and $K2$ are related as

$$h_0(K1) + N/2 = h_0(K2),$$

the sequence for $K1$ traces the same positions as the sequence for $K2$ only after one-half of the table positions have been searched. By considering the length of search, the author has drawn the conclusion that the method is practically free from "primary clustering." I would like to suggest the following modification for generating the key sequence. The method is as follows:

For any table of size $N = 2^n$, half the key indices can be obtained from a complementary relation,

$$h_i' = (N - 1) - h_i, \quad i = 0, 1, 2, \ldots, ((N/2) - 1).$$

Instead of going through a sequence of

$$i = 1, 2, 3, \ldots, N - 1$$

for generating the hash addresses from relation (4) in Luccio's method, we can generate h_i' as next key address once h_i is calculated and all the positions will be visited by using the relation (4) only $((N/2) - 1)$ times. For example, if we take $N = 16$, the sequence of generation for different initial hash functions for two typical cases will be as follows:

Initial function $h_0 = (K) \bmod N$	Key index sequence for table size = 16 $h_i \rightarrow i = 0, 1, 2, \ldots 15$
0	0 15 1 14 2 13 3 12 4 11 5 10 6 9 7 8
8	8 7 9 6 10 5 11 4 12 3 13 2 14 1 15 0

It is thus found that even if $K1$ and $K2$ are related as $h_0(K1) + N/2 = h_0(K2)$, the sequence does not trace the same positions and hence the method is fully free from primary clustering. Generation of h_i' will involve one subtraction in the main loop since $(N - 1)$ is calculated outside the loop. Hence apart from making the sequence of indices free from primary clustering, this modification will also make the computation faster. Radke [2] has used the concept of complementary relations for probing all the positions of the table.

Received November 1975; revised July 1976

References

1. Fabrizio, L. Weighted increment linear search for scatter tables. *Comm. ACM 15*, 12 (Dec. 1972), 1045–1047.
2. Radke, C.E. The use of quadratic residue research. *Comm. ACM 13*, 2 (Feb. 1970), 103–105.

Author's address: Aeronautical Development Establishment, Bangalore 560001, India.

Programming Techniques
G. Manacher, S. L. Graham Editors

Sorting on a Mesh-Connected Parallel Computer

C.D. Thompson and H.T. Kung
Carnegie-Mellon University

Two algorithms are presented for sorting n^2 elements on an $n \times n$ mesh-connected processor array that require O (n) routing and comparison steps. The best previous algoritmhm takes time O(n log n). The algorithms of this paper are shown to be optimal in time within small constant factors. Extensions to higher-dimensional arrays are also given.

Key Words and Phrases: parallel computer, parallel sorting, parallel merge, routing and comparison steps, perfect shuffle. processor interconnection pattern

CR Categories: 4.32, 5.25, 5.31

1. Introduction

In the course of a parallel computation, individual processors need to distribute their results to other processors and complicated data flow problems may arise. One way to handle this problem is by sorting "destination tags" attached to each data element [2]. Hence efficient sorting algorithms for parallel machines with some fixed processor interconnection pattern are relevant to almost any use of these machines.

In this paper we present two algorithms for sorting $N = n^2$ elements on an $n \times n$ mesh-type processor array that require $O(n)$ unit-distance routing steps and $O(n)$

This research was supported in part by the National Science Foundation under Grant MCS75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422.

Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

comparison steps (n is assumed to be a power of 2). The best previous algorithm takes time $O(n \log n)$ [7]. One of our algorithms, the s^2-way merge sort, is shown optimal within a factor of 2 in time for sufficiently large n, if one comparison step takes no more than twice the time of a routing step. Our other $O(n)$ algorithm, an adaptation of Batcher's bitonic merge sort, is much less complex but optimal under the same assumption to within a factor of 4.5 for *all* n, and is more efficient for moderate n.

We believe that the algorithms of this paper will give the most efficient sorting algorithms for ILLIAC IV-type parallel computers.

Our algorithms can be generalized to higher-dimensional array interconnection patterns. For example, our second algorithm can be modified to sort N elements on a j-dimensionally mesh-connected N-processor computer in $O(N^{1/j})$ time, which is optimal within a small constant factor.

Efficient sorting algorithms have been developed for interconnection patterns other than the "mesh" considered in this paper. Stone [8] maps Batcher's bitonic merge sort onto the "perfect shuffle" interconnection scheme, obtaining an N-element sort time of $O(\log^2 N)$ on N processors. The odd-even transposition sort (see Appendix) requires an optimal $O(N)$ time on a linearly connected N-processor computer. Sorting time is thus seen to be strongly dependent on the interconnection pattern. Exploration of this dependence for a given problem is of interest from both an architectural and an algorithmic point of view.

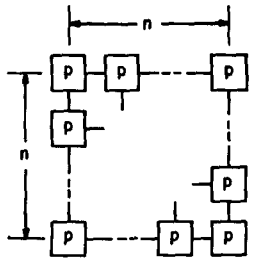
In Section 2 we give the model of computation. The sorting problem is defined precisely in Section 3. A lower bound on the sorting time is given in Section 4. Batcher's 2-way odd-even merge is mapped on our 2-dimensional mesh-connected processor array in the next section. Generalizing the 2-way odd-even merge, we introduce a $2s$-way merge algorithm in Section 6. This is further generalized to an s^2-way merge in Section 7, from which our most efficient sorting algorithm for large n is developed. Section 8 shows that Batcher's bitonic sort can be performed efficiently on our model by choosing an appropriate processor indexing scheme. Some extensions and implications of our results are discussed in Section 9. The Appendix contains a description of the odd-even transposition sort.

2. Model of Computation

We assume a parallel computer with $N = n \times n$ identical processors. The architecture of the machine is similar to that of the ILLIAC IV [1]. The major assumptions are as follows:

(i) The interconnections between the processors are a subset of those on the ILLIAC IV, and are defined by the two dimensional array shown in Figure 1, where the p's denote the processors. That is, each processor is connected to all its neighbors. Processors at the perimeter have two or three rather than four neighbors; there are no "wrap-around" connections as found on the ILLIAC IV.

Fig. 1.



The bounds obtained in this paper would be affected at most by a factor of 2 if "wrap-around" connections were included, but we feel that this addition would obscure the ideas of this paper without substantially strengthening the results.

(ii) It is a SIMD (*S*ingle *I*nstruction stream *M*ultiple *D*ata stream) machine [4]. During each time unit, a single instruction is broadcast to all processors, but only executed by the set of processors specified in the instruction. For the purpose of the paper, only two instruction types are needed: the routing instruction for interprocessor data moves, and the comparison instruction on two data elements in each processor. The comparison instruction is a conditional interchange on the contents of two registers in each processor. Actually, we need both "types" of such comparison instructions to allow either register to receive the minimum; normally both types will be issued during "one comparison step."

(iii) Define t_R = time required for one unit-distance routing step, i.e. moving one item from a processor to one of its neighbors, t_C = time required for one comparison step. Concurrent data movement is allowed, so long as it is all in the same direction; also any number (up to N) of concurrent comparisons may be performed simultaneously. This means that a comparison-interchange step between two items in adjacent processors can be done in time $2t_R + t_C$ (route left, compare, route right). A number of these comparison-interchange steps may be performed concurrently in time $2t_R + t_C$ if they are all between distinct, vertically adjacent processors. A mixture of horizontal and vertical comparison-interchanges will require at least time $4t_R + t_C$.

3. The Sorting Problem

The processors may be indexed by any function that is a one-to-one mapping from $\{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$ onto $\{0, 1, \ldots, N-1\}$. Assume that N elements from a linearly ordered set are initially loaded in the N processors, each receiving exactly one ele-

Fig. 2.

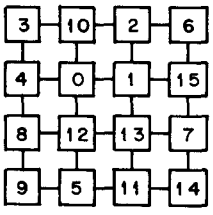


Fig. 3.

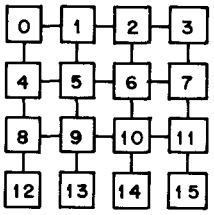


Fig. 4.

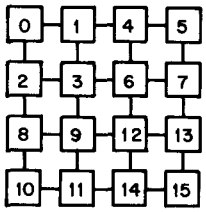
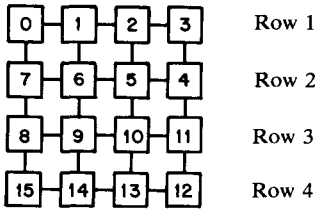


Fig. 5.



ment. With respect to any index function, the sorting problem is defined to be the problem of moving the jth smallest element to the processor indexed by j for all $j = 0, 1, \ldots, N - 1$.

Example 3.1

Suppose that $n = 4$ (hence $N = 16$) and that we want to sort 16 elements initially loaded as shown in Figure 2. Three ways of indexing the processors will be considered in this paper.

(i) *Row-major indexing.* After sorting we have the indexing shown in Figure 3.

(ii) *Shuffled row-major indexing.* After sorting we have the indexing shown in Figure 4. Note that this indexing is obtained by shuffling the binary representation of the row-major index. For example, the row-major index 5 has the binary representation 0101. Shuffling the bits gives 0011 which is 3. (In general, the shuffled binary number, say, "abcdefgh" is "aebfcgdh".)

(iii) *Snake-like row-major indexing.* After sorting we have the indexing shown in Figure 5. This indexing is obtained from the row-major indexing by reversing the ordering in even rows.

The choice of a particular indexing scheme depends upon how the sorted elements will be used (or accessed), and upon which sorting algorithm is to be used. For example, we found that the row-major indexing is poor for merge sorting.

It is clear that the sorting problem with respect to any index scheme can be solved by using the routing and comparison steps. We are interested in designing algorithms which minimize the time spent in routing and comparing.

4. A Lower Bound

Observe that for any index scheme there are situations where the two elements initially loaded at the opposite corner processors have to be transposed during the sorting (Figure 6).

It is easy to argue that even for this simple transposition we need at least $4(n - 1)$ unit-distance routing steps. This implies that no algorithm can sort n^2 elements in time less than $O(n)$. In this paper, we shall show two algorithms which can sort n^2 elements in time $O(n)$. One will be developed in Sections 5 through 7, the other in Section 8.

A similar argument leads to an $O(n)$ lower bound for the multiplication or inversion of $n \times n$ matrices on a mesh-connected computer of n^2 processing elements [5].

5. The 2-Way Odd-Even Merge

Batcher's odd-even merge [2, 6] of two sorted sequences $\{u(i)\}$ and $\{v(i)\}$ is performed in two stages. First, the "odd sequences" $\{u(1), u(3), u(5), \ldots, u(2j + 1), \ldots\}$ and $\{v(1), v(3), \ldots, v(2j + 1), \ldots\}$ are merged concurrently with the merging of the "even sequences" $\{u(2), u(4), \ldots, u(2j), \ldots\}$ and $\{v(2), v(4), \ldots, v(2j), \ldots\}$. Then the two merged sequences are interleaved, and a single parallel comparison-interchange step produces the sorted result. The merges in the first stage are done in the same manner (that is, recursively).

We first illustrate how the odd-even method can be performed efficiently on linearly connected processors, then the idea is generalized to 2-dimensionally connected arrays. If two sorted sequences {1, 3, 4, 6} and {0, 2, 5, 7} are initially loaded in eight linearly connected processors, then Batcher's odd-even merge can be diagrammed as shown in Figure 7.

Step L3 (p. 266) is the "perfect shuffle" [8] and step L1 is its inverse, the "unshuffle." Note that the perfect shuffle can be achieved by using the triangular interchange pattern shown in Figure 8, where the double-headed arrows indicate interchanges. Similarly, an inverted triangular interchange pattern will do the unshuffle. Therefore both the perfect shuffle and unshuffle can be done in $k - 1$ interchanges (i.e. $2k - 2$ routing steps) when performed on a row of length $2k$ in our model.

We now give an implementation of the odd-even merge on a rectangular region of our model. Let $M(j, k)$ denote our algorithm of merging two j by $k/2$ sorted adjacent subarrays to form a sorted j by k array, where j, k are powers of 2, $k > 1$, and all the arrays are arranged in the snake-like row major ordering. We first consider the case where $k = 2$. If $j = 1$, a single comparison-interchange step suffices to sort the two

unit "subarrays". Given two sorted columns of length $j > 1$, $M(j, 2)$ consists of the following steps:

J1. Move all odds to the left column and all evens to the right. Time: $2t_R$.
J2. Use the "odd-even transposition sort" (see Appendix) to sort each column. Time: $2jt_R + jt_C$.
J3. Interchange on even rows. Time: $2t_R$.
J4. One step of comparison-interchange (every "even" with the next "odd"). Time: $2t_R + t_C$.

Figure 9 illustrates the algorithm $M(j, 2)$ for $j = 4$.

For $k > 2$, $M(j, k)$ is defined recursively in the following way. Steps M1 and M2 unshuffle the elements, step M3 recursively merges the "odd sequences" and the "even sequences," steps M4 and M5 shuffle the "odds" and "evens" together, and step M5 performs the final comparison-interchange. The algorithm $M(4, 4)$ is given in Figure 10, where the two given sorted 4 by 2 subarrays are initially stored in 16 processors as shown in the first figure.

Let $T(j, k)$ be the time needed by $M(j, k)$. Then we have

$$T(j, 2) = (2j + 6)t_R + (j + 1)t_C,$$

and for $k > 2$,

$$T(j, k) = (2k + 4)t_R + t_C + T(j, k/2).$$

These imply that

$$T(j, k) \leq (2j + 4k + 4 \log k)t_R + (j + \log k)t_C.$$

(All logarithms in this paper are taken to base 2.)

An $n \times n$ sort may be composed of $M(j, k)$ by sorting all columns in $O(n)$ routes and compares by, say, the odd-even transposition sort, then using $M(n, 2), M(n, 4), M(n, 8), \ldots, M(n, n)$, for a total of $O(n \log n)$ routes and compares. This poor performance may be assigned to two inefficiencies in the algorithm. First, the recursive subproblems ($M(n, n/2), M(n, n/4), \ldots, M(n, 1)$) generated by $M(n, n)$ are not decreasing in size along both dimensions: they are all $O(n)$ in complexity. Second, the method is extremely "local" in the sense that no comparisons are made between elements initially in different halves of the array until the last possible moment, when each half has already been independently sorted.

The first inefficiency can be attacked by designing an "upwards" merge to complement the "sideways" merge just described. Even more powerful is the idea of combining many upwards merges with a sideways one. This idea is used in the next section.

6. The 2s-Way Merge

In this section we give an algorithm $M'(j, k, s)$ for merging $2s$ arrays of size j/s by $k/2$ in a j by k region of our processors, where j, k, s are powers of $2, s \geq 1$, and the arrays are in the snake-like row-major ordering. The algorithm $M'(j, k, s)$ is almost the same as the algorithm $M(j, k)$ described in the previous section,

Fig. 6.

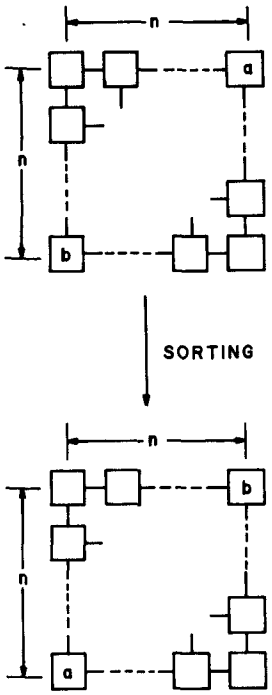
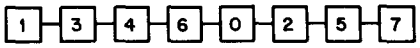
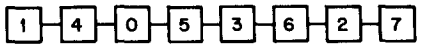


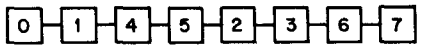
Fig. 7.



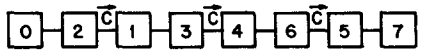
L1. Unshuffle: Odd-indexed elements to left, evens to right.



L2. Merge the "odd sequences" and the "even sequences."



L3. Shuffle.



L4. Comparison-interchange (the C's indicate comparison-interchanges).

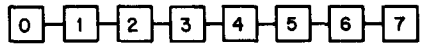
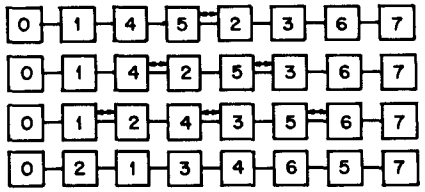


Fig. 8.



except that $M'(j, k, s)$ requires a few more comparison-interchanges during step M6. These steps are exactly those performed in the initial portion of the odd-even transposition sort mapped onto our "snake" (see Appendix). More precisely, for $k > 2$, M1 and M6 are replaced by

M1'. Single interchange step on even rows if $j > s$, so that columns contain either all evens or all odds. If $j = s$, do nothing: the columns are already segregated. Time: $2t_R$.

M6'. Perform the first $2s - 1$ parallel comparison-interchange steps of the odd-even transposition sort on the "snake." It is not difficult to see that the time needed is at most $s(4t_R + t_C) + (s - 1)(2t_R + t_C) = (6s - 2)t_R + (2s - 1)t_C$.

Note that the original step M6 is just the first step of an odd-even transposition sort. Thus the 2-way merge is seen to be a special case of the $2s$-way merge. Similarly, for $M'(j, 2, s), j > s$, J4 is replaced by M6', which takes time $(2s - 1)(2t_R + t_C)$. $M'(s, 2, s)$ is a special case analogous to $M(1, 2)$, and may be performed by the odd-even transposition sort (see Appendix) in time $4st_R + 2st_C$.

The validity of this algorithm may be demonstrated by use of the 0–1 principle [6]: if a network sorts all sequences of 0's and 1's, then it will sort any arbitrary sequence of elements chosen from a linearly ordered set. Thus, we may assume that the inputs are 0's and 1's. It is easy to check that there may be as many as $2s$ more zeros on the left as on the right after unshuffling (i.e. after step J1 or step M2). After the shuffling, the first $2s - 1$ steps of an odd-even transposition sort suffice to sort the resulting array.

Let $T'(j, k, s)$ be the time required by the algorithm $M'(j, k, s)$. Then we have that

$$T'(j, 2, s) \leq (2j + 4s + 2)t_R + (j + 2s - 1)t_C$$

and that for $k > 2$,

$$T'(j, k, s) \leq (2k + 6s - 2)t_R + (2s - 1)t_C + T'(j, k/2, s).$$

These imply that

$$T'(j, k, s) = (2j + 4k + (6s)\log k + O(s + \log k))t_R + (j + (2s)\log k + O(s + \log k))t_C.$$

For $s = 2$, a merge sort may be derived that has the following time behavior:

$$S'(n, n) = S'(n/2, n/2) + T'(n, n, 2).$$

Thus

$$S'(n, n) = (12n + O(\log^2 n))t_R + (2n + O(\log^2 n))t_C.$$

Suddenly, we have an algorithm that sorts in linear time. In the following section, the constants will be reduced by a factor of 2 by the use of a more complicated multiway merge algorithm.

7. The s^2-Way Merge

The s^2-way merge $M''(j, k, s)$ to be introduced in this section is a generalization of the 2-way merge $M(j, k)$. Input to $M''(j, k)$ is s^2 sorted j/s by k/s arrays in a j by k region of our processors, where j, k, s are powers of 2 and $s > 1$. Steps M1 and M2 still suffice to move odd-indexed elements to the left and evens to the right so long as $j > s$ and $k > s$; $M''(j, s, s)$ is a special case analogous to $M(j, 2)$ of the 2-way merge. Steps M1 and M6 are now replaced by

M1''. Single interchange step on even rows if $j > s$, so that columns contain either all evens or all odds. If $j = s$, do nothing: the columns are already segregated. Time: $2t_R$

M6''. Perform the first $s^2 - 1$ parallel comparison-interchange steps of the odd-even transposition sort on the "snake" (see Appendix). The time required for this is

$$(s^2/2)(4t_R + t_C) + (s^2/2 - 1)(2t_R + t_C) = (3s^2 - 2)t_R + (s^2 - 1)t_C.$$

The motivation for this new step comes from the realization that when the inputs are 0's and 1's, there may be as many as s^2 more zeros on the left half as the right after unshuffling.

$M''(j, s, s), j \geq s$, can be performed in the following way:

N1. $(\log s/2)$ 2-way merges: $M(j/s, 2), M(j/s, 4), \ldots, M(j/s, s/2)$.
N2. A single $2s$-way merge: $M'(j, s, s)$.

If $T''(j, k, s)$ is the time taken by $M''(j, k, s)$, we have for $k = s$,

$$T''(j, s, s) = (2j + O((s + j/s)\log s))t_R + (j + O((s + j/s)\log s))t_C$$

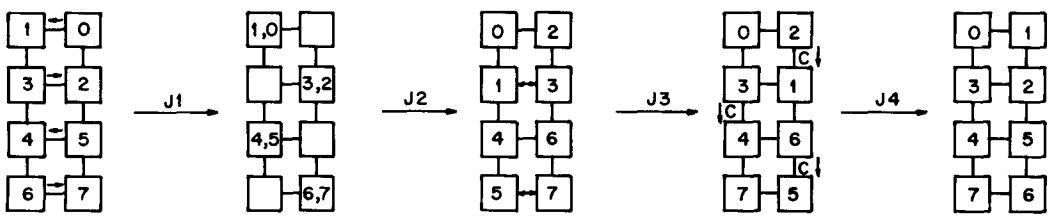
and for $k > s$,

$$T''(j, k, s) = (2k + 3s^2 + O(1))t_R + (s^2 + O(1))t_C + T''(j, k/2, s).$$

Therefore

$$T''(j, k, s) = (4k + 2j + 3s^2\log(k/s) + O((s + j/s)\log s + \log k))t_R + (j + s^2\log(k/s) + O((s + j/s)\log s + \log k))t_C.$$

Fig. 9.



A sorting algorithm may be developed from the s^2-way merge; a good value for s is approximately $n^{1/3}$ (remember that s must be a power of 2). Then the time of sorting $n \times n$ elements satisfies

$$S''(n, n) = S''(n^{2/3}, n^{2/3}) + T''(n, n, n^{1/3}).$$

This leads immediately to the following result.

THEOREM 7.1. *If the snake-like row-major indexing is used, the sorting problem can be done in time:*

$$(6n + O(n^{2/3} \log n))t_R + (n + O(n^{2/3} \log n))t_C.$$

If $t_C \leq 2t_R$, Theorem 7.1 implies that $(6n + 2n + O(n^{2/3} \log n))t_R$ is sufficient time for sorting. In Section 4, we showed that $4(n - 1)t_R$ time is necessary. Thus, for large enough n, the s^2-way algorithm is optimal to within a factor of 2. Preliminary investigation indicates that a careful implementation of the s^2-way merge sort is optimal within a factor of 7 for *all* n, under the assumption that $t_C \leq 2t_R$.

8. The Bitonic Merge

In this section we shall show that Batcher's bitonic merge algorithm [2, 6] lends itself well to sorting on a mesh-connected parallel computer, once the proper indexing scheme has been selected. Two indexing schemes will be considered, the "row-major" and the "shuffled row-major" indexing schemes of Section 3.

The bitonic merge of two sections of a bitonic array of $j/2$ elements each takes $\log j$ passes, where pass i consists of a comparison-interchange between processors with indices differing only in the ith bit of their binary representations. (This operation will be termed "comparison-interchange on the ith bit".) Sorting an entire array of 2^k elements by the bitonic method requires k comparison-interchanges on the 0th bit (the least significant bit), $k - 1$ comparison-interchanges on the first bit, . . . , $(k - i)$ comparison-interchanges on the ith bit, . . . , and 1 comparison-interchange on the most significant bit. For any fixed indexing scheme, in general a comparison-interchange on the ith bit will take a different amount of time than when done on the jth bit; an optimal processor indexing scheme for the bitonic algorithm minimizes the time spent on comparison-interchange steps. A necessary condition for optimality is that a comparison-interchange on the jth bit be no more expensive than the $(j + 1)$-th bit for all j. If this were not the case for some j, then a better indexing scheme could immediately be derived from the supposedly optimal one by interchanging the jth and the $(j + 1)$-th bits of all processor indices (since more comparison-interchanges will be done on the original jth bit than on the $(j + 1)$-th bit).

The bitonic algorithm has been analyzed for the row-major indexing scheme: it takes

$$O(n \log n)t_R + O(\log^2 n)t_C$$

time to sort n^2 elements on n^2 processors (see Orcutt [7]). However, the row-major indexing scheme is decidedly nonoptimal. For the case $n^2 = 64$, processor indices have six bits. A comparison-interchange on bit 0 takes just $2t_R + t_C$, for the processors are horizontally adjacent. A comparison-interchange on bit 1 takes $4t_R + t_C$, since the processors are two units apart. Similarly, a comparison-interchange on bit 2 takes $8t_R + t_C$, but a comparison-interchange on bit 3 takes only $2t_R + t_C$ because the processors are vertically adjacent. This phenomenon may be analyzed by considering the row-major index as the concatenation of a 'Y' and an 'X' binary vector: in the case $n^2 = 64$, the index is $Y_2Y_1Y_0X_2X_1X_0$. A comparison-interchange on X_i takes more time than one on X_j when $i > j$; however, a comparison-interchange on Y_i takes exactly the same time as on X_i. Thus a better indexing scheme may be derived by "shuffling" the 'X' and 'Y' vectors, obtaining (in the case $n^2 = 64$) $Y_2X_2Y_1X_1Y_0X_0$; this "shuffled row-major" scheme satisfies our optimality condition.

Geometrically, "shuffling" the 'X' and 'Y' vectors ensures that all arrays encountered in the merging process are nearly square, so that routing time will not be excessive in either direction. The standard row-major indexing causes the bitonic sort to contend with subarrays that are always at least as wide as they are tall; the aspect ratio can be as high as n on an $n \times n$ processor array.

Programming the bitonic sort would be a little tricky, as the "direction" of a comparison-interchange step depends on the processor index. Orcutt [7] covers these gory details for row-major indexing; his algorithm may easily be modified to handle the shuffled row-major indexing scheme. An example of the bitonic merge sort on a 4×4 processor array for the shuffled row-major indexing is presented below and in Figure 12; the comparison "directions" were derived from Figure 11 (see [6], p. 237).

Stage 1. Merge pairs of adjacent 1×1 matrices by the comparison-interchange indicated. Time: $2t_R + t_C$.

Stage 2. Merge pairs of 1×2 matrices; note that one member of a pair is sorted in ascending order, the other in descending order. This will always be the case in any bitonic merge. Time: $4t_R + 2t_C$.

Stage 3. Merge pairs of 2×2 matrices. Time: $8t_R + 3t_C$.

Stage 4. Merge the two 2×4 matrices. Time: $12t_R + 4t_C$.

Let $T'''(2^i)$ be the time to merge the bitonically sorted elements in processors 0 through $2^i - 1$, where the shuffled row-major indexing is used. Then after one pass of comparison-interchange, which takes time $2^{\lceil i/2 \rceil}t_R + t_C$, the problem is reduced to the bitonic merge of elements in processors 0 through $2^{i-1} - 1$, and that of elements in processors 2^{i-1} to $2^i - 1$. It may be observed that the latter two merges can be done concurrently. Thus we have

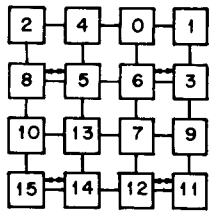
$$T'''(1) = 0, \quad T'''(2^i) = T'''(2^{i-1}) + 2^{\lceil i/2 \rceil}t_R + t_C.$$

Hence

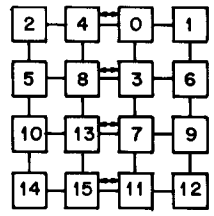
$$T'''(2^i) = (3*2^{(i+1)/2} - 4)t_R + it_C, \text{ if } i \text{ is odd},$$
$$= (4*2^{i/2} - 4)t_R + it_C, \quad \text{if } i \text{ is even}.$$

Let $S'''(2^{2j})$ be the time taken by the corresponding

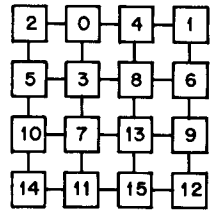
Fig. 10.



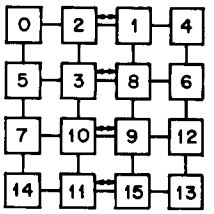
M1. Single interchange step on even rows if $j > 2$, so that columns contain either all evens or all odds. If $j = 2$, do nothing: the columns are already segregated. Time: $2t_R$.



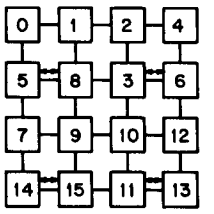
M2. Unshuffle each row. Time: $(k - 2)t_R$.



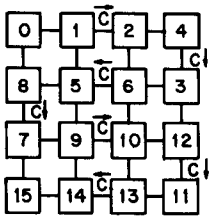
M3. Merge by calling $M(j, k/2)$ on each half. Time: $T(j, k/2)$.



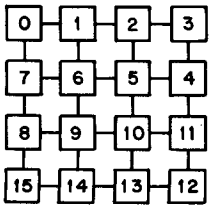
M4. Shuffle each row. Time: $(k - 2)t_R$.



M5. Interchange on even rows. Time: $2t_R$.



M6. Comparison-interchange of adjacent elements (every "even" with the next "odd"). Time: $4t_R + t_C$.



sorting algorithm (for a square array). Then

$$S'''(1) = 0,$$
$$S'''(2^{2j}) = S'''(2^{2j-1}) + T'''(2^{2j})$$
$$= S'''(2^{2(j-1)}) + T(2^{2j}) + T(2^{2j-1}).$$

Hence $S'''(2^{2j}) = (14(2^j - 1) - 8j)t_R + (2j^2 + j)t_C$.

In our model, we have $2^{2j} = N = n^2$ processors, leading to the following theorem.

THEOREM 8.1. *If the shuffled row-major indexing is used, the bitonic sort can be done in time*

$$(14(n - 1) - 8\log n)t_R + (2\log^2 n + \log n)t_C.$$

If $t_C \leq 2t_R$, it may be seen that the bitonic merge sort algorithm is optimal to within a factor of 4.5 for all n (since $4(n - 1)t_R$ time is necessary, as shown in Section 4). Preliminary investigation indicates that the bitonic merge sort is faster than the s^2-way odd-even merge sort for $n \leq 512$, under the assumption that $t_C \leq 2t_R$.

9. Extensions and Implications

In this section the following extensions and implications are presented.

(i) By Theorem 7.1 or 8.1, the elements may be sorted into snake-like row-major ordering or in the shuffled row-major ordering in $O(n)$ time. By the following lemma we know that they can be rearranged to obey any other index function with relatively insignificant extra costs, provided each processor has sufficient memory size.

LEMMA 9.1. *If $N = n \times n$ elements have already been sorted with respect to some index function and if each processor can store n elements, then the N elements can be sorted with respect to any other index function by using an additional $4(n - 1)t_R$ units of time.*

The proof follows from the fact that all elements can be moved to their destinations by four sweeps of $n - 1$ routing steps in all four directions.

(ii) If the processors are connected in a $k \times m$ rectangular array (Figure 13) instead of a square array, similar results can still be obtained. For example, corresponding to Theorem 7.1, we have:

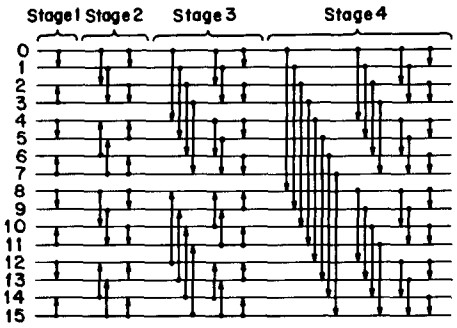
THEOREM 9.1. *If the snake-like row-major indexing is used, the sorting problem for a $k \times m$ processor array (k, m powers of 2) can be done in time*

$$(4m + 2k + O(h^{2/3}\log h))t_R + (k + O(h^{2/3}\log h))t_C,$$

where $h = \min(k, m)$, by using the s^2-way merge sort with $s = O(h^{1/3})$.

(iii) The number of elements to be sorted could be larger than N, the number of processors. An efficient means of handling this situation is to distribute an approximately equal number of elements to each processor initially and to use a merge-splitting operation for each comparison-interchange operation. This idea is discussed by Knuth [6], and used by Baudet and Ste-

Fig. 11.



venson [3]. Baudet and Stevenson's results will be immediately improved if the algorithms of this paper are used, since they used Orcutt's $O(n \log n)$ algorithm.

(iv) Higher-dimensional array interconnection patterns, i.e. $N = n^j$ processors each connected to its $2j$ nearest neighbors, may be sorted by algorithms generalized from those presented in this paper. For example, $N = n^j$ elements may be sorted in time

$$((3j^2 + j)(n - 1) - 2j \log N)t_R + (\tfrac{1}{2})(\log^2 N + \log N)t_C,$$

by adapting Batcher's bitonic merge sort algorithm to the "j-way shuffled row-major ordering." This new ordering is derived from the binary representation of the row-major indexing by a j-way bit shuffle. If $n = 2^3$, $j = 3$, and $Z_2Z_1Z_0Y_2Y_1Y_0X_2X_1X_0$ is a row-major index, then the j-way shuffled index is $Z_2Y_2X_2Z_1Y_1X_1Z_0Y_0X_0$. This formula may be derived in the following way. The t_C term is not dimension-dependent: the same number of comparisons are performed in any mapping of the bitonic sort onto an N processor array. The t_R term is the solution of $\sum_{1 \le i \le \log n} 2^i \sum_{1 \le k \le j} ((\log N) - ij + k)$, where the 2^i term is the cost of a comparison-interchange on the $(i\text{-}1)$th bit of any of the "kth-dimension indices" (i.e. Z_{i-1}, Y_{i-1}, and X_{i-1} when $j=3$ as in the example above). The $((\log N) - ij + k)$ term is the number of times a comparison-interchange is performed on the $(ij\text{-}k)$th bit of the j-way shuffled row-major index during the bitonic sort. Therefore we have the following theorem:

THEOREM 9.2. *If N processors are j-dimensionally mesh-connected, then the bitonic sort can be performed in time $O(N^{1/j})$, using the j-way shuffled row-major index scheme.*

By using the argument of Section 4, one can easily check that the bound in the theorem is asymptotically optimal for large N.

Appendix. Odd-Even Transposition Sort

The odd-even transposition sort [6] may be mapped onto our 2-dimensional arrays with snake-like row-

Fig. 12.

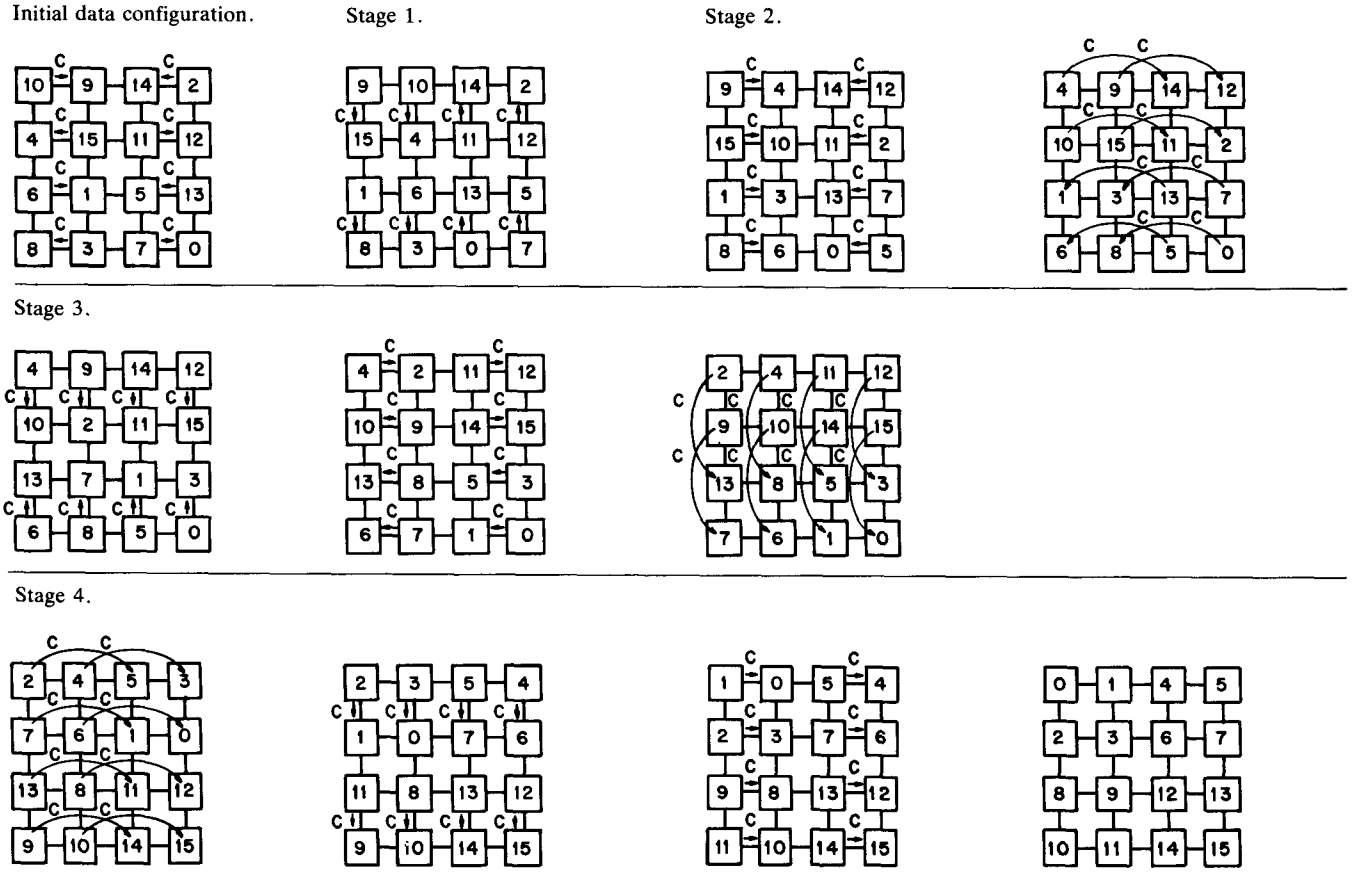
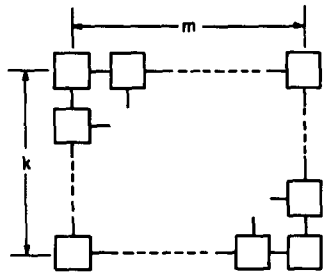


Fig. 13.



major ordering in the following way. Given N processors initially loaded with a data value, repeat $N/2$ times:

O1. "Expensive comparison-interchange" of processors $\#(2i + 1)$ with processors $\#(2i + 2)$, $0 \leq i < N/2 - 1$. Time: $4t_R + t_C$ if processor array has more than two columns and more than one row; 0 if $N = 2$; and $2t_R + t_C$ otherwise.

O2. "Cheap comparison-interchange" of processors $\#(2i)$ with processors $\#(2i + 1)$, $0 \leq i \leq N/2 - 1$. Time: $2t_r + t_C$.

If $T_{oe}(j, k)$ is the time required to sort jk elements in a $j \times k$ region of our processor by the odd-even transposition sort into the snake-like row-major ordering, then

$$T_{oe}(j, k) = \begin{cases} 0, & \text{if } jk = 1 \text{ else} \\ 2t_R + t_C, & \text{if } jk = 2 \text{ else} \\ jk(2t_R + t_C), & \text{if } j = 1 \text{ or } k = 2 \text{ else} \\ jk(3t_R + t_C) & \end{cases}$$

Step J2 of the 2-way odd-even merge (Section 5) cannot be performed by the version of the odd-even transposition sort indicated above. Since N is even here ($N = 2j$), step O2 may be placed before step O1 in the algorithm description above (see Knuth [6]). Now step O2 may be performed in the normal time of $2t_R + t_C$, even starting from the nonstandard initial configuration depicted in Section 5 as the result of step J1.

Received March 1976; revised August 1976

References

1. Barnes, G.H., et al. The ILLIAC IV computer. *IEEE Trans. Comptrs. C-17* (1968), 746–757.
2. Batcher, K.E. Sorting networks and their applications. Proc. AFIPS 1968 SJCC, Vol. 32, AFIPS Press, Montvale, N.J., pp. 307–314.
3. Baudet, G., and Stevenson, D. Optimal sorting algorithms for parallel computers. Comptr. Sci. Dep. Rep., Carnegie-Mellon U., Pittsburgh, Pa., May 1975. To appear in *IEEE Trans. Comptrs*, 1977.
4. Flynn, M.J. Very high-speed computing systems. *Proc. IEEE 54* (1966), 1901–1909.
5. Gentleman, W.M. Some complexity results for matrix computations on parallel processors. Presented at Symp. on New Directions and Recent Results in Algorithms and Complexity, Carnegie-Mellon U., Pittsburgh, Pa., April 1976.
6. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
7. Orcutt, S.E. Computer organization and algorithms for very-high speed computations. Ph.D. Th., Stanford U., Stanford, Calif., Sept. 1974, Chap. 2, pp. 20–23.
8. Stone, H.S. Parallel processing with the perfect shuffle. *IEEE Trans. Comptrs. C-20* (1971), 153–161.

Programming Techniques

G. Manacher, S.L. Graham Editors

Proof Techniques for Hierarchically Structured Programs

Lawrence Robinson and Karl N. Levitt
Stanford Research Institute

A method for describing and structuring programs that simplifies proofs of their correctness is presented. The method formally represents a program in terms of levels of abstraction, each level of which can be described by a self-contained nonprocedural specification. The proofs, like the programs, are structured by levels. Although only manual proofs are described in the paper, the method is also applicable to semi-automatic and automatic proofs. Preliminary results are encouraging, indicating that the method can be applied to large programs, such as operating systems.

Key Words and Phrases: hierarchical structure, program verification, structured programming, formal specification, abstraction, and programming methodology

CR Categories: 4.0, 4.29, 4.9, 5.24

The research described in this paper was partially sponsored by the National Science Foundation under Grant DCR74-18661.
Authors' address: Stanford Research Institute, Menlo Park, CA 94025.