III. Conclusions

It has been shown that the proposed method is the fastest among known pipelined vector-reduction methods. While the SM- and AM-methods proposed by Ni and Hwang are improvements of Kogge's method for $n \geq q$, the SM- and AM-methods perform worse than Kogge's method for $n < q$. The proposed $\overline{\text{SM}}$- and $\overline{\text{AM}}$-methods are faster than Kogge's method for all values of n. The hardware implementation requirements for the $\overline{\text{SM}}$- and $\overline{\text{AM}}$-method are about the same as for the SM- and AM-methods.

References

[1] D. J. Kuck, *The Structure of Computers and Computations, Vol. 1.* New York: Wiley, 1978.

[2] P. M. Kogge, *The Architecture of Pipelined Computers.* New York: McGraw-Hill, 1981.

[3] L. M. Ni and K. Hwang, "Vector reduction methods for arithmetic pipelines," *IEEE Trans. Comput.*, vol. C-34, no. 5, May 1985.

[4] H. X. Lin and H. J. Sips, "Vector-reduction algorithms and architectures," *J. Parallel Distributed Comp.*, vol. 5, no. 2, 1988.

Setup Algorithms for Cube-Connected Parallel Computers Using Recursive Karnaugh Maps

A. Yavuz Oruç and M. Mittal

***Abstract*— This paper presents optimal setup procedures for cube-connected networks. The setup patterns include paths, transpositions, cycles, and one-pass permutations. For an N-input cube-connected network, the procedure for paths requires $O(\log_2 N)$ steps, and $O(N)$ space, that for transpositions and cycles requires $O(N)$ steps, and $O(N)$ space, and that for permutations takes $O(N \log_2 N)$ steps, and $O(N)$ space. Furthermore, the time complexities of the setup procedures for transpositions, cycles, and permutations can be improved as $O(\log_2 N)$ by using $O(N)$ processors.**

***Index Terms*—Cube-connected network, cycle, Karnaugh map, permutation, setup algorithm, transposition.**

I. Introduction

A parallel computer with cube interconnection topology can be formed in one of two ways [1]–[4], [11]–[13]: 1) by placing N processors, where $N = 2^r$, at the vertices of an N-cube whose edges represent physical links among the processors; 2) by setting up an N-input, N-output and r-stage network of 2×2 crossbar cells such that the cells in the ith stage correspond to the edges along the ith dimension of the N-cube. A cube-connected network is then used to interconnect system components such as processors and memory units.

In this paper, we consider the routing problems that arise in parallel computers with cube-connected networks. The term *routing* here refers to specifying connection patterns between the inputs and the outputs of a cube-connected network. More specifically, we present fast algorithms to set up paths, pairwise exchanges, cycles, and permutations on a cube-connected network. These routing problems have been previously addressed in the literature [5], [7], [8], [11], [14]–[16]. Setting up a path between two processors can be done by using the self-routing property of a cube-connected network [7]. On the other hand, setting up transpositions, cycles, and permutations is not so straightforward.

A cube-connected network cannot realize all permutations on a set of N elements; previous work mostly characterized the one-pass permutations that can be realized by a cube-connected network, with conditions on the binary representations of the inputs and outputs [7], [11], [16], [8], [5]. One then uses these conditions to determine whether a specific permutation is realizable by a cube-connected network in a single pass, and if so, to set up the required paths. This approach works quite well for permutations which satisfy the asserted conditions.

Lawrie [7] developed another method which decomposes a specific permutation into paths among pairs of inputs and outputs, and sets the cells until a conflict is encountered. In this paper, rather than allowing any conflict to arise as paths are being set up, we monitor possible conflicts of a given permutation stage by stage.

We view a cube-connected network as a three-part structure: two cube-connected subnetworks and an extra stage of 2×2 cells, each drawing exactly one input from each subnetwork as shown in Fig. 1. The setup procedure, employing the *divide and conquer* approach, proceeds from the output side recursively through the network toward the input side as long as the cells within each subnetwork can accommodate the connection requirements. This divide and conquer process is facilitated with the use of recursive Karnaugh maps which provide a compact characterization of the interconnection structure of a cube-connected network without resorting to algebraic maps to describe the functionality of the switching cells and the links between the stages.

A setup procedure for realizing pairwise exchanges, or transpositions between pairs of inputs and outputs, is also presented. Even though the setup procedure for permutations can also be used for transpositions, a separate one is developed as it requires only $O(N)$ steps for an N-stage cube-connected network. As a byproduct, an $O(N)$ setup procedure for certain cycles is also produced.

II. Recursive Karnaugh Map Representation

Karnaugh maps [6] are a common tool of logic designers for simplifying Boolean expressions. They can also be used to capture the structure of a cube-connected network as shown in Fig. 2. Adjacent cells in the Karnaugh map correspond to the pairings of symbols at the inputs (outputs) of the switching cells in the network. Karnaugh maps such as the one shown in Fig. 2 provide a convenient way of representing a cube-connected network. However, they need to be put in a recursive form to exploit the recursive structure of such a network.

Definition 1: An *r-half,* denoted $H_r; r \geq 0$ is a Karnaugh map with 2^r cells, and is recursively defined as $H_0 = [0]; H_r = [H_{r-1}; H_{r-1} + 2^{r-1}]; r \geq 1$ such that any two cells x and y in H_r are adjacent if and only if *either* they are also adjacent in H_{r-1}, or $H_{r-1} + 2^{r-1}$, *or* $|x - y| = 2^{r-1}$. □

Here, $H_{r-1} + 2^{r-1}$ is a vector consisting of as many elements as there are in H_{r-1} and whose entries are determined by adding 2^{r-1} to those in H_{r-1}. In Fig. 3, the entries of H_1 are formed by concatenating H_0 and $H_0 + 2^0$; H_2 and H_3 are obtained similarly.

It is evident from Definition 1 that, for any $r \geq 1$, the r-half H_r consists of two halves which will be denoted $\mathcal{H}_0$ and $\mathcal{H}_1$, where $\mathcal{H}_0 \Longleftrightarrow H_{r-1}$, and $\mathcal{H}_1 \Longleftrightarrow H_{r-1} + 2^{r-1}$. It can be easily be shown that the binary representation of any number between 0 and $N - 1$, where $N = 2^r$, corresponds to a unique location in the r-half. That is, each number is mapped into a distinct cell such that its most significant bit matches the index of a half at the outermost level, the next most significant bit matches the index of a half in the next level, and so on until the least significant bit matches the index of a half at the innermost level.

Manuscript received January 29, 1988; revised August 15, 1988.

A. Y. Oruç is with the Department of Electrical Engineering, University of Maryland, College Park, MD 20742.

M. Mittal is with Digital Equipment Corporation, Hudson, MA 01748.

IEEE Log Number 9037993.

0018–9340/91/0200–0217$01.00 © 1991 IEEE

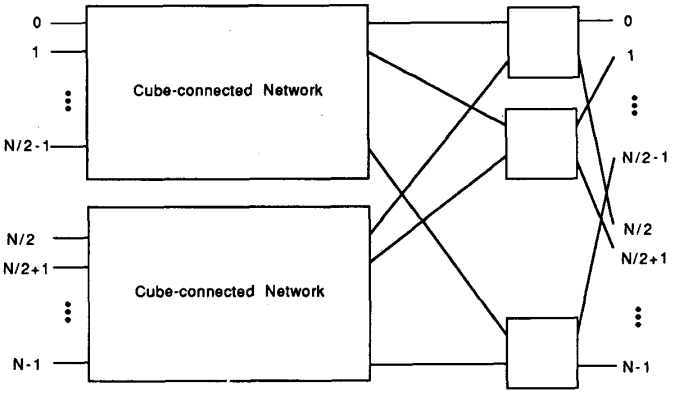
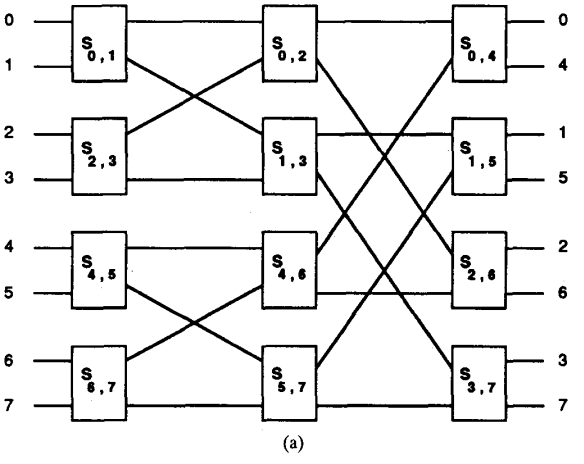
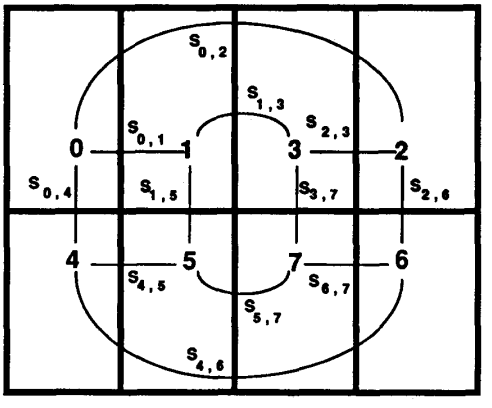


Fig. 1. The recursive representation of a cube-connected network.



(a)



(b)

Fig. 2. (a) A cube-connected network and (b) its Karnaugh map.

The recursive Karnaugh maps are naturally tied with the subnetwork structure of a cube-connected network as follows. Each half, $\mathcal{H}_0$ or $\mathcal{H}_1$, represents an $N/2$-input and $(r-1)$-stage cube-connected network via the adjacencies among its cells. Furthermore, the adjacencies formed by pairing each cell in $\mathcal{H}_0$ with its adjacent counterpart in $\mathcal{H}_1$ correspond to another set of $N/2$ 2×2 cells. These three components combined together form an (N-input, r-stage) cube-connected network as shown in Fig. 1. This equivalence between map H_r and an r-stage cube-connected network forms the basis of all the routing procedures presented here.

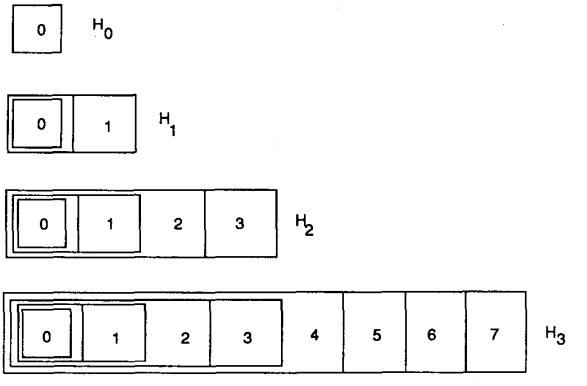


Fig. 3. The recursive Karnaugh map for $N = 8$.

III. Realization of Paths

This section presents a recursive procedure to realize paths in a cube-connected network. The objective in developing a recursive version is to show that the recursive Karnaugh maps form a compact data structure that can be used in connection with any routing algorithm for cube-connected networks.

Definition 2:

a) A sequence of cells $x_1x_2\cdots x_m$ in H_r is said to be *monotonically increasing* if x_i is adjacent to $x_{i+1}; 1 \le i \le m-1$, and $|x_{i+1}-x_i| \ge |x_i - x_{i-1}|; 2 \le i \le m-1$. 0137 and 3204 are both monotonically increasing sequences in H_3.

b) Cell x is said to be the *dual* of cell y in H_r if $|x-y| = 2^{r-1}$. □

Theorem 1: Let x and y be any two cells in H_r. There exists one and only one monotonically increasing sequence which starts with x and ends with y.

Proof: The proof uses induction on r. Assuming that there exists one and only one monotonically increasing sequence between every two cells in $\mathcal{H}_0 = H_{r-1}$, we need to show that there also exists one and only such sequence between every two cells in H_r. There are three cases to consider. If x and y both belong to $\mathcal{H}_0$, the statement is true by the induction assumption. If they both belong to $\mathcal{H}_1$, then we know that there exists a monotonically increasing sequence between the duals of x and y in $\mathcal{H}_0$. By translating each of the cells in this

sequence to their duals in $\mathcal{H}_1$, a monotonically increasing sequence between x and y is obtained. The uniqueness of this sequence follows from that of the original sequence. Finally, if x and y belong to different halves of H_r, then we have a unique monotonically increasing sequence from one of the cells, say x, to the dual of the other. Let this sequence be $xx_1x_2\cdots x_i$ where x_i is the dual of y. Since $|y - x_i| > |x_j - x_{j-1}|$ for all $j = 2, 3, \cdots, i$; $xx_1x_2\cdots x_iy$ is also a monotonically increasing sequence. Its uniqueness follows from that of $xx_1x_2\cdots x_i$. □

As an immediate corollary, we have:

Corollary 1.1: There exists one and only path from any input x to any output y in a cube-connected network.

As described in the following procedure, this path is established by appropriately setting the switching cells in the network whose inputs correspond to the adjacent cells in the monotone increasing sequence between x and y. For example, for $x = 0$ and $y = 13$, the monotonically increasing sequence is 0 1 5 13. To set up the path from input 0 to output 13, the top input of cell $S_{0,1}$ is connected to its bottom output, the top input of cell $S_{1,5}$ is connected to its bottom output, and the top input of cell $S_{5,13}$ is connected to its bottom output.

```
Procedure PATH(x, y, r; var P);
If r = 1
  then P := x, y
  else
    begin
    If HALF(x, r) = HALF(y, r)
      then P := PATH(x, y, r − 1)
      else P := PATH(x, DUAL(y, r), r − 1), y
    end;
endprocedure;
```

The function $\mathrm{HALF}(x, r)$ determines the half to which x belongs in H_r, and $\mathrm{DUAL}(y, r)$ computes the dual of y in H_r. Each of these functions requires checking a single bit and hence can be evaluated in $O(1)$ time. The time complexity of PATH is thus determined by the number of times it calls itself, which is $O(r) = O(\log_2 N)$. The optimality of the number of steps follows from the fact that at least $\log_2 N$ bits must be examined to decode the end point of a path. The space complexity of PATH is also determined by the number of recursive calls which is again $O(\log_2 N)$.

IV. Realization of Transpositions and Cycles

A transposition over a set of symbols is a permutation that fixes all but two of these symbols. Transpositions form an important family of permutations because they represent pairwise communications among a set of processors. This section presents an optimal routing procedure for realizing such maps on cube-connected networks. It is also shown that a transposition of any two symbols can be realized in at most two passes. The proof of this result indirectly leads to the characterization of certain cycles which are realizable by a cube-connected network in a single pass.

Let x and y be any two symbols in H_r. If x and y are adjacent in H_r, then there exists a cell in the corresponding cube-connected network with inputs x and y. Therefore, the two can be transposed in one pass. On the other hand, if x and y are not adjacent in H_r, then x and y cannot be transposed in one pass as the following result shows.

Theorem 2: Symbols x and y can be transposed by a cube-connected network in one pass if and only if x and y are adjacent in H_r.

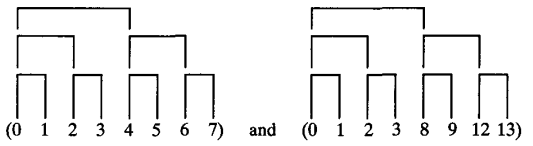
Proof: See [9].

The above theorem asserts that at least two passes are needed to transpose any two symbols when they are not adjacent in H_r. The next question that arises is whether it is possible to realize any transposition in two passes. The answer to this question is in the affirmative as shown below.

First a few facts about cycle maps are needed.

Definition 3: A cycle of m symbols, denoted $(x_1x_2\cdots x_m)$, is a permutation which maps x_i to $x_{i+1(\mathrm{mod}\, m)}$. A cycle $(x_1x_2\cdots x_m)$ of length $m = 2^r$ is called a u-cycle if, after dividing it in the middle, the leftmost entries on the two sides are adjacent in a subhalf of H_r, say H_s, after dividing each side in the middle, the leftmost entries on the two sides are adjacent in a subhalf of H_s, etc. □

For example,



are both u-cycles of length 8 with the adjacencies as indicated. Hence, 0 and 4 are adjacent, 0 and 2 are adjacent, 4 and 6 are adjacent, etc. Notice that the entries need not be consecutive numbers; they must just be adjacent as marked by the overbars.

Lemma 3.1:

1) Each half of a u-cycle is also a u-cycle.
2) The concatenation of two disjoint u-cycles $(x_1x_2\cdots x_m)$ and $(y_1y_2\cdots y_m)$, that is, $(x_1x_2\cdots x_my_1y_2\cdots y_m)$, is also a u-cycle if a) x_1 and y_1 are adjacent, and b) $|x_1 - x_{m/2+1}|$, $|y_1 - y_{m/2+1}| < |x_1 - y_1|$.

Proof: It is straightforward and omitted. □

Theorem 3: An r-stage cube-connected network, where $N = 2^r$, can realize any u-cycle of length 2^k in one pass for all $k \leq r$.

Proof: We proceed by induction on r, and without loss of generality, let $k = r$. The proof of the case $k < r$ follows directly from the case $k = r$. Now, for $r = 1$, the only cycle, i.e., (0 1), is a u-cycle, and it is realizable by a 1-stage cube-connected network which is just a 2×2 cell. Assume all u-cycles of length 2^{r-1} are realizable by an $(r-1)$-stage cube-connected network in one pass. Consider a u-cycle of length 2^r, $(x_1x_2\cdots x_{2^{r-1}}x_{2^{r-1}+1}x_{2^{r-1}+2}\cdots x_{2^r})$. Factor this cycle as $(x_1x_2\cdots x_{2^{r-1}})(x_{2^{r-1}+1}x_{2^{r-1}+2}\cdots x_{2^r})(x_1x_{2^{r-1}+1})$. Now, by the first part of Lemma 3, the first two subcycles are both u-cycles of length 2^{r-1}, and hence by induction, each can be realized by an $(r-1)$-stage cube-connected network in one pass. Moreover, since they are disjoint, we can realize both of them by the first $r-1$ stages of an r-stage cube-connected network in one pass by assigning one to the upper $(r-1)$-stage subnetwork, and the other to the lower $(r-1)$-stage network. Finally, since the original cycle is a u-cycle, x_1 and $x_{2^{r-1}+1}$ are adjacent in H_r, and hence $(x_1x_{2^{r-1}+1})$ can be realized by the last stage of the r-stage cube-connected network. Therefore, the entire expression can be realized by an r-stage cube-connected network in one pass and the assertion follows. □

The following particular form of u-cycles will prove to be essential for realizing transpositions in two passes by cube-connected networks.

Definition 4: A u-cycle whose inverse is also a u-cycle is called a w-cycle. □

For example, (0 1 2 3 4 5 6 7) is a w-cycle as its inverse (7 6 5 4 3 2 1 0) is also a u-cycle. On the other hand, since 13 and 3 are not adjacent, (0 1 2 3 8 9 12 13) is not a w-cycle, i.e., its inverse (13 12 9 8 3 2 1 0) is not a u-cycle.

As an immediate corollary to Theorem 3, we have:

Corollary 3.1: An r-stage cube-connected network can realize any w-cycle of length 2^k in one pass for all $k \leq r$. □

Furthermore, we have:

Lemma 4.1: The concatenation of two w-cycles $(x_1x_2\cdots x_m)$ and $(y_1y_2\cdots y_m)$, i.e., $(x_1x_2\cdots x_my_1y_2\cdots y_m)$, is also a w-cycle if 1) x_1 and y_1 are adjacent, 2) x_m and y_m are adjacent, 3) $|x_1 - x_{m/2+1}|, |y_1 - y_{m/2+1}| < |x_1 - y_1|$, and 4) $|x_m - x_{m/2+1}|, |y_m - y_{m/2+1}| < |x_m - y_m|$. □

Theorem 4: Let x and y be any two symbols in H_r. It is always possible to form a w-cycle starting with x and ending with y.

Proof: We proceed by induction on r. The assertion is, clearly, valid for $r = 1$. Suppose that it is also valid for $r-1$, i.e., it is possible to form a w-cycle between any two symbols in H_{r-1}. To prove the assertion, we consider two cases. The first case is when x and y appear in the same half of H_r. If that half is $\mathcal{H}_0$ then the assertion is valid by the induction hypothesis. If it is $\mathcal{H}_1$, then let x' and y', respectively, be the duals of x and y in $\mathcal{H}_0$. Now, by hypothesis, there exists a w-cycle which begins with x' and ends with y'. To form a w-cycle between x and y, translate this w-cycle by adding 2^{r-1} to all of its entries. The second case is when x and y appear in different halves of H_r, say x in $\mathcal{H}_0$ and y in $\mathcal{H}_1$. To construct a w-cycle from x to y, let x' and y', respectively, be the duals of x in $\mathcal{H}_1$ and y in $\mathcal{H}_0$. Now, by induction hypothesis, there exists a w-cycle in $\mathcal{H}_0$ which begins with x and ends with y', and a w-cycle in $\mathcal{H}_1$ which begins with x' and ends with y. Moreover, x and x', and y and y', are adjacent in H_r. Finally, $|x-\alpha|, |y-\beta| < |x-x'|, |y-y'|$ for all $\alpha \in \mathcal{H}_0$, and $\beta \in \mathcal{H}_1$. Hence, the hypothesis of Lemma 4.1 is satisfied, and the cycle obtained by concatenating the two w-cycles in $\mathcal{H}_0$ and $\mathcal{H}_1$ is also a w-cycle. □

The following procedure forms a w-cycle starting with x and ending with y.

```
Procedure CYCLE(x, y, r; var w);
If r = 1
  then w := x, y
  else
    begin
    If HALF(x, r) = HALF(y, r)
      then w := CYCLE(x, y, r − 1)
      else w := CYCLE(x, DUAL(y, r), r − 1),
                CYCLE(DUAL(x, r), y, r − 1)
    end;
endprocedure;
```

In the worst case, each call to CYCLE results in at most two calls to itself until the index r of H_r reduces to 1. Since after each call, r reduces by one, in the worst case, a total of $2^r - 1$, or $N - 1$ calls will be made. Each call can be processed in $O(1)$ time, and hence CYCLE takes $O(N)$ steps on a single processor. The optimality of the number of steps follows from the fact that one may have a cycle of up to N symbols, and hence, in the worst case, must read the images of up to N symbols to set up a cycle. If a separate processor is used to execute the call to CYCLE for each value of r, then the overall time becomes $O(r) = O(\log_2 N)$ using N processors. As for the space complexity, we need $O(N)$ space to stack the $O(N)$ calls.

Now the main result of this section follows.

Theorem 5: A cube-connected network can realize the transposition of any two of its inputs in at most two passes.

Proof: By induction on r, we will prove that any transposition $(x\,y)$ can be realized in two passes. If $r = 1$ then the assertion is valid, since there is only one transposition, and it is realizable by 1-stage cube-connected network which is a 2×2 cell. Suppose the assertion is true for $r - 1$, i.e., it is always possible to realize a transposition of any two symbols in H_{r-1} by an $(r-1)$-stage cube-connected network in at most two passes. To prove the assertion, we consider two cases. The first case is when x and y appear in the same half of H_r. If that half is $\mathcal{H}_0$, then the assertion is valid by induction hypothesis. If it is $\mathcal{H}_1$, then, the transposition of x and y can be induced directly from the transposition of their duals in $\mathcal{H}_0$. The second case is when x and y appear in different halves of H_r, say x in $\mathcal{H}_0$ and y in $\mathcal{H}_1$. To prove the assertion, we observe that $(x\,y)$ can be factored as $(y' x_1 x_2 \cdots x_m x)(y'y)(y' x_1 x_2 \cdots x_m x)^{-1}$ where $(y' x_1 x_2 \cdots x_m x)$ is a w-cycle and y' is the dual of y in $\mathcal{H}_0$. Now, by Corollary 3.1, both the first and last cycles in this factorization are realizable in one pass by an r-stage cube network. Moreover, since y and y' are adjacent in H_r, and form the inputs of a cell in the rth stage of an r-stage network, the product $(y' x_1 x_2 \cdots x_m x)(y'y)$ is realizable by that network in one pass. Thus, the entire product, i.e., $(y' x_1 x_2 \cdots x_m x)(y'y)(y' x_1 x_2 \cdots x_m x)^{-1}$ is realizable in two passes, and hence the assertion. □

Thus, the transposition of any two symbols, x and y, in H_r can always be factored into a product of two w-cycles, and another transposition which is realizable in at most two passes. Once the dual of y is determined, the w-cycle from x to the dual of y can be obtained by using Procedure CYCLE. This w-cycle and its inverse, together with the transposition (yy') can then be used to realize $(x\,y)$ in at most two passes. Note that the time and space complexities of this procedure will be the same as those for CYCLE.

Theorem 5 holds trivially if the switches in the network can be set to route their inputs to their outputs on a nonpermutation basis. That is, if either input can be routed to either output with the other input and output left floating, then $(x\,y)$ can be realized simply by routing x to y in the first pass, and y to x in the second pass. The constraint that each switch can only be set to either the identity or transposition map makes the realization of transpositions more involved.

V. Realization of Permutations

We now present a procedure to set up a cube-connected network for a arbitrary permutation which it can realize in one pass. First, recall that the half representation partitions the inputs of an r-stage cube-connected network into two disjoint sets $\mathcal{H}_0$ and $\mathcal{H}_1$. This partition induces the three-part network structure shown in Fig. 1, and leads to the following theorem.

Theorem 6: Let p be a permutation such that $p(x) = y$ for some two symbols x and y in H_r. If p is realizable by an r-stage cube-connected network in one pass then

a) If $x, y \in \mathcal{H}_0$, or $x, y \in \mathcal{H}_1$ then the last stage cell, one of whose outputs is y, must be set to the identity state.
b) If $x \in \mathcal{H}_0$ and $y \in \mathcal{H}_1$, or $y \in \mathcal{H}_0$ and $x \in \mathcal{H}_1$ then the last stage cell, one of whose outputs is y, must be set to the transposition state.

Proof: The proof is straightforward and omitted. □

Thus, for a given permutation, the states of the cells in the last stage of an r-stage cube-connected network can be determined without considering the states of the cells in the $(r-1)$-stage subnetworks. The state of each such cell is determined according to whether its outputs draw their inputs from the same or different subnetworks under the specified permutation. More precisely, the specified permutation cannot be realized if the outputs of any cell in the last stage draw their inputs from the same $(r-1)$-stage subnetwork. As described in the following procedure, this principle can be applied recursively to the last stages of each subnetwork until a subnetwork consists of a 2×2 cell. For convenience, we work with $q = p^{-1}$ in the procedure.

```
Procedure PERMUTATION(q, N; var y, realizable);
For y := 0 to N/2 − 1 do
  If q(y) ∈ H0 and q(y + N/2) ∈ H1
  then set cell S_{y,y+N/2} to identity
  else
    If q(y) ∈ H1 and q(y + (N/2)) ∈ H0
      then set cell S_{y,y+N/2} to transposition;
      else realizable := false
endfor;
If N > 2 and realizable
  then
    begin
    PERMUTATION(LOWER(q), N/2)
    If realizable then
    PERMUTATION(UPPER(q), N/2)
    else {skip}
    end;
endprocedure;
```

The variable *realizable* is initialized to *true* before the execution of PERMUTATION starts. The *for* loop determines the states of the cells in the last stage of the network subject to the condition that their outputs be drawn from different subnetworks. If there is no conflict, then the procedure calls itself to work on the upper subnetwork, and then the lower subnetwork provided that *realizable*

remains *true*. Functions LOWER(q) and UPPER(q) in the recursive calls, respectively, refer to the first and second halves of map q. The procedure repeats calling itself until either *realizable* becomes *false*, or $N = 2$. Consequently, it either sets the network for map p, or it signals that p is not realizable.

Each call to PERMUTATION results in two calls to itself, and after each call, the size of the network reduces by a factor of 2. Thus, PERMUTATION is called once with value N, twice with value $N/2$, four times with value $N/4$, and so on up to $N/2$ times with value 2. Moreover, when it is called with value N/i; $i = 1, 2, 4, \cdots, N/2$, it takes $O(N/2i)$ times to process the *for* loop. Summing these steps, the serial time complexity of this procedure becomes $O(N \log_2 N)$. This is optimal, since one needs at least $O(N \log_2 N)$ bits to code the set of one-pass permutations of an N-input cube-connected network. The space complexity of the procedure is determined by the stack space for recursive calls which is $O(N)$.

Furthermore, by using $N/2$ processors, the time needed to execute the *for* loop can be reduced to $O(1)$ per each call, and hence the execution of the overall procedure to $O(\log_2 N)$ steps. This just requires making *realizable* a shared variable, and executing the calls to PERMUTATION for the upper and lower subnetworks in parallel. As a result, the procedure halts after $O(\log_2 N)$ calls with each call completed in $O(1)$ time.

VI. Concluding Remarks

This paper presented optimal setup algorithms for cube-connected networks using recursive Karnaugh maps. All procedures have been shown to run in polynomial time and space. Procedure PERMUTATION is the most general of all the procedures which are presented in the paper, and can be used to determine whether a transposition, a cycle, or a permutation is realizable in one pass by a cube-connected network. It also specifies the states of the cells for the network in question whenever the network can realize the given map. On the other hand, procedure CYCLE requires fewer steps, and can be used to realize w-cycles and arbitrary transpositions including those which require two passes.

These procedures can easily be generalized to other $O(\log_2 N)$ stage networks as well as to multipass cube-connected networks. It may also be worthwhile to extend them to delta and generalized interconnection networks [10], [1], since such networks share the *unique path* property of cube-connected networks.

References

[1] L. N. Bhuyan and D. P. Agrawal, "Design and performance of generalized interconnection networks," *IEEE Trans. Comput.*, vol. C-32, no. 12, pp. 1081–1090, Dec. 1983.

[2] ——, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Trans. Comput.*, vol. C-33, no. 4, pp. 323–333, Apr. 1983.

[3] T. Feng, "A survey of interconnection networks," *IEEE Comput. Mag.*, vol. 14, no. 12, pp. 12–17, Dec. 1981.

[4] J. Hayes *et al.*, "A microprocessor-based hypercube supercomputer," *IEEE Micro*, vol. 6, no. 5, pp. 6–17, Oct. 1986.

[5] S.-T. Huang and S. K. Tripathi, "Finite state model and compatibility theory: New analysis tools for permutation networks," *IEEE Trans. Comput.*, vol. C-35, no. 7, pp. 591–601, July 1986.

[6] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *AIEE Trans.*, vol. 72, no. 9, pp. 593–599, Sept. 1953.

[7] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, no. 12, pp. 1145–1155, Dec. 1975.

[8] K. Y. Lee, "On the rearrangeability of $2(\log_2 N) - 1$ stage permutation networks," *IEEE Trans. Comput.*, vol. C-34, no. 5, pp. 412–425, May 1985.

[9] A. Y. Oruç and M. Mittal, "New algorithms for realizing paths and permutations through cube-connected networks," in *Proc. ACM Comput. Sci. Conf.*, Cincinnati, OH, Feb. 1986, pp. 137–146.

[10] J. H. Patel, "Performance of processor–memory interconnections for multiprocessors," *IEEE Trans. Comput.*, vol. C-30, no. 10, pp. 771–780, Oct. 1981.

[11] M. C. Pease, "The indirect binary N-cube multiprocessor array," *IEEE Trans. Comput.*, vol. C-26, no. 5, pp. 458–473, May 1977.

[12] C. L. Seitz, "The cosmic cube," *Commun. ACM*, vol. 28, pp. 22–33, Jan. 1985.

[13] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* Lexington, MA: Lexington Books, 1985.

[14] D. Steinberg, "Invariant properties of the shuffle-exchange and simplified cost-effective version of the omega network," *IEEE Trans. Comput.*, vol. C-32, no. 5, pp. 444–450, May 1983.

[15] C.-L. Wu and T. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Comput.*, vol. C-29, no. 8, pp. 694–702, Aug. 1980.

[16] ——, "The reverse-exchange interconnection network," *IEEE Trans. Comput.*, vol. C-29, no. 9, pp. 801–810, Sept. 1980.

A Parallel Time/Hardware Tradeoff $T \cdot H = O(2^{n/2})$ for the Knapsack Problem

Afonso G. Ferreira

***Abstract*— In this paper, we propose a parallel algorithm that solves a knapsack problem of size n in time $T = O(n * (2^{n/2})^{\varepsilon})$ when $P = O((2^{n/2})^{(1-\varepsilon)})$, $0 \leq \varepsilon \leq 1$, processors are available. The algorithm needs $S = O(2^{n/2})$ memory space in a shared memory. Let H (for hardware) be the number of processors plus the number of memory cells used by a parallel algorithm. The parallel algorithm that we describe here takes a linear time proportional to $(n/2)$ to find a solution when $P = O(2^{n/2})$, leading to a tradeoff $T \cdot H = O(2^{n/2})$.**

***Index Terms*— Knapsack problem, NP-complete, parallel algorithms, time/processor/memory tradeoff.**

I. Introduction

Given n positive integers $W = (w_1, w_2, \cdots, w_n)$ and a positive integer M, the knapsack problem is the decision problem of finding a set $I \subseteq [1, n]$, such that $\sum w_i = M$, for $i \in I$. In other words, one searches for a binary n-tuple $C^* = (x_1, x_2, \cdots, x_n)$ that solves the equation $\sum w_i x_i = M$. This problem was proved to be NP-complete [10] and, unless NP = P, its complexity is exponential in n. The original search space has 2^n possible values. An exhaustive search would then take $O(2^n)$ time to find a solution in the worst case.

Solving the knapsack problem can be seen as a way to study some large problems in number theory and, because of its exponential complexity, some public-key cryptosystems are based on it [16]. Therefore, much effort has been done in order to find techniques which could lead to practical algorithms with reasonable running times.

In the literature, because of the exponential complexity of the problem, factors which are single degree polynomials in the size of the problem are usually neglected in the O notation. Notice that the size of a knapsack problem can be defined either as the cardinality

Manuscript received February 26, 1988; revised September 25, 1989. The author was on leave from the University of Sao Paulo, Brazil, and was supported in part by a CAPES/COFECUB fellowship, Grant 503/86-9.

The author is with Laboratoire de l'Informatique du Parallélisme—IMAG, Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07 France.

IEEE Log Number 9037991.

0018–9340/91/0200–0221$01.00 © 1991 IEEE