Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network

DAVID NASSIMI

Northwestern University, Evanston, Illinois

AND

SARTAJ SAHNI

University of Minnesota, Minneapolis, Minnesota

Abstract. $O(k \log N)$ algorithms are obtained to permute and sort N data items on cube and perfect shuffle computers with $N^{1+1/k}$ processing elements, $1 \le k \le \log N$. These algorithms lead directly to a generalized-connection-network construction having $O(k \log N)$ delay and $O(kN^{1+1/k} \log N)$ contact pairs. This network has the advantage that the switches can be set in $O(k \log N)$ time by either a cube or perfect shuffle computer with $N^{1+1/k}$ processing elements.

Categories and Subject Descriptors. B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems)—*topology*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*interconnection architectures, parallel processors, single-instruction-stream multiple-data-stream architectures (SIMD)*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*routing and layout, sorting and searching*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Permutation, cube computer, perfect shuffle computer, generalized connection network

1. *Introduction*

Computer systems may be broadly classified into four categories [6]. These are: (i) SISD (*S*ingle *I*nstruction Stream, *S*ingle *D*ata Stream); (ii) SIMD (*S*ingle *I*nstruction Stream, *M*ultiple *D*ata Stream); (iii) MISD (*M*ultiple *I*nstruction Stream, *S*ingle *D*ata Stream); and (iv) MIMD (*M*ultiple *I*nstruction Stream, *M*ultiple *D*ata Stream). This paper is concerned solely with SIMD-type computers. An SIMD-type computer system is comprised of some number of processing elements (PEs), each having some local memory. We assume that the PEs are indexed 0 through $p - 1$ and refer to the ith PE as PE(i). The PEs are synchronized and operate under the control of a single instruction stream. An enable/disable mask may be used to select a subset of PEs

This research was supported in part by the National Science Foundation under Grant MCS 78-15455 and represents a portion of the first author's doctoral dissertation.

Authors' addresses: D. Nassimi, Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201; S. Sahni, Department of Computer Science, 136 Lind Hall, University of Minnesota, Minneapolis, MN 55455.

that will perform the instruction to be executed at any given time. All enabled PEs perform the same instruction. Four SIMD architectures are relevant to our discussion:

(i) *Shared Memory Model* (*SMM*). In this model there is a common memory available to all PEs. This is in addition to the local memory available to each PE. Data may be transmitted from PE(i) to PE(j) by simply having PE(i) write the data into the common memory and then letting PE(j) read it. It thus takes $O(1)$ time to transmit a word from one PE to another in this model. Two PEs are not permitted to write into the same word of common memory simultaneously. PEs may or may not be allowed to read the same word of common memory simultaneously. Algorithms that require two or more PEs to read the same common memory word simultaneously will be said to have *memory fetch conflicts*.

(ii) *Mesh Connected Computer* (*MCC*). In this model there is no shared or common memory. The PEs may be thought of as logically arranged as in a k-dimensional array $A(n_{k-1}, n_{k-2}, \ldots, n_0)$, where n_i is the size of the ith dimension and $p = n_{k-1} * n_{k-2} * \cdots * n_0$. The PE at location $A(i_{k-1}, \ldots, i_0)$ is connected to the PEs at locations $A(i_{k-1}, \ldots, i_j \pm 1, \ldots, i_0)$, $0 \leq j < k$, provided they exist. Data may be transmitted from one PE to another only via this interconnection pattern.

(iii) *Cube Connected Computer* (*CCC*). Assume that $p = 2^q$, and let $i_{q-1} \cdots i_0$ be the binary representation of i for $i \in [0, p-1]$. Let $i^{(b)}$ be the number whose binary representation is $i_{q-1} \cdots i_{b+1}\bar{i}_b i_{b-1} \cdots i_0$, where $\bar{i}_b$ is the complement of i_b and $0 \leq b < q$. In the cube model, PE(i) is connected to PE($i^{(b)}$), $0 \leq b < q$. As in the mesh model, data can be transmitted from one PE to another only via the interconnection pattern.

(iv) *Perfect Shuffle Computer* (*PSC*). Let p, q, i, and $i^{(b)}$ be as in the cube model. In the perfect shuffle model, PE(i) is connected to PE($i^{(0)}$), PE($i_{q-2}i_{q-3} \cdots i_0 i_{q-1}$), and PE($i_0 i_{q-1} i_{q-2} \cdots i_1$). These three connections will be called *exchange*, *shuffle*, and *unshuffle*, respectively. Once again, data transmission from PE to PE is possible only via the connection scheme.

It should be noted that the MCC model requires $2k$ connections per PE, the CCC model requires $\log p$ (all logarithms in this paper are base 2), and the PSC model requires only three connections per PE. The SMM requires a large amount of PE to memory connections to permit simultaneous memory access by several PEs.

Each of these models has received much attention in the literature. Arjomandi [1], Csanky [4], Eckstein [5], and Hirschberg [7] have developed algorithms for certain matrix and graph problems using an SMM. Hirschberg [8], Muller and Preparata [12], and Preparata [18] have considered the sorting problem for the SMM. The evaluation of polynomials on the SMM has been studied by Munro and Paterson [13], while arithmetic expression evaluation has been considered by Brent [3] and others. Efficient algorithms to sort and perform data permutations on an MCC can be found in Thompson and Kung [23], Nassimi and Sahni [14, 15], Swanson [21], and Thompson [22]. Thompson's algorithms [22] can also be used to perform permutations on a CCC or PSC. The shuffle–exchange connection was originally proposed by Stone [20]; its usefulness was shown for sorting as well as some numeric problems. Permutation capabilities of variants of the shuffle–exchange network have been studied by Lang [10], Lang and Stone [11], and others.

In this paper we are primarily concerned with the development of efficient algorithms to sort and permute data on a CCC and a PSC. For both problems we assume that there are $N = 2^n$ records. Initially, record i is in PE(i), $0 \leq i < N$. Each record i has a field $A(i)$. In the sorting problem the N records are to be rearranged into nondecreasing order of $A(i)$, one record to a PE. For the permutation problem, $A(i) \in [0, N-1]$, $0 \le i < N$, and record i is to be relocated to PE($A(i)$), $0 \le i < N$. N records can be permuted in $O(1)$ time using the SMM and N PEs. PE(i) first writes its record into location $A(i)$ of the common memory (assuming the common memory is large enough to hold N records) and then reads back the record in location i. Clearly, this algorithm has no read/write conflicts. N records can be sorted on an N-PE SMM in $O(\log^2 N)$ time, using Batcher's algorithm [2]. Hirschberg [8] and Preparata [18] have developed sorting algorithms for SMM for the case when $N^{1+1/k}$ PEs are available. Their algorithms run in $O(k \log N)$ time. Hirschberg's algorithm has memory fetch conflicts, while Preparata's does not. Thompson and Kung [23] and Nassimi and Sahni [14] show how to sort $N = n^2$ data on an $n \times n$ MCC in $O(n)$ time. Both these papers also generalize their results to higher dimensional MCCs. For the permutation problem on MCCs, Nassimi and Sahni [15] have developed an algorithm that is optimal. Their algorithm, however, can be used on only a special class of permutations.

It is well known that N records can be sorted in $O(\log^2 N)$ time on both an N-PE CCC and an N-PE PSC [20]. Algorithms to perform certain permutations in less time on these models can be found in [10, 11, 20]. (Thompson's algorithm [22] can perform *any* permutation in $O(\log N)$ time *if the permutation is known in advance.* Otherwise, the best known algorithm for arbitrary permutations requires $O(\log^2 N)$ time on an N-PE CCC or PSC.)

In Section 2 we develop an algorithm to perform arbitrary permutations in $O(k \log N)$ time on a CCC and a PSC with $N^{1+1/k}$ PEs. In Section 3 we develop an algorithm for the sorting problem. This algorithm has the same asymptotic complexity as the permutation algorithm of Section 2. Finally, in Section 4 we develop a generalized connection network (GCN). An (N, N)-GCN is an N-input, N-output switching network capable of implementing any one-to-many mapping of inputs onto outputs. The GCN constructed here has $O(k \log N)$ delay and $O(kN^{1+1/k} \log N)$ contact pairs. In this respect it is inferior to the GCN construction of Thompson [22], which has $7.6N \log N$ contact pairs (using three-way branching) and a delay of $3.8 \log N$. However, our GCN has the advantage that its switch settings can be determined in $O(k \log N)$ time using a CCC or PSC with $N^{1+1/k}$ PEs. In fact, the algorithm to determine switch settings is almost identical to the permutation algorithm of Section 2. Note that the asymptotically fastest algorithms known to determine the switch settings for Thompson's GCN run in $O(\log^2 N)$, $O(N^{1/2})$, and $O(k \log^3 N)$ time, respectively, on an N-PE SMM, an $N^{1/2} \times N^{1/2}$ MCC, and an $N^{1+1/k}$-PE CCC or PSC [16].

2. *Permutations*

Let $G(i)$, $0 \le i < N = 2^n$, be N records. $G(i)$ is initially located in PE(i). Let $A(i)$ be a field in record $G(i)$ such that $A(0), \ldots, A(N-1)$ defines a permutation of $(0, 1, \ldots, N-1)$. The records are to be permuted so that following the permutation, $G(i)$ is in PE($A(i)$), $0 \le i < N$. We assume that this permutation is to be performed on a CCC or a PSC having $N^{1+1/k}$ PEs, where $k = n/m$ for some integer m, $1 \le m \le n$. For purposes of discussion we view these $N^{1+1/k} = 2^{n+m}$ PEs as arranged in a $2^m \times 2^n$ array. The PEs are indexed in row-major order, with the result that record $G(i)$ is initially located in the PE in row 0 and column i, $0 \le i < N$.

2.1 INFORMAL DESCRIPTION OF THE ALGORITHM. The permutation algorithm to be described is essentially a parallel version of MSD (most-significant-digit) radix sort (see [9]). The radix used is 2^m. Using this radix, the $\lceil n/m \rceil$ digits of $A(i)$ are

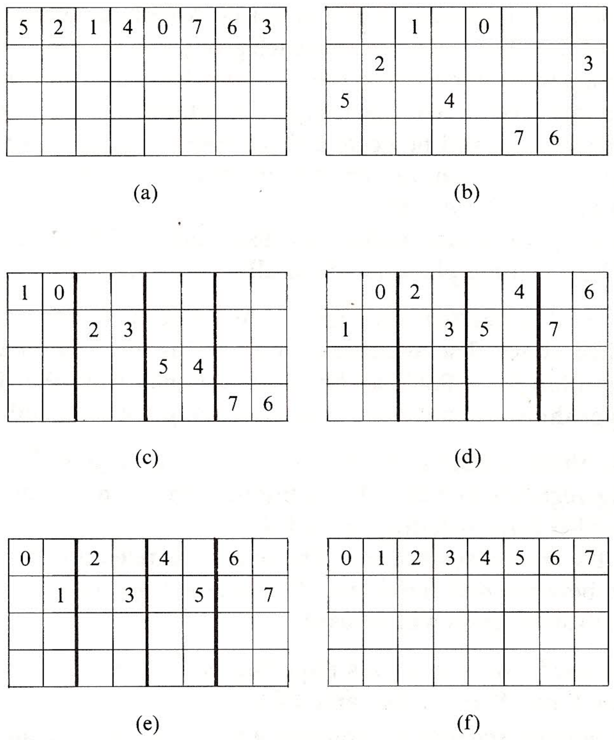


FIG. 1. Permutation of 8 elements on 4 × 8 PEs.

obtained as follows. First, the binary representation of $A(i)$ is obtained. The m most significant bits yield the most significant digit, the next m bits give the next digit, and so on. For example, if $m = 2$ and $n = 3$, then all numbers in the range [0, 7] are represented as d_1d_0 with radix $2^m = 4$. The representations are: 00, 01, 10, 11, 20, 21, 30, and 31. An example will illustrate how the permutation algorithm proceeds. Let us assume that $m = 2$ and $n = 3$. Hence, 8 records are to be permuted using 32 PEs. Figure 1a represents a possible initial configuration. Each square represents a PE, and the number within a square is the value of $A(i)$. The earlier example gives the d_1 and d_0 values of each $A(i)$. Since there are two digits (using radix 2^m) in the representation of the $A(i)$'s, the permutation algorithm will go through two phases. In the first the records will be ordered with respect to d_1 and in the second with respect to d_0. In phase 1 the record in PE(i) is routed to the PE in column i and row $D1(i)$, where $D1(i)$ is the d_1 value of $A(i)$. Figure 1b shows the configuration following this routing. Next, the records are ranked. The *rank*, $R(i)$, of a record in PE(i) is equal to the number of records located in PEs with index less than i. (Recall that PEs are indexed in row–major order.) For example, for the configuration of Figure 1b we have $R(2) = 0$, $R(4) = 1$, $R(9) = 2$, etc. Having determined the ranks, each record is routed to the column specified by its rank. (Note that columns are numbered 0 through $N - 1$.) During this routing, records do not change rows. This results in the configuration of Figure 1c. The routing carried out will be referred to as *concentration.* This completes phase 1.

Following phase 1, records are ordered (by columns) with respect to d_1. It remains to perform the ordering with respect to d_0. This is done independently for groups of records having the same d_1 value. For the example there are four such groups. Each group consists of two columns, $2i$ and $2i + 1$. The procedure for this ordering is the same as that for phase 1. Records are routed to row d_0 without changing the column (Figure 1d). Records are then ranked within each group and routed to the appropriate column in that group without changing rows (Figure 1e). This completes phase 2. At the end of this phase, record $G(i)$ is located in column $A(i)$. (In general, $\lceil n/m \rceil$ phases, as described above, will be needed to achieve this condition.) To obtain the desired final record distribution, the records are routed to row 0 without changing the column of any record (Figure 1f).

To summarize, our permutation algorithm consists of $\lceil n/m \rceil$ route-rank-concentrate phases, followed by a routing phase in which all records are routed to row 0.

2.2 Permutation Algorithm for a CCC. We now implement the permutation procedure described above. The algorithm for a CCC is specified first. We shall see later that the algorithm for a PSC can be easily obtained from that for a CCC. In specifying the algorithms, we make use of the following notation and assumptions:

(1) Each PE has three registers R, S, and T. $R(i)$, $S(i)$, and $T(i)$ refer to the corresponding registers in PE(i). In addition, each PE has enough memory to hold one record G (this includes the field A).
(2) We shall use the special symbol *null*. The only requirement is that it be possible to distinguish between *null* and a record (a one-bit tag could be used).
(3) Three types of assignments will be used:
    (a) := will be used for assignments requiring no routing. For example, $T(i) := S(i)$ (both T and S are in the same PE).
    (b) $\leftarrow$ will denote an assignment requiring a route. We shall require that the PEs denoted by the left- and right-hand sides be connected by a direct link in the PE interconnection pattern. For example, $T(i^{(b)}) \leftarrow S(i)$ is valid for a CCC (recall that $i^{(b)}$ is obtained from i by complementing bit b in the binary representation of i). Each assignment of this type will be referred to as a *unit-route.*
    (c) $\leftrightarrow$ will denote an exchange requiring a route. The requirements are the same as for $\leftarrow$, and the cost is also one unit-route (note that if two-way transmission is not allowed, this can be accomplished in two routes).
(4) i_b will denote bit b in the binary representation of i, and $(i)_{j:l}$ will denote the integer with the binary representation $i_j i_{j-1} \cdots i_{l+1} i_l$.
(5) PE selectivity can be done using a mask. The mask is specified in parentheses following the statement. Some examples of masks are:
    (i) ($i_b = 1$): this enables all PEs for which the binary representation of the PE index has bit b equal to 1.
    (ii) ($A(i) = null$): this enables all PEs with $A(i) = null$.

When no mask is specified, all PEs are enabled. Instructions are executed only on enabled PEs.

The complexity of an algorithm will be measured in terms of the PE time needed and the number of unit-routes. In keeping with the assumption made in most algorithm analysis, we assume that arithmetic operations can be performed in $O(1)$ time. While this is true only if all numbers are small enough to fit into one word each, it is quite appropriate in our case here, as our algorithms will not use numbers larger than the number of PEs. Hence, with hardware to perform 60-bit arithmetic, additions, comparisons, etc. of integers in the range $[-(2^{59} - 1), 2^{59} - 1]$ can be carried out in $O(1)$ time. So, our assumption will be valid so long as the number of PEs is no more than 2^{59}.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

FIG. 2. 2^2 blocks.

```
procedure RANK($k$)
    //Rank records in groups of $2^k$ consecutive PEs//
    global $A, R, S$
    //initialize for single PE groups//
    $R(i) := 0$
    $S(i) := 1, (A(i) \neq null)$
    $S(i) := 0, (A(i) = null)$
    for $b := 0$ to $k - 1$ do
        $T(i^{(b)}) \leftarrow S(i)$
        $R(i) := R(i) + T(i), (i_b = 1)$
        $S(i) := S(i) + T(i)$
    end
end RANK
```

FIG. 3. Algorithm 2.1.

```
procedure CONCENTRATE($k$)
    //route records to appropriate PEs as determined by $R$//
    global $G, R$  //$A$ is a field in $G$; $G$ denotes the record//
    for $b := 0$ to $k - 1$ do
        $(G(i^{(b)}), R(i^{(b)}) \leftarrow (G(i), R(i)), (A(i) \neq null$ and $R(i)_b \neq i_b)$
    end
end CONCENTRATE
```

FIG. 4. Algorithm 2.2.

Finally, let us define a 2^k-*block* as a sequence of 2^k PEs $\langle 2^k i, 2^k i + 1, \ldots, 2^k(i + 1) - 1\rangle$, where $i \in [0, 2^{n+m-k} - 1]$. All indices in a 2^k-block are equal in bits $n + m - 1, \ldots, k + 1, k$. The indices vary only in the least significant k bits. In a $2^m \times 2^n$ array of PEs, each row is a 2^n-block. Figure 2 shows 2^2-blocks in a $2^2 \times 2^3$ array. A 2^k-*column-block* consists of 2^k consecutive columns; each row of this block is a 2^k-block. The indices in a 2^k-column-block vary in the least significant k bits along each row and in the most significant m bits along each column. Bits $n - 1, \ldots, k + 1, k$ are invariant. In Figure 1c the dark lines partition the array into 2^1-column-blocks.

Before presenting the permutation algorithm PERMUTE we develop two subroutines that will be used by PERMUTE. One of these ranks records and the other concentrates them. Both RANK and CONCENTRATE operate on 2^k-blocks, $0 \leq k \leq n$.

The rank of a record in a 2^k-block is the number of records preceding it in that block. Procedure RANK determines this quantity for each record in every block. The working of RANK is best described recursively. Divide a 2^k-block into two 2^{k-1}-blocks. Let $R(i)$ be the rank of the record (if any) in PE(i) within the 2^{k-1}-block. Let $S(i)$ be the total number of records in the 2^{k-1}-block containing PE(i). Then the rank of a record in a 2^k-block is $R(i)$ if $i_{k-1} = 0$ (note that $i_{k-1} = 0$ for the left 2^{k-1}-block of a 2^k-block) and $R(i) + S(i^{(k-1)})$ if $i_{k-1} = 1$. Unfolding the recursion yields the iterative procedure RANK (see Figure 3). It is easy to see that RANK uses $O(k)$ PE time and exactly k unit routes.

Procedure CONCENTRATE (see Figure 4) concentrates records within each 2^k-block. Recall that during concentration, records are moved to consecutive PEs. Let

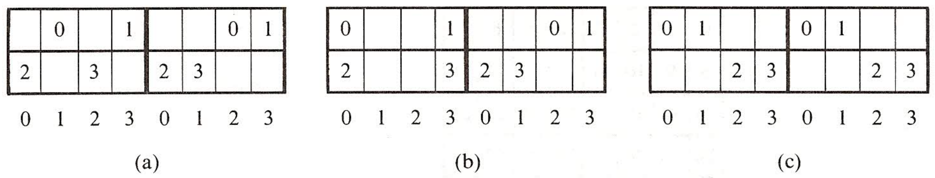


FIG. 5. Concentration with $n = 3$, $m = 1$, and $k = 2$.

$R(i)$ be such that the record (if any) in PE(i) is to be moved to the $R(i)$th PE in the 2^k-block. (The $R(i)$ used here differs from the $R(i)$ obtained by RANK by an additive constant to be determined later.) Concentration is carried out by first moving all records in the block to PEs such that the PE index and $R(i)$ agree in bit 0. The next routing ensures that PE indices and $R(i)$ agree in bits 0 and 1; and so on until records have been routed to the correct PEs.

Example 2.1. Let $n = 3$, $m = 1$, and $k = 2$. Figure 5a shows a possible initial configuration. There are four groups each containing exactly two records. The numbers in the PE boxes are the R values. The remaining PEs contain no record. The numbers outside the boxes identify the ith PE in a 2^k-block. The first iteration of the **for** loop of CONCENTRATE results in the configuration of Figure 5b. The second iteration yields Figure 5c and completes the concentration. □

The correctness of CONCENTRATE is not immediate. Suppose at the start of some iteration b, we have

$$A(i) \neq null \quad \textbf{and} \quad A(i^{(b)}) \neq null \quad \textbf{and} \quad R(i)_b = i_b \quad \textbf{and} \quad R(i^{(b)})_b \neq (i^{(b)})_b.$$

Then, during this iteration the record in PE(i) will get destroyed (overwritten) by the incoming record from PE($i^{(b)}$). This will be called a *collision* in PE(i). To establish the absence of collisions, consider a pair of records originating from PE(j) and PE(l) in the same 2^k-block. The rankings are such that $|j - l| \geq |R(j) - R(l)|$. These records will collide in PE(i) during iteration b only if

$$i = (j_{n+m-1:b+1}, R(j)_{b:0}) = (l_{n+m-1:b+1}, R(l)_{b:0}).$$

The right equality implies $|j - l| < 2^{b+1}$ and $|R(j) - R(l)| \geq 2^{b+1}$. Hence $|j - l| < |R(j) - R(l)|$, which contradicts the earlier inequality. Therefore no two records will ever collide, and so every record will get to its destination.

As far as the complexity of CONCENTRATE is concerned, the number of unit-routes is k (assuming that both G and R can be routed in one route).

We are now ready to discuss algorithm PERMUTE (see Figure 6). As stated earlier, the permutation is performed in $\lceil n/m \rceil$ phases (lines 3–13 of PERMUTE). In phase s, the records $G(i)$ are sorted using the digit $D(i) = A(i)_{k-1:r}$ as the sort key, where $k = n - (s - 1)m$ and $r = \max(k - m, 0)$. The sorting in each phase is carried out independently for each 2^k-column-block. The sort is accomplished by first routing each record, $G(i)$, to row $D(i)$ in the same column. (As will become apparent, at the start of each phase there is exactly one record per column.) This routing is achieved by lines 4–8 of PERMUTE. First, in lines 4–6 the record in each column is replicated over all PEs in that column. Then, in lines 7–8 all copies of a record in a column are deleted except for the one in the proper row. (The row number of PE(i) is $\lfloor i/2^n \rfloor$.)

The next step is to rank the 2^k records in a 2^k-column-block. First, records are ranked within each 2^k-block (line 9). Since each row of a 2^k-column-block contains

```
line procedure PERMUTE($n$, $m$)
         //permute $2^n$ records on a $2^m \times 2^n$ CCC according to field $A$ of $G$//
         global $G$, $R$
 1       $A(i) :=$ null, $((i)_{n+m-1:n} \neq 0)$  //initialize remaining rows//
 2       $k := n$  //$2^k$ is the number of columns in a group//
 3       for $s := 1$ to $\lceil n/m \rceil$ do  //$\lceil n/m \rceil$ phases of radix sort//
 4           for $b := n$ to $n + m - 1$ do  //copy records over columns//
 5               $G(i^{(b)}) \leftarrow G(i)$, $(A(i) \neq$ null$)$
 6           end
 7           $r :=$ max$(k - m, 0)$  //bits $k - 1, \ldots, r$ form the digit//
 8           $A(i) :=$ null, $(\lfloor i/2^n \rfloor \neq (A(i))_{k-1:r})$
 9           call RANK($k$)
10           $R(i) := R(i) + \lfloor i/2^n \rfloor * 2^r$
11           call CONCENTRATE($k$)
12           $w := k$; $k := r$  //each partition is $2^r$ columns now//
13       end
         //move records to first row//
14       for $b := n$ to $n + w - 1$ do  //elements are in top $2^w$ rows//
15           $G(i^{(b)}) \leftarrow G(i)$, $(A(i) \neq$ null$)$
16       end
17   end PERMUTE
```

FIG. 6. Algorithm 2.3.

exactly 2^r records (this follows from the fact that A is a permutation), the rank of a record in any 2^k-column-block is given by the right-hand side of line 10. The concentration of line 11 partitions the records from the 2^k-column-block into 2^r-column-blocks, each partition containing records whose A values differ only in the least significant r bits. In addition, the $(A)_{n-1:r}$ values increase from one partition to the next (left to right). That is, after the concentration we have $(A(i))_{n-1:r} = (i)_{n-1:r}$ for every record $G(i)$. So, following the last concentration phase, where $r = 0$, we will have $(A(i))_{n-1:0} = (i)_{n-1:0}$. This means that each record has been routed to its appropriate column and now needs only to be routed to the PE in row 0 of that column. This is accomplished in lines 14–16. (Actually this code results in the record being replicated over all PEs in rows 0 to $2^w - 1$ of the column containing it, where $w \leq m$ is the number of bits in the last digit.)

Let $f(n, m)$ be the number of unit-routes required by PERMUTE to perform a permutation. The number of routes for lines 14–16 is $w = n - (\lceil n/m \rceil - 1)m$. So, the total contribution of lines 4–6 and lines 14–16 to $f(n, m)$ is $\lceil n/m \rceil m + w = n + m$. Lines 9 and 11 each contribute k per iteration of the loop of lines 3–13. Substituting for k, and recalling that $m \leq n$, we get

$$\begin{aligned} f(n, m) &= n + m + 2(n + (n - m) + (n - 2m) + \cdots + w) \\ &\leq n + m + (\lceil n/m \rceil + 1)n \\ &\leq (\lceil n/m \rceil + 3)n. \end{aligned}$$

The number of unit-routes is therefore $O(k \log N)$ when $N^{1+1/k}$ PEs are available. (The number of records to be permuted is $N = 2^n$, and $k = n/m$.) The PE processing time is $O(f(n, m))$.

It should be pointed out that PERMUTE can also be used to perform any *subpermutation*. Let $\langle A(0), A(1), \ldots, A(N - 1) \rangle$ define the desired subpermutation: $A(i) =$ *null* if PE(i) is not *transmitting* data. (Note that PE(i) *transmits* its data if $A(i) = j$ for some j; it *receives* data if $A(j) = i$ for some j. A given PE may only transmit, only receive, or both transmit and receive.) The fact that PERMUTE will correctly handle subpermutations may be easily verified.

2.3 PERMUTATIONS ON A PSC. The permutation algorithm for a PSC is essentially a simulation of that for a CCC. The loop of lines 4–6 of PERMUTE can be carried out as below:

```
for $b := n$ to $n + m - 1$ do
    shuffle on $G$
end
for $b := n$ to $n + m - 1$ do
    $G(i^{(0)}) \leftarrow G(i), (A(i) \neq null)$
    unshuffle on $G$
end
```

During each shuffle (unshuffle), the data in the G space of a PE are routed along the shuffle (unshuffle) connection. The number of unit-routes needed for this simulation of lines 4–6 of PERMUTE is $3m$. In a similar manner, RANK and CONCENTRATE can be implemented to run on a PSC using $3k$ unit-routes. Lines 14–16 of PERMUTE can be carried out in $3w$ unit-routes. So, permutations on a PSC require at most three times as many unit-routes as they do on a CCC. Let $h(n, m)$ be the number of unit-routes needed by the PSC permutation algorithm, $h(n, m) \leq 3f(n, m) \leq 3(\lceil n/m \rceil + 3)n$.

Thus, even though a CCC has $(n + m)/3$ times as many connections per PE as a PSC, the permutation algorithm is only three times as fast!

2.4 LEAST-SIGNIFICANT-DIGIT (LSD) RADIX SORT. The permutation algorithm described so far was based on MSD radix sort. This confined the data routings to smaller regions of PEs in successive phases of the algorithm. More specifically, after phase s, $1 \leq s \leq \lfloor n/m \rfloor$, all routings become local to each 2^{n-sm}-column-block. As a result, the number of unit-routes in RANK and CONCENTRATE become successively smaller.

It should be apparent by now that a permutation algorithm could also be obtained using LSD radix-sort. This would yield a conceptually simpler but less efficient algorithm. In each of the $\lceil n/m \rceil$ phases, data is vertically routed as before (starting with the least significant digit, of course). Then, the elements in each *row* are ranked and concentrated, this in every phase being identical to the first phase of the MSD algorithm. That is, RANK and CONCENTRATE always operate on 2^n-blocks rather than successively smaller blocks. The resulting LSD algorithm will still run in $O((n/m)\log N)$ time, though slower than the corresponding MSD algorithm by a constant factor.

However, the LSD radix-sort technique offers more generality: It can be used to *sort* arbitrary (but bounded) *integers*. Assume $N = 2^n$ integers in the range $[0, 2^q - 1]$ for some q. (We have 2^{n+m} PEs.) The LSD sorting algorithm will have $\lceil q/m \rceil$ phases. Each phase starts with vertical routing as before. Now, in contrast to the permutation algorithm, the rows will not in general end up with an equal number of elements. The data compression in each phase is performed by the following code:

```
(1) call RANK($n + m$)
(2) call CONCENTRATE($n$)
```

The first line computes the rank of each element with respect to the entire set of PEs, that is, the rank within the 2^{n+m}-block. The second line concentrates the elements within each row. The complexity of the resulting algorithm on a CCC is readily determined. The number of unit-routes for vertical routing, ranking, and concentrating is respectively m, $n + m$, and n (assuming one-word representation for the integers). This gives $2(n + m)$ routes for each phase. After completing the $\lceil q/m \rceil$ phases, the elements must be routed up to row 0. (See lines 14–16 of PERMUTE.) This will require w' unit-routes, where $w' = q - (\lceil q/m \rceil - 1)m \leq m$. So, the total number of unit-routes is $2(n + m)\lceil q/m \rceil + w' = O((q/m)\log N)$. The number of unit-routes on a PSC is, of course, of the same order.

Unfortunately, the more efficient MSD algorithm does not lend itself to the sorting of arbitrary integers. This is because the PE array after the first phase will be partitioned into groups of adjacent columns, with different number of columns in different groups. This creates difficulties in subsequent ranking and concentration.

The next section will present a more general sorting algorithm for arbitrary *numbers.*

3. *Sorting*

$N = 2^n$ elements can be sorted in $O(k \log N)$ time on a CCC or PSC having $N^{1+1/k}$ PEs. The algorithm is quite similar to that developed by Preparata [18] for the shared memory model. Preparata's algorithm consists of three distinct steps:

(1) *Counting.* Divide the N-element sequence into some number j of subsequences. Sort each subsequence recursively. Then, for each element A_i in subsequence l determine the number, C_{ip}, of elements in subsequence p that are

  (a) no greater than A_i, $p < l$,
  (b) to the left of A_i, $p = l$,
  (c) less than A_i, $p > l$.

(2) *Ranking.* The rank, R_i, of the element A_i is its final position in the sorted sequence. This is just the sum of the counts for that element. That is, $R_i = \sum_p C_{ip}$.

(3) *Routing.* Each element is routed to the PE corresponding to its rank.

We shall show that these three steps can be implemented on a CCC or PSC to obtain a sorting algorithm with the same asymptotic complexity as that of Preparata's algorithm. We shall describe our implementation explicitly for a CCC only. The PSC algorithm may be obtained from the CCC algorithm as described in Section 2. We assume that 2^{n+m}, $1 \leq m \leq n$, PEs are available. These PEs may be viewed as arranged in a $2^m \times 2^n$ array. The elements to be sorted are $A(0\!:\!N - 1)$, and initially the ith element is in PE(i). For the remaining PEs, $A = null$.

Before discussing the sorting algorithm (procedure SORT), we introduce some terminology. PE(i) is a *diagonal* PE under 2^r-blocking iff

$$\lfloor i/2^n \rfloor = \lfloor i/2^r \rfloor \bmod 2^m. \tag{3.1}$$

Figure 7a shows the diagonal PEs (labeled D) when $r = 2$, $m = 2$, and $n = 4$. Figure 7b shows the diagonal PEs when $r = 1$, $m = 2$, and $n = 4$. Each sequence of 2^r diagonal PEs forms a *diagonal* 2^r*-block.* A *left* PE is one for which

$$\lfloor i/2^n \rfloor > \lfloor i/2^r \rfloor \bmod 2^m, \tag{3.2}$$

and a *right* PE is one for which

$$\lfloor i/2^n \rfloor < \lfloor i/2^r \rfloor \bmod 2^m. \tag{3.3}$$

Let DIAG(r, i), LEFT(r, i), and RIGHT(r, i) be Boolean functions that are true iff (3.1), (3.2), and (3.3), respectively, hold for r and i.

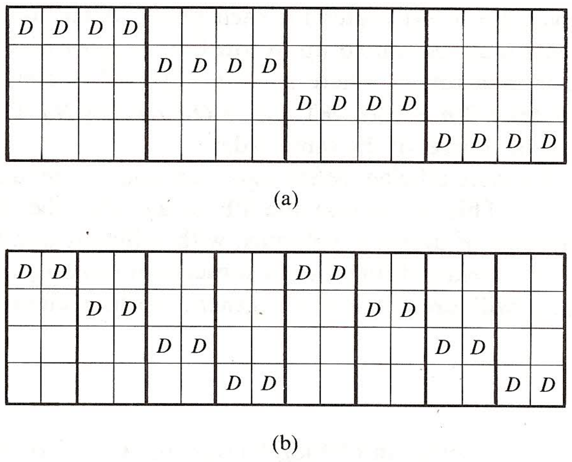


FIG. 7. Diagonal PEs under 2^r-blocking. (a) $r = 2$. (b) $r = 1$.

3.1 SORT. Our sorting algorithm consists of $\lceil n/m \rceil$ phases. Let $w = n - (\lceil n/m \rceil - 1)m$. In phase 1, each subsequence of length 2^w is sorted. This is done in each 2^w-column-block. (Only 2^w rows of PEs are needed for the first phase.) Each of the remaining $\lceil n/m \rceil - 1$ phases performs 2^m-way merges to obtain successively larger sorted subsequences. Phase s, $2 \leq s \leq \lceil n/m \rceil$, starts with sorted subsequences of length 2^r, $r = w + m(s - 2)$. Each subsequence resides in a 2^r-column-block, one item per column. At the end of this phase we end up with sorted subsequences of length 2^{r+m}, each in a 2^{r+m}-column-block. (It is interesting to note that the sorting algorithm runs somewhat the "reverse" of our MSD permutation algorithm.)

Procedure SORT (see Figure 8) is now examined more closely. The $\lceil n/m \rceil$ phases are performed by lines 2–22. Each phase starts with sorted sequences of size 2^r and ends with sorted sequences of size 2^k. (For the first phase, $r = 0$ and $k \leq m$. For all other phases, $k = r + m$.) First, lines 3–6 replicate the single element in each column over the entire column. At this point, every 2^r-block contains a sorted sequence of length 2^r (in the S registers); all blocks in the same 2^r-column-block contain copies of the same sequence. Lines 7–9 choose the sequence in each diagonal 2^r-block; lines 10–12 replicate this sequence horizontally in the containing 2^k-block. (Each 2^k-block contains one diagonal 2^r-block with a sorted sequence. This is copied to all nondiagonal 2^r-blocks.) The replicated data is placed in the T registers. Now every 2^r-block contains a sorted sequence in its S registers and another in its T registers. For a 2^r-block in the ith group and jth row, the S sequence is from group i and the T sequence from group j. Procedure COUNT (to be specified later) determines the count $R(i)$ corresponding to each element $S(i)$, as discussed earlier in our informal presentation of counting.

The rank of each element is then determined by summing up the count values for that element. This is done by summing the count values in each column (lines 14–16).

The final step in each phase is to route each element to the appropriate column. Recall that each row of a 2^k-column-block (i.e., each 2^k-block) contains one diagonal 2^r-block. So, the elements from the 2^r-block are *spread out* within the 2^k-block (while preserving their relative order). This is just the inverse of concentration, and so the results of Section 2 establish the correctness of lines 17–19. Now each 2^k-column-block contains a sorted sequence; the ith element of the sequence resides somewhere in the ith column.

```
line procedure SORT($n$, $m$)
          //Sort $N = 2^n$ records on a $2^m \times 2^n$ CCC//
     global $A, R, S, T, m, n$
 1   $r := 0$; $k := n - (\lceil n/m \rceil - 1)m$
 2   for $s := 1$ to $\lceil n/m \rceil$ do  //$\lceil n/m \rceil$ phases of multiway merge//
          //Merge sorted sequences of length $2^r$ to get//
          //sorted sequences of length $2^k$. Use $2^{k-r}$ rows.//
          //Counting//
 3        $S(i) := A(i)$
 4        for $b := n$ to $n + k - r - 1$ do  //copy vertically//
 5             $S(i^{(b)}) \leftarrow S(i)$, $(S(i) \neq null)$
 6        end
 7        $T(i) := S(i)$, (DIAG($r$, $i$))
 8        $T(i) := null$, (not DIAG($r$, $i$))
 9        $A(i) := T(i)$  //$A$ is diagonalized//
10        for $b := r$ to $k - 1$ do  //copy each diagonal $2^r$-block horizontally//
11             $T(i^{(b)}) \leftarrow T(i)$, $(T(i) \neq null)$
12        end
13        call COUNT($r$)  //merge-and-unmerge in each $2^r$-block//
          //ranking//
14        for $b := n$ to $n + k - r - 1$ do  //column sum//
15             $R(i) \leftarrow R(i) + R(i^{(b)})$
16        end
          //routing//
          //spread each diagonal $2^r$-block within its $2^k$-block//
17        for $b := k - 1$ down to 0 do
18             $(A(i^{(b)}), R(i^{(b)})) \leftarrow (A(i), R(i))$, $(A(i) \neq null$ and $(R(i))_b \neq i_b)$
19        end
20        $r := k$  //size of sorted sequences//
21        $k := k + m$  //$2^m$-way merge on next phase//
22   end
23   for $b := n$ to $n + m - 1$ do  //route to the first row//
24        $A(i^{(b)}) \leftarrow A(i)$, $(A(i) \neq null)$
25   end
26   end SORT
```

FIG. 8. Algorithm 3.1.

After $\lceil n/m \rceil$ phases the entire sequence is sorted. Lines 23–25 route the single element in each column up to the first row. (The code of lines 23–25 actually replicates each element over the whole column.)

3.2 COUNT. Procedure COUNT considers each 2^r-block of PEs in parallel. Each such block contains a sorted S and T sequence. For each element $S(i)$ in the block, the procedure is to find the corresponding count $R(i)$. If the 2^r-block is a diagonal block, then the S and T sequences are identical. The count, R, for the ith S in a diagonal 2^r-block is i. If the 2^r-block is a left block, then the count for the ith S is the number of T values in the T sequence that are less than it. When the 2^r-block is a right block, then the count for the ith S is the number of T values not greater than it. The count R is determined by merging the S and T sequences together (a stable merge is used). Figure 9 shows an S and T sequence. The arrows from row 1 to row 2 show where an S or T element gets positioned following the merge. It should be easy to see that if the ith S value is located in position j following the merge, then its R value is $j - i$. The third row of Figure 9 gives the R values for the S sequence.

The S and T sequences are merged using Batcher's bitonic merge [9]. Informally speaking, a bitonic merge sorts a bitonic sequence. For our purposes it is sufficient to know that a sequence of nondecreasing numbers followed by a sequence of

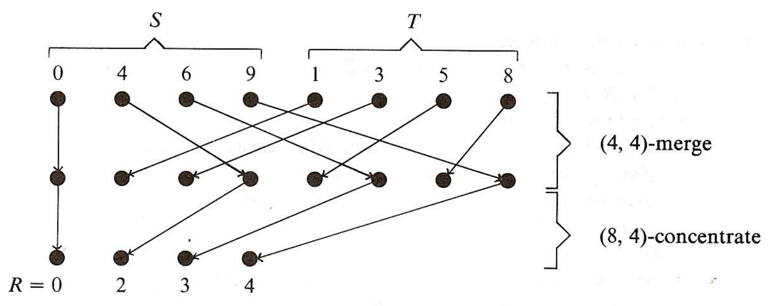


FIG. 9. Sketch of COUNT.

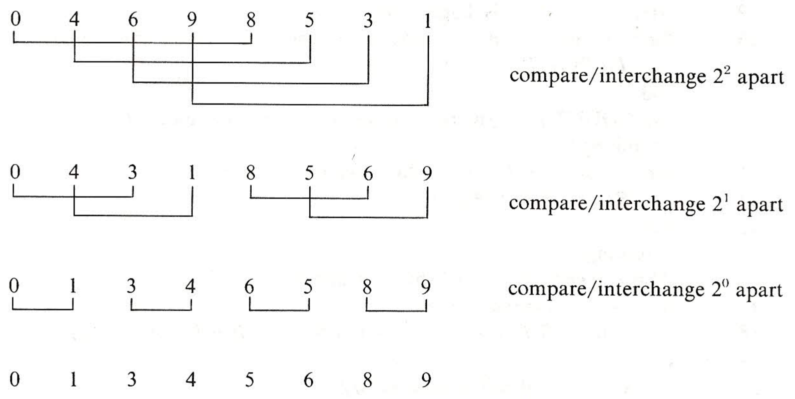

0 4 6 9 8 5 3 1
compare/interchange 2^2 apart
0 4 3 1 8 5 6 9
compare/interchange 2^1 apart
0 1 3 4 6 5 8 9
compare/interchange 2^0 apart
0 1 3 4 5 6 8 9

FIG. 10. Batcher's bitonic merge.

nonincreasing numbers is a bitonic sequence. A bitonic sequence of length 2^p can be arranged into nondecreasing order as follows:

```
for $i := p - 1$ downto 0 do
    compare pairs of elements $2^i$ apart and interchange them if the one on the left is larger than the one
    on the right
end
```

The S and T sequences in each 2^r-block together form a 2^{r+1} sequence (if we consider the T sequence as following the S sequence as in Figure 9). This 2^{r+1} sequence is not bitonic. However, it can be made bitonic by reversing the T sequence. The first row of Figure 10 shows the resulting bitonic sequence. The remaining rows of this figure show the progress of the bitonic merge algorithm.

The implementation of the outlined procedure, COUNT (see Figure 11), is straightforward, though tedious. Lines 1–13 of COUNT merge S and T. First, T is reversed in lines 1–3. In line 4 the original position of S is saved in S'. This allows us to route the final position of an S back to its originating PE. In lines 5 and 6, T' is set so that a stable merge (with respect to the original element positions) of S and T can be carried out. Observe that for i and j in the same 2^r-block, $S'(i) < T'(j)$ iff the T sequence originates from the right of the S sequence. Otherwise, $S'(i) > T'(j)$. Lines 7–13 implement Batcher's bitonic merge. Note that, initially, elements 2^r apart are in the S and T registers of the same PE. These elements are compared and interchanged (if necessary) in line 9. On the next iteration we need to compare elements 2^{r-1} apart.

```
line procedure COUNT($r$)
          //Determine the count for each $S$ in a $2^r$-block//
          global $T, S, R$
   1      for $b := 0$ to $r-1$ do   //reverse the $T$ sequences//
   2          $T(i^{(b)}) \leftrightarrow T(i)$
   3      end
   4      $S'(i) := i \bmod 2^r$   //save initial position//
   5      $T'(i) := 2^r$, (LEFT($r, i$))   //needed to handle equal elements//
   6      $T'(i) := -1$, (RIGHT($r, i$))
          //bitonic merge//
   7      $b := r - 1$
   8      loop   //:=: denotes an interchange within a PE//
   9          $(S(i), S'(i)) :=: (T(i), T'(i))$,
                   $(S(i) > T(i)$ or $(S(i) = T(i)$ and $S'(i) > T'(i)))$
  10          if $b < 0$ then exit   //go to line 14//
  11          $(T(i^{(b)}), T'(i^{(b)})) \leftrightarrow (S(i), S'(i)), (i_b = 1)$
  12          $b := b - 1$
  13      repeat
  14      $S(i) := 2(i \bmod 2^r)$   //Final position//
  15      $T(i) := S(i) + 1$
          //destroy old $T$ values//
  16      $S(i) := null, (S'(i) = 2^r$ or $S'(i) = -1)$
  17      $T(i) := null, (T'(i) = 2^r$ or $T'(i) = -1)$
          //route final $S$ locations back to original PEs//
  18      for $b := 0$ to $r-1$ do
  19          $(S(i), S'(i)) :=: (T(i), T'(i))$,
                   $((S(i) \neq null$ and $(S'(i))_b = 1)$ or $(T(i) \neq null$ and $(T'(i))_b = 0))$
  20          $(T(i^{(b)}), T'(i^{(b)})) \leftrightarrow (S(i), S'(i)), (i_b = 1)$
  21      end
  22      $(S(i), S'(i)) := (T(i), T'(i)), (T(i) \neq null)$
  23      $R(i) := S(i) - S'(i)$   //count//
  24      $R(i) := i \bmod 2^r$, (DIAG($r, i$))
  25  end COUNT
```

FIG. 11. Algorithm 3.2.

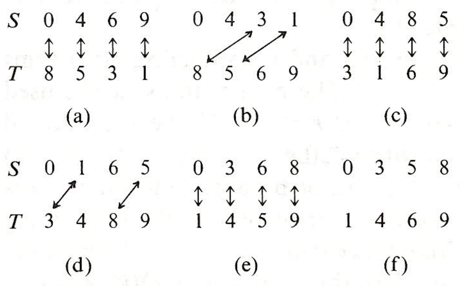


FIG. 12. Merging S and T sequences. (a) Compare 2^2 apart. (b) Exchange of line 9. (c) Compare 2^1 apart. (d) Exchange of line 9. (e) Compare 2^0 apart. (f) Final result.

The interchange of line 11 moves elements 2^{r-1} apart into the same PE. Figure 12 shows the progress of this loop for the example of Figure 10. When the loop is completed, the merged sequence is $S(0)$, $T(0)$, $S(1)$, $T(1)$, $S(2)$, $T(2)$, ... (PE indices are modulo 2^r). From this it follows that lines 14 and 15 compute the final position of each element.

Now the final position of each S element has to be routed back to the PE originally containing that element. First the S and T values that correspond to elements that were originally in T are destroyed (lines 16 and 17). The final positions for the S elements are then routed back in lines 18–22. The correctness of this routing may be established by observing that the path traversed by an S element during the merge (lines 8–13) is traversed backward during the routing of lines 18–22. In line 19 the (S, S') registers are loaded with data to be routed to a PE with bit $b = 0$, and the (T, T') registers are loaded with data to be routed to a PE with bit $b = 1$. This routing is the reverse of that done in line 9. Similarly, the routing performed by line 20 is the reverse of that done by line 11. Having routed the final positions back to the originating PEs, lines 23–24 compute the count R. The count for nondiagonal PEs is computed in line 23 and that for diagonal PEs in line 24.

3.3 COMPLEXITY OF SORT. We first observe that COUNT needs $3r$ unit-routes. Let $f'(n, m)$ be the number of unit-routes needed by SORT. Lines 3–21 of SORT (i.e., one phase) use $4k$ unit-routes. So,

$$\begin{aligned} f'(n, m) &= 4\{n + (n - m) + (n - 2m) + \cdots + n - (\lceil n/m \rceil - 1)m\} + m \\ &\le 2n(\lceil n/m \rceil + 1) + m \\ &\le 2n(\lceil n/m \rceil + 1.5). \end{aligned}$$

(Recall that $m \le n$.)

Procedures SORT and COUNT may be implemented on a PSC using the transformations described earlier in connection with procedure PERMUTE. The number of unit-routes needed will be at most $3f'(n, m)$. The computing time for the sorting algorithm on both CCCs and PSCs is $O(f'(n, m))$.

3.4 COMPLEXITY OF DATA BROADCASTING. An immediate consequence of our sorting results is that "data broadcasting" among N PEs of a CCC or PSC can also be carried out in $O(k \log N)$ time when $N^{1+1/k}$ PEs are available. Let $D(i)$ be a data item in PE(i), $0 \le i \le N - 1$. One form of data broadcasting treated in [17] is called Random-Access-Read (RAR). An RAR is defined by a vector $\langle R(0), \ldots, R(N-1) \rangle$ with $R(i)$ residing in PE(i); $R(i) \in \{0, 1, \ldots, N-1, \mathit{null}\}$. PE($i$) is to receive its data from PE($R(i)$), $0 \le i \le N - 1$. That is, $D(i) \leftarrow D(R(i))$. If $R(i) = \mathit{null}$, then PE(i) receives no data. Note that many PEs may request data from the same PE. The RAR algorithm of [17] involves a sorting step and runs in $O(\log^2 N)$ time on an N-PE CCC or PSC. When $N^{1+1/k}$ PEs are available, the sorting algorithm developed here may be used to perform an RAR in $O(k \log N)$ time [17].

So far, we have described the permutation, sorting, and broadcasting algorithms for performing transfers among PEs $0, 1, \ldots, N - 1$. These algorithms can be used to transfer records among *any* N PEs. Let $G(i_r)$, $0 \le r \le N - 1$, be the selected records, where $G(i_r)$ is in PE(i_r). We first "concentrate" the records (i.e., bring $G(i_r)$ to PE(r)). Then we apply the algorithm (permute, sort, or broadcast) to the records in PEs $0, 1, \ldots, N - 1$. Afterward, we "spread out" the records to the originating PEs. That is, the record in PE(r) is sent to PE(i_r). Since concentrating and spreading require $O(n + m) = O(\log N)$ steps, the overall algorithm will still be $O(k \log N)$.

4. *A Generalized Connection Network*

An (N, N) generalized connection network (GCN) is a switching network capable of connecting any subset of its N inputs to any subset of its N outputs. An input may be connected to many outputs, but each output can be connected to at most one input. A GCN may be represented as a directed graph in which edges represent switches. A switch may be either "on" ("closed") or "off" ("open"). Any desired connection of inputs to outputs in a GCN corresponds to a subgraph of the graph representing the GCN. This subgraph includes all edges (or switches) in the on state. Input vertex i is connected to output j if there is a path from i to j in the subgraph just described.

(Note that paths starting from distinct input vertices must be disjoint.) The *fan-out* and *fan-in* of a GCN are, respectively, the outdegree and indegree of the corresponding graph. The number of *contact pairs* in a GCN is the number of *edges* in its graph. The *delay* of a GCN is defined as the maximum number of edges on any input–output path. The *setup* time is defined as the time needed to obtain the on/off state of the switches (edges) to establish the desired connection.

Once a GCN is set up, data may be transmitted from inputs to outputs, the transmission time being proportional to the "delay." Transmission from each input port may proceed independently (i.e., asynchronously). In fact, the inputs may be *analog* signals (assuming the appropriate conducting media, of course). A GCN may have applications in telephone switching or telecommunication. It may also be used as the interconnection network in a parallel processor or a distributed system.

An $N \times N$ crossbar switch is an example of a GCN with $O(N^2)$ contact pairs and unit-delay. (The delay is $O(\log N)$ when fan-out and fan-in are restricted to a constant.) A crossbar switch is easily set up. Thompson's GCN [22] has $O(N \log N)$ contact pairs and $O(\log N)$ delay but is difficult to set up. The fastest known algorithm to set up his GCN runs in $O(k \log^3 N)$ time on an $N^{1+1/k}$-PE CCC or PSC. (See [16].)

On the basis of our MSD permutation algorithm we propose a GCN that is relatively easy to set up. Let $N = 2^n$, $m \in [1, n]$, and $k = n/m$. We construct an (N, N)-GCN with $O(kN^{1+1/k} \log N)$ contact pairs and $O(k \log N)$ delay. Our GCN can be set up in $O(k \log N)$ time using an $N^{1+1/k}$-PE CCC or PSC. So, for situations in which the switch settings have to be computed on-line, our GCN will have a lesser total delay (i.e., time to compute switch settings + GCN delay time) than that of Thompson.

Our GCN can be used in an SIMD computer for PE-to-PE or PE-to-memory interconnection. Suppose, for example, that an $N^{1+1/k}$-PE PSC is also interconnected by the GCN. PE(i) is connected to input i and output i of the GCN, $0 \leq i \leq N - 1$. From the previous sections we know that the PSC itself can perform arbitrary one-to-many transfers between N PEs in $O(k \log N)$ *routing steps.* We will see later that it takes about the same amount of time for the PSC to set up the GCN. The transmission time on the GCN is $O(k \log N)$ *gate delays*, a "gate delay" being much shorter in time than a "routing step." So, if the records to be transmitted are only one or two words each, the PSC itself is used to do the transfer. With longer records, however, the GCN may provide a faster transmission. (If the records are not in PEs $0, 1, \ldots, N - 1$, they are first "concentrated"; then they are transmitted through the GCN; afterward, they are "spread out" to the originating PEs.) For long records, the GCN transmission time can be further reduced by *pipelining*. In this mode, the second and subsequent words will suffer only $O(1)$ delay. The bandwidth of the GCN's lines need not be full-word; even bit-serial transmission on the GCN can be faster than the PSC itself. Our GCN may also be useful for other "circuit-switching" applications if the setup time is crucial.

The construction of our GCN is given in Section 4.1. The setup algorithms are presented in Sections 4.2 and 4.3. Section 4.4 provides some comments on the structure of the GCN's *control memory*. (The on/off state of the switches are stored in this memory. Each bit of this memory controls one switch.)

4.1 THE GCN CONSTRUCTION. Our GCN construction is shown diagramatically in Figure 13. $N = 2^n$ is the number of inputs and outputs, m is an integer in the range $[1, n]$, and $M = 2^m$. A higher value of m results in a GCN with more edges and less delay. We shall restrict the fan-out and fan-in of our GCN to 2.

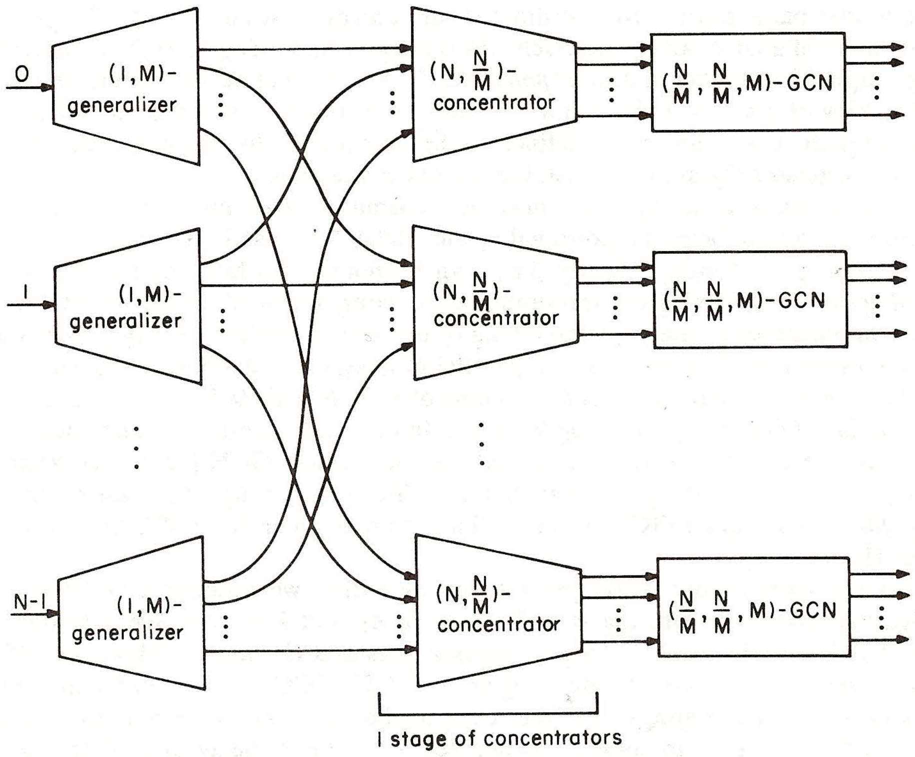


FIG. 13. (N, N, M)-GCN; $N = 2^n$, $M = 2^m$, $1 \le m \le n$.

A (1, M)-generalizer is a full binary tree with M leaf vertices. (Note that the tree has $\log_2 M + 1$ levels of vertices. If we allow a fan-out of more than 2, the generalizer can be simply a two-level tree of degree M.) For the generalizer, the switches are always *on*, as its function is to make M "copies" of its input. (That is, the input of the generalizer is always connected to all of its M outputs.)

An (N, N/M)-concentrator has N inputs and N/M outputs. Its function is to connect any p of its inputs to its top p outputs, $0 \le p \le N/M$. The connection of inputs to outputs is one-to-one and preserves the relative order. This concentrator is obtained from the (N, N)-concentrator of [22]. For completeness, the latter is described below.

The (N, N)-concentrator of [22] (Figure 14) has N inputs and N outputs. It can connect any subset of its inputs to a consecutive subset of its outputs (i.e., not necessarily starting with the top output). The connection is one-to-one and preserves the relative order of the inputs. From Figure 14 we see that an (N, N)-concentrator consists of $\log N + 1$ columns of N vertices. Let us label the vertices as $V^b(i)$, $0 \le i \le N - 1$, $0 \le b \le \log N$. There are two edges incident from each vertex $V^b(i)$, $0 \le b < \log N$: one edge goes to $V^{b+1}(i)$ and the other to $V^{b+1}(i^{(b)})$. (Recall that $i^{(b)}$ differs from i only in bit b.) Figure 15 shows an example connection on an (8, 8)-concentrator. Only the *on* switches (edges) are shown. R_i is the rank of input i. We wish to connect input i to output R_i. The path $\langle i, R_i \rangle$ is determined left to right. Bit b of R_i determines the edge from column b to column $b + 1$. (See Figure 15.)

An (N, N/M)-concentrator is obtained from an (N, N)-concentrator in the following way. Only the top N/M outputs of the (N, N)-concentrator are needed; so we remove the edges (switches) that cannot lead to any of these outputs. Figure 16

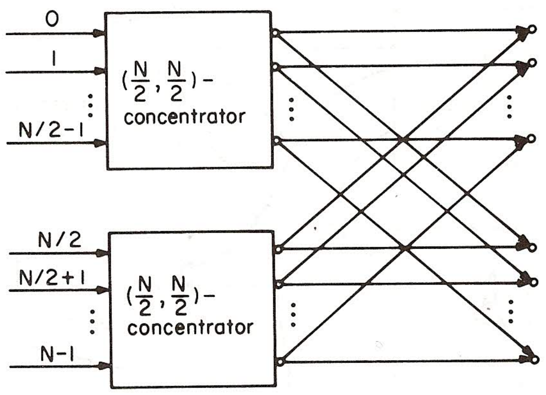


FIG. 14. (N, N)-concentrator; $N = 2^n$.

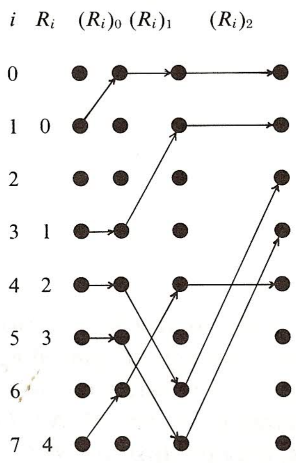


FIG. 15. A connection on an (8, 8)-concentrator.

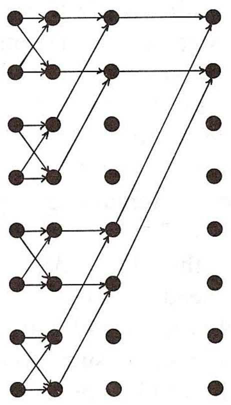
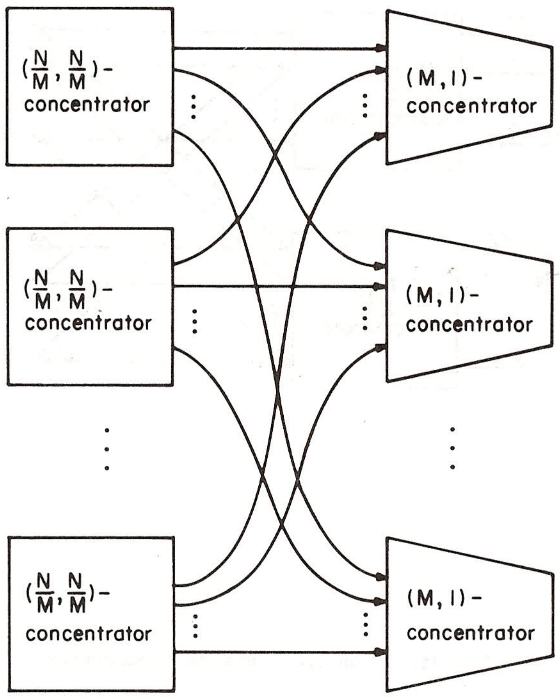


FIG. 16. An (8, 2)-concentrator.

shows an (8, 2)-concentrator constructed from an (8, 8)-concentrator. Observe that an (8, 2)-concentrator is really four (2, 2)-concentrators connected by two full binary trees of height 3 (i.e., these binary trees are the reverse of $(1, M)$-generalizers). One can easily show that an $(N, N/M)$-concentrator obtained as described above is M $(N/M, N/M)$-concentrators connected together by N/M binary trees of height $\log M + 1$ (see Figure 17). These full binary trees will be called $(M, 1)$-concentrators. All switches in an $(M, 1)$-concentrator need only one state, that is, *on*. (At most one



$(\frac{N}{M},\frac{N}{M})$-concentrator
$(\frac{N}{M},\frac{N}{M})$-concentrator
$(\frac{N}{M},\frac{N}{M})$-concentrator
(M,1)-concentrator
(M,1)-concentrator
(M,1)-concentrator

FIG. 17. An $(N, N/M)$-concentrator.

of the inputs to an $(M, 1)$-concentrator will be active; the other inputs will not get connected through the preceding stage of $(N/M, N/M)$-concentrators. So, an $(M, 1)$-concentrator can safely connect *all* of its inputs to its output.)

To summarize, an $(N, N/M)$-concentrator has $\log N$ levels of edges. Assuming $N > M$, only the first $\log(N/M)$ levels need to get set up; the remaining $\log M$ levels are always *on.* The setup rule for the case $N = M$ is slightly different. From Figure 13 we observe that an (N, N, N)-GCN (which is called an $N \times N$ *crossbar*) consists of a stage of $(1, N)$-generalizers followed by a stage of $(N, 1)$-concentrators. (The third stage of Figure 13 becomes null as $N = M$.) All switches in the $(1, N)$ generalizer stage are *on* as always. Therefore, the $(N, 1)$-concentrator stage can not have all of its switches *on.* In fact, each $(N, 1)$-concentrator needs to connect *at most* one of its inputs to its output, as all of its inputs are now active. This is done by setting up the first level of switches in an $(N, 1)$-concentrator; the remaining $\log N - 1$ levels of switches will still be *on.*

We have shown the recursive construction of the (N, N, M)-GCN for $N \geq M$ (Figure 13). To complete the construction, we need to specify the GCN for $1 \leq N < M$. When $1 < N < M$, the (N, N, M)-GCN is replaced by an (N, N, N)-GCN. And a $(1, 1, M)$-GCN is a null switch as stated earlier. In summary, the (N, N, M)-GCN consists of $\lceil n/m \rceil$ stages of generalizers followed by concentrators.

The correspondence between the GCN just described and PERMUTE (Algorithm 2.3) is readily established. Suppose that the GCN is to be used to perform a permutation on its inputs. The generalizer-concentrator stage of Figure 13 corresponds to the first phase of PERMUTE. The $(1, M)$-generalizers simply perform the routings of lines 4–6 of PERMUTE. (The M outputs of a generalizer may be regarded as the PEs in a column of a CCC.) The $(N, N/M)$-concentrators perform the concentration of records from each row ($N = 2^n$ PEs) to groups of $N/M = 2^{n-m}$ PEs (i.e., line 11 of PERMUTE). From this point on, the GCN (and PERMUTE) route the N/M outputs of each $(N, N/M)$-concentrator independently.

TABLE I. CHARACTERISTICS OF AN (N,N,M)-GCN

	Delay	Number of contact pairs	Compares with
$m = 1$	$O(\log^2 N)$	$O(N \log^2 N)$	Batcher's sorting network [2]
$m = n$	$O(\log N)$	$O(N^2)$	Full crossbar (restricted fan-out)
$n/m = k$	$O(k \log N)$	$O(kN^{1+1/k} \log N)$	—

We will now find the delay, $D(n, m)$, and total number of edges, $E(n, m)$, for our (N, N, M)-GCN, $N = 2^n$, $M = 2^m$. The delays in a $(1, M)$-generalizer and an $(N, N/M)$-concentrator are, respectively, $\log M = m$ and $\log N = n$, as discussed earlier. The recursive structure of Figure 13, therefore, yields the following recurrence for the delay of the (N, N, M)-GCN:

$$D(n, m) = \begin{cases} m + n + D(n - m, m), & n > m, \\ 2n, & n \le m. \end{cases}$$

(Note that $D(n - m, m)$ is the delay in an $(N/M, N/M, M)$-GCN.) It is easy to show that

$$D(n, m) \le \frac{1}{2}\left(\left\lceil \frac{n}{m} \right\rceil + 3\right) n.$$

The equality holds when n is a factor of m.

As for the number of edges, we observe that each $(1, M)$-generalizer and each $(M, 1)$-concentrator is a binary tree with $2(M - 1)$ edges. An $(N/M, N/M)$-concentrator has $2(N/M)\log(N/M)$ edges. (See Figure 14.) So, from Figure 17 we see that the number of edges in an $(N, N/M)$-concentrator is $2N \log(N/M) + 2(M - 1)N/M$. Using these, and referring to Figure 13, we obtain the following recurrence for the number of edges in the (N, N, M)-GCN:

$$E(n, m) = \begin{cases} 2MN \log \dfrac{N}{M} + 4(M - 1)N + M \cdot E(n - m, m), & n > m, \\ 4(N - 1)N, & n \le m. \end{cases}$$

The exact solution may easily be found. However, the following bound will serve our purposes:

$$E(n, m) < \left\lceil \frac{n}{m} \right\rceil \left(\log \frac{N}{M} + 4\right) MN.$$

Table I gives the characteristics of our GCN for different values of the ratio m/n.

The switch settings for an (N, N, M)-GCN can be obtained using PERMUTE when only a permutation of the inputs is desired, and using SORT when the input–output mapping is not a permutation. In either case, the time needed to obtain the switch settings is $O((n/m)\log N)$ if a 2^{n+m}-PE CCC or PSC is used. The setup algorithm for one-to-one connections will be faster than the algorithm for one-to-many connections by a constant factor.

4.2 SETUP ALGORITHM FOR ONE-TO-ONE CONNECTIONS. Here, procedure PERMUTE (Algorithm 2.3) may be used to set up the GCN. We assume that the desired connection is given by $A(0{:}N - 1)$. Input i is to get connected to output $A(i)$, $0 \le i \le N - 1$; if $A(i) = null$, input i does not get connected to any output.

When CONCENTRATE(k) is invoked by PERMUTE (where $k = n - (s - 1)m$ for phase s), it corresponds to the $(2^k, 2^r)$-concentrator stage of the GCN, where $r = \max(k - m, 0)$. This stage of concentrators consists of $(2^r, 2^r)$-concentrators followed

$V(0)$	$V(1)$	$V(2)$	$V(3)$	$V(8)$	$V(9)$	$V(10)$	$V(11)$
$V(4)$	$V(5)$	$V(6)$	$V(7)$	$V(12)$	$V(13)$	$V(14)$	$V(15)$

FIG. 18. PE-to-vertex correspondence for $k = 2$.

by $(2^{k-r}, 1)$-concentrators (see Figure 17). As stated earlier, we only need to determine the switch settings for the $(2^r, 2^r)$-concentrator stage. This is done by the first r iterations of CONCENTRATE(k). The concentrator stage consists of 2^{n+m-r} $(2^r, 2^r)$-concentrators. Each concentrator has $r + 1$ columns of vertices. Therefore, the stage forms an array of $2^{n+m} \times (r + 1)$ vertices. Label these vertices $V^b(i)$, $0 \leq b \leq r$, $0 \leq i < 2^{n+m}$. Thus, V^0 and V^r, respectively, represent the input and output vertices of the $(2^r, 2^r)$-concentrators. First, consider running PERMUTE on a CCC (with 2^{n+m} PEs). During iteration b, $0 \leq b < r$, of CONCENTRATE(k), the set of PEs correspond to the column V^b of vertices. The PE-to-vertex correspondence is shown in Figure 18 for $n = 3$, $m = 1$, and $k = 2$. Note that $k = 2$ specifies 2^2-blocking. PEs 0–3 correspond to the top (4, 2)-concentrator, PEs 8–11 correspond to the second (4, 2)-concentrator, and so on.

Thus, during iteration b of CONCENTRATE(k), PE(i) corresponds to $V^b(j)$, where

$$j = \mu(i, k) = i_{n-1} \cdots i_k i_{n+m'-1} \cdots i_n i_{k-1} \cdots i_0,$$
$$0 \leq i \leq 2^{n+m'} - 1, \quad m' = \min(k, m).$$

Note that for the first stage, $j = \mu(i, n) = i$, and so PE(i) corresponds simply to $V^b(i)$. (Procedure PERMUTE may easily be modified so that the PE(i)-to-$V^b(i)$ correspondence would hold for all stages. We will not consider the modifications here.)

Iteration b of the **for** loop in CONCENTRATE determines the settings for switches from V^b to V^{b+1}. During this iteration, if $A(i) \neq$ *null* and $R(i)_b \neq i_b$ for some i, then the switch $\langle V^b(j), V^{b+1}(j^{(b)})\rangle$, $j = \mu(i, k)$, is set *on*. If $A(i) \neq$ *null* and $R(i)_b = i_b$, then $\langle V^b(j), V^{b+1}(j)\rangle$ is set *on*. If $A(i) =$ *null*, then both of these switches are set *off*. (Note that PE(i) determines the state of these switches.)

The last call to CONCENTRATE is made with $k = n - (\lceil n/m \rceil - 1)m$. (Hence $r = 0$ at this point.) This corresponds to the $(2^k, 1)$-concentrators (binary trees) of the last stage. Let $V^0(j)$, $0 \leq j \leq 2^{n+k} - 1$, be the leftmost column of vertices (i.e., the leaves) in this stage. We only need to set up the switches incident from these vertices. (Recall from Section 4.1 that the edges here are not all *on* as they are in the binary trees of other stages.) Let $j = \mu(i, k)$ as defined above. The single edge (switch) incident from vertex $V^0(j)$ is set *on* if and only if $A(i) \neq$ *null* at the start of iteration $b = 0$ of CONCENTRATE. Our setup algorithm terminates at this point, as the remaining $k - 1$ levels of edges are all *on*.

If a PSC is used to run PERMUTE, the PE-to-vertex correspondence will be different: During iteration b of CONCENTRATE, PE(i) will correspond to $V^b(j)$, where $j = \mu(i', k)$, $i' = i_{q-1-b} \cdots i_0 i_{q-1} \cdots i_{q-b}$, $q = n + m$, and μ is as defined above.

4.3 SETUP ALGORITHM FOR ONE-TO-MANY CONNECTIONS. For one-to-many mappings the switch settings are determined *right-to-left* (starting with the output terminals and working toward the inputs). This is done using a modified version of procedure SORT (Algorithm 3.1). Before discussing the needed modifications, we informally discuss how the switch settings may be determined. We assume that the desired input–output mapping is given by $A(0{:}N - 1)$, where $A(i) =$ *null* iff output

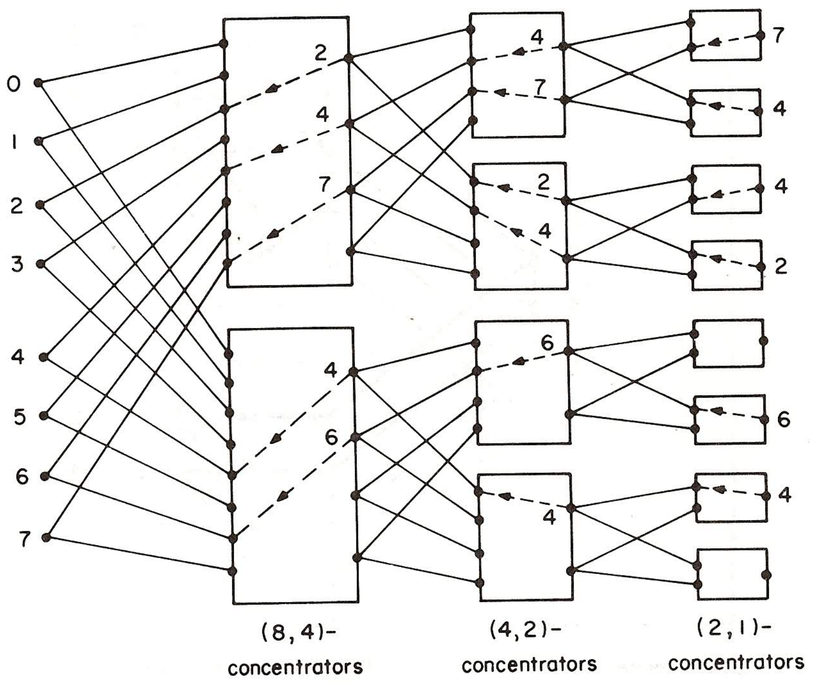


FIG. 19. An (8, 8, 2)-GCN. (Setting up by sorting output-request vector.)

i is not connected to any input. If $A(i) \neq$ *null*, then output i is to be connected to input $A(i)$. A may be thought of as an *output request vector* for the GCN.

Consider the (8, 8, 2)-GCN of Figure 19. The output request vector $A(0{:}7)$ is (7, 4, 4, 2, *null*, 6, 4, *null*). The (8, 8, 2)-GCN consists of a stage of (1, 2)-generalizers followed by a stage of (8, 4)-concentrators, followed by (1, 2)-generalizers, (4, 2)-concentrators, (1, 2)-generalizers and (2, 1)-concentrators. The output request vector for each (2, 1)-concentrator is obtained directly from $A(0{:}7)$. The output request vector for a (4, 2)-concentrator may be obtained by merging together the output request vectors for the two (2, 1)-concentrators it is connected to via (1, 2)-generalizers. So, the output request vectors for the top two (4, 2)-concentrators of Figure 19 are (4, 7) and (2, 4), respectively. In general, the output request vector for any $(p, p/M)$-concentrator is obtained by merging together the output request vectors of the M $(p/M, p/M^2)$-concentrators it is connected to via $(1, M)$-generalizers. *During this merge, equal request values must be replaced by a single value.* Thus the output request vector for the top (8, 4)-concentrator of Figure 19 is (2, 4, 7, *null*).

The switch settings for any $(p/M, p/M^2)$-concentrator may be determined by making use of its output request vector and the output request vector of the $(p, p/M)$-concentrator that connects to it (via $(1, M)$-generalizers). The setup process for the top two (4, 2)-concentrators of Figure 19 is illustrated in Figure 20. In this figure, R is the *output-rank vector* of each concentrator: Output i of a concentrator must get connected to input $R(i)$ of that concentrator. (If $R(i) =$ *null*, the output does not get connected to any input.) The output request vectors for the two (4, 2)-concentrators are (4, 7) and (2, 4). These request vectors must "merge" (thinking of the requests as moving right to left) to give the request vector (2, 4, 7, *null*) at the output of the (8, 4)-concentrator. Therefore, the output-rank vectors for the two (4, 2)-concentrators are, respectively, (1, 2) and (0, 1). Once the output rank vector of a concentrator is obtained, the switch settings for that concentrator are determined

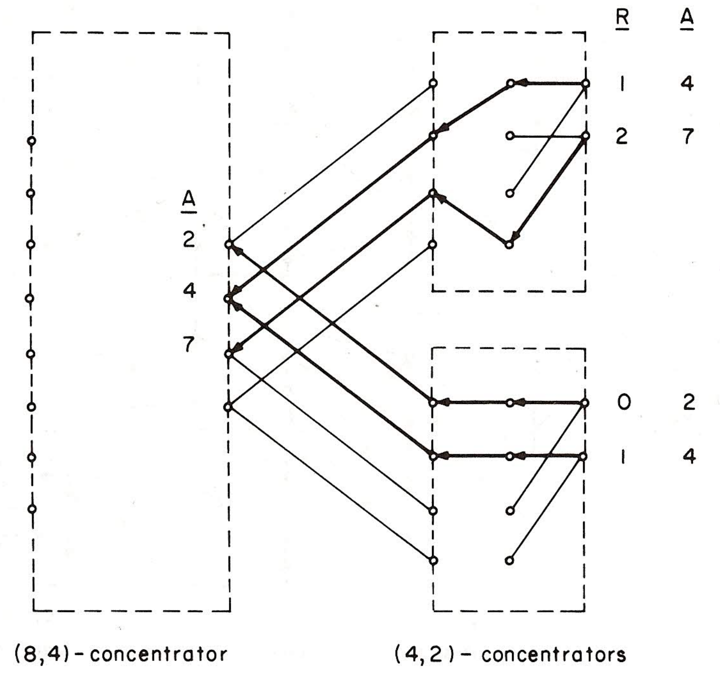


FIG. 20. Setup for the top two (4, 2)-concentrators of Figure 19.

right to left. (Each request is routed right to left using the bits of its rank, the most significant bit first. See Figure 20.)

For the leftmost concentrator stage (i.e., for the $(N, N/M)$-concentrators), no rank computation is needed. The output-request vector of a concentrator here serves as its output-rank vector. For example, in Figure 19 the request value for the top output of the top (8, 4)-concentrator is 2; so this output simply gets connected to input 2 of the concentrator.

Our switch setting algorithm is obtained by making some simple changes to procedures SORT and COUNT. Let the stages of concentrators in GCN be numbered 1 to $\lceil n/m \rceil$ left to right. Phase s of SORT, $s = 1, 2, \ldots, \lceil n/m \rceil$, will start with the output request vectors for the concentrators in stage $s' = \lceil n/m \rceil - s + 1$; it will find the switch settings for this stage (right to left); and it will end up with the output request vectors for stage $s' - 1$.

First, we make the following changes to COUNT:

(1) Insert the following code before line 1:

$$S(i) := \infty, (S(i) = null))$$
$$T(i) := \infty, (T(i) = null))$$

(2) Replace lines 5 and 6 by

$$T'(i) := 2^r$$

The effect of the above changes is that for each element of S, COUNT will now find the number of elements in T that are *strictly less* than it. Consequently, $R(i)$ following lines 14–16 of SORT will be the number of elements in the 2^k-block strictly less than $A(i)$. (2^k is the size of sorted sequences to be obtained in the current phase of SORT.) Note that after lines 14–16 of SORT, equal A values will have equal rank.

```
16.1   $W(i) := i$  //return-address of this PE//
16.2   $R(i) := W(i) := null$, $(A(i) = null)$  //take diagonal $2^r$-blocks//
16.3   for $b := k - 1$ downto 0 do  //Route to appropriate columns//
16.4       $\langle R(i^{(b)}), W(i^{(b)})\rangle \leftarrow \langle R(i), W(i)\rangle$, $(R(i) \neq null$ and $R(i)_b \neq i_b)$
16.5   end
16.6   $S(i) := 0$, $(R(i) = null)$
16.7   $S(i) := 1$, $(R(i) \neq null)$
16.8   for $b := n$ to $n + m - 1$ do  //Mark nonnull columns//
16.9       $S(i) \leftarrow (S(i)$ or $S(i^{(b)}))$  //Boolean or//
16.10  end
16.11  call RANK'$(k)$  //set $R(i)$ = number of nonnull columns to the left//
16.12  for $b := 0$ to $k - 1$ do  //return the computed rank//
16.13      $\langle R(i^{(b)}), W(i^{(b)})\rangle \leftarrow \langle R(i), W(i)\rangle$, $(W(i) \neq null$ and $W(i)_b \neq i_b)$
16.14  end
```

FIG. 21. Subalgorithm 4.1: Rank adjustment after line 16 of SORT.

We now need to adjust the rank, $R(i)$, to be the number of *distinct* A values in the 2^k-block strictly less than $A(i)$. This rank adjustment is accomplished by lines 16.1–16.14 (Subalgorithm 4.1; see Figure 21). The code segment is inserted after line 16 of SORT.

The loop of lines 16.3–16.5 routes each nonnull rank element $R(i)$, tagged with its originating address $W(i) = i$, to the $R(i)$th column within the 2^k-block. (Note that the routing of lines 16.3–16.5 is the same as that performed by lines 17–19 of SORT. Each diagonal 2^r-block, where r is defined in SORT, is "spread out" within its 2^k-block.) After this routing, equal rank elements will be in the same column. The number, j, of nonnull columns to the left of an element $R(i)$ will now be the number of *distinct* R values smaller than $R(i)$. Lines 16.6–16.11 find this j for each element $R(i)$. First, lines 16.6–16.10 set $S = 1$ for all PEs in a column if that column contains any (nonnull) R values; otherwise $S = 0$ for all PEs in that column. Line 16.11 ranks the $S = 1$ values in each 2^k-block. The result is $R(i) = j$ where j is as defined above. RANK′ of line 16.11 is procedure RANK of Section 2 with the two lines initializing S omitted. Finally, lines 16.12–16.14 route back the updated value of each $R(i)$ to its originating PE, $W(i)$. (The path followed in lines 16.3–16.5 is traversed backward by the routing of lines 16.12–16.14.)

As stated earlier, the leftmost stage of concentrators in the GCN (corresponding to the last phase of SORT) does not require any rank computation; the output request vector A in this stage serves as the output rank vector. This situation is handled by inserting the following code after line 2 of SORT:

```
2.1 if $s = \lceil n/m \rceil$ then  //this is the last phase//
2.2     {$R(i) := A(i)$
2.3     go to line 17}
```

When line 17 of SORT is reached (either from line 2.3 or after completion of the loop of lines 16.12–16.14), each diagonal 2^r-block will have a sorted sequence A (with distinct values) and its rank sequence R. These will be, respectively, the output request vector and output rank vector of a $(2^k, 2^r)$-concentrator. The loop of lines 17–19 of SORT spreads each 2^r-sequence A within its containing 2^k-block. This corresponds to moving backward (right to left) in the $(2^k, 2^r)$-concentrator stage. Let $r' = \max(r, 1)$. Recall that only the left r' columns of switches in this stage need to be set up. (The right $k - r'$ columns of edges are always *on.*) Iteration b of the loop of lines 17–19 of SORT, $b = r' - 1, \ldots, 0$, is used to set up the switches incident from the vertex column V^b. (The input vertices are V^0.) This is done in a manner similar to that described in Section 4.2. The PE-to-vertex correspondence is also as defined there.

4.4 STRUCTURE OF GCN'S CONTROL MEMORY. GCN's control memory consists of one bit for each *controlled edge* of the GCN. (The always-on edges are not controlled.) We will now show how a CCC or PSC may transfer the computed settings to this memory without requiring random-access capability.

Let a *controlled vertex* be one with some controlled edges incident from it. The controlled vertices form a $2^{n+m} \times c$ array for some c. Label these vertices as $V^b(i)$, $0 \le i < 2^{n+m}$, $0 \le b < c$. Each vertex in the last column has one outgoing edge. The remaining vertices each have two outgoing edges. The *control memory* will be organized as a $2^{n+m} \times c$ array of 2-bit cells $M^b(i)$, $0 \le i < 2^{n+m}$, $0 \le b < c$. Each cell is associated with one vertex; each of the two bits will control one outgoing edge. (The vertices in the last column each use only one bit.) For each column V^b, we have established a correspondence between PE(i) and $V^b(j)$, where i and $j \in [0, 2^{n+m} - 1]$. Accordingly, the cell $M^b(i)$ is associated with the vertex $V^b(j)$. (The choice of a CCC or PSC *fixes* this association.)

The CCC (or PSC) computes the settings for one column at a time. The columns are computed left to right for one-to-one connections and right to left for one-to-many connections. In either case, PE(i) computes the value for $M^b(i)$, $0 \le b < c$. First, consider the setup algorithm for one-to-many connections. Each PE(i) is connected to $M^0(i)$. And each cell $M^b(i)$ is connected to $M^{b+1}(i)$, $0 \le b \le c - 2$, $0 \le i < 2^{n+m}$. As each column is computed, it is delivered to the leftmost column of the control memory, while the content of each cell is transferred to its right cell. This algorithm, of course, can also handle one-to-one mappings, hence requiring no additional hardware. The earlier algorithm, however, can obtain a faster setup. Since that algorithm computes the columns left to right, each computed column now should be delivered to the rightmost column of the control memory while shifting the memory contents to the left.

Finally, the setup algorithms may be easily modified to enable us to associate each cell $M^b(i)$ simply to $V^b(i)$, if so desired.

5. *Conclusions*

We have developed a permutation algorithm for CCCs and PSCs. This algorithm permutes $N = 2^n$ records in $O(k \log N)$ time when 2^{n+m} PEs are available, $k = n/m$. PERMUTE is based on MSD radix-sorting. We showed that an LSD version of this algorithm can *sort* N integers in the range $[0, 2^q - 1]$ in $O((q/m) \log N)$ time.

We presented a general sorting algorithm in Section 3. This algorithm has the same asymptotic complexity as PERMUTE. However, it is slower than PERMUTE by a constant factor. The sorting algorithm also has the same asymptotic complexity as that developed by Preparata [18] for the shared memory model.

Using the sorting algorithm developed here, the data broadcasting algorithm of [17] can also be run in $O(k \log N)$ time. This algorithm performs arbitrary one-to-many transfers among N PEs.

Finally, while the GCN construction obtained here is inferior to that of [22] in terms of delay and number of contact pairs, it can be set up far more efficiently than that of [22]. When switch settings have to be determined in real time, our GCN will outperform that of [22].

REFERENCES

1. ARJOMANDI, E. A study of parallelism in graph theory. Ph.D. Dissertation, Computer Science Dep., Univ. of Toronto, Toronto, Ontario, Dec. 1975.
2. BATCHER, K.E. Sorting networks and their application. *Proc. AFIPS 1968 SJCC*, Vol. 32, AFIPS Press, Arlington, Va., pp. 307–314.
3. BRENT, R.P. The parallel evaluation of general arithmetic expression. *J. ACM 21,* 2 (Apr. 1974), 201–206.
4. CSANKY, L. Fast parallel matrix inversion algorithms. *SIAM J. Comput. 5,* 4 (Dec. 1976), 618–623.
5. ECKSTEIN, D. Parallel graph processing using depth-first search and breadth-first search. Ph.D. Dissertation, Univ. of Iowa, Ames, Iowa, 1977.
6. FLYNN, M.J. Very high speed computing systems. *Proc. IEEE 54* (Dec. 1966), 1901–1909.
7. HIRSCHBERG, D.S. Parallel algorithms for the transitive closure and the connected component problems. Proc. 8th ACM Symp. on Theory of Computing, Hershey, Pa., May 1976, pp. 55–57.
8. HIRSCHBERG, D.S. Fast parallel sorting algorithms. *Commun. ACM 21,* 8 (Aug. 1978), 657–661.
9. KNUTH, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
10. LANG, T. Interconnections between processors and memory modules using the shuffle–exchange network. *IEEE Trans. Comput. C-25,* 5 (May 1976), 496–503.
11. LANG, T., AND STONE, H. A shuffle–exchange network with simplified control. *IEEE Trans. Comput. C-25,* 1 (Jan. 1976), 55–65.
12. MULLER, D.E., AND PREPARATA, F.P. Bounds to complexities of networks for sorting and for switching. *J. ACM 22,* 2 (Apr. 1975), 195–201.
13. MUNRO, I., AND PATERSON, M. Optimal algorithms for parallel polynomial evaluation. *J. Comput. Syst. Sci. 7* (1973), 189–198.
14. NASSIMI, D., AND SAHNI, S. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Comput. C-28,* 1 (Jan. 1979), 2–7.
15. NASSIMI, D., AND SAHNI, S. An optimal routing algorithm for mesh-connected parallel computers. *J. ACM 27,* 1 (Jan. 1980), 6–29.
16. NASSIMI, D., AND SAHNI, S. Parallel algorithms to set up the Benes permutation network. *IEEE Trans. Comput. C-31,* 2 (Feb. 1982), 148–154.
17. NASSIMI, D., AND SAHNI, S. Data broadcasting in SIMD computers. *IEEE Trans. Comput. C-30,* 2 (Feb. 1981), 101–107.
18. PREPARATA, F.P. New parallel-sorting schemes. *IEEE Trans. Comput. C-27,* 7 (July 1978), 669–673.
19. SAVAGE, C. Parallel algorithms for graph theoretic problems. Ph.D. Dissertation, Univ. of Illinois, Urbana, Ill., Aug. 1978.
20. STONE, H. Parallel processing with the perfect shuffle. *IEEE Trans. Comput. C-20,* 2 (1971), 153–161.
21. SWANSON, R.C. Interconnections for parallel memories to unscramble *p*-ordered vectors. *IEEE Trans. Comput. C-23,* 11 (1974), 1105–1115.
22. THOMPSON, C.D. Generalized connection networks for parallel processor intercommunication. *IEEE Trans. Comput. C-27,* 12 (Dec. 1978), 1119–1125.
23. THOMPSON, C.D., AND KUNG, H.T. Sorting on a mesh-connected parallel computer. *Commun. ACM 20,* 4 (Apr. 1977), 263–271.

RECEIVED APRIL 1979; REVISED DECEMBER 1980; ACCEPTED MARCH 1981