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

MANOJ KUMAR, MEMBER, IEEE, AND DANIEL S. HIRSCHBERG

***Abstract*—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 $11\,n$ 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 $14\,n$ route steps for practical values of n, $4 \le n \le 512$ (i.e., mesh-connected processor arrays containing 16 to 262 144 processors). The merge algorithms for the mesh-connected processor array use wrap around connections. These connections can be avoided at the expense of some extra hardware.

***Index Terms*—Linearly connected processor arrays, mesh-connected processor arrays, odd-even merge algorithm, SIMD machines.**

INTRODUCTION

BATCHER'S odd-even merge and bitonic merge algorithms [2] have been popular among designers of parallel sorting and merge algorithms [6], [12], [3], perhaps because of their inherent parallelism. Sorting schemes using Batcher's merge algorithm require $n \log^2 n^1$ fetch, compare and store steps to sort an array of size n on a SISD machine [4]. On an SIMD machine [4], 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 [5], [11].

A commonly used interconnection pattern for SIMD machines is the mesh connection [1], [8]–[10]. In this model, the processors are arranged in a two-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 instruction executes in t_c time.

Illiac IV has a similar architecture [1]. 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

$$
\begin{aligned}
&\text{if } 0 \le v \le n-1 \text{ then}\\
&\quad x = u;\ y = v\\
&\text{else if } v = n \text{ then}\\
&\quad x = (u+1) \bmod n;\ y = 0\\
&\text{else if } v = -1 \text{ then}\\
&\quad x = (u-1) \bmod n;\ y = n-1
\end{aligned}
$$

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 an $O(n)$ lower bound for sorting using the Illiac IV interconnection pattern, because it requires at least $4n - 1$ route steps to sort a file of size $n \times n$, in which the smallest and the largest elements are on wrong ends. 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 machines with a higher dimensional interconnection pattern, such as mesh-connected machines (two-dimensional interconnection pattern) [7]. 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 [6], which makes use of Batcher's

Manuscript received November 21, 1980; revised August 17, 1982. This work was supported by the National Science Foundation under Grants MCS-80-03431 and MCS-82-00362.

M. Kumar is with the Department of Electrical Engineering, Rice University, Houston, TX 77001.

D. S. Hirschberg is with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

[1] 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.

0018-9340/83/0300-0254$01.00 © 1983 IEEE

bitonic merge algorithm. Their algorithm requires $\simeq 14\,n$ route steps and $\simeq 2\log^2(n)$ 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. C. D. Thompson and H. T. Kung have made use of Batcher's odd-even merge algorithm to implement a sorting scheme for mesh-connected computers which requires $\simeq 6n + 0(n^{2/3}\log n)$ route steps and $n + 0(n^{2/3}\log n)$ compare-exchange steps asymptotically (optimal within a factor of 2). Preliminary investigation by Thompson and Kung indicates that this algorithm is optimal within a factor of 7 for all n, under the assumption that $t_c \leq 2t_r$.

H. S. Stone has used an interconnection pattern called the Perfect Shuffle [11]. With this interconnection pattern, Batcher's odd-even merge algorithm can be used to sort n elements in $0(\log^2 n)$ steps.

We will give an implementation of Batcher'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 ($\simeq 6n + 0(n^{2/3}\log n)$ route steps and $n + 0(n^{2/3}\log n)$ compare-exchange steps, for $m = n$). It is interesting to note that Thompson and Kung'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.

The horizontal and the vertical merge algorithms, as presented in this paper, use the horizontal and vertical wrap around connections of the mesh-connected processor arrays. However, the use of these connections can be avoided by providing buffer memory with the processors in the last row of the mesh-connected processor array.

Linearly Connected Processor Array

A linearly connected processor array of size n is an SIMD machine [4] 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 (accept) and the R (reject) register.

3) Each processor is capable of executing the following instructions.

i) 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.

ii) 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 (during a route instruction all data movement is in one direction). This instruction requires t_r time to execute.

iii) 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 [8], a set of processors can be prevented from executing the broadcast instruction. 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 an SIMD machine consisting of $m * n$ 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 Thompson and Kung [12], and Nassimi and Sahni [6].

Batcher's Odd-Even Merge Algorithm

The odd-even merge algorithm to merge S and T, two sorted lists of sizes s and t (elements numbered $0, 1, 2, \cdots$), 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:

i) Split the lists S and T into their odd-indexed elements s_o, t_o and their even-indexed elements s_e, t_e.

ii) 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.

iii) For all i, compare the ith element of m_e with the $(i + 1)$th element of m_o (if it exists) and interchange their positions if they are out of order.

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 $(x{:}y)$. When the argument of an operation specifies more than one processor, all the specified processors perform the operation simultaneously.

EXCHANGE$[X]$

The processors $P[X]$ (if X is $x{:}y$ then we mean $P[x]$, $P[x + 1], \cdots, P[y]$) 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 nonzero positive (negative) integer and $X = x_1{:}x_2$, processors $P[x_1 - i{:}x_2 - i]$, for $i = 1, 2, \cdots, j$ $(i = -1, -2, \cdots, j)$, 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 $P[x_1 - j{:}x_2 - j]$ 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 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 $(y - x + 1) * t_r + (y - x + 2) * t_e$ time by the following algorithm:

```
For (w = y = 1, y, y − 1, ···, x + 1) Do
{       MOVE [−1, w:2y − w + 2]
        EXCHANGE [w − 1]
}
EXCHANGE [x:2y − x + 1].
```

Fig. 1(a) illustrates the UNFOLD [0:1] operation.

Algorithm M—An Efficient Implementation of Batcher's Merge Algorithm

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

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, n/8, ···, 1) Do
{   COMPAREHI [x:n/2 − 1]
    MOVE [⌈x/2⌉]
}
```

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 are 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 $(n/2) * t_r + (n/2 + 1) * t_e$ time.

Therefore, the total time required by Algorithm M is

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

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 Fig. 1(b) and (c)].

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

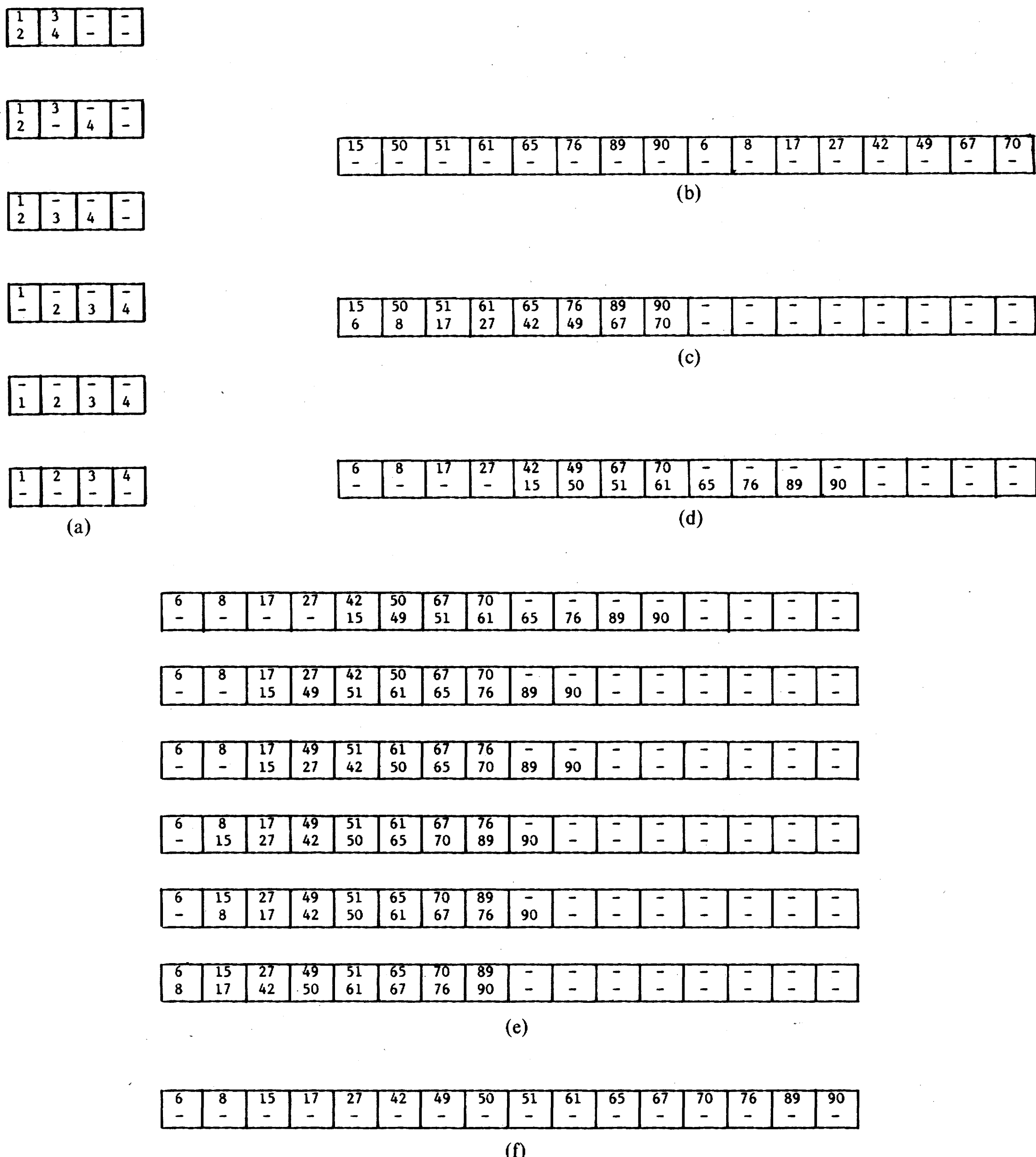


Fig. 1. Implementation of odd-even merge algorithm on linearly connected processor arrays. (a) UNFOLD [0:1] operation for processors $P[0:3]$. (b) Initial data configuration. (c) Data configuration after Step 1. (d) Data configuration after Step 2. (e) Data configuration during execution of Step 3. (f) Final data configuration.

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 Fig. 1(d) and (e)].

Step 4 arranges the sorted file in the A-registers of the processor array [see Fig. 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 one used by Steps 2 and 3. Although 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.

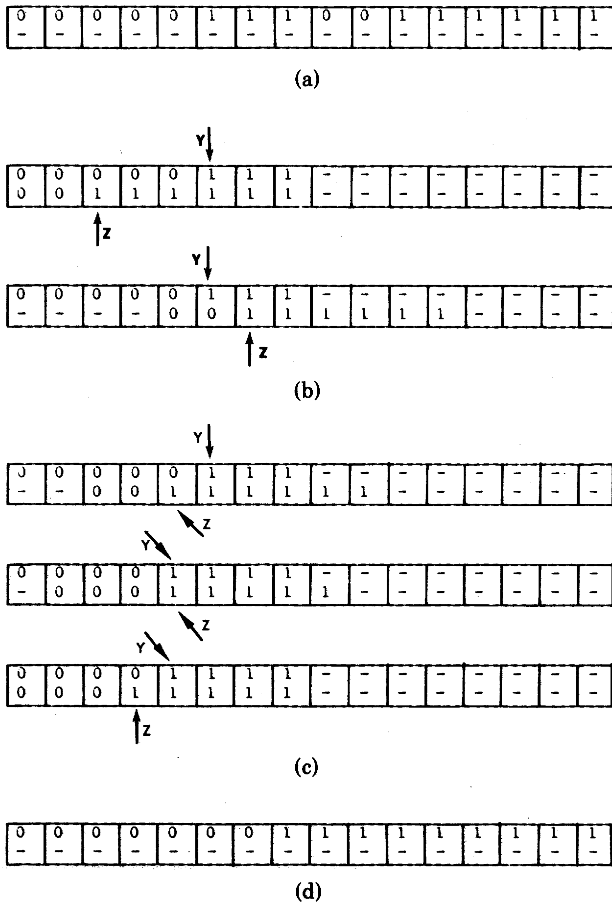


Fig. 2. Merging files of binary data. (a) Initial data configuration. (b) Data configurations during execution of Steps 1 and 2. (c) Data configurations during execution of Step 3. (d) Final data configuration.

Correctness of Algorithm M

To show the correctness of Algorithm M, we will make use of the Zero-One Principle [5].

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 Fig. 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.

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 Fig. 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 Fig. 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 Fig. 2(d)].

Extension of Algorithm M to Two Dimensions

The algorithm developed in the previous section required $0(n)$ 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 Fig. 3(a)]. The input to the vertical merge algorithm consists of two sorted subfiles which are organized as a pile [see Fig. 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.

COMPAREHI$[R, C]$

When $R = r_1{:}r_2$ and $C = c_1{:}c_2$, the processors in the set $\{P(r, c] \mid r_1 \leq r \leq r_2$ 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$[j, R, C]$

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 $P[r - j, c]$, provided they exist. Since $|j|$ route instuctions 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 and 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).

COPY[i, j, R, C]

In this operation, i and j specify the source and destination registers (one of A, R and T) for the copy instruction, which is executed by each process in the set $\{P[r, c] | c \in C \text{ and } r \in R\}$. This operation requires t_e time.

EXCHANGE [i, j, R, C]

Here i and j specify registers (one of A, R and T) for the exchange instruction, which is executed by each processor in the set $\{P[r, c] | c \in C \text{ and } r \in R\}$. If i and j are not specified, registers A and R are used. This operation requires t_e time.

Algorithm HM—Horizontal Merge

The two sorted arrays to be merged, $A[0{:}m * n/2 - 1]$ and $B[0{:}m * n/2 - 1]$, 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 Fig. 3(a)]. The merged output of the two input arrays will be placed in processors $P[0{:}m - 1, 0{:}n - 1]$ in row-major order.

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:

EXCHANGE [all rows, $0{:}n/4 - 1$]
COPY [R, T, all rows, $0{:}n/2 - 1$]
MOVEHORZ [$-n/4$]
COPY [A, R, all rows, $0{:}n/4 - 1$]
COPY [R, T, all rows, $n/2{:}3n/4 - 1$]
MOVEHORZ [$-n/4$]
EXCHANGE [R, T, all rows, $n/4{:}3n/4 - 1$]
COPY [T, A, all rows, $0{:}n/4 - 1$ and $n/2{:}3n/4 - 1$]
MOVEHORZ [$n/4$]
COPY [T, R, all rows, $n/4{:}n/2 - 1$]
/* Execution of this step changes the data configuration shown in Fig. 3(b) to that of Fig. 3(c). */

Step 3:

For
($x = n/4, n/8, \cdots, 1$)
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, $0{:}n/2 - 1$]

Fig. 3(d) illustrates the Horizontal Merge algorithm for $m = 2$ and $n = 8$.

Step 1 of Algorithm HM uses the vertical wrap around connections while performing linear merge on contents of each column, because the last few data elements of a column which are shifted down from the last processor are temporarily stored in the first few processors of that column. Since these data elements are not used in any computation by the first few processors, the use of wrap around connections can be avoided by providing the last processor in each column with $n/2$ buffer registers.

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 ($\log m + 1$) compare instructions to merge the subfiles in each column. The total time required by Step 1 is thus $(n/2 + m) * t_r + (\log m + 1) * t_c + t_e$. Step 2 requires $(3n/4) * t_r + 7 * t_e$ time to execute.

The **For** loop in Step 3 is iterated $(\log n) - 1$ times. In each iteration of the **For** loop, one COMPARE and two MOVEVERT operations are performed. The total number of horizontal route instructions performed over all the iterations of the **For** loop is $n/4$. Hence Step 3 requires $(\log n - 1) * t_c + (n/4 + 2 \log n - 2) * t_r$ time to execute. Step 4 requires $(n/2) * t_r + (n/2 + 1) * t_e$ time.

Therefore the total time required by Algorithm HM is

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

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 Fig. 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)

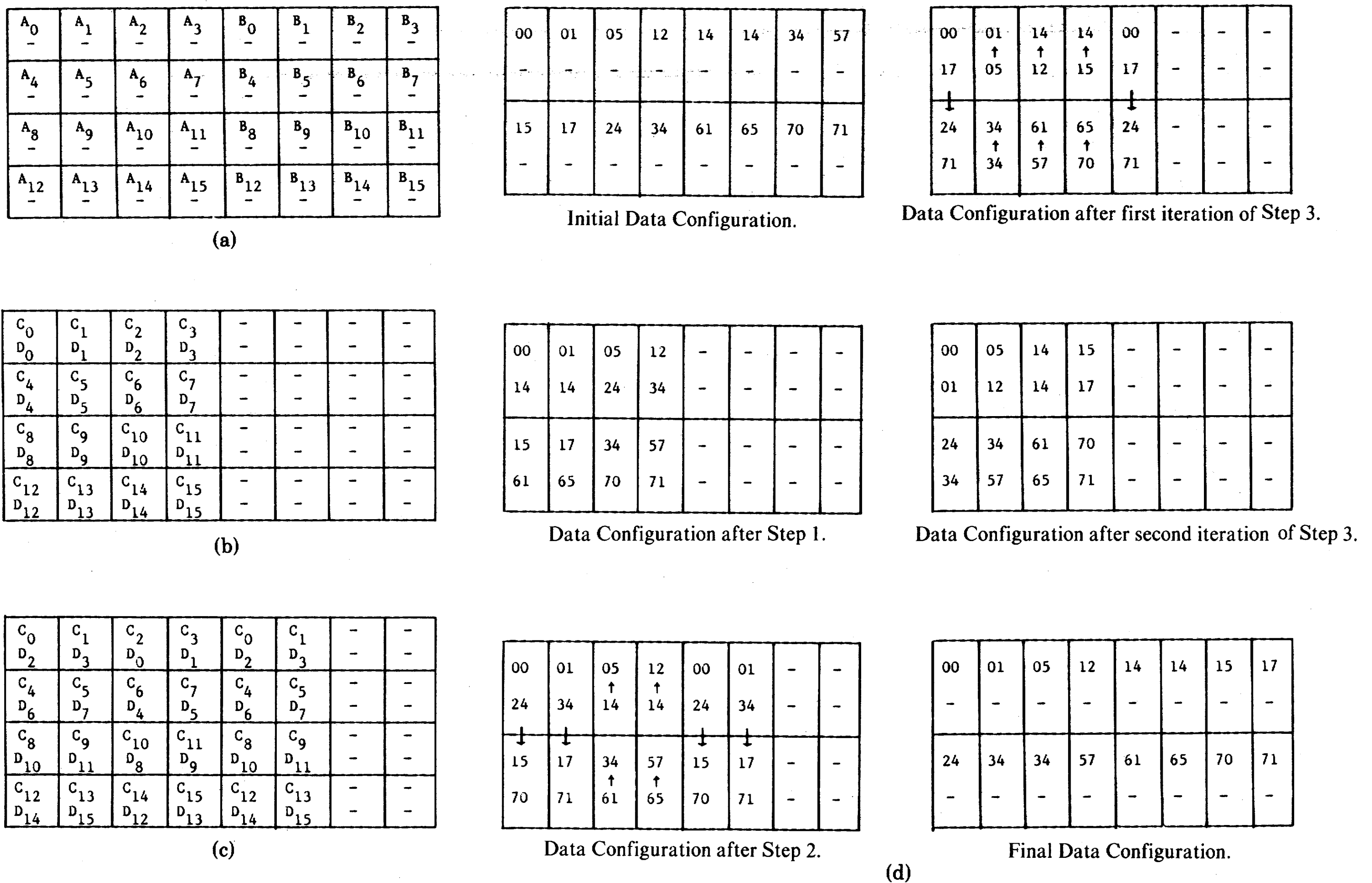


Fig. 3. Implementation of horizontal merge. (a) Initial data configuration. (b) Data configuration after execution of Step 1. (c) Data configuration after execution of Step 2. (d) Example of Horizontal Merge.

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 are 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, at most 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 Fig. 4(b)].

At this point we would like to view the $m \times n/2$ mesh-connected processor array as a linear processor array of size $m * n/2$, with the exception that the connection between every $(i * n/2)$th and $(i * n/2 + 1)$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 Fig. 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 $(n/2)$th and $(n/2 + 1)$th processor [see Fig. 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 $P[i + 1, 0{:}x - 1)$ are now available in $P[i, n/2{:}n/2 + x]$. In all succeeding iterations, the data in the first x columns are 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.

Algorithm VM—Vertical Merge

The two sorted arrays to be merged, $A[0{:}m * n/2 - 1]$ and $B[0{:}m * n/2 - 1]$, are stored in the processors $P[0{:}m/2 - 1,$

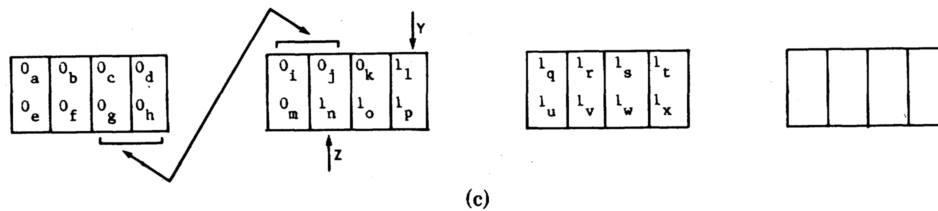
- 1	- 1	- 1	- 1	- 1	- 1	- 1	- 1
- 1	- 1	- 1	- 1	- 1	- 1	- 1	- 1
- 1	- 0	- 0	- 0	- 1	- 1	- 1	- 0
- 0	- 0	- 0	- 0	- 0	- 0	- 0	- 0

(a)

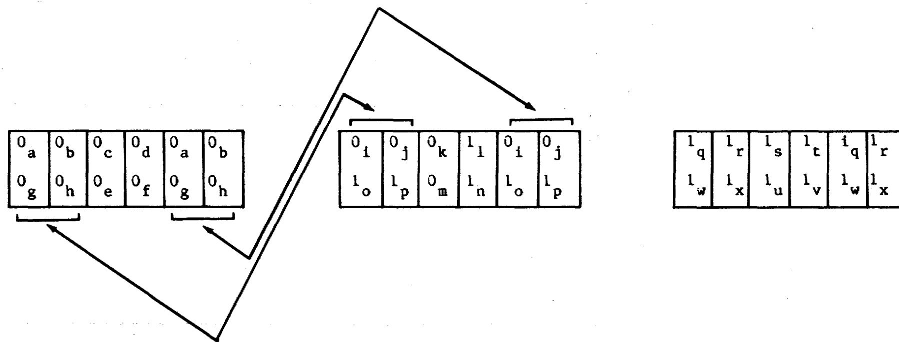
0_a 0_e	0_b 0_f	0_c 0_g	0_d 0_h	- -	- -	- -	- -
0_i 0_m	0_j 1_n	0_k 1_o	1_l 1_p	- -	- -	- -	- -
1_q 1_u	1_r 1_v	1_s 1_w	1_t 1_x	- -	- -	- -	- -
1 1	1 1	1 1	1 1	- -	- -	- -	- -

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

(b)



(c)



(d)

Fig. 4. Horizontal merge on files of binary data. (a) Initial data configuration. (b) Data configurations after Steps 1 and 2. (c) Horizontal connections required for linear adjacency. (d) Effective linear adjacency obtained by data duplication.

$0{:}n-1]$ and $P[m/2{:}m-1], 0{:}n-1]$ in row-major order [see Fig. 5(a)]. 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.

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}

(a)

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

(b)

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

(c)

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

(d)

Fig. 5. Vertical merge. (a) Initial data configuration. (b) Example of initial data configuration. (c) Data configuration after MOVEHORZ operation in Step 1. (d) Data configuration after completion of Step 1.

Time Complexity of Algorithm VM

Step 1 of this algorithm requires $3m/2 * t_r + \log m * t_c + (m/2 + 2) * t_e$ 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 $(3n/2 + 2 \log n - 2) * t_r + (n/2 + 8) * t_e + ((\log n) - 1) * t_c$ time (from the complexity of Algorithm HM). Thus the total time required for Algorithm VM is

$$(\log m + \log n) * t_c + \left(\frac{3m}{2} + 2n + 2 \log n\right) * t_r + \left(\frac{m}{2} + \frac{n}{2} + 11\right) * t_e$$

Correctness of Algorithm VM

Initially each column j $(0 \leq j < n/2)$ of the processor array contains the two sorted subfiles $A[j, n]$[2] and $B[j, n]$ [see Fig. 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 Fig. 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 Fig. 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 $B[j, n/2]$. 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 Fig. 5(d) with Fig. 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.

Algorithm S—Sorting Algorithm

The n^2 elements to be sorted are stored in the A-registers of processors $P[0{:}n - 1, 0{:}n - 1]$. The sorted output will be placed in the processors $P[0{:}n - 1, 0{:}n - 1]$ in row-major order.

Step 1:

```
For all odd i 1 ≤ i ≤ n −1 Do
{   EXCHANGE [i, 0:n −1]
    MOVEVERT [1, i, 0:n −1]
}
For all even i 0 ≤ i ≤ 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, 4, 8, ···, n/2) Do
{   Perform Algorithm HM on each s × s subarray
    Perform Algorithm VM on each s × 2s subarray
}
Perform Algorithm HM on processors P[0:n − 1, 0:n − 1]
```

Time Complexity of Algorithm S

Algorithm S uses the horizontal merge algorithm iteratively to produce sorted subfiles of size $2 \times 2, \cdots, n/2 \times n/2, n \times n$, by merging horizontally adjacent sorted subfiles of size 2

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

$\times 1, \cdots, n/2 \times n/4, n \times n/2$. So the total time used by horizontal merge is

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

The vertical merge algorithm is used iteratively to produce sorted subfiles of size $4 \times 2, \cdots, n/2 \times n/4, n \times n/2$, by merging vertically-adjacent sorted subfiles of size $2 \times 2, \cdots, n/4 \times n/4, n/2 \times n/2$. Hence the time used by vertical merge is

$$= \sum_{i=2}^{\log n} [(2^i + 3 * 2^{i-1} + 2i - 2) * t_r + (i + i - 1) * t_c + (2^{i-1} + 2^{i-2} + 11) * t_e]$$
$$= [5(2^{\log n} - 2) - (\log^2 n - \log n)] * t_r + (\log^2 n - 1) * t_c + [2^{\log n} + 2^{(\log n)-1} - 14 + 11 \log n] * t_e$$
$$= (5n + \log^2 n - \log n - 10) * t_r + (\log^2 n - 1) * t_c + \left(\frac{3n}{2} + 11 \log n - 14\right) * t_e$$

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

$$[11n + 2(\log^2 n - \log n - 7)] * t_r + [2 \log^2 n + \log n] * t_c + \left[\frac{5n}{2} + 20 \log n - 13\right] * t_e \simeq 11nt_r + 2 \log^2 nt_c + \frac{5n}{2} t_e$$

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.

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 [3]. With this modification the time required by Algorithm S is

$$(11 j^{1/2} n * t_r + \left[j \log j + 4 j \log^2 \left(\frac{n}{j^{1/2}}\right)\right] * t_c$$

A Special Case of Two-Dimensional Merge

If the size of each of the two files to be merged is equal to the size of the mesh-connected processor array, then the wrap around connections can be used effectively to reduce the time required to merge them. The two sorted files of size n^2 each are stored, in row major order, in the A- and R-registers of a $n \times n$ mesh-connected processor array. Using Steps 2 and 3 of Algorithm M, we first merge the subfiles in each column, and then merge the subfiles in each row similarly. This leaves the merged output stored in row major order, with each processor holding two elements of the output, the smaller of which is in the A-register. The total time required is $2n * t_r + (2 \log n + 1) * t_c$.

Summary

In this paper we presented an implementation of Batcher'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 [12], which requires $4n$ route steps, and the row merge and column merge algorithms proposed by Nassimi and Sahni [6], 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.

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 a sorting algorithm which uses the horizontal merge and the vertical merge algorithms to sort n^2 elements stored in an $n \times n$ mesh-connected processor array in $11n$ route steps (the contribution of the low order terms is less than n for all values of n), $0(\log^2 n)$ compare-exchange steps and $5n/2$ exchange steps. This algorithm requires $3n$ fewer route steps and $5n/2$ more exchange steps, than the algorithm proposed by Nassimi and Sahni. Thus, we have reduced interprocessor communication by introducing comparable amount of intraprocessor communication.

The sorting algorithm proposed by Thompson and Kung requires $6n + 0(n^{2/3} \log n)$ route steps and $n + 0(n^{2/3} \log n)$ compare-exchange steps. Preliminary investigation by Thompson and Kung indicates that for all values of n, their algorithm is optimal only within a factor of 7, under the assumption that $t_c \leq 2t_r$. When $t_c \leq 2t_r$ condition is not valid,

this optimality factor will be still higher. Therefore, for some values of n ($4 \leq n \leq 512$), our sorting algorithm will be faster than Thompson and Kung's algorithm.

References

[1] G. H. Barnes, "The Illiac IV computer," *IEEE Trans. Comput.*, vol. C-17, pp. 746–757, Aug. 1968.

[2] K. E. Batcher, "Sorting networks and their application," in *Proc. AFIPS 1968 SJCC*, AFIPS Press, Montvale, NJ, vol. 32, pp. 307–314.

[3] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers," *IEEE Trans. Comput.*, vol. C-27, pp. 84–87, Jan. 1978.

[4] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.

[5] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.

[6] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Trans. Comput.*, vol. C-28, pp. 2–7, Jan. 1979.

[7] D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network," Univ. Minnesota, Minneapolis, Tech. Rep. 79-9, 1979.

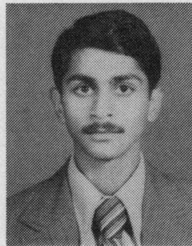
[8] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Trans. Comput.*, vol. C-28, pp. 907–917, Dec. 1979.

[9] H. J. Siegel, "Interconnection networks for SIMD machines," *Computer*, vol. 12, pp. 57–65, June 1979.

[10] D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The SOLOMON Computer," *Proc. FJCC, AFIPS*, vol. 22, Washington, DC, Spartan, pp. 97–107, 1962.

[11] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, Feb. 1971.

[12] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 263–271, Apr. 1977.



Manoj Kumar (M'81) received the B.Tech. degree from the Indian Institute of Technology, Kanpur, India, in 1979, and the M.S. degree from Rice University, Houston, TX, in 1981, both in electrical engineering.

He is currently completing the Ph.D. degree in Electrical Engineering at Rice University. His research interests include parallel computation and interconnection networks for parallel/distributed processing.

Daniel S. Hirschberg received the B.E. degree in electrical engineering from the City College of New York in 1971, and the Ph.D. degree in computer science from Princeton University, Princeton, NJ, in 1975.

From 1975–1981 he was an Assistant Professor of Electrical Engineering at Rice University, Houston, TX. Since 1981 he has been an Associate Professor in the Department of Information and Computer Science at the University of California, Irvine. His research has centered on the design and complexity analysis of algorithms and has concentrated on common subsequence problems, knapsack problems, graph algorithms, and algorithms using a parallel processor.

Dr. Hirschberg is a member of the Association for Computing Machinery and the Special Interest Group on Automata and Computability Theory.