A Programming Methodology for Designing Parallel Prefix Algorithms *

Min-Hsuan Fan, Chua-Huang Huang, Yeh-Ching Chung, Jen-Shiuh Liu
Department of Information Engineering
Feng Chia University
Taichung, Taiwan, R.O.C.

Jei-Zhii Lee
Department of Computer Science and Information Engineering
National Dong Hwa University
Hualien, Taiwan, R.O.C.

Abstract

In this paper, we use the tensor product notation as the framework of a programming methodology for designing various parallel prefix algorithms. In this methodology, we first express a computational problem in its matrix form. Next, we formulate a matrix equation for the matrix of the computational problem. Then, solve the matrix equation to obtain some simple matrices. Finally, we recursively factorize the subproblem to obtain a tensor product formula representing an algorithm for this problem. We will use the parallel prefix computation problem to illustrate our methodology and derive various parallel prefix algorithms including divide-and-conquer and recursive doubling algorithms.

1. Introduction

Parallel prefixes are also called prefix sums or scans. The mathematical representation of the parallel prefix problem is as the following:

Given an n-element sequence $x_0, x_1, \cdots, x_{n-1}$, computes $y_k = \sum_{i=0}^{k} x_i$ for $k = 0, 1, \cdots, n-1$. The "summation" operation stands for a commutative and associative operation such as addition, multiplication and maximum operation, *etc.*, in the parallel prefix problems. The parallel prefix computation is used by many applications. Evaluation of polynomials, solution of linear recurrence equations, carry-look-ahead circuits, radix sorting, quick sorting and scheduling problems are some of the applications in the domain of the parallel prefix computation [1].

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] and Strassen's matrix multiplication [7, 8, 13]. The tensor product notation is suitable for expressing block-recursive algorithms, data distribution, and interconnection networks [11, 12]. The tensor product operations can be mapped to corresponding programming constructs. Therefore, the tensor product notation provides a framework of designing and implementing parallel programs [3, 5]. In this paper, we will formulate the parallel prefix problem using the tensor product notation and develop different parallel prefix algorithms of 2^n sequences.

The methodology we will use is divided into four steps. First, expresses a computational problem in its matrix form. Next, formulate a matrix equation for the matrix of the computational problem. Third, solve the matrix equation to obtain some simple matrices [6]. Finally, recursively factorize the subproblem to obtain a tensor product formula that represent an algorithm for the parallel prefix problem. We will use the parallel prefix computation problem to illustrate our methodology and derive various parallel prefix algorithms including divide-and-conquer and recursive doubling algorithms [2].

The organization of this paper is as the following. 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 a general block recursive algorithm and explains the programming methodology by the parallel prefix problem. Section 4 illustrates the generation of various parallel prefix algorithms. The final section is conclusions and future works.

*This work was supported in part by National Science Council, R.O.C. under grant NSC 89-2213-E-259-005.

0190-3918/01 $10.00 © 2001 IEEE

2 The Tensor Product Notation

In this section, we provide a brief overview of the tensor product definition and relevant properties. Tensor product operation is a bilinear form that constructs a block matrix from two matrices. Its definition is given below.

Definition 2.1 (tensor product of matrices)

Let $A_{m\times n}$ be an $m \times n$ matrix and $B_{p\times q}$ be a $p \times q$ matrix. The tensor product of A and B is the block matrix obtained by replacing each element $a_{i,j}$ by the matrix $a_{i,j}B_{p\times q}$, i.e., is an $mp \times nq$ matrix, defined as

$$A_{m\times n}\otimes B_{p\times q} = \begin{bmatrix} a_{0,0}B_{p\times q} & \cdots & a_{0,n-1}B_{p\times q} \\ \vdots & \ddots & \vdots \\ a_{n-1,0}B_{p\times q} & \cdots & a_{n-1,n-1}B_{p\times q} \end{bmatrix}_{mp\times nq}$$

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

Another important operation is a stride permutation.

Definition 2.2 (stride permutation)

$$L^{mn}_n(e^m_i \otimes e^n_j) = e^n_j \otimes e^m_i$$

L^{mn}_n is the stride permutation of size mn stride distance n. When a matrix is stored by rows, the tensor basis $e^m_i \otimes e^n_j$ is isomorphic to $E^{m,n}_{i,j}$. In addition, tensor basis $e^n_j \otimes e^m_i$ is isomorphic to $E^{m,n}_{i,j}$ when a matrix is stored by columns. Therefore, L^{mn}_n transposes an $m \times n$ matrix from the row-major order to the column-major order.

The followings are some properties of the tensor products and stride permutations:

1. $A \otimes B \otimes C = (A \otimes B) \otimes C = A \otimes (B \otimes C)$
2. $(A_1 \otimes \cdots \otimes A_k)(B_1 \otimes \cdots \otimes B_k) = (A_1B_1 \otimes \cdots \otimes A_kB_k)$
3. $(A_1 \otimes B_1)(A_2 \otimes B_2)\cdots(A_k \otimes B_k) = (A_1A_2\cdots A_k \otimes B_1B_2\cdots B_k)$
4. $\prod_{i=0}^{n-1}(I_n \otimes A_i) = I_n \otimes \prod_{i=0}^{n-1} A_i$
5. $\prod_{i=0}^{n-1}(A_i \otimes I_n) = \prod_{i=0}^{n-1} A_i \otimes I_n$
6. $(L^{mn}_n)^{-1} = L^{mn}_m, L^n_n = I_n$
7. $L^{rst}_{rs} = L^{rst}_r L^{rst}_s$
8. $L^{rst}_t = (L^{rt}_t \otimes I_s)(I_r \otimes L^{st}_t)$

3 A Programming Methodology for Block Recursive Algorithms

Many block recursive algorithms can be represented by tensor product formulas. Although these block recursive algorithms have been reported, there is still no programming methodology for deriving various block recursive algorithms for a computational problem. In this section, we propose a programming methodology for deriving various block recursive algorithms based on the tensor product notation.

The programming methodology contains four steps. We explain them as following:

Step 1. Represent the computation problem by a matrix Q. For example, the parallel prefix computation problem can be represented by $Y_n = Q_nX_n$, where X_n is an ordered column vector, Y_n is the prefix result column vector, and $Q_n$is an $n \times n$ matrix where all the elements on and below the diagonal of Q_n are 1's and all the elements above the diagonal of Q_n are 0's. That is,

$$Q_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 0 \\ & \ddots & & \vdots \\ 1 & \cdots & 1 & 1 \end{bmatrix}_{n\times n}$$

Step 2. Formulate a matrix equation of Q_n to factorize Q_n into a smaller size of Q_{n_2} with additional pre-operation R_n and post-operation P_n as Equation (1)[1]:

$$Q_n = P_n(I_{n_1} \otimes Q_{n_2} \otimes I_{n_3})R_n \qquad (1)$$

where $n = n_1n_2n_3$, $Q_{n_2}$is the same problem matrix with smaller size n_2. I_{n_1} and I_{n_2} are the identity matrices with size n_1 and n_3, respectively.

Step 3. Solve the Equation (1) for two unknown matrices P_n and R_n with three given values n_1, n_2 and n_3. If the solution P_n and R_n are simple matrices, then go to Step 4, otherwise, find another solution P_n and R_n for Equation (1) again.

A simple matrix is a matrix which each row contains a certain number of none-zero elements and represents a sequence of simple operation. A simple operation matrix should be implemented as a sequence of simple program statements.

For example, the parallel prefix problem matrix of size 2^m can be formulated as the following matrix equation:

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

This equation corresponds to the general form of factorization Equation (1) with $n_1 = 2$, $n_2 = 2^{m-1}$, $n_3 = 1$, and

[1]The matrix operations are applied to an input vector from right to left. Therefore, R is an operation "before" P.

R_{2^m} is I_{2^m}. P_{2^m} can be solved as following.

$$\begin{aligned} P_{2^m} &= Q_{2^m}(I_2 \otimes Q_{2^{m-1}})^{-1} \\ &= \begin{bmatrix} I_{2^{m-1}} & 0_{2^{m-1}} \\ T_{2^{m-1}} & I_{2^{m-1}} \end{bmatrix} \\ &= I_{2^m} + \begin{bmatrix} 0_{2^{m-1}} & 0_{2^{m-1}} \\ T_{2^{m-1}} & 0_{2^{m-1}} \end{bmatrix}, \end{aligned}$$

where

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

is a $2^{m-1} \times 2^{m-1}$ matrix which all elements of last column are 1's and elsewhere are 0's. The solution matrices P_{2^m} and R_{2^m} are both simple matrices.

In a share memory multiprocessor environment, the solution P_{2^m} can be implemented to a sequence of 2^{m-1} assignment statements as following:

$$\begin{aligned} y_{2^{m-1}} &= x_{2^{m-1}} + x_{2^{m-1}-1} \\ y_{2^{m-1}+1} &= x_{2^{m-1}+1} + x_{2^{m-1}-1} \\ y_{2^{m-1}+2} &= x_{2^{m-1}+2} + x_{2^{m-1}-1} \\ &\cdots \\ y_{2^m-1} &= x_{2^m-1} + x_{2^{m-1}-1} \end{aligned}$$

If there are more than one processors used to execute the program, the statements will be distributed to different processor and executed.

In a distributed memory multiprocessor environment, the implementation requires a multicast communication from processor index $2^{m-1}-1$ to the second half processors and performs a simple addition operation in each receiving processors.

Step 4. This step is to recursively factorize the subproblem Q_{n_2} with the same factorization in Step 3 until n_2 is small enough such that Q_{n_2} can be directly implemented as simple program statements.

As the example in Step 3, the parallel prefix problem matrix $Q_{2^{m-1}}$ of size 2^{m-1} can be expanded to the product of $P_{2^{m-1}}$ and $I_2 \otimes Q_{2^{m-2}}$ recursively. We obtain the following result.

$$\begin{aligned} Q_{2^m} &= P_{2^m}(I_2 \otimes Q_{2^{m-1}}) \\ &= P_{2^m}(I_2 \otimes P_{2^{m-1}}(I_2 \otimes Q_{2^{m-2}})) \\ &= P_{2^m}(I_2 \otimes P_{2^{m-1}})(I_2 \otimes I_2 \otimes Q_{2^{m-2}}) \\ &= P_{2^m}(I_2 \otimes P_{2^{m-1}})(I_4 \otimes Q_{2^{m-2}}) \\ &\cdots \\ &= \prod_{i=1}^{m}(I_{2^{m-i}} \otimes P_{2^i}) \end{aligned}$$

The final tensor product formula

$$\prod_{i=1}^{m}(I_{2^{m-i}} \otimes P_{2^i}) = P_{2^m}(I_2 \otimes P_{2^{m-1}}) \cdots (I_{2^{m-1}} \otimes P_2)$$

represents a parallel prefix computation algorithm that is known as the divide-and-conquer parallel prefix algorithm.

From the above four steps, we can get an algorithm for parallel prefix computation problem. The algorithm can be represented by a tensor product formula and translated to program statements directly. Since theoretically there are infinitely possible solutions for the factorization of Equation (1), we may find some meaningful solutions for different criteria's such as various computer architecture and data distribution.

4 Some Parallel Prefix Algorithms by The Methodology

Besides the divide-and-conquer algorithm for parallel prefix computation problem we presented in Section 3, we demonstrate the programming methodology by generating various parallel prefix computation algorithms. For simplicity, we classify the possible equations into parallel form and vector form. We present several solutions in the following.

4.1 The Parallel Form Case

The parallel form of the parallel prefix problem can be represented as following:

$$Q_{2^m} = P_{2^m}(I_2 \otimes Q_{2^{m-1}})R_{2^m} \qquad (2)$$

where Q_{2^m} is the prefix computation matrix, P_{2^m} is the post-computation matrix, and R_{2^m} is the pre-computation matrix. Equation (2) corresponds to the general form of factorization Equation (1) with $n_1 = 2$, $n_2 = 2^{m-1}$, $n_3 = 1$. There are many possible solutions of P_{2^m} and R_{2^m} which satisfy the Equation (2).

Algorithm 4.1.1: This algorithm is mentioned in Section 3. That is $R_{2^m} = I_{2^m}$ and

$P_{2^m} = I_{2^m} + \begin{bmatrix} 0_{2^{m-1}} & 0_{2^{m-1}} \\ T_{2^{m-1}} & 0_{2^{m-1}} \end{bmatrix}$, where

$$T_{2^{m-1}} = \begin{bmatrix} 0 & \cdots & 0 & 1 \\ 0 & \cdots & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 1 \end{bmatrix}.$$

The tensor product formula for the divide-and-conquer algorithm is $Q_{2^m} = \prod_{i=1}^{m}(I_{2^{m-i}} \otimes P_{2^i})$. We can prove the formula by mathematical induction.

Proof. Base case: $m = 1$, $Q_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = P_2$.

Induction step: assume $Q_{2^k} = \prod_{i=1}^{k}(I_{2^{k-i}} \otimes P_{2^i})$

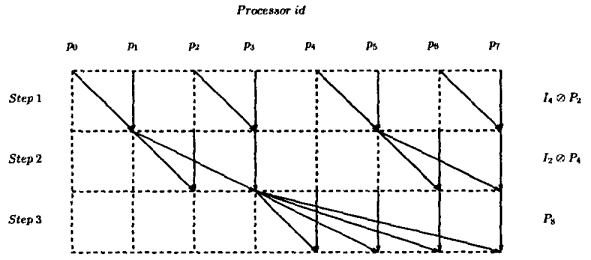

Figure 1. Algorithm 4.1.1 for $n = 8$

$$\begin{aligned} Q_{2^{k+1}} &= P_{2^{k+1}}(I_2 \otimes Q_{2^k}) \\ &= P_{2^{k+1}}(I_2 \otimes \prod_{i=1}^{k}(I_{2^{k-i}} \otimes P_{2^i})) \\ &= P_{2^{k+1}}(\prod_{i=1}^{k}(I_2 \otimes I_{2^{k-i}} \otimes P_{2^i})) \\ &= P_{2^{k+1}}(\prod_{i=1}^{k}(I_{2^{k+1-i}} \otimes P_{2^i})) \\ &= \prod_{i=1}^{k+1}(I_{2^{k+1-i}} \otimes P_{2^i}) \end{aligned}$$

■

From the formula we know that the parallel prefix computation of 2^m elements can be finish in m steps. Each step needs one multicast communication time and one computation time if each processor contains just one element. That is to say the time complexity of parallel prefix computation is $O(\log n)$.

We illustrate an example in a distributed memory multiprocessors environment to explain how the algorithm is progressing. Let $m = 3$ and the binary operation be the addition operation. Then the formula of Q_8 can be factorized as following:

$$Q_8 = P_8(I_2 \otimes P_4)(I_4 \otimes P_2)$$

We use Figure 1. to show the operations. The three communication and computation steps of this algorithm are as following:

1. The communications are among neighboring processors and perform an addition operation. The sums are stored in the processors of higher index. Operation P_2 is a one-to-one unicast. $I_4 \otimes P_2$ means 4 copies of P_2 are performed.

2. The sums stored in processors p_1 and p_5 are multicasted to the following two processors, then perform an add operation. Operation P_4 is a one-to-two multicast and two copies of P_4 are performed in $I_2 \otimes P_4$.

3. The sum stored in p_3 is multicasted to the following four processors and then an addition operation is performed. Operation P_8 is a one-to-four multicast.

From the example we know that each factor of the tensor product formula is a sequence of simple assignments. It is easy to automatically generate the corresponding parallel code for the parallel prefix tensor product formula of Algorithm 4.1.1.

Algorithm 4.1.2: In this algorithm, we set $R_{2^m} = L_2^{2^m}$. Equation (2) becomes $Q_{2^m} = P_{2^m}(I_2 \otimes Q_{2^{m-1}})L_2^{2^m}$. Operation P_{2^m} can be solved as

$$\begin{aligned} P_{2^m} &= Q_{2^m}(L_2^{2^m})^{-1}(I_2 \otimes Q_{2^{m-1}})^{-1} \\ &= \begin{bmatrix} T_{(2^m-1)\times 2^{m-1}} & 0_{(2^m-1)\times 2^{m-1}} \\ [0\cdots 00]_{1\times 2^{m-1}} & 0_{1\times 2^{m-1}} \end{bmatrix} \\ &+ \begin{bmatrix} 0_{1\times 2^{m-1}} & [0\cdots 01]_{1\times 2^{m-1}} \\ 0_{(2^m-1)\times 2^{m-1}} & T_{(2^m-1)\times 2^{m-1}} \end{bmatrix}, \end{aligned}$$

where

$$T_{(2^m-1)\times 2^{m-1}} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & 0 & & \vdots & \vdots \\ 0 & 1 & & \vdots & \vdots \\ 0 & 1 & & 0 & 0 \\ \vdots & 0 & \ddots & 0 & 0 \\ \vdots & \vdots & & 1 & 0 \\ \vdots & \vdots & & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}_{(2^m-1)\times 2^{m-1}}$$

We notice that when $m = 1$, $Q_2 = P_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

Operation $L_2^{2^m}$ is a stride permutation of input column vector. Matrix $I_2 \otimes Q_{2^{m-1}} = \begin{bmatrix} Q_{2^{m-1}} & 0 \\ 0 & Q_{2^{m-1}} \end{bmatrix}$ represents two half size of prefix computation matrix Q_{2^m} be computed in parallel. Matrix P_{2^m} is the post computation that processor i receives two elements from processor $\lfloor i/2 \rfloor$ and processor $\lfloor (i-1)/2 \rfloor + 2^{m-1}$. Then processor i adds the two elements together except processor 0. Because there are at most two 1's in each row of P_{2^m}. P_{2^m} is a simple operation and can be done in just two units communication time and one computation time if each processor contain just one element.

We can factorize $Q_{2^{m-1}}$ further and obtain the following result.

$$\begin{aligned} Q_{2^m} &= P_{2^m}(I_2 \otimes Q_{2^{m-1}})L_2^{2^m} \\ &= P_{2^m}(I_2 \otimes P_{2^{m-1}}(I_2 \otimes Q_{2^{m-2}})L_2^{2^{m-1}})L_2^{2^m} \\ &= P_{2^m}(I_2 \otimes P_{2^{m-1}})(I_{2^2} \otimes Q_{2^{m-2}})(I_2 \otimes L_2^{2^{m-1}})L_2^{2^m} \\ &\cdots \\ &= \prod_{i=1}^{m}(I_{2^{m-i}} \otimes P_{2^i}) \prod_{i=m}^{1}(I_{2^{m-i}} \otimes L_2^{2^i}) \end{aligned}$$

The matrix $\prod_{i=m}^{1}(I_{2^{m-i}} \otimes L_2^{2^i})$ is the bit reversal operation. It can be done in just one step. This algorithm requires

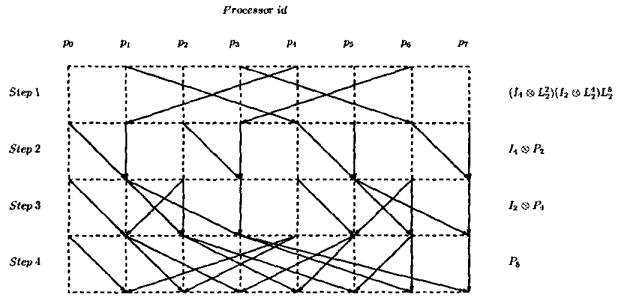

Figure 2. Algorithm 4.1.2 for $n = 8$

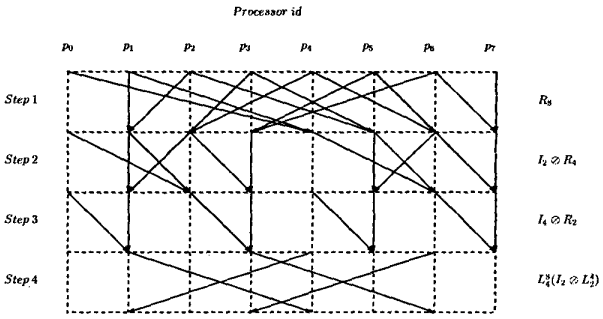

Figure 3. Algorithm 4.1.3 for $n = 8$

only m steps with a bit reversal permutation. The bit reversal permutation is implemented as data relocation. The other m steps need two communication operations and one computation operation if each processor contains only one element. That is, the time complexity of this algorithm is still $O(\log n)$.

We omit the proof and illustrate an example. Let $m = 3$ and the binary operation is the addition operation. Then, the formula of Q_{2^3} can be factorized as following:

$$Q_8 = P_8(I_2 \otimes P_4)(I_4 \otimes P_2)(I_4 \otimes L_2^2)(I_2 \otimes L_2^4)L_2^8$$

The computation and communication steps of Q_8 are illustrated in Figure 2.

Algorithm 4.1.3: Set $P_{2^m} = L_{2^{m-1}}^{2^m}$. Equation (2) becomes $Q_{2^m} = L_{2^{m-1}}^{2^m}(I_2 \otimes Q_{2^{m-1}})R_{2^m}$. Operation R_{2^m} can be solved as

$$\begin{aligned} R_{2^m} &= (I_2 \otimes Q_{2^{m-1}})^{-1}(L_{2^{m-1}}^{2^m})^{-1}Q_{2^m} \\ &= \begin{bmatrix} T_{2^{m-1}\times(2^m-1)} & 0_{(2^{m-1})\times 1} \\ 0_{(2^{m-1})\times(2^{m-1})} & 0_{(2^{m-1})\times(2^{m-1})} \end{bmatrix} \\ &+ \begin{bmatrix} 0_{2^{m-1}\times 1} & 0_{2^{m-1}\times 2^{m-1}} \\ \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{2^{m-1}\times 1} & T_{2^{m-1}\times 2^{m-1}} \end{bmatrix} \end{aligned}$$

where

$$T_{2^{m-1}\times 2^{m-1}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & & 0 \\ 0 & 1 & 1 & 0 & 0 & & & 0 \\ 0 & 0 & 0 & 1 & 1 & & & 0 \\ \vdots & & & & & \ddots & 0 & 0 \\ 0 & & \cdots & & & 0 & 1 & 1 \end{bmatrix}.$$

We can factorize $Q_{2^{m-1}}$ further and obtain the following result.

$$\begin{aligned} Q_{2^m} &= L_{2^{m-1}}^{2^m}(I_2 \otimes Q_{2^{m-1}})R_{2^m} \\ &= L_{2^{m-1}}^{2^m}(I_2 \otimes (L_{2^{m-2}}^{2^{m-1}}(I_2 \otimes Q_{2^{m-2}})R_{2^{m-1}}))R_{2^m} \\ &= L_{2^{m-1}}^{2^m}(I_2 \otimes L_{2^{m-2}}^{2^{m-1}})(I_4 \otimes Q_{2^{m-2}})(I_2 \otimes R_{2^{m-1}})R_{2^m} \\ &\cdots \\ &= \prod_{i=m}^{1}(I_{2^{i-1}} \otimes L_{2^{m-i}}^{2^m}) \prod_{i=m}^{1}(I_{2^{m-i}} \otimes R_{2^i}) \end{aligned}$$

The matrix $\prod_{i=m}^{1}(I_{2^{i-1}} \otimes L_{2^{m-i}}^{2^m})$ is the bit reversal operation. It can be done in just one step. This algorithm requires only m steps with a bit reversal permutation. The bit reversal permutation is implemented as data relocation. The other m steps need two communication operations and one computation operation if each processor contains only one element. That is, the time complexity of this algorithm is still $O(\log n)$.

We omit the proof and illustrate an example. Let $m = 3$ and the binary operation is the addition operation. Then, the formula of Q_{2^3} can be factorized as following:

$Q_8 = L_4^8(I_2 \otimes L_2^4)(I_4 \otimes R_2)(I_2 \otimes R_4)R_8$

The computation and communication steps of Q_8 are illustrated in Figure 3.

4.2 The Vector Form Case

The vector form of parallel prefix problem can be represented as the following tensor product formula:

$$Q_{2^m} = P_{2^m}(Q_{2^{m-1}} \otimes I_2)R_{2^m} \qquad (3)$$

where Q_{2^m} is the prefix computation matrix, P_{2^m} is the post-computation matrix, and R_{2^m} is the pre-computation matrix. Equation (3) matches to the general form of factorizing Equation (1) with $n_1 = 1, n_2 = 2^{m-1}, n_3 = 2$. There are many possible solutions of P_{2^m} and R_{2^m} which satisfy Equation (3).

Algorithm 4.2.1: Set $P_{2^m} = I_{2^m}$. Equation (3) becomes $Q_{2^m} = (Q_{2^{m-1}} \otimes I_2)R_{2^m}$. Operation R_{2^m} can be solved

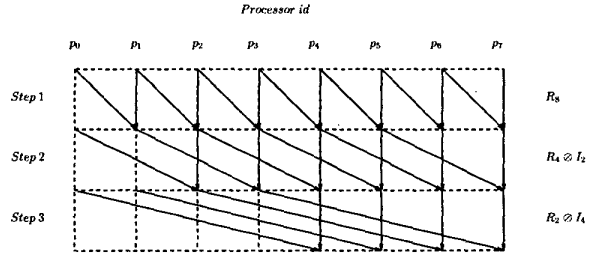

Figure 4. Algorithm 4.2.1 for $n = 8$

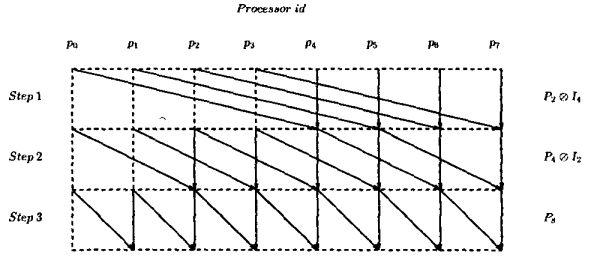

Figure 5. Algorithm 4.2.2 for $n = 8$

as

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

The elements on and below the diagonal of R_{2^m} are 1's and elsewhere are 0's. Operation R_{2^m} represents a pre-operation that each input element is added to the next element except the last one. R_{2^m} is a simple operation and can be implemented using a single addition.

Operation $Q_{2^{m-1}} \otimes I_2$ represents the prefix computation of 2^{m-1} blocks with two data elements in each block. One data element is over the odd indices of the input vector and another data element is over the even indices of the input elements. We can factorize $Q_{2^{m-1}}$ further and obtain the following result.

$$\begin{aligned} Q_{2^m} &= (Q_{2^{m-1}} \otimes I_2) R_{2^m} \\ &= ((Q_{2^{m-2}} \otimes I_2) R_{2^{m-1}} \otimes I_2) R_{2^m} \\ &= (Q_{2^{m-2}} \otimes I_2 \otimes I_2)(R_{2^{m-1}} \otimes I_2) R_{2^m} \\ &= (Q_{2^{m-2}} \otimes I_{2^2})(R_{2^{m-1}} \otimes I_2) R_{2^m} \\ &\cdots \\ &= \prod_{i=m}^{1} (R_{2^i} \otimes I_{2^{m-i}}) \end{aligned}$$

The equation $Q_{2^m} = \prod_{i=m}^{1} (R_{2^i} \otimes I_{2^{m-i}})$ computes the parallel prefix of 2^m elements and can be completed in m steps. Each step needs only one addition if each processor contains one element. That is, the time complexity of Equation 4.2.1 is $O(\log n)$.

We omit the proof and illustrate an example. We explain Algorithm 4.2.1 for $n = 3$: $Q_8 = (R_2 \otimes I_4)(R_4 \otimes I_2) R_8$ in details.

1. R_8 has 7 one-to-one unicasts, the communication are among neighboring processors.
2. $R_4 \otimes I_2$ has 6 one-to-one unicasts, the communication is among processors of distance 2.
3. $R_2 \otimes I_4$ has 4 one-to-one unicasts, the communication is among processors of distance 4.

Algorithm 4.2.1 is known as the recursive-doubling algorithm of parallel prefix computation. The computation and communication steps of Q_8 are illustrated in Figure 4.

Algorithm 4.2.2: Set $R_{2^m} = I_{2^m}$. Equation (3) becomes $Q_{2^m} = P_{2^m}(Q_{2^{m-1}} \otimes I_2)$. Operation P_{2^m} can be solved as

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

The elements on and below the diagonal of P_{2^m} are 1's and elsewhere are 0's. Operation P_{2^m} represents a post-operation that each input element is added to the next element except the last one. P_{2^m} is a simple operation matrix and can be implemented using a single addition in multiprocessor environment.

We can factorize $Q_{2^{m-1}}$ further and obtain the following result.

$$\begin{aligned} Q_{2^m} &= P_{2^m}(Q_{2^{m-1}} \otimes I_2) \\ &= P_{2^m}(P_{2^{m-1}}(Q_{2^{m-2}} \otimes I_2) \otimes I_2) \\ &= P_{2^m}(P_{2^{m-1}} \otimes I_2)(Q_{2^{m-2}} \otimes I_2 \otimes I_2) \\ &= P_{2^m}(P_{2^{m-1}} \otimes I_2)(Q_{2^{m-2}} \otimes I_{2^2}) \\ &\cdots \\ &= \prod_{i=1}^{m} (P_{2^i} \otimes I_{2^{m-i}}) \end{aligned}$$

The equation $Q_{2^m} = \prod_{i=1}^{m} (P_{2^i} \otimes I_{2^{m-i}})$ computes the parallel prefix of 2^m elements and can be completed in m steps. Each step needs only one addition if each processor contains only one element. That is, the time complexity of Equation 4.2.1 is $O(\log n)$.

We explain Algorithm 4.2.1 for $n = 3$: $Q_8 = P_8(P_4 \otimes I_2)(P_2 \otimes I_4)$ in details.

1. $P_2 \otimes I_4$ has 4 one-to-one unicasts, the communication is among processors of distance 4.

2. $P_4 \otimes I_2$ has 6 one-to-one unicasts, the communication is among processors of distance 2.

3. P_8 has 7 one-to-one unicasts, the communication are among neighboring processors.

Algorithm 4.2.2 is known as the reverse recursive-doubling algorithm of parallel prefix computation. We omit the proof and illustrate an example for $n = 3$ in Figure 5. Algorithms 4.2.1 and 4.2.2 have exactly the same computations. However, their computations are in the opposite order.

Algorithm 4.2.3: In this algorithm, we set $R_{2^m} = L^{2^m}_{2^{m-1}}$. Equation 4.2 becomes $Q_{2^m} = P_{2^m}(Q_{2^{m-1}} \otimes I_2)L^{2^m}_{2^{m-1}}$.

Operation P_{2^m} can be solved as

$$
\begin{aligned}
P_{2^m} &= Q_{2^m}(L^{2^m}_{2^{m-1}})^{-1}(Q_{2^{m-1}} \otimes I_2)^{-1} \\
&= \begin{bmatrix} T_{2^{m-1}\times(2^m-1)} & 0_{(2^{m-1})\times 1} \\ 0_{2^{m-1}\times 1} & T_{2^{m-1}\times(2^m-1)} \end{bmatrix} \\
&+ \begin{bmatrix} 0_{2^{m-1}\times(2^m-2)} & 0_{2^{m-1}\times 2} \\ 0_{2^{m-1}\times(2^m-2)} & \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \end{bmatrix}_{2^{m-1}\times 2} \end{bmatrix}
\end{aligned}
$$

where

$$
T_{2^{m-1}\times(2^m-1)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & & 0 \\ 0 & 0 & 1 & 0 & 0 & & & \vdots \\ 0 & 0 & 0 & 0 & 1 & & & \\ \vdots & & & & & \ddots & 0 & 0 \\ 0 & & \cdots & & & 0 & 0 & 1 \end{bmatrix}.
$$

We notice that when $m = 1$, $Q_2 = P_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

Operation $L^{2^m}_{2^{m-1}}$ is a stride permutation of input column vector. Matrix P_{2^m} is the post computation that the first half processors receive one element from processor $2i$, where i is the processor index. In the second half processors, the processor i receives two elements from processor $2(i - 2^{m-1}) + 1$ and processor $2^m - 2$. Then the processor adds the two elements together. Because there are at most two 1's in each row of P_{2^m}. P_{2^m} is a simple operation matrix and can be done in just two units communication time and one computation time if each processor contain just one element.

We can factorize $Q_{2^{m-1}}$ further and obtain the following result.

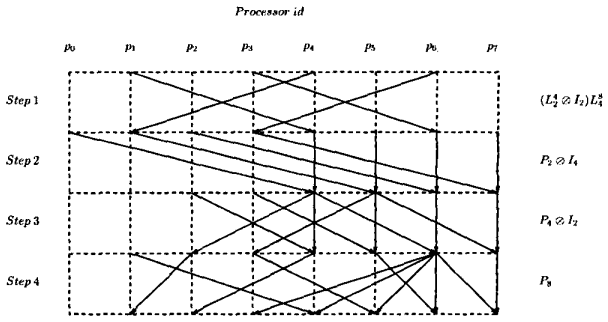

Figure 6. Algorithm 4.2.3 for $n = 8$

$$
\begin{aligned}
Q_{2^m} &= P_{2^m}(Q_{2^{m-1}} \otimes I_2)L^{2^m}_{2^{m-1}} \\
&= P_{2^m}(P_{2^{m-1}}(Q_{2^{m-2}} \otimes I_2)L^{2^{m-1}}_{2^{m-2}} \otimes I_2)L^{2^m}_{2^{m-1}} \\
&= P_{2^m}(P_{2^{m-1}} \otimes I_2)(Q_{2^{m-2}} \otimes I_{2^2})(L^{2^{m-1}}_{2^{m-2}} \otimes I_2)L^{2^m}_{2^{m-1}} \\
&\cdots \\
&= \prod_{i=1}^{m}(P_{2^i} \otimes I_{2^{m-i}}) \prod_{i=m}^{1}(L^{2^i}_{2^{i-1}} \otimes I_{2^{m-i}})
\end{aligned}
$$

The matrix $\prod_{i=m}^{1}(L^{2^i}_{2^{i-1}} \otimes I_{2^{m-i}})$ is the bit reversal operation. It can be done in just one step. This algorithm requires only m steps with a bit reversal permutation. The bit reversal permutation is implemented as data relocation. The other m steps need two communication operations and one computation operation if each processor contains only one element. That is, the time complexity of this algorithm is still $O(\log n)$.

We omit the proof and illustrate an example. Let $m = 3$ and the binary operation is the addition operation. Then, the formula of Q_8 can be factorized as following:

$Q_8 = P_8(P_4 \otimes I_2)(P_2 \otimes I_4)(L^4_2 \otimes I_2)L^8_4$

The computation and communication steps of Q_8 are illustrated in Figure 6.

The other solutions of Equations (2) and (3) can be obtained by choosing different P_{2^m} and R_{2^m}. Some solutions may yield effective and efficient algorithms; some may deliver poor algorithms. The methodology has a degree of freedom in the decision of formula factorization and pre-/post-conditions. For different computer architectures and data allocation, this methodology provides a feasible approach to design efficient algorithms.

5 Conclusions

In this paper, we present a programming methodology for designing block recursive algorithms. We employ the tensor product notation to formulate computational problems and derive different algorithms of given problems. Various parallel prefix algorithms are derived using this

methodology. The algorithms generated by the methodology are represented as tensor product formulas. Then, the tensor product formulas can be mapped to parallel programs easily.

The key idea of the programming methodology is to factorize the problem matrix. The factors of the problem matrix should contain only tensor products of simple matrix operations. The simple matrix operations should be easily implemented as simple programming statements.

Since tensor product can also model different computer architectures, the methodology can be extended to generate various algorithms for different computer architectures. In vector processor machine, we can generate algorithms in vector form by tensor product notation. In parallel machine, we can generate algorithms in parallel form to fit the architecture.

The future work 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] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[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*, pages 49–58, Los Alamitos, USA, 1994. IEEE Comput. Soc. Press.

[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. *J. Parallel and Distributed Computing*, 34:137–153, 1996.

[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. *Appl. Math Letters*, 3(3):104–108, 1990.

[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 1992 International Conference on Parallel Processing*, volume III, Algorithms and Applications, pages III:104–108, Boca Raton, Florida, August 1992. CRC Press.

[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*, 9(4):450–500, 1990.

[10] R. W. Johnson, C.-H. Huang, and J. R. Johnson. Multilinear algebra and parallel programming. *The Journal of Supercomputing*, 5(2–3):189–217, October 1991.

[11] S. D. Kaushik, S. Sharma, and C.-H. Huang. An algebraic theory for modeling multistage interconnection networks. *Journal of Information Science and Engineering*, 9(1):1–26, 1993.

[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*, 12(1):25–49, 1996.

[13] 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*, 4(4):275–289, Winter 1995.