Constant Time Inner Product and Matrix Computations on Permutation Network Processors

Ming-Bo Lin and A. Yavuz Oruç

Abstract—**Inner product and matrix operations find extensive use in algebraic computations. In this brief contribution, we introduce a new parallel computation model, called a permutation network processor, to carry out these computations efficiently. Unlike the traditional parallel computer architectures, computations on this model are carried out by composing permutations on permutation networks. We show that the sum of N algebraic numbers on this model can be computed in $O(1)$ time using N processors. We further show that the inner product and matrix multiplication can both be computed on this model in $O(1)$ time at the cost of $O(N)$ and $O(N^3)$, respectively, for N element vectors, and $N \times N$ matrices. These results compare well with the time and cost complexities of other high level parallel computer models such as PRAM and CRCW PRAM.**

Index Terms—**Complex inner product, complex matrix multiplication, permutation networks, real inner product, and real matrix multiplication.**

I. Introduction

Inner product and matrix operations form the core of computations of vector and array processors and signal and image processing algorithms. Traditional architectures for carrying out such operations are based on reducing vector computations into scalar operations such as binary addition and multiplication [13], [14]. As a result, much of the computations in vector and array processors is handled by conventional arithmetic circuits such as carry look ahead adders, recoded and cellular array multipliers and dividers [4]. While these conventional circuits are optimized for speed and hardware, they still rely on a variety of building blocks such as adder, subtractor and multiplier cells which often lead to nonuniform arithmetic circuits for vector processors.

In this brief contribution, we propose a new concept to carry out vector and matrix computations. Unlike the traditional architectures, this concept is based on coding not only the operands but also the operations over the operands in such a way that a vector or matrix computation reduces to composing permutation maps. Each operand is coded into a permutation and addition or multiplication of two operands is carried out by composing the permutations that correspond to these operands on a permutation network. As a result, both addition and multiplication are reduced to a single computation, i.e., that of composing permutations. In addition, any other computation involving addition, subtraction and multiplication operations are also reduced to composing permutations.

We show that, on this new computation model, called a permutation network processor, the sum of N n-bit numbers, the inner product of two vectors, each containing N n-bit elements, and the multiplication of two $N \times N$ matrices with n-bit entries can all be computed in $O(1)$ steps. The first two computations require $O(N)$ processors and the matrix multiplication requires (N^3) processors, where each processor handles an n-bit input, and has $O((n + \lg N)^2)$ bit-level cost and $O(n + \lg N)$ bit-level delay.

We note that these results compare well with the complexities for the same computations on other models. For example, on a PRAM model [1, 5], all three computations take $O(\lg N)$ time with the same numbers of processors, where each processor has two $O(n + \lg N)$-bit inputs, and uses arithmetic circuits with $O(n^2 + \lg N)$ bit-level cost. On a cube-connected parallel computer, the same three computations also take $O(\lg N)$ time with the same numbers of processors and with the same processor bit-level complexity [1]. On the combining CRCW PRAM, the same three computations can all be done in $O(1)$ time and with the same numbers of processors, where each processor has two $O(n + \lg N)$-bit inputs, and with $O(n^2 + \lg N)$ bit-level cost. In addition, this model must have a circuit to combine up to N concurrent writes.

We also note that, even though the permutation network processor model stands on its own, it ties with some earlier computation models that were reported in the literature. One such model, called a processing network, was given in [12] where a mesh of processing elements was used to compute certain algebraic formulas. The processing elements in this model can be programmed for arithmetic and routing functions whose combinations lead to various algebraic expressions on the mesh topology. The main difference between this model and the permutation network processor model is that the latter does not rely on an explicit use of adder or multiplier circuits; rather it combines them together using shift permutations. More recently, a new parallel computer model, called a reconfigurable bus system, has been introduced to solve a wide range of problems including sorting problems [15], graph problems [9], [16], and string problems [2]. All these problems have been shown to be solvable in $O(1)$ time on the reconfigurable bus system model. As in the processing network model, processors are connected in this model by some fixed topology such as the mesh, and each processor can be programmed for some data processing as well as routing functions. It is assumed that the signals can be broadcast between processors in constant time regardless of how far the broadcast is carried [8], [15], [16]. The essence of this assumption is that once the processors are simultaneously programmed for some routing functions, the signals that pass through them only encounter a propagation delay which is short enough so as to be considered a constant. The same assumption also holds for our model. Again, the main difference between this model and the permutation network processor is that the latter relies only on permutation maps while the former allows its processors to perform both data processing and routing functions.

Finally, we should note that all computations described in the brief contribution are carried out modulo N. In the case that N is not a power of 2 (which is typically the case because of coprimality contstraints), the results should be converted to binary and this will exact additional time and hardware cost. Also, if the operands are given in binary they must be encoded before they can be computed on. Our complexity expressions do not include these additional encoding and decoding time and cost. The time and hardware complexities of encoding and decoding steps are given in [6], [7] and will be published elsewhere.

Manuscript received July 31, 1992; revised June 5, 1993 and September 13, 1993. This work was supported in part by the Ministry of Education, Taipei, Taiwan, Republic of China and in part by the Minta Martin Fund of the School of Engineering at the University of Maryland.

M.-B. Lin is with the Electronic Engineering Department, National Taiwan Institute of Technology, 43, Keelung Road Section 4, Taipei, Taiwan.

A. Y. Oruç is with the Electrical Engineering Department, Institute of Advanced Computer Studies, University of Maryland, College Park, MD 20742-3025 USA; e-mail: yavuz@eng.umd.edu.

IEEE Log Number 9404362.

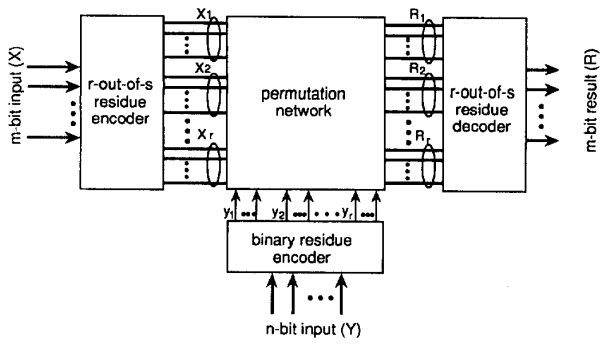

Fig. 1. Organization of a permutation network arithmetic processor.

II. The Permutation Network Processor Model

The computations to be described in subsequent sections all rely on a permutation network processor model which was introduced in [10]. Here we give a brief overview of this model and introduce some changes so as to make it suitable for these computations.

A. The Model

A permutation network processor is obtained by cascading three components together as shown in Fig. 1: an r-out-of-s *residue encoder*, a permutation network stage, and an r-out-of-s *residue decoder*, where r and s are positive integers. The r-out-of-s residue encoder has m inputs, representing an m-bit number X, and r sets of outputs, $X_1, X_2, \cdots, X_r$, where X_i contains m_i outputs, for $i = 1, 2, \cdots, r$. Based on the value of its input X, the r-out-of-s residue encoder sets exactly one output in each X_i to 1 and all other outputs to 0. More precisely, the jth output in X_i is set to 1, where $j = X \bmod m_i$. The r-out-of-s residue decoder is an r-out-of-s residue encoder whose inputs and outputs are switched. That is, it has r sets of inputs $R_1, R_2, \cdots, R_r$. each of which contains a 1 in exactly one of its inputs, and a set of m outputs that represents an n-bit result. The 1's in $R_1, R_2, \cdots, R_r$ are combined into an m-bit result whose residues with respect to moduli $m_1, m_2, \cdots, m_r$ are indicated by the positions of 1's in $R_1, R_2, \cdots, R_r$. The variable s specifies the number of outputs (inputs) of the r-out-of-s residue encoder (decoder). That is, $s = m_1 + m_2 + \cdots + m_r$.

The center stage of the permutation network arithmetic processor consists of a permutation network with s inputs and s outputs. In addition to s inputs that are connected to the outputs of the r-out-of-s residue encoder, the permutation network also has an n-bit control input that represents the second operand Y to the arithmetic processor. The permutation network encompasses r subnetworks $N_1, N_2, \cdots, N_r$, where the lines in X_i from the r-out-of-s residue encoder form the inputs of network N_i and the lines in R_i form its outputs. Network N_i consists of $\lceil \lg m_i \rceil$ stages of switches, numbered $\lceil \lg m_i \rceil - 1, \cdots, 1, 0$ in that order, from left to right, each having m_i inputs and m_i outputs such that the switch in stage $k, k = 0, 1, \cdots, \lceil \lg m_i \rceil - 1$ has two switching states:

state 0: input j is connected to output j, for all $j = 0, 1, \cdots, m_i - 1$;

state 1: input j is connected to output $j + 2^k \bmod m_i$, for all $j = 0, 1, \cdots, m_i - 1$.

Thus, the switch in stage k of network N_i realizes either the identity permutation on its inputs (state 0) or the circular right shift permutation where all inputs are circularly shifted to right by $2^k \bmod m_i$ positions (state 1). The permutations that are realized by networks $N_1, N_2, \cdots, N_r$ are determined by the residues, $y_i = Y \bmod m_i; 1 \leq i \leq r$. The residue y_i is computed from Y by the binary residue encoder in Fig. 1 which converts Y mod m_i to its binary representation. The kth least significant bit of y_i then determines the state of the switch in the kth stage in network N_i. If that bit is 0 then the stage is set to state 0 and if it is 1 then the stage is set to state 1.

In most of the computations that follow, we will need to cascade several permutation network processors together. In such cases, the adjacent r-out-of-s residue decoders and encoders in the intermediate stages of the cascade are redundant (they cancel out), and therefore, will be removed from the model. In this reduced model, we only retain the shift network and binary residue encoder sections of the permutation network arithmetic processors in the intermediate stages. The r-out-of-s residue encoder of the processor in the first stage and the r-out-of-s decoder of the processor in the last stage are also retained.

B. The Cost and Time Assumptions

In the reduced permutation network processor model, each processor has an n-bit input and r simple shift networks with $m_1, m_2, \cdots, m_r$ inputs. These networks when combined together provide a numerical range extending from 0 to $m_1 m_2 \cdots m_r - 1$ for unsigned numbers and from $-\lfloor m_1 m_2 \cdots m_r/2 \rfloor$ to $\lfloor m_1 m_2 \cdots m_r/2 \rfloor$ for signed numbers in 2's complement form.

Let $M = m_1 m_2 \cdots m_r$. It was shown in [7] that a permutation network processor encompassing r subnetworks with $m_1, m_2, \cdots, m_r$ inputs can be constructed by using $O(\lg^2 M)$ logic gates. The two of the computations to be implemented on permutation network processors, namely, the sum of N n-bit 2's complement numbers and the inner product of two N-element vectors each of whose elements is an n-bit 2's complement number requires that $N 2^{n-1} \approx M$. Thus, each of the N permutation network processors needed for these two computations can be constructed with $O((\lg N + n)^2)$ logic gates.

The third computation, i.e., the product of two $N \times N$ matrices can be carried out by performing N^2 inner product operations. Thus, the matrix multiplication problem can be solved by using N^3 permutation network processors each constructed from $O((\lg N + n)^2)$ logic gates.

As for the delay of the permutation network processors needed for these three computations, it was shown in [7] that a permutation network processor encompassing r subnetworks with $m_1, m_2, \cdots, m_r$ inputs has $O(\lg \lg M)$ bit-level delay. Given that $M \approx 2^{n-1} N$, it follows that all three computations mentioned above can be performed by using permutation network processors with $O(n + \lg N)$ bit-level delay.

We point out that these bit-level cost and delay complexities are comparable with those for other parallel computer models. The last two computations require a multiplier circuit for n-bit operands and this exacts $O(n^2)$ bit-level cost to attain $O(\lg n)$ bit-level delay regardless of the model used. Given this, in obtaining the cost and time complexities of the algorithms that follow, we will assume that our permutation network processors have constant cost and constant time where the cost and time are expressed in word level as in other parallel computer models.

III. Inner Product Processors

In this section, we show how to sum N n-bit numbers and compute real and complex inner products using permutation network processors.

A. Summation of N n-Bit Numbers

Assume that N n-bit numbers to be added together are all in 2's complement form. This implies that the sum can be as large as $2^{n-1}N$ and to avoid a possible overflow, the dynamic range M of the permutation network processor must satisfy $2^{n-1}N \leq M/2$, that is, $2^nN \leq M$.

The set $Z_M = \{0, 1, \cdots, M-1\}$ under addition modulo M forms a group which is isomorphic to a cyclic permutation group $(\langle\pi\rangle, \cdot)$ of order M and generated by a permutation π defined over $\{0, 1, \cdots, M-1\}$. The isomorphism between $(Z_M, +_M)$ and $(\langle\pi\rangle, \cdot)$ is fixed by mapping a generator of Z_M onto π. Now let $M = m_1 m_2 \cdots m_r$, where m_i and m_j are relatively prime for all $i \neq j; 1 \leq i, j \leq r$. For Z_M, we fix 1 as its generator and let π be the product of r disjoint cycles, $\pi_1, \pi_2, \cdots, \pi_r$, where

$$\begin{aligned}\pi_1 &= (0 \quad 1 \quad \cdots \quad m_1 - 1)\\ \pi_i &= (m_1 + m_2 + \cdots + m_{i-1}\\ &\quad m_1 + m_2 + \cdots + m_{i-1} + 1 \quad \cdots\\ &\quad m_1 + m_2 + \cdots + m_{i-1} + m_i - 1) \qquad 2 \leq i \leq r. \end{aligned} \tag{1}$$

Noting that $X = \overbrace{1 + 1 + 1 + \cdots + 1}^{X \text{ times}}$, under the isomorphism fixed by mapping 1 to π, element $X \in Z_M$ is mapped to $\pi^X = (\pi_1 \pi_2 \cdots \pi_r)^X$ or since $\pi_1, \pi_2, \cdots, \pi_r$ are disjoint, X is mapped to $\pi_1^X \pi_2^X \cdots \pi_r^X$. As a consequence, the sum $X_1 + X_2 + \cdots + X_N$ modulo M, $X_i \in Z_M$ for $1 \leq i \leq N$ is mapped to

$$\begin{aligned}\pi^{X_1 +_M X_2 +_M \cdots +_M X_N} &= \pi_1^{X_1 +_M X_2 +_M \cdots +_M X_N}\\ &\quad \pi_2^{X_1 +_M X_2 +_M \cdots +_M X_N} \cdots\\ &\quad \pi_r^{X_1 +_M X_2 +_M \cdots +_M X_N}\end{aligned} \tag{2}$$

where $+_M$ denotes modulo M addition. Since π_i is a cycle of length m_i, and $\pi_1, \pi_2, \cdots, \pi_r$ are disjoint

$$\begin{aligned}\pi^{X_1 +_M X_2 +_M \cdots +_M X_N} &= \pi_1^{X_1 +_{m_1} X_2 +_{m_1} \cdots +_{m_1} X_N}\\ &\quad \pi_2^{X_1 +_{m_2} X_2 +_{m_2} \cdots +_{m_2} X_N} \cdots\\ &\quad \pi_r^{X_1 +_{m_r} X_2 +_{m_r} \cdots +_{m_r} X_N}\end{aligned} \tag{3}$$

or by Horner's rule

$$\begin{aligned}\pi^{X_1 +_M X_2 +_M \cdots +_M X_N} &= \pi_1^{((X_1 +_{m_1} X_2) +_{m_1} \cdots +_{m_1} X_N)}\\ &\quad \pi_2^{((X_1 +_{m_2} X_2) +_{m_2} \cdots +_{m_2} X_N)} \cdots\\ &\quad \pi_r^{((X_1 +_{m_r} X_2) +_{m_r} \cdots +_{m_r} X_N)},\end{aligned} \tag{4}$$

where $+_{m_i}$ denotes modulo $m_i; 1 \leq i \leq r$, addition.

To compute (4), we cascade N permutation network processors together and each $X_i; 1 \leq i \leq N$ is converted into its corresponding residue code $(x_{i,1}, x_{i,2}, \cdots, x_{i,r})$ by using a binary residue encoder such as one of those described in [7]. These converted residue codes $(x_{i,1}, x_{i,2}, \cdots, x_{i,r}); 1 \leq i \leq N$, are then used to set up the switching states of corresponding permutation networks. The complete algorithm for computing the summation of N n-bit numbers on a permutation network processor is then given as follows.

Algorithm 1 (Summation of N n -bit 2's complement numbers)
Input: N n-bit 2's complement numbers, $X_1, X_2, \cdots, X_N$.
Output: Sum of $X_1, X_2, \cdots, X_N$ in 2's complement form.
Step 1: Convert X_i's into their corresponding binary residue codes $(x_{i1}, x_{i2}, \cdots, x_{ir})$, in parallel.
Step 2: Add $X_1, X_2, \cdots, X_N$.
  Step 2.1: Set up the switching states of permutation network processors in parallel by the binary residue codes obtained in Step 1.
  Step 2.2: Feed all input lines marked $0, m_1, m_1 + m_2, \cdots, m_1 + m_2 + \cdots + m_{r-1}$ with a "1" and all the other input lines with a "0."
Step 3: Decode the r-out-of-s residue code obtained at the outputs of the last permutation network processor in the cascade into its binary equivalent.

Fig. 2 shows an example with $N = 3$, $n = 5$, and $X_1 = 13, X_2 = -12$, and $X_3 = 9$. The shaded lines indicate the paths of "1" between inputs and outputs. To avoid a possible overflow, the dynamic range of the processor is chosen so that it satisfies $2^nN \leq M$, that is, $3 \times 2^5 \leq M$. Therefore, M is chosen as $105 = 3 \times 5 \times 7$.

This algorithm requires $O(N)$ processors and since it does not contain any loop and each step takes $O(1)$ time, the total execution time of this algorithm is $O(1)$.

B. Inner Product Processor

Let $\mathbf{X} = (X_1, X_2, \cdots, X_N)$ and $\mathbf{Y} = (Y_1, Y_2, \cdots, Y_N)$ be two N-element vectors, where $X_i, Y_i \in Z_M = \{0, 1, \cdots, M-1\}$. Then the real inner product of $\mathbf{X}$ and $\mathbf{Y}$ is a real-valued function defined as

$$\mathbf{X} \cdot \mathbf{Y} = \sum_{j=1}^{N} X_j Y_j. \tag{5}$$

To compute X_jY_j on a permutation network processor, we note that the multiplication modulo M over the set Z_M forms a monoid. Let $Z_{m_i} = \{0, 1, 2, \cdots, m_1 - 1\}$ and $(Z_{m_i}, \times_{m_i})$ denote the monoid under modulo m_i multiplication of elements in $Z_{m_i}; 1 \leq i \leq r$. Let $M = m_1 m_2 \cdots m_r$, where $m_1, m_2, \cdots, m_r$ are all primes. Let $\overline{Z}_M = Z_{m_1} \otimes Z_{m_2} \otimes \cdots \otimes Z_{m_r}$ denote the direct product of $Z_{m_i}; 1 \leq i \leq r$, whose elements are r-tuples $(x_1, x_2, \cdots, x_r)$, where $x_i \in Z_{m_i}$. For any $(x_1, x_2, \cdots, x_r), (y_1, y_2, \cdots, y_r) \in \overline{Z}_M$ define

$$\begin{aligned}&(x_1, x_2, \cdots, x_r) \otimes (y_1, y_2, \cdots, y_r)\\ &\qquad = (x_1 \times_{m_1} y_1, x_2 \times_{m_2} y_2, \cdots, x_r \times_{m_r} y_r).\end{aligned} \tag{6}$$

Now we carry over the product X_iY_i onto $(\overline{Z}_M, \otimes)$ by setting up a monoid isomorphism f between Z_M and $\overline{Z}_M$ as follows:

$$f(X_j) = (x_{j,1}, x_{j,2}, \cdots, x_{j,r}), \qquad \forall X_j \in Z_M, 1 \leq j \leq N \tag{7}$$

where $x_{j,i} = X_j \bmod m_i; 1 \leq i \leq r$. Let $X_j, Y_j \in Z_M$. It is easy to show that

$$\begin{aligned}&f(1) = (1, 1, \cdots, 1)\\ \text{and} \qquad &f(X_j \cdot Y_j) = f(X_j) \otimes f(Y_j)\end{aligned} \tag{8}$$

and hence f is an isomorphism, and using this isomorphism, we can compute the real inner product $\mathbf{X} \cdot \mathbf{Y}$ in $\overline{Z}_M$ as

$$\begin{aligned}\mathbf{X} \cdot \mathbf{Y} &= \sum_{j=1}^{N} X_j Y_j\\ &= \sum_{j=1}^{N} (x_{j,1}, x_{j,2}, \cdots, x_{j,r}) \otimes (y_{j,1}, y_{j,2}, \cdots, y_{j,r})\\ &= \left(\sum_{j=1}^{N} x_{j,1} \times_{m_1} y_{j,1}, \sum_{j=1}^{N} x_{j,2} \times_{m_2} y_{j,2}, \cdots, \right.\\ &\qquad \left. \sum_{j=1}^{N} x_{j,r} \times_{m_r} y_{j,r} \right).\end{aligned} \tag{9}$$

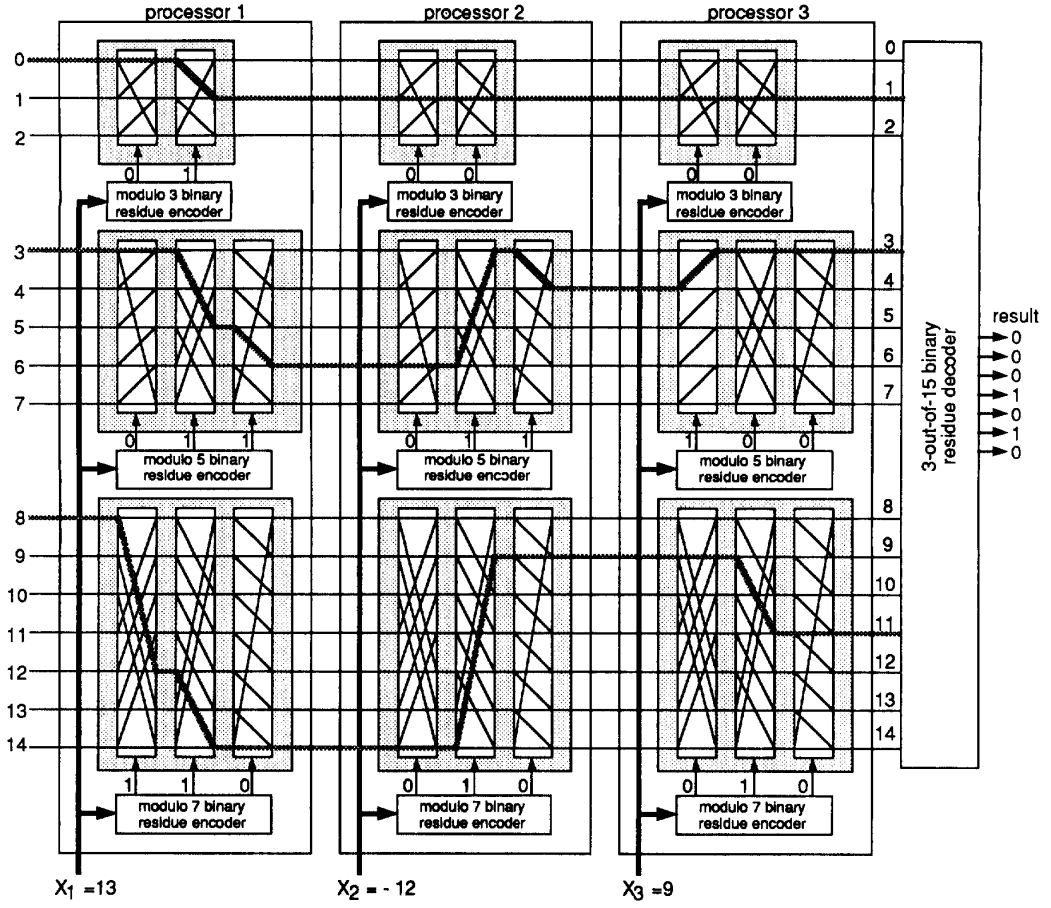

Fig. 2. The permutation network processor shown to compute $13 - {}_{105}12 + {}_{105}9 = 10$.

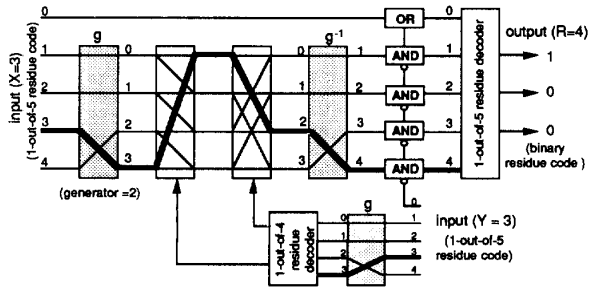

Fig. 3. Permutation network modulo 5 multiplier.

The corresponding permutation network real inner product processor is constructed as follows. For each modulus, m_i, N modulo m_i permutation network processors are cascaded. One operand of each processor comes from the previous stage and the other operand comes from the output of a modulo m_i permutation network multiplier. An example of modulo m_i permutation network multiplier is shown in Fig. 3 for $m_i = 5$. Briefly, shaded boxes labeled by g code X and Y into the powers of the generator of the modulo 5 multiplication table, which, in this example, is 2. The 2-stage shift network then computes the product of X and Y and the product is converted back to a 1-out-of-4 code by the shaded box labeled with g^{-1}. The binary product $R = XY$ is then obtained by converting this 1-out-of-4 code by a residue decoder. Note that $X = 0$, or $Y = 0$, or $X = 0$ and $Y = 0$ are treated as special cases via the logic circuit before the 1-out-of-5 decoder section. A more detailed description of modulo m_i permutation network multiplier can be found in [6], [7].

The following algorithm shows how the inner product is carried out using such multipliers.

Algorithm 2 (Real inner product of two vectors)

Input: Two vectors $\mathbf{X} = (X_1, X_2, \cdots, X_N)$ and $\mathbf{Y} = (Y_1, Y_2, \cdots, Y_N)$, where each X_i and Y_i is an n-bit 2's complement number.

Output: The real inner product of $\mathbf{X}$ and $\mathbf{Y}$ represented in 2's complement form.

Step 1: Convert each element of $\mathbf{X}$ and $\mathbf{Y}$ into its corresponding residue code in parallel.

Step 2: Multiply x_{ji} and y_{ji} for $1 \leq j \leq N$ and $1 \leq i \leq r$ in parallel.

Step 3: Compute the inner product by adding together the products obtained in Step 2 over the residue code domain.

> **Step 3.1**: Set up the switching states of permutation network processors in parallel by the binary residue codes obtained in Step 2.
>
> **Step 3.2**: Feed all input lines marked $0, m_1, m_1 + m_2, \cdots, m_1 + m_2 + \cdots + m_{r-1}$ with a "1" and all the other input lines with a "0."

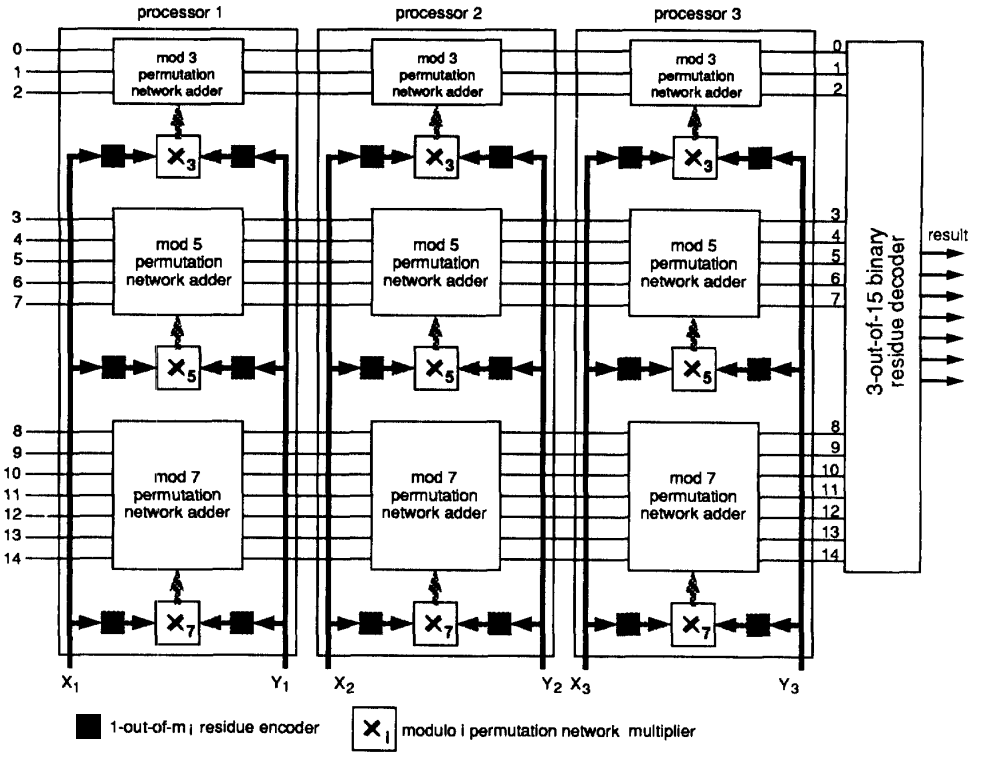

Fig. 4. The permutation network processor for computing the real inner-product of $\mathbf{X}$ and $\mathbf{Y}$ ($N = 3$ and $n = 3$).

Step 4: Decode the r-out-of-s residue code obtained at the outputs of the last permutation network processor in the cascade into its binary equivalent.

An example with $N = 3$ and $n = 3$ is shown in Fig. 4. It is easy to see that Algorithm 2 requires $O(N)$ processors and its total execution time is $O(1)$ since each step takes $O(1)$ time.

We leave the construction of a complex inner product processor to the reader and just note that the inner product of two N-element complex vectors can also be computed in $O(1)$ time using $O(N)$ permutation network processors.

IV. Matrix Multiplication

Let $\mathbf{A}$ and $\mathbf{B}$ be $N \times N$ real matrices and $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. C can be computed in $O(1)$ steps as follows.

Algorithm 3 (Multiplication of two real matrices)
Input: Two $N \times N$ real matrices $\mathbf{A}$ and $\mathbf{B}$. Each element of $\mathbf{A}$ and $\mathbf{B}$ is an n-bit 2's complement number.
Output: A real product matrix $\mathbf{C} = \mathbf{A} \times \mathbf{B}$.
Step 1: Convert each element of $\mathbf{A}$ and $\mathbf{B}$ into its corresponding residue code in parallel.
Step 2: Compute the N^2 inner products using Algorithm 2.

All inner product processors operate in parallel in Step 2. Since each executes in $O(1)$ time, Algorithm 3 takes $O(1)$ time to execute and it needs a total of N^2 inner product processors each having $O(N)$ cost and hence its cost is $O(N^3)$.

The multiplication of two complex matrices can also be carried out in $O(1)$ time using $O(N^3)$ permutation network processors using four real matrix multiplications, one matrix addition, and one matrix subtraction.

V. Concluding Remarks

In this brief contribution, we proposed permutation network processors to compute algebraic sums, inner and matrix products. It has been shown that the algebraic sum of N n-bit 2's complement numbers can be computed in $O(1)$ time on such a processor with $O(N)$ cost. The inner product of two N-element vectors (both real and complex) with n-bit elements can also be done in $O(1)$ time using a similar processor with $O(N)$ cost. On the other hand, the matrix multiplication takes $O(1)$ time but with $O(N^3)$ permutation network processors.

These results are important in that they establish that one can avoid using conventional adder and multiplier circuits to carry out vector and matrix computations.

Acknowledgment

The authors thank the anonymous reviewers for their constructive comments.

References

[1] Selim G. Akl, *The Design and Analysis of Parallel Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
[2] D. M. Champion and J. Rothstein, "Immediate parallel solution of the longest common subsequence problem," in *IEEE Int. Conf. Parallel Processing*, 1987, pp. 70–77.
[3] K. M. Elleithy, M. A. Bayoumi, and K. P. Lee, "$\theta(\lg N)$ architecture for RNS arithmetic decoding," in *IEEE 9th Comput. Arith. Symp.*, 1989, pp. 202–209.
[4] K. Hwang, *Computer Arithmetic: Principle, Architecture, and Design*. New York: John Wiley, 1979.
[5] S. Lakshmivarahan and Sudarshan K. Dhall, *Analysis and Design of Parallel Algorithms*, McGraw-Hill Pub., 1990.

[6] M.-B. Lin and A. Yavuz Oruç, "The design of a network-based arithmetic processor," Tech. Rep. UMIACS-TR-91-141, Univ. of Maryland, College Park, MD, Oct. 1991.
[7] M.-B. Lin, "Unified algebraic computations on permutation networks," Ph.D. dissertation, EE Dept., Univ. of Maryland, College Park, 1992.
[8] M. Maresca and H. Li, "Connection autonomy in SIMD computers: A VLSI implementation," *J. Parallel Distrib. Computing*, vol. 7, pp. 302–320, 1989.
[9] R. Miller, V. K. Prasanna Kumar, D. Reisis, and Q. F. Stout, "Data movement operations and applications on reconfigurable VLSI arrays," in *Int. Conf. Parallel Processing*, St. Charles, IL, vol. I, Aug. 1988, pp. 205–208.
[10] A. Yavuz Oruç, V. G. J. Peris, and M. Yaman Oruç, "Parallel modular arithmetic on a permutation network," in *Int. Conf. Parallel Processing*, St. Charles, IL, vol. 1, Aug. 1991, pp. 706–707.
[11] S. J. Piestrak, "Design of residue generators and multi-operand modular adders using carry-save adders," in *IEEE 10th Comput. Arith. Symp.*, 1991, pp. 100–107.
[12] W. Shen and A. Yavuz Oruç, "Mapping algebraic formulas onto mesh connected processor networks," *Inform. Sci. Syst. Conf.*, Princeton Univ., NJ, pp. 535–538, 1986.
[13] S. P. Smith and H. C. Torng, "Design of a fast inner product processor," in *Proc. IEEE 7th Comput. Arith. Symp.*, 1985, pp. 38–43.
[14] E. E. Swartzlander, Jr., B. K. Gilbert and I. S. Reed, "Inner product computers," *IEEE Trans. Comput.*, vol. C-27, pp. 21–31, Jan. 1978.
[15] B. F. Wang, G. H. Chen, and F. C. Lin, "Constant time sorting on a processing array with a reconfigurable bus system," *Inform. Processing Lett.*, pp. 187–192, 1990.
[16] B. F. Wang and G. H. Chen, "Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, pp. 500–507, Oct. 1990.

Structural and Tree Embedding Aspects of Incomplete Hypercubes

Nian-Feng Tzeng and Hsing-Lung Chen

***Abstract*—Since the hypercube is not incrementally scalable, a variant hypercube topology with more flexibility in the system size, called an *incomplete hypercube*, is examined. An incomplete hypercube may also result from a complete hypercube which operates in a degraded manner after some nodes fail. Elementary properties, including diameter, mean internode distance, and traffic density, of incomplete hypercubes with size $2^n + 2^k$, $0 \leq k \leq n$, are derived. Interestingly, traffic density over links in such an incomplete hypercube is found to be bounded by 2 (messages per link per unit time), despite its structural nonhomogeneity. Thus, cube links can easily be constructed so as to avoid any single point of congestion, guaranteeing good performance. The minimum incomplete hypercubes able to embed binary trees with node adjacencies preserved are determined.**

***Index Terms*— Hypercubes, incomplete hypercubes, message routing, network topology, structural properties, tree embeddings.**

Manuscript received August 26, 1992; revised May 13, 1993 and December 16, 1993. this work was supported in part by the NSF under Grants MIP-9201308 and CCR-9300075 and by the State of Louisiana under Contract LEQSF(92-94)-RD-A-32.

N.-F. Tzeng is with the Center for Advanced Computer Studies, University of Southwestern Louisiana, P.O. Box 70504–4433, Lafayette, LA 70504–4330 USA; e-mail: tzeng@cacs.usl.edu.

H.-L. Chen is with the Department of Electronic Engineering, National Taiwan Institute of Technology, Taipei, Taiwan, R.O.C.

IEEE Log Number 9404364.

I. Introduction

Unlike a complete hypercube, an *incomplete hypercube* allows for the construction of a system with size not necessary a power of 2. It may also result from a complete hypercube after some nodes become faulty and the system is reconfigured, as discussed in [7]. Simple and deadlock-free algorithms for routing and for broadcasting messages in the incomplete hypercube have been developed in [5]. In this brief contribution, we deal with the incomplete hypercube comprising two complete hypercubes, one of size 2^n and the other of size 2^k $(0 \leq k \leq n)$. A related structure introduced recently is the Fibonacci cube, which consists of two smaller Fibonacci cubes of unequal sizes and is a subgraph of a hypercube [2].

Here we are interested in finding whether or not an incomplete hypercube exhibits any heavy traffic link or node that may become a vulnerable point with respect to performance and reliability. Structural properties of incomplete hypercubes, including diameter, mean internode distance, and traffic density, are obtained. The highest traffic density over links in incomplete hypercubes is bounded by 2, despite its nonhomogeneity. With bounded traffic density, incomplete hypercubes are clearly superior to other nonhomogeneous topologies, such as trees and stars, where points of congestion are likely to exist and serious performance degradation may result because in such a topology, the highest traffic density is proportional to its size. If one wants to put a complete hypercube into operation even after some nodes become faulty and the system is reconfigured into an incomplete hypercube, cube links can easily be so designed that every link is still below half saturated in the presence of faults to prevent any traffic bottleneck from occurring.

In the second part of this work, embedding binary trees in the incomplete hypercube is pursued and compared to that in its complete counterpart. It is illustrated that the incomplete hypercube is capable of better supporting binary trees than a compatible complete hypercube. The embedding results also reveal that a complete hypercube still can effectively support binary trees even after cube links/nodes fail, provided that the operating portion of the injured hypercube is no smaller than the respective incomplete hypercubes able to embed those binary trees.

II. Notations and Background

An n-dimensional complete hypercube, denoted by H_n, comprises 2^n nodes, each with n bidirectional links connecting to n immediate neighbors. An incomplete hypercube of interest IH_k^n comprises two complete cubes, H_n and H_k, which respectively have 2^n and 2^k nodes, where $0 \leq k \leq n$. Nodes in IH_k^n are labeled from 0 to $2^n + 2^k - 1$ by an $(n+1)$-bit binary representation in such a way that any two nodes connected by a link differ in their labels by exactly one bit, and that nodes in H_n are numbered from 0 to $2^n - 1$, while nodes in H_k are from 2^n to $2^n + 2^k - 1$. Fig. 1 shows an incomplete hypercube with 12 nodes, IH_2^3. A d-dimensional subcube in IH_k^n contains 2^d nodes and is represented by a string of $n+1$ symbols over $\{0, 1, *\}$ such that there are exactly d $*$'s (which denote *don't care*). The link between two neighboring nodes A and B is referred to as λ_B^A. A *path* from node A to node B is denoted by Λ_B^A, and we are interested in the *shortest paths* only. A link is assumed to have *link number* i if it connects two nodes whose addresses differ in the ith bit position (starting with the least significant bit as bit 0).

As opposed to those in a complete hypercube, nodes in an incomplete hypercube no longer play an identical role. For instance, every node in subcube $(00**)$ of IH_2^3 shown in Fig. 1 has four links