Bitonic Sort on a Mesh-Connected Parallel Computer

DAVID NASSIMI AND SARTAJ SAHNI

Abstract—**An $0(n)$ algorithm to sort n^2 elements on an Illiac IV-like $n \times n$ mesh-connected processor array is presented. This algorithm sorts the n^2 elements into row-major order and is an adaptation of Batcher's bitonic sort. A slight modification of our algorithm yields an $0(n)$ algorithm to sort n^2 elements into snake-like row-major order. Extensions to the case of a j-dimensional processor array are discussed.**

Index Terms—**Bitonic sort, complexity, mesh-connected parallel computer, parallel sorting, SIMD machine.**

I. INTRODUCTION

BATCHER'S bitonic sort [2], [6, pp. 232–233, 237] is based upon his algorithm to sort a bitonic sequence into nondecreasing order. A sequence $X = (x_1, x_2, \cdots, x_N)$ is said to be *bitonic* [2], [8] if either 1) there is an index i, $1 \le i \le N$, such that $x_1 \le x_2 \le \cdots \le x_i \ge x_{i+1} \ge \cdots \ge x_N$ or 2) the sequence can be shifted cyclically so that condition 1) is satisfied. Batcher's algorithm to sort a bitonic sequence X is to recursively sort the bitonic sequences XODD $= (x_1, x_3, x_5, \cdots)$ and XEVEN $= (x_2, x_4, x_6, \cdots)$ and then perform the comparison-interchanges $x_1 : x_2, x_3 : x_4, x_5 : x_6, \cdots$. During the comparison-interchange $x_i : x_{i+1}$, x_i is replaced by the smaller of x_i and x_{i+1} and x_{i+1} becomes the larger of the two. Any sequence $Y = (y_1, y_2, \cdots, y_N)$ may be sorted by recursively sorting $(y_1, y_2, \cdots, y_{\lceil N/2 \rceil})$ into nonincreasing order, $(y_{\lceil N/2 \rceil + 1}, \cdots, y_N)$ into nondecreasing order (or vice versa) and then sorting the bitonic sequence $(y_1, y_2, \cdots, y_N)$ into nondecreasing order using Batcher's method.

Bitonic sort has been adapted by Orcutt [7] and Thompson and Kung [9] for an $n \times n$ mesh-connected parallel computer. The computer consists of $N = n^2$ identical processors configured in a manner similar to the Illiac IV machine [1]. The assumptions we shall be making on the machine model are as follows.

1) It is an SIMD type [4] machine. The $N = n \times n$ identical processors may be thought of as positioned according to an $n \times n$ array $P(0{:}n-1, 0{:}n-1)$. Each processor $P(i, j)$ is connected to its neighbor processors $P(i+1, j)$, $P(i-1, j)$, $P(i, j+1)$, and $P(i, j-1)$ if they exist. The end-around connections of the Illiac IV are not assumed here.

2) Each processor has three registers: one routing register R_r and two storage registers R_s, and R_t.

Manuscript received February 1, 1978; revised June 6, 1978 and July 24, 1978. This work was supported in part by the National Science Foundation under NSF Grant MCS 76-21024.

The authors are with the Department of Computer Science, University of Minnesota, Minneapolis, MN 55455.

3) A REGISTER INTERCHANGE instruction with time $= \tau_I$. Each selected processor unconditionally interchanges the contents of two of its registers. (The same registers are used for all processors.) In our algorithm, only column-selectability and row-selectability of processors is needed.

4) A ROUTE instruction with time $= \tau_r$. *All* processors route the contents of their R_r to their immediate neighbor in the same direction. Thus, this instruction simply *shifts* the entire R_r-array (end-off, zero-filled) unit-distance in one of the four directions up, down, left, or right.

5) A COMPARE-INTERCHANGE instruction with time $= \tau_c$. *All* processors do the (hardware) equivalent of the following statement:

$$\textbf{If } \text{SIGN}(I, S) * (R_r - R_s) < 0$$
$$\textbf{then } \text{interchange } (R_r, R_s)$$

where I = processor index, and S = "pass number" of the algorithm. The function SIGN will be specified later. After a compare-interchange instruction, we shall refer to the element in R_s as the "accepted" element (to be kept by the processor) and the one in R_r as the "rejected" element (to be routed back). Note that even though *all* processors carry-out this instruction, only $N/2$ of the processors would be doing "useful work." The result of the other half is "don't care."

The sorting problem studied in [3], [7], and [9] is that of routing the contents of the $n \times n$ routing registers to destination processors. Each data item is to be routed to a distinct processor. The processors are assumed indexed in some manner and the routing is such that the Ith processor is to contain the Ith smallest element, $1 \le I \le N$. Three different indexing schemes have been considered by Thompson and Kung [9]: row-major, shuffled row-major, and snake-like row-major. In row-major order, the index I of processor $P(i, j)$ is $i * n + j$ (i.e., processors are indexed left to right, top to bottom). In shuffled row-major, the index of a processor is obtained by shuffling its row-major index. For example, if the row-major index in binary is $b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8$ then its shuffled index is $b_1 b_5 b_2 b_6 b_3 b_7 b_4 b_8$. Snake-like row-major indexing is obtained by indexing the processors by row (as in row-major). Processors on even rows are indexed left to right while those on odd rows are indexed right to left (recall that rows are numbered 0 through $n-1$).

Thompson and Kung [9] present fast parallel algorithms for sorting into snake-like row-major and shuffled row-major order. For snake-like row-major order, they present an s^2-way merge algorithm requiring $6n + 0(n^{2/3} \log n)$ routing steps and $n + 0(n^{2/3} \log n)$ comparison-interchanges. Thus, the time needed to sort n^2 elements into snake-like row-major order is $(6n + 0(n^{2/3} \log n))\tau_r + (n + 0(n^{2/3} \log n))\tau_c$. Following a sort into snake-like row-major order the elements may be rearranged into row-major order by reversing the order of elements in odd numbered rows. The additional time needed for this is $2(n-1)\tau_r + 0(\log n)\tau_I$. Thompson and Kung also analyze bitonic sort for shuffled row-major order. Their algorithm takes $(14(n-1) - 8 \log n)\tau_r + (2 \log^2 n + \log n)\tau_c$ time. Their algorithms require each processor to have only two registers. They also point out that if $n \times n$ elements have already been sorted by some index function, and if each processor can store n elements, then the $N = n^2$ elements can be sorted with respect to any other index function using an additional $4(n-1)\tau_r$ units of time. Orcutt [7] analyzes bitonic sort for the case of row-major order. His algorithm takes $0((n \log n)\tau_r + (\log^2 n)(\tau_c + \tau_I))$ time to sort n^2 elements.

In this paper we shall obtain a different adaptation of bitonic sort for row-major ordering. Our bitonic sort algorithm will require $(14(n-1) - 8 \log n)\tau_r + (2 \log^2 n + \log n)\tau_c + (4.5 \log^2 n + 1.5 \log n)\tau_I$ time. If we include the register interchanges needed by the algorithm of [9], the time for that algorithm becomes $(14(n-1) - 8 \log n)\tau_r + (2 \log^2 n + \log n)(\tau_c + 2\tau_I)$. Hence, our adaptation for row-major order is almost as fast as that of Thompson and Kung [9] for shuffled row-major order. Our adaptation is, of course, faster than that of Orcutt [7] by a factor of $0(\log n)$. However, the algorithm uses more routes and interchanges than does the s^2-way merge algorithm of [9] followed by an odd-even transposition sort. The importance of the algorithm developed here lies in the fact that bitonic sort is faster than s^2-way merge sort for $n \leq 512$ [9]. Hence, while s^2-way merge followed by an odd-even transposition sort will be faster than our algorithm for large n, it will not be so for smaller (and perhaps more practical) values of n. Secondly, [9] states that the row-major indexing scheme is "decidedly" nonoptimal for bitonic sort. Our adaptation shows that this statement is inaccurate. Finally, it is worth noting that every sorting algorithm for a mesh connected machine must result in at least $4(n-1)$ routes in the worst case [9]. Hence, our algorithm (as well as those of [9]) is optimal to within a constant factor. Gentleman [5] proves lower bounds for matrix multiplication on a machine model similar to that used here.

In Section II, we present our algorithm, and also specify the SIGN function to be used in comparison-interchange. In Section III, we extend our algorithm to the case of a j-dimensional array processor.

II. Row-Major Bitonic Sort

Our row-major bitonic sort algorithm is specified as a series of subalgorithms in algorithmic notation. In analyzing the algorithms, we shall count only N_r, N_I, and N_c which are respectively the number of routes, register interchanges, and comparison-interchange steps. The analysis will assume that the number of elements involved is a power of 2. All logarithms throughout this paper are in base 2.

A. Row Merge

Our first subalgorithm, ROW_MERGE(K), sorts a bitonic sequence of size K. The K elements are in K adjacent processors on one row of the $n \times n$ array.

procedure ROW_MERGE(K)
1) Let $P_1, \cdots, P_K$ be the processors corresponding to the elements
2) **if** $K = 1$ **then** return
3) shift elements from $P_{K/2+1}, \cdots, P_K$ respectively to $P_1, \cdots, P_{K/2}$
4) perform a comparison-interchange on $P_1, \cdots, P_{K/2}$
5) shift rejected elements from $P_1, \cdots, P_{K/2}$ respectively to $P_{K/2+1}, \cdots, P_K$.
6) Invoke in parallel, ROW MERGE($K/2$) for $P_1, \cdots, P_{K/2}$ and $P_{K/2+1}, \cdots, P_K$ (note that this is not a recursive call but simply *a* **go to** step 1 with K updated).

end ROW_MERGE

The analysis for ROW_MERGE is

$$N_r^R(K) = \begin{cases} K + N_r^R(K/2), & \text{if } K > 1 \\ 0, & \text{if } K = 1 \end{cases}$$

$$N_c^R(K) = \begin{cases} 1 + N_c^R(K/2), & \text{if } K > 1 \\ 0, & \text{if } K = 1. \end{cases}$$

If we assume all elements to initially and finally be in the routing registers then, preceding Step 3, elements in $P_1, \cdots, P_{K/2}$ have to be transferred to register R_s. Following Step 5, elements from R_s in $P_1, \cdots, P_{K/2}$ have to be transferred to R_r. Hence

$$N_I^R(K) = 2N_c^R(K).$$

Solving these recurrences, we get (recall K is a power of 2)

$$N_r^R(K) = 2K - 2; \; N_c^R(K) = \log K$$

and

$$N_I^R(K) = 2 \log K.$$

B. Column Merge

Procedure COLUMN_MERGE(K) is identical to ROW MERGE(K) except that it sorts a bitonic sequence of K elements which are in K adjacent processors on one column of the $n \times n$ array. The analysis is identical to that for ROW MERGE. We shall use $N_r^C(K)$, $N_c^C(K)$, and $N_I^C(K)$ to denote the counts.

C. Vertical Merge

Procedure VERTICAL_MERGE(J, K) sorts into either nonincreasing or nondecreasing row-major order a $J \times K$ array which is made up of two vertically aligned $J/2 \times K$ arrays. One of these is in nondecreasing row-major order and the other is in nonincreasing row-major order.

procedure VERTICAL MERGE(J, K)
1) **for** all columns in parallel **do**
COLUMN_MERGE(J);

2) **for** all rows in parallel **do**
ROW_MERGE(K);
end VERTICAL_MERGE

An example of vertical merge is illustrated in Fig. 1. An arrow indicates a compare-interchange. The head of an arrow points to the processor which retains the larger element.

The correctness of VERTICAL_MERGE may be established by considering the sequence of comparison-interchanges that take place during the bitonic sort of a bitonic sequence $X = (x_1, x_2, \cdots, x_p)$. Unfolding the recursion, we see that if p is a power of 2 then comparison-interchanges take place in the order

compare-interchange elements $p/2$ apart
compare-interchange elements $p/4$ apart
compare-interchange elements $p/8$ apart
$\vdots$
compare-interchange elements 1 apart.

VERTICAL MERGE begins with a bitonic sequence of $J * K$ elements in row-major order. If we look at Step 1 then the following sequence of comparison-interchanges takes place:

compare-interchange elements $JK/2$ apart
compare-interchange elements $JK/4$ apart
$\vdots$
compare-interchange elements K apart.

Finally, in Step 2 the following sequence is performed:

compare-interchange elements $K/2$ apart
compare-interchange elements $K/4$ apart
$\vdots$
compare-interchange elements 1 apart.

Hence, VERTICAL_MERGE is identical to bitonic sort and so must correctly sort the $J \times K$ bitonic array. The analysis for VERTICAL_MERGE is

$$N_r^V(J, K) = N_r^C(J) + N_r^R(K) = 2(J + K) - 4$$
$$N_c^V(J, K) = N_c^C(J) + N_c^R(K) = \log (JK)$$
$$N_I^V(J, K) = N_I^C(J) + N_I^R(K) = 2 \log (JK).$$

D. Horizontal Merge

In this section we give an algorithm to sort a $J \times K$ array which is made up of two horizontally aligned and adjacent $J \times K/2$ arrays. One of these is already sorted in nondecreasing row-major order while the other is in nonincreasing row-major order. But first we give an algorithm, TWO COLUMN_MERGE, to sort a bitonic sequence $< a_0, a_1, \cdots, a_{2J-1} >$ initially loaded in a column of J processors $< P_0, P_1, \cdots, P_{J-1} >$ such that P_i contains a_i and a_{i+J}, $0 \le i < J$. If the sorted sequence is $< b_0, b_1, \cdots, b_{2J-1} >$ then at termination, processor P_i contains elements b_{2i} and b_{2i+1}.

procedure TWO COLUMN_MERGE(J)
1) Let $P_0, P_1, \cdots, P_{J-1}$ be the J processors
2) Compare-interchange the elements in each processor

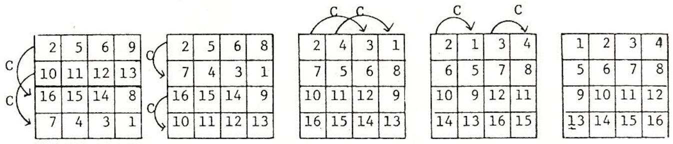


Fig. 1. Vertical merge of two 2×4 arrays.

3) **if** $J > 1$ **then**
a) Exchange the rejected elements of $P_0, \cdots, P_{J/2-1}$ with the accepted elements of $P_{J/2}, \cdots, P_{J-1}$;
b) In parallel perform TWO_COLUMN_MERGE($J/2$) on the processors $P_0, \cdots, P_{J/2-1}$ and $P_{J/2}, \cdots, P_{J-1}$
end TWO_COLUMN_MERGE

Fig. 2 illustrates the working of TWO COLUMN MERGE on an example.

The correctness of TWO_COLUMN MERGE may be established using an argument similar to that used for VERTICAL MERGE. Analyzing the number of steps, we obtain

$$N_r^T(J) = \begin{cases} J + N_r^T(J/2) & \text{if } J > 1 \\ 0 & \text{if } J = 1 \end{cases}$$
$$= 2J - 2$$
$$N_c^T(J) = \begin{cases} 1 + N_c^T(J/2) & \text{if } J > 1 \\ 1 & \text{if } J = 1 \end{cases}$$
$$= 1 + \log J.$$

Register interchanges are needed in Step 3a) to exchange rejected and accepted elements. This can be done with three register interchanges: first move the rejected elements on $P_{J/2}, \cdots, P_{J-1}$ to R_t; next route the rejected elements from $P_0, \cdots, P_{J/2-1}$ to $P_{J/2}, \cdots, P_{J-1}$; now interchange between R_s and R_r on $P_{J/2}, \cdots, P_{J-1}$; route from $P_{J/2}, \cdots, P_{J-1}$ to $P_0, \cdots, P_{J/2-1}$; finally move from R_t to R_r on $P_{J/2}, \cdots, P_{J-1}$. Hence

$$N_I^T(J) = \begin{cases} 3 + N_I^T(J/2) & \text{if } J > 1 \\ 0 & \text{if } J = 1 \end{cases}$$
$$= 3 \log J.$$

We are now ready for the horizontal merge algorithm.

procedure HORIZONTAL_MERGE(J, K)
1) Let the K columns be $C_1, C_2, \cdots, C_K$
2) Move in parallel elements from the J processors in each of the columns $C_{K/2+1}, \cdots, C_K$ to the corresponding processors in the columns $C_1, C_2, \cdots, C_{K/2}$ respectively
3) For each of the columns $C_1, C_2, \cdots, C_{K/2}$ perform in parallel, TWO_COLUMN_MERGE(J)
4) Move, in parallel, the rejected elements back to the processors in $C_{K/2+1}, \cdots, C_K$
5) **if** $K > 2$ **then** invoke in parallel ROW_MERGE($K/2$) for each of the $2J$ rows of size $K/2$ // note: $2J$ rows, each containing $K/2$ adjacent processors, are obtained by splitting each of the original J rows into two. //
end HORIZONTAL MERGE

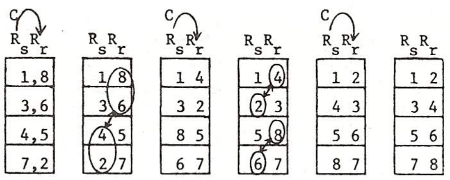


Fig. 2. Two-column merge (performed in *one* column of processors).

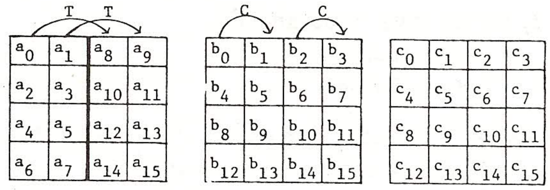


Fig. 3. Horizontal merge of two 4 × 2 arrays ("*T*" = Two-column Merge; "*C*" = Compare-Interchange).

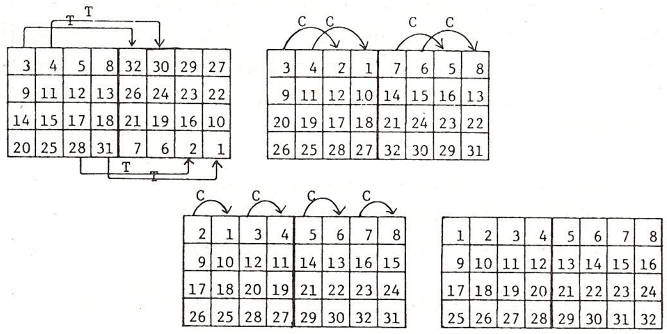


Fig. 4. Horizontal Merge of two 4 × 4 arrays.

Figs. 3 and 4 illustrate the working of HORIZONTAL_MERGE. The correctness of the algorithm follows from an argument similar to that used for VERTICAL_MERGE.

The number of routing steps is given by

$$N_r^H(J, K) = K/2 + N_r^T(J) + K/2 + N_r^R(K/2)$$
$$= 2(J + K) - 4.$$

For the number of comparison-interchanges, we get

$$N_c^H(J, K) = N_c^T(J) + N_c^R(K/2)$$
$$= \log (JK).$$

The number of register interchanges is

$$N_I^H(J, K) = N_I^T(J) + N_I^R(K/2) + 2$$

where the 2 comes from Steps 2 and 4.

Substituting, we get

$$N_I^H(J, K) = 3 \log J + 2 \log K.$$

E. The Main Procedure

Having defined the subalgorithms, we now give the main procedure, SORT, that will sort n^2 elements into nondecreasing row-major order. This algorithm also defines a pass number S which will be used (as explained in the next section) to determine how comparison-interchanges are to be performed.

procedure SORT(n, n)
  1) $K \leftarrow S \leftarrow 1$
  2) **while** $K < n$ **do**
  a) Consider the $n \times n$ processor array as composed of many adjacent $K \times 2K$ subarrays
  b) **do** in parallel for each $K \times 2K$ array HORIZONTAL_MERGE$(K, 2K)$
  c) $S \leftarrow S + 1$
  d) Consider the $n \times n$ processor array as composed of many adjacent $2K \times 2K$ subarrays
  e) **do** in parallel for each $2K \times 2K$ array VERTICAL_MERGE$(2K, 2K)$
  f) $S \leftarrow S + 1$; $K \leftarrow 2 * K$
  end
end SORT

The total number of routing steps is

$$N_r^S(n, n) = N_r^S(n/2, n/2) + N_r^H(n/2, n) + N_r^V(n, n)$$
$$= N_r^S(n/2, n/2) + 7n - 8, \; n > 1$$

and

$$N_r^S(1, 1) = 0.$$

Hence, $N_r^S(n, n) = 14(n - 1) - 8 \log n$.

The number of comparison-interchanges is

$$N_c^S(n, n) = N_c^S(n/2, n/2) + N_c^H(n/2, n) + N_c^V(n, n)$$
$$= N_c^S(n/2, n/2) + 4 \log n - 1$$
$$= 2 \log^2 n + \log n.$$

The number of register-interchanges is

$$N_I^S(n, n) = N_I^S(n/2, n/2) + N_I^H(n/2, n) + N_I^V(n, n)$$
$$= N_I^S(n/2, n/2) + 5 \log n - 3 + 4 \log n$$
$$= N_I^S(n/2, n/2) + 9 \log n - 3$$
$$= 4.5 \log^2 n + 1.5 \log n < 2.25\, N_c^S(n, n).$$

F. The SIGN *Function*

In order for procedure SORT to work correctly, it is necessary that the $K \times 2K$ and $2K \times 2K$ subarrays being sorted in Steps 2b) and 2e) satisfy the initial conditions of HORIZONTAL_MERGE and VERTICAL_MERGE, respectively. In order to meet these conditions, it is necessary to sort some of the subarrays into nonincreasing order and others into nondecreasing order. The order into which a subarray gets sorted is determined by the SIGN function used during a comparison-interchange. Recall that the comparison-interchange instruction was defined in Section I to be

If SIGN$(I, S) * (R_r - R_s) < 0$ **then** interchange (R_r, R_s).

If during the sort of a $K \times 2K$ (or $2K \times 2K$) subarray SIGN is $+1$ for all processors on which comparison-interchanges are made, then the $K \times 2K$ (or $2K \times 2K$) subarray will be sorted into nondecreasing order. If the SIGN is -1, then the subarray will be sorted into nonincreasing order. One may easily verify that the following SIGN function will serve our purpose:

procedure SIGN(SI, S)
  //SI = shuffled row-major index of processor
    (as explained in Section I)//
  //S = pass number defined in SORT//
  If $\lfloor SI/2^S \rfloor$ is even **then return** $(+1)$
    else return (-1)
end SIGN

Thus, each processor can determine the SIGN for its comparison-interchange if "S" is broadcast to all processors. Fig. 5 illustrates the working of procedure SORT on a 4×4 mesh-connected computer. The "T" operation, when $S = 3$, represents a two-column merge. Four pairs of 2×1 columns are merged in parallel.

III. Extensions

Procedure SORT is easily modified to sort into snake-like row-major order. Only the SIGN function needs to be changed for the last VERTICAL_MERGE, i.e., the call VERTICAL MERGE (n, n). During this call, SIGN is to be altered only when ROW_MERGE is invoked from Step 2 of VERTICAL_MERGE. This alteration is such that the SIGN for odd rows becomes -1 and remains $+1$ for even rows (recall rows are indexed 0 through $n-1$).

Following along the lines of Thompson and Kung [9], we may extend our row-major bitonic sort to the case of a j-dimensional array processor. We now have $N = n^j$ processors arranged as in a $n \times n \times \cdots \times n$ j-dimensional array. Each processor is connected to all of its neighbors. As before, the number of elements to be sorted is N and each processor is assumed to have three registers. For this extension, we define LINEAR_MERGE(i, K) to be identical to ROW_MERGE(K) or COLUMN_MERGE(K) except that the K elements are on the ith axis of the j-dimensional array, $1 \le i \le j$. We also define TWO_COLUMN_MERGE(i, K) to be the same as the corresponding algorithm for a two-dimensional array except the "column" is now the ith axis. When $K = 1$, this algorithm is modified to do nothing. This will avoid redundant comparisons in the following algorithm. Procedure MERGE will merge along the ith axis two subarrays of size $K_j \times \cdots \times K_i/2 \times \cdots \times K_1$ to result in an array of size $K_j \times \cdots \times K_i \times \cdots \times K_1$ sorted in row-major.

procedure MERGE(i, $K_j, \cdots, K_i, \cdots, K_1$)
1) Move elements from the second subarray to corresponding processors in the first subarray
2) **for** $A = j, j-1, \cdots, i+1$ **do**
   TWO_COLUMN_MERGE(A, K_A)
3) compare-interchange elements
4) move rejected elements back to corresponding processors in the second subarray
5) LINEAR_MERGE(i, $K_i/2$)
6) **for** $A = i-1, i-2, \cdots, 1$ **do**
   LINEAR_MERGE(A, K_A)

end MERGE

Note that the "**for**" loops of Steps 2 and 6 are done sequentially for each value of A. One may verify that for $j = 2$, procedure MERGE reduces to HORIZONTAL_MERGE when $i = 1$ and to VERTICAL_MERGE when $i = 2$. The number of routing steps is

$$N_r^M(K_j, \cdots, K_1) = 2(K_1 + K_2 + \cdots + K_j) - 2j.$$

The sorting algorithm for $N = n^j$, then, is recursively defined as

procedure JSORT(n^j)
1) JSORT($n^j/2^j$);
2) MERGE($1, n/2, n/2, \cdots, n/2, n$)
3) MERGE($2, n/2, \cdots, n/2, n, n$)
  ⋮
$j+1$) MERGE($j, n, n, \cdots, n$)

end JSORT

The "sign" of comparison is determined by a simple extension of the method of Section II-*F*.

The total number of routing steps will be

$$N_r^J(n^j) = N_r^J(n^j/2^j) + \sum_{i=1}^{j} (2ni + n(j-i)) - 2j^2$$

which gives

$$N_r^J(n^j) = (3j^2 + j)(n-1) - 2j \log N.$$

(This is the same number of routing steps as in [9] for the shuffled indexing.)

The number of compare-interchange steps N_c^J invariant to the interconnection scheme, is the same as for SORT. And, the number of register-interchange steps N_I^J will still be less than $3N_c^J$.

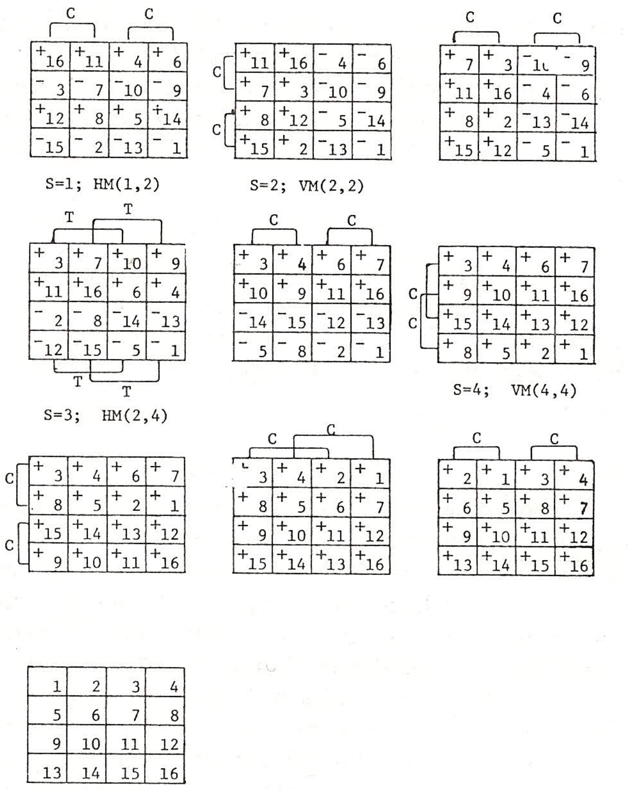


Fig. 5. A complete example of sorting a 4×4 array.

It is interesting to consider the case of maximal connectivity (with respect to the bitonic sort algorithm), where $N = 2^{\log N}$ processors are interconnected ($\log N$)-dimensionally. Then, each processor would be connected to exactly $\log N$ other processors. Upon substituting $n = 2$ and $j = \log N$, the number of routing steps is reduced to

$$N_r^J = \log^2 N + \log N = 2N_c^J.$$

This is as expected as every pair of processors involved in a compare-interchange will be adjacent. Stone's "perfect-shuffle" network [7] also sorts N elements in $O(\log^2 N)$ time. His network uses a far smaller processor connectivity than $\log N$.

IV. Conclusions

We have shown that bitonic sort can be adapted to sort n^2 elements into row-major order on an $n \times n$ mesh-connected computer in $O(n)$ time. This is an improvement over Orcutt's [7] adaptation which requires $O(n \log n)$ time. Our algorithm makes the same number of routes and comparison-interchanges as does that of Thompson and Kung [9]. Their algorithm, however obtains a shuffled row-major order. Thus, row-major order is not "decidedly" nonoptimal for bitonic sort as claimed in [9]. Our algorithm needs about 12.5 percent more register-interchanges than does that of [9]. These are, however, much cheaper than "routes." Our algorithm for row-major bitonic sort can be extended to snake-like row-major ordering and also to sorting on a j-dimensionally connected computer. In the latter case, the algorithm requires as many routes and comparison-interchanges as does that of [9].

References

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

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

[3] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers," Carnegie-Mellon Univ., Pittsburgh, PA, Dep. Comput. Sci. Rep., May 1975, *IEEE Trans. Comput.*, to be published.

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

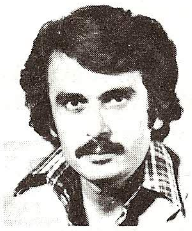
[5] W. M. Gentleman, "Some complexity results for matrix computations on parallel processors," *J. Ass. Comput. Mach.*, vol. 25(1), p.p. 112–115, 1978.

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

[7] S. E. Orcutt, "Computer organization and algorithms for very-high speed computations," Ph.D. dissertation, Stanford Univ., Stanford, CA, Sept. 1974, ch. 2.

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

[9] 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.



David Nassimi was born in Iran on December 22, 1944. He received the B.S. degree in electrical engineering and the M.S. degrees in electrical engineering and computer science, in 1968, 1978, and 1978, respectively, all from the University of Minnesota, Minneapolis.

During the period 1968–1972, he was employed by Control Data Corporation where he developed programs for computer systems diagnosis. He is currently completing the Ph.D. degree in computer science at the University of Minnesota. He has held research assistantships in several areas, including computer-aided instruction, simulation of computer networks, and parallel computations. His Ph.D. dissertation is on the design and analysis of parallel algorithms.

Sartaj Sahni was born in Poona, India, on July 22, 1949. He received the B.Tech (elec. eng.) degree from the Indian Institute of Technology, Kanpur, India, in 1970, and the M.S. and Ph.D. degrees in computer science from Cornell University, Ithaca, NY, in 1972 and 1973, respectively.

He is currently an Associate Professor of Computer Science at the University of Minnesota, Minneapolis. He has published in *JACM*, *JCSS*, *SIAM Journal on Computing*, IEEE Transactions on Computers, *ACM Transactions on Mathematical Software*, *Operations Research*, *International Journal on Theoretical Computer Science*, *Mathematics of Operations Research*, and the *Journal of Statistical Computation and Simulation*. His publications are on topics concerned with the design and analysis of efficient algorithms. He is also the coauthor of *Fundamentals of Data Structures* (Computer Science Press, 1976) and *Fundamentals of Computer Algorithms* (Computer Science Press, 1978).