In Proceedings of the Fifth Distributed Memory Computing Conference, pp. 800-808, IEEE CS Press, 1990

Optimal Self–Routing of Linear-Complement Permutations in Hypercubes[1]

Rajendra Boppana and C. S. Raghavendra
Dept. of Electrical Engineering–Systems
University of Southern California, Los Angeles, CA 90089–0781

Abstract

In this paper we describe an algorithm to route the class of linear-complement permutations on Hypercube SIMD computers. The class of linear-complement permutations are extremely useful in devising storage schemes for parallel array access. The proposed algorithm is self-routing and minimal, that is, the path established by the algorithm between each pair of source and destination processors is via a minimal path using only the destination processor address. Furthermore, the algorithm requires only the optimal number of routing steps to realize any linear-complement permutation. The best known previous routing algorithms for the Hypercubes are for the class of bit-permute-complement permutations, a subset of the class of linear-complement permutations. Those algorithms are either non-optimal or not self-routing. The algorithm presented is self-routing, optimal, and it routes a larger class of permutations. Also, this algorithm can route the class of linear-complement permutations in multi-dimensional meshes in optimal number of routing steps.

Key words: hypercube, interconnection network, linear permutations, minimal routing, self-routing.

1 Introduction

A parallel computer consists of a large number of processors and an interconnection network to exchange information among them. For parallel computers, efficient schemes to move data among the processors are necessary to obtain fast and efficient parallel algorithms. The problem of moving data among processors is called the 'data routing problem'. Nassimi and Sahni [9] have developed communication schemes for the general case of the data routing problem; however, these schemes take too much time. Efficient and even optimal (in terms of number of routing steps) schemes can be developed when the data movement is regular and systematic, for example, one-one communication in which each processor sends data to a processor and each processor receives data from a processor. This can be effectively modeled as a permutation from the set of processors to itself. Important problems like FFT, matrix transposition, polynomial evaluation, etc. can effectively be solved in parallel computers which have an interconnection network to support the permutation type of communication. In such cases, the problem of moving data between pairs of source and destination processors can be treated as the problem of realizing the corresponding permutation using the interconnection network among the processors.

In this paper, we are interested in developing efficient schemes to realize some important classes of permutations on SIMD[2] multicomputers with static interconnection networks, specifically the hypercube computer [15, 16]. In a static (also called, direct) interconnection network, each processor is directly connected via physical communication links to other processors which are termed adjacent processors. Communication between non-adjacent processors requires routing data through intermediary processors.

In a hypercube computer, there are $N = 2^n$, $n > 0$, processors. Each processor is given a unique index (also called, address) from the set $\{0, 1, \ldots, N-1\}$. Processor i (that is, processor with index i) is connected to processor j, if and only if the binary representations of the indices i and j differ in exactly one bit; hence, each processor has n adjacent processors (neighbors). A hypercube of 8 nodes is shown in figure 1.

[2]The acronym SIMD stands for "Single Instruction stream and Multiple Data streams". See Flynn [3] for more details.

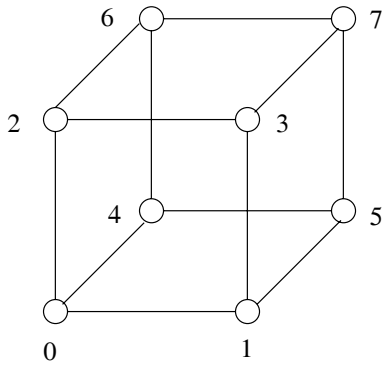


Figure 1: An 8 processor Hypercube.

[1]This research is supported by the NSF grant No. MIP 8452003, a grant from AT&T, DARPA/ARO Contract No. DAAG29–84–K–0066, ONR Contract No. N00014–86–K–0602.

An efficient method to realize arbitrary permutations in hypercubes is to use Stone's adaptation of Batcher's sorting technique [1, 17]; this takes $O(\log^2 N)$ time. An important feature of this technique is, at any time during routing, there are no more than two messages in a processor, messages can be routed using only their destination addresses, and each processor uses only the destination addresses of the messages, it has, to make routing decisions. This routing technique is termed as self-routing; the important advantage of self-routing algorithms is, control overheads are greatly minimized. Thus it is desirable to develop self-routing schemes that have smaller delay. Another important consideration in routing is to send each message via a shortest path. Any routing scheme that achieves this is called a minimal routing scheme.

The previous known work on fast realization of permutations consists of adapting the self-routing algorithms developed for multistage interconnection networks such as the Beneš network to the hypercube [8]. However this routing is not minimal, since data paths will be of length $2 \log N - 1$, in the worst case. (The length of a data path in a minimal routing is the hamming distance of the source and destination processors' addresses, which is, at most, n.) Also, Nassimi and Sahni [10] developed a non-self-routing algorithm for minimal routing of bit-permute-complement permutations in hypercubes. Valiant and Brebner [18] gave a randomized algorithm to route arbitrary permutations with distributed control. Their algorithm typically guarantees that routing will be completed in $O(\log N)$ time with very high probability; however, it is noteworthy that with a small probability their algorithm might take up to $O(\sqrt{N})$ time or lead to deadlock. In this paper, we are interested in obtaining routing algorithms that are deterministic with respect to completion time.

In this paper, we present a minimal self-routing algorithm to route the class of linear-complement permutations. In a linear-complement permutation, destination-processor-address bits are linear combinations of the source-processor-address bits. An important application of linear-complement permutations is in the skewed matrix storage schemes [4, 12]. The set of bit-permute-complement permutations is contained in the set of linear-complement permutations. The algorithm presented is the first such effort to self-route some of the most frequently used permutations. Furthermore, the algorithm requires only n routing steps, each step requiring a constant amount of time to process.

In the next section, we give the algorithm and discuss its working. Also, we prove that the algorithm does minimal routing of linear-complement permutations. Later, we use the minimal routing property of the algorithm to realize linear-complement permutations for circuit switched hypercube. We show that the algorithm developed for the hypercube can be used to realize linear-complement permutations on multi-dimensional meshes. We also show that the algorithm realizes many more permutations other than linear-complement permutations.

2 A self-routing algorithm

In a hypercube ($\mathcal{Q}_n$) of $N = 2^n$ processors, there are n dimensions: $0, 1, \ldots, n-1$. If i is the index of a processor, then $i = (i_{n-1}, \ldots, i_0)$, an n-bit vector, such that $i = \Sigma_{p=0}^{n-1} 2^p i_p$. If two processors i and j are connected in dimension p, then the binary representations of i and j differ in bit p only.

2.1 Model of execution

We assume that each communication link can operate in full duplex mode and one word of data can be sent at a time. Each processor has a processing unit capable of computing simple arithmetic functions and make simple logic decisions, constant number of registers, and some local memory. Data movement is done by message passing; that is, data to be moved between any pair of processors are formatted into a message with header consisting of the destination processor's address; this message is routed through intermediary processors (if necessary) to the destination processor. Furthermore, a processor can receive at most one message at a time from any of its n neighbors, and send at most one message at a time to any of its neighbors. This model of execution accurately reflects some of the existing hypercube computers.

2.2 Preliminaries

In what follows, we define the class of linear-complement permutations ($\mathcal{LC}_n$) of $N = 2^n$ numbers. Let $V = \{0, 1, \ldots, N-1\}$. Also, let x be any number and y be its image under some mapping. Exclusive-or operation '$\oplus$' is used for boolean addition.

Definition 1 *A permutation on V is said to be a linear permutation* [11]*, if there exists a non singular binary matrix $Q_{n\times n}$ such that, for every $x \in V$, its image is given by the equation, $y^T = Qx^T$.*

So, a linear permutation on V is the permutation that maps each $x \in V$ to a number such that each bit in the binary form of this number is a linear combination of the bits of x. If complement of bits is allowed,

then the permutation is a linear-complement permutation.

Definition 2 *Let* $x' = (x_{n-1} \ldots x_0 1)$. *A permutation on* V *is a linear-complement permutation (*$\mathcal{LC}$*) if there exists a binary matrix* $P = (Q \mid k)$, *where* Q *is as defined above and* k *is an* n*-bit binary vector, such that, for every* $x \in V$, *its image is given by the equation,*

$$y^T = P(x')^T \tag{1}$$

A bit-permute-complement permutation is a linear-complement permutation with a permutation matrix (each row and column of the matrix has exactly one 1) as Q. A linear-complement permutation with matrix $P = (Q \mid k)$, Q an $n \times n$ boolean matrix and k some n-bit vector, has the same properties as the linear permutation corresponding to Q has.

In the literature [2, 6, 14], the linear permutations are termed as the non-singular linear transformations of the n-dimensional vector space over the field $GF(2)$ — the field consisting of two elements: 0, and 1. (There are exactly 2^n elements in this vector space, and each element corresponds to a processor index in binary form.) There is one-one correspondence between the boolean matrices of size $n \times n$ and linear transformations on n-dimensional boolean vector space [6]. A linear transformation is invertible, if and only if the corresponding boolean matrix is invertible. A linear transformation can also be viewed as a homomorphism on the group underlying the vector space.

In the context of a hypercube, a permutation is called a linear-complement permutation, if there exists a boolean matrix P satisfying equation (1) for every pair of source and destination processor indices. Hence, if a given permutation is a linear-complement permutation, each bit of the destination processor index is described by a linear combination of the bits of the source processor index, and a constant term that could be either 0 or 1.

We use the word *tag* to indicate the packet containing the destination processor index and the message. For the purpose of the routing algorithm the message is unimportant. Hence, we assume that the tags contain only the the destination processor index, an n-bit vector. If processor x has tag y, then it is called the host processor of the tag y. The goal of the algorithm is to route tags, so that, at the end of the routing process, each tag matches with the address of its host processor.

During routing, the number of tags in a processor may vary. However, the algorithm assures that there are only two cases: one tag per processor, and two tags per processor for $N/2$ processors and no tags for the remaining processors. We say that the hypercube is in **state A**, when each processor has one tag. Before routing the hypercube is in state A. When there are 2 tags for each of $N/2$ processors and no tags for the remaining $N/2$ processors, then the hypercube is said to be in **state B**. At any time during the routing process, the hypercube is, as proved later, in one of the two states.

During routing, the effect of sending a tag from a processor to another processor with index differing in bit position i is succinctly stated that the tag is routed along the dimension i. We use the expression 'a tag is routed to correct bit i' to mean that the processor having that tag routes the same along the dimension i, if the tag's bit i differs from that of processors index in binary form. Note that it makes sense to say that a tag is routed to correct bit i even when bit i of the tag and that of the processor address match; however, in this case the tag remains in the processor itself during that routing step. In each routing step, all the tags are examined and all of them are routed to correct a particular bit, say i; an alternate way of saying this is 'bit i is corrected'. After a bit is corrected, all the tags are in the correct processors with respect to that bit.

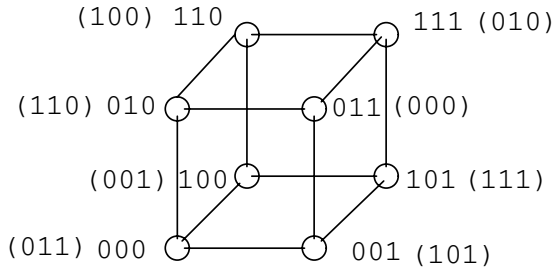
2.3 Statement and discussion of the algorithm

Algorithm **HL**:
If the hypercube $\mathcal{Q}_n$ is in state A, each processor routes its tag to correct the least significant bit that has not been used in earlier routing steps. If the hypercube is in state B, processors with no tags do nothing, and processors with two tags compare the two tags and route one of them to correct the least significant bit in which they differ. This is repeated n times. ∎

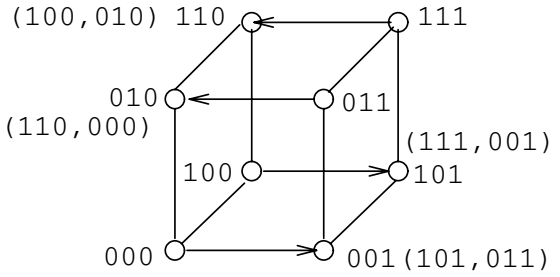
An example showing the routing of a linear-complement permutation in $\mathcal{Q}_3$ is shown in figure 2; the following set of linear equations specify the linear-complement permutation used.

$$\begin{aligned} y_2 &= x_1 \oplus x_0, \\ y_1 &= x_2 \oplus x_0 \oplus 1, \\ y_0 &= x_1 \oplus 1. \end{aligned}$$

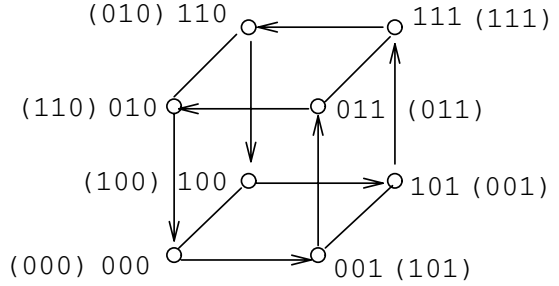
The index of each processor and its initial tag (in parenthesis), in binary form, are shown in figure 2(a); this indicates the allocation of tags to processors in the hypercube. The effect of correcting bit 0 in the first routing step is shown in figure 2(b). Here processors with indices $000, 011, 100$, and 111 send their tags the processors adjacent to them in dimension 0. It can be seen that the tags in these processors differ from their respective host processor indices in the least significant



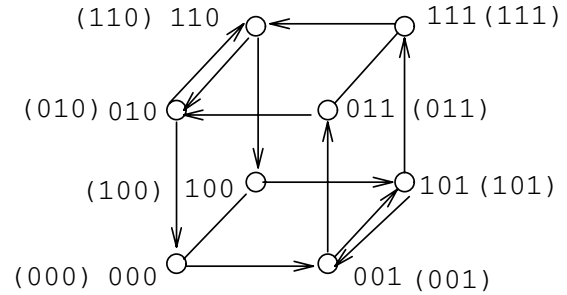
(a) Before routing



(b) After correcting bit 1



(c) After correcting bit 2



(d) After correcting bit 3

Figure 2: Routing a linear-complement permutation in $\mathcal{Q}_3$.

bit. The paths traced by the tags are shown by arrows; and the tags in a processor (if present) are given in parenthesis.

After the first routing step, half the processors of $\mathcal{Q}_3$ have two tags each, and the remaining processors have none. From figure 2(b), we can see that in each of the four processors with two tags, the tags differ in both bit positions. Each of these four processors picks to correct bit 1, as specified by the algorithm, independently. Figure 2(c) shows the effect of correcting bit 1 in the second routing step. The path traced by the tags that are routed are shown by arrows.

After correcting bits 0 and 1, each processor again has one tag. So, each processor picks to correct bit 2, as required by the algorithm, independently. In this routing step, processors 001 and 010 exchange tags with processors 101 and 110 respectively. This is shown by two parallel lines, pointing in opposite directions arrow heads, between the exchanging processors to indicate the path traced by each tag. After the third routing step, all the tags are at their correct destinations.

2.4 Proof of correctness

We use the well known concepts in linear algebra to prove the correctness of the algorithm. When the hypercube is in state B, half the processors have two tags each and the remaining processors have none; so, the relation from the set of processors to the set of tags is not a mapping. To avoid this, we use the mapping from tags to processors so that the results of linear algebra are applicable regardless of the state of the hypercube.

Let $y = (y_{n-1}, \ldots, y_0)$ represent a tag and $x = (x_{n-1}, \ldots, x_0)$ represent its host processor address. Also, let the tags be distributed among processors according to some affine linear transform f; that is, the mapping from tags to processors is of the form: $f : y \rightarrow x,\ x = f(y) = T(y) \oplus k$, where, T is some linear transformation and k an n-bit vector. In other words, for each tag, the address of its host processor can be specified by the following bit equations.

$$\left.\begin{array}{rcl} x_{n-1} &=& \lambda_{0,0} y_{n-1} \oplus \cdots \oplus \lambda_{0,n-1} y_0 \oplus k_{n-1} \\ &\vdots& \\ x_p &=& \lambda_{n-1-p,0} y_{n-1} \oplus \cdots \\ && \oplus \lambda_{n-1-p,n-1} y_0 \oplus k_p \\ &\vdots& \\ x_0 &=& \lambda_{n-1,0} y_{n-1} \oplus \cdots \oplus \lambda_{n-1,n-1} y_0 \oplus k_0 \end{array}\right\} \quad (2)$$

Here, the $n \times n$ matrix $(\lambda_{i,j})$, $\lambda_{ij} \in \{0, 1\}$, represents the transformation matrix of the linear transform T. It is clear that if the matrix $(\lambda_{i,j})$ is non-singular, the

linear transform T is invertible, and T is a linear permutation. A non-zero vector for k indicates translation of the linear transform, and it does not affect the invertibility of T. The affine linear transform f is denoted by T_k; note that $T = T_0$.

Now, suppose that the tags are moved among the processors using the following rule: "Each processor moves the tags that do not agree with its index bit p to its neighbor processor in the dimension p." Then, we claim the following.

Claim 1 *When tags are moved as specified above, the mapping between tags and their host processor addresses is still an affine linear transform.*

Proof: After the routing step, each tag agrees with its host processor in bit p. Hence, for all the tags the equation for bit p is, simply, $x_p = y_p$.

Now, if a tag is moved during the routing step, then it is moved to a new host processor that differs from the old host processor only in the bit p. Hence, irrespective of whether the tag is moved or not, the other bit equations are unchanged. ∎

If the mapping of tags to processors is an affine linear transform of the form, $x = T(y) \oplus k$, Then, $x \oplus k = T(y)$ is a linear transform. That is, if each processor address is translated by k, then the tag distribution is simply a linear transform T. Now, ker $T = \{y \mid T(y) = 0\}$, that is, ker T is the set of all tags assigned to processor with address 0, under the linear transform T.

From the property of linear transforms, $T(0) = 0$; so, $|\ \ker T\ | \geq 1$. Also, T can be treated as a homomorphism from the group underlying the vector space of processor indices to itself. From the first isomorphism theorem [14], we get that each processor with at least one tag will have the same number of tags that processor 0 has under the linear transform T. Therefore, we have the following lemma.

Lemma 2 *If the tags are distributed among the processors such that some processors have one tag, some other processors have two tags, while the remaining processors have none, then the mapping between tags and processor addresses is not an affine linear transform.* ∎

Lemma 3 *Suppose the tags are distributed among processors, according to some affine linear transform T_k, such that half of the processors have two tags each and the remaining processors have no tags. Then, the two tags in a processor, if present, differ in the same bit positions.*

Proof: The linear transform T represents the mapping between tags and processors after translating the processors addresses by k. Under T, processor 0 has two tags, namely, 0 and a, for some $a \neq 0$. Now, take any processor x that has two tags b and c. To prove the lemma, it is sufficient to show that $b \oplus c = a$, which is true in view of the following.

$$
\begin{aligned}
& T(0) = T(a) = 0 \\
& T(b) = T(c) = x \\
\Rightarrow\ & T(b \oplus c) = T(b) \oplus T(c) = x \oplus x = 0 \\
\Rightarrow\ & (b \oplus c) = 0, \text{ or } a \\
\Rightarrow\ & b \oplus c = a, \text{ since } b \neq c
\end{aligned}
$$
∎

Lemma 4 *The algorithm HL routes tags such that the following are always true: in any routing step, (a) any processor moves at most one tag, and, after each routing step, (b) the tag distribution is given by some affine linear transformation, and (c) the hypercube is in either state A or state B.*

Proof: We prove this by induction on the number of routing steps.

Whenever the hypercube is in state A, and the tag distribution is according to some linear-complement permutation, the tag movement specified by the algorithm HL and the rule used in the claim 1 is the same. So, (a),(b) are true. Since, the tag distribution should be according to some affine linear transform, by lemma 2, either each processor has one tag (state A), or half the processors have two tags (state B). So (c) is also true after the routing step.

Since the hypercube is state A before routing the linear-complement permutation under question, the lemma holds for the routing step 1.

Now, let us assume that the lemma is true for the first $m \geq 1$ routing steps. We need to show that the lemma holds after the routing step $(m + 1)$.

After routing step m, if the hypercube is in state A, then, by the above argument, the condition (a) is true during the routing step $(m+1)$ and the conditions (b) and (c) are true after the routing step $(m + 1)$.

However, if the hypercube is in state B, then lemma 3, tells that the two tags in a processor (if present) differ in the same bits. But, in that case, using the algorithm HL, each processor with two tags chooses one dimension unambiguously. Hence, for each processor with two tags, exactly one tag matches with its index in the bit chosen for correction in the next routing step. So, in the routing step $(m+1)$, each processor with two tags routes exactly one tag, so that the tags that differed from the host processor index in the routing bit will now match with the new host processor index. This shows that (a) is true for the routing step $(m + 1)$. From the above argument and claim 1, (b) is true after the routing step $(m + 1)$. Since, a processor with two tags moves one tag in the $(m + 1)$ routing step, after the routing step, it has 1 or 2 tags. Using lemma 2 and the fact that (b) holds after the routing

step $(m+1)$, we conclude either each processor has one tag or half the processors have two tags each and the rest have none; therefore, (c) also is true after the routing step $(m+1)$. ∎

Corollary 1 *Any routing step given by the algorithm HL, and the rule given for the claim 1 are equivalent, provided each time the bit chosen for the rule is same as the routing bit chosen by the algorithm.*

Proof: This follows directly from the above lemma. ∎

Corollary 2 *The algorithm HL routes tags such that each bit is chosen for routing exactly once. And, after correcting a bit, each tag matches with its host processor index in that bit.*

proof: The above corollary says that when a bit is used in a routing step, all the tags match with the host processor index, after completing the routing step. The fact that each bit is chosen as routing bit exactly once, is easy to see. ∎

Theorem 1 *The algorithm, HL, routes any linear-complement permutation in a hypercube, $\mathcal{Q}_n$, in n routing steps. Furthermore, each processor needs to route at most one tag in a routing step.*

Proof: The proof follows from the lemma 4, and the corollaries following the lemma. ∎

3 Scope and use of the algorithm HL

In this section, we describe other aspects of the algorithm HL in routing permutations in various types of hypercubes. First, we describe how to use this algorithm to route linear-complement permutations in multi-dimensional meshes in optimal number of steps. Next, we describe how the algorithm HL can be used to route messages in a circuit switched mode of transmission. Then, we characterize a larger class of permutations that are routed by the algorithm HL in hypercubes with the constraint of choosing of same dimension links by all processors, in a routing step, is relaxed.

3.1 Routing linear-complement permutations in a q-mesh

The algorithm HL can be used for routing the linear-complement permutations in multi-dimensional mesh connected computers by direct simulation of the hypercube connections.

In a 2 dimensional mesh connected computer (2-mesh), the processors are arranged in a rectangular array of size $N_1 \times N_0$ $(= N)$. Each processor is denoted by a 2-tuple (x_1, x_0), which means that the processor in row x_1 and column x_0. Each processor is connected to the other processors in the neighbor rows and columns, if they exist. An 8×8 2-mesh has the same structure of a 64 processor ILLIAC-4 machine with wrap around connections missing. In a similar manner, a q-mesh can be defined. In a q-mesh, each processor is denoted by a q-tuple $(x_{q-1}, \ldots, x_0)$, $0 \leq x_a < N_a$, $0 \leq a < q$, and $N = \Pi_{a=0}^{q-1} N_a$. Processor $(x_{q-1}, \ldots, x_0)$ is connected the processors $(x_{q-1}, \cdots, x_a \pm 1, \cdots, x_0)$, $0 \leq a < q$, provided they exist. Processor $(x_{q-1}, \ldots, x_0)$ is given the index $\Sigma_{a=0}^{q-1} x_a (N_{a-1} \times \cdots \times N_0)$, $N_{-1} = 1$.

Let us consider a 2-mesh with $N = 2^n$, n even, processors. Also let the number of processors in a row or column is $2^{n/2}$. Dimension i, $0 \leq i < n$, connection of n-cube can be simulated on the 2-mesh, in 2^x routing steps, where $x = i$ or $i - n/2$ depending on $i < n/2$ or $\geq n/2$. Hence, the time required to route linear-complement permutations on a 2-mesh is:

$$2(2^0 + 2^1 + \cdots + 2^{\frac{n}{2}-1}) \;=\; 2(2^{\frac{n}{2}} - 1).$$

This is the optimal number of steps to route the class of linear-complement permutations in a square 2-mesh [7].

In general, we can show that for a q-mesh with $N_a = 2^{n_a}$, $0 \leq a < q$, such that $N = 2^n = \Pi_{a=0}^{q-1} N_a = 2^{\Sigma_{a=0}^{q-1} n_a}$ the algorithm routes the class of linear-complement permutations in $\Sigma_{a=0}^{q-1}(2^{n_a} - 1)$ steps, which is optimal.

Theorem 2 *The algorithm HL can route the class of linear-complement permutations in a multi-dimensional mesh in optimal number of steps.* ∎

3.2 Routing linear-complement permutations in a circuit switched hypercube

So far, we have assumed that a message is attached to the tag and moved with it to the destination. However, when the message is long, it is faster to send it using the circuit switching scheme. In this scheme, a physical path is established between each pair of source and destination processors, and then messages are transferred at high rates. At the end of transmission of the messages, the paths are released. To transfer messages in circuit switching mode in a hypercube, we will assume that each processor has necessary hardware to establish temporary physical paths between the input and output links used by the tags that passed through

the processor. Furthermore, in the circuit switching scheme, the link between any two adjacent processors can be used by each of these two processors at most once. Hence, the self-routing algorithms developed for the Beneš network [13, 8] are not useful to route linear-complement permutations in a circuit switched hypercube.

Lemma 5 *When the algorithm HL is used to realize linear-complement permutations in a hypercube, any two adjacent processors communicate at most once.*

Proof: Any two adjacent processors have their indices differ in only one bit, say p. So if at all they communicate, they do so only in the routing step to correct bit p. ∎

In the previous section, we have shown that a processor needs to move at most one tag in a routing step. Hence, it immediately follows from the above lemma that the link between any two adjacent processors is used by each of the two processors to route at most one tag. Thus the algorithm HL can be used to move data among processors by circuit switched scheme. To send messages by circuit switching scheme, first the tags (without messages) are routed using the algorithm HL. Then, each processor uses the path traced by its tag to send message to the destination processor. Since each processor need to send and receive at most one tag in a routing step, this algorithm can be implemented with very little overhead.

3.3 On routing a larger set of permutations

So far, a hypercube is assumed to be operating in SIMD mode, and it was proved that the algorithm HL can route linear-complement permutations. In the following discussion, we use a stronger mode of operation for hypercube, and show that the algorithm HL routes a larger set of permutations.

An n-cube can be partitioned into 2^k, $1 \leq k \leq n$, $(n-k)$-cubes. In a routing step, if all the processors in a subcube choose the same dimension for routing, but processors in different subcubes may choose different dimensions, then it is clear that each subcube is operating in SIMD mode but the n-cube consisting of these subcubes is not operating in SIMD mode. Such a mode of operation is called Multiple-SIMD or M-SIMD. In general, each subcube can again be partitioned and partitioning of one subcube may be different from the partitioning of another subcube, etc. For the following, we assume that a hypercube can operate in M-SIMD mode when partitioning of processors, as discussed above, is considered.

After the first routing step, for routing purposes, the $\mathcal{Q}_n$ can be viewed as two subcubes, $\mathcal{Q}_{n-1}$, with each subcube having 2^{n-1} tags. In the remaining steps of the algorithm, each subcube routes tags among its processors, and does not send any tag to the other subcube. This observation can be applied for the remaining routing steps too. This gives us the motivation to relax the constraint of SIMD mode of operation to route a larger class of permutations by the algorithm HL. For the following discussion, we assume that after each routing step, the subcubes may choose different dimensions for the next routing step. We call this mode of operation as Multiple-SIMD or M-SIMD mode.

Let the set $A = \{n-1, \ldots, 0\}$ be partitioned into two subsets $B = \{n-1, \ldots, n-k\}$ and $C = \{n-k-1, n-k-1, \ldots, 0\}$, where $1 \leq k \leq n$. B can be used to partition the set of numbers $\{0, 1, \ldots, 2^n - 1\}$ such that if i, j are in the same partition, then $i_x = j_x$, $\forall\, n-k \leq x < n$. This idea can be used to partition the processors in a hypercube such that there are 2^k partitions, 2^{n-k} processors in each subcube.

Suppose processors in a hypercube are partitioned as given above. Define a permutation π that permutes partitions of processors by some permutation in $\mathcal{LC}_k$, and processors in each partition by some permutation in $\mathcal{LC}_{n-k}$ (permutations of processors in different partitions can be different).

Lemma 6 *The algorithm HL routes the set of permutations as discussed above in a hypercube operating in M-SIMD mode with partitions as described above.*

Proof: In the first $n-k$ routing steps, in each subcube a linear-complement permutation is realized by the algorithm HL. This preserves the SIMD mode of operation for each subcube. Once the processors in each partition are permuted, the permutation of partitions is achieved as follows. Partitions are rearranged so that, bits $n-k-1, \ldots, 0$ are used in partitioning the hypercube. So each subcube will have 2^k processors and there are 2^{n-k} such subcubes. Now each subcube has to route a permutation in $\mathcal{LC}_k$, the permutation that is defined on the bits $n-1, \ldots, n-k$ of the n-cube, to complete the original routing task. Since this can be done by the algorithm HL, the lemma holds. ∎

This idea of partitioned permutations can be used recursively on each partition, and on the permutation of partitions itself. We call such permutations as partitioned linear-complement permutations ($\mathcal{PLC}$).

Definition 3 *For $n \in \{0, 1, 2\}$, $\mathcal{PLC}_n = \mathcal{LC}_n$. For $n \geq 3$, $\mathcal{PLC}_n$ is defined as follows. A permutation is in the set $\mathcal{PLC}_n$, if it permutes the partitions of processors by a permutation in $\mathcal{PLC}_k$, and processors in*

each partition are permuted by some permutation in the set $\mathcal{PLC}_{n-k}$, where $1 \leq k \leq n$ and the partitioning of processors is as given above.

Lemma 7 *The algorithm HL can route any $\mathcal{PLC}$ permutation in a hypercube operating in M-SIMD mode.*

Proof: By considering the partitioning of the hypercube to be the same as the partitioning of the $\mathcal{PLC}$ permutation being realized, it can be shown that proof for this lemma is a simple generalization of that of the previous lemma. ∎

3.3.1 The set $\mathcal{PLC}$

In what follows, we give an estimate on the size of the the set of partitioned linear-complement permutations.

A decomposable linear permutation [5] is a linear permutation whose matrix Q can be partitioned as follows.

$$Q = \left(\begin{array}{c|c} Q_1 & 0 \\ \hline 0 & Q_2 \end{array}\right)$$

Q_1 and Q_2 are square boolean matrices, and define linear permutations on smaller size set of numbers. Let us define a subset of $\mathcal{LC}_k$, that contains the permutations that can not be decomposed such that rows 0 and $k-1$ of the original Q matrix are in different partitions. With complement of bits considered, this set is denoted as $\mathcal{LC}_k{}'$.

We now give an alternate definition of partitioned linear-complement permutations. It can be shown that this definition is equivalent to the one given earlier.

Definition 4 *A permutation is in the set of partitioned linear-complement permutations, if there is a partition of most significant k bits, for some $1 \leq k \leq n$, such that partitions of the processors are permuted by some permutation in $\mathcal{LC}_k{}'$, and the processors in each partition are permuted by some permutation in $\mathcal{PLC}_{n-k}$.*

From this, we get

$$|\mathcal{PLC}_n| = \sum_{k=1}^{n} |\mathcal{LC}_k{}'| \cdot |\mathcal{PLC}_{n-k}|^{2^k} \qquad (3)$$

Since, $|\mathcal{LC}_k{}'| \leq |\mathcal{LC}_k|$, and $|\mathcal{LC}_k| = 2^k \times |\mathcal{LIN}_k|$, where $\mathcal{LIN}_k$ is the set of linear permutations, $|\mathcal{LIN}_k| = 2^{\frac{k(k-1)}{2}}(2^k-1)(2^{k-1}-1)\cdots(2-1)$. Hence, the upper bound given below can be used to approximate the size of $\mathcal{PLC}_n$.

$$|\mathcal{PLC}_n| \leq \sum_{k=1}^{n} |\mathcal{LC}_k| \cdot |\mathcal{PLC}_{n-k}|^{2^k}$$

Since, $\mathcal{PLC}_k \geq \mathcal{LC}_k \geq 2^{\frac{k^2}{2}}$, the size of the set $\mathcal{PLC}_n$ is at least $\Omega\left(N^{\sqrt{N}\log N/8}\right)$, which is the lower bound on the term with $k = n/2$, in the summation of the right side of the identity 3. Thus the size of the set grows exponentially with N.

The idea of operating a hypercube in M-SIMD mode gives a method to construct exponential number of rearrangeable Beneš-like networks. A link between two processors in the hypercube is assumed to be replaced by a switch such that tag routing between the two processors is simulated by the switch. Each routing step in the hypercube is equivalent to the operation of a stage of switches in a corresponding multistage interconnection network constructed as below. Consider routing a partitioned linear-complement permutation in a M-SIMD hypercube. In the first routing step, all processors in the cube use dimension 1 links. So, switches in the first stage have inputs lines with addresses that differ only in bit 1. In the next routing step, each subcube chooses dimension links depending on the partitioned linear-complement permutation being realized. For different partitioned linear-complement permutations, different dimensions are chosen in each subcube. The switches simulating the links used in the second routing step form the second stage of switches. The interconnection pattern between the first and second stage of switches is such that the top half of the switches will have input lines with even addresses, and the lower half of switches will have input lines with odd addresses, or vice versa. Furthermore, input lines to a switch will differ in only the bit that is the dimension of the link simulated by the switch. Now, top half of switches are considered to form a sub-network, and bottom half of switches are considered to form another sub-network. This is repeated recursively in each routing step. The next routing steps can be used in this manner to obtain the first n stages of a network. After n routing steps, the first n stages, of a Beneš-like multistage network, are formed. The complete network of $(2n-1)$ stages is such that stages $n+1, \ldots, 2n-1$ are mirror images of stages $n-1, \ldots, 1$ respectively. The Beneš network is obtained when dimensions $\{0, \ldots, n-1\}$ are chosen in consecutive routing steps in each subcube. For M-SIMD mode of operation, different partitioned linear-complement permutations give different Beneš like networks, which by construction are rearrangeable. The algorithm given in [13] with appropriate modifications can be used to route the class of linear-complement permutations in these networks.

4 Conclusions

In this paper, we have presented an algorithm to realize the class of linear-complement permutations in a hypercube. The algorithm is simple, self-routing, and

optimal. It routes any linear-complement permutation in $(\log N)$ routing steps, with each step requiring a constant time. In message passing scheme, time required for a routing step is proportional to the length of the message.

Since, this algorithm picks to correct bit 1 in routing step 1, and whenever the hypercube is in state A, the least significant bit that is not yet corrected is picked in the next routing step, this routes inverse omega (Ω^{-1}) permutations trivially. If the algorithm picks to correct the most significant bit, whenever there is a choice of bits that can be picked to correct in a routing step, then it is clear that it still routes linear-complement permutations and also the class of omega (Ω) permutations.

All the permutations of order 4 are routed by this algorithm, since they all are in $\mathcal{LC}_2$. It is shown that the class of permutations realizable by this algorithm on a hypercube M-SIMD computer is larger than the class of permutations realizable on a hypercube SIMD computer. When the hypercube is allowed to operate in the MIMD mode, an even larger class of permutations can be routed by the algorithm HL. For example, in an 8 processor hypercube $(\mathcal{Q}_3)$, any arbitrary permutation can be realized if the mode of operation is MIMD; the same is not true even if M-SIMD mode of operation is used. We note that the algorithm HL not only shows the rearrangeability of a circuit switched $\mathcal{Q}_3$, but also shows how to route arbitrary permutations.

The algorithm routes many permutations that are not in the linear-complement permutation class. An interesting and useful problem would be the characterization of the class of permutations realizable by the algorithm. Knowing this characterization, one can perform a pre-processing step to convert an arbitrary permutation to a permutation realizable by this algorithm.

References

[1] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[2] G. Birhkoff and S. MacLane. *A survey of modern algebra.* Macmillan, fourth edition, 1977.

[3] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec. 1966.

[4] J. M. Frailong, W. Jalby, and J. Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *Proc. 1985 Int. Conf. on Parallel Processing*, pages 276–283, 1985.

[5] I. N. Herstein. *Topics in Algebra.* John-Wiley and Sons, second edition, 1975.

[6] K. Hoffman and R. Kunze. *Linear Algebra.* Prentice-Hall, second edition, 1971.

[7] D. Nassimi and S. Sahni. An optimal routing algorithm for mesh-connected parallel computers. *J. of Assoc. for Comput. Machinery*, 27(1), 1980.

[8] D. Nassimi and S. Sahni. A Self–Routing Beneš Network and Parallel Permutation Algorithms. *IEEE Trans. on Computers*, C-30(5), 1981.

[9] D. Nassimi and S. Sahni. Data broadcasting in simd computers. *IEEE Trans. on Computers*, c-30(2), 1981.

[10] D. Nassimi and S. Sahni. Optimal BPC Permutations on a Cube Connected SIMD Computer. *IEEE Trans. on Computers*, C-31(4), 1982.

[11] M. C. Pease III. The indirect binary n-cube microprocessor array. *IEEE Trans. on Computers*, C-26(5), 1977.

[12] C. S. Raghavendra and R. Boppana. On methods for fast and efficient parallel memory access. In *Proc. 1990 Int. Conf. on Parallel Processing*, pages I76–83, Aug. 1990.

[13] C. S. Raghavendra and R. V. Boppana. On self–routing in Beneš and $(2n - 1)$-stage shuffle-exchange networks. *IEEE Trans. on Computers*, 40(9):1057–1064, Sept. 1991.

[14] J. J. Rotman. *An introduction to the theory of groups.* Wm. C. Brown Publishers, third edition, 1988.

[15] C. L. Seitz. The cosmic cube. *Comm. of Assoc. for Comput. Machinery*, 28(1):22–33, 1985.

[16] J. S. Squire and S. M. Palais. Programming and Design Considerations for a Highly Parallel Computer. In *Proc. Spring Joint Computer Conf.*, 1963.

[17] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. on Computers*, C-20(2):153–161, February 1971.

[18] L. G. Valiant and J. Brebner. Universal schemes for parallel communication. In *Proc. 13th Annual ACM Symp. on Theory of Computing*, 1981.