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 $O(\log^2 N)$ time when a fully interconnected parallel computer with $N$ processing elements is used. The algorithm runs in $O(N^{1/2} \log(N))$ time on an $N^{1/2} \times N^{1/2}$ mesh-connected computer and $O(\log^2 N)$ time on both a cube connected and a perfect shuffle computer with $N$ processing elements. It runs in $O(k \log^3 N)$ 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 algorithms, 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 generalization 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 $O(N \log N)$ time on a single processor computer. Thus, the set-up time for the network is much larger than the network delay [which is $O(\log N)$]. One cannot set up the Benes network in less than $O(N \log N)$ time using a single processor as the network has $O(N \log N)$ 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 work for all permutations. Another alternative is to precompute and store the switch settings for some permutations. This requires $O(N \log N)$ 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 $i$th 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 $O(1)$.

0018-9340/82/0200-0148$00.75 © 1982 IEEE
2) Mesh-Connected Computer (MCC): In this model, the PE’s may be thought of as logically arranged as in a k-dimensional array \( A(n_k-1, n_{k-1} \cdots n_0) \), where \( n_i \) is the size of the \( i \)-th dimension and \( M = n_k \cdots n_2 \cdots n_0 \). The PE at location \( A(i_k-1, \cdots i_0) \) is connected to the PE’s at location \( A(i_k-1, \cdots i_1 \pm 1, i_0), 0 \leq j \leq k \), provided they exist.

3) Cube Connected Computers (CCC): Assume that \( M = 2^q \) and let \( i_q \cdots i_0 \) be the binary representation of \( i \) for \( i \in [0, M-1] \). Let \( i^{(b)} \) be the number whose binary representation is \( i_q \cdots i_1 i_0 \cdots i_1 \) where \( i_0 \) is the complement of \( i_0 \), 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 \( M, q, i, j \), and \( i^{(b)} \) be as in the cube model, PE(i) is connected to PE(\( i^{(b)} \)), PE(\( i_q \cdots i_1 i_0 \cdots i_1 \)), and PE(\( i_q \cdots i_1 i_0 \cdots i_1 \)). 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, \( 2^k \) connections per PE (except boundary PEs’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 \( B(n) = 0(\log^2 N) \) on an \( N \) PE CIC, \( 0(\log^N N) \) on an \( N \) PE CCC and on an \( N \) PE PSC, \( O(\log \log N) \) on an \( N \) PE MCC and \( N^{1+1/K} \) 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 \( O(\sqrt{N \log N}) \) 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 \( i/2 \)). 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 \( (x_i, y_j) \). 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 \( (y_0, x_1, y_2, x_5, y_3, x_3, y_0) \). (The bipartite multigraph is for exposition purposes; the algorithm only uses the permutation \( D \) and its inverse.) 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 \( (x_1, y_3) \) 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].
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 $D_1$ and $D_{2n}$, 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: $D_{2n}(E(i)/2) = i/2$
R4: $D_{1}(N/2 + E(i)/2) = N/2 + D((E(i))^{(0)})/2$.

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 $J = \{0, 1, \ldots, N - 1\}$ 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 $B_u(n - 1)$. 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 $0 \notin J_u$, 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 $i \notin Q$, 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 $j \in Q_2$, $j^{(0)} \in Q_1$. Moreover $Q_0 \subseteq J_u$ and $Q_1 \subseteq J - J_u$. Note that $0 = \min\{i | i \in Q_0\}$ and $1 = \min\{k | k \in Q_1\}$. 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 $j^{(0)} \notin Q_0$ and $Q_1$. 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 $J_u = \{0, 2, 4, 7\}$. 

---

**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.
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 \( j \), 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.

```plaintext
line procedure EQUIV
// determine equivalence classes/
global arrays R, D, E
1 R(i) := (D(i))(0)
2 R(i(0)) := R(i)
3 (E(D(i)), R(D(i))) := (i, R(i))
end EQUIV
```

Algorithm 1

In specifying our algorithms, we have made use of two kinds of assignment statements. Those using the operator "\( \Leftarrow \)" 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 \( i \), \( i_0 \) and \( i_0^{(0)} \) are as defined earlier, and \( (i)_c \) denotes the integer with binary representation \( i_1 i_{i-1} \cdots i_0 \).

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 \( (D(i))(0) \) must be in the same class. So we set \( R(D(i)) := (D(i))(0) \) in line 3. (Observe that for \( j = D(i), r(0) = (D(i))(0) \).) 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, \( PE(i \star 2^k) \) defines a \( 2^k \)-block for each \( i, 0 \leq i < 2^{n-k} \) (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 \( D \) 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 \( i, 0 \leq i < N \). It assumes that cycles are localized to \( 2^k \)-blocks and that no cycle is longer than \( 2^{k-1} \).

```plaintext
line procedure LEAST(k)
global arrays R, M; array T
1 M(i) := i
2 for b := 0 to k - 2 do
3 \( (T(i), R(i)) \) := \( (M(R(i)), R(R(i))) \)
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 \( j \), \( 0 \leq i < N/2 \) and \( j \leq 2n - 2 \). It is assumed that \( S(i, 0: 2n - 2) \) are memory cells associated with \( PE(i), 0 \leq i < N/2 \). 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) Bens 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 $D_i$ 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, PE$(2^k)$, $\cdots$, PE$(2^k + 2^k - 1)$ represent the $t$th B(k) network (indexing top to bottom in a stage), $0 \leq t < 2^n - k$. In each $2^k$ block, $D$ specifies the permutation to be realized by the corresponding B(k) network. That is, $|D(2^k)\cdots, D(2^k + 2^k - 1)|$ is a permutation of $[2^k\cdots, 2^k + 2^k - 1]$. Note that $2^k$ is the index of output “0” of the $t$th B(k) network and $D(t)$ is $0$. 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': $D_j(N/2 + i/2) = N/2 + D(i)/2, D(i) \notin J_u$

When $k < n$, rules R3'-R4' must be applied with respect to each $2^{k-1}$-block.

line procedure SET-UP(n)

//set up the Bens 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 $k := n$

loop //set-up stages s and 2n−2−s//

2 $s := n - k$ //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 $D(0) \leftarrow D(i), (F(i) \neq i_0)$

12 $D(i_0+1\cdots i_{k−1}+(k−1) \leftarrow D(i)$

13 $D(i) := i_{k−1}+1+(k−1)\leftarrow D(i)$

14 $k := k − 1$

15 repeat //go to line 2//

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 deter-

mines 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 $F(i) = 0$ if 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 $i$th $2^k$-block the indices are $2^k;i, 2^k;i + 1, \cdots, 2^k;i + 2^k - 1$. 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 $O(n^2) = O(log^2 N)$. 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, $S(0) = (0, 0, 1, 1), S(4) = (0, 1, 0, 1), D_u = (0, 2, 1, 3), D_i = (5, 7, 4, 6)$, 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 $J_u = (0, 3, 4, 7), S(0) = (0, 1, 1, 0), S(3) = (0, 1, 0, 1)$, 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 $(E(i), M(i))$ in PE(i), $0 \leq i < N$, and then sort these records on the field $E(i)$. The results of the sort is that record $(E(i), M(i))$ will reside in PE(E(i)) following the sort. Similarly, for line 8 we set up the record $(i/2, i_0)$ in PE(i) if $F(i) = 0$. If $F(i) \neq 0$ then the record $(\infty, \infty)$ 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
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 \( O(S_N \log^2 N) \). 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 \( O(1) \) time and a sort is not needed. Also, line 3 of LEAST can be implemented as below:

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

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

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 \( V(R(i^{(0)})) = j^{(0)} \) 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\((2i)\), \( 0 \leq i < N/2 \). If this is done, lines 8 and 10 become

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

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

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 \( i, 0 \leq i < N \). Note that lines 8' and 10' each require only \( O(1) \) time.

Sorting on an \( N \) PE CCC and PSC takes \( O(\log^2 N) \) time [1]. Hence, \( B(n) \) can be set up in \( O(\log^4 N) \) time.

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 \( O(\log n) \) delay per instruction. Therefore, it is necessary to multiply all our times by a log \( n = \log n \) factor.

REFERENCES


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, Evanston, 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.