RICE UNIVERSITY

An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and
its Application in Parallel Sorting Schemes

by

Manoj Kumar

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Master of Science

APPROVED, THESIS COMMITTEE :

A handwritten signature in cursive script, reading "D. S. Hirschberg", written over a horizontal line.

D. S. Hirschberg,
Assistant Professor of Computer Science in
the Department of Electrical Engineering,
Chairman

A handwritten signature in cursive script, reading "J. B. Sinclair", written over a horizontal line.

J. B. Sinclair,
Assistant Professor of Computer Science in
the Department of Electrical Engineering

A handwritten signature in cursive script, reading "J. R. Jump", written over a horizontal line.

J. R. Jump,
Professor of Computer Science in
the Department of Electrical Engineering

Houston, Texas

March, 1981

ABSTRACT

AN EFFICIENT IMPLEMENTATION OF BATCHER'S ODD-EVEN MERGE ALGORITHM AND ITS APPLICATION IN PARALLEL SORTING SCHEMES

by

Manoj Kumar

An algorithm is presented to merge two subfiles of size $n/2$ each, stored in the left and the right halves of a linearly-connected processor array, in $3n/2$ route steps and $\log n$ compare-exchange steps. This algorithm is extended to merge two horizontally adjacent subfiles of size $m \times n/2$ each, stored in an $m \times n$ mesh-connected processor array in row-major order, in $m + 2n$ route steps and $\log mn$ compare-exchange steps. These algorithms are faster than their counterparts proposed so far.

Next, an algorithm is presented to merge two vertically aligned subfiles, stored in a mesh-connected processor array in row-major order. Finally, a sorting scheme is proposed that requires $11n$ route steps and $2 \log^2 n$ compare-exchange steps to sort n^2 elements stored in an $n \times n$ mesh-connected processor array. The previous best sorting algorithm requires $14n$ route steps (for practical values of n , $4 \leq n \leq 512$).

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation and gratitude to Dr. D. S. Hirschberg, my thesis advisor for his able guidance. I also wish to thank Dr. J. R. Jump and Dr. J. B. Sinclair for serving on my thesis committee and their useful suggestions.

I am also grateful to my friends Daniel Dias, John F. Dooley and Balakrishna Iyer for the interesting discussions I had with them.

This research was supported by National Science Foundation grant MCS-80-03431 .

TABLE OF CONTENTS

1	Introduction	1
2	An efficient implementation of Batchers merge algorithm	9
3	Extension of Algorithm M to two dimensions	23
4	Vertical merge algorithm	37
5	Sorting algorithm	43
6	Extensions	45
7	Summary	49
8	References	51

1. INTRODUCTION

Batcher's odd-even merge and bitonic merge algorithms [Bt] have been popular among designers of parallel sorting and merge algorithms [NS1],[TK],[BS], perhaps because of their inherent parallelism. Sorting schemes using Batcher's merge algorithm require $n \log^2 n$ [†] fetch, compare and store steps to sort an array of size n on a SISD machine [F1]. On a SIMD machine [F1], if all processors are allowed to share a common memory (shared memory model), an array of size n can be sorted in $\log^2 n$ time using n processors.

A commonly used interconnection pattern for SIMD machines is the mesh connection [Si1],[Si2],[Br],[S1]. In this model, the processors are arranged in a 2-dimensional array $A[0:n-1;0:n-1]$. The processor at location $A[i,j]$ is connected to the processors at locations $A[i,j-1]$, $A[i-1,j]$, $A[i+1,j]$ and $A[i,j+1]$, provided they exist. Data may be transmitted from one processor to another only through this interconnection pattern. The processors connected directly by the interconnection pattern will be referred as neighbors. A processor can communicate with its neighbor with a route instruction which executes in t_r time. The processors also have a compare-exchange instruction which compares the contents of any two of each processor's internal registers and places the smaller of them in a specified register. This

[†] All logarithms in this paper are to the base 2.

For all the algorithms presented in this paper, the size of the input files is an integer power of 2.

instruction executes in t_c time.

ILLIAC IV has a similar architecture [Br]. The processor at location $A[i,j]$ in the array is connected to the processors at locations $A[WA(i,j-1)]$, $A[(i-1) \bmod n, j]$, $A[(i+1) \bmod n, j]$ and $A[WA(i,j+1)]$.

$WA(u,v)$ is a two input function, and its output is a pair of integers $\langle x,y \rangle$ defined as follows

```

if  $0 \leq v \leq n-1$  then
     $x = u$  ;  $y = v$ 
else if  $v = n$  then
     $x = (u+1) \bmod n$  ;  $y = 0$ 
else if  $v = -1$  then
     $x = (u-1) \bmod n$  ;  $y = n-1$ 

```

Thus processors at the boundaries of the array, which previously had some neighbors missing, now have all neighbors defined (In the SOLOMON computer, the connections for the undefined neighbors were used for input-output applications.)

There is a linear time lower bound for sorting using the ILLIAC IV interconnection pattern, because it requires at least $4n$ route steps to sort a file in which the smallest and the largest elements are on the wrong ends in the unsorted file. Thus, mesh-connected processors do not have the data routing capability required by Batcher's merge algorithm to sort in sublinear time.

Linearly-connected processor arrays are the building blocks of several machines with a higher dimensional interconnection pattern, such

as mesh-connected machines (two-dimensional interconnection pattern) and cube-connected machines [NS2]. In this interconnection pattern, the processors are logically arranged in a one-dimensional array and each processor can communicate with its two logical neighbors (if they exist). The simplicity of this interconnection pattern makes it easier to implement parallel algorithms which can be later generalized to higher dimensions. This approach has been adopted in the present paper.

Nassimi and Sahni have implemented a sorting scheme on a mesh-connected computer [NS1], which makes use of Batcher's bitonic merge algorithm. Their algorithm requires $\approx 14 n$ route steps and $\approx 2 \log^2(n)$ compare-exchange steps to sort a two-dimensional array of size $n \times n$. However, the merge algorithms proposed by them require one of the input subfiles being merged to be sorted in nondecreasing order and the other in nonincreasing order. H. T. Kung and C. D. Thompson have made use of Batcher's odd-even merge algorithm to implement a sorting scheme for mesh-connected computers which requires $\approx 6n + O(n^{2/3} \log n)$ route steps and $n + O(n^{2/3} \log n)$ compare-exchange steps asymptotically. Since $n^{2/3} \log n > n$ for most practical values of n ($4 \leq n \leq 512$), the linear bound stated by Thompson and Kung has a larger coefficient for practical values of n . The sorted output produced by Thompson and Kung's algorithm is in snake-like row-major order [NS1, TK].

H. S. Stone has used an interconnection pattern called the Perfect Shuffle [St]. With this interconnection pattern, Batcher's odd-even merge algorithm can be implemented in $O(\log n)$ steps. However, processor arrays with this interconnection pattern have not been built

so far.

We will give an implementation of Batchner's odd-even merge algorithm for a linearly-connected processor array of n processors. Our algorithm merges two sorted subfiles of size $n/2$, placed in the left and the right halves of the processor array, in $3n/2$ route steps and $\log n$ compare-exchange steps. Next we will generalize this algorithm to merge two sorted subfiles of size $m^*n/2$, which are placed in the left and right halves of an $m \times n$ mesh-connected processor array in row-major order. The merged output is in row-major order and the algorithm requires $m+2n$ route steps and $\log m + \log n$ compare-exchange steps. Finally, we will make use of our merge algorithm to sort a file of size m^*n on the mesh-connected processor array producing output in row-major order.

The time complexity of the two-dimensional merge algorithm proposed in this paper compares favorably with its counterparts used by Nassimi and Sahni ($2m+2n$ route steps and $\log mn$ compare-exchange steps) and Thompson and Kung ($\approx 6n + O(n^{2/3} \log n)$ route steps and $n + O(n^{2/3} \log n)$ compare-exchange steps, for $m=n$). It is interesting to note that Kung and Thompson's algorithm requires the same order of time for merging two subfiles of size $m \times n/2$ each, and for sorting m^*n elements organized as an $m \times n$ matrix.

Linearly-connected processor array

A linearly-connected processor array of size n is a SIMD machine [Fl] consisting of n identical processors. Each processor has the following characteristics :

- 1 Each processor is connected to both of its neighbors in the array, provided they exist.
- 2 Each processor has two internal registers, the A (for accept) and the R (for reject) register.
- 3 Each processor is capable of executing the following instructions
 - 3.1 The compare-exchange instruction compares the contents of a processor's two internal registers and places the smaller of them in the R register and the other one in the A register. This instruction takes t_c time to execute.
 - 3.2 The route instruction allows a processor to copy the contents of a neighbor's R register into its own R register. All processors executing this instruction (simultaneously) copy the contents of either their left neighbor's or their right neighbor's R register. This instruction requires t_r time to execute.
 - 3.3 The exchange instruction allows the processor to swap the contents of its A and R-registers. This instruction requires t_e time to execute.
- 4 The processors execute the instructions broadcast by a common controller. However, by using the address masking scheme [Sil], a

set of processors can be prevented from executing the broadcast instruction.

Address masking scheme

The address masking scheme uses an m -position mask ($m = \log n$) to specify which processors are to be activated, each position in the mask corresponding to a bit position in the address of the processors. Each position in the mask will contain either a 0, 1, or X (Don't care). The only processors that will be activated are those whose address matches the mask.

Mesh-connected processor array

A mesh-connected processor array of size $m \times n$ is a SIMD machine consisting of $m \times n$ identical processors, each of which has the following characteristics :

- 1 Each processor is connected to its two horizontal and two vertical neighbors. The wrap-around connections (described earlier) are present for the end-processors.
- 2 Each processor has three internal registers referred as A, R and T.
- 3 The computational capability of each processor is similar to that described for the linear-processor array. Additionally, each processor can exchange the contents of any two of its registers and copy the contents of any register into either of the remaining ones, in t_e time.

- 4 The decoding and execution of the instruction stream is identical to the scheme for the linear-processor array.

Similar models of computation have been used by Kung and Thompson [TK] and Nassimi and Sahni [NS1].

Batcher's odd-even merge algorithm

The odd-even merge algorithm to merge S and T , two sorted lists of sizes s and t , to produce a sorted list of size $s+t$ can be defined recursively in the following way :

- 1 If s and t are both 1, then compare the two elements and interchange their positions if they are out of order.
- 2 Else
 - 2.1 Split the lists S and T into their odd-indexed elements s_o, t_o and their even-indexed elements s_e, t_e .
 - 2.2 Recursively, merge the sublists of odd-indexed elements (s_o and t_o) to obtain list m_o . Recursively, merge the even-indexed elements (s_e and t_e) to obtain list m_e .
 - 2.3 For $1 \leq i \leq (s+t-1)/2$, compare the i^{th} element of m_e with the $(i+1)^{\text{th}}$ element of m_o and interchange their positions if they are out of order.

Example of Batcher's odd-even merge algorithm

Let $s = 1, 4, 7, 8$; and $t = 2, 3, 5, 9$; be the two sorted subfiles to be merged. The odd-indexed elements of the two subfiles are $s_o = 1, 7$; and $t_o = 2, 5$. The even-indexed elements are $s_e = 4, 8$; and $t_e = 3, 9$. Merging the odd-indexed elements and the even-indexed elements separately gives the sorted files $m_o = 1, 2, 5, 7$; and $m_e = 3, 4, 8, 9$. Now we combine m_o and m_e into a single file whose odd-indexed elements are the file m_o and whose even-indexed elements are the file m_e . The elements in this file are 1, 3, 2, 4, 5, 8, 7, 9. In this file we compare the 1st, 2nd and 3rd element of m_o with the 2nd, 3rd and 4th element of m_e and interchange the two if they are out of order. This gives us the sorted file 1, 2, 3, 4, 5, 7, 8, 9.

The proof of correctness of this algorithm was given by Batcher [Bt].

2. AN EFFICIENT IMPLEMENTATION OF BATCHER'S MERGE ALGORITHM

To implement Batcher's odd-even merge algorithm on our model of computation we will use the operations defined below. An argument in the form of a capital letter (X) indicates a subset of $0:n-1$ which can be a single value (x) or one or more ranges of values (x:y).

EXCHANGE[X]

The processors $P[X]$ (if X is x:y then we mean $P[x]$, $P[x+1]$, ..., $P[y]$) interchange the contents of their A and R-registers. Since, only one exchange instruction is required to complete this operation, it requires t_e time.

MOVE[j , X]

If j is a non-zero positive (negative) integer and $X = x_1:x_2$, processors $P[x_1-i:x_2-i]$, for $i = 1, 2, \dots, j$ ($i = -1, -2, \dots, -j$), if they exist, copy the contents of their right (left) neighbor's R-register into their own R-register. This step is repeated $|j|$ times for the $|j|$ different values of i. The net result of this operation is to move the contents of the R-registers of $P[x_1:x_2]$ to the R-registers of $P[x_1-j:x_2-j]$ ($P[x_1+j:x_2+j]$), provided they exist. When the second argument of the MOVE operation (X) is unspecified, the default is the set of all processors. Since $|j|$ route instructions are required to complete this operation, it will take $|j|*t_r$ time to perform this operation.

COMPARELO[X]

The processors $P[X]$ compare the contents of their A- and R-registers and if the contents of the A-register are greater than the contents of the R-register, the two are interchanged. Thus, after a COMPARELO instruction, the contents of the A- (accepting) register are smaller than the contents of the R-register. Only one compare-exchange instruction is needed to perform this operation. Therefore, it requires t_c time.

COMPAREHI[X]

The processors $P[X]$ compare the contents of their A- and R-registers and if the contents of the R-register are greater than the contents of the A-register, the two are interchanged. After a COMPAREHI instruction, the contents of the A-register are greater than the contents of the R-register. This operation also requires t_c time.

UNFOLD[X]

If $X = x:y$ then for all w ($x \leq w \leq y$), the contents of the A-register of $P[w]$ are copied into the A-register of $P[2w-x]$, and the contents of the R-register of $P[w]$ are copied into the A-register of $P[2w-x+1]$, if these processors exist. This operation moves the $2(y-x+1)$ elements, stored in the A- and R-registers of processors $P[X]$ in column-major order, into the A-registers of processors $P[x:2y-x+1]$. The unfold operation can be completed in $(y-x+1) * t_r$ time. The

implementation of this operation depends on the relative speed of the route instruction and the compare-exchange instruction.

If the processors are loosely-coupled, i. e., the time required to broadcast the instruction from the common controller to each processor is much less than the time required to move data from one processor to another, then $t_r \gg t_e$. In this case most of the time t_r is spent in transferring the data from one processor to another. This situation occurs when each processor is fabricated on a single chip. Then, because of the pin limitations on the chip, we are forced to transfer data serially in small fractions of the word at a time. This makes the route instruction much more expensive than the exchange instruction.

For loosely-coupled processor arrays, the operation to unfold the elements stored in the A- and R-registers of the processors $P[x:y]$ can be implemented by the following algorithm

```

For (  $w = y + 1$  ;  $w > x + 1$  ;  $w = w - 1$  ) Do
{
    MOVE [-1,  $w:2y-w+2$ ]
    EXCHANGE [ $w-1$ ]
}
EXCHANGE [ $x:2y-x+1$ ]

```

Figure 1(a) illustrates the unfold operation for loosely-coupled processor-arrays.

If the processors are tightly-coupled, i. e., the time required to broadcast the instruction from the common controller to each processor

1	3	-	-
2	4	-	-

1	3	-	-
2	-	4	-

1	-	-	-
2	3	4	-

1	-	-	-
-	2	3	4

-	-	-	-
1	2	3	4

1	2	3	4
-	-	-	-

Figure 1(a)

1	3	-	-
2	4	-	-

1	-	3	-
2	-	4	-

1	-	3	-
-	2	-	4

1	2	3	4
-	-	-	-

Figure 1(b)

is much more than the time required to move data from one processor to another, then $t_r \approx t_e$. In this case most of the time t_r and t_e is spent in broadcasting the respective instruction. This situation occurs when all the processors are fabricated on a single chip, and all the bits of the data are transferred in parallel.

For tightly-coupled processor arrays we will include in our model of computation an extra instruction called route-double. The processor executing this instruction copies the contents of its left neighbors A and R-registers to its own A and R-registers. For linearly-connected processor arrays, the argument given with the route-double instruction specifies the set of processors which execute this instruction.

Though the amount of information (data) transferred in this instruction is twice that of route instruction, it does not require extra interconnections between the processors. The contents of the two registers are transferred serially. Since the time required to transfer the data between the two processors is much less than the time required for broadcasting the instruction, the time required to transfer the data a second time is a very small fraction of the time required by the route-double instruction. Hence the time required for a route-double instruction will be almost equal to the time required for a route instruction ($\approx t_r$).

For tightly-coupled processor arrays the unfold operation can be implemented by

For ($w = y + 1$; $w > x + 1$; $w = w - 1$) Do

```

{
    Route-Double [w:2y-w+2]
}
MOVE [-1]
EXCHANGE [ all even-indexed processors ]

```

Figure 1(b) illustrates the unfold operation for tightly-coupled processor-arrays.

It can be shown that the unfold operation requires $\approx k \cdot t_r + t_e$ time to unfold $2k$ elements, for both loosely-coupled and tightly-coupled processor arrays.

Algorithm M -- An efficient implementation of Batcher's merge algorithm

Initial data configuration

The two sorted arrays, $A[0:n/2-1]$ and $A[n/2:n-1]$, to be merged are stored in the A-registers of $P[0:n/2-1]$ and $P[n/2:n-1]$.

Final data configuration

The merged output will be placed in the A-registers of $P[0:n-1]$.

Algorithm M

Step 1 :

```

EXCHANGE [n/2:n-1]
MOVE [n/2]

```

Step 2 :

COMPARELO [0:n/2-1]

MOVE [-n/4]

Step 3 :

For ($x = n/4$; $x \geq 1$; $x = x/2$) Do

{

 COMPAREHI [x:n/2-1]

 MOVE [$\lceil x/2 \rceil$]

}

Step 4 :

UNFOLD [0:n/2 - 1]

Time complexity of Algorithm M

Step 1 of Algorithm M requires $(n/2)*t_r$ time for the MOVE operation and t_e time for the EXCHANGE operation. Step 2 requires $(n/4)*t_r$ time for the MOVE operation and t_c time for the COMPARE operation for a total of $t_c + (n/4)*t_r$ time. Step 3 is iterated $(\log n/4)+1 = (\log n)-1$ times and in each iteration a COMPARE operation is performed. During all the iterations of Step 3, data in the R-registers is moved a total of $n/4$ positions to the left. Thus, Step 3 requires $((\log n)-1)*t_c + (n/4)*t_r$ time to execute. Step 4 requires $(n/2)*t_r + t_e$ time.

Therefore, the total time required by Algorithm M is

$$(3n/2)*t_r + (\log n)*t_c + 2*t_e$$

Step 1 of the algorithm moves the two input subfiles to be merged, from the first and second halves of the set of A-registers, to the A and R-registers of the first half of the linearly-connected processor array (see Figures 1(c) and 1(d)).

Steps 2 and 3 carry out the compare-exchange instructions required by Batcher's odd-even merge algorithm. At the conclusion of Step 3 the merged output is in the form of a $2 \times n/2$ matrix where each column represents a processor, and the contents of the A and R-registers form the first and second row of the matrix. The sorted file is stored in column-major order (see Figure 1(e)).

Step 4 arranges the sorted file in the A-registers of the processor array (see Figure 1(f)).

At the beginning of Step 2, the first element in one of the input files can be $n/4$ positions to the left of its final position at the conclusion of Step 3. Similarly the last element of the other input file can be $n/4$ position to the right of its final position in Step 3. Hence, Steps 2 and 3 will require at least $(n/4 + n/4)$ route steps. Therefore, Steps 2 and 3 of the algorithm (where the merge process is actually carried out) are optimal in the number of route steps.

Steps 1 and 4 of the algorithm are not a part of the merge process. They allow us to have inputs and outputs in a format different from the

15	50	51	61	65	76	89	90	6	8	17	27	42	49	67	70
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Figure 1(c)

15	50	51	61	65	76	89	90	-	-	-	-	-	-	-	-
6	8	17	27	42	49	67	70	-	-	-	-	-	-	-	-

6	8	17	27	42	49	67	70	-	-	-	-	-	-	-	-
-	-	-	-	15	50	51	61	65	76	89	90	-	-	-	-

Figure 1(d)

one used by Steps 2 and 3. Though Step 2 is optimal, inclusion of Steps 1 and 4 increases the complexity of the algorithm by a factor of 1.5 .

In applications where the input subfiles to be merged are originally in the format accepted by Step 2, and the output format desired is the one produced by Step 3, Steps 1 and 4 of Algorithm M are not required and the merge process can be performed in time, $(n/2)*t_r + (\log n)*t_c$, which is optimal.

Correctness of Algorithm M

To show the correctness of Algorithm M, we will make use of the Zero-One Principle [Kn]

Theorem (Zero-One Principle)

If an algorithm sorts all sequences of zeros and ones into nondecreasing order, it will sort any arbitrary sequence of integers into nondecreasing order.

Since the two input files to be merged are already sorted, if the elements in them are from the set $\{0,1\}$ only, then each of them must be a sequence of zeros followed by a sequence of ones (see Figure 2(a)). Throughout the algorithm, temporary files in the A and R-registers will consist of a string of zeros followed by a string of ones.

0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	1
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Figure 2(a)

0	0	0	0	0	1	1	1	-	-	-	-	-	-	-	-
0	0	1	1	1	1	1	1	-	-	-	-	-	-	-	-

$\begin{matrix} \uparrow \\ Z \end{matrix}$
 $\begin{matrix} Y \\ \downarrow \end{matrix}$

0	0	0	0	0	1	1	1	-	-	-	-	-	-	-	-
-	-	-	-	0	0	1	1	1	1	1	1	-	-	-	-

Figure 2(b)

0	0	0	0	0	1	1	1	-	-	-	-	-	-	-	-
-	-	0	0	1	1	1	1	1	1	-	-	-	-	-	-

$\begin{matrix} Y \rightarrow \\ \searrow \end{matrix}$
 $\begin{matrix} \nearrow \\ Z \end{matrix}$

0	0	0	0	1	1	1	1	-	-	-	-	-	-	-	-
-	0	0	0	1	1	1	1	1	-	-	-	-	-	-	-

$\begin{matrix} Y \rightarrow \\ \searrow \end{matrix}$
 $\begin{matrix} \nearrow \\ Z \end{matrix}$

0	0	0	0	1	1	1	1	-	-	-	-	-	-	-	-
0	0	0	1	1	1	1	1	-	-	-	-	-	-	-	-

$\begin{matrix} \uparrow \\ Z \end{matrix}$

Figure 2(c)

0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Figure 2(d)

In Step 1 the two files are placed in the A and R-registers of the first $n/2$ processors. During Steps 2 and 3, let us denote the first processor position which has a 1 in its A-register by y and the first processor position which has a 1 in its R-register by z . After the COMPARELO operation in Step 2 has been performed, y will be at most $n/2$ positions to the left of z . Later in Step 2, the contents of the R-registers are moved $n/4$ positions to the left. Now y will be at most $n/4$ positions to the left or right of z (see Figure 2(b)).

When the For loop in Step 3 is entered, x denotes the maximum difference possible between y and z , which in the first iteration is $n/4$. After the execution of the COMPAREHI operation, y will be at most x positions to the left of z , and never to the right of z (see Figure 2(c)).

Later in Step 3, the contents of the R-registers are moved towards the left by a distance $x/2$ and the For loop condition is tested again with the value of x replaced by $x/2$. The last iteration of the For loop is executed for the value of $x=1$. After the COMPAREHI operation in this last iteration, either y and z will both refer to the same processor location or y will be one position to the left of z . So when the MOVE[1] operation in Step 3 is performed, either y will be one position to the right of z or both y and z will point to the same processor location. Both of these cases correspond to a sorted sequence of binary digits represented in column-major order.

Step 4 takes the sorted file represented in column-major order in the A and R-registers of the first $n/2$ processors and moves it to the A-registers of all the n processors in the array maintaining the sorted order (see Figure 2(d)).

3. EXTENSION OF ALGORITHM M TO TWO DIMENSIONS

The algorithm developed in the previous section required $O(n)$ time to merge two subfiles of $n/2$ records. Linear time was required because of the limited data routing capacity of linearly-connected processor arrays.

An obvious way to improve the merge time is to enhance the data routing capability of the processors. However, attaching an arbitrarily large number of input/output lines to a processor is unfeasible.

In this section we will give two merge algorithms, Horizontal Merge (Algorithm HM) and Vertical Merge (Algorithm VM), for mesh-connected processor arrays. The input to the Horizontal Merge algorithm consists of two sorted subfiles placed side by side (see Figure 3(a)). The input to the Vertical Merge algorithm consists of two sorted subfiles which are organized as a pile (see Figure 5(a)). Both of these algorithms are extensions of Algorithm M, developed in the previous section.

The operations EXCHANGE, UNFOLD and COMPAREHI are redefined for mesh-connected computers and two new operations, MOVEVERT and MOVEHORZ are defined below. In the following definitions $P[r,c]$ refers to the processor in row r and column c .

EXCHANGE[R,C]

When $R = r_1:r_2$ and $C = c_1:c_2$, the processors in the set $\{ P[r,c] \mid r_1 \leq r \leq r_2 \text{ and } c_1 \leq c \leq c_2 \}$ interchange the contents of their A and R-registers. This operation takes t_e time.

COMPAREHI[R,C]

When $R = r_1:r_2$ and $C = c_1:c_2$, the processors in the set $\{ P[r,c] \mid r_1 \leq r \leq r_2 \text{ and } c_1 \leq c \leq c_2 \}$ compare the contents of their A and R-registers and if the contents of the R-register are greater than the contents of the A-register, the two are interchanged. This operation requires t_c time.

MOVEVERT[j, R, C]

The argument R is a subset of the rows and C is a subset of the columns of the mesh-connected processor array. If j is a non-zero positive (negative) integer, processors $P[r-i,c]$ ($P[r+i,c]$) for all $r \in R$, $c \in C$, if they exist, and for i iteratively taking the values 1, 2, ..., j (-1, -2, ..., -j), copy the contents of the R-registers of processors logically below (above) into their own R-register. This step is repeated j times. This operation moves the contents of the R-registers of $P[r,c]$, where $r \in R$ and $c \in C$, to the R-registers of $P[r-j,c]$ ($P[r+j,c]$), provided they exist. Since $|j|$ route instructions are required to complete this operation, it will take $|j|*t_r$ time to perform

this operation.

MOVEHORZ[j , R, C]

Each row in the subset R of the rows of the mesh-connected processor array (of size $m \times n$) acts like a linearly-connected processor array of size n and performs the operation MOVE[j , C]. If the second and third arguments (R and C) are dropped, the default is the set of all processors. A total of |j| route steps and therefore $|j| * t_r$ time is required to complete this operation.

UNFOLD[R, C]

Each row in the subset R of the rows of the mesh-connected processor array (of size $m \times n$) acts like a linearly-connected processor array of size n and performs the operation UNFOLD[C]. The time required to complete this operation is $|C| * t_r + t_e$ (|C| is the number of columns in the subset C).

Algorithm HM -- Horizontal Merge

Initial data configuration

The two sorted arrays, $A[0:m*n/2-1]$ and $B[0:m*n/2-1]$, to be merged are stored in the processors $P[0:m-1, 0:n/2-1]$ and $P[0:m-1, n/2:n-1]$ in row-major order (see Figure 3(a)).

Final data configuration

The merged output of the two input arrays will be placed in processors $P[0:m-1, 0:n-1]$ in row-major order.

Algorithm HM

Step 1 :

EXCHANGE [all rows , $n/2:n-1$]

MOVEHORZ [$n/2$]

Merge the two subfiles in each column using Steps 2 and 3 of Algorithm M, considering each column as a linearly-connected processor array.

Step 2 :

MOVEHORZ [$-n/4$]

Create a copy of the contents of R-registers of $P[0:m-1, n/2:3n/4-1]$ in the R-registers of $P[0:m-1, 0:n/4-1]$, and a copy of the contents of the A-registers of $P[0:m-1, 0:n/4-1]$ in the A-registers of $P[0:m-1, n/2:3n/4-1]$. (see Figure 3(b))

We note that Step 2 can be carried out using $3n/4$ route instructions as illustrated in Figure 3(c). The necessity of the T-registers is obvious here. Figure 3(c) also shows the data configuration after the execution of Step 2.

Step 3 :

For ($x = n/4$; $x \geq 1$; $x = x/2$) Do
 {

```

MOVEVERT [-1 , all rows , 0:x-1 and n/2:n/2+x-1]
COMPAREHI [ all rows, all columns]
MOVEVERT [ 1 , all rows , 0:x-1 and n/2:n/2+x-1]
MOVEHORZ [  $\lceil x/2 \rceil$  ]
}

```

Step 4 :

```
UNFOLD [ all rows, 0:n/2 - 1]
```

Figures 3(d) and 3(e) illustrate the Horizontal-Merge algorithm for $m = 2$ and $n = 8$.

Time complexity of Algorithm HM

Step 1 of the algorithm requires $n/2$ route instructions and an exchange instruction to move data to the left $n/2$ columns, and m route instructions and $\log m$ compare instructions to merge the subfiles in each column. The total time required by Step 1 is thus

$(n/2+m)*t_r + (\log m)*t_c + t_e$. From Figure 3(c) it is easy to see that

Step 2 requires $(3n/4)*t_r + 7*t_e$ time to execute.

The For loop in Step 3 is iterated $\log n$ times. In each iteration of the For loop, one COMPARE operation is performed. The total number of route instructions performed over all the iterations of the For loop is $n/4$. Hence Step 3 requires $(\log n)*t_c + (n/4)*t_r$ time to execute.

Step 4 requires $(n/2)*t_r + t_e$ time.

A ₀	A ₁	A ₂	A ₃	B ₀	B ₁	B ₂	B ₃
-	-	-	-	-	-	-	-
A ₄	A ₅	A ₆	A ₇	B ₄	B ₅	B ₆	B ₇
-	-	-	-	-	-	-	-
A ₈	A ₉	A ₁₀	A ₁₁	B ₈	B ₉	B ₁₀	B ₁₁
-	-	-	-	-	-	-	-
A ₁₂	A ₁₃	A ₁₄	A ₁₅	B ₁₂	B ₁₃	B ₁₄	B ₁₅
-	-	-	-	-	-	-	-

Figure 3(a)

A ₀	A ₁	A ₂	A ₃	A ₀	A ₁	-	-
B ₂	B ₃	B ₀	B ₁	B ₂	B ₃	-	-
A ₄	A ₅	A ₆	A ₇	A ₄	A ₅	-	-
B ₆	B ₇	B ₄	B ₅	B ₆	B ₇	-	-
A ₈	A ₉	A ₁₀	A ₁₁	A ₈	A ₉	-	-
B ₁₀	B ₁₁	B ₈	B ₉	B ₁₀	B ₁₁	-	-
A ₁₂	A ₁₃	A ₁₄	A ₁₅	A ₁₂	A ₁₃	-	-
B ₁₄	B ₁₅	B ₁₂	B ₁₃	B ₁₄	B ₁₅	-	-

Figure 3(b)

Figure 3(c)

Initial Data Configuration :

00	01	05	12	14	14	34	57
-	-	-	-	-	-	-	-
15	17	24	34	61	65	70	71
-	-	-	-	-	-	-	-

Data Configuration after Step 1

00	01	05	12	-	-	-	-
14	14	24	34	-	-	-	-
15	17	34	57	-	-	-	-
61	65	70	71	-	-	-	-

Data Configuration after Step 2

00	01	05	12	00	01	-	-
24	34	↑ 14	↑ 14	24	34	-	-
↓	↓			↓	↓		
15	17	34	57	15	17	-	-
70	71	↑ 61	↑ 65	70	71	-	-

Figure 3(d)

Data Configuration after first iteration of Step 3

00	01	14	14	00	-	-	-
17	↑ 05	↑ 12	↑ 15	17	-	-	-
↓				↓			
24	34	61	65	24	-	-	-
71	↑ 34	↑ 57	↑ 70	71	-	-	-

Data Configuration after second iteration of Step 3

00	05	14	15	-	-	-	-
01	12	14	17	-	-	-	-
24	34	61	70	-	-	-	-
34	57	65	71	-	-	-	-

Final Data Configuration

00	01	05	12	14	14	15	17
-	-	-	-	-	-	-	-
24	34	34	57	61	65	70	71
-	-	-	-	-	-	-	-

Figure 3(e)

Therefore the total time required by Algorithm HM is

$$(\log m + \log n)t_c + (m + 2n)t_r + 9t_e$$

Correctness of Algorithm HM

We will prove the correctness of this algorithm using the Zero-One Principle.

If each sorted input subfile consists of integers from the set $\{0,1\}$, the difference between the number of zeros (or ones) in any two columns can be at most 1 (see Figure 4(a)).

Therefore, when the elements from the second file are moved to the processors holding the correspondingly-indexed elements from the first file in Step 1, the difference between the number of zeros (or ones) in any two columns will be at most 2. If each file is organized as an $m \times n/2$ matrix then the first few columns on the left will have the same number of zeros (say w) in each column, the next few columns will have $w-1$ zeros and the remaining columns will have $w-2$ zeros in each column.

Since each processor is holding two data elements, the data is organized as a $2m \times n/2$ matrix in the $m \times n/2$ processor array. When the contents of each column are sorted independently, only two rows of data can contain both zeros and ones. All the rows above them will contain all zeros and all the rows below them will contain all ones (see Figure 4(b)).

0	0	0	0	0	0	0	0
-	-	-	-	-	-	-	-
0	1	1	1	0	0	0	1
-	-	-	-	-	-	-	-
1	1	1	1	1	1	1	1
-	-	-	-	-	-	-	-
1	1	1	1	1	1	1	1
-	-	-	-	-	-	-	-

Figure 4(a)

0 _a	0 _b	0 _c	0 _d	-	-	-	-
0 _e	0 _f	0 _g	0 _h	-	-	-	-
0 _i	0 _j	0 _k	1 _l	-	-	-	-
0 _m	1 _n	1 _o	1 _p	-	-	-	-
1 _q	1 _r	1 _s	1 _t	-	-	-	-
1 _u	1 _v	1 _w	1 _x	-	-	-	-
1	1	1	1	-	-	-	-
1	1	1	1	-	-	-	-

0 _a	0 _b	0 _c	0 _d	0 _a	0 _b	-	-
0 _g	0 _h	0 _e	0 _f	0 _g	0 _h	-	-
0 _i	0 _j	0 _k	1 _l	0 _i	0 _j	-	-
1 _o	1 _p	0 _m	1 _n	1 _o	1 _p	-	-
1 _q	1 _r	1 _t	1 _t	1 _q	1 _r	-	-
1 _w	1 _x	1 _u	1 _v	1 _w	1 _x	-	-
1	1	1	1	1	1	-	-
1	1	1	1	1	1	-	-

Figure 4(b)

horizontal connections required for linear adjacency

Figure 4(c)

Figure 4(d)

At this point we would like to view the $m \times n/2$ mesh-connected processor array as a linear processor array of size $m \times n/2$, with the exception that the connection between every $(i \times n/2)^{\text{th}}$ and $(i \times n/2 + 1)^{\text{th}}$ processor is missing. Also, it is obvious that the position, marked as y , of the first 1 in A-registers of this linear array will be at most $n/2$ distance to the right of the position, marked as z , of the first 1 in the R-register, (see Figure 4(c)).

The duplication of data in Step 2 allows us to use the vertical connections of the processor mesh in place of the missing horizontal connections between every $(n/2)^{\text{th}}$ and $(n/2 + 1)^{\text{th}}$ processor (see Figure 4(d)).

In the first iteration of Step 3 the computation in the first $n/4$ columns of each row is duplicated at the end of the preceding row. The data that had to be shifted to $P[i, n/2 - x : n/2 - 1]$ from $P[i + 1, 0 : x - 1]$ is now available in $P[i, n/2 : n/2 + x]$. In all succeeding iterations, the data in the first x columns is identical to the data in columns $n/2$ to $n/2 + x - 1$. Thus, the need to move data from the beginning of each row to the end of the preceding row is obviated. Therefore, if we consider the processor mesh as a linear array, Step 3 of this algorithm is identical to Step 2 of Algorithm M.

At the conclusion of Step 1, the position of the first 1 in the A-registers is at most $n/2$ position to the right of the first 1 in the R-registers. Data movement in Step 2 reduces this difference to $n/4$ positions right or left.

We have shown earlier that Step 2 of Algorithm M suffices to merge two files placed in the A and R-registers of a linear processor array provided the variable x is initialized to at least the difference between the position of first 1 in the A and R-registers of the processor array. From the same argument we conclude that Step 3 of Algorithm HM will merge the two files completely.

At the end of Step 3, the n data elements in each row of the final merged output are stored in a column-major order in the A and R-registers of the first $n/2$ processors. Step 4 unfolds them, thus providing the final output in row-major order.

4. VERTICAL MERGE ALGORITHM

Algorithm VM -- Vertical Merge

Initial data configuration

The two sorted arrays, $A[0:m*n/2-1]$ and $B[0:m*n/2-1]$, to be merged are stored in the processors $P[0:m/2-1, 0:n-1]$ and $P[m/2:m-1, 0:n-1]$ in row-major order (see Figure 5(a)).

Final data configuration

The merged output of the two input arrays will be placed in the processors $P[0:m-1, 0:n-1]$ in row-major order.

Algorithm VM

Step 1 :

Merge the two subfiles in each column using Algorithm M, considering each column as a linearly-connected processor array.

EXCHANGE [all rows, $n/2:n-1$]

MOVEHORZ [$n/2$]

MOVEVERT [-1 , all rows, $0:n/2-1$]

COMPAREHI [all rows , $0:n/2-1$]

MOVEVERT [1 , all rows, $0:n/2-1$]

Step 2 :

Perform Steps 2, 3 and 4 of Algorithm HM.

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅	A ₆	A ₇
-	-	-	-	-	-	-	-
A ₈	A ₉	A ₁₀	A ₁₁	A ₁₂	A ₁₃	A ₁₄	A ₁₅
-	-	-	-	-	-	-	-
B ₀	B ₁	B ₂	B ₃	B ₄	B ₅	B ₆	B ₇
-	-	-	-	-	-	-	-
B ₈	B ₉	B ₁₀	B ₁₁	B ₁₂	B ₁₃	B ₁₄	B ₁₅
-	-	-	-	-	-	-	-

Figure 5(a)

00	01	05	12	15	17	24	34
-	-	-	-	-	-	-	-
14	14	34	57	61	65	70	71
-	-	-	-	-	-	-	-

Figure 5(b)

00	01	05	12	-	-	-	-
15	17	24	34	-	-	-	-
↓	↓	↓	↓				
14	14	34	57	-	-	-	-
61	65	70	71	-	-	-	-

Figure 5(c)

00	01	05	12	-	-	-	-
14	14	24	34	-	-	-	-
15	17	34	57	-	-	-	-
61	65	70	71	-	-	-	-

Figure 5(d)

Time complexity of Algorithm VM

Step 1 of this algorithm requires $3m/2*t_r + \log m*t_c + 2*t_e$ time to merge the contents of each column and a total of $(n/2 + 2)*t_r + t_e + t_c$ time for the rest of the computations in that step. Step 2 requires $3n/2*t_r + 8*t_e + (\log n)*t_c$ time (from the complexity of Algorithm HM). Thus the total time required for Algorithm VM is

$$\begin{aligned} & (\log m + \log n + 2)*t_c + \left(\frac{3m}{2} + 2n + 2\right)*t_r + 11*t_e \\ & \approx (\log mn)*t_c + \left(2n + \frac{3m}{2}\right)*t_r + 11*t_e \end{aligned}$$

Correctness of Algorithm VM

Initially each column j ($0 \leq j < n/2$) of the processor array contains the two sorted subfiles $A[j, n]^\dagger$ and $B[j, n]$ (see Figure 5(b)). In Step 1 the two subfiles in each column are merged together. Next, the subfile in column $j+n/2$ (for $0 \leq j < n/2$) is moved into column j (see Figure 5(c)) and each element of the subfile originally in column j (except the last one) is compared to the next indexed element of the file coming from column $j+n/2$. The two elements being compared are interchanged if they are out of order (see Figure 5(d)).

The subfiles $A[j, n]$ and $B[j, n]$, in column j , consist of the odd-indexed elements of the subfiles $A[j, n/2]$ and $B[j, n/2]$.

[†] If $A[0:n]$ is a file, then $A[j, x]$ represents the subfile $A[j], A[j+x], A[j+2x], \dots, A[j + \lfloor \frac{n-j}{x} \rfloor * x]$.

Similarly the subfiles $A[j + n/2, n]$ and $B[j + n/2, n]$, in column $j + n/2$, are the even-indexed elements of the subfiles $A[j, n/2]$ and $B[j, n/2]$. Hence Step 1 of the Vertical-Merge algorithm places the subfiles $A[j, n/2]$ and $B[j, n/2]$ in column j in sorted order (for $0 \leq j < n/2$).

In Algorithm HM, columns j and $j + n/2$ ($0 \leq j < n/2$) contain the subfiles $A[j, n/2]$ and $B[j, n/2]$ respectively and Step 1 merges the two subfiles and stores the sorted result in column j .

Thus, the intermediate result obtained by applying Step 1 of Algorithm HM on two sorted subfiles stored in the right and left halves of an $m \times n$ mesh-connected processor array is identical to the result obtained on applying Step 1 of Algorithm VM on two sorted subfiles stored in the lower and the upper halves of the $m \times n$ mesh-connected processor array (compare Figure 5(d) with Figure 3(d)). Since the remainder of Algorithm VM is the same as the remainder of Algorithm HM, the correctness of Algorithm VM follows from the correctness of Algorithm HM and the abovementioned facts.

5. SORTING ALGORITHM

Algorithm S -- Sorting Algorithm

Initial data configuration

The n^2 elements to be sorted are stored in the A-registers of processors $P[0:n-1, 0:n-1]$.

Final data configuration

The sorted output will be placed in the processors $P[0:n-1, 0:n-1]$ in row-major order.

Algorithm S

Step 1 :

```
For all odd  $i$   $1 \leq i \leq n-1$  Do
{
    EXCHANGE [  $i$  ,  $0:n-1$  ]
    MOVEVERT [  $i$  ,  $i$  ,  $0:n-1$  ]
}
For all even  $i$   $0 \leq i \leq n-2$  Do
{
    COMPARELO [  $0:n-1$  ,  $0:n-1$  ]
    MOVEVERT [  $-1$  ,  $0:n-1$  ]
    EXCHANGE [  $i+1$  ,  $0:n-1$  ]
}
```

Step 2 :

```

For ( s = 1 ;  $2^s \leq \frac{n}{2}$  ; s = s+1 ) Do
{
    Perform Algorithm HM on processors
    P [  $i*2^s:(i+1)*2^s-1$  ,  $j*2^s:(j+1)*2^s-1$  ]
        for  $0 \leq i < \frac{n}{2^s}$  and  $0 \leq j < \frac{n}{2^s}$ 

    Perform Algorithm VM on processors
    P[ $i*2^{s+1}:(i+1)*2^{s+1}-1$  ,  $j*2^s:(j+1)*2^s-1$ ]
        for  $0 \leq i < \frac{n}{2^s}$  and  $0 \leq j < \frac{n}{2^s}$ 
}

Perform Algorithm HM on processors P [ 0:n-1 , 0:n-1 ]

```

Time Complexity of the Algorithm S

Algorithm S uses the Horizontal-Merge algorithm iteratively to produce sorted subfiles of size 2×2 , ..., $n/2 \times n/2$, $n \times n$, by merging horizontally-adjacent sorted subfiles of size 2×1 , ..., $n/2 \times n/4$, $n \times n/2$. So the total time used by Horizontal-Merge is

$$\begin{aligned}
 & \sum_{i=1}^{\log n} [(2+1)2^i * t_r + 2i * t_c + 9 t_e] \\
 &= 3 (2^{(\log n)+1} - 1) * t_r + \log n (\log n + 1) * t_c + (9 \log n) * t_e \\
 &\approx 6n * t_r + (\log^2 n) * t_c + 9 (\log n) * t_e
 \end{aligned}$$

The Vertical-Merge algorithm is used iteratively to produce sorted subfiles of size 4×2 , ..., $n/2 \times n/4$, $n \times n/2$, by merging vertically-

adjacent sorted subfiles of size 2×2 , ..., $n/4 \times n/4$, $n/2 \times n/2$. Hence the time used by Vertical-Merge is

$$\begin{aligned} & \sum_{i=1}^{\log n} [(2^i + 3 \cdot 2^{i-1}) * t_r + (i+i-1) * t_c + 11 t_e] \\ &= 5 (2^{\log n} - 1) * t_r + (\log^2 n) * t_c + 11 (\log n) * t_e \\ &\approx 5n * t_r + (\log^2 n) * t_c + 11 (\log n) * t_e \end{aligned}$$

Step 1 requires only $t_c + 2t_r$ for the COMPARE and the MOVEVERT instructions. Hence the total time required by the sorting algorithm is

$$11n * t_r + 2 (\log^2 n) * t_c + 20 (\log n) * t_e$$

Step 1 of this algorithm produces sorted subfiles of size 2×1 stored in two vertically adjacent processors. Step 2 applies Horizontal-Merge followed by Vertical-Merge iteratively until we are left with only two horizontally adjacent subfiles. The last instruction in Step 2 merges these two horizontally adjacent subfiles to produce the sorted output.

6. EXTENSIONS

In all the parallel sorting and merging algorithms presented in this paper we have assumed that the processors have no local memory, other than the three internal registers. The number of processors required by these algorithms is equal to the size of the input.

However, if the model of computation is modified to allow $4j$ words of local memory in each processor (sufficient to hold $4j$ elements of the input file), the algorithms requiring n^2 processors with no local memory can be modified to work with as few as n^2/j processors, using the scheme discussed in [BS]. The memory is viewed as consisting of four blocks of size j each, any two of which can be tagged as M_a and M_r (counterparts of the A and R -registers). The input files are partitioned into n^2/j blocks of size j each, and the blocks are assigned to the M_a block of the processors. The modified merging or sorting algorithm which uses n^2/j processors, proceeds in the same way as the unmodified algorithm which uses n^2 processors would have, but it replaces all the COMPARE operations by a MERGE operation, which merges the two input blocks stored in the M_a and M_r blocks of a processor. Route steps are replaced by a ROUTE operation which copies either the M_a or the M_r block of a processor's logical neighbor into the processor's own local memory. The counterpart of the EXCHANGE operation can be performed in negligible time by switching the tags of the memory blocks. In the case of sorting, the block stored in each processor is sorted locally using an $O(j \log j)$ sorting algorithm.

Since the merge operation requires essentially $2j$ compare steps, it can be performed in $T_m = 2j * t_c$ time. The ROUTE operation can be performed in $t_r = j * t_r$ time.

For the algorithms implemented on mesh-connected processor arrays, since the number of processors required is reduced by a factor of j , each dimension of the processor array is reduced by a factor of $j^{1/2}$. Therefore, using the modified model of computation the time required by Algorithm S is

$$\begin{aligned} & \frac{11nj}{j^{1/2}} * t_r + [j * \log j + 4j * \log^2 \frac{n}{j^{1/2}}] * t_c \\ = & 11j^{1/2}n * t_r + [j * \log j + 4j * \log^2(\frac{n}{j^{1/2}})] * t_c \end{aligned}$$

Similarly the time required by algorithm HM is

$$(m + 2n) * j^{1/2} * t_r + \log \frac{mn}{j} * t_c$$

Therefore, for large n and small j , we can reduce the number of processors required by a factor of j , at the cost of increasing the execution time by a factor of $j^{1/2}$.

The algorithms presented in this paper to merge data stored in a mesh-connected processor array, can also be used to merge data stored in only a portion (a few consecutive rows and columns) of the mesh connected processor array. This is possible because the data shifted out from the processor at the end of a row is not used by the processors at the beginning of the next row (the end around connections are used only to circumvent the necessity of extra processors required to

accumulate the data being shifted out at the end of each row and column). If the data is stored only in a few consecutive rows and columns, the processor holding the last element of a row is not logically adjacent to the processor holding the first element of the next row.

It is necessary to have merge algorithms that manipulate only a portion of the array because, in our sorting algorithm, the subfiles being merged are smaller than the processor array size.

However, if the job is to merge two subfiles of size n^2 each using a mesh-connected processor array of size $n \times n$, it can be done more efficiently by using the end-around connections as described in Algorithm EM below.

Algorithm EM -- Merge algorithm which uses end-around connections

Initial data configuration

The two sorted subfiles (of size n^2 each) to be merged are stored in the A and R-registers of the $n \times n$ mesh-connected processor array respectively, in row-major order.

Final data configuration

The merged output will be stored in row-major order, with each processor holding two elements of the output, the smaller of which is in the A-register.

Algorithm EM

Step 1 :

Merge the two subfiles in each column using Steps 2 and 3 of Algorithm M, considering each column as a linearly connected processor array.

Step 2 :

Merge the two subfiles in each row using Steps 2 and 3 of Algorithm M.

It can be easily seen that the time required by algorithm EM is

$$2n \cdot t_r + (2 \log n) \cdot t_c + 4 \cdot t_e$$

Algorithm EM is similar to Algorithm HM except that the computations carried out in the first half of each row are not duplicated at the end of the previous row. The duplication of computations had been used to make up for the missing horizontal connections between the processors at the end of a row and the beginning of the next row.

7. SUMMARY

In this paper we presented an implementation of Batchier's odd-even merge algorithm for a linearly-connected processor array of n processors. Our algorithm merges two sorted subfiles of size $n/2$ placed in the left and the right halves of the processor array in nondecreasing order, in $3n/2$ route steps and $\log n$ compare-exchange steps. This is faster than the algorithm proposed by Thompson and Kung [TK], which requires $4n$ route steps, and the Row-Merge and Column-Merge algorithms proposed by Nassimi and Sahni [NS1], which merge a nondecreasing and a nonincreasing sequence of size $n/2$ each in $2n$ route steps.

We generalized this first algorithm to a Horizontal-Merge algorithm which merges two sorted subfiles of size $m \times n/2$, stored in the left and the right halves of an $m \times n$ mesh-connected processor array in nondecreasing order, in $m + 2n$ route steps and $(\log m + \log n)$ compare-exchange steps. This is faster than its counterpart used by Thompson and Kung, which requires $2m + 4n$ route steps and $m + \log n$ compare-exchange steps. It is also faster than the Horizontal-Merge algorithm used by Nassimi and Sahni, which requires $2m + 2n$ route steps and $(\log m + \log n)$ compare-exchange steps to merge two subfiles of size $m \times n/2$ each, one of which is in nondecreasing order and the other is in nonincreasing order.

Then we gave a Vertical-Merge algorithm to merge two vertically-aligned subfiles of size $m/2 \times n$, stored in nondecreasing order in an $m \times n$ mesh-connected processor array. Our algorithm requires $3m/2 + 2n$

route steps and $(\log m + \log n)$ compare-exchange steps. Nassimi and Sahni have proposed a Vertical-Merge algorithm to merge a vertically-aligned pair of subfiles in $2m + 2n$ route steps and $(\log m + \log n)$ compare-exchange steps, provided one of the subfiles being merged is sorted in nondecreasing order and the other in nonincreasing order.

Finally we gave an sorting algorithm which uses the Horizontal-Merge and the Vertical-Merge algorithms proposed by us, to sort n^2 elements stored in an $n \times n$ mesh-connected processor array, in $11n$ route steps and $O(\log^2 n)$ compare-exchange steps. This latter algorithm requires $3n$ fewer route steps than the algorithm proposed by Nassimi and Sahni. The sorting algorithm proposed by Thompson and Kung requires $6n + O(n^{2/3} \log n)$ route steps and $n + O(n^{2/3} \log n)$ compare-exchange steps. But for practical values of n ($4 \leq n \leq 512$), $n^{2/3} \log n$ is greater than n and therefore, for practical values of n , our algorithm will be faster than the algorithm proposed by Thompson and Kung.

8. REFERENCES

- [Br] G. H. Barnes, The ILLIAC IV computer. IEEE Trans. on Comp. vol. c-17, no. 8 (Aug. 1968), pp. 746-757.
- [Bt] K. E. Batcher, Sorting networks and their application. Proc. AFIPS 1968 SJCC, vol. 32, AFIPS Press, Montvale, N.J., pp. 307-314.
- [BS] G. Baudet and D .Stevenson, Optimal sorting algorithms for parallel computers. IEEE Trans. on Comp. vol. c-27, no. 1 (Jan. 1978), pp. 84-87.
- [Fl] M. J. Flynn, Very high-speed computing systems. Proc. IEEE vol. 54, no. 12 (Dec. 1966), pp. 1901-1909.
- [Kn] D. E. Knuth, The Art of Computer Programming. Vol. 3 : Sorting and Searching. Addison-Wesly, Reading, Mass., 1973.
- [NS1] D. Nassimi and S. Sahni, Bitonic sort on a mesh-connected parallel computer. IEEE Trans. on Comp. vol. c-28, no. 1 (Jan. 1979), pp. 2-7.
- [NS2] D. Nassimi and S. Sahni, Parallel permutation and sorting algorithms and a new generalized connection network. University of Minnesota, TR 79-9, 1979.
- [S11] H. J. Siegel, A model of SIMD machines and a comparison of various interconnection networks. IEEE Trans. on Comp. vol. c-

28, no. 12 (Dec. 1979), pp. 907-917.

- [S12] H. J. Siegel, Interconnection networks for SIMD machines.
Computer vol. 12, no. 6 (June 1979), pp. 57-65.
- [S1] D. L. Slotnick, W. C. Borck and R. C. McReynolds, The SOLOMON
Computer. Proc. FJCC, AFIPS, vol. 22 (1962). Washington D.C.
Spartan, pp. 97-107.
- [St] H. S. Stone, Parallel Processing with the Perfect Shuffle. IEEE
Trans. on Comp. vol. c-20, no. 2 (Feb. 1971), pp. 153-161.
- [TK] C. D. Thompson and H. T. Kung, Sorting on a mesh-connected
parallel computer. Comm. ACM vol. 20, no. 4 (April 1977), pp.
263-271.