Parallel Algorithms to Set Up the Benes Permutation Network

DAVID NASSIMI AND SARTAJ SAHNI, MEMBER, IEEE

***Abstract*—A parallel algorithm to determine the switch settings for a Benes permutation network is developed. This algorithm can determine the switch settings for an N input/output Benes network in $0(\log^2 N)$ time when a fully interconnected parallel computer with N processing elements is used. The algorithm runs in $0(N^{1/2})$ time on an $N^{1/2} \times N^{1/2}$ mesh-connected computer and $0(\log^4 N)$ time on both a cube connected and a perfect shuffle computer with N processing elements. It runs in $0(k \log^3 N)$ time on cube connected and perfect shuffle computers with $N^{1+1/k}$ processing elements.**

***Index Terms*—Benes permutation network, complexity, cube connected computer, fully connected SIMD computer, mesh-connected computer, parallel algorithm, perfect shuffle computer, set-up algorithm.**

I. Introduction

The Benes permutation network $B(n)$ is a network with $N = 2^n$ inputs and outputs. The network is capable of delivering at its output end any permutation of its N inputs. This network has been proposed for use in telephone networks, self-repairing multiprocessors [10], as an interconnection network in parallel computers [12], etc. It forms the heart of a common generalized connection network [7].

Fig. 1 gives a schematic of $B(n)$ and Fig. 2 gives the two possible states of a switch. Observe that there are $N/2$ switches at the input stage and only $N/2 - 1$ switches at the output stage. So the network of Fig. 1 incorporates the switch saving scheme suggested by Waksman [9]. From Fig. 1 it follows that $B(n)$ has $2n - 1$ switch stages and $N \log N - N + 1$ switches (note that $B(1)$ is just a single switch). Let the switch stages be numbered 0 through $2n - 2$ left to right (see Fig. 1).

Waksman [9] has shown that the network $B(n)$ is capable of delivering at its output end any permutation of its N inputs. His proof of this fact is constructive and it directly leads to a switch setting algorithm. This algorithm runs in $0(N \log N)$ time on a single processor computer. Thus, the set-up time for the network is much larger than the network delay [which is $0(\log N)$]. One cannot set up the Benes network in less than $0(N \log N)$ time using a single processor as the network has $0(N \log N)$ switches. In order to obtain a set-up algorithm of complexity comparable to the delay time, it is therefore necessary to consider parallel algorithms. An alternative is to make the network self-setting, as has been done by Nassimi and Sahni [5]. Their self-routing scheme, however, does not

Manuscript received March 23, 1981; revised August 4, 1981. This work was supported in part by the National Science Foundation under Grant 78-15455.

D. Nassimi is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201.

S. Sahni is with the Department of Computer Science, University of Minnesota, Minneapolis, MN 55455.

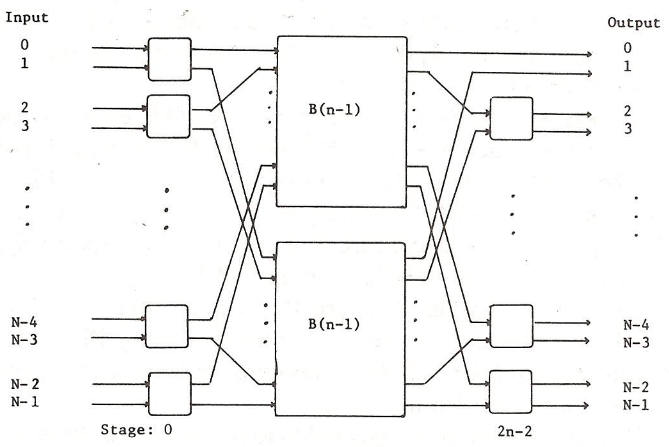


Fig. 1. The Benes permutation network $B(n)$, $N = 2^n$.

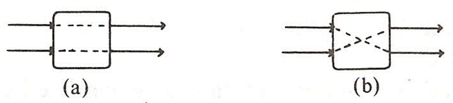


Fig. 2. States of a binary switch. (a) State 0. (b) State 1.

work for all permutations. Another alternative is to precompute and store the switch settings for some permutations. This requires $0(N \log N)$ bits of storage per permutation. Furthermore, the precomputation approach is not suitable for dynamic situations.

In this paper we develop a parallel set-up algorithm that is significantly faster than the single processor algorithm of Waksman. The complexity of our algorithm depends on the parallel computer model and the number of processing elements available. We consider four computer models. All four are SIMD (single instruction stream, multiple data stream) type computers (see Flynn [13]). An SIMD computer consists of some number M of processing elements (PE's), each having some local memory. The PE's are indexed 0 through $M - 1$. We shall refer to the ith PE as PE(i). The PE's are synchronized and operate under the control of a single instruction stream. An enable/disable mask may be used to select a subset of PE's that will perform the instruction to be executed at any given time. All enabled PE's perform the same instruction. The four SIMD models we shall consider differ in the way the PE's are interconnected. The PE interconnection pattern is important as PE's can communicate only through the interconnection network. The four PE interconnection networks defining our four SIMD models are as follows.

1) *Completely Interconnected Computer* (*CIC*): In a CIC model every pair of PE's is directly connected. The time needed to transfer data from one PE to another is 0(1).

2) *Mesh-Connected Computer* (*MCC*): In this model, the PE's may be thought of as logically arranged as in a k-dimensional array $A(n_{k-1}, n_{k-2}, \cdots, n_0)$, where n_i is the size of the ith dimension and $M = n_{k-1}n_{k-2} \cdots n_0$. The PE at location $A(i_{k-1}, \cdots, i_0)$ is connected to the PE's at location $A(i_{k-1}, \cdots, i_j \pm 1, \cdots, i_0)$, $0 \leq j < k$, provided they exist.

3) *Cube Connected Computers* (*CCC*): Assume that $M = 2^q$ and let $i_{q-1} \cdots i_0$ be the binary representation of i for $i \in [0, M-1]$. 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($i^{(b)}$), $0 \leq b < q$. Siegel [14] discusses this interconnection scheme in some detail.

4) *Perfect Shuffle Computer* (*PSC*): This employs the shuffle connection of stone [15]. Specifically, let M, 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, respectively, be called *exchange*, *shuffle*, and *unshuffle*.

One may verify that there are $M - 1$ connections per PE in a CIC, $2k$ connections per PE (except boundary PE's) in a k-dimensional MCC, log M connections per PE in a CCC, and 3 connections per PE in a PSC.

The complexity of our algorithm to set up $B(n)$ is $0(\log^2 N)$ on an N PE CIC, $0(\log^4 N)$ on an N PE CCC and on an N PE PSC, $0(k \log^3 N)$ on an $N^{1+1/k}$ PE CCC and $N^{1+1/k}$ PE PSC, $1 \leq k \leq \log N$, and $0(N^{1/2})$ on an $N^{1/2} \times N^{1/2}$ two-dimensional MCC.

It should be pointed out that a generalized-connection-network (GCN, a network capable of performing any one-to-many mapping of its inputs onto its outputs) of the type described in [7] can also be set up in the asymptotic times just given for a Benes network. This is so, as the GCN of [7] consists of a hyperconcentrator, infrageneralizer, and a Benes permutation network. The first two of these networks are relatively easy to set up (see [3], [4], or [7]).

In Section II we develop our $0(\log^2 N)$ CIC algorithm. Section III analyzes a minor modification of this algorithm for CCC's, PSC's, and MCC's. Finally, we should point out that a short version of this paper was presented at the Workshop on Interconnection Networks held at Purdue University, April 1980. Lev *et al.* [16] consider the set-up problem in a more general framework, but do not consider implementing their algorithms on parallel computers with a fixed interconnection network. The work reported here was done independently of [16].

II. Set-Up Algorithm for a CIC

As remarked earlier, the constructive proof provided by Waksman (showing that $B(n)$ can realize all $N!$ permutations of its N inputs) leads to an $0(N \log N)$ single-processor set-up algorithm.[1] It is instructive to review this proof as it also leads to our parallel algorithm. Let $D = D(0{:}N-1) = (D(0), D(1), \cdots, D(N-1))$ be the desired permutation. Input i is to be routed by output $D(i)$, $0 \leq i < N$. Let the upper $B(n-1)$ network of Fig. 1 be denoted by $B_u(n-1)$ and the lower by $B_l(n-1)$. We shall obtain the switch settings for the switches in stages 0 and $2n-2$, and also the permutations D_u and D_l to be performed by $B_u(n-1)$ and $B_l(n-1)$. The switch settings for the first and last stage and the permutations D_u and D_l are such that if $B_u(n-1)$ realizes D_u and $B_l(n-1)$ realizes D_l, then $B(n)$ realizes D. Since $B(1)$ can realize all permutations of 2 inputs, it will follow by induction that $B_u(n-1)$ can be set up to realize D_u and $B_l(n-1)$ can be set up to realize D_l.

We shall assume that the switches in each stage are indexed 0 through $N/2 - 1$ top to bottom. This requires us to assign indexes to nonexistent switches (for example, switch 0 of stage $2n-2$ refers to the switch with outputs 0 and 1 which has been removed from Fig. 1). Let $S(i, j)$ denote switch i of stage j, $0 \leq i < N/2, 0 \leq j \leq 2n-2$. We shall have $S(i, j) = 0$ if switch i of stage j is to be in state 0, and $S(i, j) = 1$ if the switch is to be in state 1. In specifying the permutations D_u and D_l, we shall assume that the inputs and outputs of $B_u(n-1)$ are numbered 0 to $N/2 - 1$, whereas those of $B_l(n-1)$ are numbered $N/2$ to $N-1$. Recall from Fig. 1 that input i of $B(n)$ is connected through switch $S(i/2, 0)$ to either input $i/2$ of B_u or input $N/2 + i/2$ of B_l. (All divisions in this paper are integer divisions. So $i/2$ means $\lfloor i/2 \rfloor$). Similarly, output i of $B(n)$ must come from either output $i/2$ of B_u or output $N/2 + i/2$ of B_l, through switch $S(i/2, 2n-2)$. Let E be the inverse of D, i.e., $E(D(i)) = i$, $0 \leq i < N$.

Corresponding to the input–output mapping (permutation) $D(0{:}N-1)$, we may define an undirected bipartite multigraph $G(D)$ in the following way. The graph has vertices $x_0, x_1, \cdots, x_{N/2-1}$ and $y_0, y_1, \cdots, y_{N/2-1}$. Vertex x_i corresponds to switch i of the first stage (i.e., stage 0); vertex y_i corresponds to switch i in the last stage (i.e., stage $2n-2$). If $D(i) = j$ in the mapping, then the multigraph contains an undirected edge $(x_{i/2}, y_{j/2})$. Fig. 3(a) shows an input–output mapping $D(0{:}7) = (3, 2, 5, 0, 4, 6, 7, 1)$ for the network $B(3)$. Fig. 3(b) gives the corresponding bipartite multigraph.

Waksman's construction in effect follows the cycles in the bipartite multigraph $G(D)$ in order to determine the state of switches in the first and last stage of $B(n)$, and obtain the permutations D_u and D_l. We shall describe the process using the example permutation of Fig 3(a). From the permutation $D(0{:}7)$, we are to find the switch settings for stages 0 and 4, and obtain $D_u(0{:}3)$ and $D_l(4{:}7)$.

Beginning with switch $S(0, 4)$, we must have $S(0, 4) = 0$ as the switch is nonexistent (see Fig. 4). Node y_0 corresponds to this switch; the cycle to follow is $(y_0, x_1, y_2, x_2, y_3, x_3, y_0)$. (The bipartite multigraph is for exposition purposes; the algorithm only uses the permutation D and its inverse E.) Since $S(0, 4) = 0$, output 0 must come from B_u. So input $E(0) = 3$ must be routed to B_u. Thus, $S(1, 0) = 1$ and $D_u(1) = 0$. The other input to switch $S(1, 0)$ is 2. This input will go to B_l. To route it through B_l, we must have $D_l(4 + 2/2) = 4 + D(2)/2$. Hence, $D_l(5) = 6$. To get input 2 to its final destination, we must have $S(2, 4) = 0$. So far, we have routed input 3 to output 0 through B_u, and input 2 to output 5 through B_l. In the bipartite multigraph this corresponds to edges (x_1, y_0) and (x_1, y_2) being routed through B_u and B_l, respectively. The part of the cycle we have traversed is y_0, x_1, y_2.

Having determined $S(2, 4) = 0$, we conclude that output 4 must come from B_u. Starting with output 4, we now repeat the above process. That is, input $E(4) = 4$ is routed to output

[1] Opferman and Wu independently discovered Waksman's construction and described it at length in [11].

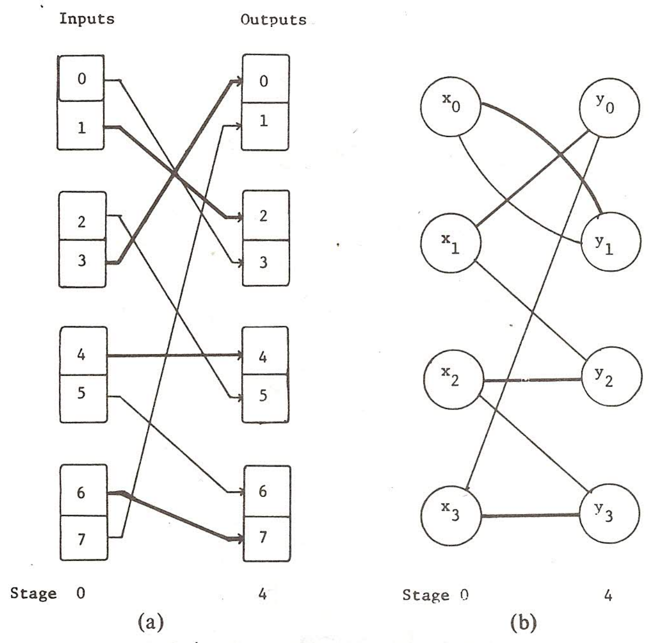


Fig. 3. An example permutation for network $B(3)$. (a) Input–output mapping. (b) Bipartite multigraph.

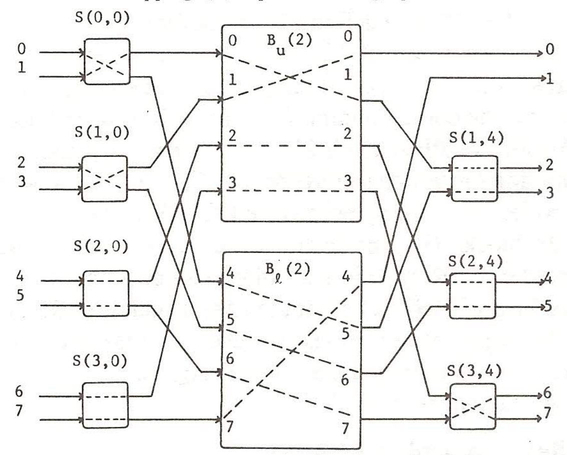


Fig. 4. The switch settings of the first and last stages corresponding to Fig. 3.

4 through B_u by setting $S(2, 0) = 0$ and $D_u(2) = 2$. Consequently, input 5 must connect to output 6 through B_l, giving $D_l(6) = 7$ and $S(3, 4) = 1$. The state of switch $S(3, 4)$ implies that output 7 must come from B_u.

Finally, starting with output 7, we must route input $E(7) = 6$ through B_u. This gives $S(3, 0) = 0$ and $D_u(3) = 3$. So input 7 goes through B_l. This requires $D_l(7) = 4$ and $S(0, 4) = 0$. But, the state of $S(0, 4)$ has already been determined. This means that we have completed a cycle of the bipartite graph.

The other cycle in the bipartite graph of Fig. 3(b) is (y_1, x_0, y_1). Starting with y_1, we arbitrarily set $S(1, 4) = 0$. Now this cycle may be processed in the same way. It gives $S(0, 0) = 1$, $D_u(0) = 1$, and $D_l(4) = 5$. Having processed all cycles of $G(D)$, all switch settings for stages 0 and 4 have been determined. The permutations to be realized by $B_u(2)$ and $B_l(2)$ are $D_u = (1, 0, 2, 3)$ and $D_l = (5, 6, 7, 4)$. (See Fig. 4.)

Let J_u be the set of outputs that receive their signals from $B_u(n - 1)$. It is clear that the following are true for any J_u determined by Waksman's algorithm:

1) $0 \in J_u$,

2) $|J_u| = N/2$ ($|J_u|$ is the number of elements in J_u),

3) if i and $j \in J_u$ and $i \neq j$, then $i \neq j^{(0)}$ and $E(i) \neq (E(j))^{(0)}$.

(Recall that $i^{(0)}$ denotes the integer differing from i only in bit 0.)

Any set J_u satisfying properties 1), 2), and 3) above will be called a *matching*. The dark lines of Fig. 3 correspond to edges incident to elements in J_u. In Fig. 3(b) observe that the dark lines match each input switch x_i with a distinct output switch y_j. That is, the set J_u defines a *complete matching* on the bipartite graph $G(D)$.

Note that to determine the switch settings for stages 0 and $2n - 2$ and to determine D_l and D_u, it is adequate to find any set J_u satisfying properties 1), 2), and 3). If $i \in J_u$, then (all divisions are integer divisions):

R1: $S(i/2, 2n - 2) = i_0$

R2: $S(E(i)/2, 0) = (E(i))_0$

R3: $D_u(E(i)/2) = i/2$

R4: $D_l(N/2 + E(i)/2) = N/2 + D\{(E(i)^{(0)})/2$.

These four rules are the same as those given in [11].

We now describe our parallel algorithm for finding a matching J_u. First, we partition the output set $J = \{0, 1, \cdots, N - 1\}$ into equivalence classes such that i and j are in the same equivalence class iff either both i and j must get their signals from $B_u(n - 1)$ or both must get them from $B_l(n - 1)$. Equivalently, i and j are in the same class iff either both must be in J_u or both must be in $J - J_u$. Note that such an equivalence class satisfies property 3) of a matching. For the example of Fig. 3, we see that since $0 \in J_u$, 4 and 7 must also be in J_u. Also, since $1 \notin J_u$, 6 and 5 must be in $J - J_u$. If $2 \in J_u$, then $3 \notin J_u$, and if $3 \in J_u$ then $2 \notin J_u$. So the equivalence classes are $\{0, 4, 7\}$, $\{1, 6, 5\}$, $\{2\}$, and $\{3\}$.

Once the equivalence classes have been determined, we can determine the set J_u. We make the following observations about the equivalence classes defined above.

1) Let Q be an equivalence class. If $i \in Q$, then $i^{(0)} \notin Q$.

2) If $i \in Q_1$ and $i^{(0)} \in Q_2$, then $Q_1 \cup Q_2$ defines a cycle of the bipartite graph $G(D)$. For every $j \in Q_1, j^{(0)} \in Q_2$. For every $j \in Q_2, j^{(0)} \in Q_1$ and $|Q_1| = |Q_2|$.

J_u may be obtained as follows. Let Q_0 and Q_1 be the equivalence classes containing 0 and 1, respectively. $Q_0 \subseteq J_u$ and $Q_1 \subseteq J - J_u$. Note that $0 = \min\{i | i \in Q_0\}$ and $1 = \min\{i | k \in Q_1\}$. If $Q_0 \cup Q_1 \neq J$, let j be the minimum index not in $Q_0 \cup Q_1$. From observation 2) it follows that $j^{(0)} \notin Q_0 \cup Q_1$. Let Q_2 and Q_3 be the equivalence classes containing j and $j^{(0)}$, respectively. $Q_2 \cup Q_3$ defines another cycle of $G(D)$. We may put either Q_2 or Q_3 into J_u. For definiteness we shall put Q_2 into J_u. Note that $j = \min\{i | i \in Q_2\}$ and $j^{(0)} = \min\{i | i \in Q_3\}$. By repeating this process, J_u can be obtained. It is apparent that the resulting J_u satisfies properties 1), 2), and 3) of a matching. One should also note that from the preceding discussion it follows that if $\min\{i | i \in Q\}$ is even, then the elements of Q are in J_u, otherwise they are in $J - J_u$. For the example of Fig. 3, we obtain $J_u = \{0, 2, 4, 7\}$.

In summary, to obtain J_u we need to first partition J into equivalence classes. Next, each element of J must find the minimum element in the equivalence class it is in. If the minimum element in the class containing output i is even, then $i \in J_u$. Otherwise, $i \notin J_u$. From J_u, the switch settings for stages 0 and $2n - 2$ together with the permutations D_u and D_l can be obtained. The remaining switch settings can be obtained by repeating this process on D_u and D_l. Our algorithm will obtain the switch settings for B_u and B_l in parallel.

First, let us see how the equivalence classes may be determined. The algorithm we are about to describe will represent each equivalence class as a cycle. Nodes in a cycle will be linked together using a field R. $R(i)$ denotes the link field in PE(i). Initially, the permutation D is distributed over the N PE's in the CIC such that $D(i)$ is in PE(i). Let j be any output terminal. j and $r = D((E(j))^{(0)})$ must be in different equivalence classes, and j and $r^{(0)}$ must be in the same equivalence class. As an example, consider output 7 of Fig. 3. $E(7) = 6$, $(E(7))^{(0)} = 7$, and $D(7) = 1$. Outputs 1 and 7 cannot get their inputs from the same $B(n - 1)$ network. So outputs 0 and 7 (and also 1 and 6) must get their signals from the same network and so must be in the same equivalence class. Procedure EQUIV sets $R(j) = r^{(0)}$ for every j. $r^{(0)}$ is as defined above. Note that since D and E are permutations, $r^{(0)}$ is different for different j's. This ensures that the resulting R values will define a cycle structure.

```
line   procedure EQUIV
          //determine equivalence classes//
          global arrays $R, D, E$
1         $R(i) := (D(i))^{(0)}$
2         $R(i^{(0)}) \leftarrow R(i)$
3         $\langle E(D(i)), R(D(i))\rangle \leftarrow \langle i, R(i)\rangle$
       end EQUIV
```

Algorithm 1

In specifying our algorithms, we have made use of two kinds of assignment statements. Those using the operator ":=" denote assignments local to a PE or to the control unit. Those using the operator "$\leftarrow$" denote assignments requiring some routing through the PE interconnection network. For any integer i, i_b and $i^{(b)}$ are as defined earlier, and $(i)_{r:t}$ denotes the integer with binary representation $i_r i_{r-1} \cdots i_t$.

Since the inverse permutation E is not known, procedure EQUIV computes R in a slightly different way than described above. For any input i, we know that $D(i)$ and $(D(i^{(0)}))^{(0)}$ must be in the same class. So we set $R(D(i))$ to $(D(i^{(0)}))^{(0)}$ in line 3. (Observe that for $j = D(i)$, $r^{(0)} = (D(i^{(0)}))^{(0)}$.) In line 3 we also compute the inverse permutation E. This is needed, later, to obtain the switch settings and D_l and D_u. Fig. 5 shows the cycles resulting from an application of procedure EQUIV to the example of Fig. 3. The time complexity of EQUIV is $O(1)$.

Having seen how to obtain the equivalence classes in the form of cycles, we proceed to the algorithm that finds the minimum element in each class. In order to describe this algorithm, we need to introduce the concept of a 2^k-block of PE's. A 2^k-block of PE's consists of 2^k PE's whose indices differ only in the least significant k bits. Thus, PE($i * 2^k$) $\cdots$, PE($i * 2^k + 2^k - 1$) define a 2^k-block for each i, $0 \le i < 2^{n-k}$ (recall that $n = \log N$). The concept of a 2^k-block is important because when we are computing the switch settings for a $B(k)$ network, the corresponding D vector will be distributed over a 2^k-block of PE's. Each 2^k-block of PE's will therefore be computing the switch settings for a different $B(k)$ network and the cycles obtained by procedure EQUIV will be localized to the 2^k-blocks (i.e., no cycle can cross a block boundary). Moreover, no R cycle in a 2^k-block can be longer than 2^{k-1}. Procedure LEAST(k) computes $M(i)$ such that $M(i)$ is the smallest element in the class (or cycle) containing i, $0 \le i < N$. It assumes that cycles are localized to 2^k-blocks and that no cycle is longer than 2^{k-1}.

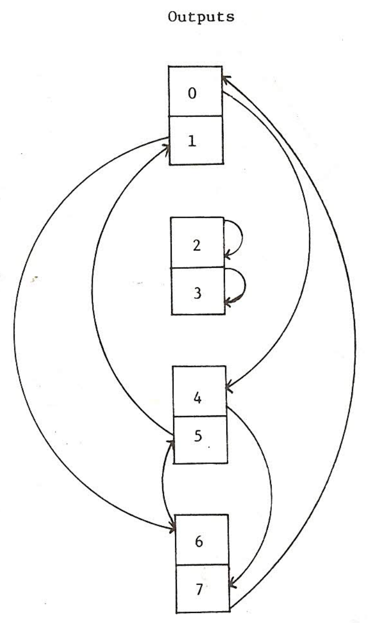


Fig. 5. Cycles corresponding to Fig. 3.

```
line   procedure LEAST($k$)
          global arrays $R, M$; array $T$
1         $M(i) := i$
2         for $b := 0$ to $k - 2$ do
3            $\langle T(i), R(i)\rangle \leftarrow \langle M(R(i)), R(R(i))\rangle$
4            $M(i) := \min\{M(i), T(i)\}$
5         end
6      end LEAST
```

Algorithm 2

It should be easy to see that after iteration b of the **for** loop, $R(i)$ points to a node originally 2^{b+1} units away from i (distance is measured along the cycle containing i with successive nodes on a cycle being one unit apart). Also, after iteration b, $M(i)$ is the minimum index among nodes up to 2^{b+1} units away from i initially. Consequently, following iteration $b = k - 2$, $M(i)$ is the minimum element on the cycle containing i. The complexity of procedure LEAST is $O(k)$.

Procedure SET_UP is the complete algorithm to determine the settings of all switches in $B(n)$. $D(0:N - 1)$ is the desired input–output permutation. The algorithm sets $S(i, j) = 0$ or 1 depending on the required state of switch i of stage j, $0 \le i < N/2$ and $0 \le j \le 2n - 2$. It is assumed that $S(i, 0:2n - 2)$ are memory cells associated with PE(i), $0 \le i < N/2$. If this much memory is not available, then the PE's can output the switch settings of each stage as they are determined. Each iteration of the loop of lines 2 to 15 determines the settings for the first and last stages of all the $B(k)$ Benes networks in the decomposition of $B(n)$ (see Fig. 1). Thus, in the first iteration (i.e., when $k = n$) the switch settings for stages 0 and $2n - 2$ are determined. The permutations D_l and D_u are also determined. In the next iteration the settings for stages 1 and $2n - 3$ (i.e., the first and last stages of both the $B(n - 1)$ networks of Fig. 1) are obtained, and so on. When determining the switch settings for the first and last stages of all the $B(k)$ networks, $\mathrm{PE}(2^k \cdot i), \cdots, \mathrm{PE}(2^k \cdot i + 2^k - 1)$ represent the ith $B(k)$ network (indexing top to bottom in a stage), $0 \le i < 2^{n-k}$. In each 2^k block, D specifies the permutation to be realized by the corresponding $B(k)$ network. That is, $\{D(2^k \cdot i), \cdots, D(2^k \cdot i + 2^k - 1\}$ is a permutation of $\{2^k \cdot i, \cdots, 2^k \cdot i + 2^k - 1\}$. Note that $2^k \cdot i$ is the index of output "0" of the ith $B(k)$ network and $(2^k \cdot i)_0 = 0$. As a result of this, our policy of routing cycles with an even least element through the upper $B(k - 1)$ network can still be applied. In SET_UP we use rule R1 to determine the switch settings for the last stage of each $B(k)$ network. Rules R2–R4 are replaced by the following equivalent rules (when $k = n = \log N$):

R2′: $S(i/2, 0) = i_0, D(i) \in J_u$

R3′: $D_u(i/2) = D(i)/2, D(i) \in J_u$

R4′: $D_l(N/2 \cdot + i/2) = N/2 + D(i)/2, D(i) \notin J_u.$

When $k < n$, rules R_3'–R_4' must be applied with respect to each 2^k-block.

```
line   procedure SET_UP(n)
          //set up the Benes network $B(n)$//
          global $D(0{:}N-1), M(0{:}N-1), E(0{:}N-1),$
             $S(0{:}\ N/2-1, 0{:}2n-2)$
 1        $k := n$
 2        loop //set_up stages $s$ and $2n-2-s$//
 3           $s := n - k$  //$D$ defines permutations for
                each $2^k$-block//
 4           call EQUIV
 5           call LEAST($k$)
 6           $M(i) := (M(i))_0$  //set $M(i)$ to 0 if output
                $i \in J_u$//
 7           $F(E(i)) \leftarrow M(i)$
 8           $S(i/2, s) \leftarrow i_0, (F(i) = 0)$  //first stage//
 9           if $k = 1$ then exit //go to line 16//
10           $S(i/2, 2n-2-s) \leftarrow i_0, (M(i) = 0)$ //last
                stage//
11           $D(i^{(0)}) \leftarrow D(i), (F(i) \neq i_0)$
12           $D(i_{n-1} \cdots i_k i_0 i_{k-1} \cdots i_1) \leftarrow D(i)$
13           $D(i) := i_{n-1:k-1}(D(i))_{k-1:1}$
14           $k := k - 1$
15        repeat  //go to line 2//
16        end SET_UP
```

Algorithm 3

Line 4 determines the equivalence classes for the permutation D. There is no need to explicitly consider the 2^k-blocks of D, as D does not cross 2^k-block boundaries. Line 5 determines the least element in each equivalence class and line 6 records if the least element is even ($M(i) = 0$) or odd ($M(i) = 1$). If the least element in an equivalence class is even, then all elements in that class must be routed through the upper $B(k - 1)$ networks. Line 7 sets F such that $F(i) = 0$ iff input i is to be routed through an upper $B(k - 1)$ network (i.e., $D(i) \in J_u$). This makes it easy to incorporate rules R2′–R4′. The construct $F(i) = 0$ of line 8 is a mask. Thus, this line of the algorithm is executed only on those processors i for which $F(i) = 0$. In line 8 the switch settings for the first stage of all $B(k)$ networks are obtained (see rule R2′). When $k = 1$, the $B(k)$ networks have only one stage and line 9 terminates the loop. When $k \neq 1$, the switch settings for the last stage of all $B(k)$ networks are obtained in line 10 (see rule R1). Line 11 updates D so that $D(i)$ corresponds to a "destination" for an upper $B(k - 1)$ network if i is even. This "destination" is relative to the $B(k)$ network and not to the individual $B(k - 1)$ networks. Line 12 routes the D values to PE's whose index corresponds to the left-hand side of R3′ and R4′. This is essentially a division by 2 within 2^k-blocks. Line 13 replaces each $D(i)$ by the quantities on the right-hand side of $R3'$ and $R4'$. Again, note that R3′ and R4′ are to be interpreted as being carried out on each 2^k-block (or equivalently for each $B(k)$ network). For the ith 2^k-block the indices are $2^k \cdot i, 2^k \cdot i + 1, \cdots, 2^k \cdot i + 2^k - 1$. A division by 2 requires us to shift bits $k - 1, \cdots, 1$ one position right and define the new bit $k - 1$ to be zero. Adding 2^{k-1} requires changing bit $k - 1$ to 1. One may verify that lines 11 to 13 implement R3′ and R4′ for each $B(k)$ network. The complexity of SET_UP is readily seen to be $0(n^2) = 0(\log^2 N)$. This is due to calling LEAST in line 5.

We now consider an example to illustrate how SET_UP works. Let $n = 3$ and $D = (0, 2, 4, 6, 1, 3, 5, 7)$. The equivalence classes are $\{0, 3\}, \{4, 7\}, \{1, 2\}$, and $\{5, 6\}$. $J_u = \{0, 3, 4, 7\}$. If we define $I_u = \{i | D(i) \in J_u\}$ to be the set of *inputs* that must be routed through the B_u network, we have $I_u = \{0, 2, 5, 7\}$. Consequently, $S(, 0) = (0, 0, 1, 1), S(, 4) = (0, 1, 0, 1), D_u = (0, 2, 1, 3), D_l = (5, 7, 4, 6)$, and $D = (0, 2, 1, 3, 5, 7, 4, 6)$. The equivalence classes for the $B(2)$ networks are $\{0, 3\}, \{1, 2\}, \{4, 7\}$, and $\{5, 6\}$. So $J_u = \{0, 3, 4, 7\}, S(, 1) = (0, 1, 1, 0), S(, 3) = (0, 1, 0, 1)$, and $D = (0, 1, 3, 2, 5, 4, 6, 7)$. This yields $S(, 2) = (0, 1, 1, 0)$. Fig. 6 shows the resulting switched network.

III. Set-Up Algorithms for MCC's, CCC's, and PSC's

When PE's are not directly connected to every other PE, the data transfer steps in procedure SET_UP could take more than 0(1) time. In general, lines 7, 8, 10, 11, and 12 of SET_UP together with lines 2 and 3 of EQUIV and line 3 of LEAST can be accomplished by means of a sort. For example, to obtain the routing required for line 7 of SET_UP we can create the records $\langle E(i), M(i)\rangle$ in $\mathrm{PE}(i)$, $0 \le i < N$, and then sort these records on the field $E(i)$. The results of the sort is that record $\langle E(i), M(i)\rangle$ will reside in $\mathrm{PE}(E(i))$ following the sort. Similarly, for line 8 we set up the record $\langle i/2, i_0\rangle$ in $\mathrm{PE}(i)$ if $F(i) = 0$. If $F(i) \neq 0$ then the record $\langle \infty, \infty\rangle$ is set up. Sorting on the first field will result in the i_0 values being routed to the correct PE's. Only line 3 of LEAST cannot be handled

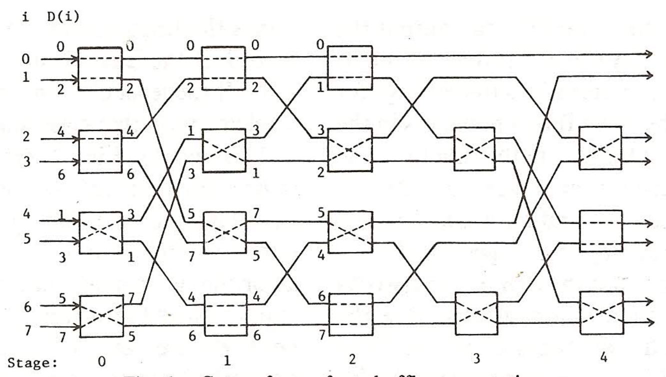


Fig. 6. Set up for perfect-shuffle permutation.

by a single sort scheme (i.e., the scheme for lines 7 and 8 of SET_UP). Line 3 of LEAST can be carried out using *two* sorts, one for each of the following instructions:

1) $V(R(i)) \leftarrow i$

2) $\langle T(V(i)), R(V(i)) \rangle \leftarrow \langle M(i), R(i) \rangle$.

If S_N is the time needed to sort on an N PE SIMD computer, then the complexity of SET_UP using the above simulation becomes $0(S_N \log^2 N)$. Some speed-up can be obtained when PE(i) is directly connected to PE($i^{(0)}$) (as is the case for MCC's, CCC's, and PSC's). In this case line 2 of EQUIV and line 11 of SET_UP require only 0(1) time and a sort is not needed. Also, line 3 of LEAST can be implemented as below:

$$3.1 \quad V(i^{(0)}) \leftarrow (R(i))^{(0)}$$

$$3.2 \quad \langle T(V(i)), R(V(i)) \rangle \leftarrow \langle M(i), R(i) \rangle.$$

The correctness of 3.1 as a replacement for 1) above can be shown by establishing that $R(i) = j$ implies $R(j^{(0)}) = i^{(0)}$ during all iterations of LEAST. One may easily verify that this is true for the example of Fig. 5. We leave it to reader to establish the correctness of this statement for all permutations D. Substituting into 3.1, we see that 3.1 simply sets $V(R(j^{(0)})) = j^{(0)}$ for all j. This is equivalent to 1). Hence, line 3 of LEAST needs only one sort (step 3.2) and a direct data transfer (step 3.1).

Additional sources of speed-up are lines 8 and 10 of SET_UP. Rather than keep $S(i,j)$ on PE(i), we could keep $S(i,j)$ on PE($2i$), $0 \le i < N/2$. If this is done, lines 8 and 10 become

$$8' \quad S(i, s) := F(i), (i_0 = 0)$$

$$10' \quad S(i, 2n - 2 - s) := M(i), (i_0 = 0).$$

The correctness of 8′ and 10′ as replacements for 8 and 10 is readily seen by recalling that $F(i) = 1 - F(i^{(0)})$ and $M(i) = 1 - M(i^{(0)})$ for all i, $0 \le i < N$. Note that lines 8′ and 10′ each require only 0(1) time.

Sorting on an N PE CCC and PSC takes $0(\log^2 N)$ time [1]. Hence, $B(n)$ can be set up in $0(\log^4 N)$ time. If $N^{1+1/k}$ PE's are available then we can sort on a CCC and a PSC in $0(k \log N)$ time [3]. In this case $B(n)$ can be set up in $0(k \log^3 N)$ time.

It takes $0(N^{1/2})$ time to sort on an $N^{1/2} \times N^{1/2}$ MCC ([2] and [8]). Hence, the preceding discussion implies that SET_UP can be run on an MCC in $0(N^{1/2} \log^2 N)$ time. The computing time can be reduced to $0(N^{1/2})$ by making the observation that during iteration k of the loop in procedure SET_UP, the data movement required in lines 4, 5, 7, and 12 is local to the 2^k-blocks. The sorts necessary to implement lines 4 (i.e., line 3 of EQUIV), 7, and 12 can therefore be restricted to sort only 2^k-blocks. If the PE's are indexed in "shuffled row-major" order, then a 2^k-block of PE's forms a $2^{\lfloor k/2 \rfloor} \times 2^{\lceil k/2 \rceil}$ region of the $N^{1/2} \times N^{1/2}$ PE array (see [4] or [6]). The sorting algorithm of [8] is easily modified to sort all 2^k-blocks in a $N^{1/2} \times N^{1/2}$ MCC in $0(2^{\lceil k/2 \rceil})$ time when shuffled row-major indexing is used.

Therefore, using shuffled row-major indexing for the MCC, lines 4, 7, and 12 of SET_UP each require $0(2^{\lceil k/2 \rceil})$ time. Using the algorithm of Nassimi and Sahni [6] for finding the connected components of degree 2 graphs, procedure LEAST can be implemented on an MCC so as to have a complexity of $0(2^{\lceil k/2 \rceil})$. And, using 8′ and 10′ in place of lines 8 and 10, the data movement in the remaining lines of procedure SET_UP take only 0(1) time. Hence, iteration k of the loop takes $0(2^{\lceil k/2 \rceil})$ time. So the complexity of SET_UP on a two-dimensional MCC is

$$0\left(\sum_{k=1}^{n} 2^{\lceil k/2 \rceil}\right) = 0(2^{n/2}) = 0(N^{1/2}).$$

IV. Conclusions

We have shown that while Waksman's set-up algorithm appears to be highly sequential in nature, it can in fact be parallelized. Our parallel set-up algorithm for an $N \times N$ Benes network has a complexity of $0(\log^4 N)$ on an N PE PSC or CCC. If $N^{1+1/k}$ PE's are available, then only $0(k \log^3 N)$ steps are needed. On an $N^{1/2} \times N^{1/2}$ MCC, our algorithm will have complexity $0(N^{1/2})$.

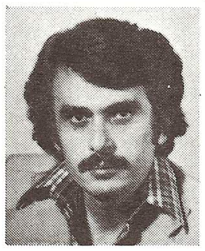
It should be pointed out that the above analysis has been carried out under the same assumptions as used in [2] through [8]. In practice, one would probably use logic with a fixed fan-in and fan-out. Hence, one would experience an $0(\log n)$ logic delay per instruction. Therefore, it is necessary to multiply all our times by a $\log n = \log\log N$ factor.

References

[1] K. Batcher, "Sorting networks and their applications," in *Proc. 1968 Spring Joint Comput. Conf.*, vol. 32, AFIPS, Montvale, NJ, pp. 307–314.

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

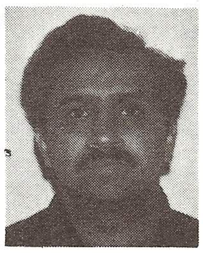
[3] ——, "Parallel permutation and sorting algorithms and a new generalized-connection-network," Univ. of Minnesota, Minneapolis, Tech. Rep. 79-8, 1979; also in *J. Ass. Comput. Mach.*, to be published.
[4] ——, "Data broadcasting in SIMD computers," Univ. of Minnesota, Minneapolis, Tech. Rep. 79-17, 1979; also in *IEEE Trans. Comput.*, vol. C-30, pp. 101–107; Feb. 1981.
[5] ——, "A self-routing Benes network," Univ. of Minnesota, Minneapolis, Tech. Rep. 79-13, 1979; also in *IEEE Trans. Comput.*, vol. C-30, pp. 332–340, May 1981.
[6] ——, "Finding connected components and connected ones on a mesh-connected parallel computer," Univ. of Minnesota, Minneapolis, Tech. Rep. 79-18, 1979; also in *SIAM J. Comput.*, vol. 9, pp. 744–757, Nov. 1980.
[7] C. Thompson, "Generalized connection networks for parallel processor intercommunication," *IEEE Trans. Comput.*, vol. C-27, pp. 1119–1125, Dec. 1978.
[8] C. Thompson and H. Kung, "Sorting on a mesh-connected parallel computer," *Commun. Ass. Comput. Mach.*, vol. 20, no. 4, pp. 263–271, 1977.
[9] A. Waksman, "A permutation network," *J. Ass. Comput. Mach.*, vol. 15, pp. 159–163, Jan. 1968.
[10] K. N. Levitt *et al.*, "A study of the data communications problems in a self-repairable multiprocessor," in *Proc. 1968 Spring Joint Comput. Conf.*, AFIPS, vol. 32, 1968, pp. 515–527.
[11] D. C. Opferman and N. T. Tsao-Wu, "On a class of rearrangeable switching networks, Part I: Control algorithm," *Bell Syst. Tech. J.*, vol. 5, pp. 1579–1600, May–June 1971.
[12] D. Stevenson, "The Phoenix project," presented at 7th Annu. IEEE Symp. Comput. Arch., 1980.
[13] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901–1909, Dec. 1966.
[14] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Trans. Comput.*, vol. C-28, pp. 907–917, Dec. 1979.
[15] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153–161, Feb. 1971.
[16] G. Lev, N. Pippenger, and L. G. Valiant, "A fast parallel algorithm for routing in permutation networks," *IEEE Trans. Comput.*, vol. C-30, pp. 93–100, Feb. 1981.



David Nassimi received the M.S. degree in electrical engineering in 1975 and the M.S. and Ph.D. degrees in computer science in 1978 and 1979, respectively, all from the University of Minnesota, Minneapolis.

He is currently an Assistant Professor with the Department of Electrical Engineering and Computer Science at Northwestern University, Evansville, IL. His research interests include parallel algorithms, interconnection networks, and computer architecture.

Dr. Nassimi is a member of the Association for Computing Machinery and the Computer Society of the IEEE.



Sartaj Sahni (M'79) received the B. Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1970 and the M.S. and Ph.D. degrees in computer science from Cornell University, Ithaca, NY, in 1972 and 1973, respectively.

Currently, he is a Professor of Computer Science at the University of Minnesota, Minneapolis. He has published articles in *JACM*, *JCSS*, *SIAM Journal of Computing*, IEEE TRANSACTIONS ON COMPUTERS, *ACM Transactions on Mathematical Software*, *Operations Research*, *International Journal on Theoretical Computer Science*, *Mathematics of Operations Research*, and the *Journal of Statistical Computation and Simulation*. His publications are on topics concerned with the design and analysis of efficient algorithms, parallel computing, interconnection networks, and design automation. He is also the coauthor of *Fundamentals of Data Structures* and *Fundamentals of Computer Algorithms*, and the author of *Concepts in Discrete Mathematics*.