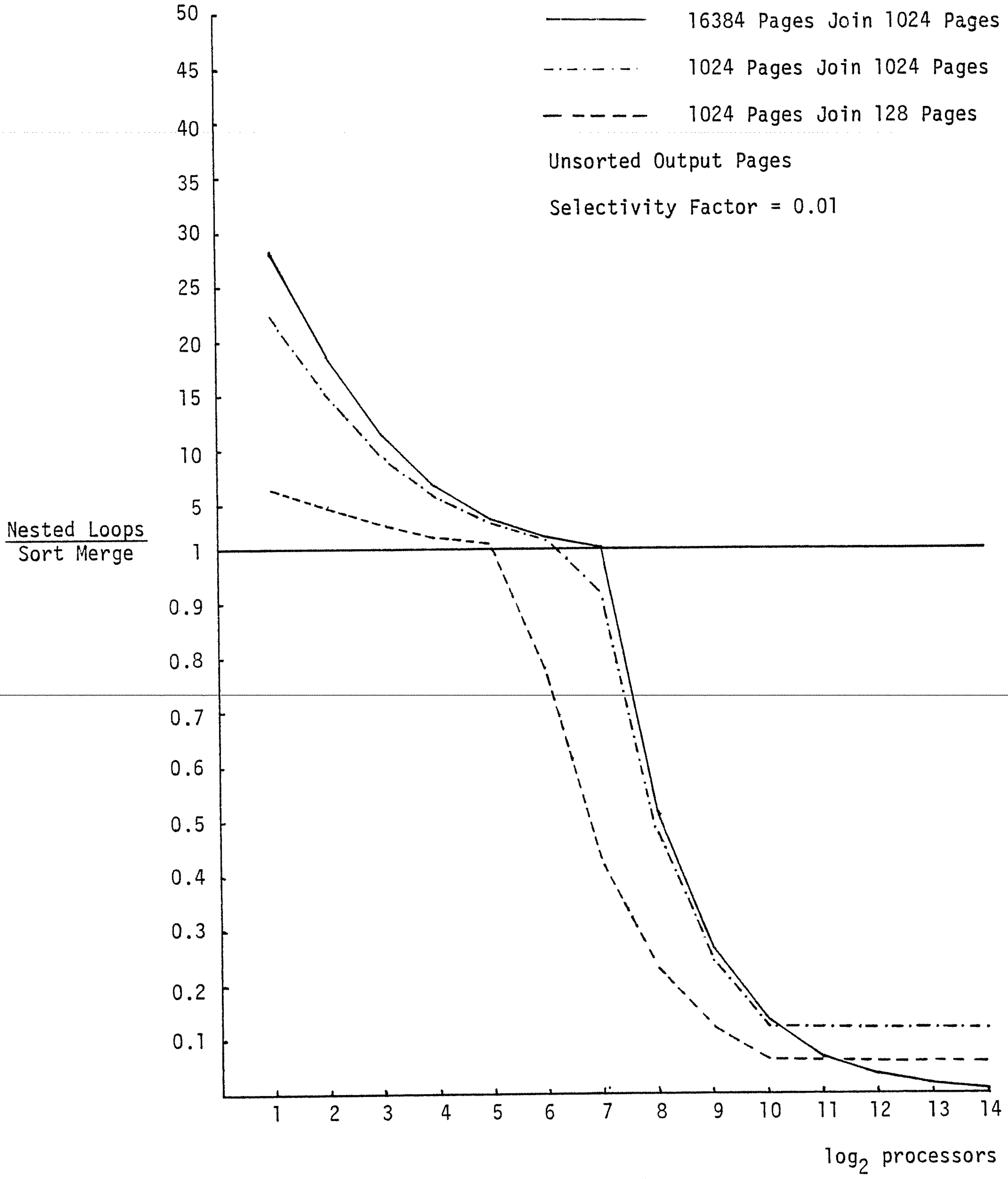


Figure 6

Comparison of the 2 Join Algorithms

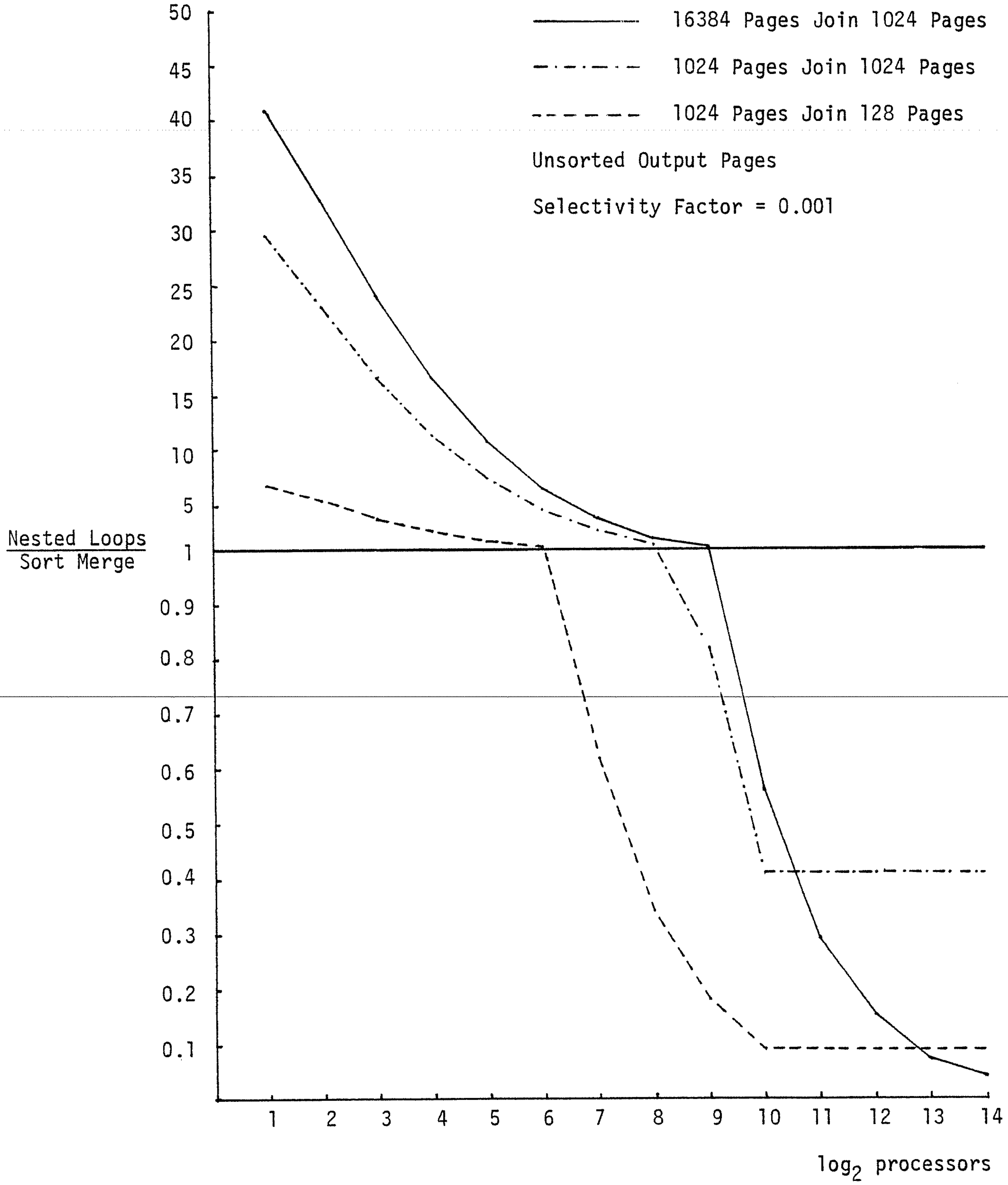


Figure 7

Comparison of the 2 Join Algorithms

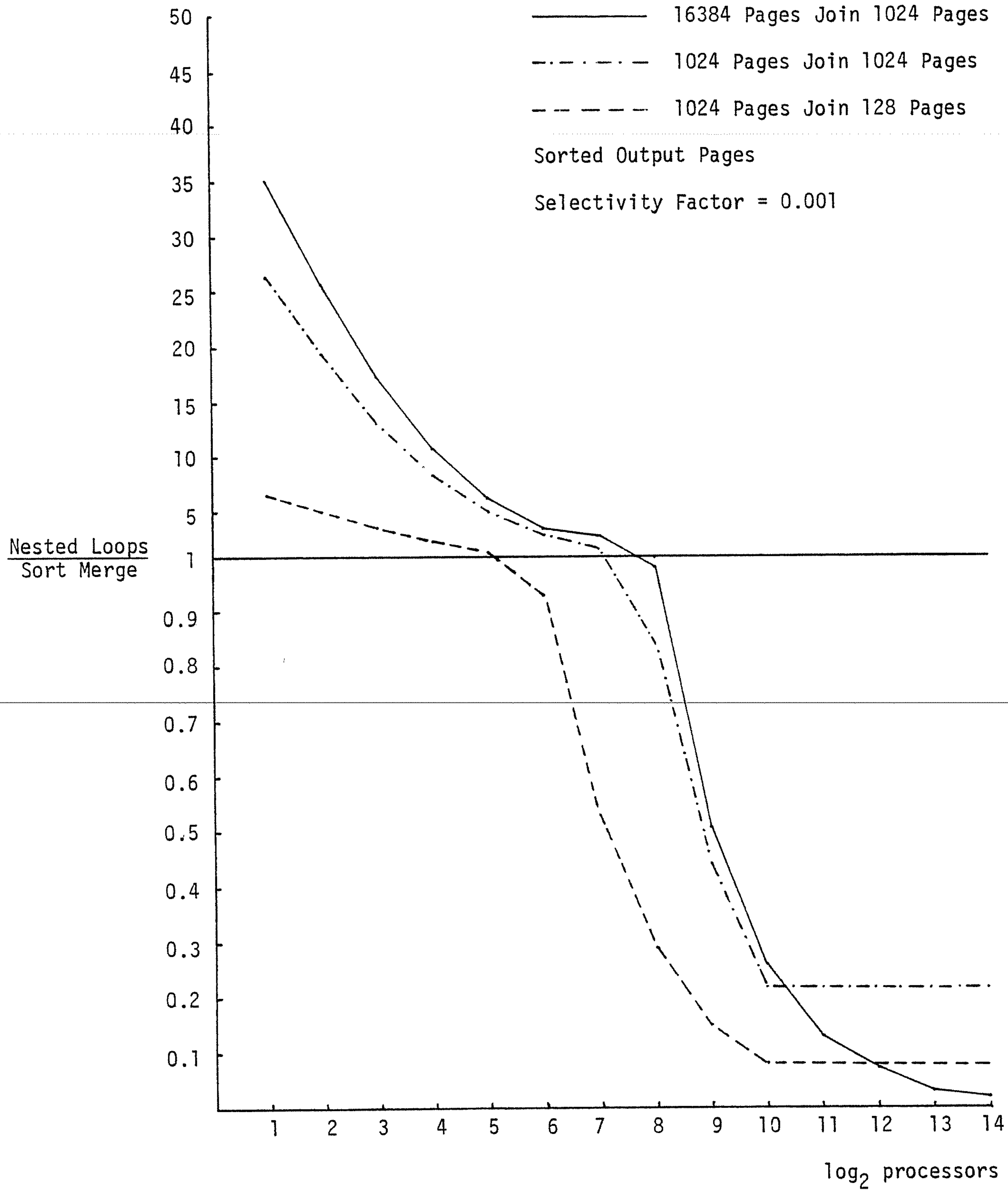


Figure 8

Comparison of the 2 Join Algorithms

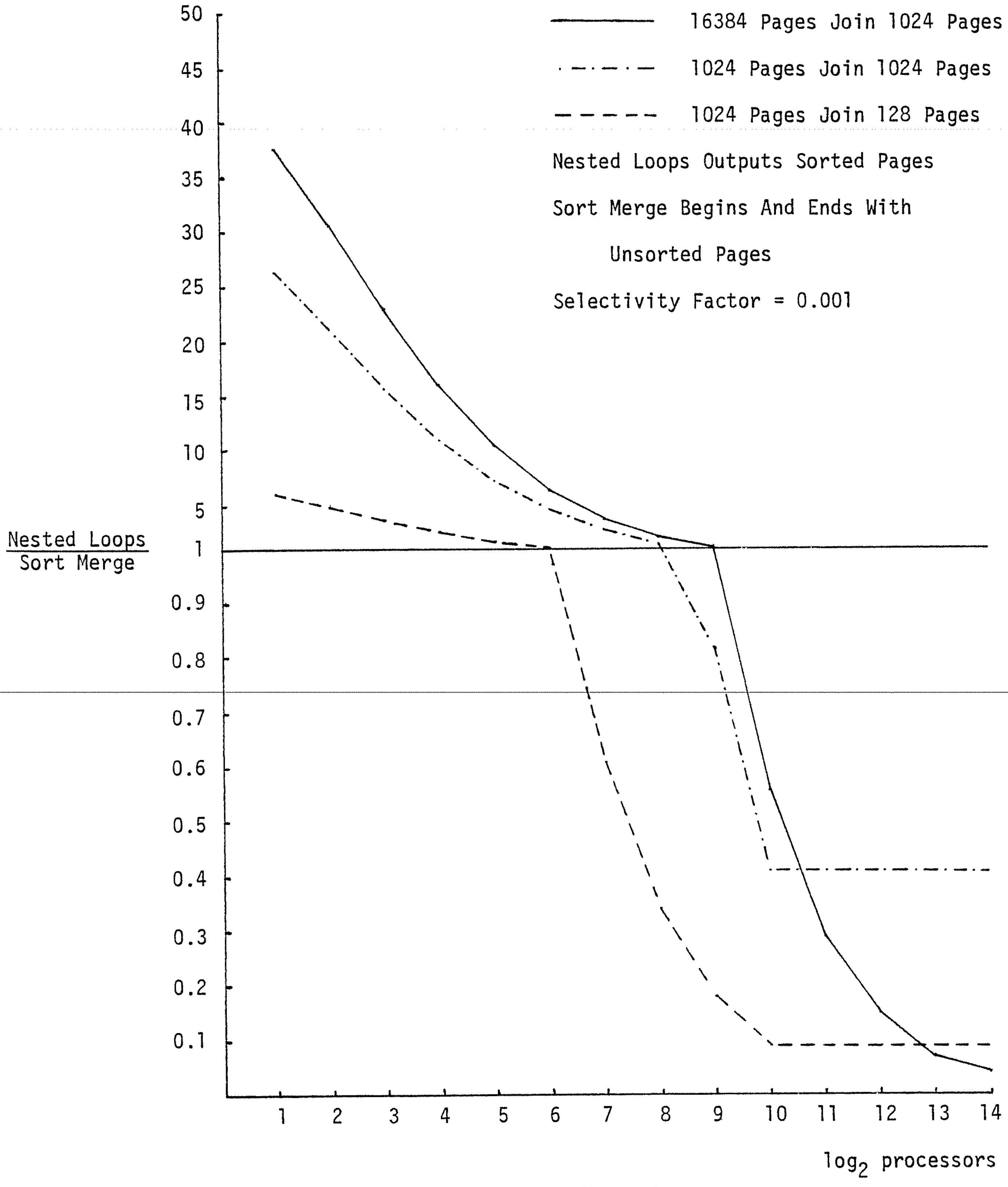


Figure 9

Comparison of the 2 Join Algorithms

only a slight improvement for the case that the sort merge algorithm sorts pages in the beginning (and outputs unsorted result pages) and the nested loops algorithm sorts its result pages at the end (this is the second improvement suggested in the previous section).

4.5. Aggregate Operations

In contrast with the relational operations join, project, select, etc., there is no commonly accepted set of aggregate operations among existing relational database systems. For our purposes, we will adopt the facilities provided by INGRES [Yous77] as being representative and develop algorithms to process them (see [Epst79] for a presentation of algorithms for processing aggregates in a uniprocessor environment). We distinguish between "scalar" aggregates and aggregate "functions". Scalar aggregates are aggregations (average, max, etc.) over an entire relation. Aggregate functions first divide a relation into non-intersecting partitions (based on some attribute value, e.g. sex) and then compute scalar aggregates on the individual partitions. Thus, given a source relation, scalar aggregates compute a single result while aggregate functions produce a set of results (i.e. a result 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 by_qual )
                            where src_qual

by_list:       by  att-1  by  att-2  by  ...  by  att-n

agg_op:        sum, avg, count, max, min, sumu, avgu, countu
```

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 1Ø).

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

Employee Relation:

Name	Dept	Task	Salary	Manager
Smith	Toys	Clerk	3ØØ.ØØ	Johnson
Miller	Shoes	Buyer	65Ø.ØØ	Bergman
Jones	Books	Acct	55Ø.ØØ	Harris
Brown	Shoes	Clerk	4ØØ.ØØ	Conners

countu (Emp.Dept) = 3

Figure 1Ø: Example of a "unique" scalar aggregate

function may depend on qualifications outside the aggregate (src_qual) (this will be 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 inter-relation operations so the relation must be pre-processed before computing the aggregate. To 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). For aggregate functions, we want to account for the space required to maintain these fields ("result tuples") and that is the purpose of parameter 'r' below. In the following discussion, we assume these parameters:

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

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 pre-processing of the source relation since inter-relation 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. The cost of the algorithm is then:

```
Tsc_agg = T(exec qual)              (if complex qual)
        + T(sort,project)           (if unique agg_op)
        + T(partial results)
        + T(combine p partials)
```

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

$$T(\text{partial results}) = (n/p) * (C_r + (q+1)*C_{sc}) + C_{msg}$$

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) = p*C. The final formula is thus:

```
Tsc_agg = T(exec qual)                      (if qualification)
        + T(sort,project)                   (if unique aggregate)
```
$$+ (n/p) * (C_r + (q+1)*C_{sc}) + C_{msg}$$
$$+ p*C$$

4.5.2. Aggregate Functions

In this section we describe and analyze the performance of 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 a count of the number of employees under each manager making 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.mgr!="Smith")
         where  emp.sal > 500
```

Clearly, in this case we want to include the count for all managers other than Smith. Thus, we need to distinguish between restrictions on the source tuples and 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 "by_qual"), in addition to selecting a subset of the source relation, have the effect of eliminating unwanted partitions (e.g. manager Smith above). While qualifications outside the aggregate (the "src_qual") primarily affect the source relation they may have the undesirable side effect of removing desired partitions (e.g. managers for whom no employees earn more than 5ØØ) and we must correct for this.

When an aggregate function contains a 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.

4.5.2.1. Algorithm 1: Sub-queries with a Parallel Merge

Our first algorithm is similar to the scalar aggregate algorithm and works best when the number of partitions is small (m less than r, the number of result tuples in a page). 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 (at most m). 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. If m, the number of partitions, is less than r, the number of result tuples per page, we have at most p pages to merge (since each processor produces one page). If m is larger than r the processors may produce several result pages each. In the extreme case ($m = k*n$, e.g. partitioning on a relation key) the algorithm could conceivably produce as many result pages as input pages.

The cost of this algorithm (assuming no qualifications and a non-unique aggregate) may be computed as:

Talg1 = T(produce partial result pages) + T(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:

$$(n/p)(C_r + k*((\log x)+1)*C)$$

We need to estimate the number of result 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 result pages produced by each processor if one assumes that the partitions uniformly distributed in the relation and so each processor sees all the partitions. Therefore, the cost for a processor to output its partial result pages is $t*C_w$.

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 1/2 the result tuples be moved down. For $x = \min(m,r)$ partitions, $x(x+1)/4$ tuples moves will be required ($x(x+1)/4$ is equal to the summation of $i/2$ for $i=1$ to x). Thus, the cost to process each of the t pages produced by a processor is:

$$t*[x(x+1)V/4 + C_w]$$

The parallel "merge" we use in the second stage is not a true merge since two partial result pages are combined to form a single result 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 <u>pipelined</u> parallel binary merge (see Section 4.2.2) 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 $C_{m'} = 2r(2C+V)$ denote the cost of a merge of two result 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:

$$\begin{aligned}\text{Talg1} = {} & (n/p)(C_r + k*((\log x)+1)*C) \\ & + t*(x(x+1)V/4 + C_w) \\ & + (t/2)\log(t/2)\, C^{2w}_p \\ & + (t+ \log p)(2C_r + C_{m'} + C_w)\end{aligned}$$

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 a 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 Ø. The size of this R' will be $\lceil m/r \rceil$ pages, i.e. t. 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 a 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 pre-processing step in which the source relation is sorted on the by_list in order to eliminate duplicates. We must account for this cost also. The final formula for Algorithm 1 is thus:

$$
\begin{aligned}
\text{Talg1} = {} & T(\text{execute by\_qual}) && \text{/* if by\_qual */} \\
& + T(\text{project on by\_list}) && \text{/* if src\_qual */} \\
& + T(\text{execute src\_qual}) && \text{/* if complex src\_qual */} \\
& + T(\text{project,sort}) && \text{/*if unique aggregate, eliminate duplicates*/} \\
& \quad \text{/* process partitions */} \\
& + (n/p)(C_r + k*((\log x)+1)*C) && \text{/* } x = \min(r,m) \text{ */} \\
& + (n/p)(q * C_{sc}) && \text{/* if simple src\_qual */} \\
& + t*(x(x+1)V/4\,C + C_w) && \text{/* } t = \lceil m/r \rceil \text{ */} \\
& \quad \text{/* perform parallel merge */} \\
& + (t/2)\log(t/2)\, C_p^2 \\
& + t * (2C_r + C_{m'} + C_w) && \text{/* if src\_qual */} \\
& + (t + \log p)(2C_r + C_{m'} + C_w)
\end{aligned}
$$

4.5.2.2. Algorithm 2: Project by-list and Broadcast Source Relation

This algorithm exploits the ability of an 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 m 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 (m/p) partitions. If the number of partitions is greater than r (the number of result tuples per page), the source relation may have to be broadcast more than once. The cost of this algorithm (assuming no qualifications and non-unique aggregates) may be summarized as:

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

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:

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

If the query has a simple src_qual, it may be processed the same time as the aggregate is computed. This adds an additional q comparisons per tuple (see parameters defined above). 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. This requires an additional comparison per tuple. Thus, the total cost for this algorithm is:

```
Talg2 = T(exec by_qual)       /* if by_qual */
      + T(project by_list)    /* determine list of partitions */
      + T(exec src_qual)      /* if complex src_qual */
      + T(sort source) + bn($C_{sc}$)    /* if unique aggregate */
      + b(n($C_r$ + (log x)$C_{sc}$) + $C_w$) /* process partitions */
      + bn(q * $C_{sc}$)                 /* if simple src_qual */
```

4.5.2.3. Comparison

In order to compare the performance of these two algorithms we selected two queries, one without a src_qual and one with a src_qual. In Figures 11 and 12 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 A). Figure 11 shows that Algorithm 1 is significantly superior to Algorithm

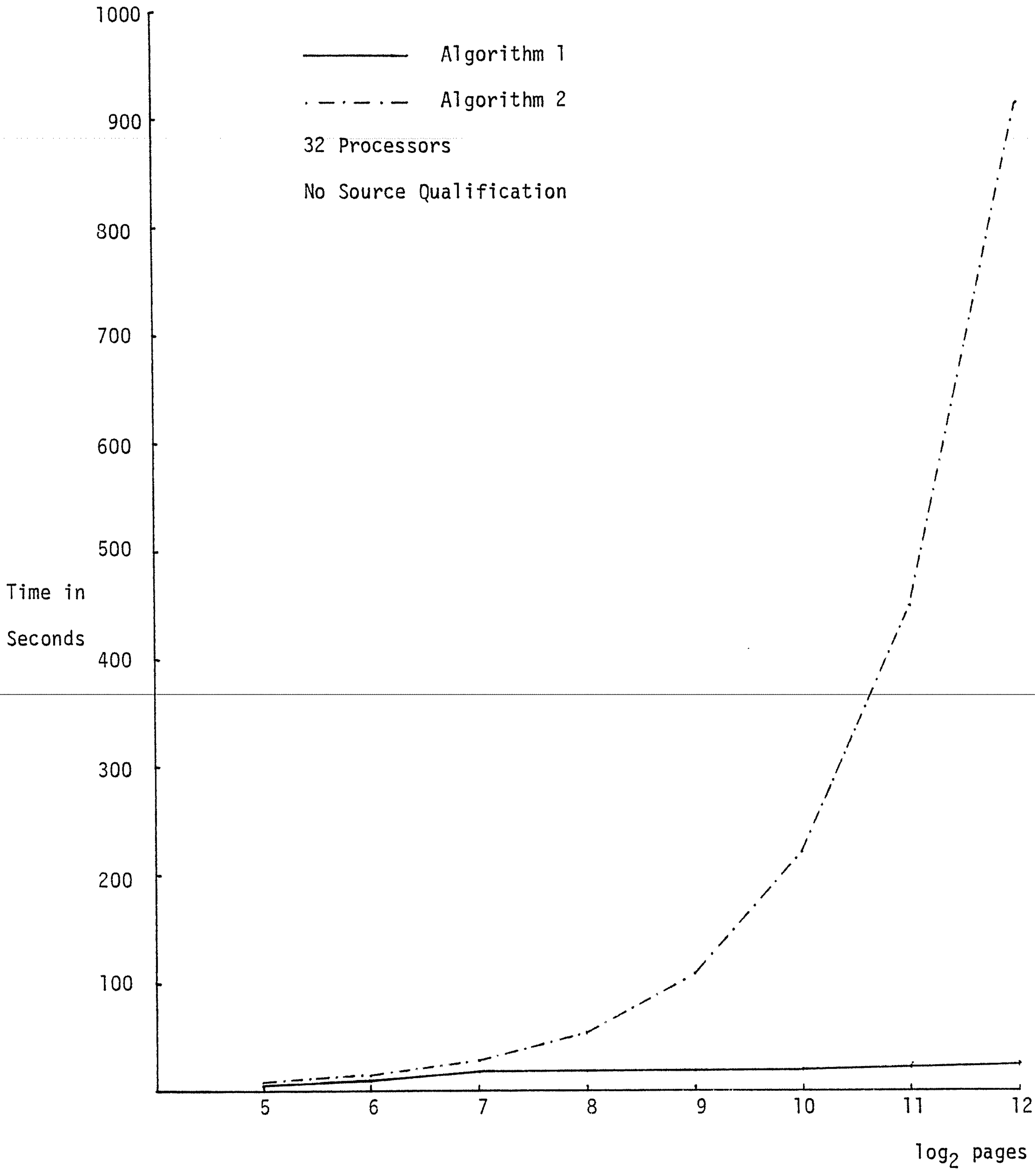


Figure 11

Comparison of the 2 Aggregate Algorithms

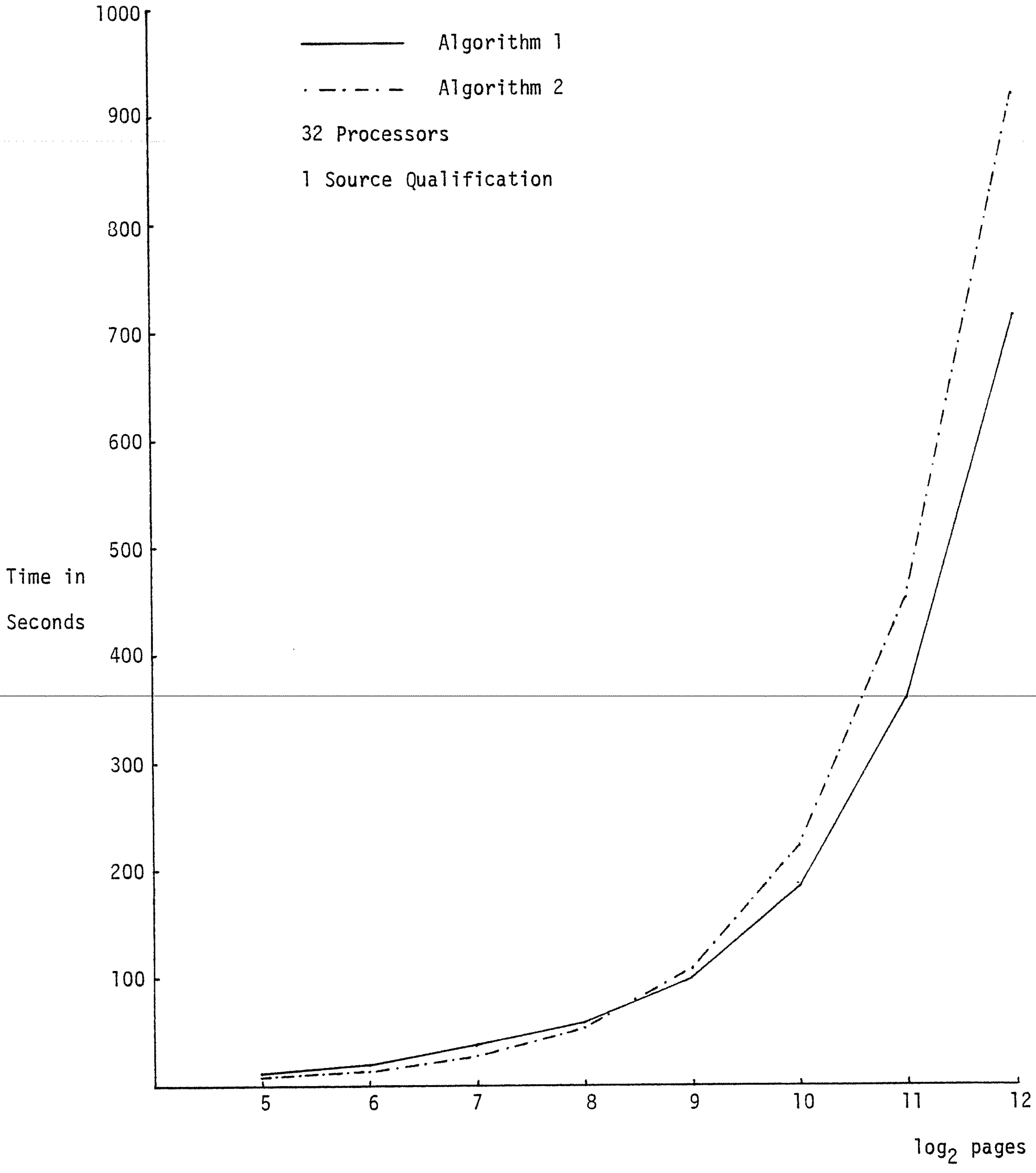


Figure 12

Comparison of the 2 Aggregate Algorithms

2 (up to two orders of magnitude) when the query does not contain a src_qual. However, as shown in Figure 12, when the query contains a src_qual, Algorithm 2 is superior except when the relation is very large. Furthermore, the performance of Algorithm 1 is sensitive to the value of t (the number of result pages produced by each processor). 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 sub-relation for each partition. As each sub-relation was produced by the sort, another processor immediately read the sub-relation 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 (e.g. project) 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 and which can be adjusted to fit different multiprocessor architectures. We have compared not only existing parallel algorithms, but other algorithms which we generalized to fit in a multiprocessor environment. In addition, we introduced (for the first time) algorithms for the parallel execution of update and aggregate operations.

Sorting can be used as a basic building block in the design of algorithms for parallel processing of relation database operations. This paper discussed three parallel sorting algorithms: the pipelined merge sort, the block-bitonic sort, and the parallel binary merge sort. While the first two algorithms have been suggested previously, we generalized them to external sorting algorithms in order to handle the case where the number of pages is significantly larger than the number of processors. The parallel binary merge algorithm with pipelining between stages is a new algorithm. Although the block bitonic sort is, in general, superior (see Figure 4), the other two algorithms perform relatively well. One conclusion that might be drawn from these results is that when I/O costs are included in the analysis of a parallel sorting algorithm (as we have done), they dominate its execution time.

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 sort merge 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 one possible area for future research.

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 that we have mentioned earlier 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 to develop techniques for evaluating the cost of controlling multiple processors on complex algorithms (such as the parallel sort merge 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. Finally, 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.

6. References

[Babb79] Babb E., "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS, Vol. 4, No. 1, Mar. 1979.

[Banc8Ø] Bancilhon F. and M. Scholl, "Design of a Backend Processor for a Data Base Machine," Proc. of the ACM SIGMOD 198Ø Int'l Conf. of Management of Data, May 198Ø.

[Bane78] Banerjee J., R.I. Baum, and D.K. Hsiao, "Concepts and Capabilities of a Database Computer," ACM TODS, Vol. 3, No. 4, Dec. 1978.

[Batc68] Batcher K.E., "Sorting Networks and Their Applications," 1968 Spring Joint Computer Conf., AFIPS Proc., Vol. 32, 1968.

[Baud78] Baudet G. and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," IEEE-TC, Vol.c-27, No. 1, Jan. 1978.

[Bora8Øa] Boral H. and D.J. DeWitt, "Design Considerations for Data-flow Database Machines," Proc. of the ACM SIGMOD 198Ø Int'l Conf. of Management of Data, May 198Ø.

[Bora8Øb] Boral H. and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," To Appear in ACM TODS. Also Comp. Sci. Tech. Rep. No. 368, University of Wisconsin Oct. 1979.

[Desp78] Despain A.M., and D.A. Patterson, "X-TREE: A Tree Structured Multi-processor Computer Architecture," Conf. Proc. of the 5th Annual Symp. on Computer Architecture, 1978.

[DeWi79a] DeWitt D.J., "Query Execution in DIRECT," Proc. of the ACM SIGMOD 1979 Int'l Conf. of Management of Data, May 1979.

[DeWi79b] DeWitt D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE-TC, Vol. c-28, No. 6, June 1979.

[Epst79] Epstein R., "Techniques for Processing of Aggregates in Relational Database Systems," Memo. No. UCB/ERL M79/8, Elec. Research Lab., Coll. of Eng., UCB, Feb. 1979.

[Even74] Even S., "Parallelism in Tape Sorting," CACM, Vol. 17, No. 4, Apr. 1974.

[Fran8Ø] Franklin M.A., "VLSI Performance Comparison of Banyan and Crossbar Communications Networks," Proc. of the Workshop on Interconnection Networks for Parallel and Distributed

Processing, Apr. 1980.

[Gavr75] Gavril F., "Merging with Parallel Processors," CACM, Vol. 18, No. 10, Oct. 1975.

[Goke73] Goke, G.R., and G.J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," Conf. Proc. of the 1st Symp. on Computer Architecture, Dec. 1973.

[Good80a] Goodman J.R. and A.M. Despain, "A Study of the Interconnection of Multiple Processors in a Database Environment," Proc. of the 1980 Int'l Conf. on Parallel Processing.

[Good80b] Goodman J.R. - Personal Communication.

[Hawt80] Hawthorn P. and D.J. DeWitt, "Performance Evaluation of Database Machines," Submitted to IEEE-TC.

[Hirs78] Hirschberg D.S., "Fast Parallel Sorting Algorithms," CACM Vol. 21, No. 8, Aug. 1978.

[Hsia78] Hsiao D.K. and K. Kannan, "Simulation Studies of the Database Computer (DBC)," Technical Report OSU-CISRC-TR-78-1, Computer & Information Science Research Center, The Ohio State University, Feb. 1978.

[Knut75] Knuth D.E., *The Art of Computer Programming - Sorting and Searching* Addison-Wesley, 1975, p. 160.

[Leil78] Leilich H.O., G. Stiege, and H.Ch. Zeidler, "A Search Processor for Data Base Management Systems," Proc. 4th Conf. on Very Large Databases, 1978.

[Lin76] Lin C.S., D.C.P. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," ACM TODS, Vol. 1, No. 1, Mar. 1976.

[Lipo80] Lipovski, J. - Personal Communication, 1980.

[Mull75] Muller D.E. and F.P. Preparata, "Bounds for Complexity of Networks for Sorting and for Switching," JACM Apr. 1975.

[Ozka75] Ozkarahan E.A., S.A. Schuster, and K.C. Smith, "RAP - An Associative Processor for Data Base Management," Proc. 1975 NCC, Vol. 45, AFIPS Press, Montvale N.J.

[Prep78] Preparata F.P., "New Parallel Sorting Schemes," IEEE-TC, Vol. c-27, No. 7, July 1978.

[Slot70] Slotnick D.L., "Logic Per Track Device," in *Advances in Computers*, Vol 10, J. Tou, ed., Academic Press, N.Y., 1970.

[Ston71] Stone H.S., "Parallel Processing with the Perfect

Shuffle," IEEE-TC, Vol. c-2Ø, No. 2, Feb. 1971.

[Su75] Su S.Y.W. and G.J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases," Proc. Int'l Conf. Very Large Data Bases, Sept. 1975.

[Thom77] Thompson C.D. and H.T. Kung, "Sorting on a Mesh Connected Parallel Computer," CACM, Vol. 2Ø, No. 4, Apr. 1977.

[Upch8Ø] Upchurch E. - Personal Communication.

[Yous77] Youssefi K. et. al., "INGRES Version 6.Ø Reference Manual,".

Appendix A

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). We assumed that:

Page size of 16K bytes.

C - the time to compare two attributes, is 1Ø microseconds.

V - the time to move a tuple, is based on the cost of 1.5 microseconds to move a single word. Thus, for a tuple length of 15Ø bytes, V is 225 microseconds.

R_m - the time to transfer a page between mass storage and the cache, was assumed to be 28 milliseconds. This is based on a transfer time of 2Ø milliseconds, latency time of 8 milliseconds, and negligible track seek time.

R_c - the time to transfer a page from the cache to the processor's memory, was assumed to be 16 milliseconds, based on a processor bus bandwidth of approximately 1 megabyte per second.

The cache hit ratios, H and H', were assigned the values .85 and .35 respectively.

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

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