10-1-1994

Dist a distribution independent parallel programs for matrix multiplication

Hyuk J. Lee
Purdue University School of Electrical Engineering

José A.B. Fortes
Purdue University School of Electrical Engineering

DATA DISTRIBUTION INDEPENDENT PARALLEL PROGRAMS FOR MATRIX MULTIPLICATION

HYUK J. LEE
JOSÉ **A.B.** FORTES

TR-EE 94-32
OCTOBER 1994

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

Data distribution independent parallel programs for matrix multiplication*

Hyuk J. Lee and José A.B. Fortes

School of Electrical Engineering

Purdue University

W. Lafayette, IN 47907

'This research was partially funded by the National Science Foundation under grant number CDA-9015696.

Abstract

This report considers the problem of writing data distribution independent (DDI) programs in order to eliminate or reduce initial data redistribution overheads for distributed memory parallel computers. The functionality and execution time of DDI programs are independent of initial data distributions. First, modular mappings, which can be used to derive many equally optimal and functionally equivalent programs, are briefly reviewed. Relations between modular mappings and input data distributions are then established. These relations are the basis of a systematic approach to the derivation of DDI programs which is illustrated for matrix-matrix multiplication($c = a \times b$). Conditions on data distributions that correspond to an optimal modular mapping are: *(1) the first row of the inverse of distribution pattern matrix of army 'a' should be equal to the second row of the inverse of distribution pattern matrix of array 'b', (2) the second row of the inverse of distribution pattern matrix of array 'a' should be linearly independent of the first row of the inverse of distribution pattern matrix of array 'b', and* ***(3)*** *each distribution pattern matrix of arrays 'a', 'b', and* 'c' *should have at least one zero entry, respectively.* It is shown that only twelve programs suffice to accomplish redistribution-free execution for the many input data distributions that satisfy the above conditions. When DDI matrix multiplication programs are used in an algorithm with multiple matrix products, half of data redistributions otherwise required can be eliminated.

Contents

1 Introduction

Optimized program modules such as basic subroutines, functions, macros, and intrinsic operations are widely used to write efficient programs for parallel machines. In general, these modules expect input data to be distributed across processor memories in a predetermined manner. Otherwise, it is necessary to carry a redistribution step which can be very expensive in time. A systematic approach is proposed to design data distribution independent (DDI) modules which eliminate or reduce the need for redistribution. The approach is conveyed and illustrated by the design of DDI programs for matrix multiplication. Using the resulting DDI matrix multiplication modules, it is possible to halve data redistribution costs in applications that require the product of 3^n matrices for any integer n.

A very large number of machines currently available or recently announced by major manufacturers (see [1]) have a physically distributed memory (possibly but not necessarily shared from a logic perspective, i.e. appearing as a single virtual address space to all processors). In these machines, bad initial *data distributions* (i.e., mappings of data into local processor memories) can slow down computations for reasons that are not inherent to the algorithms. The topic of data distribution in distributed memory machines has been extensively studied [2]-[11]. For some classes of programs techniques have been developed to enable a compiler to optimize (i.e., "minimize") the extent of data distribution/communication required to execute a program. However, this is an NP-complete problem and heuristic approaches must be used in practical solutions [2, 3].

In order to eliminate or reduce initial data redistributions, a new approach called *data distribution independent (DDI)* parallel programming has been recently proposed [12]. Ideally, a DDI parallel programming paradigm would be based on libraries of DDI computational modules. *An ideal DDI computational module is a parallel program whose execution time and functionality are independent of input data distribution.* In addition, ideal DDI module executes as fast as the fastest data distribution dependent (DDD) module for the same function and a given fixed input data distribution. In practice, this requirement might not be perfectly met by some DDI modules because of inevitable

overheads. However, even DDI modules that do not execute in minimal time and may themselves implement some redistribution of data may yield better programs than their DDD counterparts. A parallel program could then be written by involsing these modules in some appropriate order without concern for how data is distributed. Such programs would inherit the DDI property and could be invoked by other programs without violating the basic paradigm. Programs would be easily ported among machines with different topologies as long as they have functionally equivalent DDI module libraries.

This report *explores the possibility* of a parallel programming paradigm that is data-distribution independent (DDI) in the sense that the user would not be required to program or even invoke data communication modules. The need for data redistribution would either be eliminated or transparent to the user. The emphasis of the work reported here is on the design of computational modules so that there is no need to redistribute input data. When this cannot be achieved, the cost of (automatic) reclistribution should be minimized but this aspect of the problem is not addressed in this report. In this context, source-to-source program transformations, called *modular mappings* [13], and properties that allow *commutative parallel processing* are to be explored as techniques and concepts that enable DDI computation. Commutative parallel processing exploits severitl types of commutativity. *Functional commutativity* corresponds to the usual mathematical definition of commutativity which allows operands of an operation to exchange position without impacting the final result. *Structural commutativity* is present among computations that are independent, i.e., share no data - they can be scheduled and allocated independently. Finally, *architectural commutativity* captures :symmetries in the target architecture that allow many virtual mappings of the processors and interconnection network into the same identical virtual architecture (thus "making" different data distributions look the same for different mappings).

As an initial work towards DDI computation, a systematic derivation of DDI parallel programs is provided for matrix-matrix multiplication. It should be clear from the presentation that the methodology has a general nature and can be applied to a large class of algorithms. In order to quantify and illustrate the improvement by DDI parallel programs in common applications, a program for triple matrix product is optimized using DDI parallel programs for matrix multiplication.

The rest of the report is organized as follows. Section 2 discusses relations between modular time-space transformations and modular distributions. Conditions also are derived that guarantee alignment of data and computations. Section 3 examines the conditions on optimal modular mappings and distributions for matrix multiplication that eliminate the need for input data redistribution. Based on these conditions, DDI parallel programs for matrix multiplication are derived. Section 4 analyzes the improvement by DDI matrix multiplication programs when they are used in programs that contain series matrix multiplications. Section 5 concludes the report.

2 Modular time-space transformations and modular data distributions

Linear transformations of programs whose execution time is mostly spent in loops have been extensively studied and used for source-to-source program transformations in parallelizing compilers and systolic array design [15]-[21]. In this framework, a given computation in a loop is represented by a vector $\bar{j}$ (of loop indices) and a linear mapping of coinputation into a domain of time (t) and processor $(\bar{x})$ specifies the schedule and the allocation of the computation. The execution time of this computation is specified by $t = \Pi\bar{j}$ while the processor executing this computation is determined by $a = S\bar{j}$ where $\Pi \in Z^{1\times dim(\bar{j})}$ and $S \in Z^{dim(processor\ array)\times dim(\bar{j})}$ are called the schedule and the allocation of a linear mapping, respectively. Combining the schedule aad the allocation, a transformation matrix $\begin{pmatrix} \Pi \\ S \end{pmatrix}$ is often used to represent a linear mapping. Extending the framework of linear algorithm transformations, this section considers modular translormations which are described by linear transformations modulo a constant vector. With modular mappings, time and processor are determined as with linear mappings except that they are computed modulo T and modulo P if execution time and numbers of processors are to be limited to T and P, respectively. Clearly, the programs that result from such transformations have the same execution time T on the same number of processors P. Hence, modular mappings can yield algorithm transformations that are equally optimal. Basic definitions and results on modular mappings are reviewed next

(an extended treatment can be found in [13]).

2.1 Modular time-space transformations

Modular time-space transformations are defined in terms of two operations, a linear transformation and a 'mod' operation.

Definition 1 (modular function) A *modular function,* $T_{\bar{m}} : Z^n \rightarrow Z^k$, *is a mapping of the form:*

$$T_{\bar{m}}(\bar{j}) = \begin{bmatrix} T(1) \cdot \bar{j}_{(mod\, m_1)} \\ T(2) \cdot \bar{j}_{(mod\, m_2)} \\ \vdots \\ T(k) \cdot \bar{j}_{(mod\, m_k)} \end{bmatrix} \tag{1}$$

where $T(i)$ *is a row vector. The matrix* $T = \begin{bmatrix} T(1) \\ \vdots \\ T(k) \end{bmatrix}$ *and vector* $\bar{m} = (m_1, \cdots, m_k)^T$ *are called the transformation matrix and modulus vector, respectively.*

Definition 2 (modular transformation) A *modular time-space transformation,* $T_{\bar{m}}$, *is a modular function that is injective when its domain is restricted to the index set* J *of an algorithm, i.e.,* $T_{\bar{m}} : J \rightarrow Z^k$ *is injective.*

Any k x n transformation matrix T and k dimensional modulus vector m can make a modular function. However, in order for any modular function to be a modular transformation of a given algorithm, T and m must be carefully chosen so that the transformation is injective when its domain is restricted to the index set of the algorithm. This section considers only the case when n = k. Let $\bar{u}$ and $\bar{v}$ be two vectors with the same number of elements. The notation $\bar{u}_{(mod\, \bar{v})}$ denotes a vector $((u_1)_{(mod\, v_1)}, (u_2)_{(mod\, v_2)}, \cdots, (u_n)_{(mod\, v_n)})$. Therefore the modular function can be described as $T_{\bar{m}}(\bar{j}) = (T\bar{j})_{(mod\, \bar{m})}$.

Initial work on linear transformations concentrated on perfectly nested loops whose body is treated as a single computation even if it contains multiple statements. Thanks to extensive work by many researchers, individual statements in arbitrarily nested loops can now be handled by using affine-by-statement mappings [19]. Modular affine-by-statement mappings can also be defined just as modular linear mappings were defined with respect to linear mappings. For simplicity, the results discussed in this report are stated in terms of modular linear mappings but are extensible to modular affine-by-statement mappings.

Linear mappings can be considered as particular cases of modular linear mappings for large enough moduli and finite domains. It follows that it is possible to use modular mappings to derive algorithms that cannot be derived by using linear mappings. Cannon's algorithm for matrix multiplication is a good example of this fact (See Example 1). Finally, regarding processor allocation, modular mappings are well suited for ring, torus and other topologies where "wrap-around" links are mathematically captured by the "mocl" operation.

Exarnple 1 *Consider the matrix-matrix multiplication algorithm which computes* $c = a \times b$ *where* a, b, *and* c *are* (5×5) *matrices.*

```
DO i = 0,4
   DO j = 0,4
      DO k = 0,4
         c(i,j) = c(i,j) + a(i,E) × b(k,j)
CONTINUE
```

Cannon's algorithm is particularly efficient and frequently used in actual parallel processors whose interconnection network is a torus[23],[24]. It is not possible to use affine mappings to derive Cannon's algorithm from the sequential matrix-matrix multiplication algorithm. Instead, the following modular transformation is required:

$$T_{\bar{b}}(i,j,k) = \left(\begin{pmatrix} -1 & -1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} (i,j,k) \right)_{(mod\ (5,5,5)))}. \tag{2}$$

This modular transformation yields the following program:

```
DO t = 0,4
    DOALL p1 = 0,4
        DOALL p2 = 0,4
            i = p1
            j = p2
            k = (t + p1 + p2)mod 5
            c(i,j) = c(i,j) + a(i,k) × b(k,j)
CONTINUE
```

The modular mapping used to "derive" Cannon's algorithm is only one of many possible such mappings. However, not all modular mappings are acceptable. In addition to being injective they must satisfy other conditions (soon to be discussed) that preserve correctness. It is not trivial to derive conditions that assure injectivity of modular mappings for arbitrary algorithms. However, sufficient conditions (which are necessary in some cases) for injectivity of modular mappings of rectangular algorithms (i.e. algorithms whose index sets are bounded by constants) have been provided in [13] and one of the main results is as follows:

Theorem 1 Let $J_{\bar{b}}$ be a rectangular index set with the boundary vector $\boldsymbol{b}$. Let $T_{\bar{b}}$ be a modular function of the index set $J_{\bar{b}}$. Let $\succ$ be an arbitrarily order on the set $\{1, 2, \cdots, n)$. $T_{\bar{b}}$ is injective if its transformation matrix T satisfies the following equations:

$$1. \qquad t_{ii} = \pm 1, \tag{3}$$

$$2. \qquad t_{ij} = 0 \quad if \;\; i \succ j \quad . \tag{4}$$

For *the* case when all elements of b are identical, $T_{\bar{b}}$ is injective if T *is* unimodular.

In this theorem, the modulus vector of a modular function is the same as the boundary vector of an index set, i.e. $\bar{m}$ is equal to b. This condition can be generalized to the case when the modulus vector results from a permutation of the entries of the boundary vector. For the particular cases when there exist some identical entries in the boundary vector, it is possible to obtain more general conditions [13].

In addition to injectivity, valid modular mappings must also preserve dependencies and, possibly, avoid broadcasts. These and other conditions have been well studied in the context of affine mappings and can be captured similarly for modular mappings. However, when functional commutativity is present, these conditions should be changed to take advantage of the possibility of reordering chains of computations. For example, consider matrix-matrix multiplication. The condition for correctly sequencing computations and removing data broadcasts is that every element of the schedule vector should be positive. If functional commutativity is taken into account, the condition can be changed to a weaker condition such that every element of the schedule vector should be different from 0. In fact, Cannon's algorithm does take advantage of addition commutativity and wrap-around links and cannot be derived unless these properties are taken into consideration. The :followingexample illustrates how a modular mapping other than Cannon's can be selected for matrix multiplication using the conditions just discussed.

Exarnple *2 Consider the matrix-matrix multiplication algorithm of Example 1 again. The modular transformation*

$$T_b(i,j,k) = (\begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} (i,j,k))_{(mod\,(5,5,5)))}. \tag{5}$$

yields the following program which is as efficient as Cannon's algorithm in the sense of processor utilization and neighborhood communication:

```
DO t = 0,4
    DOALL p1 = 0,4
        DOALL p2 = 0,4
            i = (t + p1 + p2)mod 5
            j = p1
            k = p2
            c(i,j) = c(i,j) + a(i,k) × b(k,j)
CONTINUE
```

2.2 Modular data distributions

In order to take advantage of modular mappings it is necessary to relate them to input data distributions that eliminate or minimize misalignments of data during the execution of the program (instead of only at the beginning of it). We consider distributions of data arrays as mappings from array indices to processor coordinates of the general form

$$p(\bar{y}, t) = (P\bar{y} + \bar{p} + \bar{\gamma}t)_{mod\ \bar{b}_X},$$

where y is a data array index, t denotes execution time, $\boldsymbol{P}$ is the data distribution pattern matrix, p is the data distribution offset, $\bar{\gamma}$ is the data distribution mobility and $\bar{b}_X$ could be any vector (of the right dimension) but is hereon assumed to have its components correspond to the sizes of the processor grid along its dimensions. The initial data array distribution is specified by $p(\bar{y}, 0)$. For example, the data distribution $p_a(\bar{y}, t) = (\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} t)_{mod\ (5,5)}$ maps the array element $a(i, j)$ into processor (j, i) of a 5×5 processor array at time zero and moves it right by one position every time unit (i.e., to processor $((j + t)_{mod\ 5}, i)$ at time t). Program array references to any data array element with index y are assumed to be of the form

$$\bar{y} = F\bar{j} + \bar{f}$$

where F is the indexing matrix, $\bar{f}$ is the *index* offset and $\bar{j}$ is a point in the iteration set of the (nested loop) program. For example, for the reference $a(2i, j + 1)$ in the body of two nested loops on i and j, the indexing matrix and offset are $F = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$ and $\bar{f} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, *respectively*

Recall that, given a transformation matrix, the loop iteration point $\bar{j}$ is mapped onto processor $\bar{x}$ at time t.

$$\begin{pmatrix} t \\ \bar{x} \end{pmatrix} = (T\bar{j})_{mod\ \bar{m}} \tag{6}$$

where the modulus vector $\bar{m}$ should be a permutation of the boundary vector of the loop iteration domain, b. Let $\bar{x}'$ be the processor to which the index point $\bar{y}$ of data array is mapped at time t. Then,

$$\bar{x}' = \boldsymbol{P}\boldsymbol{y} + p + \quad \text{yt.} \tag{7}$$

Recall that the first row of the transformation matrix is the scheclule vector Π and, therefore t is $(\Pi\bar{j})_{mod b_\Pi}$ where b_Π represents the modulus of the schedule, i.e., the first entry of the modulus vector of a given time-space transformation. When the index set of the algorithm is cubic (i.e., b_Π is identical to all entries of $\bar{b}_X$). Then, Eq. 7 becomes:

$$\bar{x}' = (PF\bar{j} + P\bar{f} + \bar{p} + \bar{\gamma}\Pi\bar{j})_{mod\,\bar{b}_X}. \tag{8}$$

To compute point $\bar{j}$ without need for communication, $\bar{x}$ should be equal to $\bar{x}'$. Hence, the following condition is obtained:

$$(T(2,3)\bar{j})_{mod\,\bar{m}(2,3)} = (PF\bar{j} + P\bar{f} + \bar{p} + \bar{\gamma}\Pi\bar{j})_{mod\,\bar{b}_X}, \tag{9}$$

where T(2, 3) denotes the second and third rows of matrix T and $\bar{m}(2,3)$ denotes the second and third elements of vector m. To satisfy above equation for arbitrary $\bar{j}$, the condition becomes:

$$T(2,3) = PF + \bar{\gamma}\Pi, \tag{10}$$

$$\bar{0} = P\bar{f} + \bar{p}, \tag{11}$$

$$\bar{m}(2,3) = \bar{b}_X. \tag{12}$$

In the general case, when the index set is not cubic these conditions are also valid. However, it is necessary to consider a larger class of distribution functions that allow for replication of data and an additional condition is imposed on the number of array copies. The general form of the distribution when there are $C = \lfloor\frac{(\Pi\bar{j})_{max}}{b_\Pi}\rfloor - \lfloor\frac{(\Pi\bar{j})_{min}}{b_n}\rfloor + 1$ copies of the array along direction $\bar{\gamma}$ is

$$p_k^{b_\Pi}(\bar{y}, t) = (P\bar{y} + \bar{p} + \bar{\gamma}(t + kb_\Pi))_{mod\,\bar{b}_x} \tag{13}$$

whert: $k = \lfloor\frac{(\Pi\bar{j})_{min}}{b_\Pi}\rfloor, \lfloor\frac{(\Pi\bar{j})_{min}}{b_\Pi}\rfloor + 1, \cdots, \lfloor\frac{(\Pi\bar{j})_{max}}{b_\Pi}\rfloor - 1, \lfloor\frac{(\Pi\bar{j})_{max}}{b_\Pi}\rfloor$. Lemma A.1 in theappendix show:; that these distribution functions guarantee the alignment between data and computations of a program that results from a modular mapping satisfying Equations 10- 12. Assume that b_Π divides all the entries in $\bar{b}_X$. Then, $p_k^{b_\Pi}$ and $p_{k+k_{lcm}}^{b_\Pi}$ generate the same data distribution where k_{lcm} is the least common multiple of b_i/b_Π for all entries b_i of $\bar{b}_X$. Hence, in this case, at most k_{lcm} data array copies are necessary instead of the number C mentioned above. Depending on the value of $\bar{\gamma}$, the number of copies can be further

reduced. In the case of a cubic index set, $k_{lcm} = 1$ and Eq. 13 particularizes to Eq. 9 (any single value of k is acceptable including k = 0).

Eq. 10 shows the relation between the pattern distribution and the transformation matrix and Eq. 11 shows the condition of the offset distribution. In this section, the conditions of Eq. 11 and Eq. 12 are assumed to be always satisfied, and only the condition of Eq. 10 will be discussed.

Eq. 10 becomes

$$T(2,3) - \bar{\gamma}\Pi = PF, \tag{14}$$

$$\begin{pmatrix} -\bar{\gamma} & \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix} \end{pmatrix} \begin{pmatrix} \Pi \\ T(2) \\ T(3) \end{pmatrix} = PF, \tag{15}$$

and

$$P^{-1} \begin{pmatrix} -\bar{\gamma} & \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix} \end{pmatrix} = FT^{-1}. \tag{16}$$

Finally, the following conditions are obtained:

$$P^{-1} = FT^{-1}_{2..3}, \tag{17}$$

$$\bar{\gamma} = -PFT^{-1}_{1}, \tag{18}$$

where T^{-1}_{1} denotes the first column of T^{-1} and $T^{-1}_{2..3}$ denotes the second and third columns of T^{-1}. These equations must be established for each variable used in the same statement and solutions that satisfy all of them must be sought. From them conditions can be derived on input data distributions that guarantee alignment of data. It is also possible to derive the transformation necessary to generate a "transformed" program that accepts such data distributions.

Exarnple *3 Consider data distribution for Cannon's algorithm. The data distribution p^a of matrix 'a' and the data distribution p^b of matrix 'b' that satisfy conditions (17) and (18) are*

$$p^a(\bar{y}, t) = (\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} t)_{mod\ (5,5)}$$

$p_1 \backslash p_2$	0	1	2	3	4
0	$a_{0,0}/b_{0,0}$	$a_{0,1}/b_{1,1}$	$a_{0,2}/b_{2,2}$	$a_{0,3}/b_{3,3}$	$a_{0,4}/b_{4,4}$
1	$a_{1,1}/b_{1,0}$	$a_{1,2}/b_{2,1}$	$a_{1,3}/b_{3,2}$	$a_{1,4}/b_{4,3}$	$a_{1,0}/b_{0,4}$
2	$a_{2,2}/b_{2,0}$	$a_{2,3}/b_{3,1}$	$a_{2,4}/b_{4,2}$	$a_{2,0}/b_{0,3}$	$a_{2,1}/b_{1,4}$
3	$a_{3,3}/b_{3,0}$	$a_{3,4}/b_{4,1}$	$a_{3,0}/b_{0,2}$	$a_{3,1}/b_{1,3}$	$a_{3,2}/b_{2,4}$
4	$a_{4,4}/b_{4,0}$	$a_{4,0}/b_{0,1}$	$a_{4,1}/b_{1,2}$	$a_{4,2}/b_{2,3}$	$a_{4,3}/b_{3,4}$

(a) Initial data distributions

$p_1 \backslash p_2$	0	1	2	3	4
0	$a_{0,1}/b_{1,0}$	$a_{0,2}/b_{2,1}$	$a_{0,3}/b_{3,2}$	$a_{0,4}/b_{4,3}$	$a_{0,0}/b_{0,4}$
1	$a_{1,2}/b_{2,0}$	$a_{1,3}/b_{3,1}$	$a_{1,4}/b_{4,2}$	$a_{1,0}/b_{0,3}$	$a_{1,1}/b_{1,4}$
2	$a_{2,3}/b_{3,0}$	$a_{2,4}/b_{4,1}$	$a_{2,0}/b_{0,2}$	$a_{2,1}/b_{1,3}$	$a_{2,2}/b_{2,4}$
3	$a_{3,4}/b_{4,0}$	$a_{3,0}/b_{0,1}$	$a_{3,1}/b_{1,2}$	$a_{3,2}/b_{2,3}$	$a_{3,3}/b_{3,4}$
4	$a_{4,0}/b_{5,0}$	$a_{4,1}/b_{1,1}$	$a_{4,2}/b_{2,2}$	$a_{4,3}/b_{3,3}$	$a_{4,4}/b_{4,4}$

(b) Data distributions after one iteration

Figure 1: Data distributions of Cannon's algorithm..

$p_1 \backslash p_2$	0	1	2	3	4
0	$a_{0,0}/b_{0,0}$	$a_{1,1}/b_{1,0}$	$a_{2,2}/b_{2,0}$	$a_{3,3}/b_{3,0}$	$a_{4,4}/b_{4,0}$
1	$a_{1,0}/b_{0,1}$	$a_{2,1}/b_{1,1}$	$a_{3,2}/b_{2,1}$	$a_{4,3}/b_{3,1}$	$a_{0,4}/b_{4,1}$
2	$a_{2,0}/b_{0,2}$	$a_{3,1}/b_{1,2}$	$a_{4,2}/b_{2,2}$	$a_{0,3}/b_{3,2}$	$a_{1,4}/b_{4,2}$
3	$a_{3,0}/b_{0,3}$	$a_{4,1}/b_{1,3}$	$a_{0,2}/b_{2,3}$	$a_{1,3}/b_{3,3}$	$a_{2,4}/b_{4,3}$
4	$a_{4,0}/b_{0,4}$	$a_{0,1}/b_{1,4}$	$a_{1,2}/b_{2,4}$	$a_{2,3}/b_{3,4}$	$a_{2,4}/b_{4,4}$

Figure 2: Initial data distributions different from those of Cannon's algorithm.

and

$$p^b(\bar{y}, t) = (\begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} t)_{mod\,(5,5)}.$$

The initial data distributions (at $t = 0$) are shown in Fig. 1a and the data distributions after first iteration (at $t = 1$) are shown in Fig. 1b. On the other hand, the initial data distributions of the modular mapping in Example 2 are shown in Figure 2.

DDI parallel programs for matrix multiplication

3.1 Conditions of an optimal modular mapping for matrix multiplication

This section investigates conditions of optimal modular mapping and data distribution for matrix multiplication. Throughout this section, a mapping is optimal if it is as efficient as Citnnon's mapping with respect to computation and communication time. Proofs of Propositions given in this section appear in Appendix.

All modular mappings whose modulus vectors tightly bound the computation domain are optimal in the sense of processor utilization. However, in the sense of communication, not all of these mappings are optimal. Therefore, conditions for an optimal modular mapping need to be investigated not only in the view of processor utilization but also with regard to communication. Communication cost depends on the target architecture and, in this section, it is assumed that it has 4-way mesh with wrap-around interconnections. In addition, it is also assumed that data movement between distant processors is more expensive than that between neighbor processors. The efficiency of a given modular mapping is estimated based on these assumptions.

The previous section investigated the relationship between a modular mapping and an initial data distribution that allows the start of computation without initial data relocation. An algorithm of matrix multiplication contains two input data arrays, matrix *a* and matrix *b*. In order to start computation without initial data movements, these two data arrays should satisfy those conditions of **Eq.** 17 and **18.** To satisfy the condition of **Eq.** 17, the data distributions of matrix *a* and b should satisfy the following relation of Proposition 1.

Proposition 1 *Let P^a and P^b be distribution pattern matrices of data array a and b. If $P^{a^{-1}} = F^a T_{2..3}^{-1}$ and $P^{b^{-1}} = F^b T_{2..3}^{-1}$, then*

1. $P^{a^{-1}}(2) = P^{b^{-1}}(1)$,

2. $P^{a^{-1}}(1)$ and $P^{b^{-1}}(2)$ are linearly independent.

For the minimization of communication, the following should be satisfied.

Proposition 2 *For a matrix multiplication algorithm, the choice of T_1^{-1} that results in minimum communication is either $(\pm 1, 0, 0)^T$, $(0, \pm 1, 0)^T$, or $(0, 0, \pm 1)^T$.*

Cannon's algorithm has elements of matrix 'a' and 'b' move to the next processors and elements of matrix 'c' stay at the same processor throughout the computation. Hence, if an algorithm for matrix multiplication requires the same amount of data movements, the data distribution mobilities of 'a','b', and 'c', should be 0 (no communication), $(\pm 1, 0)^T$ (shift, vertically to the neighbor processor), or $(0, \pm 1)^T$ (shift horizontally to the neighbor processor).

Proposition 3 *Let $\bar{\gamma}$ be a distribution mobility of a data array. The optimal choice of $\bar{\gamma}$ is $\bar{0}$, $(\pm 1, 0)^T$, or $(0, \pm 1)^T$.*

Proposition 4 follows from Proposition **3** and Eq. 17.

Proposition 4 *P^a, P^b and P^c should have at least one zero entry, respectively.*

Proposition 1 and Proposition 4 give conditions on the pattern distribution matrices and consequently from Eq. 17 give conditions on $T_{2..3}^{-1}$. On the other hand, Proposition 2 and Proposition **3** give conditions on distribution mobilities and therefore from Eq. 18 give conditions on T_1^{-1}. Consider $T_{2..3}^{-1}$ and corresponding P^a and P^b based on the relation of Eq. 17 that satisfy the conditions in Proposition 1 and Proposition 4. It is not difficult to set: that such $T_{2..3}^{-1}$ should belong to one of the following six types:

$$\begin{pmatrix} \# & 0 \\ 0 & \# \\ \# & \# \end{pmatrix}, \begin{pmatrix} \# & 0 \\ \# & \# \\ 0 & \# \end{pmatrix}, \begin{pmatrix} 0 & \# \\ \# & 0 \\ \# & \# \end{pmatrix}, \begin{pmatrix} 0 & \# \\ \# & \# \\ \# & 0 \end{pmatrix}, \begin{pmatrix} \# & \# \\ \# & 0 \\ 0 & \# \end{pmatrix}, \begin{pmatrix} \# & \# \\ 0 & \# \\ \# & 0 \end{pmatrix},$$

when: $\#s$ denote arbitrary nonzero integers which are not necessarily identical.

To find the entire transformation matrix T, the next step is to find T_1^{-1}. Consider the first type of $T_{2..3}^{-1}$. It follows from Eq. 17 that the data distributions should be of the form

$$P^{a^{-1}} = \begin{pmatrix} \# & 0 \\ \# & \# \end{pmatrix}, P^{b^{-1}} = \begin{pmatrix} \# & \# \\ 0 & \# \end{pmatrix}, P^{c^{-1}} = \begin{pmatrix} \# & 0 \\ 0 & \# \end{pmatrix} \tag{19}$$

The inverse of data patterns become:

$$P^a = \begin{pmatrix} \# & 0 \\ \# & \# \end{pmatrix}, P^b = \begin{pmatrix} \# & \# \\ 0 & \# \end{pmatrix}, P^c = \begin{pmatrix} \# & 0 \\ 0 & \# \end{pmatrix}. \tag{20}$$

Proposition 2 allows three possible choices of T_1^{-1} : $(\pm 1,0,0)^T$, $(0,\pm 1,0)^T$, and $(0,0,\pm 1)^T$. If $T_1^{-1} = (\pm 1,0,0)^T$, then $\bar{\gamma}^a = (\#,\#)^T, \bar{\gamma}^b = (0,0)^T, \bar{\gamma}^c = (\#,0)^T$. If $T_1^{-1} = (0,\pm 1,0)^T$, then $\bar{\gamma}^a = (0,0)^T, \bar{\gamma}^b = (\#,\#)^T, \bar{\gamma}^c = (0,\#)^T$. If $T_1^{-1} = (0,0,\pm 1)^T$, then $\bar{\gamma}^a = (0,\#)^T, \bar{\gamma}^b = (\#,0)^T, \bar{\gamma}^c = (0,0)^T$. The first and the second choices do not satisfy Proposition **3.** Hence, the optimal choice is $T_1^{-1} = (0,0,\pm 1)^T$. Similarly, the optimal choice of T_1^{-1} can be found for other pattern distributions resulting in the following six types of the transformation matrices which guarantee that the corresponding programs run as efficiently as Cannon's algorithm.

$$\begin{pmatrix} 0 & \# & 0 \\ 0 & 0 & \# \\ \pm 1 & \# & \# \end{pmatrix}, \begin{pmatrix} 0 & \# & 0 \\ \pm 1 & \# & \# \\ 0 & 0 & \# \end{pmatrix}, \begin{pmatrix} 0 & 0 & \# \\ 0 & \# & 0 \\ \pm 1 & \# & \# \end{pmatrix}, \tag{21}$$
$$\begin{pmatrix} 0 & 0 & \# \\ \pm 1 & \# & \# \\ 0 & \# & 0 \end{pmatrix}, \begin{pmatrix} \pm 1 & \# & \# \\ 0 & \# & 0 \\ 0 & 0 & \# \end{pmatrix}, \begin{pmatrix} \pm 1 & \# & \# \\ 0 & 0 & \# \\ 0 & \# & 0 \end{pmatrix}.$$

3.2 DDI program module for matrix multiplication

Consider an SIMD or SPMD program for matrix multiplication. Without loss of generality, consider the case when modular mappings are of the form $\begin{pmatrix} 0 & \# & 0 \\ 0 & 0 & \# \\ \pm 1 & \# & \# \end{pmatrix}$.

Since $T^{-1} = (0,0,\pm 1)^T$, there are four possible choices of optimal distribution mobilities and corresponding data movements:

1. $\bar{\gamma}^a = (0,1)^T, \bar{\gamma}^b = (1,0)^T \rightarrow a$: east, b : south.

```
DO t = 0,4
      c = c + a * b
      MOVE-WEST(^)
      MOVE_SOUTH(b)
CONTINUE
```

(a) $\bar{\gamma}^a = (0,-1)^T, \bar{\gamma}^b = (1,0)^T$

```
DO t = 0,4
      c = c + a * b
      MOVE-WEST(;%)
      MOVE-NORTH(b)
CONTINUE
```

(b) $\bar{\gamma}^a = (0,-1)^T, \bar{\gamma}^b = (-1,0)^T$

Figure **3:** Programs for modular mappings with T of the form $\begin{pmatrix} 0 & \# & 0 \\ 0 & 0 & \# \\ & \# & \# \end{pmatrix}$.

2. $\bar{\gamma}^a = (0,1)^T, \bar{\gamma}^b = (-1,0)^T \rightarrow$ a : east, b : north.

3. $\bar{\gamma}^a = (0,-1)^T, \bar{\gamma}^b = (1,0)^T \rightarrow$ a : west, b : south.

4. $\bar{\gamma}^a = (0,-1)^T, \bar{\gamma}^b = (-1,0)^T \rightarrow$ a : west, b : north

Among these four data movements, the first and the fourth movements generate the same results. Similarly, the second and the third movements also generate the same results. Thus, the first and the second cases can be discarded and only the third and the fourth cases need to be kept. The programs for the third data movement and fourth data movement are shown in Fig. **3.** In these figures, variable a, b, c are assumed to be the appropriate elements of $a(i,j)$, $b(i,j)$, and $c(i,j)$, respectively.

For each of the other five forms of modular mappings, only two programs are also sufficient. Hence, in total, twelve programs cover all possible optimal modular data distributions. With these twelve optimal programs, a DDI parallel program module (DDIPPM) for matrix multiplication can be built. For a given data distribution, an optimal modular mapping can be found from the relation of Eq. 17 and Eq. 18. Then, the program corresponding to this modular mapping can be selected among the twelve programs in the DDIPPM.

Exarnple 4 *Consider* the initial distributions for *Cannon's* algorithnz shown in *Exam-*

```
DO t = 0,4
        c = c + a * b
        MOVE-WEST(a)
        MOVE-NORTH(b)
CONTINUE
```

(a) $P^a = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$, $P^b = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$

```
DO t = 0,4
        c = c + a * b
        MOVE-NORTH(a)
        MOVE-WEST(c)
CONTINUE
```

(b) $P^a = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$, $P^b = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Figure 4: Optimal programs for a given initial data distribution.

ple 1. Data pattern distributions of a and b are

$$P^a = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \text{ and } P^b = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \tag{22}$$

Hence, the inverses of data pattern distributions are

$$P^{a^{-1}} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \text{ and } P^{b^{-1}} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{23}$$

It follows from Eq. 17 that

$$T_{2..3}^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

From Eq. 21, T_1^{-1} should be $(0,0,\pm 1)^T$. If T_1^{-1} is chosen to be $(0,0,1)^T$, then $\gamma^a = (0,-1)^T$ and $\gamma^b = (-1,0)^T$ are obtained. Hence, the program of Fig. 4a is derived. This program is exactly the same as that in Example 1. Similarly, for the data distribution in Fig. 2, the optimal program of Fig. 4b can be derived.

4 Triple matrix product algorithm optimized by DDI matrix multiplication

This section considers a program for a sequence of matrix multiplications and implements this program by repeatedly invoking DDI parallel programs for matrix multiplication.

The number of data movements in this program is compared to the program that uses DDD programs for matrix multiplication.

Consider the triple matrix product:

$$\boldsymbol{Y} = L X R$$

where Y, L, X and $\boldsymbol{R}$ are matrices whose sizes are assumed to be suitable for the computation. Digital signal processing and control theory applications that require triple matrix products include discrete Fourier transform, discrete Lyapunov and Ricatti equations, and Kalman filtering [26].

The program for triple matrix product can be described by the following pseudo code:

```
TRI_MAT_PROD(Y,L,X,R)
MATRIX Y,L,X,R;

    MATRIX Z;
    MATMUL(Z,X,R);
    MATMUL(Y,L,Z);
}
```

In this program, keyword MATRIX represents a two dimensional array which is distributed in a canonical manner, i.e., whose distribution can be described by $p(\bar{y}) = ((\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \bar{y})_{mod\, b_{\bar{X}}}$. This distribution is called a *canonical distribution.* For simplicity, the sizes of matrices are not shown in this program, but it is implicitly assumed that they are appropriate for this computation. MATMUL is the subprogram for matrix multiplication which performs Cannon's algorithm with the assumption of canonically distributed input/output matrices. Given that initial data distribution for Cannon's algorithm is different from canonical array distribution (see Fig. 1), it is necessary to redistribute the input matrices. Hence, two array redistributions are necessary to start Cannon's algorithm. The distribution of the output matrix of Cannon's algorithm is the same as the canonical distribution. Hence, no redistribution is necessary for the output matrix.

Hence, subprogram MATMUL requires two data redistributions, and therefore, the entire program requires four data redistributions.

Now, consider the DDI approach to optimize triple matrix product. Let DDI_MATMUL(C, A, B, P_c, P_a, P_b) represent the DDI version of matrix multiplication routine which computes $c =$ a x b, where P_a and P_b represent the initial distributions of matrix a and b, respectively, and P_c represents the output distribution of matrix c. The initial distributions of matrix a and b must satisfy the conditions discussed in the previous section, and one of the twelve optimal programs corresponding the initial distribution is selected to compute matrix product. When DDI_MATMUL is used for triple matrix product, it is possible to select the program whose input data distribution matches initial data distribution of TRI_MAT_PROD (canonical distribution). Hence, data movements for redistribution can be reduced. Consider the following program:

```
TRI_MAT_PROD(Y,L,X,R)
MATRIX Y,L,X,R;
{
        MATRIX Z;
```

$$/* \; P_z = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}, \; P_x = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}, \; P_r = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \; */$$

```
        DDI_MATMUL(Z, X, R, P,, Px, P,)
```

$$/* \; P_y = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \; P_l = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}, \; */$$

```
        DDI_MATMUL(Y, L, Z, Py, Pl, P,)
}
```

In this program, $P_z, P_x, P,, P_y$, and P_l represent distribution pattern matrices of Z, X, R, Y, and L, respectively. These distributions are shown in Fig. 5. Since all matrices are assumed to be initially distributed in a canonical manner, matrix X and matrix

$p_1 \backslash p_2$	0	1	2	3	4
0	$z_{0,0}/x_{0,0}/r_{0,0}$	$z_{1,1}/x_{1,0}/r_{0,1}$	$z_{2,2}/x_{2,0}/r_{0,2}$	$z_{3,3}/x_{3,0}/r_{0,3}$	$z_{4,4}/x_{4,0}/r_{0,4}$
1	$z_{1,0}/x_{1,1}/r_{1,0}$	$z_{2,1}/x_{2,1}/r_{1,1}$	$z_{3,2}/x_{3,1}/r_{1,2}$	$z_{4,3}/x_{4,1}/r_{1,3}$	$z_{0,4}/x_{0,1}/r_{1,4}$
2	$z_{2,0}/x_{2,2}/r_{2,0}$	$z_{3,1}/x_{3,2}/r_{2,1}$	$z_{4,2}/x_{4,2}/r_{2,2}$	$z_{0,3}/x_{0,2}/r_{2,3}$	$z_{1,4}/x_{1,2}/r_{2,4}$
3	$z_{3,0}/x_{3,3}/r_{3,0}$	$z_{4,1}/x_{4,3}/r_{3,1}$	$z_{0,2}/x_{0,3}/r_{3,2}$	$z_{1,3}/x_{1,3}/r_{3,3}$	$z_{2,4}/x_{2,3}/r_{3,4}$
4	$z_{4,0}/x_{4,4}/r_{4,0}$	$z_{0,1}/x_{0,4}/r_{4,1}$	$z_{1,2}/x_{1,4}/r_{4,2}$	$z_{2,3}/x_{2,4}/r_{4,3}$	$z_{3,4}/x_{3,4}/r_{4,4}$

(a) Z, X and R.

$p_1 \backslash p_2$	0	1	2	3	4
0	$y_{0,0}/l_{0,0}/z_{0,0}$	$y_{0,1}/l_{0,1}/z_{1,1}$	$y_{0,2}/l_{0,2}/z_{2,2}$	$y_{0,3}/l_{0,3}/z_{3,3}$	$y_{0,4}/l_{0,4}/z_{4,4}$
1	$y_{1,0}/l_{1,1}/z_{1,0}$	$y_{1,1}/l_{1,2}/z_{2,1}$	$y_{1,2}/l_{1,3}/z_{3,2}$	$y_{1,3}/l_{1,4}/z_{4,3}$	$y_{1,4}/l_{1,0}/z_{0,4}$
2	$y_{2,0}/l_{2,2}/z_{2,0}$	$y_{2,1}/l_{2,3}/z_{3,1}$	$y_{2,2}/l_{2,4}/z_{4,2}$	$y_{2,3}/l_{2,0}/z_{0,3}$	$y_{2,4}/l_{2,1}/z_{1,4}$
3	$y_{3,0}/l_{3,3}/z_{3,0}$	$y_{3,1}/l_{3,4}/z_{4,1}$	$y_{3,2}/l_{3,0}/z_{0,2}$	$y_{3,3}/l_{3,1}/z_{1,3}$	$y_{3,4}/l_{3,2}/z_{2,4}$
4	$y_{4,0}/l_{4,4}/z_{4,0}$	$y_{4,1}/l_{4,0}/z_{0,1}$	$y_{4,2}/l_{4,1}/z_{1,2}$	$y_{4,3}/l_{4,2}/z_{2,3}$	$y_{4,4}/l_{4,3}/z_{3,4}$

(b) Y, L and Z.

Figure 5: Data distributions for a triple matrix product program

L need to be redistributed. However, matrix R and Y do not have to be redistributed. Hence, only two data redistributions are necessary in this program. Therefore, two data redistributions can be saved when comparing with the program that uses MATMUL subprogram.

Consider the general case when 3^n matrices are multiplied sequentially:

$$Y = A_1 A_2 \cdots A_{3^n}.$$

It is possible to use the subprogram for triple matrix product repeatedly to perform this computation:

$$\begin{array}{lll} T_1^1 = A_1 A_2 A_3, & T_2^1 = A_4 A_5 A_6, & \cdots,\ T_{3^{n-1}}^1 = A_{3^n-2} A_{3^n-1} A_{3^n} \\ T_1^2 = T_1^1 T_2^1 T_3^1, & T_2^2 = T_4^1 T_5^1 T_6^1, & \cdots,\ T_{3^{n-2}}^2 = T_{3^{n-1}-2}^1 T_{3^{n-1}-1}^1 T_{3^{n-} \\ \vdots & \vdots & \vdots \end{array}$$

$$T_1^{n-1} = T_1^{n-2} T_2^{n-2} T_3^{n-2}, \quad T_2^{n-1} = T_4^{n-2} T_5^{n-2} T_6^{n-2}, \quad T_3^{n-1} = T_7^{n-2} T_8^{n-2} T_9^{n-2}$$

$$Y = T_1^{n-1} T_2^{n-1} T_3^{n-1}$$

Therefore, a total of $(3^{n-1} + 3^{n-2} + \ldots 3 + 1)$ triple matrix products are needed. Hence, $2(3^{n-1} + 3^{n-2} + \cdots 3 + 1)$ data redistributions are necessary in this computation. Suppose that $(3^{n-1} - 1)$ MATMUL subprograms are used for this computation. Then, 2 x (3" – 1) data redistributions are necessary. Given that

$$\frac{2(3^{n-1} + 3^{n-2} + \cdots 3 + 1)}{2 \times 3^n - 1} = \frac{1}{2},$$

half of data redistributions are removed with the optimized triple matrix product subprogram.

5 Conclusions

As an initial step towards DDI computation, a methodology has been proposed to systematically derive DDI parallel programs for matrix multiplication. The resulting DDI programs accept a large number of input data distributions to run as efficiently as Cannon's algorithm. When the DDI parallel programs are used to multiply several matrices, it is possible to save half of data redistributions needed by a non-DDI approach. Future work will address the problem of handling input data distributions that are not accepted by the DDI programs derived in this report. In the derivation of DDI matrix multiplication, it is assumed that the target machine has as many processors as needed. Future work will address the case when data and algorithm need to be partitioned for execution on arrays of fixed "small" size. The extent to which a DDI paradigm could replace existing approaches, complement them or merely apply to special application domains is unclear and this is another issue to be clarified by future research.

References

[1] Kai Hwang, *Advanced computer architecture : parallelism, scalability, programmability,* New York, McGraw-Hill, 1993.

[2] Alain Darte and Yves Robert, "Communication-minimal mapping of uniform loop nests onto distributed memory architectures," in *Proc. Int. Conf. Application-Specific Array Processors,* pp. 1-14, Oct. 1993.

[3] Jingke Li and Marina Chen, "The data alignment phase in compiling programs for distributed-memory machines," **J.** *Parallel Distributed Computing,* vol. 13, no. 2, pp. 213-221, Oct. 1991.

[4] Siddhartha Chatterjee, John R. Gilbert and Robert Schreiber, "Mobile and replicated alignment of-arrays in data-parallel programs," in *Supercomputing'93,* pp. 420-429, Nov. 1993.

[5] J. Ramanujam, and P. Sadayappan, "Compile-time techniques for data distribution in distributed memory machines," *IEEE Trans. Parallel Distributed Syst.,* vol. 2, no. 4, pp. 472-482, Oct. 1991.

[6] Paul Feautrier, "Toward automatic distribution," Laboratoire MASI, Institut Blaise Pascal, Tech. Rep. 92-95, Dec. 1992.

[7] Kathleen Knobe, Joan D. Lukas and Guy L. Steele Jr., "Data optimization: Allocation of arrays to reduce communication on SIMD machines," **J.** *Parallel Distributed Computing,* vol. 8, no. 2, pp. 102-118, Feb. 1990.

[8] Edgar T. Kalns and Lionel M. Ni, "Processor mapping techniques toward efficient data distribution," in *Proc. 9'th Int. Parallel Processing Symp.,* .April 1994.

[9] Manish Gupta and Prithviraj Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distributed Syst.,* vol. 3, pp. 179-193, Mar. 1992.

[10] Jennifer M. Anderson, and Monica Lam, "Global optimization lor parallelism and locality on scalable parallel machines," in *Proc. ACM SIGPLAN '93 Conf. Programming Language Desing Implementation,* pp. 112-125, June 1993.

[11] Wei Li and Keshav Pingali, "Access normalization: Loop restructuring for NUMA compilers," Dep. Comput. Sci., Cornell Univ., Ithaca, NY, Tech. Rep. TR 92-1278, April 1992.

[12] Robert D. Falgout, Anthony Skjellum, Steven G. Smith, and Charles H. Still, "The multicomputer Toolbox approach to concurrent BLAS," in *Proc. Scalable High Performance Computing Conf.,* pp. 121-128, Apr. 1992.

[13] Hyuk J. Lee and José A.B. Fortes, "On the injectivity of modular mappings," in *Proc. Int. Conf. Application-Specific Array Processors,* pp. 236-247, Aug. 1994.

[14] Mar Le Fur, Jean-Louis Pazat and Francoise Andre, "Static domain analysis for compiling commutative loop nests," IRISA, Tech. Rep. 757, Sep. 1993.

[15] Sun-Yuan Kung, *VLSI array processors,* Prentice-Hall, 1988.

[16] Paul Feautrier "Some efficient solutions to the affine scheduling problem "Part I: one-dimensional time," IBP/MASI, France, Tech. Rep. 92.28, May 1992.

[17] Guo-Jie Li and Benjamin W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.,* vol. C-34, pp. 66-77, Jan. 1985.

[18] Weijia Shang and José A.B. Fortes, "Time optimal linear schedules for algorithms with uniform dependencies," *IEEE Trans. Comput.,* vol. C-40, pp. 723-742, June 1991.

[19] Alain Darte and Yves Robert, "Affine-by-statement scheduling of uniform loop nests over parametric domains," LIP, Ecole Normale Superieure de Lyon, France, Tech. Rep. 92-16, April 1992.

[20] Patrice Quinton and Vincent Van Dongen, "The mapping of linear recurrence equations on regular arrays," *Int. J. VLSI Signal Processing,* vol. 1, no. 2, pp. 95-113, 1989.

[21] Michael Wolfe, "Massive parallelism through program restructuring," in *Proc. 3rd Symp. Frontiers Massively Parallel Computation,* pp. 407-415, Oct. 1990.

[22] L.E. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. dissertation, Montana State Univ., Bozeman, MT, 1969.

[23] S. Lennart Johnsson, "Communication efficient basic linear algebra computations on hypercube architectures," *J. Parallel Distributed Computing,* vol 4, no. 2, pp. 132-172, April 1987.

[24] P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic, "Efficient matrix multiplication on SIMD computers," *SIAM J. Matrix Anal. Appl.,* vol. 13, no. 1, pp. 386-401, Jan. 1992.

[25] Jorge L. Aravena and William A. Porter, "Nonplanar switchable arrays," *Circuits Systems and Signal Processing,* vol. 7, no. 2, pp. 213-234, 1988.

[26] Jorge L. Aravena, "Triple matrix product architectures for fast signal processing," *IEEE Trans. Circuits Syst.,* vol. 35, no. 1, pp. 119-122, 1988.

A Appendix

Proposition 1

(Proof) Since $F^a = \begin{pmatrix} 1 & \\ 0 & \end{pmatrix}$ and $F^b = \begin{pmatrix} & 0 \\ & 1 \end{pmatrix}$, it follows from Eq. 17 that

$$P^{a^{-1}} = \begin{pmatrix} T_{2..3}^{-1}(1) \\ T_{2..3}^{-1}(3) \end{pmatrix}, \quad P^{b^{-1}} = \begin{pmatrix} T_{2..3}^{-1}(3) \\ T_{2..3}^{-1}(2) \end{pmatrix} \tag{24}$$

Hence, the condition of $P^{a^{-1}}(2) = P^{b^{-1}}(1)$ is obtained.

Eq. 17 should also hold for data array c. Hence,

$$P^{c^{-1}} = F^c T_{2..3}^{-1}. \tag{25}$$

Since $F^c = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, $P^{c^{-1}}$ ran be represented by $P^{a^{-1}}$ and $P^{b^{-1}}$:

$$P^{c^{-1}} = \begin{pmatrix} P^{a^{-1}}(1) \\ P^{b^{-1}}(2) \end{pmatrix} \tag{26}$$

Since $P^{c^{-1}}$ should be nonsingular, $P^{a^{-1}}(1)$ and $P^{b^{-1}}(2)$ should be linearly independent.

Proposition 2

(Proof) Suppose that $T_1^{-1} \in$ NuZ1(Fu)for a data array v where Null(Fu) denotes the null space of matrix **Fu.** Then, $\bar{\gamma}^v = \bar{0}$, i.e., data array v is not moving during the computation. Hence, in order to minimize data movements, it is desirable to find T_1^{-1} that are in the null spaces of indexing pattern matrices of as many data, arrays as possible. For matrix multiplication case, the indexing functions of three data arrays have distinct null spaces. Hence, it is impossible to find T_1^{-1} that are in the null space of more than one data array indexing function. The best choice of T_1^{-1} is a vector in the null space of one data array. Since $(c,0,0)^T \in$ NulZ(F^b), $(0,c,0)^T \in$ Null(F^a), and $(0,0,c)^T \in$ Null(F^c) for $c \in Z$, these three vectors are equally optimal choices. If $|c| > 1$, $\bar{\gamma}$ has an entry whose absolute value is larger than 1. This implies non-neighbor communication. To eliminate this possibility, $(\pm 1,0,0)^T$, $(0,\pm 1,0)^T$, and $(0,0,\pm 1)^T$ should be chosen as T_1^{-1}.

Proposition 3

(Proof) Suppose that all entries of P^a are not equal to zero. Then, it is also true that all entries of $P^{a^{-1}}$ are not equal to zero. If $T_1^{-1} \# (0,\pm 1,0)^T$, then no entry of $\bar{\gamma}^a$ is equal to zero. Hence, this is not the optimal choice. Therefore, T_1^{-1} should be $(0,\pm 1,0)^T$. Since $P^{a^{-1}}(2) = P^{b^{-1}}(1)$, no entry in $P^{b^{-1}}(1)$ is equal to zero. Therefore, no entry in P_1^b is equal to zero. On the other hand, since $F^b T_1^{-1} = (0,\pm 1)^T$, no entry of $\bar{\gamma}^b = P^b F^b T_1^{-1}$ is equal to zero. This results in non-neighbor communication in 4-way mesh interconnections.

Therefore, there should exist at least one entry equal to zero in P^a. Similar derivation can bring the same condition for P^b and P^c, too.

Lemma A.1: Suppose that a data array has a finite number of data array copies whose distribution is given by $p_k^{b_\Pi}$ where

$$p_k^{b_\Pi}(\bar{y}, t) = (P\bar{y} + \bar{p} + \bar{\gamma}(t + kb_\Pi))_{(mod\, \bar{b}_x)},$$

and $k = \lfloor \frac{(\Pi\bar{j})_{min}}{b_\Pi} \rfloor, \lfloor \frac{(\Pi\bar{j})_{min}}{b_\Pi} \rfloor + 1, \cdots, \lfloor \frac{(\Pi\bar{j})_{max}}{b_\Pi} \rfloor - 1, \lfloor \frac{(\Pi\bar{j})_{max}}{b_\Pi} \rfloor$. Given $\bar{j} \in J$,

$$\begin{pmatrix} t \\ \bar{x} \end{pmatrix} = (T\bar{j})_{mod\, \bar{m}},$$

where $\bar{x}$ denotes the index of the processor that executes computation with index $\bar{j}$ at time t. Processor $\bar{x}$ contains the correct data element $\bar{y} = F\bar{j} + \bar{f}$ at time t if

$$T(2,3)\bar{j} = PF\bar{j} + P\bar{f} + \bar{p} + \bar{\gamma}\Pi\bar{j},$$

and

$$m(2,3) = \bar{b}_X.$$

(Proof) It suffices to show that there exists $p_k^{b_\Pi}(\bar{y}, t)$ such that $(T(2,3)\bar{j})_{mod\, \bar{b}_X} = p_k^{b_\Pi}(F\bar{j} + \bar{f}, t)$. Let $k = \lfloor \frac{\Pi\bar{j}}{b_\Pi} \rfloor$, then $p_k^{b_\Pi}(\bar{y}, t) = (PF\bar{j} + P\bar{f} + \bar{p} + \bar{\gamma}(t + \lfloor \frac{\Pi\bar{j}}{b_\Pi} \rfloor))_{mod\, \bar{b}_X}$. Since $t = (\Pi\bar{j})_{mod\, b_\Pi} = \Pi\bar{j} - \lfloor \frac{\Pi\bar{j}}{b_\Pi} \rfloor b_\Pi$,

$$\begin{aligned} p_k^{b_\Pi}(F\bar{j} + \bar{f}, t) &= (PF\bar{j} + P\bar{f} + \bar{p} + \bar{\gamma}(t + \lfloor \tfrac{\Pi\bar{j}}{b_\Pi} \rfloor b_\Pi))_{mod\, \bar{b}_X} \\ &= (PF\bar{j} + P\bar{f} + \bar{p} + \bar{\gamma}(\Pi\bar{j}))_{mod\, \bar{b}_X} \\ &= (T(2,3)\bar{j})_{mod\, \bar{b}_X}. \end{aligned}$$