JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **34,** 137–153 (1996)
ARTICLE NO. 0051

A Framework for Generating Distributed-Memory Parallel Programs for Block Recursive Algorithms[1]

S. K. S. GUPTA*[,2,3], C.-H. HUANG†[,4], P. SADAYAPPAN†[,5], AND R. W. JOHNSON‡[,6]

School of Electrical Engineering and Computer Science, Ohio University, Athens, Ohio 45701, and Department of Computer Science, Duke University, Durham, North Carolina 27708-0129; †Department of Computer and Information Science, The Ohio State University, Columbus, Ohio 43210; and ‡Department of Computer Science, St. Cloud State University, St. Cloud, Minnesota 56301

A framework for synthesizing communication-efficient distributed-memory parallel programs for block recursive algorithms such as the fast Fourier transform (FFT) and Strassen's matrix multiplication is presented. This framework is based on an algebraic representation of the algorithms, which involves the tensor (Kronecker) product and other matrix operations. This representation is useful in analyzing the communication implications of computation partitioning and data distributions. The programs are synthesized under two different target program models. These two models are based on different ways of managing the distribution of data for optimizing communication. The first model uses point-to-point interprocessor communication primitives, whereas the second model uses data redistribution primitives involving collective all-to-many communication. These two program models are shown to be suitable for different ranges of problem size. The methodology is illustrated by synthesizing communication-efficient programs for the FFT. This framework has been incorporated into the EXTENT system for automatic generation of parallel/vector programs for block recursive algorithms. © 1996 Academic Press, Inc.

1. INTRODUCTION

In this paper, we present a framework for synthesizing communication-efficient distributed-memory programs. This framework is useful in synthesizing programs for block recursive algorithms. The mathematical framework used is based on the tensor (Kronecker) product [9] and other matrix operations. The tensor product has been used for modeling algorithms with recursive computational structure, occurring in application areas such as digital signal processing [10, 21], image processing [22], linear system design [4], and statistics [11]. In recent years, the tensor product framework has been successfully used to design and implement high-performance algorithms to compute the discrete Fourier transform (DFT) [17, 24] and matrix multiplication [14, 15] on shared-memory vector multiprocessors. The significance of the tensor product lies in its ability to model both the computational structures occurring in block-recursive algorithms and the underlying hardware structures, such as the interconnection networks [19, 20].

The goal of this paper is to develop a framework for synthesizing efficient distributed-memory programs for block recursive algorithms. In this framework we start with a mathematical specification of the computation using tensor products. We present techniques for developing efficient message passing codes by analyzing the structure of the input mathematical formulas. Designing and implementing an algorithm using the tensor product involves viewing the computation as a linear transformation, rewriting the linear transformation as a matrix product of tensor products of smaller computation matrices, recursively applying the rewriting rule to smaller computation matrices, and translating the components of the resulting tensor product formula into vector/parallel operations, iterative loops, and data movement operations. Once expressed using the tensor product notation, several forms of the algorithm with different performance characteristics can be derived by exploiting the algebraic properties of its matrix representation.

In this paper, we provide a framework for designing and implementing programs for distributed-memory MIMD machines. We first present an algebraic representation based on the tensor product for describing the semantics of regular data distributions. Using this algebraic representation, we develop criteria for identifying data distributions which will permit a communication-free implementation of the computation represented by a given tensor product formula. Using these criteria we develop techniques for synthesizing programs for distributed-memory machines with the goal of minimizing the communication overhead. This leads to generation of programs under two different programming models. These two models are based on dif-

[1] This work was supported in part by ARPA, Order 7898, monitored by NIST under Grant 60NANB1D1151, and ARPA, Order 7899, monitored by NIST under Grant 60NANB1D1150.

[2] This work was done while the author was at the Ohio State University.

[3] E-mail: gupta@ace.cs.ohiou.edu; sandeep@cs.duke.edu.

[4] E-mail: chh@cis.ohio-state.edu.

[5] E-mail: saday@cis.ohio-state.edu.

[6] E-mail: rwj@eeyore.stcloud.msus.edu.

ferent ways of managing the distribution of data for optimizing communication. In the first model, distributions of arrays are kept static and so communication is needed whenever a processor requires data elements which it does not own. In the second model, distributions of the data arrays are dynamically changed to ensure that computation is localized in every computation step. This results in programs with different communication overhead characteristics due to use of different communication primitives for performing data movement. The first model uses point-to-point interprocessor communication primitives, whereas the second model uses data redistribution primitives involving collective all-to-many communication. These two program models are shown to be suitable for different ranges of problem size. Although the program with redistributions uses more messages than the program with point-to-point communication, the program with redistributions has lower communication volume than the program using point-to-point communication primitives. Therefore, the program using redistributions has lower communication overhead than the program with point-to-point communication when the problem size is large.

We have implemented programs for the Cooley–Tukey FFT and the Stockham FFT using the methodology presented in this paper, and evaluated their performance on an Intel iPSC/860 system. This framework has been incorporated in EXTENT (an *EX*pert system for *TEN*sor product *T*ranslation), which is a parallel programming environment for automatic generation of parallel/vector programs from tensor product formulas [6].

The paper is organized as follows. Section 2 gives an overview of the theory of tensor products. In Section 3, we motivate the two programming models used in this paper. Section 4 presents the algebraic semantics of regular data distributions using tensor products. Section 5 characterizes data distributions, if any, which permit communication-free implementation of a computation corresponding to a tensor product. In Section 6, we describe a procedure for synthesizing programs under the point-to-point communication model. Section 7 describes the synthesis of programs using the alternate model with redistribution commands. Performance results on the Intel iPSC/860 system are presented in Section 8. Conclusions are provided in Section 9.

2. AN OVERVIEW OF THE TENSOR PRODUCT

In this section, we illustrate the formulation of block recursive algorithms using tensor products. We begin with some preliminary definitions which are essential for understanding the rest of the paper.

2.1. Preliminaries

The tensor product is useful in expressing the block structure in a matrix. Let A be an $m \times n$ matrix and B be a $p \times q$ matrix. The *tensor product* $A \otimes B$ is a block matrix obtained by replacing each element $a_{i,j}$ with the matrix $a_{i,j}B$; i.e.,

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

The above tensor product can be factored as

$$\begin{aligned} A^{m,n} \otimes B^{p,q} &= (A^{m,n} \otimes I_p)(I_n \otimes B^{p,q}) \\ &= (I_m \otimes B^{p,q})(A^{m,n} \otimes I_q), \end{aligned}$$

where I_n represents the $n \times n$ identity matrix. A tensor factorization can be used to efficiently compute Y^{mp} obtained by applying $C^{mp,nq} \equiv (A^{m,n} \otimes B^{p,q})$ to vector X^{nq}; i.e., $Y^{mp} = C^{mp,nq}(X^{nq})$. For example, direct application of $C^{mp,nq}$ to X^{nq} requires $O(mpnq)$ scalar operations. However, the following algorithm based on the tensor factorization of $C^{mp,nq}$: $Z^{mq} = (A^{m,n} \otimes I_q)(X^{nq})$; $Y^{mp} = (I_m \otimes B^{p,q})(Z^{mq})$, requires only $O(qmn + mpq)$ scalar operations.

A tensor product involving an identity matrix can be implemented as parallel operations. For example, consider the application of $(I_m \otimes A^{p,n})$ to X^{mn}; i.e.,

$$\begin{bmatrix} A^{p,n} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & A^{p,n} \end{bmatrix} \begin{bmatrix} X^{mn}(0:n-1) \\ X^{mn}(n:2n-1) \\ \vdots \\ X^{mn}((m-1)n:mn-1) \end{bmatrix} = \begin{bmatrix} A^{p,n}X^{mn}(0:n-1) \\ A^{p,n}X^{mn}(n:2n-1) \\ \vdots \\ A^{p,n}X^{mn}((m-1)n:mn-1) \end{bmatrix}.$$

This can be interpreted as m copies of $A^{p,n}$ acting in parallel on m disjoint segments of X^{mn}. However, to interpret the application $(A^{p,n} \otimes I_m)$ to X^{mn} as parallel operations we need to understand stride permutations.

Stride permutations (shuffle permutations [7]) belong to a special class of permutations called tensor permutations [16]. The stride permutation L_n^{mn} of a vector X^{mn} is a vector Y^{mn}, where $Y^{mn} = [X^{mn}(0:mn-1:n); X^{mn}(1:mn-1:n); \ldots; X^{mn}(m-1:mn-1:n)]$; i.e., the first m elements of Y^{mn} are $X^{mn}(0:mn-1:n)$, which represents elements of X^{mn} at stride n starting with element 0. The next m elements are elements of X^{mn} at stride n, starting with element 1. The stride permutation L_n^{mn} can be represented as an $mn \times mn$ transformation. For example, the effect of applying L_2^6 to X^6 can be expressed in matrix form as follows:

$$L_2^6 X^6 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_1 \\ x_3 \\ x_5 \end{bmatrix}.$$

Stride permutations can also be defined in terms of a permutation of the tensor product of vector bases. A *vector basis* e_i^m, $0 \le i < m$, is a column vector of length m with a one at position i and zeros elsewhere. The tensor product of vector bases is called a *tensor basis*. A tensor basis $e_{i_1}^{m_1} \otimes \cdots \otimes e_{i_t}^{m_t}$ can be linearized into a vector basis $e_{i_1 m_2 \cdots m_t + \cdots + i_{t-1} m_t + i_t}^{m_1 \cdots m_t}$. Equivalently, a vector basis e_i^M can be factored into a tensor product of vector bases $e_{i_1}^{m_1} \otimes \cdots \otimes e_{i_t}^{m_t}$, where $M = m_1 \cdots m_t$ and $i_k = (i \text{ div } M_{k+1}) \bmod m_k$, $M_k = \prod_{i=k}^t m_i$, $M_{t+1} = 1$. For example, $e_8^{12} = e_1^2 \otimes e_1^3 \otimes e_0^2$. The stride permutation L_n^{mn} can now be defined as

$$L_n^{mn}\,(e_i^m \otimes e_j^n) = e_j^n \otimes e_i^m.$$

This gives the relationship between the indexing of the input and the output vectors. By linearizing the input tensor basis $e_i^m \otimes e_j^n$ to e_{in+j}^{mn}, we get the indexing function of the input vector to be $in + j$. Similarly, the indexing function of the output vector is obtained by linearizing the output tensor basis to be $jm + i$. Therefore, the effect of applying the stride permutation L_n^{mn} to an input vector is that the element at index $in + j$ of the input vector is stored in location at index $jm + i$ of the output vector.

Using stride permutations, an application of $(A^{p,n} \otimes I_m)$ to X^{mn} can also be interpreted as m parallel applications of $A^{p,n}$ to disjoint segments of X^{mn} by using the identity $L_m^{pm}\,(A^{p,n} \otimes I_m) = (I_m \otimes A^{p,n})\,L_m^{mn}$ as follows: $L_m^{pm}(Y^{pm}) = (I_m \otimes A^{p,n})(L_m^{mn}(X^{mn}))$, i.e.,

$$\begin{bmatrix} Y^{pm}(0:pm-1:m) \\ Y^{pm}(1:pm-1:m) \\ \vdots \\ Y^{pm}(m-1:pm-1:m) \end{bmatrix} = \begin{bmatrix} A^{p,n}X^{mn}(0:mn-1:m) \\ A^{p,n}X^{mn}(1:mn-1:m) \\ \vdots \\ A^{p,n}X^{mn}(m-1:mn-1:m) \end{bmatrix}.$$

However, the inputs for each application of $A^{p,n}$ are accessed at a stride of m and the outputs are also stored at a stride of m.

2.2. Tensor Product Formulation of Block Recursive Algorithms

A *block recursive algorithm* is obtained from a recursive tensor factorization of a computation matrix. For example, FFT algorithms are derived by tensor factorization of the discrete Fourier transform (DFT) matrix. The algorithms obtained from tensor factorization are computationally more efficient than those that directly use the unfactorized matrix. For example, computing the DFT of a vector of size N by directly multiplying it by an $N \times N$ DFT matrix requires $O(N^2)$ operations, compared to only $O(N \log N)$ operations using an FFT algorithm. Some other examples of block recursive algorithms are Strassen's matrix multiplication [15, 18], convolution [10], and fast sine/cosine transforms [24].

A tensor product formulation of a block recursive algorithm has the generic form $\prod_{j=1}^k (I_{l_j} \otimes A^{m_j,n_j} \otimes I_{r_j})$, where A^{m_j,n_j} is an $m_j \times n_j$ linear transformation, and $\prod_{i=1}^k F_i$ denotes $F_k \cdots F_1$. The computation performed at each step j is $U_j = (I_{l_j} \otimes A^{m_j,n_j} \otimes I_{r_j})(V_j)$. Due to the presence of identity terms, it is easy to express each computation step using parallel operations. However, the task of harnessing this inherent parallelism in each computation step with the goal of minimizing communication cost for the entire computation is nontrivial. We next present tensor product formulations of two FFT algorithms which are used as examples in this paper.

Fast Fourier Transform. The tensor product formulations of various FFT algorithms are presented in [17, 24]. These formulations are obtained by different tensor factorizations of the discrete Fourier transform matrix. Although all of these algorithms are computationally equivalent, they have different computational structures and different data access patterns. For example, consider the following tensor product formulation of the radix-2 decimation-in-time Cooley–Tukey FFT:

$$F_{2^n} = \left(\prod_{i=1}^n (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(I_{2^{n-i}} \otimes T_{2^{i-1}}^{2^i}) \right) R_{2^n}, \qquad (1)$$
$$R_{2^n} = \prod_{i=1}^n (I_{2^{i-1}} \otimes L_2^{2^{n-i+1}}), \quad \text{and} \quad F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

$T_{2^{i-1}}^{2^i}$ represents a diagonal matrix of constants and R_{2^n} permutes the input sequence to a bit-reversed order. As can be seen from Eq. (1), for an FFT on 2^n points, there are n steps in the computation after performing the initial bit-reversal permutation. At each step, the data array from the previous step is scaled by multiplying by twiddle factors $Y = (I_{2^{n-1}} \otimes T_{2^{i-1}}^{2^i})(X_{i-1})$, followed by the butterfly computation $X_i = (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(Y)$.

As another example, consider the tensor product formulation of the radix-2 decimation-in-time Stockham FFT [23]:

$$F_{2^n} = \prod_{i=1}^{n} ((F_2 \otimes I_{2^{n-1}})(T^{2^i}_{2^{i-1}} \otimes I_{2^{n-i}})(L^{2^i}_2 \otimes I_{2^{n-i}})). \quad (2)$$

As in the case of the Cooley–Tukey FFT, the Stockham FFT on 2^n points also has n steps. At step i, an address-bit permutation corresponding to a right cyclic shift of the most significant i address bits is performed on the input array. In the tensor product formulation, this permutation corresponds to performing the computation $Y = (L^{2^i}_2 \otimes I_{2^{n-i}})(X_{i-1})$. Then, Y is multiplied by appropriate twiddle factors $Z = (T^{2^i}_{2^{i-1}} \otimes I_{2^{n-i}})(Y)$ and a butterfly operation F_2 is performed with the lower half and upper half of Z as the two inputs to the butterfly operation $X_i = (F_2 \otimes I_{2^{n-1}})(Z)$. The Stockham FFT is an especially important FFT algorithm as it has the bit-reversal permutation implicitly embedded in its computation and therefore does not require the extra bit-reversal permutation needed in other FFT algorithms.

3. PROGRAMMING MODELS FOR THE TARGET CODE

Synthesizing a distributed-memory program from a tensor product formula involves identifying data distributions for the arrays, partitioning the computation, determining the communication, and generating a node program containing explicit communication commands. On a distributed-memory machine, it is important to minimize communication overhead to achieve high-performance.

As illustrated in the previous section, a tensor product formulation of a block recursive algorithm consists of a matrix product of several factors: $\mathscr{F} \equiv \prod_{j=1}^{n} F_j$. Each factor F_j is a tensor product and corresponds to a computation step in the algorithm. At the jth step, the computation performed is $X^{(o)}_j = F_j(X^{(i)}_j)$, where $X^{(i)}_j$ is the input and $X^{(o)}_j$ is the output of step j. The output from the jth computation step becomes the input of the next computation step; i.e., $X^{(i)}_{j+1} = X^{(o)}_j$. On a distributed-memory machine, both the input and output arrays are distributed among the local memories of the interconnected processors. The computation is partitioned to exploit data-parallelism. The distribution of the arrays and the computation partitioning determine the communication required to perform a computation step. At a computation step, a processor may require communication for obtaining the required inputs and storing the resulting outputs. A commonly used strategy for computation partitioning is the owner computes rule: the processor on which a data element is resident (which owns a data element) performs all the computation which modify it. Under the owner computes rule, which is also used in this paper, no communication is required to store the outputs of a computation step.

The distributions for the input and output array of each computation step should be such that the resulting communication overhead is minimized. For a computation step j, the distributions of $X^{(i)}_j$ and $X^{(o)}_j$ which permit a communication-free implementation of F_j are the most desirable distributions. The choice of communication-free distributions for a computation step is constrained by the data distribution requirements of the adjoining steps. For example, it is desirable that the output distribution for F_j should be a suitable input distribution for a communication-free implementation of F_{j+1}. If no such output distribution for F_j exists, then either of the following strategies can be used:

1. Point-to-Point strategy: Let the input distribution for F_{j+1} be the same as the output distribution for F_j, i.e., the arrays $X^{(o)}_j$ and $X^{(i)}_{j+1}$ have the same distribution. Then no communication is needed for performing the computation $X^{(o)}_j = F_j(X^{(i)}_j)$. However, communication is needed while performing the computation $X^{(o)}_{j+1} = F_{j+1}(X^{(i)}_{j+1})$.
2. Redistribution strategy: Choose different distributions for the output of F_j and input of F_{j+1}, such that the computations $X^{(o)}_j = F_j(X^{(i)}_j)$ and $X^{(o)}_{j+1} = F_{j+1}(X^{(i)}_{j+1})$ are communication-free. However, since $X^{(i)}_{j+1} = X^{(o)}_j$, communication is needed to redistribute the output of F_j to the distribution chosen for the input of F_{j+1}.

These two strategies correspond to different approaches to managing the distribution of the data. The point-to-point strategy implies that the distribution of the arrays should be static, whereas the redistribution strategy implies that the distribution of the arrays can be dynamic. We develop program generation techniques for both target program models. In a program using the point-to-point model, the distribution of an array is fixed and so communication is needed whenever a processor requires data elements which it does not own. This results in programs with explicit point-to-point communication primitives. On the other hand, in a program using the redistribution model, the distribution of the data arrays is dynamically changed to ensure that computation is localized in every computation step. A data redistribution usually involves a collective all-to-many communication primitive. Hence, the programs synthesized under these two models have different communication overhead characteristics, which results in programs with different performance behavior.

Synthesizing programs under either model requires the determination of data distribution for the arrays. For determining data distributions which will minimize the communication overhead, each computation step in a tensor product formula is analyzed to determine its data access pattern. This data access pattern is summarized in an algebraic form which is used to determine the distributions, if any, which will permit a communication-free implementation of the associated computation step. If no such distribution exists, then the algebraic representation is used to quantify the communication overhead resulting from distributing the arrays in a particular manner. In order to achieve a communication-efficient implementation of the entire tensor product formula, the data access information

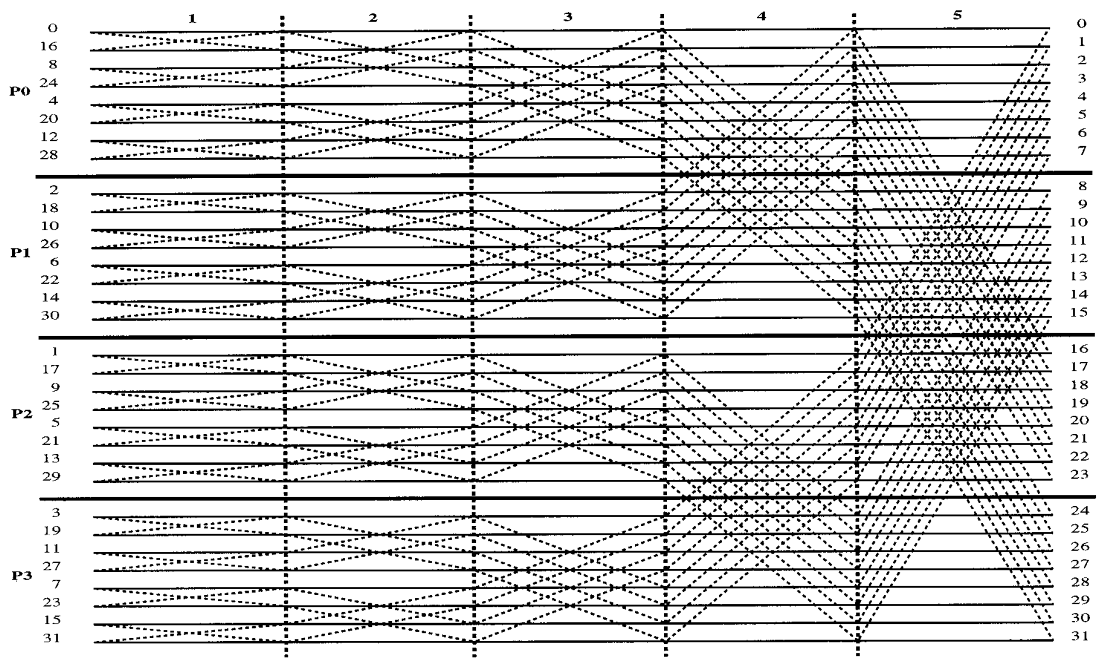

FIG. 1. Data flow for the Cooley–Tukey FFT under the point-to-point model.

for all the steps is analyzed to determine the distributions which will result in a minimal communication overhead. Once the initial distributions for the arrays are fixed, a node program under the point-to-point model can be synthesized. In the synthesized program, communication is needed for those steps which are not communication-free with respect to the initial distribution of the arrays. For synthesizing a program under the redistribution model, the tensor product formula is partitioned into communication-free phases. Each phase consisting of several computation steps which can be completely localized by appropriately choosing the distribution for their input and output arrays. Redistribution is required at the boundary of these phases. A redistribution generally involves all-to-many personalized communication which can be very costly. In order to achieve communication-efficient target code, the number of redistributions is minimized.

To illustrate the differences in the data flow pattern resulting from programs synthesized under these two models, the data flow patterns corresponding to the synthesized programs for a 32 point Cooley–Tukey FFT on four processors are presented here. The input array is assumed to be in bit-reversed order and block distributed on four processors. The data flow pattern for the program synthesized under the point-to-point model is shown in Fig. 1. The first three steps of the computation are communication-free, but communication is needed for the remaining two steps as indicated by lines crossing processor boundaries. The data flow pattern for the program synthesized under the redistribution model is shown in Fig. 2. As in the case of the program using point-to-point communication, the first three steps of the computation are communication-free. However, by performing a single redistribution, the remaining two steps have been made communication-free.

We will next present a tensor product based algebraic representation, called the distribution basis, to describe the semantics of regular data distributions [12].

4. ALGEBRAIC SEMANTICS OF DATA DISTRIBUTIONS

The most common regular distributions used for arrays are the block, cyclic, and block-cyclic distributions. These distributions are used in Fortran D [3, 13], Vienna Fortran [5], High Performance Fortran (HPF) [8], and pC++ [1]. Block and cyclic distributions may be viewed as special cases of the block-cyclic distribution. A block-cyclic distribution partitions an array into equal sized blocks of consecutive elements and then maps them to the processors in a cyclic manner. The elements mapped to a processor are stored in increasing order of their indices in its local memory. We use the following convention to express the block-cyclic distribution of a one-dimensional array $A(0:N-1)$ of size N on P processors:[7]

- $A(\text{cyclic}(B))$: the block size is B and element $A(k)$ at *global index* k is on processor with *processor index* $p = (k \text{ div } B) \bmod P$ at *local index* $l = (k \bmod B) + (k \text{ div } (PB))B$.[8]

[7] We assume that N is a multiple of P. In case N is not divisible by P, the array can be assumed to be of size N', where N' is the smallest integer greater than N which is divisible by P. The elements at indices greater than N are treated as dummy elements and no space needs to be actually allocated for them.

[8] The operator div represents the integer division operation.

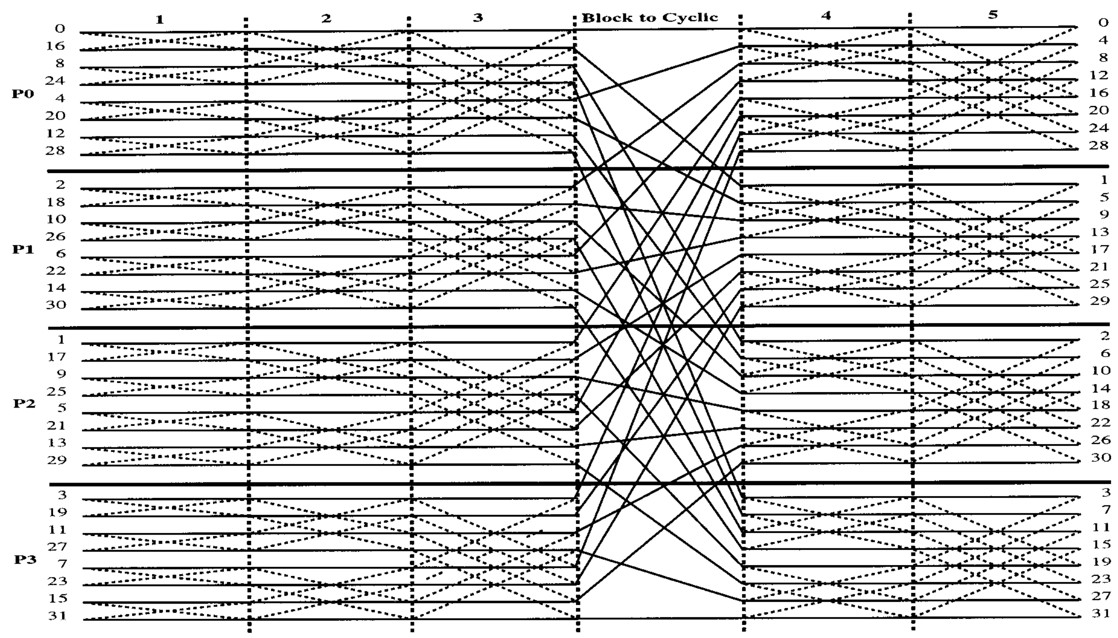

FIG. 2. Data flow for the Cooley–Tukey FFT under the redistribution model.

The block and cyclic distribution of array A are denoted by A(block) and A(cyclic), respectively. A(block) is equivalent to A(cyclic(N/P)) and A(cyclic) is equivalent to A(cyclic(1)). The first three entries of Table I gives examples of block, cyclic, and cyclic(2) distributions of an array of size 16 over four processors. The global indices mapped to each of the four processors P_0 to P_3 are shown in increasing order of their local indices under the last four columns of the table.

Before we present the algebraic semantics of data distributions, we describe the following HPF language constructs which are used to express the distribution and rearrangement of data in a global address space. The DISTRIBUTE directive is used to express the initial distribution of an array. Each array is distributed on a virtual processor array which is declared using the PROCESSORS directive. The REDISTRIBUTE directive is used to change the distribution of an array. For an array to be redistributed, it must be declared to be DYNAMIC. For example, the following code declares array A distributed using a block distribution and array C distributed using a block-cyclic distribution with a block size of B on a linear processor array PROC of size P. Furthermore, C is declared as a dynamic array and can be redistributed in the program:

```
PROCESSORS PROC(P)
REAL   A(N), C(N)
DISTRIBUTE A(block) ONTO PROC
DISTRIBUTE C(cyclic(B)) DYNAMIC  ONTO  PROC
...
REDISTRIBUTE C(block) ONTO  PROC
```

An algebraic representation for a block-cyclic distribution of $A(0:N-1)$ on P processors can be obtained by associating index k of A with vector basis e_k^N [12]. Associating indices of an array with vector bases helps in determining the indexing needed when the array is segmented or viewed as a multidimensional array. For example, consider the segmented view of an array of size N corresponding to block distribution on P processors.

TABLE I
Examples of Data Distributions for an Array of Size 16 on Four Processors

Array view	Proc. view	Distribution	Dist. basis	P_0	P_1	P_2	P_3
1d: 16	1d: 4	(block)	$\rho_p^4 \otimes e_l^4$	0, 1, 2, 3	4, 5, 6, 7	8, 9, 10, 11	12, 13, 14, 15
1d: 16	1d: 4	(cyclic)	$e_l^4 \otimes \rho_p^4$	0, 4, 8, 12	1, 5, 9, 13	2, 6, 10, 14	3, 7, 11, 15
1d: 16	1d: 4	(cyclic(2))	$e_{l_2}^2 \otimes \rho_p^4 \otimes e_{l_1}^2$	0, 1, 8, 9	2, 3, 10, 11	4, 5, 12, 13	6, 7, 14, 15
2d: 8 × 2	2d: 2 × 2	(block, block)	$\rho_{p_2}^2 \otimes e_l^4 \otimes \rho_{p_1}^2$	0, 2, 4, 6	1,3, 5, 7	8, 10, 12, 14	9, 11, 13, 15
2d: 4 × 4	2d: 2 × 2	(block, block)	$\rho_{p_2}^2 \otimes e_{l_2}^2 \otimes \rho_{p_1}^2 \otimes e_{l_1}^2$	0, 1, 4, 5	2, 3, 6, 7	8, 9, 12, 13	10, 11, 14, 15
2d: 4 × 4	2d: 2 × 2	(cyclic, cyclic)	$e_{l_2}^2 \otimes \rho_{p_2}^2 \otimes e_{l_1}^2 \otimes \rho_{p_1}^2$	0, 2, 8, 10	1, 3, 9, 11	4, 6, 12, 14	5, 7, 13, 15

The relationship between the global index k and indices p and l can be algebraically represented by the identity

$$e_k^N = e_p^P \otimes e_l^B,$$

where $B = N/P$, $p = k$ div B, and $l = k$ mod B.

Next we consider the block-cyclic distribution cyclic (B) of an array consisting of N elements on P processors. Suppose that an array of size N is segmented into N/B blocks; each of size B. Under cyclic(B) distribution, these blocks are grouped into $N/(PB)$ groups, each group consisting of P blocks. This corresponds to tensor factorization $e_c^{N/(PB)} \otimes e_p^P \otimes e_b^B$ of e_k^N, where $c = k$ div PB, $p = (k$ div $B)$ mod P, and $b = k$ mod B.

We denote by ρ_p^P any vector basis e_p^P used for processor indexing. The *distribution basis* for cyclic(B) distribution is $e_c^{N/(PB)} \otimes \rho_p^P \otimes e_b^B$. We next define the indexing functions for a block-cyclic distribution basis. These indexing functions select vector bases of the distribution basis, which determine the processor address, the local address, and the global address of an index of the distributed array.

Definition 4.1 (Indexing Functions). For a one-dimensional array of size N, let the distribution basis be $e_c^C \otimes \rho_p^P \otimes e_b^B$, where $C = N/(PB)$. Let elements assigned to a processor be stored in the local array in increasing order of their global indices. Then the global index function, `global()`, the local index function, `local()`, and the processor index function, `proc()`, are defined as follows:

$$\begin{aligned}
&\texttt{global}(e_c^C \otimes \rho_p^P \otimes e_b^B) = e_{PBc+Bp+b}^N,\\
&\texttt{local}(e_c^C \otimes \rho_p^P \otimes e_b^B) = e_{Bc+b}^{CB},\\
&\texttt{proc}(e_e^C \otimes \rho_p^P \otimes e_b^B) = \rho_p^P.
\end{aligned}$$

In general, a distribution basis can have more than one ρ basis. These distributions correspond to viewing a linear array as a multi-dimensional array with block-cyclic distribution along each dimension. For example, Table I shows all the distributions for an array of size 16 on four processors which can be expressed by a distribution basis. The first three distributions correspond to one-dimensional views of the array with block, cyclic, and block-cyclic distribution on a one-dimensional processor array. The fourth distribution basis, $\rho_{p_2}^2 \otimes e_l^4 \otimes \rho_{p_1}^2$, corresponds to viewing the array as an 8×2 array with block distribution along both the dimensions of a 2×2 processor grid. Note that many multidimensional block-cyclic distributions of a linear array can result in the same distribution basis. For example, the distribution basis $\rho_{p_2}^2 \otimes e_l^4 \otimes \rho_{p_1}^2$ also corresponds to viewing the array as a 2×8 array distributed over a 2×2 processor grid with block distribution along the first dimension and cyclic distribution along the second dimension. We next define the indexing functions for tensor product of block-cyclic distribution bases.

Definition 4.2 (Indexing Functions). Let δ_i be a block-cyclic distribution basis for a one-dimensional array of size N_i distributed on P_i processors, where $1 \leq i \leq k$. The distribution basis $\delta \equiv \delta_k \otimes \cdots \otimes \delta_1$ corresponds to a distribution of an array of size $N = N_k \cdots N_1$ on a processor array of size $P = P_k \cdots P_1$. The indexing functions are defined as follows:

$$\begin{aligned}
&\texttt{global}(\delta) = \texttt{global}(\delta_k) \otimes \cdots \otimes \texttt{global}(\delta_1),\\
&\texttt{local}(\delta) = \texttt{local}(\delta_k) \otimes \cdots \otimes \texttt{local}(\delta_1),\\
&\texttt{proc}(\delta) = \texttt{proc}(\delta_k) \otimes \cdots \otimes \texttt{proc}(\delta_1).
\end{aligned}$$

Similarly, the distribution of a multidimensional array on a multidimensional mesh can be formalized as a tensor product of the distribution bases along each dimension. We next present a characterization of distribution bases which permit a communication-free implementation of a computation expressed by a tensor product.

5. COMMUNICATION-FREE COMPUTATIONS

In implementing a computation on a distributed-memory multiprocessor, it is important to determine appropriate distributions for data arrays, in order to minimize communication overhead. If possible, distributions that totally eliminate communication should be used. In this section, we characterize distribution bases, if any, which permit communication-free implementation of the computation expressed by a tensor product. This characterization is later used to determine data distributions for communication-efficient implementation of a tensor product formula on a distributed-memory multiprocessor.

We begin with a characterization for communication-free computations. We will refer to the distribution basis of an input array as an *input distribution basis* and denote it as $\delta^{(i)}$. Similarly, an *output distribution basis*, $\delta^{(o)}$, refers to the distribution basis of an output array. In order to achieve a communication-free implementation of a tensor product formula, the computation corresponding to it must be appropriately partitioned. As illustrated in Section 2, an identity matrix in a tensor product can be used to partition the computation to obtain parallel operations. However on a distributed-memory machine, in order to avoid communication, the input and the output arrays should be distributed such that the inputs and outputs of a computation performed on each processor are local to that processor. The following definition specifies the requirements for a communication-free computation:

Definition 5.1 (Communication-Free Computation). Let arrays U and V be distributed across P processors, with distribution bases $\delta^{(o)}$ and $\delta^{(i)}$, respectively. Let $\{U_p : 0 \leq p < P\}$ and $\{V_p : 0 \leq p < P\}$ be partitions of arrays U and V, with U_p and V_p mapped to processor p under distribution bases $\delta^{(o)}$ and $\delta^{(i)}$, respectively. Then a tensor product formula $\mathscr{F}$ is said to be *communication-free* with respect to $\delta^{(i)}$ and $\delta^{(o)}$ if the computation $U = \mathscr{F}(V)$ can be performed as P independent subcomputations $U_p = \mathscr{F}_p(V_p)$, $0 \leq p < P$.

As we are interested in implementing block recursive algorithms on a distributed-memory machine, we consider only the tensor products of the form $F \equiv (I_r \otimes A^{m,n} \otimes I_s)$. The simplest form of a tensor product is one in which $A^{m,n}$ is an identity matrix. Obviously, for such a tensor product, both the input and output distribution bases should be identical in order to eliminate communication. The following theorem gives a characterization of the input and output distribution bases which permit communication-free implementations of F, for an arbitrary computation matrix.

THEOREM 5.1. *Let $A^{m,n}$ be an arbitrary computation matrix. The computation $U^{rms} = (I_r \otimes A^{m,n} \otimes I_s)\,(V^{rns})$ can be performed in a communication-free manner with the input distribution basis as $\delta_1 \otimes e_j^n \otimes \delta_2$ and the output distribution basis as $\delta_1 \otimes e_l^m \otimes \delta_2$, where δ_1 and δ_2 are distribution bases for arrays of sizes r and s, respectively. Furthermore, the local computation performed on each node is $u^{rms/P} = (I_{r/P_1} \otimes A^{m,n} \otimes I_{s/P_2})\,(v^{rns/P})$, where $u^{rms/P}$ and $v^{rns/P}$ are the local arrays corresponding to arrays U and V, respectively, P_1 and P_2 are such that* $\mathtt{proc}(\delta_1) = \rho_p^{P_1}$ *and* $\mathtt{proc}(\delta_2) = \rho_q^{P_2}$, *and the total number of processors $P = P_1P_2$.*

Proof. The computation $U^{rms} = (I_r \otimes A^{m,n} \otimes I_s)\,(V^{rns})$ can be implemented as follows:

```
DO $i = 0, r-1$
  DO $k = 0, s-1$
    DO $l = 0, m-1$
      DO $j = 0, n-1$
        $U(ims + ls + k) = U(ims + ls + k) + A^{m,n}(l,j) *$
        $V(ins + js + k)$
ENDDO  ENDDO  ENDOO  ENDDO
```

However, the input vector V^{rns} can be viewed as a three-dimensional array $V^{r,n,s}$, where $V^{r,n,s}(i,j,k) = V^{rns}(ins + js + k)$. Similarly, the output vector U^{rms} can be viewed as a three-dimensional array $U^{r,m,s}$, where $U^{r,m,s}(i,l,k) = U^{rms}(ims + ls + k)$. Then the code for $U^{rms} = (I_r \otimes A^{m,n} \otimes I_s)\,(V^{rns})$ can be expressed as follows:

```
DO $i = 0, r-1$
  DO $k = 0, s-1$
    DO $l = 0, m-1$
      DO $j = 0, n-1$
        $U(i,l,k) = U(i,l,k) + A^{m,n}(l,j) * V(i,j,k)$
ENDDO  ENDDO  ENDDO ENDDO
```

Note that the i and k loops are completely parallel. This computation can be mapped onto a P-processor distributed-memory machine as follows:

- View the P processors as forming a logical two-dimensional $P_1 \times P_2$ grid; processor p has indices (p div P_2, p mod P_2) of the 2-d grid.
- Distribute the first dimensions of both the input and the output arrays identically over P_1 processors along the first dimension of the processor grid using the distribution corresponding to δ_1. Distribute the third dimensions of the input and output arrays identically over P_2 processors along the second dimension of the processor grid using the distribution corresponding to δ_2. Note that the second dimension of both the arrays has been sequentialized. Thus the array elements $U(i_1, l_1, k_1)$ and $U(i_2, l_2, k_2)$ are assigned to the same processor if $i_1 = i_2$ and $k_1 = k_2$. The distribution bases for a multidimensional array is the tensor product of the distribution basis along each dimension. Hence, the distribution bases for $U^{r,m,s}$ and $V^{r,n,s}$ are $\delta_1 \otimes e_l^m \otimes \delta_2$ and $\delta_1 \otimes e_j^n \otimes \delta_2$, respectively. The elements of arrays U and V assigned to a processor can be viewed as three-dimensional arrays $u^{r/P_1,m,s/P_2}$ and $v^{r/P_1,n,s/P_2}$, respectively.
- Partition the i and k loops to match the partitioning of arrays U and V along the first and third dimension, respectively. Therefore, each node will perform the following computation:

```
DO $i' = 0, r/P_1 - 1$
  DO $k' = 0, s/P_2 - 1$
    DO $l = 0, m-1$
      DO $j = 0, n-1$
        $u(i', l, k') = u(i', l, k') + A^{m,n}(l,j) * v(i', j, k')$
ENDDO  ENDDO  ENDDO ENDDO
```

Linearizing the local arrays gives the node code as shown in Fig. 3. The node code shown in Fig. 3 corresponds to the the tensor product $u^{rms/P} = (I_{r/P_1} \otimes A^{m,n} \otimes I_{s/P_2})\,(v^{rns/P})$. ■

Hence, according to Theorem 5.1, a tensor product F, with $\delta^{(i)}$ and $\delta^{(o)}$ as the distribution bases, can be implemented in a communication-free manner if

- $\mathtt{proc}(\delta^{(i)}) = \mathtt{proc}(\delta^{(o)})$, i.e., indices corresponding to the same vector bases are chosen in both the input and output basis for the processor indexing.
- The nonidentity operators in F correspond to e bases in both the input and output distribution bases. For example, $F \equiv (I_2 \otimes F_2 \otimes I_2)$ is not communication-free with respect to the input distribution basis $e_i^2 \otimes \rho_j^2 \otimes e_k^2$ because applying F to the input distribution basis results in $I_2(e_i^2) \otimes F_2(\rho_j^2) \otimes I_2(e_k^2)$,[9] i.e., the computation matrix F_2 corresponds to the vector basis chosen for the processor indexing.

The local computation corresponding to $F \equiv I_r \otimes A^{m,n} \otimes I_s$, which is communication-free with respect to $\delta^{(i)}$, will be denoted by $F|\mathtt{local}(\delta^{(i)})$. The local computation corresponding to F is of the form $I_{r'} \otimes A^{m,n} \otimes I_{s'}$, where $r's' = rs/P$. The corresponding node code is synthesized using the local input basis $\mathrm{local}(\delta^{(i)}) = e_{i'}^{r'} \otimes e_j^n \otimes e_{k'}^{s'} = e_{ns'i'+s'j+k'}^{r'ns'}$ and the local output basis $\mathrm{local}(\delta^{(o)}) = e_{i'}^{r'} \otimes e_l^m \otimes e_{k'}^{s'} = e_{ms'i'+s'l+k'}^{r'ms'}$. The local input basis determines the indexing for the input array and the local output basis

[9] Using the identity $(A^{m,n} \otimes B^{p,q})(e_i^n \otimes e_j^q) = A^{m,n}\,(e_i^n) \otimes B^{p,q}(e_j^q)$.

```
DO $i' = 0, r' - 1$
  DO $k' = 0, s' - 1$
    DO $l = 0, m - 1$
      DO $j = 0, n - 1$
        $u(ms'i' + s'l + k') = u(ms'i' + s'l + k') + A^{m,n}(l, j) * v(ns'i' + s'j + k')$
ENDDO  ENDDO  ENDDO ENDDO
```

FIG. 3. Node code for $U^{rms} = (I_r \otimes A^{m,n} \otimes I_s)(V^{rms})$ on P processors ($P = P_1P_2$, $r' = r/P_1$, $s' = s/P_2$).

determines the indexing for the output array. The synthesized node program is as shown in Fig. 3. In the node program, there is a loop corresponding to each of the four indices i', j, k', and l. Different implementations can be obtained by changing the ordering of these four loops as they are fully permutable. However, different orderings of the loops will result in different data access patterns. This will result in codes with different performance characteristics on processors with cache memory.

When F is a tensor permutation, i.e., it is of the form $F \equiv I_r \otimes L^n_m \otimes I_s$, the local computation amounts to a local rearrangement of data. This local permutation can be determined from the local input basis local$(\delta^{(i)}) = e^{r'}_i \otimes e^{n/m}_j \otimes e^m_l \otimes e^{s'}_k = e^{r'ns'}_{ns'i+ms'j+s'l+k}$ and the output basis local$(\delta^{(o)}) = e^{r'}_i \otimes e^m_l \otimes e^{n/m}_j \otimes e^{s'}_k = e^{r'ns'}_{ns'i+(n/m)s'l+s'j+k}$. The synthesized loop nest consists of four loops; one each for indices i, j, k, and l. The body of the loop nest is a single assignment statement which copies the element at index $(ns'i + ms'j + s'l + k)$ of the input array to the location at index $(ns'i + (n/m)s'l + s'j + k)$ of the output array.

We now consider code generation for $\mathscr{F} \equiv \prod^n_{j=1} F_j$. The computation $U = \mathscr{F}(V)$ can be implemented without any need for communication if each F_j is communication-free with respect to its input and output distribution basis. Node code with a single loop nest can be generated for $U = \mathscr{F}(V)$. The node code will have a outer j loop. The body of the j loop will consist of the loop nest for the local code corresponding to the generic tensor product F_j.

In the remaining paper, we consider code generation for a non-communication-free tensor product formula. First, we will present code generation under the point-to-point model and the redistribution model described in Section 3, with the assumption that the distributions for all the arrays to be used in the generated code are given to the code generator. Then we will show how to determine the initial distributions for the arrays. It will turn out that the procedure for determining redistributions can be easily extended to determine the initial data distributions as well.

6. GENERATING PROGRAMS WITH POINT-TO-POINT COMMUNICATION

In this section, we present a methodology for generating programs with explicit communication commands from a tensor product formula. We present the program generation under the assumption that the distribution for the arrays are given.

We begin with determination of the communication and the resulting node computation for a tensor product F, given that the input distribution basis is $\delta^{(i)}$ and the output distribution basis is $\delta^{(o)}$. As indicated in the previous section, communication may be needed when a computation matrix corresponds to a processor basis in the input distribution basis. In the generated node code, all the necessary communication will be performed before performing the computation. For determining the required communication, each processor p needs to determine the following information:

- Send processor set: set of processors to which p has to send data.
- Send data index set: indices of the array elements which are resident on p but are needed by a processor q in the send processor set.
- Receive processor set: set of processor from which p receives data.
- Receive data index set: indices of the array elements where the data in the message from a processor q in the receive processor set is to be stored.

As we will show below, the communication for F will depend on the structure of the computation matrix in F. The send processor set for a processor p is determined by the positions of non-zero entries in a particular column of the computation matrix. Similarly, the receive processor set for p is determined by the position of non-zero entries in a particular row of the computation matrix. Furthermore, each message will consist of all the local elements on the sender processor. The processor receiving the message will store it in a temporary array.

The code for performing the communication and the computation is determined as follows. For simplicity, the following discussion is restricted to the generic tensor product $F \equiv (I_r \otimes A_m \otimes I_t)$, where A_m is an arbitrary $m \times m$ computation matrix. Let the input distribution basis $\delta^{(i)}$, with $\mathtt{proc}(\delta^{(i)}) = \rho^P_p$, be expressed as $\delta_1 \otimes \rho^{P_2}_{p_2} \otimes \delta_2$, where $P_2 = m$, and δ_1 and δ_2 are distribution bases for arrays of sizes r and t, respectively. Let $\mathtt{proc}(\delta_1) = \rho^{P_1}_{p_1}$, and $\mathtt{proc}(\delta_2) = \rho^{P_3}_{p_3}$. Since $\mathtt{proc}(\delta^{(i)}) = \mathtt{proc}(\delta_1) \otimes \rho^{P_2}_{p_2} \otimes \mathtt{proc}(\delta_2)$, we get $\rho^P_p = \rho^{P_1}_{p_1} \otimes \rho^{P_2}_{p_2} \otimes \rho^{P_3}_{p_3} = \rho^P_{P_3P_2p_1+P_3p_2+p_3}$. The indices p_1, p_2, and p_3 can be computed from the processor index p as follows: $p_1 = p$ div P_3P_2, $p_2 = (p \bmod P_3P_2)$ div P_3, and $p_3 = p \bmod P_3$. The send processor set for processor p can be determined as follows:

• Instantiate indices p_1 and p_3 in $\texttt{proc}(\delta^{(i)})$ according to p.

• Vary index p_2 to determine the processors which must receive a message from p. A processor q corresponding to $p_2 = a$, i.e., $q = P_3P_2p_1 + P_3a + p_3$, receives a message from processor p if $A_m(a, b) \neq 0$, where $p_2 = b$ corresponds to processor p.

• Each message contains the entire set of local elements of the input array on processor p.

Similarly, the receive processor set for processor p will be all processors q corresponding to $p_2 = a$, such that $A_m(b, a) \neq 0$. After communication, each processor performs the computation corresponding to the tensor product $(I_r|\texttt{local}(\delta_1)) \otimes A_m(b, :) \otimes (I_t|\texttt{local}(\delta_2))$ in its local space, where $A_m(b, :)$ is the bth row of A_m. Let τ be the maximum number of non-zero entries in any row or column of A_m. If t_s is the message setup time and t_p is the message transmission time per data element, then the associated communication cost can be estimated to be $\tau \cdot (t_s + (N/P)t_p)$, where $N = rmt$ and P is the number of processors used.

We now describe the procedure for synthesizing a node program with explicit communication commands for a tensor product formula $\mathscr{F} \equiv \prod_{j=1}^{n} F_j$, where F_j is a tensor product. The main steps of the synthesis procedure are as follows:

1. Identify computation steps that can be executed locally. This is done by applying the distribution basis of the input vector to the tensor product F_j in the formula and then finding the conditions under which the bases used for the processor indexing remain intact. This categorizes the ranges of iteration index j as either communication-free or non-communication-free.

2. Determine communication commands for computation steps involving communication. This involves determination of:

   (a) The data to be sent from a processor's local data space along with the receiver processor's id.

   (b) The data to be received in a processor's local data space along with the sender processor's id.

3. Generate node code using the information generated in steps 1 and 2. The node program for the tensor product formula $\mathscr{F}$ will consist of a sequence of loops. Each loop corresponds to a sequence of adjacent communication-free computation steps or a sequence of adjacent non-communication-free computation steps.

For the entire tensor product formula $\mathscr{F}$, the communication cost will be the summation of the communication cost for non-communication-free steps.

6.1. An Example: Cooley–Tukey FFT

We now illustrate the above procedure by determining communication for the 2^n points Cooley–Tukey decimation-in-time FFT on $P = 2^m$ processors. For simplicity, we will assume that twiddle factors are replicated on all the processors and so the scaling by twiddle factors require no communication. Therefore, for the purpose of analysis consider the following formula, which characterizes the core computation in the Cooley–Tukey FFT Formula (Eq. 1):

$$A = \prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(A).$$

Also, let the distribution of array A be block, i.e., both the input and output distribution bases are characterized by $\delta = \rho_p^{2^m} \otimes e_l^{2^{n-m}}$. For any step i such that $i \leq n - m$, F_2 does not correspond to a processor basis. Therefore, the first $n - m$ steps are communication-free. However, the remaining m steps are not communication-free. The structure of the generated code is shown in Fig. 4.

```
/* $A$ has a *block* distribution on $2^m$ processors*/
/* $A'$ is the local array corresponding to $A$ */
DO $i = 1, n - m$
  $A' = (I_{2^{n-m-i}} \otimes T^{2^i}_{2^{i-1}})(A')$
  $A' = (I_{2^{n-m-i}} \otimes F_2 \otimes I_{2^{i-1}})(A')$
ENDDO
$p = myid()$
DO $i = n - m + 1, n$
  $j = i - (n - m)$
  $P_1 = 2^{m-j}$, $P_2 = 2$, $P_3 = 2^{j-1}$
  $p_1 = p \ div \ P_3P_2$, $p_2 = (p \ mod \ P_3P_2) \ div \ P_3$, $p_3 = p \ mod \ P_3$
  $t = 2 * p_2 + p_3$, $q = (p_2 + 1) \ mod \ 2$
  $A' = T^{2^i}_{2^{i-1}}(t * P_3 : (t + 1) * P_3 - 1, t * P_3 : (t + 1) * P_3 - 1)(A')$
  send $(q, A')$
  recv $(q, buf)$
  IF $(p_2 == 0)$
    THEN $A' = A' + buf$
    ELSE $A' = buf - A'$
  ENDIF
ENDDO
```

FIG. 4. Node code for the Cooley–Tukey FFT using explicit message passing.

7. GENERATING PROGRAMS WITH REDISTRIBUTIONS

In this section, we sketch a procedure for implementing tensor product formulas on a distributed-memory multiprocessor using redistribution commands. We illustrate this procedure by designing communication-efficient implementations for the Cooley–Tukey FFT and the Stockham FFT. We first present a communication cost metric for a redistribution. This communication cost metric is used to guide the program synthesis procedure to generate communication-efficient target code.

7.1. Determining Communication for Redistribution

In terms of a distribution basis, a redistribution corresponds to using a different set of vector bases for processor indexing. For example, consider that the distribution of an array of size P^2 is changed from a block distribution on P processors to a cyclic distribution on the same number of processors. The distribution basis changes from $\rho_p^P \otimes e_q^P$ to $e_p^P \otimes \rho_q^P$. Under the block distribution, index p is used for processor indexing and index q is used for indexing the local array on each processor; but after changing the distribution to a cyclic distribution, the roles of index p and q get interchanged. The communication needed to perform a redistribution can be determined from the source and target distribution bases. For example, in the case of redistribution from $\rho_p^P \otimes e_q^P$ to $e_p^P \otimes \rho_q^P$, processor p sends its qth local element to processor q, which stores it at local index p.

In general, when redistributing from δ_{source} to δ_{target}, the following procedure can be used by a processor p to determine the set of processors it needs to send messages.

1. Instantiate the indices in $\texttt{proc}(\delta_{\text{source}})$ according to its processor id p.
2. The indices in $\texttt{proc}(\delta_{\text{target}})$ which are also present in $\texttt{proc}(\delta_{\text{source}})$ get automatically instantiated in Step 1. The remaining indices can be varied independently within their respective ranges to determine all the processors to which messages are to be sent.

The local indices for elements to be sent to processor q can be determined by using the following procedure:

1. Instantiate the indices in $\texttt{proc}(\delta_{\text{target}})$ according to the target processor id q.
2. The indices in $\texttt{local}(\delta_{\text{source}})$ which are also present in $\texttt{proc}(\delta_{\text{target}})$ get automatically instantiated in Step 1. The remaining indices can be varied independently within their respective ranges to generate all the local indices for elements to be sent to processor q.

Similar procedures can be used to determine the set of processors from which messages are to be received, and the local indices where the elements from each received message are to be stored.

Communication Cost Metric for Redistribution. We next model the communication cost of a redistribution. In order to estimate the cost of a redistribution, we need to determine the number of messages a processor sends and the size of each message. If the redistribution is performed from δ_{source} to δ_{target}, then:

- The number of sends, NumSends(δ_{source}, δ_{target}), is equal to the product of the sizes of vector bases in $\texttt{proc}(\delta_{\text{source}})$ but not in $\texttt{proc}(\delta_{\text{target}})$.
- The size of each message, MsgSize(δ_{source}, δ_{target}), is equal to (N/P)/NumSends(δ_{source}, δ_{target}).

Hence, the cost of redistribution can be estimated as

$$\begin{aligned}\text{Cost}(\delta_{\text{source}}, \delta_{\text{target}}) &= \text{NumSends}(\delta_{\text{source}}, \delta_{\text{target}})\\ &\quad (t_s + \text{MsgSize}(\delta_{\text{source}}, \delta_{\text{target}})t_p)\\ &= \text{NumSends}(\delta_{\text{source}}, \delta_{\text{target}})t_s\\ &\quad + (N/P)t_p.\end{aligned}$$

When $N \gg P$, the dominating term in the above cost metric will be $(N/P)t_p$. Furthermore, this term is independent of the redistribution being performed. Therefore for $N \gg P$, the cost of redistributing is nearly uniform. It will be shown in Section 8 that the program synthesized under the redistribution model performs better than the program synthesized under the point-to-point model when $N \gg P$. Therefore, in synthesizing programs with redistributions, we will assume that $N \gg P$. Under this assumption, minimizing the number of redistributions is a good heuristic to achieve communication-efficient programs.

7.2. Determining Redistributions

We first describe how the tensor product $F \equiv (I_r \otimes A^{m,n} \otimes I_s)$ can be implemented using redistributions. Let the input distribution basis be $\delta^{(i)}$ and the output distribution basis be $\delta^{(o)}$. If F is communication-free with respect to $\delta^{(i)}$ and $\delta^{(o)}$, no redistribution is required. Otherwise, redistribution may be needed to redistribute the input and/or the output array. Redistribution is required for the input array if a processor basis in the input distribution basis corresponds to nonidentity operators in F. Similarly, a redistribution is required for the output array if a processor basis in the output distribution basis corresponds to nonidentity operators in F. If redistribution is required only for the input array, then the target distribution $\delta^{(i)}_{\text{target}}$ can be chosen for the input array such that F is communication-free with respect to $\delta^{(i)}_{\text{target}}$ and $\delta^{(o)}$. The target distribution basis $\delta^{(i)}_{\text{target}}$ can be simply determined by ensuring that the bases in $\texttt{proc}(\delta^{(o)})$ are also the processor bases in $\delta^{(i)}_{\text{target}}$. Similarly, if the redistribution is only required for the output array then $\delta^{(o)}_{\text{target}}$ is chosen such that $\texttt{proc}(\delta^{(o)}_{\text{target}}) = \texttt{proc}(\delta^{(i)})$. If redistribution is required for both the input array and the output array, then the input target distribution $\delta^{(i)}_{\text{target}}$ and the output target distribution $\delta^{(o)}_{\text{target}}$ should be such that F is communication-free with respect to $\delta^{(i)}_{\text{target}}$ and $\delta^{(o)}_{\text{target}}$. This requires that $\texttt{proc}(\delta^{(i)}_{\text{target}}) = \texttt{proc}(\delta^{(o)}_{\text{target}})$ and that no processor basis in $\delta^{(i)}_{\text{target}}$ and $\delta^{(o)}_{\text{target}}$ corresponds to any nonidentity operators in F. This is possible only when there exist a set of bases in both the input and output distribution bases such that:

- They do not correspond to the nonidentity operators in F.
- The product of the dimensions of these bases (chosen for processor indexing) should be equal to the number of processors P.

To satisfy the above criteria, some of the bases in the input and output bases may have to be further factorized. If such a set of bases exists, then $\delta^{(i)}_{\text{target}}$ can be determined from $\delta^{(i)}$ by first changing all the ρ bases back to e bases and then changing the selected e bases to ρ bases. Similarly, $\delta^{(o)}_{\text{target}}$ can be determined from $\delta^{(o)}$. We next describe a mechanism, called VIEWAS, for optimizing redistributions for tensor products with stride permutation operators.

7.2.1. *Optimization using VIEWAS*

On a distributed-memory machine, communication may be needed to implement tensor products involving stride permutations. It is important to optimize the communication for such data rearrangement steps. We will refer to a tensor product of stride permutations as a *tensor permutation*. Note that the identity operator is also considered a stride permutation operator.

The procedure outlined above to determine redistributions for a tensor product treats a stride permutation operator as any other other non-identity computation matrix. However, it is sometimes possible to eliminate the redistribution by using the algebraic properties of the stride permutation operator. For example, consider the tensor permutation $Q \equiv (L_{PR}^{CPR} \otimes I_B)$. Let the input distribution be cyclic(RB) and the output distribution be cyclic(RCB). Therefore, the input and output distribution bases are $\delta^{(i)} \equiv e_c^C \otimes \rho_p^P \otimes e_l^{RB}$ and $\delta^{(o)} \equiv \rho_{p'}^P \otimes e_{l'}^{CRB}$. As the stride operator L_{PR}^{CPR} overlaps with the processor basis in both the input and output distribution bases, it would seem that both the input and output array need to be redistributed. But as we show next, Q can in fact be implemented without redistributing the arrays. To understand how this can be done, consider the basis obtained by applying Q to the input distribution basis $\delta^{(i)}$. To apply Q to $\delta^{(i)}$, we need to first factor the basis e_l^{RB} as $e_r^R \otimes e_b^B$. Applying Q to $\delta^{(i)} \equiv e_c^C \otimes \rho_p^P \otimes e_r^R \otimes e_b^B$ results in $\delta \equiv \delta_p^P \otimes e_r^R \otimes e_c^C \otimes e_b^B$. Note that δ also corresponds to a cyclic(RCB) distribution on P processors. In fact, $\delta^{(o)}$ has to be equal to δ as both are distribution bases for the same output array. Now note that $\texttt{proc}(\delta^{(i)}) = \texttt{proc}(\delta)$. This implies that Q can be implemented on each processor by locally copying the element at local index $\texttt{local}(\delta^{(i)})$ of the input array to the location at local index $\texttt{local}(\delta)$ of the output array. Since $\texttt{local}(\delta^{(i)}) = e_c^C \otimes e_r^R \otimes e_b^B = e_{RBc+Br+b}^{CRB}$ and $\texttt{local}(\delta) = e_r^R \otimes e_c^C \otimes e_b^B = e_{CBr+Bc+b}^{CRB}$, the following is a communication-free node program for $Y = Q(X)$,

```
    DO $r = 0, R-1$
      DO $c = 0, C-1$
        DO $b = 0, B-1$
          $y(CBr + Bc + b) = a(RBc + Br + b)$
  ENDDO  ENDDO  ENDDO,
```

where input array $X(0{:}N-1)$, $N = CPRB$, has a cyclic(RB) distribution and output array Y has a cyclic(RCB) distribution on P processors. On each processor, the local arrays $x(0{:}N/P-1)$ and $y(0{:}N/P-1)$ corresponds to arrays X and Y, respectively.

In general, any tensor permutation Q can be implemented in a communication-free manner if the input distribution basis is $\delta^{(i)}$, the output distribution basis is $Q(\delta^{(i)})$, and $\texttt{proc}(Q(\delta^{(i)})) = \texttt{proc}(\delta^{(i)})$. Whenever this communication-free condition is satisfied, Q can be performed as a local permutation on each processor. The local permutation amounts to copying the element at index $\texttt{local}(\delta^{(i)})$ of the local input array to index $\texttt{local}(Q(\delta^{(i)}))$ of the local output array. We will use the notation $\texttt{VIEWAS}(A, Q)$ to express the fact that the array A has been locally permuted according to Q and its final distribution is viewed as $Q(\delta)$, where δ is the initial distribution of A.

7.2.2. *Compute-Dist-Factor Procedure*

We now consider the implementation of a tensor product formula of the form $\mathscr{F} \equiv \prod_{j=1}^n F_j$, where each F_j is a tensor product. We begin with the following observation: If $\mathscr{F} \equiv F_2 F_1$, then $\mathscr{F}$ is communication-free with respect to the input and output distribution basis pair $(\delta^{(i)}, \delta^{(o)})$ only if there exists a distribution basis δ such that F_1 is communication-free with respect to $(\delta^{(i)}, \delta)$, and F_2 is communication-free with respect to $(\delta, \delta^{(o)})$. Otherwise, $\mathscr{F}$ can be implemented using a redistribution if there exist distribution bases δ_1 and δ_2, such that F_1 is communication-free with respect $(\delta^{(i)}, \delta_1)$ and F_2 is communication-free with respect to $(\delta_2, \delta^{(i)})$. A redistribution from δ_1 to δ_2 will be performed in between the communication-free codes for F_1 and F_2. In general, $\mathscr{F} \equiv \prod_{j=1}^n F_j$ will require at most $n-1$ redistributions; one redistribution after each of the first $n-1$ computation steps. The number of redistributions required can be reduced by factoring $\mathscr{F}$ as $\mathscr{F}_m \cdots \mathscr{F}_1$, $1 \le m \le n$, where each factor $\mathscr{F}_k$ for $1 \le k \le m$ is communication-free with respect to $(\delta_k^{(i)}, \delta_k^{(o)})$, with the restriction that $\delta_1^{(i)}$ is the same as the input distribution basis $\delta^{(i)}$ and $\delta_m^{(o)}$ is the same as the output distribution basis $\delta^{(o)}$.

The Compute-Dist-Factor procedure (see Fig. 5) is used for determining distributions for tensor product formulas of the form $\mathscr{F} \equiv \prod_{j=1}^n F_j$, where F_j is a tensor product. This procedure uses a basis marking mechanism to keep track of the bases which correspond to a nonidentity computation matrix in a tensor product formula. It uses the marker μ to mark any e bases which corresponds to a nonidentity computation matrix in an input basis *ibasis* and an output basis *obasis*. The input basis and output basis for a tensor product $F \equiv I_r \otimes A^{m,n} \otimes I_s$ are $e_i^r \otimes e_j^n \otimes e_k^s$ and $e_i^r \otimes e_{j'}^m \otimes e_k^s$, respectively. The bases in the input and output bases may need to be further factorized in order to meet the various requirements of the algorithm. For example, it is required that $obasis_{j-1} = ibasis_j$, this implies that $obasis_{j-1}$ and $ibasis_j$ should have identical factorization. Furthermore, for the basis marking to work, the $ibasis_j$ and $obasis_j$ should be identical in all bases except those which correspond to a nonidentity matrix in F_j. The β distribution bases are used to determine the redistributions and δ distribution bases are used to determine the local computation corresponding to a computation step. The local computation corresponding to computation step j is $F_j | \texttt{local}(\delta_j^{(i)})$. The array is redistributed from $\beta_k^{(o)}$ to $\beta_{k+1}^{(i)}$ between communication-free computation $\mathscr{F}_k$ and $\mathscr{F}_{k+1}$.

The target code for the tensor product formula $\mathscr{F}$ can be expressed using HPF language constructs as

```
procedure Compute-Dist-Factor
input: $\mathcal{F} \equiv \prod_{j=1}^{n} F_j$, $P$, $\delta^{(i)}$, $\delta^{(o)}$.
output: $\mathcal{F} \equiv \prod_{k=1}^{m} \mathcal{F}_k$, $(\beta_k^{(i)}, \beta_k^{(o)}) : 1 \le k \le m$, $(\delta_j^{(i)}, \delta_j^{(o)}) : 1 \le j \le n$.
begin

  • $start = 1$, $k = 0$.

  • for $j = 1$ to $n$ (number of steps in the computation) do

      – Basis Marking: Compute the marked output basis $obasis_j$ for step $j > 1$ by using the
        marked output basis $obasis_{j-1}$ of the previous step, i.e., $ibasis_j = obasis_{j-1}$. If $j = 1$
        then use the initial input basis $ibasis_1$ to compute the marked output basis $obasis_1$.

          * Apply the factor at step $j$, $F_j$, to the marked output basis of the previous step
            $obasis_{j-1}$. If a factor is a tensor permutation, then permute the corresponding
            vector bases. If a factor has an $r \times s$ non-identity operator, then identify the basis
            on which it operates by changing it from $e_l^s$ to $\mu_l^s$ in the $ibasis_j$. The $obasis_j$ is
            obtained from $ibasis_j$ by replacing $\mu_l^s$ by $\mu_l^r$.

      – Formula Splitting: If there does not exist a set of $e$-bases such that:

          1. the product of dimensions of $e$-bases is equal to the number of processors $P$, and
          2. choosing these bases as processor bases ensures that $\mathtt{proc}(ibasis_{start}) =$
             $\mathtt{proc}(obasis_l)$; $start \le l \le j$. If $start = 1$ then, $\mathtt{proc}(ibasis_1) = \mathtt{proc}(\delta^{(i)})$. If
             $j = n$ then, try choosing the bases in $\mathtt{proc}(\delta^{(o)})$ as the processor bases.

        then split the formula before step $j$: $k = k + 1$ and $\mathcal{F}_k = \prod_{l=start}^{j-1} F_l$. Determine
        $(\delta_l^{(i)}, \delta_l^{(o)})$ by marking the appropriate vector bases in $(ibasis_l, obasis_l)$ as the processor
        bases. Set $(\beta_k^{(i)}, \beta_k^{(o)}) = (\delta_{start}^{(i)}, \delta_{j-1}^{(o)})$. Obtain the marked output basis of step $j$ by using
        a fresh or totally unmarked input basis. Set $start = j$.

end procedure
```

FIG. 5. Procedure for determining appropriate redistributions for a tensor product formula.

```
PROCESSORS PROC($P$)
DISTRIBUTE $V_1(\mathrm{dist}(\beta_1^{(o)}))$, ..., $V_m(\mathrm{dist}(\beta_m^{(o)}))$,
  $V(\mathrm{dist}(\beta_1^{(i)}))$ DYNAMIC ONTO PROC
$V_1 = \mathcal{F}_1(V)$
DO $k = 2, m$
  REDISTRIBUTE $V_{k-1}(\mathrm{dist}(\beta_k^{(i)}))$ ONTO PROC
  $V_k = F_k(V_{k-1})$
ENDDO,
```

where dist(β) is the distribution corresponding to the distribution basis β.

Note that the Compute-Dist-Factor procedure can easily be modified to obtain the initial distribution of the input array. This can be done by allowing ibasis$_1$ to be any distribution which permits a communication-free implementation of $\mathcal{F}_1$.

In the Compute-Dist-Factor procedure, we have assumed that communication-free distribution bases exist for each step of the computation. This may not be true for some formulas. In that case, a redistribution model program cannot be synthesized. For such formulas, it may be best to synthesize programs only under the point-to-point model. However, there are other possibilities such as factorizing the formula $\mathcal{F}$ as $\mathcal{F} \equiv \mathcal{F}_m \cdots \mathcal{F}_1$ in a manner in which redistribution model program can be synthesized for most of the subformulas $\mathcal{F}_k$. For the remaining subformuals a point-to-point model program can be synthesized. Note that the Compute-Dist-Factor procedure can be modified to obtain such a factorization.

7.3. Examples

We illustrate the Compute-Dist-Factor procedure for the Cooley–Tukey FFT and the Stockham FFT.

7.3.1. Cooley–Tukey FFT

For simplicity, we reconsider the implementation of the computation

$$A = \prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})\,(A)$$

on $P = 2^m$ processors. The above algorithm will compute the marked output basis of step i to be $e_p^{2^{n-i}} \otimes \mu_l^{2^i}$. Since $(n - i) \ge m$ only when $i \le (n - m)$, the formula will be split before step $(n - m + 1)$ and the arrays will be assigned a block distribution for the first $(n - m)$ steps. The marked output basis for steps $i > (n - m)$ will be computed to be: $e_p^{2^{n-i}} \otimes \mu_l^{2^{i-(n-m)}} \otimes e_k^{2^{n-m}}$. If $(n - m) \ge m$, i.e. $n \ge 2m$, a cyclic distribution can be chosen for the remaining steps. In fact, any cyclic(2^b) distribution with $b \le (n - 2m)$ can be chosen. The psuedocode for the Cooley–Tukey FFT with a block to cyclic redistribution is shown in Fig. 6.

```
PROCESSORS $PROC(2^m)$
DISTRIBUTE $A(block)$ DYNAMIC ONTO $PROC$
DO $i = 1, n - m$
  $A = (I_{2^{n-i}} \otimes T^{2^i}_{2^{i-1}})(A)$
  $A = (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(A)$
ENDDO
REDISTRIBUTE $A(cyclic)$ ONTO $PROC$
DO $i = n - m + 1, n$
  $A = (I_{2^{n-i}} \otimes T^{2^i}_{2^{i-1}})(A)$
  $A = (I_{2^{8-i}} \otimes F_2 \otimes I_{2^{i-1}})(A)$
ENDDO
```

FIG. 6. Program for the Cooley–Tukey FFT using redistribution.

7.3.2. Stockham FFT

We now illustrate how the Stockham FFT can be implemented using data redistributions. For analysis, we consider the following formula, which captures the essence of the computational structure of the Stockham FFT (Formula 2):

$$A = \prod_{i=1}^{n} ((F_2 \otimes I_{2^{n-1}})(L^{2^i}_2 \otimes I_{2^{n-i}}))(A).$$

We consider the implementation of a 2^n point FFT using 2^m processors, $m \leq n$. The marked output basis for step i, $1 \leq i < n$, is $\mu_l^{2^{n-i}} \otimes e_p^{2^i}$. Therefore, for step $i = (n - m + 1)$ the product of e bases will be less than 2^m and so the formula will be split before step $(n - m + 1)$. The initial distribution will be chosen to be cyclic. Assuming that $n \geq 2m$ and starting with a fresh input basis for step $(n - m + 1)$, the marked input and output basis for step i, $(n - m + 1) \leq i \leq n$, will be $\mu_l^{2^{i-(n-m)-1}} \otimes e_p^{2^{n-m}} \otimes e_k^{2^{n-i+1}}$ and $\mu_{l'}^{2^{i-(n-m)}} \otimes e_p^{2^{n-m}} \otimes e_{k'}^{2^{n-i}}$, respectively. If cyclic(2^m) distribution is chosen as the starting distribution for the remaining m steps, then the remaining steps can be performed in a communication-free manner with the input distribution for step i to be cyclic(2^{n-i+1}) and the output distribution to be cyclic(2^{n-i}). The pseudocode for the Stockham FFT is shown in Fig. 7.

```
PROCESSORS $PROC(2^m)$
DISTRIBUTE $(cyclic)$ DYNAMIC ONTO $PROC$: $A(0 : 2^n - 1)$
DO $i = 1, n - m$
  $A = (L^{2^i}_2 \otimes I_{2^{n-i}})(A)$
  $A = (F_2 \otimes I_{2^{n-1}})(T^{2^i}_{2^{i-1}} \otimes I_{2^{n-i}})(A)$
ENDDO
REDISTRIBUTE $A(cyclic(2^m))$ ONTO $PROC$
DO $i = n - m + 1, n$
  VIEWAS($A$, $(L^{2^i}_2 \otimes I_{2^{n-i}})$)
  $A = (F_2 \otimes I_{2^{n-1}})(T^{2^i}_{2^{i-1}} \otimes I_{2^{n-i}})(A)$
ENDDO
/* The final distribution of $A$ is $cyclic$ */
```

FIG. 7. Program for the Stockham FFT using redistributions.

8. PERFORMANCE RESULTS

In this section, we present performance measurements for the Cooley–Tukey and Stockham FFT on the Intel iPSC/860 system for both the target models proposed. The Intel iPSC/860 is a distributed-memory multiprocessor with a hypercube topology. Redistribution primitives, which require all-to-all communication, were implemented using the pairwise exchange algorithm [2]. Performance was measured using the millisecond node timer `mclock`.

Parallel programs under the point-to-point model were implemented for both the Cooley–Tukey FFT (CT-PP) and the Stockham FFT (ST-PP). On a hypercube of size P, both CT-PP and ST-PP use $\log(P)$ communication steps. However, these communication steps are nearest-neighbor only for CT-PP.

Programs for the Cooley–Tukey FFT under the redistribution model were implemented based on the program in Fig. 6. The program of Fig. 6 requires communication only during the redistribution step, where distribution of array A is changed from block to cyclic. The two computation loops require no communication since all data required are locally mapped. If array A is to be returned at the end of the computation to its original block distribution, then another redistribution will be required at the end of the program. Hence two versions of the Cooley–Tukey FFT were implemented, one using only a single redistribution step (CT-1R) and another one performing an additional block to cyclic redistribution to restore A to its original distribution (CT-2R).

A version of the Stockham FFT (ST-1R) was implemented based on the program in Fig.7. The program in Fig. 7 requires communication for performing the redistribution from cyclic to cyclic(2^m), where $m = \log(P)$. Since the final distribution of the array A is cyclic, no extra redistribution is required to restore the distribution to cyclic. An initial bit-reversal permutation step is required in all the Cooley–Tukey FFT programs. The implementation for the bit-reversal permutation uses one block to cyclic redistribution.

Before presenting the experimental performance results, we compare analytically communication cost for CT-PP and CT-1R. For simplicity, we ignore the communication overhead due to initial bit-reversal permutation for both CT-PP and CT-1R. The time spent in communication per processor in CT-PP and CT-1R can be estimated to be

$$T_{\text{CT-PP}} = \log(P)(t_s + (N/P)t_p) = \log(P)t_s + (\log(P)N/P)t_p$$

and

$$\begin{aligned} T_{\text{CT-1R}} &= (P-1)(t_s + (N/P^2)t_p) \\ &= (P-1)t_s + ((P-1)N/P^2)t_p. \end{aligned}$$

It can be noted that CT-1R uses more number of messages than CT-PP ($(P - 1)$ vs $\log(P)$). However, CT-1R has a lower overall communication volume (aggregate of mes-

TABLE II
Estimated Communication Time (ms) for CT-PP and CT-1R (without Bit-Reversal) for the Intel iPSC/860 System with $P = 32$, $t_s = 164$ μs and $t_p = 3.2$ μs (for Complex Data-Type)

log(N)	$T_{\text{CT-PP}}$	$T_{\text{CT-1R}}$	$T_{\text{CT-PP}}/T_{\text{CT-1R}}$
10	1.3	5.2	0.3
11	1.8	5.3	0.3
12	2.9	5.5	0.5
13	4.9	5.9	0.8
14	9.0	6.7	1.4
15	17.2	8.3	2.1
16	33.6	11.4	2.9
17	66.4	17.8	3.7
18	131.9	30.5	4.3
19	263.0	55.9	4.7
20	525.1	106.7	4.9
21	1049.4	208.2	5.0

sage sizes over all the processors) than CT-PP ($P(P-1)(N/P^2) \approx N$ vs $P\log(P)(N/P) = N\log(P)$). Table II shows the estimated communication time for CT-PP and CT-1R on the Intel iPSC/860 system with $P = 32$. It can be observed that for small problem sizes CT-PP will have lower communication overhead than CT-1R. However, as the problem size is increased CT-1R eventually becomes more communication-efficient than CT-PP. Furthermore, the ratio of $T_{\text{CT-PP}}$ to $T_{\text{CT-1R}}$ increases with the problem size.

Experimental performance results for data sizes ranging from 1K to 2M, on 32 processors, are shown in Fig. 8 using log–log scale. From Fig. 8 it can be observed that, for small sizes of N, CT-PP performs better than CT-1R. However, for large values of N, CT-1R performs better than CT-PP and CT-2R performs as well as CT-PP. For $N \geq 4$K, ST-1R demonstrates much better performance than all the other programs. The reason it performs better than the Cooley–Tukey FFT programs is that it does not require the initial bit-reversal permutation.

9. CONCLUSIONS

We have presented an algebraic framework based on the tensor product for describing the semantics of regular data distributions. We have used this framework for generating programs in languages that allow explicit specification of data distributions and permit dynamic redistribution of arrays through executable statements, as well as, node programs with explicit communication commands. We have demonstrated this methodology by developing communication efficient distributed-memory programs for the fast Fourier transform. The methodology presented in this paper has been incorporated in EXTENT. The system EXTENT can presently synthesize programs for the Intel iPSC/860, Intel Paragon, and Cray T3D.

The framework presented in this paper depends on knowing the factorization of the problem size N. This may be considered as a drawback of this scheme. However, it should be noted that efficient algorithms for block recursive algorithms depend upon the factorizations of N. Different factorizations of N lead to different algorithms with different performance characteristics. The system EXTENT has been developed to assist in searching for an efficient implementation among the many possible implementations even for a single problem size N.

In the paper we have implicitly assumed that the number of processors P can be factorized into, say, m factors n_j; 1

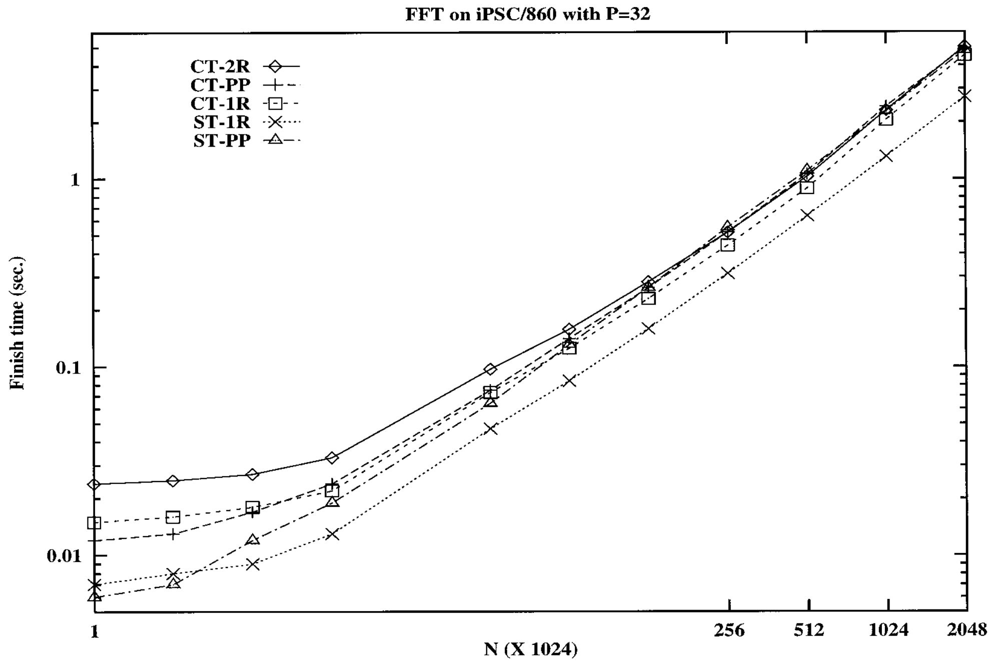

FIG. 8. FFT on the Intel iPSC/860 system with $P = 32$.

$\leq j \leq m$, such that m vector bases can be chosen for processor indexing. The dimension of jth basis being n_j. However, this may not necessarily be true. In the system EXTENT we use the following virtual processor approach to remove this restriction on the number of processors:

- Select a number $V \geq P$, such that distributions over V virtual processors can be represented as a distribution basis.
- Synthesize a node program for P processors by assuming some mapping of the V virtual processors onto P physical processors which achieves good load balancing.

For example, consider the implementation of $F \equiv A = (I_{3^9} \otimes F_3)(A)$ on four processors. The input and output array can be assumed to be distributed on nine virtual processors with the first three processors each assigned two virtual processors and the fourth processor assigned three virtual processors. Node code with the following structure can be then synthesized for F:

```
IF ( $myid < 3$) THEN
 FOR $v = 0, 1$
  /* Computation corresponding to $v$th vir
     tual processor on $myid$. */
      END FOR
ELSE
 FOR $v = 0, 2$
  /* Computation corresponding to $v$th vir
     tual processor on $myid$. */
      END FOR
ENDIF
```

We feel that some of the techniques presented in this paper may be useful for efficient compilation of HPF programs. In particular, it is worth investigating how the optimization based on VIEWAS can be incorporated in an HPF compiler.

ACKNOWLEDGMENTS

We are grateful to the NASA Lewis Research Center for providing access to the Intel iPSC/860 system. We would also like to thank the anonymous referees for helping improve the technical quality and presentation of the paper. In particular, we greatly appreciate the very detailed and constructive comments from Referee A.

REFERENCES

1. Bodin, F., Beckman, P., Gannon, D., Yang, S., Kesavan, S., Malony, A., and Mohr, B. Implementing a parallel C++ runtime system for scalable parallel systems. *Supercomputing '93.* Pp. 588–597.
2. Bokhari, S. H. Complete exchange on the iPSC. Tech. Rep. 91-4, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, 1991.
3. Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., and Ranka. S. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design implementation and performance results. *Supercomputing '93.* Pp. 351–360.
4. Brewer, J. W. Kronecker products and matrix calculus in system theory. *IEEE Trans. Circuits Systems* **25** (1978), 772–781.
5. Chapman, B. M., Mehrotra, P., and Zima, H. P. Vienna Fortran—A Fortran language extension for distributed memory multiprocessors. In Saltz, J., and Mehrotra, P. (Eds.). *Language, Compilers and Runtime Environments for Distributed Memory Machines.* North-Holland, Amsterdam, 1992. Pp. 39–62.
6. Dai, D. L., Gupta, S. K. S., Kaushik, S. D., Lu, J. H., Singh, R. V., Huang, C.-H., Sadayappan, P., and Johnson, R. W. EXTENT: A portable programming environment for designing and implementing high performance block recursive algorithms. *Supercomputing '94.* Pp. 49–58.
7. Davio, M. Kronecker products and shuffle albegra. *IEEE Trans. Comput.* **30,** 3(Feb. 1981), 116–125.
8. High Performance Fortran Forum. High Performance Fortran language specification, Version 1.0. Tech. Rep. CRPC-TR92225, Rice University, 1993.
9. Graham, A. *Kronecker Products and Matrix Calculus: With Applications.* Ellis Horwood, Chichester, 1981.
10. Granta, J., Conner, M., and Tolimieri R. Recursive fast algorithms and the role of the tensor product. *IEEE Trans. Signal Process.* **40,** 12(Dec. 1992), 2921–2930.
11. Graybill, F. A. *Matrices, with Applications in Statistics.* Wadsworth, Belmong, CA, 1983.
12. Gupta, S. K. S., Kaushik, S. D., Sharma, S., Huang, C.-H., Johnson, J. R., Johnson, R. W., and Sadayappan, P. A methodology for the generation of data distributions to optimize communication. *Proc. 1992 Fourth IEEE Symposium on Parallel and Distributed Processing.* Pp. 436–441.
13. Hiranandani, S., Kennedy, K., and Tseng, C.-W. Compiling Fortran-D for MIMD distributed-memory machines. *Comm. ACM* **35,** 8(Aug. 1992), 66–80.
14. Huang, C.-H., Johnson, J. R., and Johnson, R. W. A tensor product formulation of Strassen's matrix multiplication algorithm. *App. Math. Lett.* **3,** 3(1990), 67–71.
15. Huang, C.-H., Johnson, J. R., and Johnson, R. W. Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm. *Proc. 1992 International Conf. Parallel Processing.* Pp. 104–108.
16. Johnson, J. R., Huang, C.-H., and Johnson, R. W. Tensor permuations and block matrix allocation. In Hains, G., and Mullin, L. (Eds.). *Second International Workship on Array Structures* (*ATABLE-92*). Pub. 841, Department of Information and Operations Research, University of Montreal, 1992.
17. Johnson, J. R., Johnson, R. W., Rodriguez, D., and Tolimieri, R. A methodology for designing modifying and implementing Fourier transform algorithms on various architectures. *Circuits Systems Signal Process.* **9,** 4(1990), 450–500.
18. Johnson, R. W., Huang, C.-H., and Johnson, J. R. Multilinear algebra and parallel programming. *J. Supercomput.* **5** (1991), 189–218.
19. Kaushik, S. D., Sharma, S., and Huang, C.-H. An algebraic theory for modeling multistage interconnection networks. *J. Inform. Sci. Engrg.* **9** (1993), 1–26.
20. Kaushik, S. D., Sharma, S., Huang, C.-H., Johnson, J. R., Johnson, R. W., and Sadayappan, P. An algebraic theory for modelling direct interconnection networks. *Supercomputing '92.* Pp. 488–497.
21. Regalia, P. A., and Mitra, S. K. Kronecker products, unitary matrices and signal processing applications. *SIAM Rev.* **31,** 4(Dec. 1989), 586–613.
22. Ritter, G. X., and Gader, P. D. Image algebra techniques and parallel image processing. *J. Parallel Distrib. Comput.* **4** (1987), 7–44.
23. Stockham, T. G. High speed convolution and correlation. *Proc. 1966 Spring Joint Computer Conf. Proc. of AFIPS.* Pp. 229–233.

24. Van Loan, C. *Computational Frameworks for the Fast Fourier Transform.* SIAM, Philadelphia, 1992.

SANDEEP KUMAR S. GUPTA received the B.Tech. degree in computer science and engineering from Institute of Technology, Banaras Hindu University, Varanasi, India, in 1987, the M.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kanpur, in 1989, and the M.S. and Ph.D. degrees in computer and information science from the Ohio State University, Columbus, Ohio, in 1991 and 1995, respectively. He is currently a visiting assistant professor in the School of Electrical Engineering and Computer Science at Ohio University, Athens, Ohio, and an adjunct research assistant professor in the Department of Computer Science at Duke University, Durham, North Carolina, where he was also a research associate in the spring and summer of 1995. His research interests include compilers for parallel machines and parallel computing.

CHUA-HUANG HUANG received the B.S. degree in mathematics from Fu-Jen University, Taiwan, in 1974, thes M.S. degree in computer science from the University of Oregon, Eugene, Oregon in 1979, and the Ph.D. degree in computer science from the University of Texas at Austin, Austin, Texas in 1987. He was an assistant researcher at the Telecommunication Laboratories in Taiwan from 1979 to 1982. Currently, he is an associate professor in the Department of Computer and Information Science at the Ohio State University. His research interests include compiler techniques for HPF, parallel I/O, and parallel algorithms.

P. SADAYAPPAN received the B.Tech. from the Indian Institute of Technology, Madras, and the M.S. and Ph.D. from the State University of New York at Stony Brook, all in electrical engineering. He is currently an associate professor in the Department of Computer and Information Science at the Ohio State University. His research interests include compiling for parallel architectures and high-performance scientific computing.

ROBERT W. JOHNSON is a professor in the Department of Computer Science at St. Cloud State University, Minnesota. He received the A.B. degree from Columbia College in 1962. He received the M.S. degree in mathematics from the City College of New York in 1965 and the Ph.D. in mathematics from the City University of New York in 1969. His research interests include algorithm design, crystallography, digital signal processing, and supercomputing. He is a member of the ACA, ACM, AMS, IEEE, and SIAM.

Received January 18, 1994; revised September 20, 1994; accepted July 11, 1995