52. Noll, R. *Reforming Regulation.* The Brookings Inst., Washington, D.C., 1971.
53. Parker, D.B. *Crime by Computer.* Scribners, New York, 1976.
54. Peterson, H.E., and Turn, R. System implications of information privacy. Proc. AFIPS 1967 SJCC, Vol. 30, AFIPS Press, Montvale, N.J., pp. 291–300.
55. Portway, P.S. EFT systems? No thanks, not yet. *Computerworld 12,* 2 (Jan. 9, 1978), 14–16, 21, 23–25.
56. Privacy Protection Study Commission. Personal Privacy in an Information Society. U.S. Gov't. Printing Office, Washington, D.C., July, 1977.
57. Prives, D. The explosion of state laws on electronic fund transfer systems. P-76-1, Prog. Inform. Technologies and Public Policy, Harvard U., Cambridge, Mass., 1976.
58. Reid, S. *The New Industrial Order: Concentration Regulation and Public Policy.* McGraw-Hill, New York, 1976.
59. Richardson, D.W. *Electric Money: Evolution of an Electronic Funds-Transfer System.* M.I.T. Press, Cambridge, Mass., 1970.
60. Rose, S. More bang for the buck: The magic of electronic banking. *Fortune 95,* 5 (1977), 202–226.
61. Rossman, L.W. Financial industry sees EFT privacy laws adequate. *American Banker CXLI,* 210 (Oct. 28, 1976), 1, 11.
62. Rule, J. *Private Lives and Public Surveillance.* Schocken Books, New York, 1974.
63. Rule, J. Value Choices in Electronic Funds Transfer Policy. Office of Telecommunications Policy, Executive Office of the President, Washington, D.C., Oct. 1975.
64. Saltzer, J., and Schroeder, M. The protection of information in computer systems. *Proc. IEEE 65,* 9 (Sept. 1975), 1278–1308.
65. Sayre, K., Ed. *Values in the Electric Power Industry.* U. of Notre Dame Press, Notre Dame, Ind., 1977.
66. Schick, B. Some impacts of electronic funds transfer on consumer transactions. Federal Reserve Bank of Boston. The economics of a national electronics funds transfer system. Conf. Ser. No. 13, Boston, Mass., Oct. 1974, pp. 165–179.
67. Schuck, P.H. Electronic funds transfer: A technology in search of a market. *Maryland Law Review 35,* 1 (1975), 74–87.
68. Schultze, L. The public use of the private interest. *Harpers 254,* 1524 (May 1977), 43–62.
69. Simpson, R.C. Money transfer services. *Computers and Society 7,* 4 (Winter 1976), 3–9.
70. Steifel, R.C. A checkless society or an unchecked society? *Computers and Automation 19* (Oct. 1970), 32–35.
71. Sterling, T., and Laudon, K. Humanizing information systems. *Datamation 22,* 12 (Dec. 1976), 53–59.
72. The time is NOW. *Forbes Magazine 120,* 1 (July 1, 1977), 61–62.
73. Turoff, M., and Mitroff, I. A case study of technology assessment applied to the "cashless society" concept. *Technol. Forecasting Soc., Change 7* (1975), 317–325.
74. Weissman, C. Secure computer operation with virtual machine partitioning. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., 1975, pp. 929–934.
75. U.S. Dept. HEW, Secretary's Advisory Committee on Automated Personal Data Systems. Records Computers, and the Rights of Citizens. Washington, D.C., 1973.
76. Walker, G. M. Electronic funds transfer systems. *Electronics* (July 24, 1975), 79–85.
77. Webber, M. The BART experience—What have we learned? *The Public Interest,* Vol. 45 (Fall 1976), 79–108.
78. Weizenbaum, J. *Computer Power and Human Reason.* Freeman, San Fransisco, 1976.
79. Wessel, M. *Freedom's Edge: The Computer Threat to Society.* Addison-Wesley, Reading, Mass., 1974.
80. Westin, A., and Baker, M. *Databanks in a Free Society.* Quadrangle Books, New York, 1972.
81. Whiteside, T. *Computer Capers.* Crowell, New York, 1978.
82. Wilcox, C., and Shepard, W. *Public Policies Towards Business.* Richard D. Irwin, Homewood, Fifth ed., 1975.
83. Winner, L. *Autonomous Technology: Technology Out-of-Control as a Theme in Political Thought.* M.I.T. Press, Cambridge, Mass., 1977.
84. Wise, D. *The American Police State.* Random House, New York, 1976.

Programming Techniques

S. L. Graham, R. L. Rivest Editors

Fast Parallel Sorting Algorithms

D. S. Hirschberg
Rice University

A parallel bucket-sort algorithm is presented that requires time $O(\log n)$ and the use of n processors. The algorithm makes use of a technique that requires more space than the product of processors and time. A realistic model is used in which no memory contention is permitted. A procedure is also presented to sort n numbers in time $O(k \log n)$ using $n^{1+1/k}$ processors, for k an arbitrary integer. The model of computation for this procedure permits simultaneous fetches from the same memory location.

Key Words and Phrases: parallel processing, sorting, algorithms, bucket sort
CR Categories: 3.74, 4.34, 5.25, 5.31

There is often a time-space tradeoff in serial algorithms. In order to solve a problem within a certain time bound, a minimal amount of space is required. This space requirement may be reduced if we allow more time for the process. In the limit, there will be a minimum amount of space required.

Much work has recently been devoted to developing algorithms for parallel processors. Problem areas include sorting [3, 6, 16, 17], evaluation of polynomials, and general arithmetic expressions [14, 4], and matrix- and graph theoretic problems [15, 5, 2, 12, 9]. In parallel algorithms, there is a similar tradeoff between time and the number of processors used. In order to solve a problem using a bounded number of processors, a minimal amount of time is required. This time requirement

Research supported by NSF grant MCS-76-07683.

Author's present address: Department of Electrical Engineering, Rice University, Houston, TX, 77001.

may be reduced if we allow more processors to help in the computation. In the limit, there will be a minimum time requirement.

We shall present an algorithm design technique which dramatically exemplifies the three-way tradeoff between space, time, and processors. It is hoped that this technique can also be applied to strictly serial algorithms.

Muller and Preparata [13] were the first to exhibit a network capable of sorting n numbers in time $O(\log n)$. Their method required $O(n^2)$ processing elements. We shall present an algorithm for sorting n numbers in time $O(\log n)$ that requires asymptotically fewer processors.

We first present parallel bucket-sorting algorithms in which, at the expense of greater space requirements, the number of processors and the amount of time used are both reduced. The algorithms are unusual in that the space requirements are greater than the processor time requirements.

Our computational model assumes that all processors have access to a common memory as well as having small local memories. All processors are synchronized and follow the instructions of a unique instruction stream. This model has been called an SIMD (Single Instruction stream, Multiple Data stream) computer [7]. The instructions may involve memory references or constants that are a linear function of the bits in the binary representation of the *processor number,* the processors being numbered by consecutive integers starting from zero. We assume that the addition of two numbers can be executed in one time unit.

Our preliminary algorithm will sort n numbers using n (parallel) processors in time $O(\log n)$ under the assumption that the numbers that are to be sorted, $\{c_i\}_{i=1}^{n}$, are from $\{0, 1, \ldots, m-1\}$ and with the proviso that duplicate numbers (should there be any) are to be discarded. The proviso will be dropped in the second algorithm. This is a parallel version of the "bucket sort."

An obvious implementation of the parallel bucket sort would be for each processor p_i (which has temporarily been "assigned to" c_i, the ith number being sorted) to place the value i in bucket c_i. The problem with this solution is that, in general, there may be several values of i with identical numbers c_i. A memory conflict would result from the simultaneous attempts of several processors to store different values of i into the same bucket.

Our answer to this problem is to eliminate duplicate copies of the same number. Processor p_i will be (temporarily) deactivated if there is another processor p_j whose index, j, is smaller than i, and $c_j = c_i$. Then, for each number appearing among the numbers being sorted, only one processor (the one with smallest index) will be active when we place i in bucket c_i.

Our implementation of the elimination procedure is interesting. We shall have m areas of memory, one for each bucket. Each area will be of size n, the number of input numbers to be sorted. Within each area, j, the processors (p_i) having $c_i = j$ will leave marks indicating their presence. Then, in a binary-tree fashion, they will search for the presence of (the marks of) other active processors. If two processors discover each other's presence (such discoveries will turn out to be simultaneous), the lower ranking one (i.e. the one with smaller index) will continue while the higher ranking one will deactivate.

There being n locations per area (numbered 0 through $n-1$), each processor p_i can make its mark at location i in area c_i without fear of memory conflict. Iteratively, each processor then determines whether or not its "buddy" is active within the same area. (Here, "buddy" is defined analogously to the definition used in the Buddy System for dynamic memory allocation [10].) If so, then the processor with higher rank (i.e. larger index i) will deactivate. If the buddy is not active or if it is active but is of higher rank, then the processor will continue, shifting its mark to the location of the buddy if that location is of lower index than the one currently in use. If the buddy was active, then, of course, no shift will occur.

After the kth iteration, a mark will be present at each location whose last k bits are zeros and whose other $(\log n) - k$ bits coincide with the corresponding bits of the address of a processor active in the same area. Thus, each such location will be marked iff any of 2^k processors had been active in that area originally. After $\log n$ iterations, the first location in an area will be marked iff any of the n processors originally were active in that area, i.e. iff any of the n numbers to be sorted was j, the area bucket number.

The algorithm is expressed formally below. Variables in capitals are in common memory, variables in lower case are in local memory (i.e. there will be one copy of each such variable for each processor).

Algorithm 1—parallel bucket sort

Input: $0 \le i \le n-1$ $A[j, i] = 0$
$0 \le j \le m-1$ $B[j] = 0$
$c_i \in \{0, 1, \ldots, m-1\}$, not necessarily distinct

Output: $0 \le i \le n-1$ $A[j, i] = 0$
$0 \le j \le m-1$ $B[j] = \min i$ s.t. $c_i = j$, 0 if none such

Let $e_k = 0 \ldots 0\ 10 \ldots 0$, all bits 0 except the kth from the right

```
for all i do
  Let i = x_{log n} ... x_2 x_1 be the binary representation of i
  x ← i                                  x is the location that p_i is
                                           marking
  A[c_i, x] ← 1
  flag ← 1                               flag = 1 iff p_i is active
  for k ← 1 step 1 until log n do
    begin
    buddy ← x ⊕ e_k                      address of buddy
    count ← A[c_i, buddy]                count ≠ 0 if buddy is active
    if x_k = 1 AND count ≠ 0 then
      flag ← 0                           if buddy is active and if we are
                                           the higher of the 2 then we'll
                                           wait and buddy will continue
  if x_k = 1 AND count = 0 AND
      flag = 1 then begin
```

```
    A[c_i, x] ← 0                    if buddy is not active then there
    x ← buddy                          is no problem in using his
    A[c_i, x] ← 1                      space
  end
end
if flag = 1 then B[c_i] ← i
A[c_i, x] ← 0
```

It is noted that this bucket-sort algorithm requires space $S = O(mn)$, time $T = O(\log n)$, and the use of n processors.

We now present another bucket-sort algorithm. This algorithm will give the actual ranking of the input numbers, equal numbers being kept in the same relative order but getting different ranks, assuming that the input numbers are from a predefined small set.

The algorithm follows the same basic pattern set by our previous algorithm. However, instead of a simple mark bit, we shall keep a running count of how many processors were originally active in each block of indices of size 2^k. If a processor encounters an active buddy then only the lower buddy continues to be active. In any case, all processors p_i (active or not) will add to their running count of the number of processors (that were originally active) having indices greater than i. Active processors keep their count at the head of the largest block that they have investigated (which will be of size 2^k). At the end there will be at most one active processor per area and $A[j, 0]$ will be the number of different i's such that $c_i = j$. An inactive processor, p_i, will keep its count at the head of the largest block which had no other processors of index smaller than i.

Algorithm 2.1—parallel bucket sort (part 1)

```
Input:   0 ≤ i ≤ n − 1   A[j, i] = 0
         0 ≤ j ≤ m − 1   B[j] = 0
                         c_i ∈ {0, 1, ..., m − 1} not necessarily distinct
Local
Output:                  r = max k s.t. ¬∃ t in same 2^k-block as i
                           with t < i and c_t = c_i
                         y = head of 2^r-block containing i
Output:  0 ≤ i ≤ n − 1   A[j, i] = 0 except A[c_i, y] = # of t ≥ i s.t.
                           c_t = c_i
         0 ≤ j ≤ m − 1   B[j] = min i s.t. c_i = j, 0 if none such
for all i do
  flag ← 1
  r ← log n
  x ← i                                  x is the head of the largest block
                                           that p_i has counted
  y ← i                                  y is the location at which p_i is
                                           accumulating that count
A[c_i, y] ← 1                            A[c_i, y] will hold # of t s.t.
                                           t ≥ i and c_t = c_i
for k ← 1 step 1 until log n do
begin
  buddy ← x ⊕ e_k
  count ← A[c_i, buddy]
  if x_k = 0 then A[c_i, y] ← A[c_i, y]  if buddy is at higher index then
    + count                                accumulate count
  else (x_k = 1)
  begin
    x ← buddy                            buddy is head of the 2^k-size
                                           block
    if count ≠ 0                         if buddy is active then we are
                                           not lowest ranked in this 2^k-
                                           block
    then if flag = 1 then [flag←0;
                           r←k−1]
                      else null          (eliminates "dangling else")
  else (count=0)                         buddy is inactive
    if flag = 1 then
    begin                                if we are still active then our
      A[c_i, x] ← A[c_i, y]                count must be moved to head
      A[c_i, y] ← 0                        of the block
      y ← x
    end (of then block)
  end (of else x_k = 1)
end (of for loop)
B[c_i] ← A[c_i, y]                       y will be zero here
```

At this point we have isolated one representative of each number that appears among the numbers to be sorted and we have obtained a count of how many times each number occurs. We now will accumulate the counts (for each number c_i) of all numbers that are greater than c_i in order to know the actual ranking of the numbers, assuming that duplicate numbers will be kept. This accumulation will be done in a manner similar to that used previously.

Algorithm 2.2—parallel bucket sort (part 2)

```
Input: from Algorithm 2.1
Output:  0 ≤ i ≤ n − 1 D[i] = sorted position of c_i only for the first
           instance of each c_i
         larger values of c_i will have smaller values of D = (# of k
           s.t. c_k > c_i) + 1
for all i do
  D[i] ← 0
  if flag = 1 then
  begin Let c_i = w_{log m} ... w_2 w_1 be the binary representation of c_i
    flag2 ← 1
    w ← c_i                              w is head of the largest block
                                           that p_i has counted
    z ← c_i                              z is the location at which p_i is
                                           accumulating that count
                                         B[z] now = # of k s.t. c_k = z
    for k ← 1 step 1 until log m do
    begin
      buddy ← w ⊕ e_k
      count ← B[buddy]                   B[z] will = # of k s.t. c_k ≥ z
      if w_k = 0 then
        B[z] ← B[z] + count
      else (w_k = 1)
    begin w ← buddy
        if count ≠ 0 then
          flag2 ← 0
        else if flag 2= 1 then
    begin
          B[w] ← B[z]
          B[z] ← 0
          z ← w
        end (of then block)
      end (of else w_k=1)
    end (of for loop)
    D[i] ← B[z] − A[c_i, y] + 1
    A[c_i, y] ← B[z]
    B[z] ← 0
  end (of if flag=1)
```

At this point, the representative of each number that appears among the $\{c_i\}$ has a count of the total number of c_t's that are greater than it plus the number of c_t's that

are equal to it. Each of the duplicates has a count of the number of c_i's that are equal to it but of higher index. The D-value, i.e. rank, is just the difference of these two quantities plus one.

The D-value of the representatives was calculated at the end of Algorithm 2.2 and so the D-values of the duplicates can be calculated by:

```
for all i do
  if flag = 0 then D[i] ← A[c_i, 0] − A[c_i, y] + 1
```

If we insist that not more than one processor may simultaneously access a location, not even for fetches, then the D-values of duplicates can be evaluated using the reverse of the procedure of Algorithm 2.1. This is done below in Algorithm 2.3.

Algorithm 2.3—parallel bucket sort (part 3)

```
Input:   from Algorithm 2.2
Output:  0 ≤ i ≤ n − 1  A[j, i]  = 0
         0 ≤ j ≤ m − 1  D[i]     = sorted position of c_i
                                 = # of k s.t. c_k > c_i
                                   + # of k ≤ i s.t. c_k = c_i
         larger values of c_i will have smaller values of D
         equal values of c_i will keep relative order
for all i do
  for k ← (log n) − 1 step − 1 until 0 do
  if r = k then
  begin                                  get value of A[c_i, 0] from buddy
  x ← y ⊕ e_r                              location
  D[i] ← A[c_i, x] − A[c_i, y] + 1
  A[c_i, y] ← A[c_i, x]
  end
  else if r > k then
    begin
      x ← y OR (i AND e_k)
      if x ≠ y then begin
          A[c_i, x] ← A[c_i, y]
          A[c_i, y] ← 0
          end (of then block)
      end (of if r>k)
  (end of for loop)
  A[c_i, y] ← 0
```

It is noted that Algorithm 2 (the sequence of Algorithms 2.1, 2.2, 2.3) requires space $S = O(mn)$, time $T = O(\log n + \log m)$, and the use of n processors.

The algorithms given above assume that an area (A) of memory has been initialized to zero. This is not unreasonable. Many instances of this algorithm can be executed one after another. The memory will be clear upon the termination of each program.

However, there are methods which make it unnecessary to initialize the area. For serial programs, one can include at each location a pointer to a backpointer on a stack. Each time an entry is accessed, verification can be made that the contents are not random by checking that the pointer in that entry points to the active region on the stack and that the backpointer points to the entry [1].

This method is also valid for parallel programs unless we add the restriction that simultaneous multiple access to the same location is prohibited even for memory-fetch instructions. It is possible that several entries that are accessed in parallel may have random contents, more than one of which points to the same location. A memory fetch conflict would ensue.

For situations similar to that in Algorithm 2, the following is a possible solution. At each step of the process, each active processor can initialize the location its buddy is working on (to zero), then reinitialize the contents of the location it is working on (to its latest value). A location will thus be initialized if either of the two processes to which it might be relevant is active.

We now present algorithms that will sort n arbitrary numbers in time $O(\log n)$. They are based on an extension of an algorithm due to Gavril [8] that merges two linearly-ordered sets in time $O(\log n)$. Our first algorithm to do this, Algorithm 3, will require the use of $n^{3/2}$ processors.

Algorithm 3—parallel sort using $n^{3/2}$ processors

Input: $0 \le i \le n-1$ $\quad c_i \in$ integers
Output: $\quad \{c_i\}$ will be stably sorted, smallest first

1. Partition the n input numbers into $n^{1/2}$ groups, each having $n^{1/2}$ elements.
2. Within each group do
   For each element, j, determine count[j] = (# of i such that $c_i < c_j$) + (# of $i \le j$ such that $c_i = c_j$). This can be done in time $O(\log n)$ using $n^{1/2}$ processors per element (a total of n processors per group or $n^{3/2}$ processors in toto). The $n^{1/2}$ processors for element j will be assigned, one to each element i in j's group, to compare c_i with c_j. Summing the results of these comparisons can be done in time $O(\log n)$.
3. Within each group do
   Bucket sort the elements, using count[j] as the key for the jth element in the group. This is done by: $c_{\text{count}[j]} \leftarrow c_j$, where j and count[j] are offsets (of value at most $n^{1/2}$) from the beginning of each group. There will be no memory conflicts since the count[j]'s within a group are all distinct. Steps 2 and 3 have effectively sorted the elements within each group using an "Enumeration Sort" [11].
4. All elements do a binary search of the $n^{1/2}$ groups. That is, each element(c_j) in group g, has $n^{1/2}$ processors which are assigned, one to each group, to do a binary search on the elements in a group (which are sorted) so as to determine, for all groups k, the value of

$$\text{count}[j,k] = \begin{cases} \text{if } k < g, \#\text{ of elements } i \text{ such that } c_i \le c_j \\ \text{if } k = g, j \\ \text{if } k > g, \#\text{ of elements } i \text{ such that } c_i < c_j \end{cases}$$

   where c_i refers to the ith element in group k and c_j is fixed.
5. For all elements, j, evaluate count[j] = sum (over k) of count[j, k]. This can be done in time $O(\log n)$ and requires $n^{1/2}$ processors per element for a total of $n^{3/2}$ processors.
6. Do a bucket sort on all n elements using count[j] as the key for the jth element. Again, there will be no memory conflicts since count[j] will be the rank of the jth element.
7. END of Algorithm 3.

We note that Algorithm 3 requires time $O(\log n)$ and the use of $n^{3/2}$ processors. We now show a simple modification of Algorithm 3 which will use the same order of magnitude of time and require only $n^{4/3}$ processors.

Algorithm 4—parallel sort using $n^{4/3}$ processors

1. Partition the n input numbers into $n^{2/3}$ groups each having $n^{1/3}$ elements.
2. Within each group do
   For each element, j, determine count[j] = # of i such that $c_i < c_j$) + (# of $i \le j$ such that $c_i = c_j$).

3. Within each group do
Bucket sort the count[j]'s obtained in step 2. This will rearrange the elements in rank order within each group.
4. Divide the $n^{2/3}$ groups into $n^{1/3}$ sectors, each sector consisting of $n^{1/3}$ groups.
5. Within each sector do
For each element (j) in group g, do a binary search of each of the $n^{1/3}$ groups in j's sector to determine, for all k, the value of

$$\text{count}[j, k] = \begin{cases} \text{if } k < g, \# \text{ of } i \text{ in group } k \text{ such that } c_i \leq c_j \\ \text{if } k = g, j \\ \text{if } k > g, \# \text{ of } i \text{ in group } k \text{ such that } c_i < c_j. \end{cases}$$

Then, for each element j, evaluate count[j] = (# of i in j's sector such that $c_i < c_j$) + (# of $i \leq j$ in j's sector such that $c_i = c_j$). This number is simply the sum (over k) of count[j, k].
6. Within each sector, do a bucket sort of the elements within the sector using count[j] as they key for element j. This will rearrange the elements in rank order within each sector.
7. For all elements (j) in sector t, do a binary search of each of the $n^{1/3}$ sectors to determine, for all k, the value of

$$\text{count}[j, k] = \begin{cases} \text{if } k < t, \# \text{ of } i \text{ in sector } k \text{ such that } c_i \leq c_j \\ \text{if } k = t, j \\ \text{if } k > t, \# \text{ of } i \text{ in sector } k \text{ such that } c_i < c_j. \end{cases}$$

Then evaluate count[j] = the sum (over k) of count[j, k].
8. Do a bucket sort of all n elements.
9. END of Algorithm 4.

In a like manner, we can exhibit an algorithm to sort n numbers in $O(k \log n)$ time that uses $n^{1+1/k}$ processors. Interestingly, by setting $k = \log n$ (initially splitting the n elements into $n/2$ groups of 2 each), we obtain an algorithm to sort n numbers in $O(\log^2 n)$ time using $O(n)$ processors, the same resources used by Batcher's algorithms.

We note that these algorithms, although avoiding memory-store conflicts, do have memory-fetch conflicts. That is, we allow more than one processor to simultaneously access the same memory location. As an open problem, we pose the question: Can n numbers be sorted in time $O(\log n)$ if memory-fetch conflicts are not permitted?

Received December 1976; revised September 1977

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1973, p. 71.
2. Arjomandi, E. A study of parallelism in graph theory. Ph.D. Th., Dept. of Comptr. Sci., U. of Toronto, Toronto, Ont., Dec. 1975.
3. Batcher, K.E. Sorting networks and their applications. Proc. AFIPS 1968 SJCC, Vol. 32, AFIPS Press, Montvale, N.J., pp. 307–314.
4. Brent, R.P. The parallel evaluation of general arithmetic expressions. *J. ACM 21*, 2 (April 1974), 201–206.
5. Csansky, L. Fast parallel matrix inversion algorithms. Proc. 16th Annual Symp. on Foundations of Comptr. Sci., IEEE, Berkeley, Calif., Oct. 1975, pp. 11–12.
6. Even, S. Parallelism in tape-sorting. *Comm. ACM 17*, 4 (April 1974), 202–204.
7. Flynn, M.J. Very high-speed computing systems, *Proc. IEEE 54* (Dec. 1966), 1901–1909.
8. Gavril, F. Merging with parallel processors. *Comm. ACM 18*, 10 (Oct. 1975), 588–591.
9. Hirschberg, D.S. Parallel algorithms for the transitive closure and the connected component problems. Proc. 8th Annual ACM Symp. on Theory of Comptng. Hershey, Pa., May 1976, pp. 55–57.
10. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., Sec. Ed., 1973.
11. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
12. Levitt, K.N., and Kautz, W.H. Cellular arrays for the solution of graph problems. *Comm. ACM 15*, 9 (Sept. 1972), 789-801.
13. Muller, D.E., and Preparata, F.P. Bounds to complexities of networks for sorting and for switching. *J. ACM 22*, 2 (April 1975), 195–201.
14. Munro, I., and Paterson, M. Optimal algorithms for parallel polynomial evaluation, *J. Comptr. Syst. Sci. 7* (1973), 189–198.
15. Muraoka, Y., and Kuck, D.J. On the time required for a sequence of matrix products. *Comm. ACM 16*, 1 (Jan. 1973), 22–26.
16. Stone, H.S. Parallel processing with the perfect shuffle. *IEEE Trans. Comptrs. C-20* (1971), 153–161.
17. Thompson, C.D., and Kung, H.T. Sorting on a mesh-connected parallel computer. Proc. 8th Annual ACM Symp. on Theory of Comptng. Hershey, Pa., May 1976, pp. 58–64.
18. Valiant, L.G. Parallelism in comparison problems. *SIAM J. Comptng. 4*, 3 (Sept. 1975), 348–355.