A Programming Methodology for Designing Block Recursive Algorithms*

MIN-HSUAN FAN, CHUA-HUANG HUANG, YEH-CHING CHUNG+
JEN-SHIUH LIU AND JEI-ZHII LEE++
Department of Information Engineering and Computer Science
Feng Chia University
Taichung, 407 Taiwan
+*Department of Computer Science*
National Tsing Hua University
Hsinchu, 300 Taiwan
++*Department of Computer Science and Information Engineering*
National Dong Hwa University
Hualien, 974 Taiwan

In this paper, we use the tensor product notation as the framework of a programming methodology for designing block recursive algorithms. We first express a computational problem in its matrix form. Next, we formulate a matrix equation for the matrix of the computational problem. Then, we try to find a solution of the matrix equation such that the solution is composed of simple matrices. Finally, we recursively factorize the subproblem to obtain a tensor product formula representing an algorithm for the given problem. In this methodology, the operations of a tensor product formula can be mapped to language constructs of high-level programming languages. That is, we can generate computer programs, including programs for parallel computers and distributed-memory multiprocessors, from tensor product formulas. In this paper, we use the parallel prefix problem and the discrete Fourier transform problem as examples to illustrate the methodology and derive various parallel prefix and fast Fourier transform algorithms.

Keywords: programming methodology, tensor product, block recursive algorithm, parallel processing, distributed processing, parallel prefix, fast Fourier transform

1. INTRODUCTION

Tensor products, also known as Kronecker products [4], have been successfully used to express and implement parallel block recursive algorithms, such as fast Fourier Transforms [9, 10], Strassen's matrix multiplication [7, 8, 14], Hilbert space-filling curves [15], and the Karatsuba's multiplier [16]. The tensor product notation is also suitable for expressing data distribution and modeling interconnection networks [11, 12]. The tensor product operations can be mapped to corresponding programming language constructs. Therefore, the tensor product notation provides a framework for designing and implementing parallel programs [5, 24].

Received January 9, 2004; revised April 29, 2004; accepted June 8, 2004.
Communicated by Gen-Huey Chen.
* This work was supported in part by the National Science Council, R.O.C., No. NSC 89-2213-E-259-005.

EXTENT is an expert system that takes various tensor product formulas of fast Fourier transform and Strassen's matrix multiplication problems, and automatically generates vector programs for the Cray Y-MP computer and distributed memory programs for the Intel Paragon computer [3]. SPIRAL is a project designed to automate the process of implementation, optimization, and platform adaptation of digital signal processing (DSP) algorithms [18]. In SPIRAL, a set of basic tensor product formulas of DSP problems is given. A computer algebra framework, AREP, is used to interactively explore fast DSP algorithms [17, 21, 22]. Both the EXTENT and SPIRAL systems start with various kinds of tensor product formulas and then employ the tensor product algebraic theory to obtain alternative formulas. In our work, we present a systematic methodology that starts with matrix specification of a given problem and then derives tensor product formulas of the computational problem stepwisely. We will use the parallel prefix problem and the discrete Fourier transform computation problem as examples to illustrate the programming methodology and to derive various parallel prefix and fast Fourier transform (FFT) algorithms.

The programming methodology is divided into four steps. First, a computational problem is expressed in its matrix form. Next, a matrix equation for the matrix of the computational problem is formulated. Third, the matrix equation is solved to obtain a feasible solution [6]. Finally, the subproblem is recursively factorized to obtain a tensor product formula that represents an algorithm for the specified computational problem. The solution to a matrix equation obtained in the third step is not unique. This will yield various tensor product formulas and, thus, different computational algorithms for the given computational problem. In this paper, we will employ the methodology to derive various parallel prefix algorithms, including divide-and-conquer and recursive doubling algorithms [2] and the fundamental Cooley-Tukey FFT algorithm [1]. Based on the Cooley-Tukey FFT algorithm, various FFT algorithms, including those presented by Pease [20], Korn-Lambiotte [13], and Stockham [23], can be derived [9].

The organization of this paper is as follows. Section 2 defines the tensor product notation and introduces the properties that we will use through this paper. Section 3 presents a programming methodology for designing block recursive algorithms. Sections 4 and 5 illustrate the methodology to by deriving parallel prefix algorithms and the Cooley-Tukey FFT algorithm. Conclusions and future works are given in section 6.

2. THE TENSOR PRODUCT NOTATION

In this section, we will give a brief overview of the tensor product definition and relevant properties. The tensor product operation is a bilinear form that constructs a block matrix from two matrices.

Definition 2.1 Tensor Product Let A be an $m \times n$ matrix, and let B be a $p \times q$ matrix. The tensor product of A and B, $A \otimes B$, is the block matrix obtained by replacing each element, $a_{i,j}$, with the matrix $a_{i,j}B$, i.e., the result is an $mp \times nq$ matrix, defined as

$$A \otimes B = \begin{bmatrix} a_{0,0}B & \cdots & a_{0,n-1}B \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B & \cdots & a_{m-1,n-1}B \end{bmatrix}.$$

Two important forms of the tensor product operation occur when one of the operands is the identity matrix. If the first operand is the identity matrix, i.e., $Y = (I_n \otimes A)X$, it can be interpreted as parallel operations on segments of X and is called the parallel form. If X and Y are partitioned into n segments with q and p elements in each segment, respectively, then $Y = (I_n \otimes A)X$ can be rewritten as the follows:

$$Y = (I_n \otimes A)X \equiv \begin{bmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_{n-1} \end{bmatrix} = \begin{bmatrix} A & 0 & \cdots & 0 \\ 0 & A & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{n-1} \end{bmatrix}.$$

The resulting formula is implemented as parallel operations, $Y_0 = AX_0 \parallel Y_1 = AX_1 \parallel \ldots \parallel Y_{n-1} = AX_{n-1}$. Note that the indices used in this paper always start from 0.

If the second operand is the identity matrix, i.e., $Y = (A \otimes I_n)X$, then it can be interpreted as vector operations on elements of X and is called the vector form.

Let A be a $p \times q$ matrix. If X is partitioned into q segments with n elements in each segment, and if Y is partitioned into p segments with n elements in each segment, then $Y = (A \otimes I_n)X$ can be rewritten as follows:

$$Y = (A \otimes I_n)X \equiv \begin{bmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_{p-1} \end{bmatrix} = \begin{bmatrix} a_{0,0}I_n & a_{0,1}I_n & \cdots & a_{0,q-1}I_n \\ a_{1,0}I_n & a_{1,1}I_n & \cdots & a_{1,q-1}I_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{p-1,0}I_n & a_{p-1,1}I_n & \cdots & a_{p-1,q-1}I_n \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{q-1} \end{bmatrix}.$$

The resulting formula is implemented as vector operations

$Y_0 = a_{0,0}X_0 + a_{0,1}X_1 + \ldots + a_{0,q-1}X_{q-1}$
$\parallel Y_1 = a_{1,0}X_0 + a_{1,1}X_1 + \ldots + a_{1,q-1}X_{q-1}$
$\parallel \ldots$
$\parallel Y_{p-1} = a_{p-1,0}X_0 + a_{p-1,1}X_1 + \ldots + a_{p-1,q-1}X_{q-1}$

The vector operation $Y_i = a_{i,0}X_0 + a_{i,1}X_1 + \ldots + a_{i,q-1}X_{q-1}$ consists of a sequence of scalar-vector multiplications and a sequence of vector-vector additions.

We use $\{e_i^m \mid 0 \le i < m\}$ to denote the basis of the m-dimensional vector space, where e_i^m is a column vector of length m with 1 at position i and 0's elsewhere; we also use $\{E_{i,j}^{m,n} \mid 0 \le i < m,\ 0 \le j < n\}$ to denote the basis of $m \times n$ matrices, where $E_{i,j}^{m,n}$ is an $m \times n$ matrix with 1 at position (i, j) and 0's elsewhere. In the tensor product theory, a permutation is often used to change data communication pattern. This permutation is

called *stride permutation*, which rearranges data elements or allows access to data elements with a fixed stride distance. A stride permutation is a mapping of a basis vector to another basis vector. The functional definition of stride permutation is

$$L_n^{mn}(e_i^{mn}) = \begin{cases} e_{mi \bmod (mn-1)}^{mn} & \text{if } 0 \le i < mn-1 \\ e_i^{mn} & \text{if } i = mn-1 \end{cases}.$$

This functional definition can be expressed as a tensor product operation. Hence, we give an operational definition of stride permutation below.

Definition 2.2 Stride Permutation $L_n^{mn}(e_i^m \otimes e_j^n) = e_j^n \otimes e_i^m$.

L_n^{mn} is the stride permutation of vector size mn stride with distance n. When a matrix is stored in a rowwise manner, the tensor basis $e_i^m \otimes e_j^n$ is isomorphic to $E_{i,j}^{m,n}$. In addition, the tensor basis $e_j^n \otimes e_i^m$ is isomorphic to $E_{i,j}^{m,n}$ when a matrix is stored in a columnwise manner. Therefore, L_n^{mn} transposes an $m \times n$ matrix from the row-major order to the column-major order. When L_n^{mn} is applied to a vector representing input data, it is interpreted as a load operation; when L_n^{mn} is applied to a vector representing output data, it is interpreted as a store operation. For example, suppose X and Y are vectors of length mn denoting input and output data, respectively. An interpretation of $Y = L_n^{mn} X$ is to load input data elements with stride distance n. An interpretation of $L_n^{mn} Y = X$, or rewritten as $Y = (L_m^{mn})^{-1} X$, is to store output elements with stride distance m.

The following are some properties of tensor products and stride permutation. The reader may refer to [4] for the proofs of these properties. Note that, $\prod_{i=0}^{n-1} F_i$ is denoted as matrix product $F_{n-1}F_{n-2} \ldots F_1F_0$, because the matrix product is not commutative and F_0 is applied to an input operand first.

1. Associative law: $A \otimes B \otimes C = (A \otimes B) \otimes C = A \otimes (B \otimes C)$.
2. Multiplicative law (1): $(A_1 \otimes \ldots \otimes A_k)(B_1 \otimes \ldots \otimes B_k) = (A_1B_1 \otimes \ldots \otimes A_kB_k)$.
3. Multiplicative law (2): $(A_1 \otimes B_1)(A_2 \otimes B_2) \ldots (A_k \otimes B_k) = (A_1A_2 \ldots A_k \otimes B_1B_2 \ldots B_k)$.
4. Distributive law tensor product over multiplication (1): $\prod_{i=0}^{n-1}(I_n \otimes A_i) = I_n \otimes \left(\prod_{i=0}^{n-1} A_i\right)$.
5. Distributive law tensor product over multiplication (2): $\prod_{i=0}^{n-1}(A_i \otimes I_n) = \left(\prod_{i=0}^{n-1} A_i\right) \otimes I_n$.
6. Inversion of tensor product: $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$.
7. Inversion of stride permutation: $(L_n^{mn})^{-1} = L_m^{mn}$.
8. Identity of stride permutation: $L_n^n = I_n$.
9. Factorization of stride permutation: $L_{rs}^{rst} = L_r^{rst} L_s^{rst}$.
10. Commutation theorem: $A_{m_1 \times n_1} \otimes B_{m_2 \times n_2} = L_{m_1}^{m_1 m_2}(B_{m_2 \times n_2} \otimes A_{m_1 \times n_1}) L_{n_2}^{n_1 n_2}$.

The multiplicative law is a key property with respect to performance issues. Let A be an $m \times n$ matrix, and let B be a $p \times q$ matrix. By the multiplicative law, we have $A \otimes B = (A \otimes I_p)(I_n \otimes B) = (I_m \otimes B)(A \otimes I_q)$. The complexity of $A \otimes B$ is $O(mnpq)$. The com-

plexity of $(A \otimes I_p)(I_n \otimes B)$ is $O(mnp + npq)$ and the complexity of $(I_m \otimes B)(A \otimes I_q)$ is $O(mpq + mnq)$. If the multiplicative law is applied repeatedly, an algorithm of $O(N^2)$ complexity can be reduced to $O(N\log N)$.

3. TENSOR PRODUCT BASED PROGRAMMING METHODOLOGY FOR DERIVING BLOCK RECURSIVE ALGORITHMS

Many block recursive algorithms can be represented by tensor product formulas, e.g., discrete Fourier transformation, discrete cosine transformation, Walsh-Haldamard transformation, discrete Hartley transformation, *etc*. Although these block recursive algorithms have been reported, we are interested in how to derive various block recursive algorithms for a given computational problem can be derived from its specification matrix step by step. In this section, we will present a programming methodology for deriving various block recursive algorithms based on the tensor product theory.

The programming methodology starts with matrix specification of a computational problem and performs stepwise refinement on it to obtain tensor product formulas of the specified matrix. The programming methodology contains four steps.

Step 1: Represent the computational problem as an $N \times N$ matrix Q_N, where N is the input problem size. Then, $Y_N = Q_N X_N$ means the application of Q_N to an input column vector X_N with the result of output column vector Y_N.

Step 2: Factorize the specified matrix Q_N to obtain a tensor product formula in parallel form, $I_r \otimes Q_{N/r}$, or in vector form, $Q_{N/r} \otimes I_r$, with pre-operation P_N and post-operation R_N. For example, if $N = 2^n$, Q_N is factorized to obtain $Q_{N/2}$, the recursive parallel form is

$$Q_N = R_N(I_2 \otimes Q_{N/2})P_N, \quad (1)$$

and the recursive vector form is

$$Q_N = R_N(Q_{N/2} \otimes I_2)P_N. \quad (2)$$

The recursive parallel form in Eq. (1) means the original problem is divided into two copies of the same problem with the original half size. The intuition behind Eq. (1) is the divide-and-conquer method. Before the dividing phase, a pre-operation P_N may be required to read data elements in a specific order other than sequential access or to perform a computation which can be easily and effectively implemented on a given computer architecture. After the conquering phase, a post-operation R_N may be required to write data elements in a specific order other than the natural order or to perform a computation which can be easily and effectively implemented on a given computer architecture. The recursive vector form in Eq. (2) means the original problem is reduced to the same problem with half the original size and applied to a collection of block data with two elements in each block. Similarly, a pre-operation P_N and a post-operation R_N may be required to re-arrange data elements or to perform a computation.

Step 3: Solve Eqs. (1) or (2) for unknown matrices P_N and R_N. Since there is only one equation and two unknown matrices, many solutions of P_N and R_N are possible. We need to use heuristics to determine P_N and R_N. Since P_N or R_N may be a data re-arrangement operation or a simple computing operation, it is not feasible to determine a computing operation. Therefore, we first select a data re-arrangement operation which is precisely a permutation operation. In this methodology, we adopt various strategies to fix an unknown matrix as a permutation operation and then solve another one. For $N = 2^n$, we have the following strategies:

Strategy 1: Set P_N or R_N to identity matrix I_N. That is, the input or output elements are accessed in the natural order.

Strategy 2: Set P_N to L_2^N in Eq. (1) or to $L_{N/2}^N$ in Eq. (2). Eq. (1) is $R_N(I_2 \otimes Q_{N/2})P_N$. If the pre-operation P_N is L_2^N, then this means that data elements are read at a stride of distance of two. The first sub-problem $Q_{N/2}$ is applied at the even position of the original input data; the second sub-problem $Q_{N/2}$ is applied at the odd position of the original input data. Eq. (2) is $R_N(Q_{N/2} \otimes I_2)P_N$. If the pre-operation P_N is $L_{N/2}^N$, then this means that data elements are read at a stride of distance of $N/2$. The first element of each data block is read from the first half of the original input data, and the second element of each data block is read from the second half of the original input data.

Strategy 3: Set R_N to $L_{N/2}^N$ in Eq. (1) or to L_2^N in Eq. (2). Eq. (1) is $R_N(I_2 \otimes Q_{N/2})P_N$. If the post-operation R_N is $L_{N/2}^N = (L_2^N)^{-1}$, then by Property (7), data elements are written at a stride of distance of two. The result of the first sub-problem $Q_{N/2}$ is written to the even position of the output data; the result of the second sub-problem $Q_{N/2}$ is written to the odd position of the output data. Eq. (2) is $R_N(Q_{N/2} \otimes I_2)P_N$. If the post-operation R_N is $L_2^N = (L_{N/2}^N)^{-1}$, then by Property (7), data elements are written at a stride of distance of $N/2$. The first element of each data block is written to the first half of the output data, and the second element of each data block is written to the second half of the output data. The choice of strategy is determined based on the intended initial and final data allocation.

Step 4: Recursively factorize the sub-problem $Q_{N/r}$ as in step 3. The final factorization is a matrix product consisting of tensor products of some simple operation matrices that can be interpreted as an algorithm for the given computational problem. A *simple operation matrix* contains at most a "few" none-zero elements in each row such that it can be implemented to a sequence of assignments or a single loop with a sequence of assignments. For example, let

$$S_{2^n} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^n \times 2^n} .$$

The elements on and below the diagonal of S_{2^n} are 1's and 0's elsewhere. Since each row of S_{2^n} contains of at most two 1's, S_{2^n} is said to be a simple operation matrix. The simple operation matrix $Y_{2^n} = S_{2^n}X_{2^n}$ can be implemented as the following basic program unit:

```
y[0] = x[0];
for i = 1 to 2^n − 1
    y[i] = x[i] + x[i − 1]
end for
```

The program means that each element of the input vector is added to the previous element except the first one. For parallel programming, the program can be run in parallel.

The key tasks in deriving a tensor product formula for a given computational problem are the choice of recursive parallel form and recursive vector form in step 2 and the application of a strategy in step 3. Since, theoretically, there are infinitely many possible solutions for the factorization of Eqs. (1) or (2), some solutions are significant and some may not be meaningful. Different criteria, such as various computer architectures and data distributions, should be considered when we design a block recursive algorithm. In the following sections, the parallel prefix problem and the discrete Fourier transform problem will be used to illustrate the programming methodology of tensor product formula derivation.

4. PARALLEL PREFIX ALGORITHMS

The parallel prefix problem is that of computing partial prefix sums of a sequence of N elements. Let the input elements be x_0, x_1, …, and x_{N-1}, and let the computational results be y_0, y_1, …, and y_{N-1}, such that $y_i = \sum_{k=0}^{i} x_i$, for $0 \le i < N$. If the input elements and resulting elements are represented as vectors X and Y, respectively, then the parallel prefix problem can be represented as $Y_N = Q_N X_N$, where Q_N is an $N \times N$ matrix such that all the lower triangular and diagonal elements of Q_N are 1's, and all the upper triangular elements of Q_N are 0's. That is,

$$Q_N = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 0 \\ \cdots & \vdots & \ddots & \cdots \\ 1 & \cdots & 1 & 1 \end{bmatrix}_{N \times N}.$$

In this section, we will derive the divide-and-conquer parallel prefix algorithm, recursive-doubling parallel prefix algorithm, reverse recursive-doubling parallel prefix algorithm, and parallel prefix algorithms with the bit reversal operation using the programming methodology.

4.1 Divide-and-Conquer Algorithm

We select the recursive parallel form of Eq. (1) to derive the divide-and-conquer parallel prefix algorithm. Initially, assuming that the input data are accessed in the nature order, we set pre-operation P_N to the identity matrix I_N. Let N be 2^n. We obtain the matrix equation $Q_{2^n} = R_{2^n}(I_2 \otimes Q_{2^{n-1}})$. Then, matrix R_{2^n} is solved as follows:

$$R_{2^n} = Q_{2^n}(I_2 \otimes Q_{2^{n-1}})^{-1} \quad \text{(multiply } (I_2 \otimes Q_{2^{n-1}})^{-1} \text{ on both side of the matrix equation)}$$

$$= \begin{bmatrix} Q_{2^{n-1}} & 0 \\ 1_{2^{n-1}} & Q_{2^{n-1}} \end{bmatrix} (I_2 \otimes Q_{2^{n-1}}^{-1}) \quad \text{(rewrite } Q_{2^n} \text{ and by Properity (6))}$$

$$= \begin{bmatrix} Q_{2^{n-1}} & 0 \\ 1_{2^{n-1}} & Q_{2^{n-1}} \end{bmatrix} \begin{bmatrix} Q_{2^{n-1}}^{-1} & 0 \\ 0 & Q_{2^{n-1}}^{-1} \end{bmatrix} \quad \text{(expand } I_2 \otimes Q_{2^{n-1}}^{-1}\text{)}$$

$$= \begin{bmatrix} Q_{2^{n-1}} Q_{2^{n-1}}^{-1} & 0 \\ 1_{2^{n-1}} Q_{2^{n-1}}^{-1} & Q_{2^{n-1}} Q_{2^{n-1}}^{-1} \end{bmatrix} \quad \text{(matrix multiplication)}$$

$$= \begin{bmatrix} I_{2^{n-1}} & 0 \\ 1_{2^{n-1}} Q_{2^{n-1}}^{-1} & I_{2^{n-1}} \end{bmatrix},$$

where $1_{2^{n-1}}$ is the $2^{n-1} \times 2^{n-1}$ matrix with all elements being 1. Define $T_{2^{n-1}}$ as the $2^{n-1} \times 2^{n-1}$ matrix with 1's in the last column and 0's elsewhere, i.e.,

$$T_{2^{n-1}} = \begin{bmatrix} 0 & \cdots & 0 & 1 \\ 0 & \cdots & 0 & 1 \\ & \cdots & & \\ 0 & \cdots & 0 & 1 \end{bmatrix}_{2^{n-1} \times 2^{n-1}}.$$

Since $1_{2^{n-1}} = T_{2^{n-1}} Q_{2^{n-1}}$, we have $1_{2^{n-1}} Q_{2^{n-1}}^{-1} = T_{2^{n-1}}$. Finally, we obtain the following result

$$R_{2^n} = \begin{bmatrix} I_{2^{n-1}} & 0 \\ T_{2^{n-1}} & I_{2^{n-1}} \end{bmatrix}.$$

Solving the equation recursively, we can derive the tensor product formula $Q_{2^n} = \prod_{i=1}^{n} (I_{2^{n-i}} \otimes R_{2^i})$, such that

$$R_{2^i} = \begin{bmatrix} I_{2^{i-1}} & 0_{2^{i-1}} \\ T_{2^{i-1}} & I_{2^{i-1}} \end{bmatrix}_{2^i \times 2^i}, \text{ where } T_{2^{i-1}} = \begin{bmatrix} 0 & \cdots & 0 & 1 \\ 0 & \cdots & 0 & 1 \\ & \cdots & & \\ 0 & \cdots & 0 & 1 \end{bmatrix}_{2^{i-1} \times 2^{i-1}}, \text{ for } 0 < i \le n. \quad (3)$$

It is easy to see that, for $i = 1$, R_2 is the base case Q_2. This tensor product formula is known as the divide-and-conquer parallel prefix algorithm, and it can be stated as the following theorem.

Theorem 4.1 Divide-and-Conquer Parallel Prefix Algorithm $Q_{2^n} = \prod_{i=1}^{n}(I_{2^{n-i}} \otimes R_{2^i})$, for $n > 0$, where R_{2^i} is defined as in Eq. (3).

Proof: Base case: For $n = 1$, $Q_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = R_2$, by Eq. (3).
Induction step: Assume the induction hypothesis $Q_{2^k} = \prod_{i=1}^{k}(I_{2^{k-i}} \otimes R_{2^i})$, for $k > 0$. We obtain

$$
\begin{aligned}
Q_{2^{k+1}} &= R_{2^{k+1}}(I_2 \otimes Q_{2^k}) && \text{(by the recursive formula)} \\
&= R_{2^{k+1}}\left(I_2 \otimes \prod_{i=1}^{k}(I_{2^{k-i}} \otimes R_{2^i})\right) && \text{(by the induction hypothesis)} \\
&= R_{2^{k+1}}\left(\prod_{i=1}^{k}(I_2 \otimes I_{2^{k-i}} \otimes R_{2^i})\right) && \text{(by Property (4))} \\
&= R_{2^{k+1}}\left(\prod_{i=1}^{k}(I_{2^{k+1-i}} \otimes R_{2^i})\right) && \text{(by the tensor product definition)} \\
&= \prod_{i=1}^{k+1}(I_{2^{k+1-i}} \otimes R_{2^i}).
\end{aligned}
$$

❑

Suppose we choose the post-operation matrix $R_{2^n} = I_{2^n}$ initially. Solving $Q_{2^n} = (I_2 \otimes Q_{2^{n-1}})P_{2^n}$, we obtain P_{2^n} as follows:

$$
P_{2^n} = \begin{bmatrix} I_{2^{n-1}} & 0_{2^{n-1}} \\ T_{2^{n-1}} & I_{2^{n-1}} \end{bmatrix}_{2^n \times 2^n}, \text{ where } T_{2^{n-1}} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \end{bmatrix}_{2^{n-1} \times 2^{n-1}}.
$$

It is clear that the definition of $T_{2^{n-1}}$ requires a sequence of addition operations that can not be parallelized. Therefore, it is considered to be an insignificant solution.

By Theorem 4.1, we know that the divide-and-conquer parallel prefix algorithm of 2^n elements can be completed in n steps. That is, its time complexity is $O(\log N)$, for $N = 2^n$. This tensor product formula can be interpreted as follows. The matrix product $\prod_{i=1}^{n}$ is mapped to a definite loop of n iterations. The right most operator, $I_{2^{n-1}} \otimes R_2$, is then applied to the input data first. Each step $I_{2^{n-1}} \otimes R_{2^i}$ means 2^{n-i} copies of R_{2^i} are performed in parallel. Operation R_{2^i} is applied to an input data segment of 2^i elements. In each segment, the last element of the first half is added to every element of the second half. Hence, the tensor product formula in Theorem 4.1 can be implemented in a shared-memory multiprocessor as follows:

```
for i = 1 to n
   forall j = 1 to 2**(n – i)
      forall k = 1 to 2**(i – 1)
         y[(j – 1)*2**i + 2**(i – 1) + k – 1]
         = x[(j – 1)2**i + 2**(i – 1) + k – 1] + x[(j – 1)2**i + 2**(i – 1) – 1]
      end forall
   end forall
   swap (x, y)
end for
```

The shared-memory code is similar to a sequential code on a single processor. The difference is in the *forall* statement. Since we are interested in the generation of distributed memory multiprocessors programs, we will show the shared-memory program in this subsection only. For distributed memory multiprocessors, we assume that data element x_i is allocated to processor p_i. The SPMD program of Theorem 4.1 is given in the following:

```
for i = 1 to n
   offset = my_rank – (my_rank mod 2**i)
   if (my_rank mod 2**i = 2**(i – 1) – 1) then
      target_processors = [offset + 2**(i – 1) : offset + 2**i – 1]
      multicast (target_processors, x)
   end if
   if (my_rank mod 2**i > 2**(i – 1) – 1) then
      receive (offset + 2**(i – 1) – 1, y)
      x = y + x
   end if
end for
```

The computation sequence of the divide-and-conquer parallel prefix algorithm, for $N = 8$, $Q_8 = R_8(I_2 \otimes R_4)(I_4 \otimes R_2)$, is illustrated in Fig. 1. The points on the horizontal axis denote processors p_0 to p_7, and each processor contains one data element. The solid lines are paths of data movement. In a given step, if a processor has two input lines, its data value is replaced by the sum of the data elements corresponding to the two input lines; otherwise, its data value remains the same.

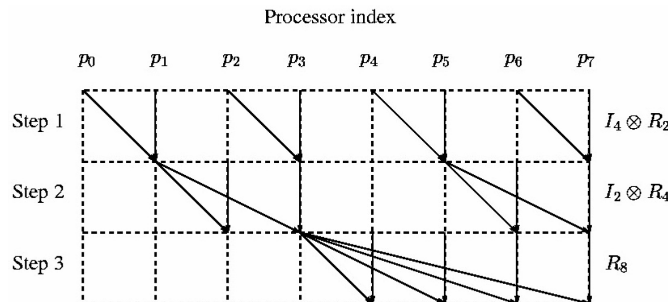

Fig. 1. Divide-and-conquer parallel prefix algorithm, for $N = 8$.

4.2 Recursive Doubling Algorithm

We select the vector form of Eq. (2) to derive the recursive doubling parallel prefix algorithm. Initially, assuming that the output data are stored in the nature order, we set post-operation R_N to the identity matrix I_N. Let N be 2^n. We obtain the matrix equation $Q_{2^n} = (Q_{2^{n-1}} \otimes I_2)P_{2^n}$. Then, matrix P_{2^n} can be solved as follows:

Define T_{2^n} as the $2^n \times 2^n$ matrix with Q_2 in the diagonal blocks, $1_2 - Q_2$ below the diagonal blocks, and 0's elsewhere, i.e.,

$$T_{2^n} = \begin{bmatrix} Q_2 & 0 & \cdots & 0 & 0 \\ 1_2 - Q_2 & Q_2 & 0 & \cdots & 0 \\ 0 & 1_2 - Q_2 & Q_2 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1_2 - Q_2 & Q_2 \end{bmatrix}_{2^n \times 2^n} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & & 0 \\ 0 & 1 & 1 & & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^n \times 2^n} .$$

Since

$$(Q_{2^{n-1}} \otimes I_2)T_{2^n} = \begin{bmatrix} I_2 & 0 & \cdots & 0 & 0 \\ I_2 & I_2 & 0 & \cdots & 0 \\ I_2 & I_2 & I_2 & \cdots & 0 \\ & & \ddots & & \\ I_2 & I_2 & \cdots & I_2 & I_2 \end{bmatrix}_{2^n \times 2^n} \begin{bmatrix} Q_2 & 0 & \cdots & 0 & 0 \\ 1_2 - Q_2 & Q_2 & 0 & \cdots & 0 \\ 0 & 1_2 - Q_2 & Q_2 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1_2 - Q_2 & Q_2 \end{bmatrix}_{2^n \times 2^n}$$
$$= Q_{2^n},$$

we obtain the following result:

$$P_{2^n} = (Q_{2^{n-1}} \otimes I_2)^{-1} Q_{2^n}$$

(multiply $(Q_{2^{n-1}} \otimes I_2)^{-1}$ on both side of the matrix equation)

$$= (Q_{2^{n-1}} \otimes I_2)^{-1}(Q_{2^{n-1}} \otimes I_2)T_{2^n}$$
$$= T_{2^n}$$
$$= \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^n \times 2^n} .$$

Solving the equation recursively, we derive the tensor product formula $Q_{2^n} = \prod_{i=n}^{1} (P_{2^i} \otimes I_{2^{n-i}})$ such that

$$P_{2^i} = (Q_{2^{i-1}} \otimes I_2)^{-1} Q_{2^i} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^i \times 2^i}, \text{ for } 0 < i \le n. \quad (4)$$

Note that the elements on and below the diagonal of P_{2^i} are 1's and 0's elsewhere. This tensor product formulation is known as the recursive-doubling parallel prefix algorithm, and it can be stated as the following theorem.

Theorem 4.2 Recursive-Doubling Parallel Prefix Algorithm $Q_{2^n} = \prod_{i=n}^{1}(P_{2^i} \otimes I_{2^{n-i}})$, for $n > 0$, where P_{2^i} is defined as in Eq. (4).

Proof: Base case: For $n = 1$, $Q_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = P_2$, by Eq. (4).

Induction step: Assume the induction hypothesis $Q_{2^k} = \prod_{i=1}^{1}(P_{2^i} \otimes I_{2^{k-i}})$, for $k > 0$. We obtain

$$
\begin{aligned}
Q_{2^{k+1}} &= (Q_{2^k} \otimes I_2)P_{2^{k+1}} && \text{(by the recursive formula)} \\
&= \left(\left(\prod_{i=k}^{1}(P_{2^i} \otimes I_{2^{k-i}}) \right) \otimes I_2 \right) P_{2^{k+1}} && \text{(by the induction hypothesis)} \\
&= \left(\prod_{i=k}^{1}((P_{2^i} \otimes I_{2^{k-i}}) \otimes I_2) \right) P_{2^{k+1}} && \text{(by Property (5))} \\
&= \left(\prod_{i=k}^{1}(P_{2^i} \otimes I_{2^{k+1-i}}) \right) P_{2^{k+1}} && \text{(by the tensor product definition)} \\
&= \prod_{i=k+1}^{1}(P_{2^i} \otimes I_{2^{k+1-i}}).
\end{aligned}
$$

❑

By Theorem 4.2, we know that the recursive-doubling parallel prefix algorithm of 2^n elements can be completed in n steps. That is, its time complexity is $O(\log N)$, for $N = 2^n$. This tensor product formula can be interpreted as follows. The matrix product $\prod_{i=n}^{1}$ is mapped to a definite loop of n iterations with loop variable i from n down to 1. The right most operator, P_{2^n}, is then applied to the input data first. In iteration i, each of the processors, except the first 2^{n-i} ones, performs an addition operation. For distributed memory multiprocessors, we assume that data element x_i is allocated to processor p_i. The SPMD program of Theorem 4.2 is given in the following:

```
for i = 1 to n
   distance = 2**(i – 1)
   if (my_rank < 2**n – distance) then
      send (processors [my_rank + distance], x [my_rank])
   end if
   if (my_rank > distance – 1) then
      receive (processor [my_rank – distance], data)
      x [my_rank] = x [my_rank] + data
   end if
end for
```

The computation sequence of the recursive-doubling parallel prefix algorithm, for $N = 8$, $Q_8 = (P_2 \otimes I_4)(P_4 \otimes I_2)P_8$, is illustrated in Fig. 2.

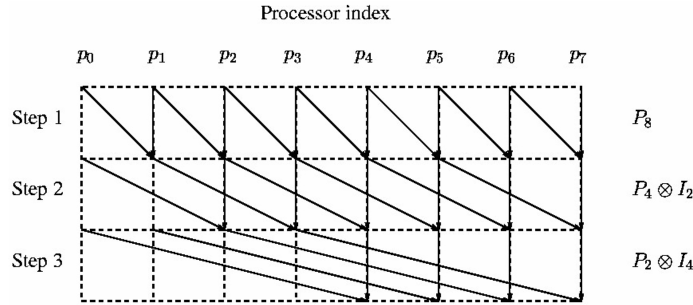

Fig. 2. Recursive-doubling parallel prefix algorithm, for $N = 8$.

4.3 Reversed Recursive-Doubling Algorithm

Since both Eqs. (1) and (2) have many solutions, we can use various strategies to exploit alternative tensor product formulas. In the derivation of the recursive-doubling parallel prefix algorithm, we set post-operation R_N to the identity matrix. Alternatively, we start with the recursive vector form of Eq. (2) and set pre-operation P_N to the identity matrix I_N. Let N be 2^n. We obtain the matrix equation $Q_{2^n} = R_{2^n}(Q_{2^{n-1}} \otimes I_2)$. Then, matrix R_{2^n} can be solved as follows:

$$R_{2^n} = Q_{2^n}(Q_{2^{n-1}} \otimes I_2)^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^n \times 2^n} .$$

Solving the equation recursively, we derive the tensor product formula $Q_{2^n} = \prod_{i=1}^{n}(R_{2^i} \otimes I_{2^{n-i}})$, such that

$$R_{2^i} = Q_{2^i}(Q_{2^{i-1}} \otimes I_2)^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^i \times 2^i}, \text{ for } 0 < i \le n. \quad (5)$$

Note that the elements on and below the diagonal of R_{2^i} are 1's and 0's elsewhere. We call this tensor product formula the reversed recursive-doubling parallel prefix algorithm and state it in the following theorem.

Theorem 4.3 Reversed Recursive-Doubling Parallel Prefix Algorithm $Q_{2^n} = \prod_{i=1}^{n}(R_{2^i} \otimes I_{2^{n-i}})$, for $n > 0$, where R_{2^i} is defined as in Eq. (5).

Proof: Base case: For $n = 1$, $Q_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = R_2$, by Eq. (5).
Induction step: Assume induction hypothesis $Q_{2^k} = \prod_{i=1}^{k}(R_{2^i} \otimes I_{2^{k-i}})$, for $k > 0$. We obtain

$$
\begin{aligned}
Q_{2^{k+1}} &= R_{2^{k+1}}(Q_{2^k} \otimes I_2) && \text{(by the recursive formula)} \\
&= R_{2^{k+1}}\left(\left(\prod_{i=1}^{k}(R_{2^i} \otimes I_{2^{k-i}})\right) \otimes I_2\right) && \text{(by the induction hypothesis)} \\
&= R_{2^{k+1}}\left(\prod_{i=1}^{k}((R_{2^i} \otimes I_{2^{k-i}}) \otimes I_2)\right) && \text{(by Property (5))} \\
&= R_{2^{k+1}}\left(\prod_{i=1}^{k}(R_{2^i} \otimes I_{2^{k+1-i}})\right) && \text{(by the tensor product definition)} \\
&= \prod_{i=1}^{k+1}(R_{2^i} \otimes I_{2^{k+1-i}}).
\end{aligned}
$$

❑

By Theorem 4.3, we know that the reversed recursive-doubling parallel prefix algorithm of 2^n elements can be completed in n steps. That is, its time complexity is $O(\log N)$, for $N = 2^n$. This tensor product formula can be interpreted as follows. The matrix product $\prod_{i=1}^{n}$ is mapped to a definite loop of n iterations with loop variable i from 1 to n. The right most operator, $R_2 \otimes I_{2^{n-1}}$, is then applied to the input data first. In iteration i, each of the processors, except the first 2^{n-i} ones, performs an addition operation. The computation sequence is exactly the reverse of the order of the recursive-doubling parallel prefix algorithm. For distributed-memory multiprocessors, we assume that data element x_i is allocated to processor p_i. The SPMD program of Theorem 4.3 is given in the following:

```
for i = 1 to n
   distance = 2**(n – i)
   if (my_rank < 2**n – distance) then
      send (processors [my_rank + distance], x [my_rank])
   end if
   if (my_rank > distance – 1) then
      receive (processor [my_rank – distance], data)
      x [my_rank] = x [my_rank] + data
   end if
end for
```

The computation sequence of the reversed recursive dobuling parallel prefix algorithm, for $N = 8$, $Q_8 = R_8(R_4 \otimes I_2)(R_2 \otimes I_4)$, is illustrated in Fig. 3.

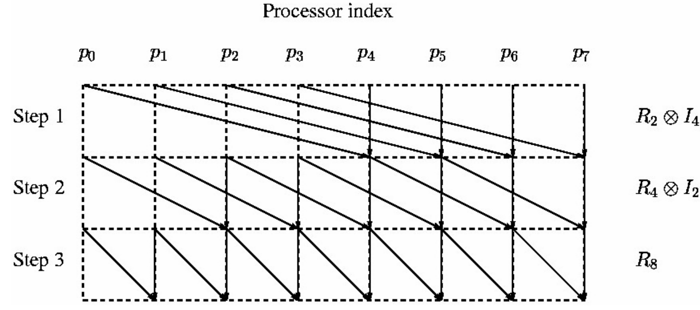

Fig. 3. Reversed recursive-doubling parallel prefix algorithm, for $N = 8$.

4.4 Parallel Prefix Algorithms with Bit Reversal Permutation

In some applications, the input or output data may be allocated in an order different from the natural order. We will demonstrate the methodology by deriving two variants of the parallel prefix algorithm. The two algorithms are the input data and the output data applied through bit reversal permutation, respectively, before and after parallel prefix operations.

4.4.1 Input data rearranged by bit reversal permutation

To derive the parallel prefix with input data rearranged through bit reversal permutation, we start with the recursive parallel form of Eq. (1) and set pre-operation P_N to the stride permutation $L_2^{2^n}$. Let N be 2^n. We obtain the matrix equation $Q_{2^n} = R_{2^n}(I_2 \otimes Q_{2^{n-1}}) L_2^{2^n}$. Then, matrix R_{2^n} can be solved follows:

$$
\begin{aligned}
R_{2^n} &= Q_{2^n} (L_2^{2^n})^{-1} (I_2 \otimes Q_{2^{n-1}})^{-1} \\
&= Q_{2^n} (L_2^{2^n})^{-1} (I_2 \otimes Q_{2^{n-1}})^{-1} (L_{2^{n-1}}^{2^n})^{-1} L_{2^{n-1}}^{2^n} \\
&= Q_{2^n} ((L_{2^{n-1}}^{2^n})(I_2 \otimes Q_{2^{n-1}})(L_2^{2^n}))^{-1} L_{2^{n-1}}^{2^n} \\
&= Q_{2^n} (Q_{2^{n-1}} \otimes I_2)^{-1} L_{2^{n-1}}^{2^n} \qquad \text{(by Property (10))} \\
&= S_{2^n} L_{2^{n-1}}^{2^n},
\end{aligned}
$$

where

$$
S_{2^n} = Q_{2^n} (Q_{2^{n-1}} \otimes I_2)^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^n \times 2^n}.
$$

Solving the equation recursively, we can derive the tensor product formula

$$Q_{2^n} = \left(\prod_{i=1}^{n}((I_{2^{n-i}} \otimes S_{2^i})(I_{2^{n-i}} \otimes L_{2^{i-1}}^{2^i}))\right)\left(\prod_{i=n}^{1}(I_{2^{n-i}} \otimes L_2^{2^i})\right)$$

such that

$$S_{2^i} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^i \times 2^i}, \text{ for } 0 < i \le n. \quad (6)$$

The tensor product formulation of the parallel prefix algorithm with the pre-bit reversal operation can be stated as the following lemma.

Lemma 4.1 $Q_{2^n} = \left(\prod_{i=1}^{n}((I_{2^{n-i}} \otimes S_{2^i})(I_{2^{n-i}} \otimes L_{2^{i-1}}^{2^i}))\right)\left(\prod_{i=n}^{1}(I_{2^{n-i}} \otimes L_2^{2^i})\right)$, for $n > 0$, where S_{2^i} is defined as in Eq. (6).

Proof: Base case: For $n = 1$, $Q_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = R_2$, by Eq. (6).
Induction step: Assume induction hypothesis $Q_{2^k} = \left(\prod_{i=1}^{k}((I_{2^{k-i}} \otimes S_{2^i})(I_{2^{k-i}} \otimes L_{2^{i-1}}^{2^i}))\right)\left(\prod_{i=k}^{1}(I_{2^{k-i}} \otimes L_2^{2^i})\right)$. for $k > 0$. We obtain

$$\begin{aligned} Q_{2^{k+1}} &= R_{2^{k+1}}(I_2 \otimes Q_{2^k})(L_2^{2^{k+1}}) && \text{(by the recursive formula)} \\ &= S_{2^{k+1}} L_{2^k}^{2^{k+1}} \left(I_2 \otimes \left(\prod_{i=1}^{k}((I_{2^{k-i}} \otimes S_{2^i})(I_{2^{k-i}} \otimes L_{2^{i-1}}^{2^i}))\right)\left(\prod_{i=k}^{1}(I_{2^{k-i}} \otimes L_2^{2^i})\right)\right) L_2^{2^{k+1}} \\ & && \text{(by the induction hypothesis)} \\ &= S_{2^{k+1}} L_{2^k}^{2^{k+1}} \left(\prod_{i=1}^{k}((I_{2^{k+1-i}} \otimes S_{2^i})(I_{2^{k+1-i}} \otimes L_{2^{i-1}}^{2^i}))\right)\left(\prod_{i=k}^{1}(I_{2^{k+1-i}} \otimes L_2^{2^i})\right) L_2^{2^{k+1}} \\ & && \text{(by Property (4))} \\ &= \left(\prod_{i=1}^{k+1}((I_{2^{k+1-i}} \otimes S_{2^i})(I_{2^{k+1-i}} \otimes L_{2^{i-1}}^{2^i}))\right)\left(\prod_{i=k+1}^{1}(I_{2^{k+1-i}} \otimes L_2^{2^i})\right). \end{aligned}$$

❑

Note that the tensor product formula in Lemma 4.1 assumes that the input data are stored in the natural order. Permutation $\prod_{i=n}^{1}(I_{2^{n-i}} \otimes L_2^{2^i})$ applied to the input data is precisely the bit reversal permutation. If the prefix computation is performed following another computation, e.g., fast Fourier transform with decimation-in-frequency, the input

data will be rearranged in the bit reversal order. From Lemma 4.1, we obtain the following parallel prefix algorithm with input data stored in the bit reversal order.

Theorem 4.4 Parallel Prefix Algorithm with Pre-Bit Reversal Operation Let the input data be stored in the bit reversal order. The parallel prefix computation can be expressed as the following formula:

$$\tilde{Q}_{2^n} = \prod_{i=1}^{n} ((I_{2^{n-i}} \otimes S_{2^i})(I_{2^{n-i}} \otimes L_{2^{i-1}}^{2^i})),$$

for $n > 0$, where S_{2^i} is defined as in Eq. (6).

The matrix product $\prod_{i=1}^{n} ((I_{2^{n-i}} \otimes S_{2^i})(I_{2^{n-i}} \otimes L_{2^{i-1}}^{2^i}))$ requires n sequential steps, and each step needs two communication operations and one computation operation if each processor contains only one element. That is, the time complexity of this algorithm is still $O(\log N)$, for $N = 2^n$. For distributed-memory multiprocessors, we assume that data element x_i is allocated to processor p_i. The SPMD program of Theorem 4.4 is given in the following:

```
for i = 1 to n
   offset = my_rank – (my_rank mod 2**i)
   send (processors [stride(2**i, 2, my_rank – offset) + offset], x [my_rank])
   receive (processor [stride(2**i, 2**(i – 1), my_rank – offset) + offset], x [my_rank])
   if (my_rank – offset < 2**i) then
      send (processor [my_rank +1], x [my_rank])
   end if
   if (my_rank – offset > 0) then
      receive (processor [my_rank – 1], data)
      x [my_rank] = x [my_rank] + data
   end if
end for
```

In the above program, the two function calls stride(2**i, 2, my_rank – offset) and stride(2**i, 2**(i – 1), my_rank – offset) are index computations derived from stride permutations $L_2^{2^i}$ and $L_{2^{i-1}}^{2^i}$, respectively. Their resulting values are

```
stride(2**i, 2, my_rank – offset) =
   (my_rank – offset) div 2                        if my_rank – offset is even
   2**(i – 1) + (my_rank – offset – 1) div 2       otherwise
stride(2**i, 2**(i – 1), my_rank – offset) =
   (my_rank – offset) * 2                          if my_rank – offset < 2**(i – 1)
   ((my_rank – offset) – 2**(i – 1) * 2 + 1)       otherwise
```

The computation sequence of the parallel prefix algorithm with input data stored in the bit reversal order, for $N = 8$, $\tilde{Q}_8 = S_8 L_4^8 (I_2 \otimes S_4)(I_2 \otimes L_2^4)(I_4 \otimes S_2)$, is illustrated in Fig. 4.

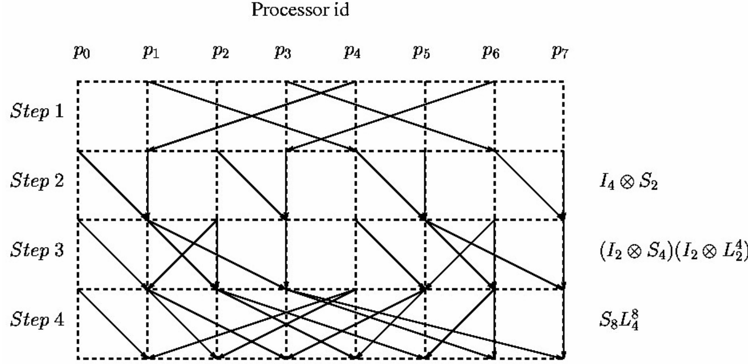

Fig. 4. Parallel prefix algorithm with input data stored in the bit reversal order, for $N = 8$.

4.4.2 Output data rearranged by means of bit reversal permutation

To derive the parallel prefix with output data rearranged through bit reversal permutation, we start with the recursive parallel form of Eq. (1) and set post-operation R_N to the stride permutation $L_{2^{n-1}}^{2^n}$. Let N be 2^n. We obtain the matrix equation $Q_{2^n} = L_{2^{n-1}}^{2^n}(I_2 \otimes Q_{2^{n-1}})P_{2^n}$. Then, matrix P_{2^n} can be solved as follows:

$$\begin{aligned}
P_{2^n} &= (I_2 \otimes Q_{2^{n-1}})^{-1}(L_{2^{n-1}}^{2^n})^{-1}Q_{2^n} \\
&= L_2^{2^n} L_2^{2^{n-1}} (I_2 \otimes Q_{2^{n-1}})^{-1}(L_{2^{n-1}}^{2^n})^{-1}Q_{2^n} \\
&= L_2^{2^n} (L_{2^{n-1}}^{2} (I_2 \otimes Q_{2^{n-1}}) L_2^{2^n})^{-1} Q_{2^n} \\
&= L_2^{2^n} (Q_{2^{n-1}} \otimes I_2)^{-1} Q_{2^n} \qquad \text{(by Property(10))} \\
&= L_2^{2^n} S_{2^n},
\end{aligned}$$

where

$$S_{2^n} = (Q_{2^{n-1}} \otimes I_2)^{-1} Q_{2^n} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & \cdots & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}_{2^n \times 2^n}.$$

Solving the equation recursively, we can derive the tensor product formula

$$Q_{2^n} = \left(\prod_{i=n}^{1} (I_{2^{i-1}} \otimes L_{2^{n-i}}^{2^{n-i+1}}) \right) \left(\prod_{i=n}^{1} ((I_{2^{n-i}} \otimes L_2^{2^i})(I_{2^{n-i}} \otimes S_{2^i})) \right)$$

such that S_{2^i} is defined as Eq. (6). The tensor product formulation of the parallel prefix algorithm with the post-bit reversal operation can be stated as the following lemma.

Lemma 4.2 $Q_{2^n} = \left(\prod_{i=n}^{1}(I_{2^{i-1}} \otimes L_{2^{n-i}}^{2^{n-i+1}})\right)\left(\prod_{i=n}^{1}((I_{2^{n-i}} \otimes L_2^{2^i})(I_{2^{n-i}} \otimes S_{2^i}))\right)$, for $n > 0$, where S_{2^i} is defined as in Eq. (6).

Proof: Base case: For $n = 1$, $Q_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = P_2$, by Eq. (6)

Induction step: Assume induction hypothesis $Q_{2^k} = \left(\prod_{i=k}^{1}(I_{2^{i-1}} \otimes L_{2^{k-i}}^{2^{k-i+1}})\right)\left(\prod_{i=k}^{1}((I_{2^{k-i}} \otimes L_2^{2^i})(I_{2^{k-i}} \otimes S_{2^i}))\right)$, for $k > 0$. We obtain

$$Q_{2^{k+1}} = L_{2^k}^{2^{k+1}}(I_2 \otimes Q_{2^k})P_{2^{k+1}} \qquad \text{(by the recursive formula)}$$

$$= L_{2^k}^{2^{k+1}}\left(I_2 \otimes \left(\left(\prod_{i=k}^{1}(I_{2^{i-1}} \otimes L_{2^{k-i}}^{2^{k-i+1}})\right)\left(\prod_{i=k}^{1}((I_{2^{k-i}} \otimes L_2^{2^i})(I_{2^{k-i}} \otimes S_{2^i}))\right)\right)\right)L_2^{2^{k+1}}S_{2^{k+1}}$$

(by the induction hypothesis)

$$= L_{2^k}^{2^{k+1}}\left(\left(\prod_{i=k}^{1}(I_{2^i} \otimes L_{2^{k-i}}^{2^{k-i+1}})\right)\left(\prod_{i=k}^{1}((I_{2^{k-i+1}} \otimes L_2^{2^i})(I_{2^{k-i+1}} \otimes S_{2^i}))\right)\right)L_2^{2^{k+1}}S_{2^{k+1}}$$

(by Property (4))

$$= \left(\prod_{i=k+1}^{1}(I_{2^{i-1}} \otimes L_{2^{k-i+1}}^{2^{k-i+2}})\right)\left(\prod_{i=k+1}^{1}((I_{2^{k-i+1}} \otimes L_2^{2^i})(I_{2^{k-i+1}} \otimes S_{2^i}))\right). \qquad ❑$$

Note that the tensor product formula in Lemma 4.2 assumes that the output data are stored in the natural order. Permutation $\prod_{i=n}^{1}(I_{2^{i-1}} \otimes L_{2^{n-i}}^{2^{n-i+1}})$ applied to the output data is precisely the bit reversal permutation. If the prefix computation is performed followed another computation, e.g., fast Fourier transform with decimation-in-time, then the output data will be rearranged in the bit reversal order. From Lemma 4.2, we obtain the following parallel prefix algorithm with output data stored in the bit reversal order.

Theorem 4.5 Parallel Prefix Algorithm with Post-Bit Reversal Operation Let the output data be stored in the bit reversal order. The parallel prefix computation can be expressed as in the following formula:

$$\tilde{Q}_{2^n} = \prod_{i=n}^{1}((I_{2^{n-i}} \otimes L_2^{2^i})(I_{2^{n-i}} \otimes S_{2^i})),$$

for $n > 0$, where S_{2^i} is defined as in Eq. (6).

The matrix product $\prod_{i=n}^{1}((I_{2^{n-i}} \otimes L_2^{2^i})(I_{2^{n-i}} \otimes S_{2^i}))$ requires n sequential steps, and each step needs two communication operations and one computation operation if each processor contains only one element. That is, the time complexity of this algorithm is still $O(\log N)$, for $N = 2^n$. For distributed-memory multiprocessors, we assume that data element x_i is allocated to processor p_i. The SPMD program of Theorem 4.5:

```
for i = n to 1
   offset = my_rank − (my_rank mod 2**i)
   if (my_rank − offset < 2**i) then
      send (processor [my_rank + 1], x [my_rank])
   end if
   if (my_rank − offset > 0) then
      receive (processor [my_rank − 1], data)
      x [my_rank] = x [my_rank] + data
   end if
   send (processor [stride(2**i, 2**(i − 1), my_rank − offset) + offset], x [my_rank])
   receive (processor [stride(2**i, 2, my_rank − offset) + offset], x [my_rank])
end for
```

The computation sequence of the parallel prefix algorithm with output data stored in the bit reversal order, for $N = 8$, $\tilde{Q}_8 = (I_4 \otimes S_2)(I_2 \otimes L_2^4)(I_2 \otimes S_4)L_2^8 S_8$, is illustrated in Fig. 5.

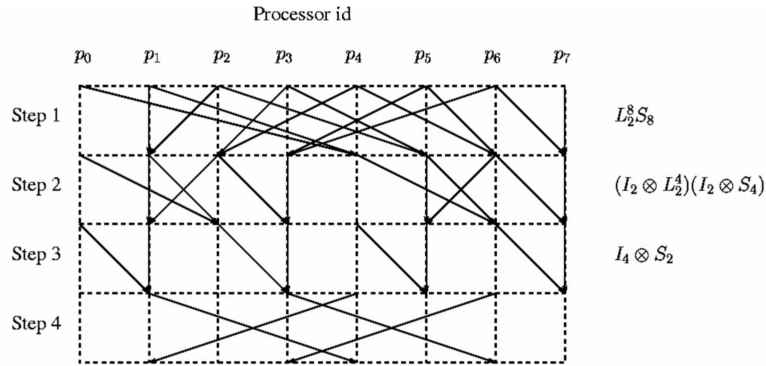

Fig. 5. Parallel prefix algorithm with output data stored in the bit reversal order, for $N = 8$.

5. COOLEY-TUKEY'S FAST FOURIER TRANSFORM

The discrete Fourier transform can be defined as $F_N x_i^N = y_i^N$, where $y_i^N = \sum_{j=0}^{N-1} \omega^{ij} x_j^N$ and $\omega^k = e^{2k\pi i/n}$. The matrix form of F_N is as the follows:

$$F_N = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{N-1} \\ \cdots & \cdots & \omega^{ij} & \cdots \\ 1 & \omega^{N-1} & \cdots & \omega^{(N-1)^2} \end{bmatrix}_{N \times N}.$$

In this section, we will derive two fast Fourier transform (FFT) algorithms using the proposed methodology. The algorithms are Cooley-Tukey's FFT decimation-in-time and decimation-in-frequency algorithms.

5.1 Cooley-Tukey's FFT Decimation-in-Time Algorithm

To derive Cooley-Tukey's FFT decimation-in-time algorithm, we start with the recursive parallel form of Eq. (1) and set pre-operation P_N to the stride permutation matrix L_2^N. Let N be 2^n. We obtain the matrix equation $F_{2^n} = R_{2^n}(I_2 \otimes F_{2^{n-1}}) L_2^{2^n}$. Matrix R_{2^n} can be solved as follows:

$$\begin{aligned} R_{2^n} &= F_{2^n} L_{2^{n-1}}^{2^n} (I_2 \otimes F_{2^{n-1}})^{-1} \\ &= \begin{bmatrix} I_{2^{n-1}} & diag(1,\ \omega,\ ...,\ \omega^{2^{n-1}-1}) \\ I_{2^{n-1}} & diag(-1,\ -\omega,\ ...,\ -\omega^{2^{n-1}-1}) \end{bmatrix} \\ &= \begin{bmatrix} I_{2^{n-1}} & I_{2^{n-1}} \\ I_{2^{n-1}} & -I_{2^{n-1}} \end{bmatrix} \begin{bmatrix} I_{2^{n-1}} & 0 \\ 0 & diag(1,\ \omega,\ ...,\ \omega^{2^{n-1}-1}) \end{bmatrix}. \end{aligned}$$

Then, we have the recursive formula of Cooley-Tukey's FFT decimation-in-time algorithm:

$$\begin{aligned} F_{2^n} &= \begin{bmatrix} I_{2^{n-1}} & I_{2^{n-1}} \\ I_{2^{n-1}} & -I_{2^{n-1}} \end{bmatrix} \begin{bmatrix} I_{2^{n-1}} & \\ & diag(1,\ \omega,\ ...,\ \omega^{2^{n-1}-1}) \end{bmatrix} (I_2 \otimes F_{2^{n-1}}) L_2^{2^n} \\ &= (F_2 \otimes I_{2^{n-1}}) T_{2^{n-1}}^{2^n} (I_2 \otimes F_{2^{n-1}}) L_2^{2^n}, \end{aligned}$$

where $T_{2^{n-1}}^{2^n}$ is the twiddle factor of FFT. Solving the equation recursively, we can derive the tensor product formula

$$F_{2^n} = \left(\prod_{i=0}^{n-1} (I_{2^{n-1-i}} \otimes F_2 \otimes I_{2^i})(I_{2^{n-1-i}} \otimes T_{2^i}^{2^{i+1}}) \right) \left(\prod_{i=0}^{n-1} I_{2^i} \otimes L_2^{2^{n-i}} \right),$$

where

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{ and } T_{2^i}^{2^{i+1}} = \begin{bmatrix} I_{2^i} & 0 \\ 0 & diag(1,\ \omega,\ ...,\ \omega^{2^i-1}) \end{bmatrix}, \text{ for } 0 \le i < n. \quad (7)$$

Note that F_2 is a butterfly operation and $T_{2^i}^{2^{i+1}}$ is a diagonal matrix. The tensor product formulation of the Cooley-Tukey's FFT Decimation-in-Time algorithm can be stated as the following theorem. Note that this theorem is the same as that given in [9].

Theorem 5.1 Cooley-Tukey's FFT Decimation-in-Time $F_{2^n} = \left(\prod_{i=0}^{n-1}(I_{2^{n-1-i}} \otimes F_2 \otimes I_{2^i})(I_{2^{n-1-i}} \otimes T_{2^i}^{2^{i+1}})\right)\left(\prod_{i=0}^{n-1} I_{2^i} \otimes L_2^{2^{n-i}}\right)$, for $n > 0$, where $T_{2^i}^{2^{i+1}}$ is defined as in Eq. (7).

Proof: Base case: For $n = 1$, $F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, by Eq. (7).
Induction step: Assume induction hypothesis $F_{2^k} = \left(\prod_{i=0}^{k-1}(I_{2^{k-1-i}} \otimes F_2 \otimes I_{2^i})(I_{2^{k-1-i}} \otimes T_{2^i}^{2^{i+1}})\right)\left(\prod_{i=0}^{k-1} I_{2^i} \otimes L_2^{2^{k-i}}\right)$, for $k > 0$. We obtain

$$F_{2^{k+1}} = (F_2 \otimes I_{2^k})T_{2^k}^{2^{k+1}}(I_2 \otimes F_{2^k})L_2^{2^{k+1}} \quad \text{(by the recursive formula)}$$

$$= (F_2 \otimes I_{2^k})T_{2^k}^{2^{k+1}}\left(I_2 \otimes \left(\left(\prod_{i=0}^{k-1}(I_{2^{k-1-i}} \otimes F_2 \otimes I_{2^i})(I_{2^{k-1-i}} \otimes T_{2^i}^{2^{i+1}})\right)\left(\prod_{i=0}^{k-1} I_{2^i} \otimes L_2^{2^{k-i}}\right)\right)\right)L_2^{2^{k+1}}$$

(by the induction hypothesis)

$$= (F_2 \otimes I_{2^k})T_{2^k}^{2^{k+1}}\left(\prod_{i=0}^{k-1}(I_{2^{k-i}} \otimes F_2 \otimes I_{2^i})(I_{2^{k-i}} \otimes T_{2^i}^{2^{i+1}})\right)\left(\prod_{i=0}^{k-1} I_{2^i} \otimes L_2^{2^{k-i}}\right)L_2^{2^{k+1}}$$

(by Property (4))

$$= \left(\prod_{i=0}^{k}(I_{2^{k-i}} \otimes F_2 \otimes I_{2^i})(I_{2^{k-i}} \otimes T_{2^i}^{2^{i+1}})\right)\left(\prod_{i=0}^{k} I_{2^i} \otimes L_2^{2^{k+1-i}}\right). \qquad \square$$

5.2 Cooley-Tukey's FFT Decimation-in-Frequency Algorithm

To derive Cooley-Tukey's FFT decimation-in-frequency algorithm, we start with the recursive parallel form of Eq. (1) and set post-operation R_N to the stride permutation matrix $L_{N/2}^N$. Let N be 2^n. We obtain the matrix equation $F_{2^n} = L_{2^{n-1}}^{2^n}(I_2 \otimes F_{2^{n-1}})P_{2^n}$. Matrix P_{2^n} can be solved as follows:

$$P_{2^n} = (I_2 \otimes F_{2^{n-1}})^{-1}(L_{2^{n-1}}^{2^n})^{-1}F_{2^n}$$

$$= \begin{bmatrix} I_{2^{n-1}} & I_{2^{n-1}} \\ diag(1,\ \omega,\ ...,\ \omega^{2^{n-1}-1}) & diag(-1,\ -\omega,\ ...,\ -\omega^{2^{n-1}-1}) \end{bmatrix}$$

$$= \begin{bmatrix} I_{2^{n-1}} & 0 \\ 0 & diag(1,\ \omega,\ ...,\ \omega^{2^{n-1}-1}) \end{bmatrix}\begin{bmatrix} I_{2^{n-1}} & I_{2^{n-1}} \\ I_{2^{n-1}} & -I_{2^{n-1}} \end{bmatrix}$$

Then, we have the recursive formula of Cooley-Tukey's FFT decimation-in-frequency algorithm:

$$F_{2^n} = L^{2^n}_{2^{n-1}}(I_2 \otimes F_{2^{n-1}}) \begin{bmatrix} I_{2^{n-1}} & 0 \\ 0 & diag(1,\ \omega,\ ...,\ \omega^{2^{n-1}-1}) \end{bmatrix} \begin{bmatrix} I_{2^{n-1}} & I_{2^{n-1}} \\ I_{2^{n-1}} & -I_{2^{n-1}} \end{bmatrix}$$
$$= L^{2^n}_{2^{n-1}}(I_2 \otimes F_{2^{n-1}}) T^{2^n}_{2^{n-1}} (F_2 \otimes I_{2^{n-1}}),$$

where $T^{2^n}_{2^{n-1}}$ is the twiddle factor of FFT. Solving the equation recursively, we derive the tensor product formula

$$F_{2^n} = \left(\prod_{i=1}^{n} (I_{2^{n-i}} \otimes L^{2^i}_{2^{i-1}}) \right) \left(\prod_{i=n-1}^{0} (I_{2^{n-i-1}} \otimes T^{2^{i+1}}_{2^i})(I_{2^{n-i-1}} \otimes F_2 \otimes I_{2^i}) \right),$$

where

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \text{ and } T^{2^{i+1}}_{2^i} = \begin{bmatrix} I_{2^i} & 0 \\ 0 & diag(1,\ \omega,\ ...,\ \omega^{2^i-1}) \end{bmatrix}, \text{ for } 0 \le i < n. \tag{8}$$

Note that F_2 is a butterfly operation and $T^{2^{i+1}}_{2^i}$ is a diagonal matrix. The tensor product formulation of Cooley-Tukey's FFT decimation-in-frequency algorithm can be stated as the following theorem.

Theorem 5.2 $F_{2^n} = \left(\prod_{i=1}^{n} (I_{2^{n-i}} \otimes L^{2^i}_{2^{i-1}}) \right) \left(\prod_{i=n-1}^{0} (I_{2^{n-i-1}} \otimes T^{2^{i+1}}_{2^i})(I_{2^{n-i-1}} \otimes F_2 \otimes I_{2^i}) \right)$, for $n >$ 0, where $T^{2^{i+1}}_{2^i}$ is defined as in Eq (8).

Proof: Base case: For $n = 1$, $F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, by Eq. (8).
Induction step: Assume induction hypothesis $F_{2^k} = \left(\prod_{i=1}^{k} (I_{2^{k-i}} \otimes L^{2^i}_{2^{i-1}}) \right) \left(\prod_{i=k-1}^{0} (I_{2^{k-i-1}} \otimes \right.$ $\left. \otimes T^{2^{i+1}}_{2^i})(I_{2^{k-i-1}} \otimes F_2 \otimes I_{2^i}) \right)$, for $k > 0$. We obtain

$$F_{2^{k+1}} = L^{2^{k+1}}_{2^k}(I_2 \otimes F_{2^k}) T^{2^{k+1}}_{2^k} (F_2 \otimes I_{2^k}) \quad \text{(by the recursive formula)}$$
$$= L^{2^{k+1}}_{2^k} \left(I_2 \otimes \left(\prod_{i=1}^{k} (I_{2^{k-i}} \otimes L^{2^i}_{2^{i-1}}) \right) \left(\prod_{i=k-1}^{0} (I_{2^{k-i-1}} \otimes T^{2^{i+1}}_{2^i})(I_{2^{k-i-1}} \otimes F_2 \otimes I_{2^i}) \right) \right) T^{2^{k+1}}_{2} (F_2 \otimes I_{2^k})$$
$$\text{(by the induction hypothesis)}$$
$$= L^{2^{k+1}}_{2^k} \left(\prod_{i=1}^{k} (I_{2^{k-i+1}} \otimes L^{2^i}_{2^{i-1}}) \right) \left(\prod_{i=k-1}^{1} (I_{2^{k-i}} \otimes T^{2^{i+1}}_{2^i})(I_{2^{k-i}} \otimes F_2 \otimes I_{2^i}) \right) T^{2^{k+1}}_{2^k} (F_2 \otimes I_{2^k})$$
$$\text{(by Property (4))}$$
$$= \left(\prod_{i=1}^{k+1} (I_{2^{k-i+1}} \otimes L^{2^i}_{2^{i-1}}) \right) \left(\prod_{i=k}^{0} (I_{2^{k-i}} \otimes T^{2^{i+1}}_{2^i})(I_{2^{k-i}} \otimes F_2 \otimes I_{2^i}) \right). \qquad \square$$

Various algorithms for fast Fourier transform were discussed in [9]. In this paper, we have employed our methodology to derive Cooley-Tukey's FFT decimation-in-time and

decimation-in-frequency algorithms. Though the resulting tensor product formulas are the same, our goal is to demonstrate stepwise derivation of these algorithms using our programming methodology.

6. CONCLUSIONS

In this paper, we have presented a programming methodology for designing block recursive algorithms. We have employed the tensor product notation to formulate computational problems and derive different algorithms of given problems. Various parallel prefix algorithms and fast Fourier transform algorithms can be derived using this methodology. The algorithms generated using the methodology are represented as tensor product formulas. Then, the tensor product formulas can be mapped to parallel programs easily. The key idea behind the programming methodology is to factorize the matrix which represents a computational problem. The factors of the problem matrix should contain only matrix products and tensor products of simple matrix operations. The simple matrix operations can be easily implemented as simple programming statements.

Since a matrix equation obtained using this methodology may contain more unknown matrix variables than of matrix equations, we have some freedom to set matrix variables to certain operations and derive alternative algorithms for a given computational problem. This characteristic of the programming methodology may raise a question as to how initial matrix operations for unknown variables can be chosen. We provide heuristics with different strategies to deal with this problem.

Because tensor products are also used to model operational paradigms, memory hierarchies, and interconnection networks, the methodology can be extended to generate various algorithms for different computer architectures. For example, in a vector processor machine, we can derive algorithms of tensor product formulas in vector form; in a parallel machine, we can derive algorithms of tensor product formulas in parallel form to fit the architecture.

We have used the parallel prefix and FFT computation problem as examples to illustrate this methodology. Some of the derived algorithms are well known algorithms, and some are new algorithms with various types of data allocation. For the prefix computation problem, the intuitive solution takes $O(N)$ time. The well known parallel prefix algorithms, divide-and-conquer and recursive-doubling algorithms takes $O(\log N)$ time. The complexity of the newly derived algorithms are also $O(\log N)$ time. For the DFT computation problem, the solution takes $O(N^2)$ time. Cooley-Tukey's FFT algorithm takes $O(N\log N)$ time. Although fast Fourier transform algorithms using ad hoc optimization approaches such as the lifting scheme, have been reported [19], those algorithms have different sets of operations from Cooley-Tukey's FFT algorithm. This optimization issue is beyond the scope of this paper.

In our future work, we will apply this methodology to other computational problems, such as solving recurrence equations and digital signal processing algorithms. Furthermore, we will extend the design methodology by considering both algorithm and architecture characteristics using the tensor product notation.

REFERENCES

1. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics Computation*, Vol. 19, 1965, pp. 297-301.
2. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1991.
3. D. L. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C. H. Huang, P. Sadayappan, and R. W. Johnson, "EXTENT: a portable programming environment for designing and implementing high-performance block-recursive algorithms," in *Proceedings of Supercomputing '94*, 1994, pp. 49-58.
4. A. Graham, *Kronecker Products and Matrix Calculus: With Applications*, Ellis Horwood Limited, 1981.
5. S. K. S. Gupta, C. H. Huang, P. Sadayappan, and R. W. Johnson, "A framework for generating distributed-memory parallel programs for block recursive algorithms," *Journal of Parallel and Distributed Computing*, Vol. 34, 1996, pp. 137-153.
6. R. A. Horn and C. A. Johnson, *Topics in Matrix Analysis*, Cambridge University Press, Cambridge, 1991.
7. C. H. Huang, J. R. Johnson, and R. W. Johnson, "A tensor product formulation of Strassen's matrix multiplication algorithm," *Applied Math Letters*, Vol. 3, 1990, pp. 104-108.
8. C. H. Huang, J. R. Johnson, and R. W. Johnson, "Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm," in *Proceedings of the International Conference on Parallel Processing*, Vol. III, 1992, pp. III:104-108.
9. J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying and implementing Fourier transform algorithms on various architectures," *Circuits Systems Signal Process*, Vol. 9, 1990, pp. 450-500.
10. R. W. Johnson, C. H. Huang, and J. R. Johnson, "Multilinear algebra and parallel programming," *The Journal of Supercomputing*, Vol. 5, 1991, pp. 189-217.
11. S. D. Kaushik, S. Sharma, and C. H. Huang, "An algebraic theory for modeling multistage interconnection networks," *Journal of Information Science and Engineering*, Vol. 9, 1993, pp. 1-26.
12. S. D. Kaushik, S. Sharma, C. H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan, "An algebraic theory for modeling direct interconnection networks," *Journal of Information Science and Engineering*, Vol. 12, 1996, pp. 25-49.
13. D. G. Korn and J. J. Lambiotte, Jr., "Computing the fast Fourier transform on a vector computer," *Mathematics Computation*, Vol. 33, 1979, pp. 977-992.
14. B. Kumar, C. H. Huang, P. Sadayappan, and R. W. Johnson, "A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction," *Scientific Programming*, Vol. 4, 1995, pp. 275-289.
15. S. Y Lin, C. S Chen, L. Liu, and C. H Huang, "Tensor product formulation for Hilbert space-filling curves," in *Proceedings of the International Conference on Parallel Processing*, 2003, pp. 99-106.
16. C. B. Liu, C. H. Huang, and C. L. Lei, "Design and implementation of long-digit Karatsuba's multiplier using tensor product formulation," in *The 9th Workshop on Compiler Techniques for High-Performance Computing*, 2003, pp. 23-30.

17. T. Minkwitz, *Algorithms Explained by Symmetry*, LNCS 900, Springer-Verlag, 1995, pp. 157-167.
18. J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso, "SPIRAL: portable library of optimized SP algorithms," 1998, http://www.spiral.net/.
19. S. Oraintara, Y. J. Chen, and T. Q. Nguyen, "Integer fast Fourier transform," *IEEE Transactions on Signal Processing*, Vol. 50, 2002, pp. 607-618.
20. M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal of Association for Computing Machinery*, Vol. 15, 1968, pp. 252-264.
21. M. Püschel, "Constructive representation theory and fast signal transforms," Technical Report Drexel-MCS-1999-1, Drexel University, 1999.
22. M. Püschel, "Decomposing monomial representations of solvable groups," Technical Report Drexel-MCS-1999-2, Drexel University, 1999.
23. T. G. Stockham, "High speed convolution and correlation," in *Proceedings of Spring Joint Computer Conference*, *AFIPS*, 1966, pp. 229-233.
24. C. Y. Tsai, M. H. Fan, and C. H. Huang, "VLSI circuit design of matrix transposition using tensor product formulation," in *Proceedings of the International Conference on Informatics*, *Cybernetics*, *and Systems*, 2003.

Min-Hsuan Fan (范敏玄) received his B.S. degree in Mathematics from National Tsing Hua University and M.S. degree in Computer Science from University of Maryland, Baltimore County, in 1985 and 1990, respectively. Since 1991, he has been a lecturer in the Dept. of Information Management at National Taichung Institute of Technology, Taiwan. His research interests include parallel processing and programming methodology.

Chua-Huang Huang (黃秋煌) received the B.S. degree in Mathematics from Fu Jen University, Taipei, Taiwan, in 1974, the M.S. degree in Computer Science from University of Oregon in 1979, and the Ph.D. degree in Computer Science from the University of Texas at Austin in 1987. From 1979 to 1982, Dr. Huang was an assistant researcher in the Telecommunication Laboratories, Ministry of Communication, R.O.C. He was a faculty member in the Dept. of Computer and Information Science, the Ohio State University between 1987 and 1997, and a professor in Dept. of Computer Science and Information Engineering, National Dong Hwa University, Hualian, Taiwan, between 1997 and 2000. Since 2000, he has been a professor in Dept. of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan. Dr. Huang's research interests include compiler techniques and programming methodologies for high-performance computing.

Yeh-Ching Chung (鍾葉青) received a B.S. degree in Computer Science from Chung Yuan Christian University in 1983, and the M.S. and Ph.D. degrees in Computer and Information Science from Syracuse University in 1988 and 1992, respectively. He joined the Dept. of Information Engineering at Feng Chia University as an Associate professor in 1992 and became a full professor in 1999. From 1998 to 2001, he was the chairman of the department. In 2002, he joined the Dept. of Computer Science at National Tsing Hua University as a full professor. His research interests include parallel and distributed computing, pervasive computing, embedded systems, and system software for SOC design. He is a member of the IEEE computer society and ACM.

Jenshiuh Liu (劉振緒) received his B.S. and M.S. degrees in Nuclear Engineering from National Tsing Hua University, also M.S. and Ph.D. degrees in Computer Science from Michigan State University in 1979, 1981, 1987 and 1992, respectively. Since 1992, he has been an associate professor in the Dept. of Information Engineering and Computer Science at Feng Chia University, Taiwan. His research interests include parallel and distributed processing, computer system security, and computer algorithms.

Jei-Zhii Lee (李介志) received a B.S. degree in Mathematics from National Central University in 1995, and the M.S. degrees in Computer Science and Information Engineering from National Dong Hwa University (NDHU) in 1999. After graduated from NDHU, he joined the Computing Centre at Academia Sinica in 2000. He participated PC cluster team and server system team, respectively. He took care system design, performance issue and maintain in the cluster system. Server system team charged him with the task of the mission-critical Web site, software deployment, new Linux technology and Grid computing.