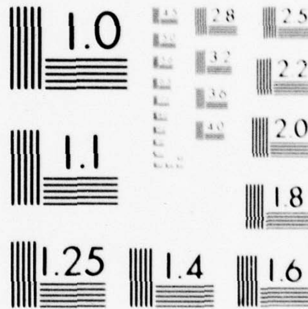
AD-A068 661 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 9/2
PARALLEL BITONIC RECORD SORT - AN EFFECTIVE ALGORITHM FOR THE R--ETC(U)
MAR 79 J BANERJEE, D K HSIAO N00014-75-C-0573
UNCLASSIFIED OSU-CISRU-TR-79-1 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

AD A068661

DDC FILE COPY

TECHNICAL REPORT SERIES

LEVEL II

12

DDC
RECEIVED
MAY 16 1979
C

This document has been approved for public release and sale; its distribution is unlimited.

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

79 04 05 028

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

AD A068661

DDC FILE COPY

OSU-CISRC-TR-79-1

(12)

PARALLEL BITONIC RECORD SORT - AN EFFECTIVE ALGORITHM FOR THE REALIZATION OF A POST PROCESSOR

BY

JAYANTA BANERJEE
AND
DAVID K. HSIAO

DDC
RECEIVED
MAY 16 1979
C

Work Performed Under
Contract N00014-75-C-0573
Office of Naval Research

Computer and Information Science Research Center
The Ohio State University
Columbus, Ohio 43210
March 1979

This document has been approved for public release and sale; its distribution is unlimited.

79 04 05 023

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Entry
1. REPORT NUMBER	OSU-CISRC-79-1 TR-79-1
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	Parallel Bitonic Record Sort - An Effective Algorithm for the Realization of a Post Processor
5. TYPE OF REPORT & PERIOD COVERED	Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Jayanta Banerjee, David K. Hsiao
8. CONTRACT OR GRANT NUMBER(s)	N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Office of Naval Research, Information Systems Program, Washington, D.C. 20360
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS	
12. REPORT DATE	Mar 1979
13. NUMBER OF PAGES	24
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	
15. SECURITY CLASS. (of this report)	
15a. DECLASSIFICATION DOWNGRADING SCHEDULE	

16. DISTRIBUTION STATEMENT (of this Report)

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 102IP	ONR, Pasadena

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database computer, record ordering, sort on value, bitonic sort, parallel sort, parallel bitonic sort, processor-to-processor interconnections, post processing, post processing controller

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A fast and parallel sorting algorithm is proposed. The algorithm is an improvement over the bitonic sort. This algorithm is particularly suitable for sorting records on the basis of field values of the records. Furthermore, it lends itself effectively for the realization of a hardware sorter with multiple parallel processors and minimal interconnections among the processors.

An analysis of the algorithm indicates that (1) using P processors with some sequential memory, there is the need of only log P interconnections among the processors, (2) it is possible to sort MP records in $O(M\log M + M\log^2 P)$ time

DD FORM 1473, 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

407 586

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

where P is a power of 2 and M, the number of records assigned to each processor, can be any positive integer.

TABLE OF CONTENTS

1. INTRODUCTION 1

2. BRIEF REVIEW OF PARALLEL SORTING 2

3. OUR OBJECTIVES 3

4. THE BITONIC SORT 3

   4.1 An Improvement 4
   4.2 Proof of Correctness of the Improved Bitonic Sort 5
   4.3 The Parallel Sorting Algorithm 7

      4.3.1 Illustration - Example with Actual Records 7
      4.3.2 Illustration of the Steps in the Algorithm 11
      4.3.3 The Algorithm 11

5. INTERCONNECTIONS OF PROCESSORS 16

6. ANALYSIS OF TIME COMPLEXITY 19

7. PROCESSOR UTILIZATION 20

8. CONCLUSIONS 21

9. REFERENCES 21

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
DISTRIBUTION/AVAILABILITY CODES
and/or SPECIAL
A

PREFACE

This work was supported by Contract N00014-75-C-0573, from the Office of Naval Research and supervised by Dr. David K. Hsiao, Professor of Computer and Information Science, The Ohio State University.

The contract was administrated and monitored by The Ohio State University Research Foundation.

1. INTRODUCTION

DBC is a database computer for very large databases [1,2]. A major function of DBC is to carry out rapid content-search cost-effectively, since a database system (which can be supported on DBC) makes use of this operation quite heavily. By using large mass memory (MM) blocks (disk cylinders) and by incorporating tracks-in-parallel readout mechanisms, every one of the blocks can be searched individually by means of a set of parallel microprocessors. Consequently, large amounts of data can be examined rapidly. By making use of the clustering mechanism of DBC's controller (DBCCP) and of the directories stored in the structure memory (SM), it is ensured that only a few blocks need be searched in order to answer a query thereby achieving higher access precision in response to user requests.

The order in which data are presented to the user is often of some concern. In conventional machines, the process of ordering records, if the ordering is to be based on data item values, is rather slow. However, the speed of this process is generally compatible with the speed at which records are searched which is also slow. The use of DBC greatly speeds up the search process. In order to maintain comparable ordering speed so that the record-ordering process does not cause a bottleneck, it is necessary that DBC be provided with a fast hardware sorter.

The post processor (PP) of DBC has the capability of sorting records, as well as carrying out other post-processing functions, such as computing set functions (max, min, average, count, sum) and performing the equality-join operation. Once the response set of a user query has been identified by the mass memory, this set of records can be sorted by the post processor. Instead of sorting all records retrieved by the mass memory, only those that are meaningful to the user are actually sorted. Sorting is, therefore, incorporated as a post-processing function rather than as a function of the mass memory.

Thus, sorting of records that have been retrieved in response to a particular query can be performed in conjunction with the evaluation of another query by the mass memory. Furthermore, the mass memory remains simple, since it does not involve with the ordering of retrieved records.

2. BRIEF REVIEW OF PARALLEL SORTING

Certain sorting methods (such as distributive partitioning [3]) are available which use a single processor to sort N elements in an expected time of O(N). Even though these methods are as fast as any sorting method can be, they rely on the use of random access memory and an amount of memory which is O(log N), instead of O(N). (All logarithms, as used in this report, are with respect to the base of 2.) Furthermore, depending on the distribution of values of the sort attribute, the worst case time for sorting is O(N log N).

Sorting methods that do not rely on the use of random access memory are also available. Some of these methods make use cf a single processor and a linear amount of sequential memory, i.e., O(N) memory, to sort N elements in O(N log N) time. One of these methods is an adaptation of the merge-sort method of Knuth [4, pp. 159-168].

With the use of P parallel processors and a linear amount of sequential memory, our aim is to achieve a sorting time of O((N log N)/P) for sorting N elements. This is because a single processor can do the task in O(N log N) time, and it may be expected that P processors should do the task P times faster. Unfortunately, there is no known way to partition the problem into P parts so that each of the P processors can work on a single part and execute it in O((N log N)/P) time.

There exist parallel sorting methods (such as the rebound sorter [5]) which use P relatively simple processors (comparators) to sort P elements in O(P) time. There also exist parallel sorting methods (such as the ones in [6] and

[7]) which use P processors to sort P elements in $O(\sqrt{P} + \log^2 P)$ time. The latter methods use an Illiac IV type of two-dimensional structure of processors. Another method by Stone [8] uses P processors to sort P elements in $O(\log^2 P)$ time. Even though these methods are quite fast, they suffer from the fact that groups of records larger than P in number will have to be sorted in separate batches of P records and then merged. Other methods reported in the literature also have the same problem.

3. OUR OBJECTIVE

We shall propose a method in which P processors can sort N elements (where $N \geq P$) in $O((N/P)\log(N/P) + (N/P)\log^2 P)$ time. In other words, P processors will sort P elements in $O(\log^2 P)$ time. Besides, more than P records can be sorted as a single batch. In fact, if each processor has enough memory to hold M records, then as many as MP records can be sorted as a single batch. Also note that when $N >> P$ such that $\log N > \log^2 P$, then the sorting time approaches $O((N/P)\log(N/P))$ which is, of course, the best that can be expected from P processors. The method uses P processors, each with some sequential memory and each processor connected directly to log P other processors (say, by using routing registers as in Illiac IV).

4. THE BITONIC SORT

Our sorting method is a variation of the bitonic sorting method originally proposed by Batcher [9]. Batcher's bitonic sorting method is based on the concept of <u>bitonic</u> <u>sequence</u>. A sequence S = (s1,s2,...sN) is bitonic if there is an index k, where $1 \leq k \leq N$, such that either

(i) $s1 \leq s2 \leq \ldots \leq sk \geq s(k+1) \geq \ldots \geq sN$ or

(ii) $s1 \geq s2 \geq \ldots \geq sk \leq s(k+1) \leq \ldots \leq sN$

If a sequence S = (s1,s2,...,sN) is bitonic then so are the sequences S'odd = (s1,s3,s5,...) and S'even = (s2,s4,s6,...). It can be shown [9] that to sort a

bitonic sequence S = (s1,s2,...,sN), it is only necessary to recursively sort the two bitonic subsequences S'odd and S'even separately and then merge the two resultant sequences by simply carrying out an element by element comparison. If S'odd is sorted to produce (x1,x3,x5,...) and S'even is sorted to produce (x2,x4,x6,...), then the sorted version of the original sequence S is simply

(min(x1,x2),max(x1,x2),min(x3,x4),max(x3,x4),...).

4.1 An Improvement

Our method of sorting, as we have already mentioned, is a variation of Batcher's bitonic sorting method. This method sorts N elements, where N is a power of 2. The method is based on the fact that if a sequence S = (s1,s2,...,sN) is bitonic, then so are the sequences

S'small = (min(s1,s[N/2+1]),min(s2,s[N/2+2]),
...,min(s[N/2],sN)) and

S'large = (max(s1,s[N/2+1]),max(s2,s[N/2+2]),
...,max(s[N/2],sN))

Furthermore, every element of S'small is no larger than any element of S'large. Thus, it is only necessary to sort the bitonic subsequences S'small and S'large separately and then concatenate them (instead of merge them) in order to get the sorted version of the given sequence S. In other words, to sort a bitonic sequence S = (s1,s2,...,sN), we compare the two halves of the sequence, element by element, put the smaller elements in one subsequence and the larger ones in another, sort these two bitonic subsequences, and finally concatenate the two sorted subsequences. This idea may be used in sorting fixed numbers of records in a mesh-connected computer [6]. But we propose to use this improved variation of the bitonic sorting method in order to parallel sort arbitrary numbers of records with P processors having log P interconnections.

4.2 Proof of Correctness of the Improved Bitonic Sort

To prove correctness of the improved bitonic sorting method we first present a definition and a lemma:

Definition: Given an ordered pair of arbitrary numbers <n1, n2>, a *compare-and-interchange* operation compares the numbers n1 and n2 and permutes them into an ordered pair

(i) <n1, n2>, if n1 $\leq$ n2

(ii) <n2, n1>, if n1 > n2.

Lemma: Suppose there is an algorithm that sorts any arbitrary sequence of 0s and 1s into a non-decreasing order by simply performing compare-and-interchange operations. Then the algorithm will sort any arbitrary sequence of arbitrary numbers into a non-decreasing order. The proof of the lemma (stated in terms of sorting networks) appears in [4, pp. 224].

Making use of the above lemma, we have only to show that, given a bitonic sequence S = (s1,s2,...,sN) of 0s and 1s, the two subsequences

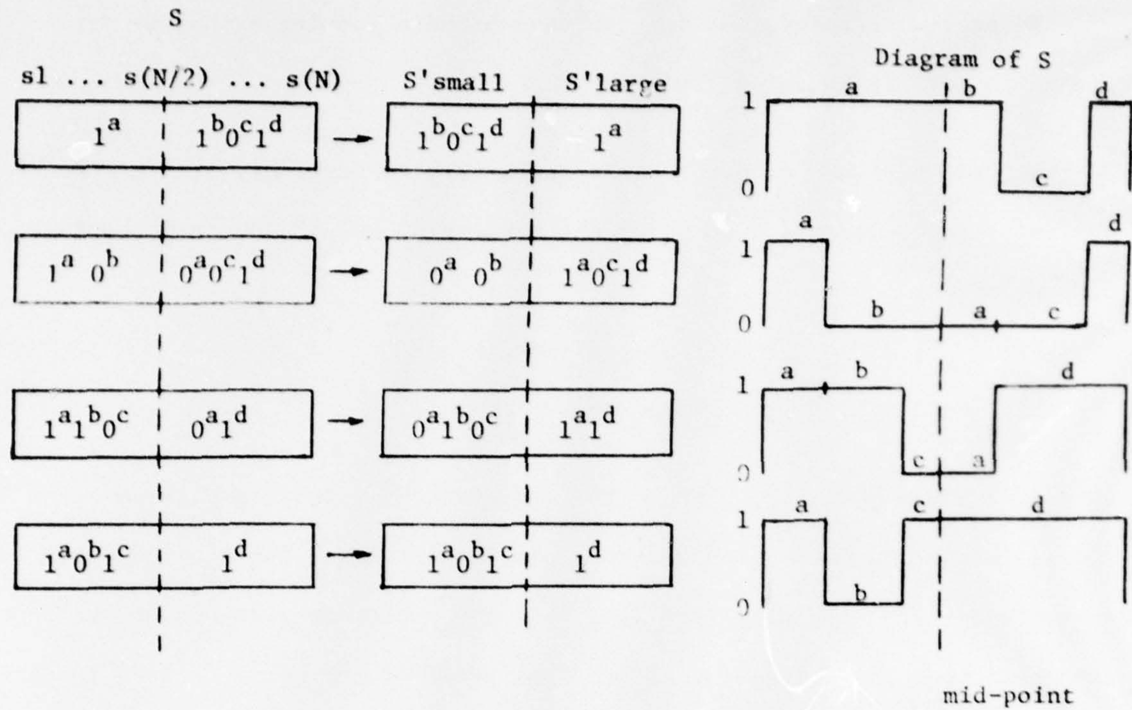
S'small = (min(s1,s[N/2+1]),min(s2,s[N/2+2]),
...,min(s[N/2],sN)) and

S'large = (max(s1,s[N/2+1]),max(s2,s[N/2+2]),
...,max(s[N/2],sN))

are bitonic, and that every element of S'small is no larger than any element of S'large.

The proof considers two cases: (i) the given sequence S consists of a sequence of 1s, followed by a sequence of 0s, followed by a sequence of 1s; (ii) the given sequence S consists of a sequence of 0s, followed by a sequence of 1s, followed by a seuqence of 0s. The two cases are proved graphically and shown separately in Figure 1. In the figure, a sequence of x 1s or x 0s is represented as 1^x or 0^x, respectively. For each of the two cases, there are

CASE 1: S is of the form $1^x0^y1^z$; $x,y,z \geq 0$



CASE 2: S is of the form $0^x1^y0^z$; $x,y,z \geq 0$

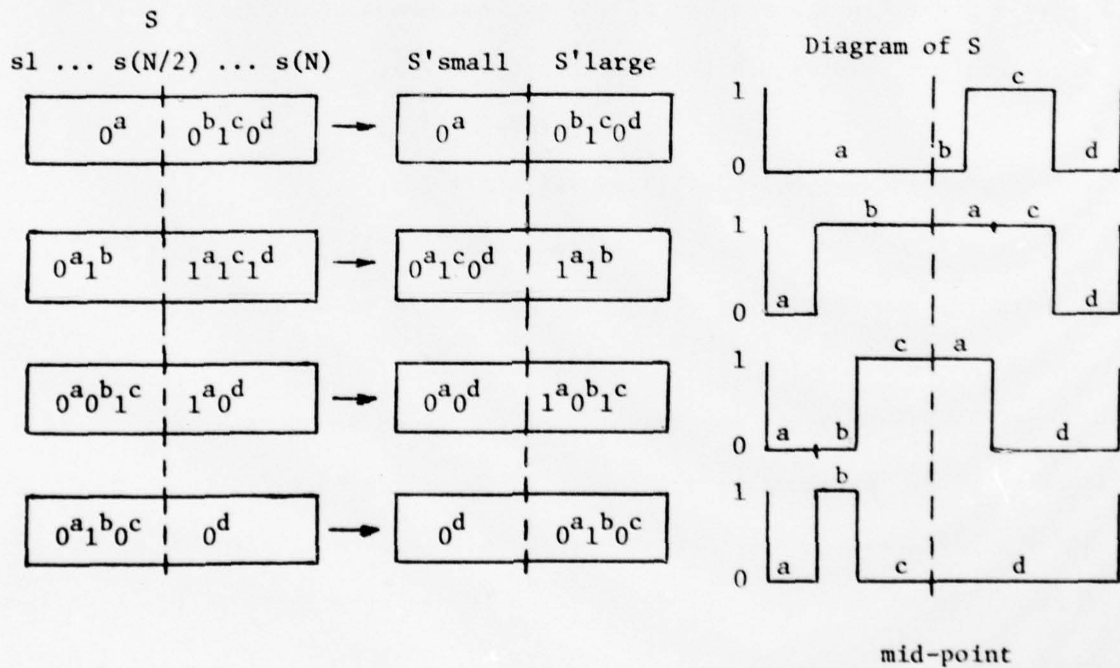


FIGURE 1. Proof of Correctness of Improved Bitonic Sort

four subcases depending on the position of the mid-point of the given sequence S. Each subcase corresponds to a row in the figure. The sequence S in each subcase is diagrammed in the rightmost column of the figure, where 1s are represented at a higher level than 0s. We notice that, in every possible subcase, S'small and S'large are bitonic and that every element of S'small is no larger than any element of S'large.

4.3 The Parallel Sorting Algorithm

In our algorithm using P parallel processors, we assume that each of the P processors has enough memory to accommodate M records. Altogether there are MP records. P is a power of 2, but M can be any positive number, not necessarily a power of 2. The processors are numbered 0 through (P-1) and the M records in processor i, for $0 \le i \le P-1$, are named R[i,1], R[i,2], ..., R[i,M], respectively. Figure 2 depicts the record indexing scheme.

4.3.1 Illustration - Example with Actual Records

The manner in which the P parallel processors sort MP records is illustrated by means of an example. We have P=4 and M=5. The initial configuration of records (represented by sort values alone) is shown in the left hand side of Figure 3. (1 + log P) steps are involved in the sorting process. Step 0 involves one pass over all the records, Step 1 involves two passes, etc.

Step 0 (as depicted in Figure 3)

Pass 1. Each processor sorts its own records. The odd-numbered processors sort in non-decreasing order, the even-numbered processors sort in non-increasing order.

Step 1 (as depicted in Figure 4)

In this step, there are 2 bitonic sequences in the two pairs of processors (0,1) and (2,3), respectively. We are to sort each of these bitonic sequences.

Records

Processor Numbers						
0	R[0,1]	R[0,2]	.	.	.	R[0,M]
1	R[1,1]	R[1,2]	.	.	.	R[1,M]
2	R[2,1]	R[2,2]	.	.	.	R[2,M]
			⋮			
P-1	R[P-1,1]	R[P-1,2]	.	.	.	R[P-1,M]

FIGURE 2. Record Indexing Scheme When There Are P Processors and M Records Per Processor

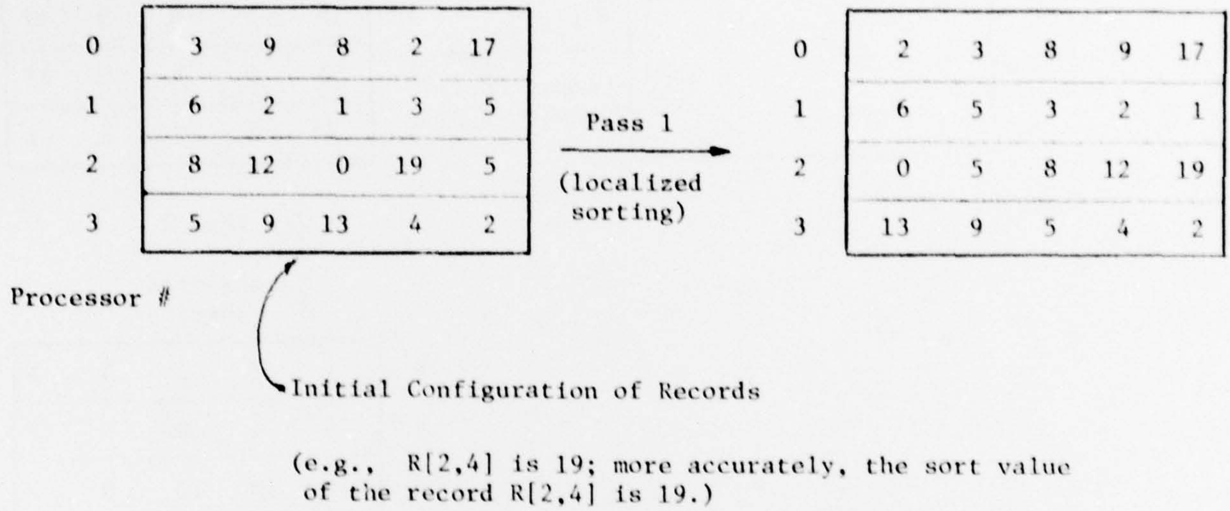


FIGURE 3. Example - Step 0

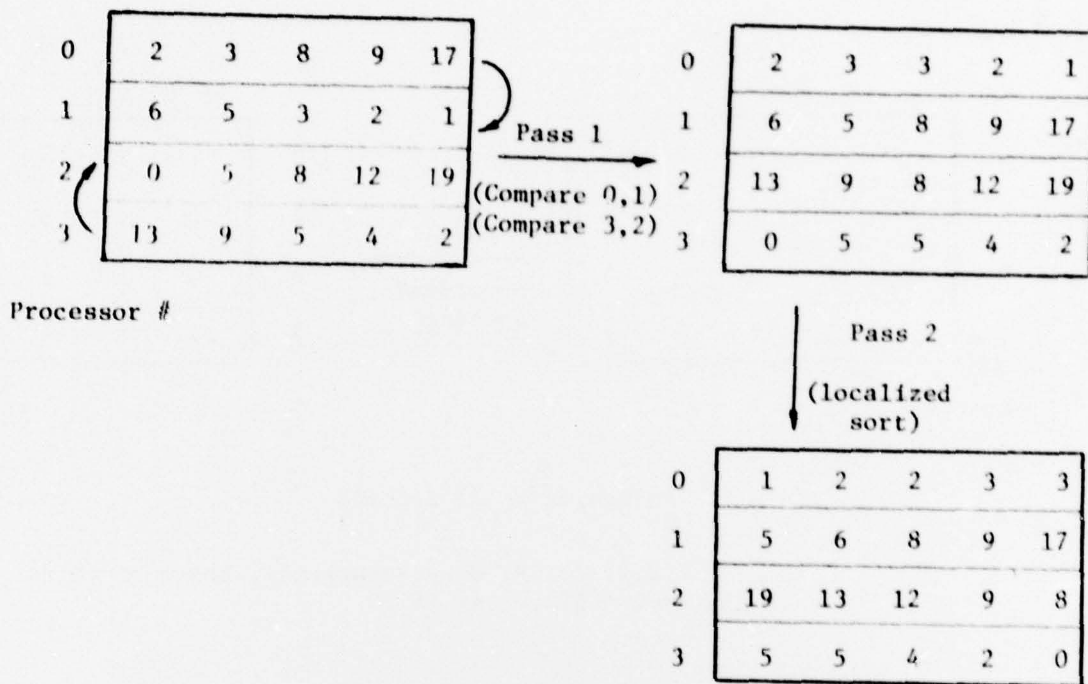


FIGURE 4. Example - Step 1

Pass 1. Processors 0 and 1 compare records (R[0,1] with R[1,1], R[0,2] with R[1,2], etc.); the smaller records (i.e., those having the smaller sort values) are placed in processor 0 and the larger ones in processor 1. Processors 3 and 2 (in this order) do the same.

Pass 2. Each processor sorts its own records. Processors 0 and 1 sort in non-decreasing order while processors 2 and 3 sort in non-increasing order.

Step 2 (as depicted in Figure 5)

In this step there is 1 bitonic sequence in the quadruple of processors (0,1,2,3). We are to sort this bitonic sequence.

Pass 1. Processors 0 and 2 compare records (R[0,1] with R[2,1], R[0,2] with R[2,2], etc.); the smaller records are placed in processor 0 and the larger ones in processor 2. Processors 1 and 3 (in this order) do the same.

Pass 2. Processors 0 and 1 compare records; the smaller records are placed in processor 0 and the larger ones in processor 1. Processors 2 and 3 do the same.

Pass 3. Each processor sorts its own records in non-decreasing order.

The final configuration of records is as shown in Figure 5.

4.3.2 Illustration of the Steps in the Algorithm

The algorithm for parallel sorting, using the improved bitonic sorting method, is illustrated in Figure 6. There are 16 processors, so that (1 + log 16) or 5 steps are involved. In each step there can be several passes over the records, the last of which involves localized sorting (where each processor sorts its own records). Localized sorting is indicated by a horizontal arrow, directed right for non-decreasing order and directed left for non-increasing order. A curved arrow from processor i to processor j (directed toward processor j) indicates that i and j must perform record by record comparisons, store the smaller ones in i and the larger ones in j.

4.3.3 The Algorithm

In all the procedures that constitute the parallel sorting algorithm, P is the number of processors, M is the number of records per processor and

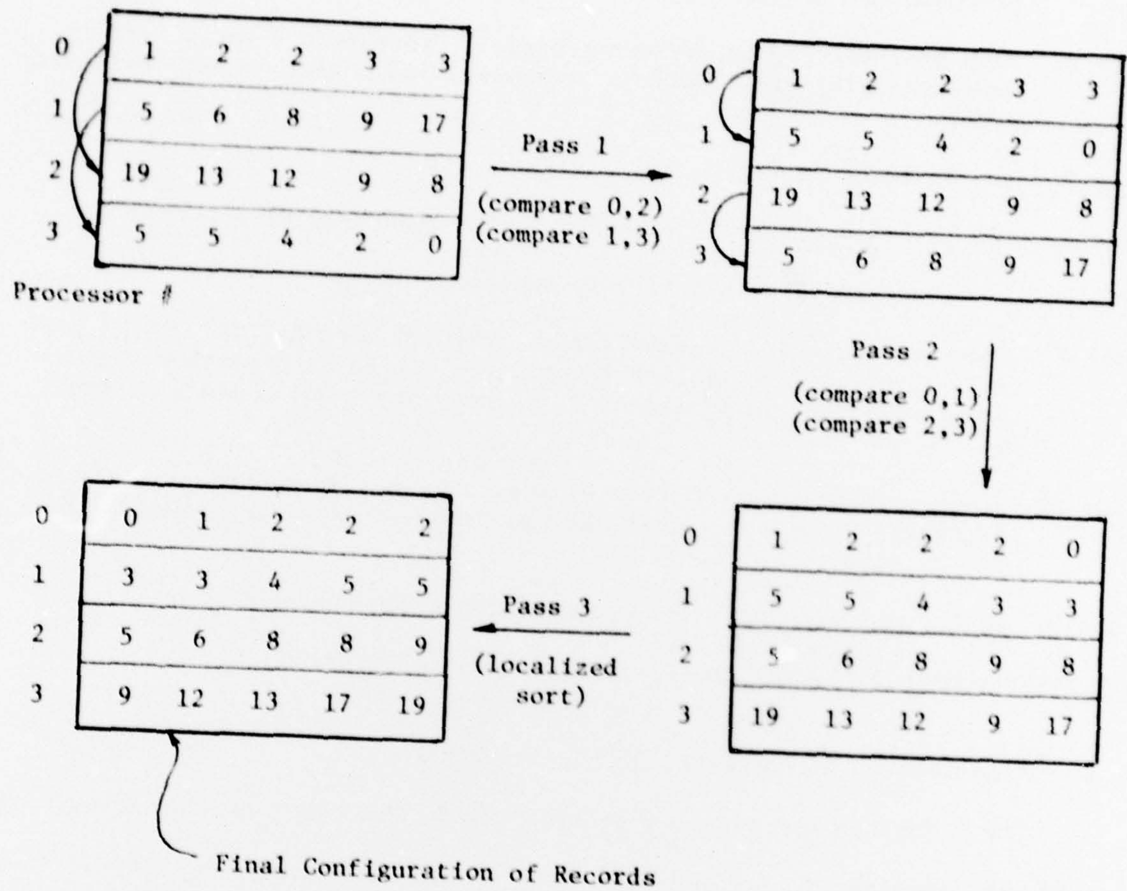


FIGURE 5. Example - Step 2

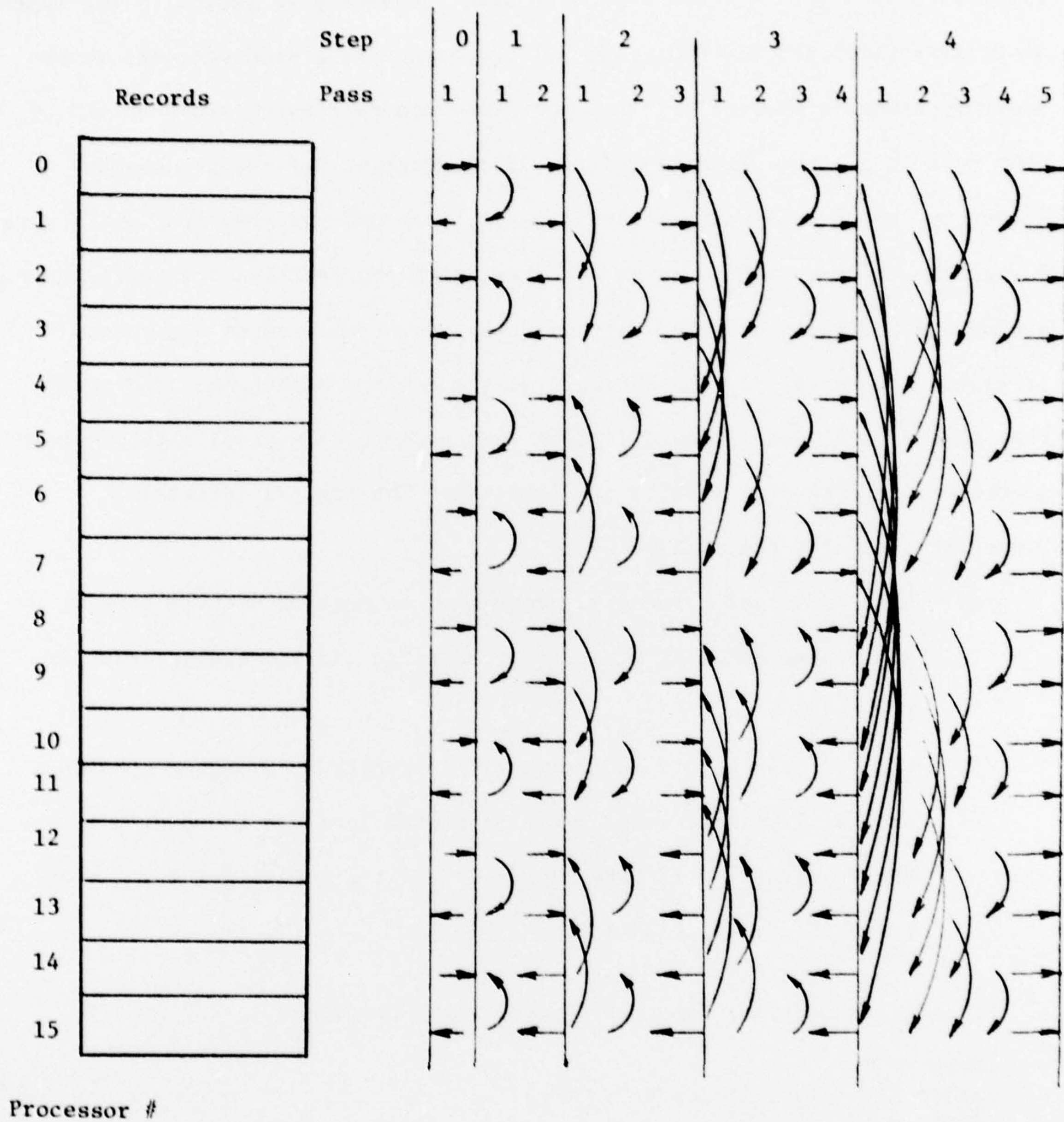


FIGURE 6. Illustration of the Improved Bitonic Sorting Method Using Parallel Processors

Step is a variable for the step number. A proc-set, defined with respect to any particular pass in some step of the algorithm, is a group of consecutive processors such that any two proc-sets have the same size (which is the number of processors in the proc-set) and all processors in a proc-set sort in the same direction (either all of them sort in a non-decreasing order or all of them sort in a non-increasing order). Each proc-set has two processors during the second-to-last pass in any step, each proc-set has four processors during the third-to-last pass in any step, each proc-set has eight processors during the fourth-to-last pass in any step, etc. The size of a proc-set is always a power of 2. The variable Diff (meaning, difference) indicates that processor i must interact (during some pass in some step) with processor j, where j is either (i + Diff) or (i- Diff). The logical variable Direction indicates the following:

(i) When localized sorting is necessary, it must be done in non-decreasing order if Direction = 0 and in non-increasing order if Direction = 1.

(ii) When processor i is to interact with processor j, those records with smaller sort values must be stored in i and those with bigger ones in j if Direction = 0, and the roles of i and j are reversed if Direction = 1.

A. Procedure Executed by the Post Processing Controller

```
Step := 0
Broadcast ('localized sort', Step)
Step := 1
while Step ≤ log P
do Diff := 2 ** Step
   while Diff ≠ 1
   do Diff := Diff/2
      Broadcast ('merge', Diff, Step)
   end
   Broadcast ('localized sort', Step)
   Step := Step + 1
end
```

The post processing controller coordinates the execution of the sorting algorithm by broadcasting, during every pass, a command ('merge' or 'localized sort') and the arguments (Diff or Step or both) to all the P processors. All the P processors can work in parallel to execute each command broadcasted by the controller.

B. Procedures Executed by a Processor (Processor i)

Processor i executes a 'merge' command (which has the arguments Step and Diff) as follows:

```
Proc-set-size := 2 * Diff
A := i mod Proc-set-size  /* Processor i  is the A-th one in its proc-set */
Direction := (Step+1)-th lowest significant bit of the number i (where the
             absolute lowest significant bit is the 1-th or first lowest
             significant bit)
if A < Diff then
   do j := i + Diff
      if Direction = 0 then Send(j) else Receive(j)
   end
else
   do j := i - Diff
      if Direction = 0 then Receive(j) else Send(j)
   end
```

The procedures Send and Receive are defined as follows:

```
Procedure Send(j)  /* as executed by processor i */
  Count := 1  /* count of records in processor i or j */
  while Count ≤ M
  do send the next record R[i,Count] to processor j
     wait for the return record from j
     call this record R[i,Count]
     Count := Count + 1
  end
end procedure Send

Procedure Receive(j)  /* as executed by processor i */
  Count := 1
  while Count ≤ M
  do wait for the next record R[j,Count] from processor j
     compare this record with own next record R[i,Count]
     if the sort value of R[i,Count] is smaller, then
           interchange R[i,Count] with R[j,Count]
     send R[j,Count] back to processor j
     Count := Count + 1
  end
end procedure Receive
```

Whenever processors i and j need to merge records, then the one which ought to keep the smaller records executes its Send procedure and the other processor executes its Receive procedure. If processor i ought to keep the smaller records, then it sends its records, one at a time, to j; processor j then keeps the bigger records (after record by record comparison) and sends the smaller ones back to processor i.

Finally, any processor i executes a 'localized sort' command issued by the controller with the argument Step, by doing the following:

```
Direction := (Step+1)-th lowest significant bit of number i
if Direction = 0 then sort all the M local records in
   non-decreasing order, else sort in non-increasing order
```

Localized sorting can be done in the 0-th step by using a merge sort method [4, pages 159-168]. Localized sorting during any other step involves sorting a bitonic sequence, and can be done by simply merging records starting at the two ends.

5. INTERCONNECTION OF PROCESSORS

It may be noted that this parallel sorting algorithm only requires a processor to interact (i.e., compare records) with exactly log P other processors, where P is the total number of processors. By providing direct interconnection of each processor with log P others, we can ensure that routing a record from one processor to another requires exactly one routing step (since the record does not have to go through any intermediate processor). For example, if P=8, we have the following interconnections:

Processor 0 is connected to Processors 1,2,4

Processor 1 is connected to Processors 0,3,5

Processor 2 is connected to Processors 0,3,6

Processor 3 is connected to Processors 1,2,7

Processor 4 is connected to Processors 0,5,6

Processor 5 is connected to Processors 1,4,7

Processor 6 is connected to Processors 2,4,7

Processor 7 is connected to Processors 3,5,6

The algorithm for finding the interconnections for any processor i during the time the post processor is initially designed is given as follows:

```
Proc-set-size := 2
while Proc-set-size < P   /* P is the total number of processors */
do A := i mod Proc-set-size   /* Processor i is the A-th processor in
                                 its proc-set */
   Diff := Proc-set-size/2
   if A < Diff then j := i + Diff else j := i - Diff
   connect processor i to processor j
   Proc-set-size := Proc-set-size * 2
end
```

The layout of processors and their interconnections are shown in Figure 7, for various values of P, the total number of processors. The actual arrangement (or positions) of P processors in a loop can be defined recursively as follows:

1) For P = 4, the arrangement of processors (in a loop) is as shown for this particular case in Figure 7.

2) For $P = 2^{n+1}$, where $n > 1$, start with the arrangement of processors for $P = 2^n$. Call this arrangement T1. From T1, create another loop of processors T2 by simply renumbering every processor i (of T1) as $i + 2^n$. Rotate T2 180 degrees about the horizontal axis. Place T2 underneath T1. Split T1 at the lowest point where the vertical axis intersects the loop (i.e., T1). Split T2 similarly but at the highest point. Connect the ends of T1 with those of T2 thus forming a new loop.

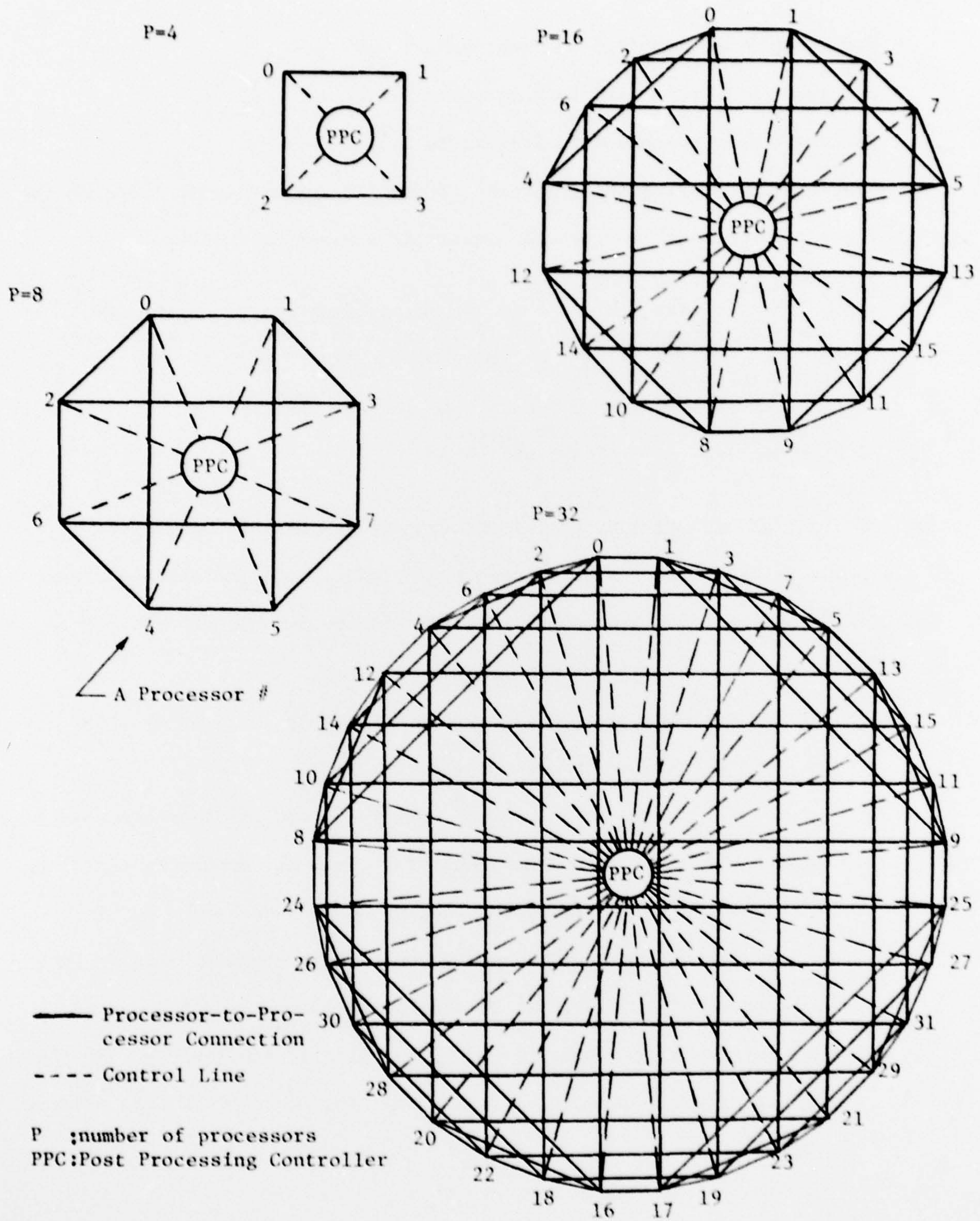


FIGURE 7. Interconnection of Processors for Post Processing

6. ANALYSIS OF TIME COMPLEXITY

Let r represent the amount of time required to route a single record from one processor to another. Let c denote the time required to compare (and interchange, if necessary) two records by the same processor. There are P processors and M records per processor.

The amount of time required by each processor to do localized sorting in the beginning (i.e., during Step 0) is given by

$$(M \log M)c$$

where we assume that M records can be sorted by a processor by using M log M comparison (and interchange) operations.

During all other steps (and there are log P other steps), localized sorting can be done by simply merging the records from the two ends. Every such sorting step requires M comparison (and interchange) operations. Hence, the total time for localized sorting after Step 0 is given by

$$(M \log P)c$$

A merge operation between two processors requires 2M record routing operations and M comparisons (and interchanges, if necessary). The 'merge' command is executed once in Step 1, twice in Step 2, etc. and log P times in the final step. Hence the total time for merging is

$$(2Mr + Mc)\ (1 + 2 + \ldots + \log P)$$

$$= (2Mr + Mc)\ (1 + \log P)\ \log P\ /\ 2$$

Thus, the total time for sorting is

$$Mc(\log M + \log P) + (2Mr+Mc)(1+\log P)\log P/2$$

Hence, the order of time is given by

$$O(M \log M + M \log^2 P)$$

7. PROCESSOR UTILIZATION

It may be expected that, in any given amount of time, P processors should perform P times as much work as can a single processor working alone. Unfortunately, due to the simultaneous need for the same resource by a number of processors, this expectation can rarely be materialized. It may be of interest, therefore, to quantify the utilization of processors, in terms of an efficiency measure. We define the <u>efficiency of a processor</u> (in the execution of a given task) as the ratio between the share H of the work actually performed by the processor in unit time and the work W it would have performed in unit time if it were acting alone.

In the context of sorting algorithms, let us say that a single processor, acting alone, can sort MP elements in $bMP \log(MP)$ units of time, where b is a constant. In that case, the processor, acting alone, can perform $1/(bMP \log(MP))$ amount of work in one unit of time. Therefore, if there are P processors working in parallel to do the same job, then the maximum amount of work any one of these P processors can be expected to perform in one unit of time is

$$W = 1/(bMP \log(MP))$$

With respect to our parallel sorting algorithm, we have noted earlier that the total time for sorting MP elements with the help of P parallel processors is given by

$$cM \log(MP) + dM \log^2 P$$

where c and d are constants. The reciprocal of this number is the amount of work performed by all the P processors together in one unit of time. Thus, the amount of work performed by any one of the P processors in unit time is given by

$$H = (1/P)\ (1/(cM \log(MP) + dM \log^2 P))$$

Hence, the efficiency E of a processor in the parallel sorting environment is

$$E = H/W$$
$$= (b \log(MP))/(c \log(MP) + d \log^2 P)$$

It may easily be observed that for a given P, E increases with M. If the constants b and c are equal, then the efficiency E approaches unity as M becomes larger and larger. This is one of the major advantages of our method over other parallel sorting methods. While the best previously known parallel sorting method can achieve a processor efficiency of at most $b/(c + d \log P)$ (by substituting 1 for M in the above efficiency formula), our method achieves a much better processor efficiency and utilization, since M can take up values greater than 1.

8. CONCLUSIONS

We have shown that by using P processors with sequential memory and log P interconnection among processors, it is possible to sort MP records in $O(M \log M + M \log^2 P)$ time. It may be noted that P is a power of 2, but M, the number of records assigned to each processor, can be any positive integer. The number of records that can be sorted in a batch is restricted only by the memory size of each processor and not by the number of processors. In fact, an efficiency measure is introduced which shows that processor utilization increases with increasing values of M.

9. REFERENCES

[1] Banerjee, J., Baum, R., and Hsiao, D. K., "Concepts and Capabilities of a Database Computer", ACM Transactions on Database Systems, 3, 4, (December 1978), pp. 347-384.

[2] Banerjee, J., Hsiao, D. K., and Kannan, K., "DBC - A Database Computer for Very Large Databases," IEEE Transactions on Computers, C-28, 6, (June 1979, to appear).

[3] Dobosiewicz, W., "Sorting by Distributive Partitioning," Information Processing Letters, Vol. 7, No. 1, January 1978, pp. 1-6.

[4] Knuth, D. E., Sorting and Searching, The Art of Computer Programming, Vol. 3, Addison-Wesley, Reading, Massachusetts, 1973.

[5] Chen, T. C., Lum, V. Y., and Tung, C., "The Rebound Sorter: An Efficient Sort Engine for Large Files," Proc. 4th International Conference on Very Large Data Bases, West Berlin, Germany, September 1978, pp. 312-318.

[6] Nassimi, D. and Sahni, S., "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Trans. on Computers, Vol. C-27, No. 1, January 1979, pp. 2-7.

[7] Thompson, C. D. and Kung, H. T., "Sorting on a Mesh-Connected Parallel Computer," Comm. ACM, Vol. 20, No. 4, April 1977, pp. 263-271.

[8] Stone, H. S., "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, Vol. C-20, 1971, pp. 153-161.

[9] Batcher, K. E., "Sorting Networks and Their Applications," Proc. AFIPS 1968 SJCC, Vol. 32, AFIPS Press, Montvale, N.J., pp. 307-314.

COMPUTER &
INFORMATION
SCIENCE
RESEARCH CENTER