Automatic Data and Computation Decomposition on Distributed Memory Parallel Computers

PEIZONG LEE
Academia Sinica
and
ZVI MEIR KEDEM
New York University

To exploit parallelism on shared memory parallel computers (SMPCs), it is natural to focus on decomposing the computation (mainly by distributing the iterations of the nested Do-Loops). In contrast, on distributed memory parallel computers (DMPCs), the decomposition of computation and the distribution of data must both be handled—in order to balance the computation load and to minimize the migration of data. We propose and validate experimentally a method for handling computations and data synergistically to minimize the overall execution time on DMPCs. The method is based on a number of novel techniques, also presented in this article. The core idea is to rank the "importance" of data arrays in a program and specify some of the dominant. The intuition is that the dominant arrays are the ones whose migration would be the most expensive. Using the correspondence between iteration space mapping vectors and distributed dimensions of the dominant data array in each nested Do-loop, allows us to design algorithms for determining data and computation decompositions at the same time. Based on data distribution, computation decomposition for each nested Do-loop is determined based on either the "owner computes" rule or the "owner stores" rule with respect to the dominant data array. If all temporal dependence relations across iteration partitions are regular, we use tiling to allow pipelining and the overlapping of computation and communication. However, in order to use tiling on DMPCs, we needed to extend the existing techniques for determining tiling vectors and tile sizes, as they were originally suited for SMPCs only. The overall method is illustrated on programs for the 2D heat equation, for the Gaussian elimination with pivoting, and for the 2D fast Fourier transform on a linear processor array and on a 2D processor grid.

This work was partially supported by the NSC under Grant NSC 88-2213-E-001-008 and 89-2213-E-001-007, by DARPA/Rome AFL under Agreement F30602-96-1-0320, and by NSF under Grant CCR-94-11590.
Part of this work was carried out when Z. M. Kedem was visiting the Center for Applied Sciences and Engineering Research and the Institute of Information Science, Academia Sinica, Nankang, Taiwan, July–August 1998.
Authors' addresses: P.-Z. Lee, Institute of Information Science, Academia Sinica, Nankang, Taipei, Taiwan 11529, Republic of China; email: leepe@iis.sinica.edu.tw; Z. M. Kedem, Dept. of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY 10003-6860; email: zvi.kedem@nyu.edu.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; E.1 [**Data Structures**]: *arrays*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Computation decomposition, data alignment, data distribution, distributed-memory computers, dominant data array, iteration space mapping vector, parallelizing compilers, spatial dependence vector, temporal dependence vector, tiling techniques

1. INTRODUCTION

Distributed memory parallel computers (DMPCs) have been playing an important role in solving computationally-intensive problems, as they are relatively easily scalable; given a large number of processing elements (PEs), they are suited for effectively solving large problems—such as Grand Challenge Problems [Hwang 1993]. However, program development for DMPCs is time-consuming and error-prone, as the programmer is forced to manage both parallelism and communication [Callahan and Kennedy 1988; Rogers and Pingali 1994; Zima and Chapman 1993]. The tools generally used for managing these are the decomposition of computation and the decomposition (distribution) of data. Our key contribution is a set of integrated techniques of jointly producing decompositions for both computation and data, focusing on data distribution first, and specifying computation decomposition based on it. In the Introduction, we start by briefly reviewing some relevant previous work and then providing an introduction to our approach.

Early pioneering work dealt with mapping of Do-loops (For-loops) with regular temporal dependence relations into *systolic arrays* by exploiting pipelining opportunities in sequential programs. Iterations in a nested Do-loop were mapped using space and time transformations into PEs and a global schedule obeying a semantically required partial order. For some theoretical and experimental work in this area, see Chen [1988], Huang and Lengauer [1987], Kung and Leiserson [1980], Lam [1987], Lee and Kedem [1988, 1990a, 1990b], Moldovan and Fortes [1986], Ribas [1990], Shang and Fortes [1992], and Tseng [1990].

As in general, the number of iterations in a nested Do-Loop is much larger than the number of PEs, a set of iterations called a *tile* is assigned to each PE, with the property that they can be executed in the PE without communication with other PEs. Of course, there cannot be any cyclic dependences among (the iterations in) the tiles. In Irigoin and Triolet [1988], a sufficient condition for existence of tiles without size restriction was presented. In Hogstedt et al. [1999], the relationship between the shape of the tiles and the execution time was studied. For a subclass of loops that constituted rectilinear iteration spaces, a closed form formula for their execution time was also presented. Others concentrated on finding tiles with size restriction to minimize execution time and communication [Boulet et al. 1994; Desprex et al. 1998; Hodzic and Shang 1998; Ramanujam and Sadayappan 1992; Wolf and Lam 1991; Xue 1997]. Previous work, however, addressed only tiling the iteration space of a single nested

Do-loop on (effectively) shared memory devices. Thus, data distribution was not considered, making this work too restrictive for DMPCs, where, for example, consideration must be given to minimizing the cost of data reorganization between consecutive Do-Loops.

Results were also obtained on deriving *communication-free* properties through loop transformations and data replication. If the null space of the space generated by temporal dependence vectors in a nested Do-loop is not empty, then if read-only data can be replicated there exists a communication-free computation decomposition, whose partitioning hyperplanes are perpendicular to a basis of that null space [Chen and Sheu 1994]. It is also possible to formulate equations for mapping both iteration space and data space into PEs, and then to find communication-free properties or data and computation decomposition properties of nested Do-loops [Anderson 1997; Huang and Sadayappan 1993; Ning et al. 1995; Ramanujam and Sadayappan 1991; Shih et al. 1996; Wolf and Lam 1991]. In Couvertier-Reyes [1996], additional methods were proposed for determining communication-free solutions for the computation and data alignment problem. However, partitioning hyperplanes found by the above methods frequently are not perpendicular to any axis of the iteration space or the data space. This implies that data arrays are not distributed independently along each dimension; for example, data arrays are stored among PEs in a skewed manner.

In Lim and Lam [1994, 1998] and Lim et al. [1999], an affine partitioning framework was presented to map each statement into the time or processor space; this could maximize parallelism while minimizing synchronization and communication for multiple Do-loops on (distributed) shared-memory computers. In Barua et al. [1996], a cost model that estimated the cost of communication and data partitioning for a Do-loop by a tiling method was presented [Agarwal et al. 1993, 1995]. Furthermore, a heuristic algorithm was proposed to deal with multiple Do-loops and data arrays. Both methods allowed to tile iteration space by nonrectangular blocks, and therefore data arrays might be stored among PEs in a skewed manner. In addition, when there are multiple Do-loops and data arrays, data distributions for them might conflict. As an exhaustive search to find the optimal solution was NP-hard and not practical, a good heuristic algorithm was important [Barua et al. 1996].

However, as mentioned above, although data distributions may be ignored on shared memory model, they are a crucial factor for gaining performance on DMPCs, as a remote memory reference is much more expensive than a local memory reference. To support data parallel programming, current High-Performance Fortran (HPF) standard only allows data arrays to be distributed in *block*, *cyclic*, *block-cyclic*, *replicated*, *fixed*, or *not-distributed* fashions [Koelbel et al. 1994]. Communication free and affine partitioning approaches only can be adopted with additional restrictions on DMPCs. All the methods employed in systolic algorithm design, tiling, communication-free design, and affine partitioning, belong to the computation decomposition approach. To use these methods, additional data distribution constraints are needed so that they can be employed for DMPCs.

Recently, *component alignment algorithms* for guiding data distributions and scheduling computation based on the owner computes rule became prominent [Gupta and Banerjee 1992a; Lee 1995b; Li and Chen 1991b]. Data realignment between program fragments can also be minimized by comparing the relative costs of different data distribution schemes, in which alignment relations and candidate distributions were represented by a weighted graph in which computation cost for a program fragment was associated with each node and redistribution cost was associated with each edge. Determining whether data redistributions were necessary, was reduced to a path-finding problem [Chatterjee et al. 1993, 1994a, 1994b; Gupta and Krishnamurthy 1998; Lee 1997; Palermo 1996]. Among them, methods by Chatterjee et al., Gupta and Krishnamurthy, and Palermo adopted a top-down approach and Lee adopted a bottom-up approach.

0–1 integer programming approaches were also proposed for deciding dynamic data distributions in the interphase data layout problem [Kennedy and Kremer 1998; Kremer 1995, 1998]. With an exhaustive search, it was possible to obtain a good data alignment and to determine data distribution. However, there is a cyclical dependence while formulating cost models for data distribution and communication. A data distribution scheme must be given before analyzing communication cost, but the determination of whether a data distribution scheme is good or not really depends on which communication primitives are involved. In order to not increase the search space too much (which itself is of exponential order), only row-wise and column-wise or other restricted data distribution schemes can be considered. Therefore, candidate data distributions found might be only suboptimal. Due to temporal dependence relations, block sizes of block-cyclic distributions and tile sizes are closely related to computation decomposition, and therefore for good performance, both data and computation decomposition are important. Both of them should be determined together, and not by optimizing one of them first and then trying to fit the second one to it. For a complete survey of other data distribution techniques, see Lee [1997].

In general, component alignment approaches are very promising for DMPCs, because dimensions on each data array can be distributed independently from each other, and following the HPF standard. What needs to be done is the combination of determining data distributions for data spaces and computation decompositions for iteration spaces. The focus of our article is a framework for doing this. It includes an alignment phase and a distribution phase. The alignment phase identifies data realignment between program fragments and determines axis alignments for a program fragment, with consecutive Do-loops within the fragment sharing the same data distribution scheme. The distribution phase simultaneously determines data distributions for data spaces and computation decompositions for iteration spaces.

We use both *temporal* and *spatial* dependence vectors (we introduce the latter) for determining which dimensions of a data array should be distributed. Temporal vectors come from data dependence/use relations in the *iteration space* of a *single nested Do-loop*. Therefore, they are useful for determining computation decomposition for that Do-loop. Spatial vectors come from data dependence/use relations in the *data space* of data arrays within a *program fragment*, which may include *several* nested Do-loops. Therefore, they are useful for determining data distributions for the whole program fragment. We show how to integrate data alignment techniques and iteration space tiling techniques to optimize both data and computation decompositions.

Our approach differs from previous work, which focused on determining the computation decomposition first, and thus implicitly determining a corresponding data decomposition. This may cause either data arrays to be stored among PEs in a skewed manner, which violates data layout standards in HPF; or have different data distributions for different nested Do-loops, which may incur heavy data redistribution cost. In contrast, we focus on data decomposition first. We start by determining axis alignments for a program fragment, with consecutive Do-loops within the fragment sharing the same data distribution scheme. To decide on data decomposition, we rank the "importance" of all data arrays, and refer to some as *dominant*. Dominant arrays are those that we do not want to migrate during the computation. We establish correspondence between iteration space mapping vectors and distributed dimensions of the dominant data array in each nested Do-loop. By focusing on such dominant arrays, we are able to produce novel algorithms for determining data and computation decompositions at the same time. Once data distributions are determined, based on either the owner computes rule or the owner stores rule with respect to the dominant data array, computation decomposition for each nested Do-loop is determined. If all temporal dependence relations across iteration partitions are regular, we propose algorithms to find tiling vectors and tile sizes, so that tiles satisfy the atomic computation constraint. Hence, iterations can be executed with a coarse-grain pipelining, overlapping the computation and communication time.

The rest of this article is organized as follows: Section 2 presents necessary definitions, models, assumptions, and background material. Section 3 presents an overview of our new method. Section 4 demonstrates the method by analyzing different data distributions for the two-dimensional heat equation. Section 5 proposes algorithms to determine data and computation decompositions together. Section 6 illustrates our tiling techniques on DMPCs. Section 7 presents experimental studies. Finally, some concluding remarks are given in Section 8.

2. DEFINITIONS, MODELS, ASSUMPTIONS, AND BACKGROUND MATERIAL

Grid-connected processors

The abstract target machine we adopt is P, a g-dimensional (g-D) grid of $N_1 \times N_2 \times \cdots \times N_g$ PEs, $g \geq 1$. An individual PE is represented by a tuple $(p_1, p_2, \ldots, p_g)$, where $0 \leq p_i \leq N_i - 1$. Such a grid can be embedded into almost any common DMPC. For example, a g-D grid can be embedded into a hypercube using a binary reflected Gray code [Ho 1990]. We assume that the abstract target machine is given in advance. For instance, it is known at compile time whether a 1D (or an embedded 1D) or a 2D processor array will be the target machine. Therefore, g and N_i are known for $1 \leq i \leq g$.

SPMD model

The parallel program for a grid generated from a sequential program corresponds to the SPMD (Single Program Multiple Data) model, in which each PE executes the same program but operates on possibly distinct data items [Gupta and Banerjee 1992a; Hiranandani et al. 1992; Li and Chen 1991a]. More precisely, in general, a source program has sequential parts and concurrent parts. Each PE will execute the sequential parts individually, while all the PEs will execute the concurrent parts jointly, using message passing communication primitives. In practice, scalar variables and small data arrays used in the program are generally replicated in all the PEs to reduce communication, while large data arrays are partitioned and distributed among PEs. We use the term "array" to stand for any dimensional array, including a 1D array (vector) or 2D array (matrix).

Subscripts

In this article, we analyze only those fragments of the program in which the subscript of every dimension of every array is an affine function of a single loop control index variable. So a typical subscript will be $l + is$, where l is an offset, i is a loop control index variable, and s is a stride. If a subscript is an affine function of more than one single loop control index variable or a subscript is a nonlinear function, then either dependence relations are irregular or the computation cannot be decomposed along a single dimension of the iteration space except for the innermost loop. We treat it as an irregular or restricted access pattern on DMPCs, and therefore, we prefer not to distribute that data dimension, or defer this decision as much as possible.

2.1 Data Distribution

`cyclic`(b), block, and cyclic data distributions

`cyclic`(b) distribution is the most general regular distribution, in which blocks of size b of a 1D data array are distributed among the PEs of a 1D PE array in a round-robin fashion. For example, let array $A(l:u)$ be indexed from l to u, where A is a 1D array; or, in general, some specific dimension of a high-dimensional array. We write here N for N_1. Then, under `cyclic`(b) distribution, the set of elements $A(l + pb : l + pb + b - 1)$, $A(l + (p + N)b : l + (p+N)b + b - 1)$, etc., is stored in pth PE, denoted by $\texttt{PE}_p$. Thus, the xth entry of A is stored in $\texttt{PE}_p$, where $p = \lfloor (x - l)/b \rfloor \bmod N$. We say that array A is distributed in a *cyclic* fashion if $b = 1$, in a *block* fashion if $b = \lceil (u - l + 1)/N \rceil$, and in a *block-cyclic* fashion if $1 < b < \lceil (u - l + 1)/N \rceil$.

Data decomposition

We now consider the assignment of elements of a k-D data array A, represented by $(a_1, a_2, \ldots, a_k)$, to the elements of a g-D grid P. Since data distributions for different dimensions of A are independent, we deal with data distribution for each dimension separately. Let the ith dimension of A be A_i and the jth dimension of P be P_j. We have the following four cases:

(1) A_i is *distributed* in $\texttt{cyclic}(\texttt{db}_i)$ along P_j if and only if there exists a function f_{A_i} of the form

$$f_{A_i}(x) = \lfloor (x - \texttt{doffset}_i)/(\texttt{db}_i) \rfloor \bmod N_j,$$

where $\texttt{doffset}_i$ is an offset, such that if $(a_1, \ldots, a_i, \ldots, a_k)$ is assigned to $(p_1, \ldots, p_j, \ldots, p_g)$, then $p_j = f_{A_i}(a_i)$.

(2) A_i is *replicated* along P_j if and only if any two elements of P of the form $(p_1, \ldots, p_{j-1}, p_j, p_{j+1}, \ldots, p_g)$ and $(p_1, \ldots, p_{j-1}, p'_j, p_{j+1}, \ldots, p_g)$ are assigned exactly the same elements of A.

(3) A_i is *fixed* along P_j if and only if for some constant c, every location of P of the form $(p_1, \ldots, p_{j-1}, p_j, p_{j+1}, \ldots, p_g)$, where $p_j \neq c$, is assigned no elements of A.

(4) A_i is *not distributed* along any dimension of P.

Thus, if A_i is either distributed, replicated, or fixed along some dimension of the PE grid, then the data distribution function of the entry $A_i(x)$ is of the form:

$$f_{A_i}(x) = \begin{cases} \lfloor (x - \texttt{doffset}_i)/(\texttt{db}_i) \rfloor \bmod N_{\texttt{map}(A_i)} & \text{if } A_i \text{ is distributed in } \texttt{cyclic}(\texttt{db}_i), \\ \mathrm{R} = [0 : N_{\texttt{map}(A_i)} - 1] & \text{if } A_i \text{ is replicated}, \\ \text{constant} & \text{if } A_i \text{ is fixed}, \end{cases}$$

where $\texttt{map}()$ is a one-to-one function, $1 \leq \texttt{map}(A_i) \leq g$, and $f_{A_i}(x)$ returns the PE index along the dimension $\texttt{map}(A_i)$ of the PE grid in which $A_i(x)$ is stored. Otherwise, if A_i is not distributed along any dimension of the processor grid, then $f_{A_i}(x)$ is not defined. From now on, we also use “R” to indicate replication and “$\times$” to indicate nondistribution.

Since two distinct dimensions of a single data array cannot be distributed along the same dimension of P, and since each dimension of the data array can only be distributed along at most one dimension of P, it is possible that the number of distributed dimensions of a data array is smaller than the dimensionality of P. Then, for each of the remaining dimensions of P, we can specify replication or “fixing.” We use a *data-matching vector* to specify which distributed dimensions of the array are mapped to which dimensions of P. For an (in general multidimensional) array A, a vector $\texttt{PE}_A$ of length g is defined

PE00	A(0 : 3, 0 : 9 : 3, 0 : 11)
	B(0 : 1, 0 : 11)
	B(6 : 7, 0 : 11)
	D(0 : 11)
		PE01	A(0 : 3, 1 : 10 : 3, 0 : 11)
			C(0 : 9 : 3, 0 : 11)
			D(0 : 11)
				PE02	A(0 : 3, 2 : 11 : 3, 0 : 11)
					D(0 : 11)

PE10	A(4 : 7, 0 : 9 : 3, 0 : 11)
	B(2 : 3, 0 : 11)
	B(8 : 9, 0 : 11)
	D(0 : 11)
		PE11	A(4 : 7, 1 : 10 : 3, 0 : 11)
			C(1 : 10 : 3, 0 : 11)
			D(0 : 11)
				PE12	A(4 : 7, 2 : 11 : 3, 0 : 11)
					D(0 : 11)

PE20	A(8 : 11, 0 : 9 : 3, 0 : 11)
	B(4 : 5, 0 : 11)
	B(10 : 11, 0 : 11)
	D(0 : 11)
		PE21	A(8 : 11, 1 : 10 : 3, 0 : 11)
			C(2 : 11 : 3, 0 : 11)
			D(0 : 11)
				PE22	A(8 : 11, 2 : 11 : 3, 0 : 11)
					D(0 : 11)

Fig. 1. Data distributions represented by (A(`block`, `cyclic`, $\times$) and $\texttt{PE}_A(A_1, A_2)$), (B(`cyclic`(2), $\times$) and $\texttt{PE}_B(B_1, 0)$) or (B(`cyclic`(2), 0) and $\texttt{PE}_B(B_1, B_2)$), (C(`cyclic`, $\times$) and $\texttt{PE}_C(C_1, 1)$) or (C(`cyclic`, 1) and $\texttt{PE}_C(C_1, C_2)$), and ($D(\times)$ and $\texttt{PE}_D(\mathrm{R}, \mathrm{R})$) or ($D(\mathrm{R})$ and $\texttt{PE}_D(D_1, \mathrm{R})$).

by (j is the position in the vector):

$$\texttt{PE}_A[j] = \begin{cases} A_i & \text{if the } i\text{th dimension of } A \text{ is distributed, replicated, or fixed along the } j\text{th dimension of } P, \\ \mathrm{R} & \text{if no dimension of } A \text{ is distributed along the } j\text{th dimension of } P\text{; in addition, any two elements of } P \text{ of the form } (p_1, \ldots, p_{j-1}, p_j, p_{j+1}, \ldots, p_g) \text{ and } (p_1, \ldots, p_{j-1}, p'_j, p_{j+1}, \ldots, p_g) \text{ are assigned exactly the same elements of } A, \\ \text{constant} & \text{if no dimension of } A \text{ is distributed along the } j\text{th dimension of } P\text{; in addition, for a single specific constant } c\text{, every location of } P \text{ of the form } (p_1, \ldots, p_{j-1}, p_j, p_{j+1}, \ldots, p_g)\text{, where } p_j \neq c\text{, is assigned no elements of } A. \end{cases}$$

See Figure 1 for some examples, where arrays $A(0 : 11, 0 : 11, 0 : 11)$, $B(0 : 11, 0 : 11)$, $C(0 : 11, 0 : 11)$, and $D(0 : 11)$ are distributed in a 3×3 PE grid P. We ignore the data-matching vector when there is no risk of confusion, or when P is a 1D PE array.

In order to specify the relation between the dimensions of the data space and the dimensions of the iteration space, which will be needed later, for example, as depicted in Eq. (3) in Section 2.3, we introduce the following representation of the mapping-relationship when a dimension of A is distributed along some dimension of P. Define the data space mapping operator for mapping the data space of a k-D data array A onto a g-D PE grid to be a $g \times k$ matrix $\texttt{DT}_{g\times k}$, such that

$$\texttt{DT} \circ (a_1, a_2, \ldots, a_k)^{\mathrm{T}} = (p_1, p_2, \ldots, p_g)^{\mathrm{T}}, \tag{1}$$

where $(a_1, a_2, \ldots, a_k)$ is an index of an element of the k-D data array A, $(p_1, p_2, \ldots, p_g)$ is an index of a PE in the g-D grid, and, for ease of readability, we use "$\circ$" to denote matrix/vector multiplication. If for some dimension of A, say $A_{\phi(\omega)}$, is either distributed, replicated, or fixed along the ωth dimension of the PE grid; then the ωth row of $\texttt{DT}$ is an elementary vector $\tilde{e}_{\phi(\omega)}$ with a functional operator $f_{A_{\phi(\omega)}}(a_{\phi(\omega)})$ in position $\phi(\omega)$, such that the ωth row has $f_{A_{\phi(\omega)}}(a_{\phi(\omega)})$ in position $\phi(\omega)$ and has 0's in other positions. Then, we have $f_{A_{\phi(\omega)}}(a_{\phi(\omega)}) = p_\omega$ or R or c. However, we ignore the relationship when no dimension of A is distributed along the ωth dimension of the PE grid.

For example, in Figure 1, for mapping the data space of a 3D data array $A(0:11, 0:11, 0:11)$ onto a 2D 3×3 PE grid based on the data distribution represented by $A(\mathtt{block}, \mathtt{cyclic}, \times)$ and $\mathtt{PE}_A(A_1, A_2)$, the data space mapping operator is

$$\mathtt{DT}_{2\times 3} = \begin{pmatrix} \lfloor(\mathtt{dpar}_1)/4\rfloor & 0 & 0 \\ 0 & ((\mathtt{dpar}_2) \bmod 3) & 0 \end{pmatrix},$$

where $\mathtt{dpar}_1$ and $\mathtt{dpar}_2$ are input parameters of the data space.

The dominant data array in a nested Do-loop

On DMPCs, data distributions of all data arrays have to be determined for the entire computation before the execution starts. The same holds for the computation distributions of all the iterations in the nested Do-loops. Of course, if an iteration is assigned to a PE, the data for this iteration must be in that PE during the execution of the iteration. Ideally, the distributions are such that the computational load is balanced and there is no redistribution (migration) of data during the computation. This is in general not possible, and therefore we try to minimize migration of data by finding those data arrays that are accessed the most often (later referred to as "dominant") and try not to change their assignment for as large fragments of computation as possible, while following either the "owner computes" rule or the "owner stores" rule, so that when they are accessed during the fragments, they, and other "related" arrays are in the PEs that need to access them. (We briefly discuss the owner computes and owner stores rules later in Section 2.3.)

A program may include *generated-and-used* arrays, which induce temporal dependence relations, *write-only* arrays, *read-only* arrays, and *privatization* arrays, which are only seen within a Do-loop. In each Do-Loop, we rank data arrays in a decreasing order according to this characteristic: generated-and-used > write-only > read-only > privatization; data arrays of equal characteristic are ranked by decreasing dimensionality; data arrays of equal characteristic and dimensionality are ranked by decreasing frequency of being generated and/or used in Do-loops.

We pick one of the highest ranked arrays and choose it as the *dominant* array (in the Do-Loop). Its distribution will be decided first and it will influence the decomposition of the computation (partitioning of the iteration space). Other data arrays will be distributed based on their alignment with the dominant array. The dominant array has the largest volume of accessed data. It is better not to migrate it to avoid excessive data communication.

Axis alignment

The *axis alignment* technique was introduced in Li and Chen [1991b], and further developed in, for example, Gupta and Banerjee [1992a] and Lee [1997].

We briefly describe it here so we can present our results, and for completeness include a more detailed description in the Appendix.

Data distributions are based on the alignment relations among components of arrays. Two dimensions, each from a different array, have an *affinity relation* if the two subscripts of these two dimensions are affine functions of the same (single) loop control index variable of a Do-loop. It is advantageous for these two dimensions of the two arrays to be aligned with each other, to avoid communication.

For an example, consider the program in Figure 2(a). The first dimension of u is aligned with the second dimension of q because subscripts of these two dimensions are affine functions (j and $j-1$) of the same (single) innermost loop control index variable j, and the second dimension of u is aligned with the first dimension of q because subscripts of these two dimensions are affine functions (i and $i-1$) of the same outermost loop control index variable i. Figure 2(b) shows the component affinity graph of the program, where `q1` and `u1` represent the first dimension of arrays q and u, respectively; `q2` and `u2` represent the second dimension of arrays q and u, respectively. Suppose that the target machine is a linear PE array of $N = 3$ PEs and the problem size is $m = 6$. Figure 2(c) shows data layouts of arrays q and u under a well-aligned data distribution scheme: q(`block`, $\times$) and u($\times$, `block`). It is easily seen that, during the computation, communication is required only to access read-only, boundary data from neighboring PEs. Figure 2(d) shows data layouts of arrays q and u under a not-aligned data distribution scheme: q(`block`, $\times$) and u(`block`, $\times$). Also, it is easily seen that, during the computation, data re-organization accesses among PEs are needed for performing a transpose operation.

In the Appendix, we describe how to construct component affinity graphs and how to determine axis alignment, using a standard approach. For example, in Figure 2(a), suppose that the dominant data array is u in this nested Do-loop, as in other parts of programs not shown here u is "more important" than q. The corresponding component affinity graph of the nested Do-loop is shown in Figure 2(b). Each edge needs to be assigned a weight. However, we do not discuss weights here; for this, see Gupta and Banerjee [1992a, 1992b, 1994], Lee [1997], and Li and Chen [1990, 1991a, 1991b]. The bold, dashed-line partitions array dimensions into two groups by the component alignment algorithm, so that dimensions among arrays in each group are aligned with one another. For instance, the first dimension of q is aligned with the second dimension of u and the second dimension of q is aligned with the first dimension of u.

2.2 Temporal Vectors and Spatial Vectors

In this section, we discuss dependence relations between the iteration space and the data space.

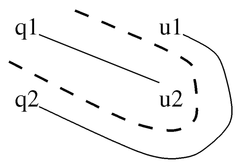
Iteration space of a depth-n nested Do-loop

Each iteration in a depth-n nested Do-loop can be represented by an n-tuple $(i_1, i_2, \ldots, i_n)$, where the value i_j is within the range of the level-j Do-loop. The *iteration space* of a depth-n nested Do-loop is the union of all its iterations. We will denote that space by $\mathcal{I}$. When it is useful to indicate n, the depth of the

(a)

```
{* q(i, 0) and u(j, 0) are zero's. *}
DO i = 1, m
   DO j = 1, m
      q(i, j) = q(i, j-1) + u(j, i) + u(j, i-1)
ENDDO ENDDO
```

(b)



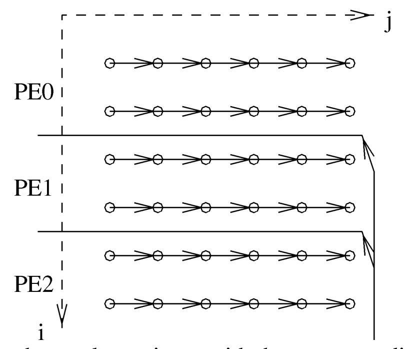
(c)

q11	q12	q13	q14	q15	q16	PE0	u11	u21	u31	u41	u51	u61
q21	q22	q23	q24	q25	q26		u12	u22	u32	u42	u52	u62
q31	q32	q33	q34	q35	q36	PE1	u13	u23	u33	u43	u53	u63
q41	q42	q43	q44	q45	q46		u14	u24	u34	u44	u54	u64
q51	q52	q53	q54	q55	q56	PE2	u15	u25	u35	u45	u55	u65
q61	q62	q63	q64	q65	q66		u16	u26	u36	u46	u56	u66

(d)

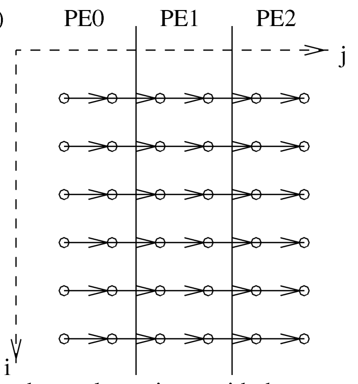
q11	q12	q13	q14	q15	q16	PE0	u11	u12	u13	u14	u15	u16
q21	q22	q23	q24	q25	q26		u21	u22	u23	u24	u25	u26
q31	q32	q33	q34	q35	q36	PE1	u31	u32	u33	u34	u35	u36
q41	q42	q43	q44	q45	q46		u41	u42	u43	u44	u45	u46
q51	q52	q53	q54	q55	q56	PE2	u51	u52	u53	u54	u55	u56
q61	q62	q63	q64	q65	q66		u61	u62	u63	u64	u65	u66

(e)



space hyperplanes i = c with the corresponding iteration space mapping vector iv = (1, 0)

(f)



space hyperplanes j = c with the corresponding iteration space mapping vector iv = (0, 1)

Fig. 2. (a) A depth-two nested Do-loop, (b) component affinity graph representing alignment relations among dimensions of arrays q and u. When the problem size $m = 6$ and the number of PEs $N = 3$, data layouts of arrays q and u under data distribution schema: (c) $q(\texttt{block}, \times)$ and $u(\times, \texttt{block})$, (d) $q(\texttt{block}, \times)$ and $u(\texttt{block}, \times)$. Cases when iteration space mapping vectors: (e) $\mathbf{iv} = (1, 0)$ and (f) $\mathbf{iv} = (0, 1)$.

Do-loop, we will write $\mathcal{I}^{(n)}$. For example, the iteration space of the depth-two nested Do-loop in Figure 2(a) is $\mathcal{I}^{(2)} = \{(i, j) \mid 1 \le i,\ j \le m\}$.

Temporal dependence vectors and temporal use vectors

In the iteration space of a nested Do-loop, each array variable may appear once, twice, or more times, resulting in its traces among the iteration space. If an array variable is first generated in some iteration α and then it is used in another iteration β, this induces one temporal dependence vector $d = \beta - \alpha$. If an array variable is used in different iterations α and β, this induces one temporal use vector $d = \beta - \alpha$. We use d_u to represent a temporal dependence vector and d_u^r to represent a temporal use vector, both for array u. (Superscript "r" stands for "read-only".) For example, in Figure 2(a), the pair $\langle q(i, j), q(i, j-1)\rangle$ induces one temporal dependence vector $d_q = (0, 1)$ and the pair $\langle u(j, i), u(j, i-1)\rangle$ induces one temporal use vector $d_u^r = (1, 0)$, both in the iteration space $\mathcal{I}^{(2)}$.

Note that, the meaning of a temporal dependence vector is the same as a data dependence vector; the meaning of a temporal use vector is the same as an input dependence vector [Wolfe 1996; Zima and Chapman 1990]. We use the phrases "temporal vectors" and "spatial vectors," which we introduce next, as we feel they better match the intuition guiding our approach. One of the referees pointed out that temporal dependences are normally just called "dependences." We add the adjective "temporal" to distinguish them from "spatial" dependences. The referee also pointed out that Kandemir and Ramanujam [2000] independently presented other data relation vectors similar to spatial vectors in this article.

In this article, we are especially interested in considering nested Do-loops with constant (i.e., regular) temporal dependence vectors, known as uniform dependence algorithms [Lee and Kedem 1990a; Shang and Fortes 1992; Xue 1997], as they will, at most, incur shift communications between neighboring PEs. For cases when some temporal dependence vector d_v is not constant (i.e., irregular), if the degree of parallelism is still greater than the dimension of the processor grid, to avoid irregular communication, we have to find a set of g iteration space mapping vectors $\mathbf{IV}$ (which we will introduce later), where g is the dimension of the target PE grid, so that $\mathbf{IV} \cdot (d_v)^{\mathrm{T}} =$ constant; otherwise, we have to use some relatively more expensive message-passing communication primitives to fetch and store data, such as broadcast, reduction, and scatter and gather by an inspector-executor technique [Wu 1995; Wu et al. 1995], which definitely will degrade the benefit of parallelism.

Spatial dependence vectors and spatial use vectors

Focusing on data arrays, we are interested in whether different variables in the same dimension are accessed simultaneously in the same iteration. For example, if $u(j, i)$ and $u(j, i-5)$ both appear in the loop body of an iteration, we associate with u a spatial vector $(0, c)$, where c is a penalty and is defined in Table I. We say that all pairs $u(j, i)$ and $u(j, i-5)$ are located at a *constant distance* (0, 5) for all indices i and j defined in the iteration space, because the difference in the first dimension is $j - j = 0$ and the difference in the second dimension is $i - (i - 5) = 5$. This indicates that for the same value (j) of the first dimension, several elements located at a constant distance with different subscripts in the second dimension will be accessed while performing an iteration.

Spatial vectors allow us to decide which dimension of an array should be fixed in PEs so that communication is not incurred. It is especially convenient to use spatial vectors when there is no nontrivial temporal vector or when temporal vectors are irregular. For example, the pair $\langle A(i, k), A(k, k)\rangle$ for $i > k$, arising

Table I. Six Different Penalties $\{0, c_1, c_2, c_3, c_4, c_5\}$ for Different Ranks of Communication Overhead Based on Two Subscripts of the Same Data Array Dimension of a Pair of Occurrences of the Same Data Array, if that Data Array Dimension is Distributed

			Penalty	
Subscripts		Communication	Dependence	Use
$f(i)$	$f(i)$	no	0	0
$f(i)$	$f(i) - c$	shift	c_2	c_1
$f(i)$	c	broadcast	c_4	c_3
c	$f(i)$	reduction	c_4	c_3
$f(i)$	unknown	gather	c_5	c_3
unknown	$f(i)$	scatter	c_5	c_3
$f_1(i)$	$f_2(j)$	many-to-many-multicast	c_5	c_3

Note: i and j are loop control index variables; $f(i)$ is an affine function of the form $a * i + b$; $f_1(i)$ and $f_2(j)$ are functions of i and j, respectively; c is a constant at compile time; and "unknown" means that the value is unknown at compile time. We use "message no. (message length)" to represent each penalty based on a 2D data array generated or used in a depth-two nested Do-loop. Then, $c_1 = 1(m)$, $c_2 = N(m)$, $c_3 = N(m^2/N)$, $c_4 = m \log N(m/N)$, $c_5 = mN(m/N)$, and $0 < c_1 < c_2 < c_3 < c_4 < c_5$, where m is the size in each data array dimension, N is the number of PEs, and a *block* distribution is adopted for the data array.

from Gaussian elimination with partial pivot, incurs a broadcast operation that cannot be represented by a constant number of temporal vectors. But it is easy to use one spatial use vector $(c_3, 0)$, where c_3 is a penalty and will be defined in Table I, to indicate that it is better not to distribute array A along the first dimension in order to avoid communication overhead due to irregular data accesses. Note that, a symbolic array variable which appears in the program is called an *array occurrence*. For example, $A(i, k)$ and $A(k, k)$ are two occurrences of array A.

Formally, we say that a pair of data array occurrences *induces a spatial dependence vector* if one occurrence is on the left-hand side (LHS) and the other occurrence is on the right-hand side (RHS) and they induce one temporal dependence vector; it *induces a spatial use vector* if both occurrences are on the RHS or if one occurrence is on the LHS and the other occurrence is on the RHS but they do not induce any temporal dependence vector.

For a spatial vector, we use six penalties $\{0, c_1, c_2, c_3, c_4, c_5\}$ to represent six different ranks (gaps) of communication overhead incurred along a data dimension, if that data dimension is distributed, depending on whether access patterns are "regular" or "irregular" and on "dependence" or "use" characteristics, as shown in Table I, where $0 < c_1 < c_2 < c_3 < c_4 < c_5$ and c_1 through c_5 are different ranks of communication overhead due to a 2D data array which is generated or used in a depth-two nested Do-loop. We assume that m is the problem size for the data array, N is the number of PEs, and the data array is distributed among PEs in a *block* fashion. The meaning of these six penalties is as follows:

(1) For a spatial dependence or use vector, if two subscripts of the same ωth dimension of the pair of occurrences are the same, then the penalty in position ω of the spatial vector is 0, which means that this will not

incur communication based on either the owner computes rule or the owner stores rule.

(2) For a spatial use vector, if the difference of these two subscripts of the same ωth dimension is a constant, then the penalty in position ω of the spatial vector is bounded by c_1, representing $1(m)$, one message of length $O(m)$, which means that this will at most incur a shift communication for read-only data before the depth-two nested Do-loop.

(3) For a spatial dependence vector, if the difference of these two subscripts of the same ωth dimension is a constant, then the penalty in position ω of the spatial vector is bounded by c_2, representing $N(m)$, N messages of length $O(m)$ each, which means that this will at most incur a shift communication for generated-and-used data for each of the N PEs in sequence.

(4) For a spatial use vector, if the difference of these two subscripts of the same ωth dimension is not a constant, then the penalty in position ω of the spatial vector is bounded by c_3, representing $N(m^2/N)$, N messages of length m^2/N each, which means that a compiler can arrange a "many-to-many-multicast" aggregate communication primitive for read-only data before the depth-two nested Do-loop. This case includes idioms such as broadcast, reduction, affine transformation, and other complex subscripts which result in replicating data in each PE.

(5) For a spatial dependence vector, if the difference of these two subscripts of the same ωth dimension is not a constant but the access pattern can be identified at compile time, then the penalty in position ω of the spatial vector is bounded by c_4, representing $m \log N(m/N)$, $m \log N$ messages of length m/N each, which means that a compiler can arrange an aggregate communication primitive which is equivalent to $\log N$ shift operations for generated-and-used data for each iteration of a loop. This case includes idioms such as broadcast or reduction or butterfly exchanges for data along one data dimension.

(6) For a spatial dependence vector, if the difference of these two subscripts of the same ωth dimension is not a constant and the access pattern cannot be determined at compile time, then the penalty in position ω of the spatial vector is bounded by c_5, representing $mN(m/N)$, mN messages of length m/N each, which means that the access pattern only can be determined at run time by an expensive inspector-executor based scatter and gather communication primitive for generated-and-used data for each iteration of a loop. This case includes subscripts being array elements (indirect memory access), two subscripts involving different loop control index variables, and other complex subscripts such as nonlinear subscripts or a subscript comprising in more than one loop control index variable.

We use s_u to represent a spatial dependence vector and s_u^r to represent a spatial use vector, both for array u. For example, in Figure 2(a), the pair $\langle q(i, j), q(i, j-1)\rangle$ induces one spatial dependence vector $s_q = (0, c_2)$ for array q and the pair $\langle u(j, i), u(j, i-1)\rangle$ induces one spatial use vector $s_u^r = (0, c_1)$ for array u.

```
(a) DO k = 1, m
      DO j = 1, m
        DO i = 1, m
          A(i, j) = A(i, j) + B(i, k) * A(k, j)
    ENDDO ENDDO ENDDO
```

```
(b) DO j = 1, m
      DO i = 1, m
        A(i, j) = A(i, j) * B(i, j) + C(i, j)
    ENDDO ENDDO

    DO i = 1, m
      A(m, i) = A(i, i) + 2.0
    ENDDO
```

Fig. 3. (a) A depth-three nested Do-loop and (b) a program fragment containing two nested Do-loops.

Note that the estimated overhead of communication primitives has been determined elsewhere. In Gupta and Banerjee [1992a, 1992b, 1994], Lee [1997], and Li Chen [1990, 1991a], communication cost was estimated, based on the assumption that an array A (LHS array) uses values of an array B or A itself (RHS array), where A and B are different. However, in Table I, the communication cost is estimated based on two subscripts of the same data array dimension. This communication cost is used to determine the data distribution of a dominant data array, which we will introduce in Section 5. Other data arrays will then align with this dominant data array.

It is instructive to compare the overhead of communication primitives listed in Table I to the overhead by a matrix transposition, which is frequently used for data redistribution. A matrix transposition can be done either by a $(\log N)$-step cascade-sum technique [Fox et al. 1988], using $\log N$ messages of length $m^2/2N$ each (represented by $\log N(m^2/2N)$), or by an ad hoc method, using N messages of length m^2/N^2 each (represented by $N(m^2/N^2)$). The communication cost of several N shift operations is smaller than the communication cost of a matrix transposition, while the communication cost of several N or $\log N$ other aggregate communication operations in the remaining set plus separate small-size computations among them is more expensive than that of a matrix transposition plus the total computations. (As before, m is the problem size and N is the number of PEs.) Therefore, we will treat a shift communication primitive as an inexpensive operation and other aggregate communication primitives as expensive operations. Incidentally, shift operations are due to regular dependence/use relations and other communication operations are due to irregular dependence/use relations.

The respective roles of temporal and spatial vectors

Temporal vectors and spatial vectors and their implications are quite different. Temporal vectors come from data dependence/use relations among the *iteration space* of a *single nested Do-loop*. Therefore, they are useful for determining computation decomposition for that specific Do-loop. Spatial vectors come from data dependence/use relations among the *data space* of data arrays within a *program fragment*, which possibly includes *several* nested Do-loops. Therefore, they are useful for determining data distributions for the whole program fragment. For example, Figure 3(a) shows a depth-three nested Do-loop program in which the pair $\langle A(i, j), A(k, j)\rangle$ induces two irregular temporal dependence vectors $d_A = \{(0, 0, \gamma), (1, 0, -\delta)\}$, where $0 \le \gamma,\ \delta < m$; and also induces one spatial

```
(a)  DO j = 1, m                    (b)  DO j = 1, m
       DO i = 1, m                         DO i = 1, m
         A(i, j) = A(i-1, j) + j             A(i, j) = A(i-1, j) + j
         B(i, j) = B(i, j-1) + i         ENDDO ENDDO
     ENDDO ENDDO
                                         DO j = 1, m
                                           DO i = 1, m
                                             B(i, j) = B(i, j-1) + i
                                         ENDDO ENDDO
```

Fig. 4. (a) The original Do-loop. (b) Two Do-loops after loop fission.

dependence vector $s_A = (c_5, 0)$. During computation decomposition, in order to avoid communication among PEs due to temporal dependence relations, we distribute the iteration space of the nested Do-loop along its second dimension among PEs. As values in the second dimension of all temporal dependence vectors are all zeros, there is no dependence relation along the second dimension in the iteration space. During data decomposition of array A, we distribute the data space of A along its second dimension among PEs. As the value in the second dimension of the spatial dependence vector is zero, subscripts of data occurrences appearing in each iteration have the same single value along the second dimension of the data space. Therefore, the distribution of data space along the second dimension will not incur communication.

Spatial vectors can help determine a quick and good solution for data distribution. Figure 3(b) shows a program fragment, which contains two Do-loops. There are no temporal dependence relations within this program fragment; however, there is a spatial use vector $s_A^{\mathrm{r}} = (c_3, 0)$ due to the pair $\langle A(m, i), A(i, i)\rangle$ for $i \leq m$. Of course, computation decompositions for these two Do-loops *cannot* be determined based on the nonexistent temporal dependence relation. However, based on the spatial use vector $s_A^{\mathrm{r}} = (c_3, 0)$, we *can* decide to distribute array A along its second dimension. Then, based on the owner computes rule, the iteration space of the first Do-loop is decomposed along its first dimension.

Relevant dependence relations

Temporal dependence relations influence parallelism, while spatial dependence relations influence data distribution. Unrelated dependence relations will degrade parallelism and increase communication. For example, in Figure 4(a), a Do-loop contains two statements, which have no relation, but they induce two pairs of temporal/spatial dependence vectors $d_A = (1, 0)$, $s_A = (c_2, 0)$, $d_B = (0, 1)$ and $s_B = (0, c_2)$. Because $(1, 0)$ and $(0, 1)$ are a set of basis vectors, which span the iteration space, this prevents finding a communication-free solution.

However, we can apply a loop fission technique in a preprocessing step according to the data dependence graph among statements [Allen and Kennedy 1987], to make the original program more amenable to parallel execution. In Figure 4(b), the original Do-loop is fissured into two Do-loops. Now each Do-loop induces only one temporal/spatial dependence vector, and therefore we can determine a data distribution scheme for $A(\times, \texttt{block})$ and $B(\texttt{block}, \times)$ so that based on the owner computes rule the execution in both Do-loops are communication-free.

2.3 The Relation between Data and Computation Decompositions

In this section, we discuss the relation between data and computation decompositions. Computation decomposition has a representation similar to that of data decomposition. As we are really interested in matching dimensions of the data space with dimensions of the iteration space, we will omit the implementation details of the PE-iteration matching vector, which are similar to the PE data-matching vector, see Section 2.1. Note, that in general, the formalism below is similar to that of Section 2.1.

Computation decomposition

In the iteration space $\mathcal{I}^{(n)}$ of a depth-n nested Do-loop, iterations in each dimension are distributed independently in `cyclic`(b), replicated, fixed, or not-distributed fashions. Let the jth dimension of the iteration space be I_j. Iterations along every iteration space dimension I_j either will be distributed or replicated or fixed along a unique dimension of the g-D PE grid P, or will not be distributed. The iteration distribution function of the entry $I_j(y)$ is of the form

$$f_{I_j}(y) = \begin{cases} \lfloor (y - \mathtt{ioffset}_j)/(\mathtt{ib}_j) \rfloor \bmod N_{\mathtt{map}(I_j)} & \text{if } I_j \text{ is distributed in } \mathtt{cyclic}(\mathtt{ib}_j), \\ \mathrm{R} = [0 : N_{\mathtt{map}(I_j)} - 1] & \text{if } I_j \text{ is replicated,} \\ \text{constant} & \text{if } I_j \text{ is fixed,} \end{cases}$$

where $\mathtt{ioffset}_j$ is an offset, $\mathtt{map}()$ is a one-to-one function, $1 \leq \mathtt{map}(I_j) \leq g$, and $f_{I_j}(y)$ returns the PE index along the dimension $\mathtt{map}(I_j)$ of the PE grid where $I_j(y)$ is stored. Otherwise, if I_j is not distributed along any dimension of the processor grid, $f_{I_j}(y)$ is not defined.

In order to specify the relation between the dimensions of the data space and the dimensions of the iteration space as depicted in Eq. (3) later, we introduce the following representation of the mapping-relationship when a dimension of the iteration space is distributed along some dimension of P. Define the iteration space mapping operator for mapping the iteration space of a depth-n nested Do-loop onto a g-D PE grid to be a $g \times n$ matrix $\mathtt{IT}_{g\times n}$, such that

$$\mathtt{IT} \circ (i_1, i_2, \ldots, i_n)^{\mathrm{T}} = (p_1, p_2, \ldots, p_g)^{\mathrm{T}}, \tag{2}$$

where $(i_1, i_2, \ldots, i_n)$ is an index of an iteration of the depth-n nested Do-loop, $(p_1, p_2, \ldots, p_g)$ is an index of a PE on the g-D grid, and, as before, "$\circ$" denotes matrix/vector multiplication. If for some dimension of the iteration space, say $I_{\psi(\omega)}$, is either distributed, or replicated, or fixed along the ωth dimension of the PE grid; then the ωth row of $\mathtt{IT}$ is an elementary vector $\tilde{e}_{\psi(\omega)}$ with a functional operator $f_{I_{\psi(\omega)}}(i_{\psi(\omega)})$ in position $\psi(\omega)$, such that the ωth row has $f_{I_{\psi(\omega)}}(i_{\psi(\omega)})$ in position $\psi(\omega)$ and has 0's in other positions. Then, we have $f_{I_{\psi(\omega)}}(i_{\psi(\omega)}) = p_\omega$ or R or c. We ignore the relationship when no dimension of $\mathcal{I}^{(n)}$ is distributed along the ωth dimension of the PE grid.

The relation between data and computation decompositions

Consider some iteration $(i_1, i_2, \ldots, i_n)$ of a Do-loop. Assume that in this iteration, the dominant k-D data array is generated or used, so that the subscript for each dimension j is an affine function of some single loop control index variable $i_{x(j)}$. From Eqs. (1) and (2), we want to match both data and computation decompositions, such that

$$\mathtt{DT} \circ \left(\mathtt{af}_1\left(i_{x(1)}\right), \mathtt{af}_2\left(i_{x(2)}\right), \ldots, \mathtt{af}_k\left(i_{x(k)}\right)\right)^{\mathrm{T}} = \mathtt{IT} \circ \left(i_1, i_2, \ldots, i_n\right)^{\mathrm{T}}, \tag{3}$$

where $\mathtt{af}_j(i_{x(j)})$ is an affine function of a loop control index variable $i_{x(j)}$ appearing in the jth position of the subscript of the data array variable. More precisely, suppose that the ωth row of $\mathtt{DT}_{g\times k}$ is $\tilde{e}^k_{\phi(\omega)}$ with a functional operator $f_{A_{\phi(\omega)}}(a_{\phi(\omega)})$ in position $\phi(\omega)$, where $e^k_{\phi(\omega)}$ is the $\phi(\omega)$th elementary vector of the k-D data space. If the subscript of the $\phi(\omega)$th dimension of the data array involves only the level-$x(\phi(\omega))$ loop control index variable $i_{x(\phi(\omega))}$, then the ωth row of $\mathtt{IT}_{g\times n}$ is $\tilde{e}^n_{x(\phi(\omega))}$ with a functional operator $f_{I_{x(\phi(\omega))}}(i_{x(\phi(\omega))})$ in position $x(\phi(\omega))$, where $e^n_{x(\phi(\omega))}$ is the $x(\phi(\omega))$th elementary vector of the n-D iteration space. In this case, we say that there is a *correspondence* between the $\phi(\omega)$th dimension of a k-D data array A and the $x(\phi(\omega))$th elementary vector in the iteration space of a depth-n nested Do-loop. Also let $\mathtt{af}_{\phi(\omega)}(i_{x(\phi(\omega))}) = l + (i_{x(\phi(\omega))})s$. Then the block size $\mathtt{db}_{\phi(\omega)}$ of the data distribution function and the block size $\mathtt{ib}_{x(\phi(\omega))}$ of the iteration distribution function satisfy $\mathtt{db}_{\phi(\omega)} = s(\mathtt{ib}_{x(\phi(\omega))})$.

For example, based on the data distribution $q(\mathtt{block}, \times)$, $\mathtt{DT}_{1\times 2} = (\lfloor(\mathtt{dpar})/2\rfloor, 0)$, and $\mathtt{IT}_{1\times 2} = (\lfloor(\mathtt{ipar})/2\rfloor, 0)$; where $\mathtt{dpar}$ is an input parameter of the data space and $\mathtt{ipar}$ is an input parameter of the iteration space, as shown in Figure 2(c) and Figure 2(e). This holds since the subscripts of the first dimension of q involve only the outermost loop control index variable i. Based on the data distribution $u(\times, \mathtt{block})$, $\mathtt{DT}_{1\times 2} = (0, \lfloor(\mathtt{dpar})/2\rfloor)$ and $\mathtt{IT}_{1\times 2} = (\lfloor(\mathtt{ipar})/2\rfloor, 0)$ as shown in Figure 2(c) and Figure 2(e), because the subscripts of the second dimension of u only involve the outermost loop control index variable i. Based on the data distribution $u(\mathtt{block}, \times)$, $\mathtt{DT}_{1\times 2} = (\lfloor(\mathtt{dpar})/2\rfloor, 0)$ and $\mathtt{IT}_{1\times 2} = (0, \lfloor(\mathtt{ipar})/2\rfloor)$ as shown in Figure 2(d) and Figure 2(f), because the subscripts of the first dimension of u only involve the innermost loop control index variable j.

Iteration space mapping vectors

Since we can use any normal vector to represent a set of parallel hyperplanes, we use the elementary vector $e^n_{x(\phi(\omega))}$, which has 1 in position $x(\phi(\omega))$ and has 0's in other positions, to represent $\tilde{e}^n_{x(\phi(\omega))}$ with a functional operator $f_{I_{x(\phi(\omega))}}(i_{x(\phi(\omega))})$ in position $x(\phi(\omega))$. Therefore, we say that we want to find g *iteration space mapping vectors*, which are g elementary vectors corresponding to the g rows in $\mathtt{IT}_{g\times n}$. For example, Figure 2(e) shows the temporal dependence relations in the iteration space, which is partitioned by the iteration space mapping hyperplanes $i = c$, whose normal vector (the iteration space mapping vector) is $\mathbf{iv} = e_1 = (1, 0)$. Figure 2(f) shows that the iteration space is partitioned by the iteration space mapping hyperplanes $j = c$, whose normal vector is $\mathbf{iv} = e_2 = (0, 1)$.

Iteration scheduling

After partitioning the iterations among PEs, we still need to schedule the iterations in each individual PE. The global schedule has to satisfy dependence constraints. We later attempt to produce schedule with the goal of minimizing execution time, accounting for both computation and communication costs.

Owner computes rule and owner stores rule

If iteration space mapping vectors are determined based on the data distribution of the LHS array, we say that the iteration scheduling is based on the owner computes rule. If iteration space mapping vectors are determined on the basis of the data distribution of a RHS array, we say that the iteration scheduling is based on the owner stores rule. If the LHS array and the RHS arrays are aligned well, then both under the owner computes rule and the owner stores rule, communication overhead incurred is not significant. However, if the LHS array and some RHS arrays are not aligned well, then whether we use the owner computes rule or the owner stores rule, significant communication cannot be avoided. We use the owner computes rule or the owner stores rule depending on whether we prefer not to move data elements of (the dominant data array which possibly is) the LHS array or a specific RHS array, in order to minimize communication.

We continue with the example in Figure 2. Consider the data distribution scheme: $q(\texttt{block}, \times)$ and $u(\times, \texttt{block})$, as shown in Figure 2(c). Suppose that the iteration space mapping vector **iv** is chosen based on the data distribution $q(\texttt{block}, \times)$ of the LHS array q. Thus, the computation decomposition is based on the owner computes rule. Since the subscripts of the first dimension of q involve only the outermost loop control index variable i, the iteration space mapping vector **iv** is thus $(1, 0)$ as shown in Figure 2(e). But if the iteration space mapping vector **iv** is chosen based on the data distribution $u(\times, \texttt{block})$, where u is a RHS array, the computation decomposition is based on the owner stores rule. Since the subscripts of the second dimension of u also involve only the outermost loop control index variable i, the iteration space mapping vector **iv** is also $(1, 0)$. This means that under iteration space mapping vector $\mathbf{iv} = (1, 0)$, all elements of arrays q and u are stored in local memory. Therefore, under this well-aligned data distribution scheme, both under the owner computes rule and the owner stores rule, the same good result is obtained.

We now consider the other not-aligned data distribution scheme: $q(\texttt{block}, \times)$ and $u(\texttt{block}, \times)$, as shown in Figure 2(d). Suppose that the iteration space mapping vector **iv** is chosen based on the data distribution $q(\texttt{block}, \times)$ of the LHS array q. As seen above, under the owner computes rule, the iteration space mapping vector is $\mathbf{iv} = (1, 0)$. Under this computation decomposition, elements of array q are stored in local memory; however, elements of array u are not. We have to perform a transpose operation to fetch elements of array u before the computation.

Suppose that the iteration space mapping vector **iv** is chosen based on the data distribution $u(\texttt{block}, \times)$, where u is a RHS array. Thus, the computation decomposition is based on the owner stores rule. Since the subscripts of the first dimension of u involve only the innermost loop control index variable j, the iteration space mapping vector **iv** is (0, 1) as shown in Figure 2(f). Under this computation decomposition, elements of array u are stored in local memory; however, elements of array q are not. Before the computation, $\mathtt{PE}_p$ has to wait for data generated by its neighboring $\mathtt{PE}_{p-1}$, for all $p > 0$, due to the temporal dependence on q. After the computation, if $q(i, j)$ will be used again later, a transpose operation is needed to send elements $q(i, j)$ to PEs according to the data distribution of array q. Therefore, under this not-aligned data distribution scheme, whether we use the owner computes rule or the owner stores rule, significant communication cannot be avoided.

3. AN OVERVIEW OF THE NEW METHOD

As seen in Section 2.3, if the subscript of each dimension of a data array is an affine function of a single loop control index variable, then each iteration space mapping vector corresponds to a dimension of data array which is distributed. Therefore, the problem of determining data distributions for all data arrays is reduced to the problem of finding a set of iteration space mapping vectors. They are based on either the owner computes rule or the owner stores rule, following the data distribution of the dominant data array in each Do-loop. The complete procedure from determining data and computation decompositions to performing computation consists of four steps, among them, the second step is an alignment phase and the third and fourth steps comprise in a distribution phase.

Step 1. We apply the loop fission techniques according to the data dependence graph among statements [Allen and Kennedy 1987], to make the original program more amenable to parallel execution.

Step 2. We construct a component affinity graph for each Do-loop, and then we apply the dynamic programming algorithm for axis alignments to decide whether data realignment is needed between adjacent program fragments [Lee 1997]. After that, all Do-loops in a program fragment will share a static data distribution scheme.

Step 3. We find a data distribution scheme for each program fragment. In each program fragment, we first determine a static data distribution scheme for some of the dominant generated-and-used data arrays based on finding iteration space mapping vectors from some of the most computationally-intensive nested Do-loops, in which these data arrays are generated or used. After that, based on alignment relations, a static data distribution scheme is determined for all data arrays throughout all Do-loops in each program fragment. (A detailed algorithm is given in Section 5, while an example is presented in Section 4.)

Step 4. For the computation in each Do-loop, based on the owner computes rule or the owner stores rule, we find the corresponding iteration space mapping vectors from the data distribution of a target (the dominant) data array. If communication cannot be avoided due to regular temporal dependences, we find tiling vectors and determine tile sizes so that iterations can be executed in a coarse-grain pipelining fashion. Otherwise, if the computation only induces temporal use vectors, then there can be a communication-free computation decomposition, provided that we can replicate the required remote read-only data. (A detailed algorithm is given in Section 6, while an example is presented in Section 4.)

An algorithm to perform Step 1

In the following, we briefly describe the algorithm for performing loop fission. The structure of Do-loops in a general program can be treated as a tree or a forest, in which assignment statements are leaves and Do statements are internal nodes. We assume that statements within each Do-loop have been topologically sorted according to dependence precedences among statements in a preprocessing step. Loop fission, which is based on the *dependence level* of a Do-loop to detect whether each level-j Do-loop is parallel or not, was proposed for vectorization [Allen and Kennedy 1987]. But even for the case when some level-j Do-loops are sequential, if temporal dependence vectors are regular, we can exploit parallelism using tiling techniques. In addition, as mentioned in Section 2.2, loop fission can separate unrelated dependences in a Do-loop and thus potentially can investigate more parallelism. Furthermore, after loop fission, we can easily generate aggregate message-passing communication primitives before the outermost loop which does not induce dependence relations for read-only data. In this article, we apply loop fission to identify the execution order of nested Do-loops in sequence.

If a Do-loop contains assignment statements and other Do-loops, we apply loop fission techniques top-down as follows. Suppose that dependence relations among all k children of a parent induce k' strongly connected components. If $k' > 1$, we apply loop fission for the parent Do-loop. Now the grandparent loses one child but gains k' children. After that, we recursively deal with each of k' children. If $k' = 1$, we do not apply loop fission for the parent Do-loop, but recursively deal with each of k children.

An algorithm to perform Step 2

We follow the tree-structure of Do-loops, obtained in Step 1. We apply a dynamic programming algorithm bottom-up to decide whether consecutive Do-loops can share the same data distribution scheme as follows. Based on axis alignments, we construct a component affinity graph for each Do-loop and various component affinity graphs for consecutive Do-loops. We heuristically determine whether data redistribution is needed between adjacent program fragments. If it is better for children Do-loops to use different data distribution schemes, we do not proceed to the parent Do-loop. If it is better for them to share a static data distribution scheme, the parent Do-loop will adopt this static scheme. We repeatedly check whether the parent's and its siblings

Do-loops can share a static distribution scheme, proceeding up to the root if possible [Lee 1997].

4. A RUNNING EXAMPLE OF COMPUTING THE 2D HEAT EQUATION

To provide the reader with the intuition helpful to the understanding of the method used in determining data and computation decomposition, we use a complete example: solving the 2D heat equation on a linear processor array with N PEs and on a 2D processor grid with $N \times N$ PEs. This nontrivial example illustrates a trade-off among parallelism, data storage used, and communication overhead. If we want to maximize parallelism, then array expansion is necessary. If we want to minimize space, then privatization arrays can be used; however, a matrix transpose operation is necessary. If we can only decompose the outer loop, then privatization arrays can be used and again a matrix transpose operation is necessary. If we can also decompose the inner loop, then array expansion is necessary; however, we can use shift operations only, avoiding a matrix transpose.

Consider the program in Figure 5(a), which solves a 2D heat equation using the alternating direction implicit (ADI) method, which reduces two-dimensional problems to a succession of one-dimensional problems. The domain of the partial differential equation $u_t = b_1 u_{xx} + b_2 u_{yy}$ is the unit square. We used the Peaceman—Rachford algorithm to formulate the numerical solution of the partial differential equation as a second-order approximation by solving two sets of tridiagonal systems of linear equations. The variables of the first set of tridiagonal systems correspond to elements from each column of an intermediate matrix, and the variables of the second set of tridiagonal systems correspond to elements from each row of a target matrix [Strikwerda 1989]. Using the Thomas algorithm, we reduce a tridiagonal system of linear equations to three sets of first-order recurrence equations.

In the program, lines 1 and 2 define two functions; line 3 defines the size of the arrays used in the program; lines 4 through 8 define scalar variables; lines 9 through 12 set initial values for $u(i, j)$; and lines 13 through 38 form the computation kernel, in which lines 14 through 25 perform a column sweep and lines 26 through 37 perform a row sweep.

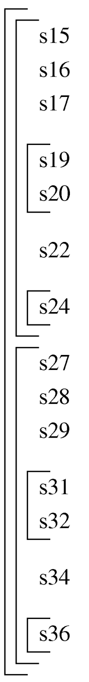
We perform Step 1 by applying loop fission to the source program. Figure 5(b) shows the structure of Do-loops included in statements from lines 13 through 38 and Figure 5(c) shows the corresponding data dependence graph among statements, where sk (letter "s" followed by integer k) denotes the statement in line number k. Due to a dependence cycle from s24 to s32 and from s36 to s20, column sweep and row sweep cannot be executed in parallel. In order to attempt more parallelism, array expansion is applied for the appropriate variables. In order to find precise data dependence vectors, we apply loop fission (loop distribution), and the original program is transformed into a sequence of nested loops, as shown in Figure 5(d).

We now continue with the example. The 2D arrays p and q are privatization arrays, which are recomputed in each loop iteration for loop control index i, and are in fact 1D arrays in a sequential program. However, in order to avoid

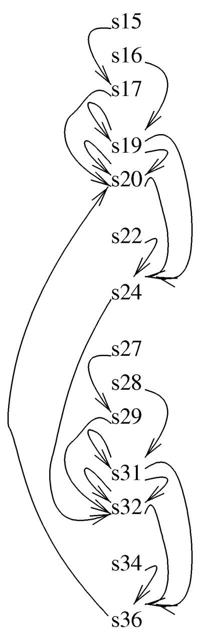
```
(a)   {*  2D heat equation: u_{t} = B1 * u_{xx} + B2 * u_{yy}.  The program is based on the ADI
          method.  The domain is a unit square: 0.0 <= x, y <= 1.0.  The spatial distances between
          grid points are DX = 1.0 / (Nx+1),   DY = 1.0 / (Ny+1).  The time interval is 0.0 <= t <= 1.0.
          The time step is DT = 1.0 / NT.  *}

1     #define eval_fun(t, x, y) = exp(1.68 * t) * sin(1.2 * (x−y)) * cosh(x + 2 * y)
2     #define boundary(t, i, j, DT, DX, DY) = {* compute boundary value for the temporary
                                                 matrix  v(i, j), where i = 0 or Nx+1. *}
3     DOUBLE  u( [0 : Nx+1], [0 : Ny+1] ),  v( [0 : Nx+1], [0 : Ny+1] ),
                  p( [0 : max{Nx, Ny}], [0 : max{Nx, Ny}+1] ),
                  q( [0 : max{Nx, Ny}], [0 : max{Nx, Ny}+1] )

4     DX = 1.0 / (Nx+1),  DY = 1.0 / (Ny+1),  DT = 1.0 / NT
5     B1 = 2.0,  mu1 = B1 * DT / (DX * DX)
6     B2 = 1.0,  mu2 = B2 * DT / (DY * DY)
7     a = −mu1 / 2.0,  b = 1.0 + mu1,  c = a
8     d = −mu2 / 2.0,  e = 1.0 + mu2,  f = d
      {*  Set u(i, j) initial value at time t = 0.0.  *}
9     DO  i = 1, Nx
10       DO  j = 0, Ny+1
11          u(i, j) = eval_fun(0.0, i * DX, j * DY)
12    ENDDO  ENDDO
      {*  Perform NT iterations. *}
13    DO  t = 1, NT
      {*  Column sweep, solve Ny tridiagonal linear systems.  *}
14       DO  i = 1, Ny
15          v(0, i) = boundary(t, 0, i, DT, DX, DY)
16          p(i, 0) = 0.0
17          q(i, 0) = v(0, i)
18          DO  j = 1, Nx
19             p(i, j) = – c / (a * p(i, j–1) + b)
20             q(i, j) = (–d * u(j, i–1) + (1.0 + 2 * d) * u(j, i)
                        –f * u(j, i+1) – a * q(i, j–1)) / (a * p(i, j–1) + b)
21          ENDDO
22          v(Nx+1, i) = boundary(t, Nx+1, i, DT, DX, DY)
23          DO  j = Nx, 1, –1
24             v(j, i) = p(i, j) * v(j+1, i) + q(i, j)
25       ENDDO  ENDDO
      {*  Row sweep, solve Nx tridiagonal linear systems.  *}
26       DO  i = 1, Nx
27          u(i, 0) = eval_fun(t * DT, i * DX, 0.0)
28          p(i, 0) = 0.0
29          q(i, 0) = u(i, 0)
30          DO  j = 1, Ny
31             p(i, j) = –f / (d * p(i, j–1) + e)
32             q(i, j) = (–a * v(i–1, j) + (1.0 + 2 * a) * v(i, j)
                        –c * v(i+1, j) – d * q(i, j–1)) / (d * p(i, j–1) + e)
33          ENDDO
34          u(i, Ny+1) = eval_fun(t * DT, i * DX, 1.0)
35          DO  j = Ny, 1, –1
36             u(i, j) = p(i, j) * u(i, j+1) + q(i, j)
37       ENDDO  ENDDO
38    ENDDO
```



(b)



(c)

Fig. 5. (a) A version of 2D heat equation program, (b) structure of Do-loops including statements from lines 13 to 38, (c) data dependence relations among statements, (d) an equivalent program after loop fission, (e) component affinity graph of lines from 14 to 25, (f) component affinity graph of lines from 26 to 37.

unnecessary dependences, we apply array expansion to *p* and *q* for the analysis phase. If under an iteration space mapping transformation, *p* and *q* do not induce dependence relations among PEs during the execution, then *p* and *q* can later be recovered as the original two 1D arrays. But if *p* and *q* do induce

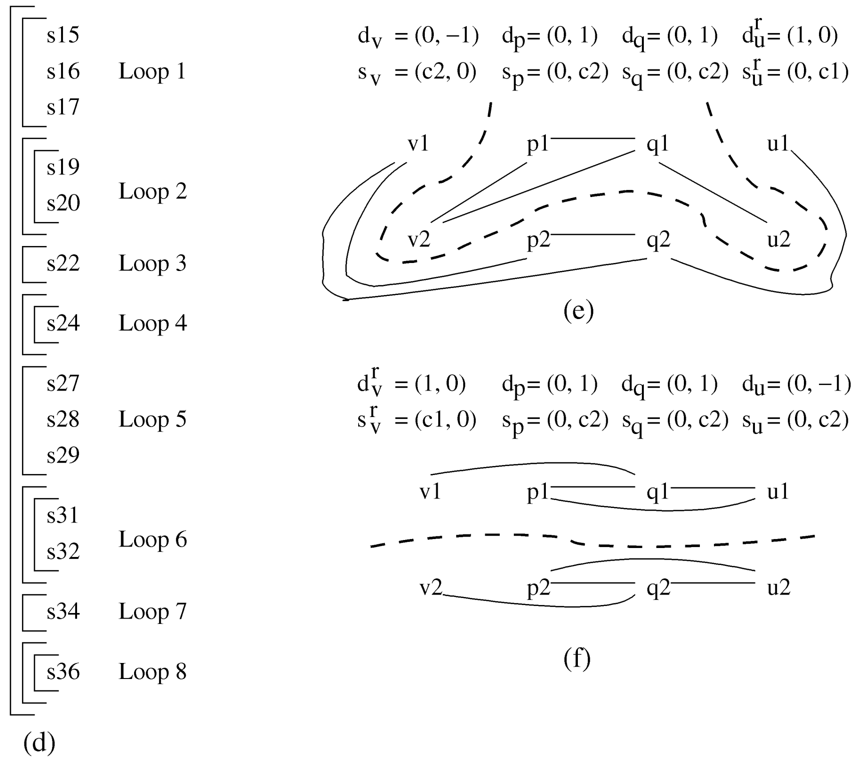

s15
s16
s17
Loop 1
s19
s20
Loop 2
s22
Loop 3
s24
Loop 4
s27
s28
s29
Loop 5
s31
s32
Loop 6
s34
Loop 7
s36
Loop 8
(d)
$d_v = (0, -1)$ $d_p = (0, 1)$ $d_q = (0, 1)$ $d_u^r = (1, 0)$
$s_v = (c2, 0)$ $s_p = (0, c2)$ $s_q = (0, c2)$ $s_u^r = (0, c1)$
v1
p1
q1
u1
v2
p2
q2
u2
(e)
$d_v^r = (1, 0)$ $d_p = (0, 1)$ $d_q = (0, 1)$ $d_u = (0, -1)$
$s_v^r = (c1, 0)$ $s_p = (0, c2)$ $s_q = (0, c2)$ $s_u = (0, c2)$
v1
p1
q1
u1
v2
p2
q2
u2
(f)

Fig. 5. (Continued).

dependence relations among PEs and tiling techniques are used, then *p* and *q* need to be maintained as two dimensional.

One of the referees remarked that the arrays *p* and *q* appear in an expanded form. It would have been appropriate to refer to them as privatized arrays if they had not been expanded with respect to the "*i*" loop—we could have then said that those (1D) arrays are privatized with respect to the "*i*" loop. Privatization, array expansion, and loop fission are frequently used in compiling programs to save data storage or to attempt more parallelism [Adve et al. 1998; Gupta 1997; Wolfe 1996; Zima and Chapman 1990]. When the parallelism is not decreased and data locality in local memory is maintained, or when we do not need additional parallelism, we can recover the original arrays made by array expansion and the original Do-loops made by loop fission; or we can even further apply loop fusion and loop interchange to improve data locality for better cache performance (which is not included in this presentation).

We now perform Step 2 to determine axis alignment for each program fragment. Figure 5(e) shows the component affinity graph of the column sweep (lines 14 through 25) and the corresponding temporal and spatial vectors. `v1` (`p1`, `q1`, and `u1`, respectively) denotes the first dimension and `v2` (`p2`, `q2`, and `u2`, respectively) denotes the second dimension of array *v* (*p*, *q*, and *u*, respectively). Note that Loops 1 through 4 can share a static data distribution scheme because axis alignments constraints are satisfied. The bold, dashed line partitions the array dimensions into two groups using the component alignment algorithm, so that array dimensions in each group are aligned with each other. For instance, the first dimension of v is aligned with the second dimension of both p and q and with the first dimension of u; the second dimension of v is aligned with the first dimension of both p and q and with the second dimension of u.

From Loop 2 in Figure 5(d), we obtain temporal and spatial vectors: $d_p = (0, 1)$, $s_p = (0, c_2)$, $d_q = (0, 1)$, $s_q = (0, c_2)$, $d_u^r = (1, 0)$, and $s_u^r = (0, c_1)$; from Loop 4, we obtain $d_v = (0, -1)$ and $s_v = (c_2, 0)$. (The second component of d_v is negative because the loop control index j is decreasing.)

Figure 5(f) shows the component affinity graph of the row sweep (lines 26 through 37) and the corresponding temporal and spatial vectors. Note that Loops 5 to 8 can also share a static data distribution scheme because axis alignments constraints are satisfied. From Loop 6 in Figure 5(d), we obtain temporal and spatial vectors: $d_p = (0, 1)$, $s_p = (0, c_2)$, $d_q = (0, 1)$, $s_q = (0, c_2)$, $d_v^r = (1, 0)$, and $s_v^r = (c_1, 0)$; from Loop 8, we can obtain $d_u = (0, -1)$ and $s_u = (0, c_2)$. In the following, we show how to compile this program for a 1D linear processor array and a 2D processor grid.

4.1 The Target Machine Is a 1D Linear Processor Array

Because all temporal vectors are regular and the dimensionality of data arrays u and v is 2, we can at least execute iterations for both dimensions in a pipelined fashion, and therefore, we have two degrees of parallelism. However, the target machine is one-dimensional, and we have to give up one degree of parallelism.

4.1.1 *Different Data Distributions for Column Sweep and Row Sweep.* We first consider the column sweep, and perform Steps 3 and 4 for it. In Step 3, we determine data distributions for the arrays. In the column sweep, v is a generated-and-used array, u is a read-only array (although u will be generated and used in the row sweep), and p and q are privatization arrays (through array expansion). v is the dominant data array and it is updated in Loop 4. Therefore, we consider all the temporal dependence vectors and the spatial dependence vectors arising from this Do-loop. There is one such temporal vector $d_v = (0, -1)$ and one such spatial vector $s_v = (c_2, 0)$. Because of the spatial vector $s_v = (c_2, 0)$, we assign a "c_2" to the first dimension of v and a "0" to the second dimension of v, and write $v(c_2, 0)$, indicating that, if the first dimension is distributed, then this will incur a penalty of c_2 for communication overhead, and, if the second dimension is distributed, then this will not incur any communication overhead.

Since the subscripts of the second dimension of v involve only the outermost loop control index variable i, the iteration space mapping vector $\mathbf{iv}$ corresponding to the second dimension of v is $(1, 0)$. As $\mathbf{iv} \cdot d_v = (1, 0) \cdot (0, -1) = 0$, the temporal dependence vector d_v will not induce communication for $\mathbf{iv} = (1, 0)$. Since the iteration space is rectangular, we choose block distribution for the second dimension of v. Therefore, following the alignment relations listed in Figure 5(e), we have the following data distributions (where, as before, "$\times$"

means "not distributed"):

$$v(\times, \texttt{block}),\quad p(\texttt{block}, \times),\quad q(\texttt{block}, \times),\quad u(\times, \texttt{block}). \tag{4}$$

We now perform Step 4, examining the actual computation. Consider the communication cost if we use data distributions listed in (4). First, iterations in either Loop 1 or Loop 3 do not induce any dependence relations, and therefore can be executed concurrently in both Do-loops. Second, in Loop 2 u is the dominant data array. Since u is distributed along the second dimension, whose subscripts involve only the outermost loop control index variable i, the iteration space mapping vector **iv** for Loop 2 is $(1, 0)$ as illustrated in Figure 6(c). As $\mathbf{iv} \cdot d_p = 0$, $\mathbf{iv} \cdot d_q = 0$, and $\mathbf{iv} \cdot d_u^r = (1, 0) \cdot (1, 0) = 1$, communications between neighboring PEs will be needed only for accessing the read-only array u. To maintain consistent memory access, some parts of u will be replicated, so that each PE has all the elements of u it needs. See Figure 6(a) for illustration, where the term, *overlap region*, is used to indicate such replication. Third, as discussed above, there is no communication while executing Loop 4. Figure 6(b) shows data layout of array v and Figure 6(d) shows temporal dependence vectors among iterations in Loop 4. Note that Loops 1 through 4 can be fused, because the iterations assigned to each PE (whether under the owner computes or owner stores rule with respect to the dominant data array in each nested Do-loop) can be executed in sequence without any dependence synchronization between neighboring PEs, once the PE has received the read-only data from neighboring PEs.

We now consider the row sweep. The discussion is very similar to that for the column sweep (though with a different result) and therefore we present it briefly. We perform Step 3 first. u is a generated-and-used array, v is a read-only array (although v has been generated and used in the column sweep), and p and q are privatization arrays. u is the dominant data array, and it is updated in Loop 8. In this Do-loop, there is one temporal dependence vector $d_u = (0, -1)$ and one spatial dependence vector $s_u = (0, c_2)$. Because of s_u, we get $u(0, c_2)$.

The iteration space mapping vector **iv** corresponding to the first dimension of u is $(1, 0)$. As $\mathbf{iv} \cdot d_u = (1, 0) \cdot (0, -1) = 0$, d_u will not induce communication for $\mathbf{iv} = (1, 0)$. Choosing block distribution for the first dimension of u and accounting for alignment relations listed in Figure 5(f), we have the following data distributions for the row sweep:

$$v(\texttt{block}, \times),\quad p(\texttt{block}, \times),\quad q(\texttt{block}, \times),\quad u(\texttt{block}, \times). \tag{5}$$

We now perform Step 4 and consider the communication overhead for data distributions listed in (5). First, iterations in either Loop 5 or Loop 7 do not induce any dependence relations and therefore they can be executed concurrently in both Do-loops. Second, in Loop 6, v is the dominant data array. Since v is distributed along the first dimension, whose subscripts only involve the outermost loop control index variable i, the iteration space mapping vector **iv** for Loop 6 is $(1, 0)$ as illustrated in Figure 7(c). As $\mathbf{iv} \cdot d_p = 0$, $\mathbf{iv} \cdot d_q = 0$, and $\mathbf{iv} \cdot d_v^r = (1, 0) \cdot (1, 0) = 1$, communication is needed only for accessing the read-only array v and, as depicted in Figure 7(a), we can use an overlap region to maintain a consistent memory access of remote read-only data

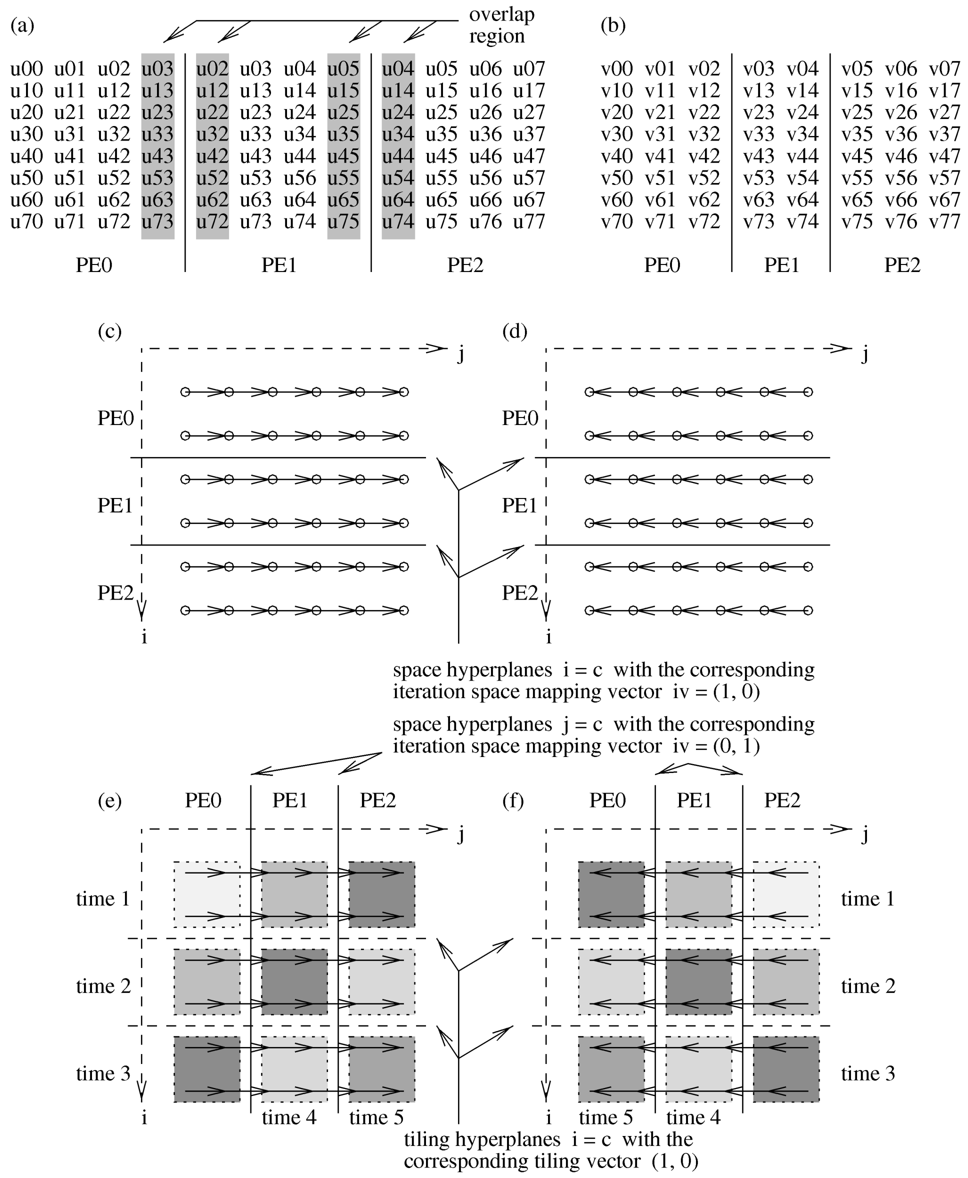

(a)
overlap region
u00 u01 u02 u03 | u02 u03 u04 u05 | u04 u05 u06 u07
u10 u11 u12 u13 | u12 u13 u14 u15 | u14 u15 u16 u17
u20 u21 u22 u23 | u22 u23 u24 u25 | u24 u25 u26 u27
u30 u31 u32 u33 | u32 u33 u34 u35 | u34 u35 u36 u37
u40 u41 u42 u43 | u42 u43 u44 u45 | u44 u45 u46 u47
u50 u51 u52 u53 | u52 u53 u56 u55 | u54 u55 u56 u57
u60 u61 u62 u63 | u62 u63 u64 u65 | u64 u65 u66 u67
u70 u71 u72 u73 | u72 u73 u74 u75 | u74 u75 u76 u77
PE0 PE1 PE2
(b)
v00 v01 v02 | v03 v04 | v05 v06 v07
v10 v11 v12 | v13 v14 | v15 v16 v17
v20 v21 v22 | v23 v24 | v25 v26 v27
v30 v31 v32 | v33 v34 | v35 v36 v37
v40 v41 v42 | v43 v44 | v45 v46 v47
v50 v51 v52 | v53 v54 | v55 v56 v57
v60 v61 v62 | v63 v64 | v65 v66 v67
v70 v71 v72 | v73 v74 | v75 v76 v77
PE0 PE1 PE2
(c)
j
PE0
PE1
PE2
i
(d)
j
PE0
PE1
PE2
i
space hyperplanes i = c with the corresponding iteration space mapping vector iv = (1, 0)
space hyperplanes j = c with the corresponding iteration space mapping vector iv = (0, 1)
(e)
PE0 PE1 PE2
j
time 1
time 2
time 3
i
time 4
time 5
(f)
PE0 PE1 PE2
j
time 1
time 2
time 3
i
time 5
time 4
tiling hyperplanes i = c with the corresponding tiling vector (1, 0)

Fig. 6. Data layout based on the schema in formula (4) when $N_x = N_y = 6$ and there are three PEs. (a) Data layout and the overlap region of array u, (b) data layout of array v, and temporal dependence among iterations in (c) Loop 2, (d) Loop 4, (e) Loop 6, and (f) Loop 8.

received from neighboring PEs. Third, there is no communication while executing Loop 8. Figure 7(b) shows data layout of array u and Figure 7(d) shows temporal dependence vectors among iterations in Loop 8. Loops 5 to 8 can be fused.

It is not surprising that the optimal data distribution for the column sweep is different from the optimal data distribution for the row sweep. If we adopt

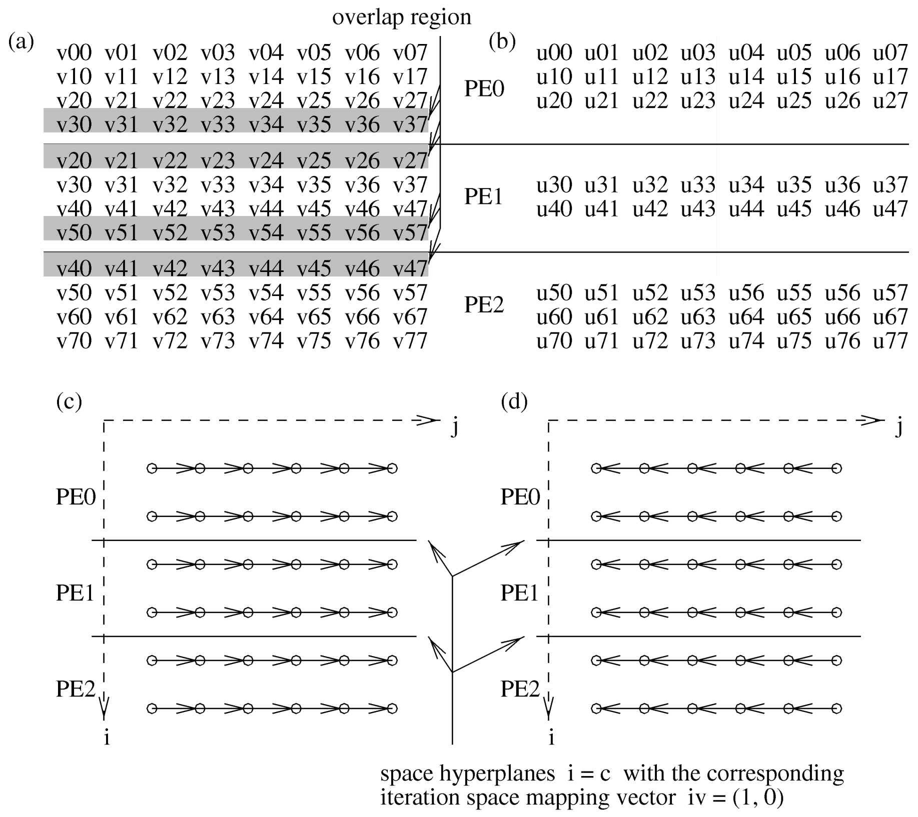

overlap region
(a)
v00 v01 v02 v03 v04 v05 v06 v07
v10 v11 v12 v13 v14 v15 v16 v17
v20 v21 v22 v23 v24 v25 v26 v27
v30 v31 v32 v33 v34 v35 v36 v37
v20 v21 v22 v23 v24 v25 v26 v27
v30 v31 v32 v33 v34 v35 v36 v37
v40 v41 v42 v43 v44 v45 v46 v47
v50 v51 v52 v53 v54 v55 v56 v57
v40 v41 v42 v43 v44 v45 v46 v47
v50 v51 v52 v53 v54 v55 v56 v57
v60 v61 v62 v63 v64 v65 v66 v67
v70 v71 v72 v73 v74 v75 v76 v77
(b)
PE0
u00 u01 u02 u03 u04 u05 u06 u07
u10 u11 u12 u13 u14 u15 u16 u17
u20 u21 u22 u23 u24 u25 u26 u27
PE1
u30 u31 u32 u33 u34 u35 u36 u37
u40 u41 u42 u43 u44 u45 u46 u47
PE2
u50 u51 u52 u53 u56 u55 u56 u57
u60 u61 u62 u63 u64 u65 u66 u67
u70 u71 u72 u73 u74 u75 u76 u77
(c)
j
PE0
PE1
PE2
i
(d)
j
PE0
PE1
PE2
i
space hyperplanes i = c with the corresponding
iteration space mapping vector iv = (1, 0)

Fig. 7. Data layout based on the schema in formula (5) when $N_x = N_y = 6$ and there are three PEs. (a) Data layout and the overlap region of array v, (b) data layout of array u, and temporal dependence among iterations in (c) Loop 6, and (d) Loop 8.

the data distributions listed in (4) and (5) for computing the column sweep and the row sweep, respectively, then we need to transpose array v from data distribution scheme listed in (4) to that of listed in (5) and to transpose array u from data distribution scheme listed in (5) to that of listed in (4). As a transpose operation is an expensive data reorganization operation, it is important to try and use a single (static) data distribution to compute both the column sweep and the row sweep with small communication overhead. We discovered that if all the temporal dependence vectors are regular, tiling techniques can help do that.

4.1.2 *Using Tiling Techniques to Reduce Communication Overhead.* Suppose that we choose the static data distribution listed in (4) for both the column sweep and the row sweep. (The symmetric case when we choose the static data distribution scheme listed in (5) will lead to a "symmetric" result.) We deal with Step 4. We have shown that the data distribution in (4) is suitable for computing the column sweep (Loop 1 through Loop 4). We now study the communication overhead for computing the row sweep (Loop 5 through Loop 8). First, iterations in either Loop 5 or Loop 7 do not induce dependence relations, and therefore can be executed concurrently in both Do-loops.

Second, in Loop 6, v is the dominant data array. Since array v is distributed along the second dimension, whose subscripts only involve the innermost loop control index variable j, the iteration space mapping vector $\mathbf{iv}$ for Loop 6 is $(0, 1)$. Because $\mathbf{iv} \cdot d_p = (0, 1) \cdot (0, 1) = 1$, $\mathbf{iv} \cdot d_q = (0, 1) \cdot (0, 1) = 1$, and $\mathbf{iv} \cdot d_v^r = (0, 1) \cdot (1, 0) = 0$, the temporal dependences of p and q will induce communication between neighboring PEs. To avoid sending many small messages between neighboring PEs, we tile the iteration space. The tiles have to satisfy the *atomic computation* constraint, that the dependence relations among tiles do not induce a cycle. After tiling, each PE executes sequentially all the iterations in a tile, then the PE sends/receives boundary data to/from neighboring PEs. The next tile can be executed by the PE using coarse grain pipelining.

Here we tile the iteration space using two sets of tiling hyperplanes. The first is the set of iteration space mapping hyperplanes represented by $j = c$, which corresponds to iteration space mapping vector $\mathbf{iv} = (0, 1)$. The second is represented by $i = c$, which corresponds to the vector $(1, 0)$. See Figure 6(e). We discuss in detail how to select tiling hyperplanes and tile sizes in Section 6.

Third, in Loop 8, u is the dominant data array. Since array u is distributed along the second dimension, whose subscripts only involve the innermost loop control index variable j, the iteration space mapping vector $\mathbf{iv}$ for Loop 8 is $(0, 1)$. As $\mathbf{iv} \cdot d_u = (0, 1) \cdot (0, -1) = -1$, communication between neighboring PEs is due to the temporal dependence of u. This case is similar to that of Loop 6, except that the innermost loop control index j is decreasing. Thus, here also we can use tiling, as depicted in Figure 6(f).

We have discussed how to use both dynamic and static data distributions for computing consecutive column and row sweeps. In general, the choice of whether to use dynamic or static data distribution possibly augmented with tiling, depends on various parameters, such as the problem size, data redistribution costs, number of PEs, communication cost, and the structure of the temporal dependence vectors (regular temporal vectors are good for tiling). This will be discussed further in the article, providing both theoretical and experimental results.

4.2 The Target Machine Is a 2D Processor Grid

Because all temporal vectors are regular and the dimensionality of data arrays u and v is equal to the dimensionality of the processor grid, which is 2, both dimensions are distributed as follows:

$$v(\texttt{block}, \texttt{block}), \quad p(\texttt{block}, \texttt{block}), \quad q(\texttt{block}, \texttt{block}), \quad u(\texttt{block}, \texttt{block}), \qquad (6)$$

where we assume that the first dimension of u and v are distributed along the first dimension of the processor grid and the second dimension of u and v are distributed along the second dimension of the processor grid. In addition, privatization arrays p and q are aligned with u and v.

In the column sweep, this involves temporal dependence vectors $d_v = (0, -1)$, $d_p = (0, 1)$, and $d_q = (0, 1)$. Because the innermost loop control index variable j only appears in the subscripts of the first dimension of v, it needs shift operations between east–west connected PEs for updating v. Also, the innermost loop control index variable j only appears in the subscripts of the second dimensions of p and q, where the second dimension of p and q are aligned with the first dimensions of u and v. Thus, it also needs shift operations between east–west connected PEs for updating p and q. Next, it involves a spatial use vector $s_u^r = (0, 1)$; thus, it needs shift operations between south–north connected PEs for fetching the read-only data array u.

In the row sweep, this involves temporal dependence vectors $d_u = (0, -1)$, $d_p = (0, 1)$, and $d_q = (0, 1)$. Because the innermost loop control index variable j only appears in the subscripts of the second dimension of u, it needs shift operations between south–north connected PEs for updating u. Also, the innermost loop control index variable j only appears in the subscripts of the second dimensions of p and q, where the second dimension of p and q are aligned with the second dimensions of u and v. Thus, it also needs shift operations between south–north connected PEs for updating p and q. Next, it involves a spatial use vector $s_v^r = (1, 0)$; thus, it needs shift operations between east–west connected PEs for fetching the read-only data array v.

5. DETERMINING DATA AND COMPUTATION DECOMPOSITIONS TOGETHER

This section describes the algorithm for Step 3 (data and computation decompositions) of the method outlined in Section 3. We assume that the program has been partitioned into program fragments as indicated in Step 2 of the proposed method in Section 3. The algorithm is run independently for each program fragment, so we assume a single program fragment in the following description. We will decide on data distribution for all generated-and-used arrays ("relevant" arrays, in the sequel).

We consider a program fragment consisting of one or more nested Do-loops. We want to find data distribution for one of the highest-dimensional generated-and-used arrays, in which g dimensions of the data array are to be distributed. That is, in the most computationally intensive nested Do-loop, which dominates the total execution time, where the target data array variables are generated or used, we want to find g iteration space mapping vectors, which correspond to g dimensions of the target data array, such that all the iterations can be mapped into the g-D grid with the execution requiring as little communication as possible. Then according to alignment relations, data distributions for other aligned data arrays can be determined. If there are still some data arrays, whose data distributions are not yet determined, we determine the data distribution for one of the highest-dimensional generated-and-used data arrays from the remaining data arrays until data distributions for all data arrays are determined.

The core of the algorithm is to specify which dimensions of each data array are to be distributed. We use eight symbols "c_0", "c_1", "c_2", "c_3", "c_4", "c_5", "$\surd$", and "$\times$", where we let $c_0 = 0$ for convenience. Each position of the k-tuple vector specifying the distribution status of each dimension of a k-D data array will contain exactly one of them in each stage of the algorithm. So, for instance, for a 5-D array A, we could have $A(c_2, \surd, c_0, c_2, \times)$. The meaning of the symbols is as follows. "c_0"–"c_5" indicate the gaps of communication penalties if that data dimension is distributed, where $c_0 = 0$ and "c_1"–"c_5" are defined in Table I. "$\surd$" indicates that we have decided to distribute on the dimension and "$\times$" indicates that we have decided not to distribute on the dimension. For every data array we are considering, we will start, of course, with all dimensions marked with c_0's. During the execution of the algorithm, "c_i" can be replaced by "c_j" for $0 \le i < j \le 5$, "$\surd$", or "$\times$". When the algorithm terminates, all the dimensions have "$\surd$" or "$\times$". As mentioned in Section 2.1, it is possible that the number of distributed dimensions (k) of a k-D data array A is smaller than the dimensionality (g) of the target g-D grid. Then, for each of the $(g - k)$ remaining dimensions of the g-D grid, we can specify replication or "fixing." If the data array A is read-only, and its replication can reduce the communication overhead, and there is still enough memory space; then we can specify replication of the data array A along these $(g - k)$ remaining dimensions of the g-D grid, otherwise, we can specify "fixing."

Including privatization arrays, we rank generated-and-used data arrays in decreasing order. We use a heuristic based on dimensionality of arrays and the frequency in which their values are generated and used in the Do-loops. Also, for each generated-and-used data array, we rank all the Do-loops in which it participates in a decreasing order based on how computationally intensive they are. Until data distributions for all relevant arrays have been determined, we repeatedly pick the highest ranked generated-and-used data array, which we did not consider before and for which its data distribution has not been completely determined, say A, and execute the steps listed below.

We find the following notation helpful. g (as before) will stand for the dimensionality of the PE grid. $|A|$ will stand for the dimensionality (not the determinant!) of A. $|\surd|$ denotes the number of $\surd$'s appearing in the dimensions of A, etc. Thus, for $A(c_2, \surd, c_0, c_2, \times)$, we have $|c_2| = 2$, $|c_0| = |\surd| = |\times| = 1$, and $|c_1| = |c_3| = |c_4| = |c_5| = 0$. Of course, $|c_0| + |c_1| + |c_2| + |c_3| + |c_4| + |c_5| + |\surd| + |\times| = |A|$ at every stage of the algorithm.

Substep 1

Action: Pick the most computationally intensive Do-loop in which A is generated or used and that has not been considered before while A was being considered. We reset to c_0's the symbols on dimensions that are not yet determined to be "$\surd$" or "$\times$," and then we replace some c_0's by c_j's for $1 \le j \le 5$ as follows. We find all spatial dependence/use vectors of A induced from this Do-loop. For each such vector, if its kth component is c_j and the kth dimension of A originally is assigned a "c_i" for $0 \le i < j \le 5$, then we put a "c_j" in the kth dimension of A.

Explanation: The condition implies that different elements (positions) of A in the kth dimension will appear in the same iteration. If the kth dimension is distributed, the highest gap of communication penalty can be determined from its spatial vectors.

Substep 2

In the general case, all eight symbols can appear in the dimensions of A (though the first time we reach this step, only c_i's for $0 \le i \le 5$, appear). If

$|\surd| = g$, we are done as we have made final decisions for all dimensions of A. Otherwise, we are not done and of course, $|\surd| < g$.

We define six candidate sets of dimensions, S_0, S_1, S_2, S_3, S_4, and S_5, where $S_i \cap S_j = \emptyset$ for $0 \leq i < j \leq 5$. S_i will contain all dimensions that have been marked with c_i's for $0 \leq i \leq 5$. Let $|S_i|$ be the cardinality of S_i and let $r \geq 0$ be the smallest value such that $\sum_{i=0}^{r} |S_i| \geq g - |\surd|$. If $\sum_{i=1}^{r} |S_i| = g - |\surd|$, we put $\surd$'s in dimensions in $\bigcup_{i=0}^{r} S_i$ of A and put $\times$'s in dimensions in $\bigcup_{i=r+1}^{5} S_i$ of A, and we are done. To obey the alignment constraints, other data arrays will inherit the $\surd$'s and $\times$'s information.

Otherwise, if $\sum_{i=0}^{r} |S_i| > g - |\surd|$, we put $\surd$'s in dimensions in $\bigcup_{i=0}^{r-1} S_i$ of A and put $\times$'s in dimensions in $\bigcup_{i=r+1}^{5} S_i$ of A. Note that the corresponding $|S_r|$ dimensions in S_r have been marked with c_r's. It remains to decide which $g - |\surd|$ dimensions of A, from those in S_r, should be selected.

We select them so as to heuristically minimize their interference with temporal dependence vectors and temporal use vectors. Before that, we find all temporal dependence vectors and temporal use vectors for all data arrays generated or used in this Do-loop. For each dimension in S_r, there is a corresponding elementary vector e_i such that a specific occurrence of A, where the owner computes rule or the owner stores rule is based in the Do-loop, refers to the level-i loop control index variable in that dimension. For each such elementary vector e_i, we define a *rank* of length three. The first component is related to the optimal choice of a tile size—which will be explained later (in Inequality (7) in Section 6). At this point, we need to know that it is better to have the flexibility to choose arbitrary tile sizes. As will be described later, if Inequality (7) is not satisfied, tile sizes are restricted if the vector e_i is chosen as an iteration space mapping vector. Therefore, the first component of the rank is 0 if e_i satisfies Inequality (7); otherwise, it is 1. If the first component of the rank is 1, it is better not to select e_i as an iteration space mapping vector, or defer this selection as late as possible. The first component of the rank will be used for such "deferral."

The second component is the number of temporal dependence vectors to which it is *not* orthogonal; the third component is the number of temporal use vectors to which it is *not* orthogonal. We order the ranks (of the vectors) in an increasing lexicographical order and for convenience, number them 1, 2, 3, ... We group the vectors into sets based on equality of ranks, say T_1, T_2, $T_3, \ldots$. Thus, a vector is in T_i if and only if its rank is i. Let $|T_i|$ be the cardinality of T_i. Choosing $g - |\surd|$ vectors from T_i for $i \geq 1$ is similar to that of choosing dimensions from $\bigcup_{i=0}^{5} S_i$ described in above. We put $\surd$'s in dimensions of A corresponding to the chosen $g - |\surd|$ vectors and put $\times$'s in dimensions of A corresponding to other vectors, and we are done.

However, it is still possible that $\bigcup_{i=1}^{r'-1} |T_i| < g - |\surd| < \bigcup_{i=1}^{r'} |T_i|$ for some r'. If data array variables of A are generated or used in other nested Do-loops that have not yet been considered, we repeat Step 1. Otherwise, if there are additional generated-and-used data arrays, whose data distributions have not yet been completely determined, then we pick a highest ranked generated-and-used data array that we did not consider before and for which its data distribution has not been completely determined, and we repeat Step 1 with the selected data array playing the role of A. If there are still remaining dimensions whose distributions need to be determined, we arbitrarily put $\surd$'s in the $g - |\surd|$ dimensions and put $\times$'s in the remaining dimensions, and we are done.

Explanation: There are correspondences between iteration space mapping vectors and distributed dimensions of the dominant data array in each nested Do-loop. In order to comply with spatial relations, data dimensions having small penalties should be distributed first. In order to comply with temporal dependence relations, iteration space mapping vectors should be in the null space of the space generated by temporal dependence vectors. Since temporal dependence vectors force the execution in sequence, and temporal use vectors may be removed by replicating the corresponding read-only data, it follows that temporal dependence vectors are "more important" than temporal use vectors.

Substep 3. /* Block sizes of data distributions are determined. */

Based on the now determined data distribution of A and on the alignment relations, data distribution of some of the other data arrays is determined. For those dimensions i that are marked with "$\surd$", we have to determine block sizes for the corresponding `cyclic(db`$_i$`)` distributions. Block sizes are chosen so that: (1) stride alignment constraints are satisfied, (2) the computational load among the PEs is balanced, and (3) communication is minimized. Currently, stride alignment constraints can be satisfied, but we still depend on table-look-up heuristics to select suitable block sizes that account properly for both load balancing and communication overhead as will be explained in the next paragraph. Although for a specific class of problem, block sizes can be determined based on finding the optimal tile sizes as described in Section 6.3, finding optimal block sizes for general cases is still an open issue.

Explanation: Block sizes have to satisfy stride alignment constraints, otherwise, irregular communication is required. For example, if $A(l_1 + is_1)$ is "axis" aligned with $C(l_2 + is_2)$, A is distributed by `cyclic(`b_1`)`, C is distributed by `cyclic(b`$_2$`)`, and $b_1/s_1 = b_2/s_2$; then their stride alignments are matched and there exist closed forms to represent communication sets. For details of generating communication sets, interested readers can refer to Lee and Chen [1997]. Next, if the iteration space is not rectangular, in order to maintain load balance, small block sizes are preferred. For example, if the iteration space is a pyramid (such as the iteration space of an LU decomposition) or a triangle (such as the iteration space of a triangular linear system), then a `cyclic(`b`)` distribution is preferred, where b is a small positive integer. However, if the iteration space is rectangular, in order to decrease communication cost due to the fetching of data from neighboring PEs, large block sizes are preferred. In general, block sizes should be a compromise for improving load balance and decreasing communication cost for the complete program fragment, for which a static data distribution scheme is adopted.

In the following, we present four examples. In the first three examples, the target machine is a linear processor array, and therefore $g = 1$. For the fourth example, both a 1D linear processor array and a 2D processor grid are used, and therefore $g = 1$ and $g = 2$, respectively.

(a)

```
DO  k = 0, m-1
  DO  j = 0, m-1
    DO  i = 0, m-1
      C(i, j) = C(i, j) + A(i, k) * B(k, j)
ENDDO  ENDDO  ENDDO
```

(b)

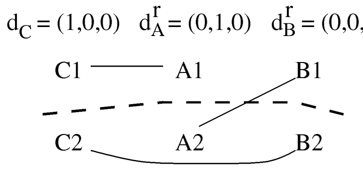


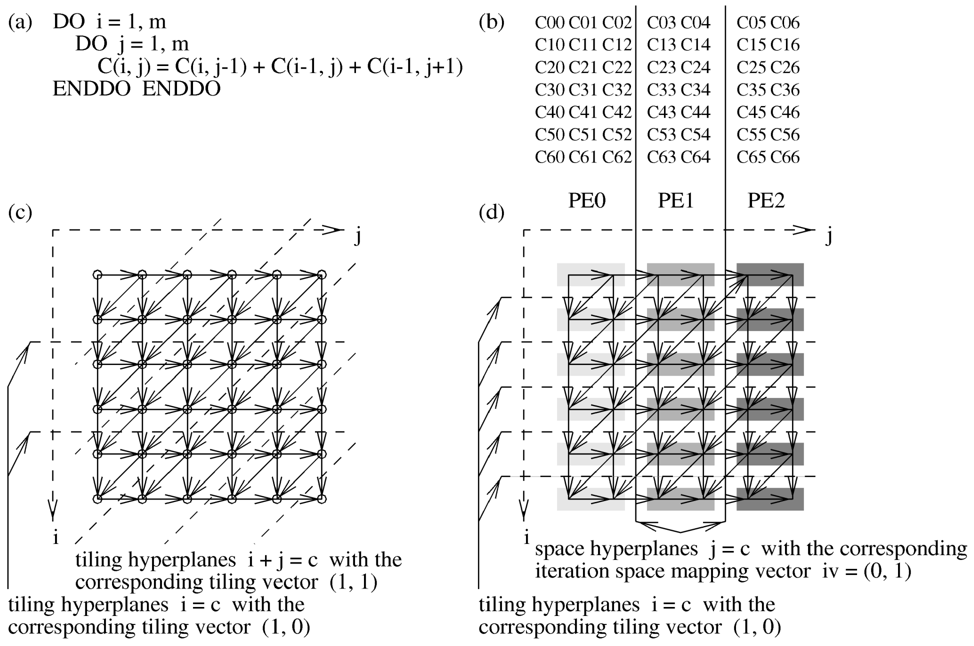
Fig. 8. (a) A matrix multiplication algorithm, (b) the corresponding component affinity graph.

Example when $g < |c_0|$: Consider the matrix multiplication algorithm as specified in Figure 8(a). The corresponding component affinity graph and temporal vectors are shown in Figure 8(b). After applying the component alignment algorithm to the component affinity graph, we obtain a partition of nodes: `C1`, `A1`, and `B1` are in one group; `C2`, `A2`, and `B2` are in the other group. Array C is the only generated-and-used data array and has no spatial vector. Thus, both dimensions of C are marked by c_0's as $C(c_0, c_0)$ and $g = 1 < |c_0| = 2$. The subscripts of the first dimension of C only involve the innermost loop control index variable i, therefore, e_3 is a candidate for the iteration space mapping vector. The subscripts of the second dimension of C only involve the level-2 loop control index variable j, therefore, e_2 is another candidate. Because $\text{rank}(e_2) = \text{rank}(e_3) = (0, 0, 1)$, $T_1 = \{e_1, e_2\}$ and $g = 1 < |T_1| = 2$. We have to sacrifice one candidate. The final iteration space mapping vector is arbitrarily chosen as e_2 for fixing array B in local memory, thus, the iteration space is decomposed along its middle Do-j loop. Because the iteration space is rectangular, `block` distribution is used. Since e_2 corresponds to the second dimension of C, according to the alignment relations, we have the following data distributions: $C(\times, \texttt{block})$, $A(\times, \texttt{block})$, and $B(\times, \texttt{block})$.

Note that, in Figure 8(b), since the dashed partition line goes across an edge (connecting `B1` and `A2`), communication overhead cannot be avoided if we can only store one copy of data array A in the PE array. But since array A is read-only, appropriately storing multiple copies of A can be used to avoid the need for communication [Lee 1995a]. This discussion of when it is worthwhile to replicate read-only data to reduce the communication cost is beyond the scope of this article.

Example when $g = |c_0|$: Consider the program fragment of the column sweep of the 2D heat equation in Figure 5(a) again. p and q are privatization arrays; u is a read-only array; thus, only v is a generated-and-used array. Array v has a spatial dependence vector $s_v = (c_2, 0)$. Therefore, we mark v to be $v(c_2, c_0)$, and $g = |c_0| = 1$. The final data distribution scheme is determined as shown in Eq. (4). Since the subscripts of the second dimension of v only involve the outermost loop control index variable i, based on the owner computes rule, the iteration space is decomposed along its outermost Do-i loop.

Example when $|c_0|+|c_1| < g < |c_0|+|c_1|+|c_2|$: Consider the depth-two nested Do-loop in Figure 9(a). Suppose that data distribution for array C has not been determined, and we proceed to do it now. Array C has two spatial dependence vectors $(0, c_2)$ and $(c_2, 0)$, therefore, we put two c_2's in both dimensions of C, thus, $C(c_2, c_2)$. We have, $|c_0| + |c_1| = 0 < g = 1 < |c_0| + |c_1| + |c_2| = 2$. The subscripts



```
(a) DO  i = 1, m
      DO  j = 1, m
        C(i, j) = C(i, j-1) + C(i-1, j) + C(i-1, j+1)
    ENDDO  ENDDO
```

(b)
C00 C01 C02 | C03 C04 | C05 C06
C10 C11 C12 | C13 C14 | C15 C16
C20 C21 C22 | C23 C24 | C25 C26
C30 C31 C32 | C33 C34 | C35 C36
C40 C41 C42 | C43 C44 | C45 C46
C50 C51 C52 | C53 C54 | C55 C56
C60 C61 C62 | C63 C64 | C65 C66
PE0
PE1
PE2
(c)
j
i
tiling hyperplanes i + j = c with the corresponding tiling vector (1, 1)
tiling hyperplanes i = c with the corresponding tiling vector (1, 0)
(d)
j
i
space hyperplanes j = c with the corresponding iteration space mapping vector iv = (0, 1)
tiling hyperplanes i = c with the corresponding tiling vector (1, 0)

Fig. 9. (a) A Do-loop has three temporal dependence vectors: (0, 1), (1, 0), and (1, −1), (b) array C is distributed as $C(\times, \texttt{block})$, (c) two tiling vectors are (1, 0) and (1, 1), (d) an iteration space mapping vector is (0, 1) and a tiling vector is (1, 0).

of the first dimension of C only involve the outermost loop control index variable i, therefore, e_1 is a candidate for the iteration space mapping vector. The subscripts of the second dimension of C only involve the innermost loop control index variable j, therefore, e_2 is another candidate. Since C has three temporal dependence vectors: (1, 0), (0, 1), and (1, −1); e_1 satisfies the Inequality (7) in Section 6 and e_2 does not. $\text{rank}(e_1) = (0, 2, 0)$ and $\text{rank}(e_2) = (1, 2, 0)$. The rank of e_1 is lexicographically smaller than the rank of e_2. Therefore, e_1 is chosen as the iteration space mapping vector, and the iteration space is decomposed along its outermost Do-i loop. Because the iteration space is rectangular, `block` distribution is used. Since subscripts of the first dimension of C only involve the outermost loop control index variable i, array C is distributed along its first dimension, say $C(\texttt{block}, \times)$.

Example when $|c_0| + |c_1| + |c_2| < g < |c_0| + |c_1| + |c_2| + |c_3|$: Figure 10 shows the Dgefa program in Linpack for computing Gaussian elimination with partial pivoting. Because there are true backward dependences from line 28 to lines 6, 8, 17, and 18, the outermost Do-k loop has to be executed sequentially. Within each outermost iteration, there is no temporal dependence relation for array A; however, there are several spatial use vectors. The pair $\langle A(i,k), A(k,k)\rangle$ for $i > k$, due to lines 6, 8, and 9 induces one spatial use vector, $s^{\text{r}}_A(1) = (c_3, 0)$; the pair $\langle A(\texttt{ip}, j), A(k, j)\rangle$ for $\texttt{ip} > k$, due to lines 17, 18, and 19 induces the same spatial use vectors; the pair $\langle A(i,k), A(k,k)\rangle$ for $i > k$, due to lines 22 and 24 induces another same spatial use vector; and two pairs $\langle A(i, j), A(k, j)\rangle$

```
    Program dgefa {* Gaussian elimination with pivot. }
1   DOUBLE A( [1 : m], [1 : m] )
2   INTEGER ipvt( [1 : m] )
3   info = 0
4   DO k = 1, m-1
    {* Find pivot. *}
5     ip = k
6     ak = dabs(A(k, k))
7     DO i = k+1, m
8       IF (dabs(A(i, k)) .GT. ak) THEN
9         ak = dabs(A(i, k))
10        ip = i
11      ENDIF
12    ENDDO
13    ipvt(k) = ip
14    IF (ak .NE. 0.0) THEN
    {* Nonzero pivot found, perform
       elimination interchange. *}
15      IF (ip .NE. k) THEN
16        DO j = k, m
17          t = A(ip, j)
18          A(ip, j) = A(k, j)
19          A(k, j) = t
20        ENDDO
21      ENDIF
    {* Compute multipliers. *}
22      t = -1.0 / A(k, k)
23      DO i = k+1, m
24        A(i, k) = A(i, k) * t
25      ENDDO
    {* Row elimination. *}
26      DO j = k+1, m
27        DO i = k+1, m
28          A(i, j) = A(i, j) + A(i, k) * A(k, j)
29      ENDDO ENDDO
30    ELSE
    {* Zero pivot, record position. *}
31      info = k
32      STOP
33    ENDIF
34  ENDDO
35  ipvt(m) = m
```

Fig. 10. A program computes Gaussian elimination with partial pivoting.

for $i > k$ and $\langle A(i,j), A(i,k)\rangle$ for $j > k$, due to line 28 induce two spatial use vectors $s^{\mathrm{r}}_{A}(1) = (c_3, 0)$ and $s^{\mathrm{r}}_{A}(2) = (0, c_3)$, respectively. Because the most computationally intensive Do-loop includes lines from 26 to 29, we put c_3's in both dimensions of A, thus, $A(c_3, c_3)$. We have $|c_0| + |c_1| + |c_2| = 0 < g \leq |c_0| + |c_1| + |c_2| + |c_3| = 2$.

If the target machine is a two-dimensional grid, thus $g = 2$, then both dimensions are distributed. Because the iteration space is a pyramid, in order to satisfy a load balance constraint, both dimension are distributed as a `cyclic`(b) distribution as $A($`cyclic`$(b),$ `cyclic`$(b))$, where b is a small positive integer. On the other hand, if the target machine is a one-dimensional linear processor array, thus $g = 1$, we have to give up one degree of parallelism. However, computation decomposition cannot be determined based on the nonexistent temporal dependence relation; it also cannot be determined based on two spatial use vectors induced from the most computationally intensive Do-loop of lines from 26 to 29, as these two spatial vectors have the same weight. However, based on the spatial use vector, $s^{\mathrm{r}}_{A}(1) = (c_3, 0)$, induced from three other less computationally intensive Do-loops, we can decide to distribute array $A(i, j)$ along its second dimension among PEs as $A(\times,$ `cyclic`$(b))$. Then, based on the owner computes rule, the iteration space for the most computationally-intensive Do-loop of lines from 26 to 29 is decomposed along its first dimension of Do-j loop, as subscripts appear in the second dimension of A only involve loop control index variable j.

6. TILING THE ITERATION SPACE ON DISTRIBUTED MEMORY MACHINES

This section discusses Step 4 of the proposed method in Section 3. On uniprocessor systems, tiling (or strip-mining) is used to improve data locality to optimize cache coherence or data reuse within the memory hierarchy [McKinley et al. 1996; Wolfe 1996]. On multiprocessor systems, tiling, in addition, is used to support coarse-grain pipelining so that data generated in one tile and used in neighboring tiles can be moved as a group, reducing communication overhead.

Returning to our setting, we concentrate on one of the nested Do-loops (recall Figure 5(d)). We decide on data distribution first, and, on that basis, on computation decomposition among PEs. We can then partition the iterations assigned to each PE into sets, called *tiles*. Then we schedule tiles globally obeying some constraints. First, a PE is executing iterations assigned to it tile by tile, possibly waiting between consecutive tiles due to dependence constraints. Second, once a PE starts executing a tile, it can complete it without waiting for other PEs. (This is the *atomic computation* constraint; the dependence relations among tiles do not induce a cycle.) We want to define tiles in a way that minimizes the total execution time.

The main difference between previous work dealing with tiling of the iterations on shared memory machines or on those machines used as peripheral parallel devices, such as systolic arrays, attached to a host computer [Boulet et al. 1994; Hodzic and Shang 1998; Lee and Kedem 1990b; Xue 1997] and this article, is the relative emphasis on data distribution vs. computation decomposition, as seen next. The optimizations that can obtained using our approach cannot be obtained using the previous methods.

6.1 Tiling on Shared Memory Machines

We start by reviewing the well-known tiling approach used for shared memory machine, where data movement is (relatively) very inexpensive. We also show by example that its utility is restricted in the context of distributed memory machines.

A depth-n nested Do-loop can be represented by an iteration space and temporal dependence/use relations among iterations. A tiling hyperplane can be represented by its normal vector, which we called the *corresponding tiling vector* in Figure 6(e) and Figure 6(f). While employing a shared memory machine model, data distribution is ignored. For data locality reasons, it is better if there are no constraints on tile size, so that cache coherence or data reuse can be optimized. A feasible set of n independent families of parallel equidistant tiling hyperplanes, which are represented by n tiling vectors, slice the iteration space into n-D parallelepiped tiles without any tile size constraint, if they satisfy the following condition [Irigoin and Triolet 1988]. For each tiling vector, say h,

$$\begin{array}{rl} \text{either} & h \cdot d_j \geq 0 \text{ for all temporal dependence vectors } d_j \\ \text{or} & h \cdot d_j \leq 0 \text{ for all temporal dependence vectors } d_j. \end{array} \qquad (7)$$

Let D be the set of all temporal dependence/use vectors. If $\text{rank}(D) < n$, there exists communication-free tiling of the iteration space; for example, with the basis of the null space of D serving as the corresponding tiling vectors. If $\text{rank}(D) = n$, then the tiling vectors can be found as follows: Consider a set of $n-1$ linearly independent temporal vectors, $d'_1, d'_2, \ldots, d'_{n-1}$ and let $h = \text{null}(d'_1, d'_2, \ldots, d'_{n-1})$ (the null space of the space generated by $d'_1, d'_2, \ldots, d'_{n-1}$). If h satisfies the constraint in Inequality (7), then h is a feasible tiling vector. By consider all such sets of $n-1$ linearly independent temporal vectors, we are guaranteed to find n linear independent tiling vectors (as $\text{rank}(D) = n$). Furthermore, there are no constraints on tile size [Boulet et al. 1994; Hodzic and Shang 1998; Xue 1997]. We refer to this method as *memory-oblivious tiling method*.

We consider now a one-dimensional distributed memory PE array, the Do-loop in Figure 9(a), and the data distribution as in Figure 9(b) (data array C is distributed along its second dimension, $C(\times, \texttt{block})$). We attempt to apply the memory-oblivious tiling method to this example. There are three temporal dependence vectors: $D = \{(0,1),\ (1,0),\ (1,-1)\}$. The rank of D is 2, the depth of the nested Do-loop, so we will consider all sets consisting of one temporal vector. The null space of $(0,1)$ is $(1,0)$, the null space of $(1,0)$ is $(0,1)$, and the null space of $(1,-1)$ is $(1,1)$. $(0,1)$ is not a feasible tiling vector because $(0,1)\cdot\{(0,1),(1,0),(1,-1)\} = \{1,0,-1\}$. $(1,0)$ and $(1,1)$ are feasible tiling vectors as depicted in Figure 9(c) because $(1,0)\cdot\{(0,1),(1,0),(1,-1)\} = \{0,1,1\}$ and $(1,1)\cdot\{(0,1),(1,0),(1,-1)\} = \{1,1,0\}$. However, the resulting computation decomposition does not match the data distribution, and therefore this tiling is not permitted. We next show, that feasible tiling exist if we search for it going beyond memory-oblivious tiling method. An example of such tiling is depicted in Figure 9(d), with the explanation following.

6.2 Tiling on Distributed Memory Machines

We consider a g-D PE grid and a nested Do-loop of depth $n > g$, to be executed on it. We assume that subscripts of one occurrence of the dominant data array are identical to some loop control index variables (possibly after a preprocessing step). Assume that data distribution for data arrays in the Do-loop has been determined. We can obtain g iteration space mapping vectors, which are elementary vectors. These vectors are naturally also tiling vectors. We only need to determine the remaining $n-g$ tiling vectors, whose $n-g$ corresponding tiling hyperplanes will slice the iterations assigned to each PE.

To finish the example of Figure 9, since the data distribution of array C is $C(\times, \texttt{block})$ and the subscripts of the second dimension of C only involve the innermost loop control index variable j, the iteration space mapping vector is $e_2 = (0,1)$, as depicted in Figure 9(d). We can choose $e_1 = (1,0)$ as the second tiling vector and the tile size as $1 \times \texttt{block}$, where the block distribution is `cyclic(block)`. Note that, we use `block` to represent both the block distribution and the block size for convenience. Then, all tiles still satisfy the atomic computation constraint.

In general, we have one of the two cases:

Case 1. *All the g iteration space mapping vectors satisfy Inequality* (7). Then, the remaining $n-g$ tiling vectors, whose corresponding tiling hyperplanes will slice iterations assigned to each PE, also can be chosen using the memory-oblivious tiling method. There are no constraints on tile sizes.

Case 2. *At least one of the g iteration space mapping vectors does not satisfy Inequality* (7). We refer to such a vector as *bad*. Let e_i be any bad vector. We want to choose a feasible tile size so that all tiles can satisfy the atomic computation constraint. Without loss of generality, the level-i loop control index is increasing. (If the level-i loop control index is decreasing, we first temporarily reverse the sign of the ith entry of all the temporal vectors; after finding the tiling vectors, we reverse the sign of the ith entry of all the tiling vectors.) Thus, the first nonzero entry of any temporal vector is positive. In addition, $i \neq 1$, because all temporal vectors d_j are lexicographically positive, and therefore $e_1 \cdot d_j \geq 0$ for every d_j, and Inequality (7) would be satisfied.

Let d_k be a temporal vector for which $e_i \cdot d_k < 0$. Let λ be the position of the first nonzero entry of d_k, say $d_{k,\lambda}$. Then e_λ is chosen as a tiling vector and the tile size along the λth dimension must be at most $d_{k,\lambda}$. Examining all such e_i's and d_k's, we get a full set of constraints on the sizes of associated tiles. The remaining tiling vectors can be chosen based on the memory-oblivious tiling method, with no constraints on the associated tile sizes. The constraints for tile sizes come from the theoretical results of *cycle shrinking* [Polychronopoulos 1988], therefore, all the tiles satisfy the atomic computation constraint. For example, in a 2D iteration space, if there is only one temporal dependence vector $(3, -1)$, then iterations within three consecutive rows have no dependence relations. Therefore, if the tile size along the first dimension is chosen as 3, this will not induce any dependence cycle among tiles.

6.3 Optimizing Tile Sizes

To minimize total execution time, it may be useful to choose tile sizes which are smaller than the upper bounds implied by the constraints above. A small tile corresponds to a fine-grain solution, incurring a large number of small communication messages among PEs. A large tile corresponds to a coarse-grain solution, incurring a long delay time among PEs due to dependent data. We provide an analytical method to determine tile shapes and sizes only for a rectangular 2D iteration space, which is represented by $1 \leq i \leq X$ and $1 \leq j \leq Y$. We assume that the distance between two parallel iteration space mapping hyperplanes is b, with b large enough so that dependent data of an iteration is either in the local memory of a PE or in its neighboring PEs. Our target machine is a linear processor array with N PEs. Thus, the data distribution function of the distributed dimension of the corresponding data array is `cyclic`(bs), where s is the stride of the affine function in the corresponding subscript. The value of b can be computed based on a near optimal tile size. Once b has been determined, then the shape of a tile can be computed.

We have the following four cases.

Case 1. *Iteration space mapping vector is* $(0, 1)$, *the other tiling vector is* $(1, 0)$, *and all the entries in each temporal vector are nonnegative as depicted in Figure* 11(a). In this case, tiles are executed column-wise. Let the tile size be $Z = b \times a$. To avoid idle time, we require $(X/a) \geq N$, or $a \leq (X/N)$. Then the total execution time is:

$$T \leq (XY/(ZN) + N - 1)(Zt_\mathrm{f} + t_\mathrm{s} + at_\mathrm{c}), \tag{8}$$

where t_f is the execution time for performing an iteration, t_s is the set-up time for

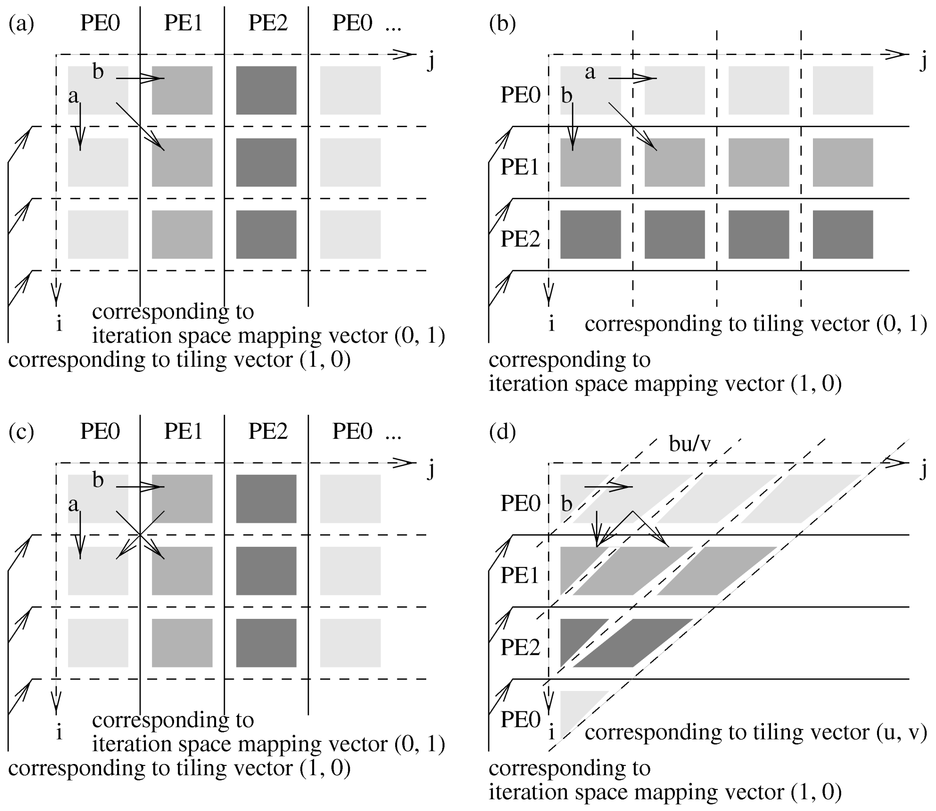

(a)
PE0
PE1
PE2
PE0 ...
b
a
j
i
corresponding to
iteration space mapping vector (0, 1)
corresponding to tiling vector (1, 0)
(b)
PE0
PE1
PE2
corresponding to tiling vector (0, 1)
corresponding to
iteration space mapping vector (1, 0)
(c)
(d)
bu/v
corresponding to tiling vector (u, v)

Fig. 11. Four cases of tiling two-dimensional rectangular iteration space.

sending–receiving a message, and t_c is the communication time of transferring a word. Since, in practice, t_s, is much larger than t_c, we ignore the term at_c in Eq. (8). We will therefore minimize $T' = (XY/(ZN) + N - 1)(Zt_f + t_s)$. Then the optimal Z is $(XYt_s/(N(N-1)t_f))^{1/2}$. (The function $f(x) = (\alpha/x + \beta)(\gamma x + \delta)$ is minimized when $x = ((\alpha\delta)/(\beta\gamma))^{1/2}$.) As T, in fact, decreases with a, we prefer for a to be small and for b to be large. If the value of b has not been determined yet, we set $b = Y/N$ if $Z \geq Y/N$, and $b = Z$ otherwise. Then $a = \min\{Z/b, X/N\}$.

Case 2. *Iteration space mapping vector is* $(1, 0)$, *the other tiling vector is* $(0, 1)$, *and all the entries in each temporal vector are nonnegative, as depicted in Figure* 11*(b)*. The derivation is similar to that of in Case 1. We get $Z = (XYt_s/(N(N-1)t_f))^{1/2}$. If the value of b has not been determined yet, we set $b = X/N$ if $Z \geq X/N$, $b = Z$ otherwise. Then $a = \min\{Z/b, Y/N\}$.

Case 3. *Iteration space mapping vector is* $(0, 1)$, *the other tiling vector is* $(1, 0)$, *and there exist temporal vectors in which one entry is positive and the other is negative, as depicted in Figure* 11*(c)*. We assume that the tiles are scheduled by an optimal data flow method, so that if more than one tile is available for execution, the lexicographically smaller one is selected. Then, the total execution time is:

$$T \leq (XY/(ZN) + 2(N-1))(Zt_{\mathrm{f}} + t_{\mathrm{s}} + at_{\mathrm{c}}). \tag{9}$$

Again we ignore the term at_{c} and obtain a near optimal tile size $Z = (XYt_{\mathrm{s}}/(2N(N-1)t_{\mathrm{f}}))^{1/2}$. If the value of b has not been determined yet, then $b = Y/N$ if $Z \geq Y/N$, and $b = Z$ otherwise. Then $a = \min\{Z/b, v\}$, where v is the smallest integer from all the positive entries appearing in the temporal vectors that have both a positive (in the first position) and a negative (in the second position) entries.

Case 4. *Iteration space mapping vector is* $(1, 0)$, *the other tiling vector is* (u, v) *when there exists a temporal vector* $(kv, -ku)$, *where u and v are relatively prime positive integers, as depicted in Figure* 11*(d)*. In this case, the shape of a tile is a parallelogram and tiles are executed in a row-wise manner. As usual, let b be the distance between two parallel iteration space mapping hyperplanes and let a be the distance between two of the other parallel tiling hyperplanes. Then a can be calculated by assuming that the line $ux + vy = c$ passes through both $(b, 0)$ and $(0, a)$. Thus, $a = bu/v$ and the tile size is b^2u/v, where we assume that b is a multiple of v. To avoid idle time, we require $(Y/(bu/v) + u/v) \geq 2N$, or $b \leq (Y/((2N - u/v)u/v))$. Then, the total execution time is:

$$\begin{aligned} T &\leq (X/(bN)(Y/(bu/v) + u/v) + (N-1)u/v)(Zt_{\mathrm{f}} + t_{\mathrm{s}} + b(u/v)t_{\mathrm{c}}) \\ &= (XY/(ZN) + (X/(bN) + N - 1)u/v)(Zt_{\mathrm{f}} + t_{\mathrm{s}} + b(u/v)t_{\mathrm{c}}). \end{aligned}$$

This case is different from the first three cases. Since $v \leq b \leq m = \min\{(Y/((2N - u/v)u/v)), X/N\}$, we let $T_v = (XY/(ZN) + (X/(vN) + N - 1)u/v)(Zt_{\mathrm{f}} + t_{\mathrm{s}} + ut_{\mathrm{c}})$, whose optimal tile size $Z_v = (XY(t_{\mathrm{s}} + ut_{\mathrm{c}})/(Nt_{\mathrm{f}}(X/(vN) + N - 1)u/v))^{1/2}$; we let $T_m = (XY/(ZN) + (X/(mN) + N - 1)u/v)(Zt_{\mathrm{f}} + t_{\mathrm{s}} + m(u/v)t_{\mathrm{c}})$, whose optimal tile size $Z_m = (XY(t_{\mathrm{s}} + m(u/v)t_{\mathrm{c}})/(Nt_{\mathrm{f}}(X/(mN) + N - 1)u/v))^{1/2}$; and we let a near optimal tile size $Z = (Z_v + Z_m)/2$. If the value of b has not been determined yet, then $b = \min\{(Zv/u)^{1/2}, (Y/((2N - u/v)u/v)), X/N\}$.

7. EXPERIMENTAL STUDIES

We validated the usefulness of our method by evaluating several implementations of three applications: the two-dimensional heat equation, the two-dimensional Fast Fourier Transform, and the Gaussian elimination with pivoting, all on one-dimensional PE arrays. The actual experiments were conducted on the two machines that were available to us: a 32-node nCUBE-2 computer and a cluster of four UltraSPARC-I workstations both located at Academia Sinica. In the nCUBE-2, each node runs at a modest clock rate of 20 MHz and has 4 MB of RAM. The four UltraSPARC-I workstations, each with 64 MB of RAM, run at the clock rate of 166 MHz, are connected by a 100 Mbs Ethernet, and use SUNOS 5.5.1 with a MPI library (MPICH version 1.0.4 [MPICH 1996]).

7.1 Two-Dimensional Heat Equation

Although optimal data distributions for column sweep and for row sweep are different, all temporal vectors are regular. Thus, even under a static data

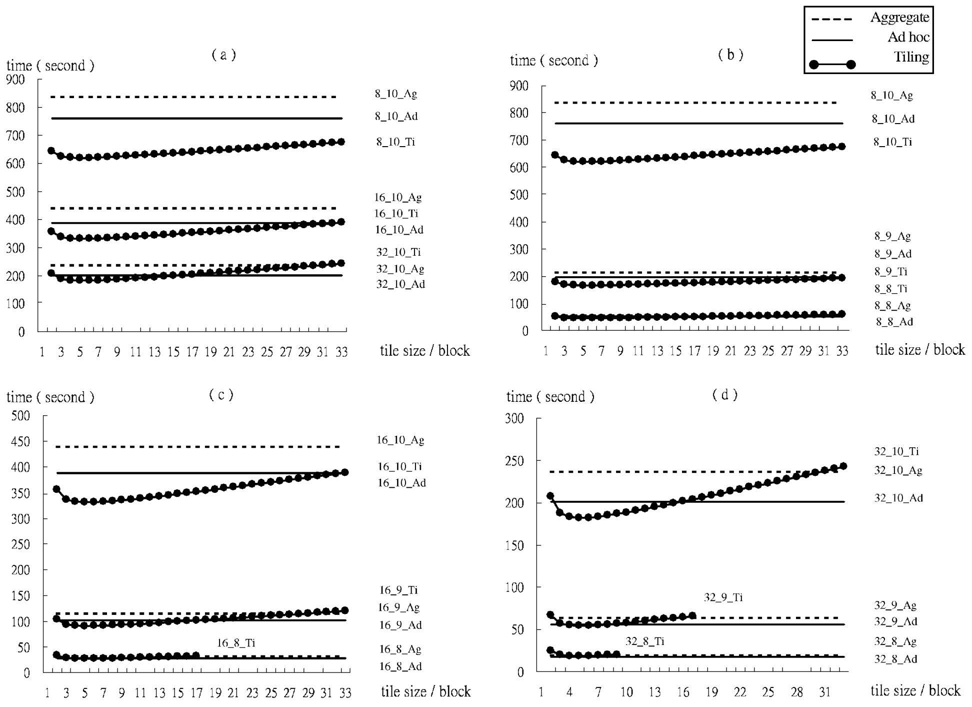


Fig. 12. Execution time of three algorithms for solving the 2D heat equation on the nCUBE-2. (a) Data size is $2^{10} \times 2^{10}$ on 32 PEs, 16 PEs, and 8 PEs. Data sizes are $2^8 \times 2^8$, $2^9 \times 2^9$, and $2^{10} \times 2^{10}$ on (b) 8 PEs, (c) 16 PEs, and (d) 32 PEs.

distribution, we can apply our tiling techniques to improve the execution time. Figure 12 shows the experiments on the nCUBE-2, using three algorithms: a dynamic data-layout algorithm (Aggregate) whose transpose operations are based on aggregate operations [Fox et al. 1988], with a transpose operation taking $\log N$ steps for N PEs; a dynamic data-layout algorithm (Ad-hoc) in which for each transpose operation, each PE sends $N - 1$ messages to other PEs; and a tiling algorithm (Tiling). `x_y_Pq` means the execution on x PEs with data size $2^y \times 2^y$, using the `Pq` algorithm; where `Pq` stands for one of the following: `Ag` (Aggregate), `Ad` (Ad-hoc), or `Ti` (Tiling). When the data size is $2^8 \times 2^8$ ($2^8 \times 2^8$ and $2^9 \times 2^9$, respectively) on 16 PEs (32 PEs and 32PEs, respectively), the maximum `tile size/block` is 16 (8 and 16, respectively). Figure 13 shows the results on the cluster using "Ad-hoc" and "Tiling." When the data size is $2^8 \times 2^8$ ($2^9 \times 2^9$ and $2^{10} \times 2^{10}$, respectively) on 4 PEs, the maximum `tile size/block` is 64 (100 and 100, respectively). Both Figure 12 and Figure 13 show scalability and speedup; the execution time decreases when the number of PEs increases and the execution time increases when the size of the problem increases.

The pure computation times for these three algorithms are basically the same. However, their communication times, which depend on the volume of data transferred, are quite different. Suppose that the data size is $N_x \times N_y$ partitioned among N PEs, with each containing $N_x N_y / N$ data. For "Aggregate," a PE sends $2(\log N) N_x N_y / (2N)$ data to other PEs; the factor 2 is due to one

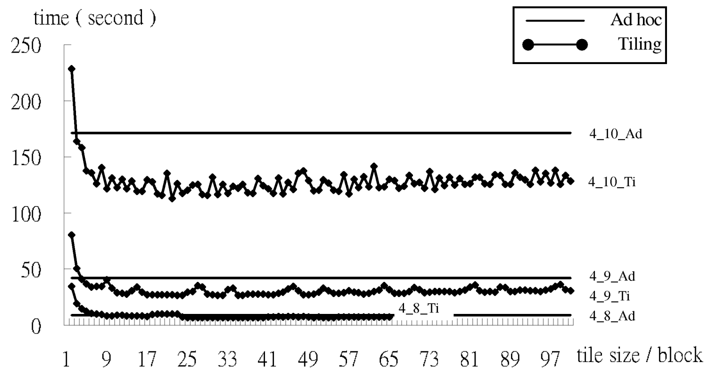


Fig. 13. Execution time of two algorithms for solving the 2D heat equation on the cluster, where data sizes are $2^8 \times 2^8$, $2^9 \times 2^9$, and $2^{10} \times 2^{10}$.

transpose for each of u and v, and each transpose operation takes $\log N$ steps on (the hypercube-based) nCUBE-2. For "Ad-hoc," a PE sends $2(N-1)N_x N_y/N^2$ data to other PEs. For "Tiling," a PE only sends $3N_x$ or $3N_y$ data to other PEs; where the factor 3 is due to solving three first-order recurrence equations for each tridiagonal system. It seems that if a good tile size is chosen, "Tiling" is likely to perform better than the other algorithms.

The experimentally obtained optimal tile sizes Z match well with those derived from the analytical model in Section 6.3. Let `block` $= N_x/N$ or N_y/N. For the nCUBE-2, $t_s = 350\ \mu$s, $t_c = 4.56\ \mu$s, $t_f = 17\ \mu$s for executing an iteration of Loop 2 in Figure 5(d), and $t_f = 8\ \mu$s for executing an iteration of Loop 4 in Figure 5(d). Then for Loop 2, $Z/$`block` $= 4$ or 5, which matches the experimental results. The factor $Z/$`block` for Loop 4 is 6 or 7. For the cluster, $t_s = 819\ \mu$s, $t_c = 0.21\ \mu$s, $t_f = 1.1\ \mu$s for executing an iteration of Loop 2, and $t_f = 0.5\ \mu$s for executing an iteration of Loop 4. Then, for Loop 2, $Z/$`block` $= 32$, which matches the experimental studies. The factor $Z/$`block` for Loop 4 is 46. It becomes clear now that when t_s dominates t_f, we have coarse-grained computation; otherwise, we have a medium- or a fine-grained computation. As the factor t_s accounts for the communication and the factor t_f accounts for the speed of CPUs, this is the expected situation.

7.2 Two-Dimensional Fast Fourier Transform

In this experimental study, we ran a 2D Fast Fourier Transform (FFT) immediately followed by running an inverse 2D FFT using the row-column method for the complex matrix (A_R, A_I), where A_R is the real part and A_I is the imaginary part. The program contains four Do-loops: Loop 1 computes a 1D FFT for each row, Loop 2 computes a 1D FFT for each column, Loop 3 computes an inverse 1D FFT for each column, and Loop 4 computes an inverse 1D FFT for each row. The size of the data will be denoted by m.

In the row sweep, data in different rows are independent; however, there is irregular data dependence along the second dimension of A_R and A_I. Thus, the spatial dependence vectors of A_R and A_I are $s_{A_R} = (0, c_4)$ and $s_{A_I} = (0, c_4)$, respectively. By our method, the data distribution functions for A_R and A_I will be

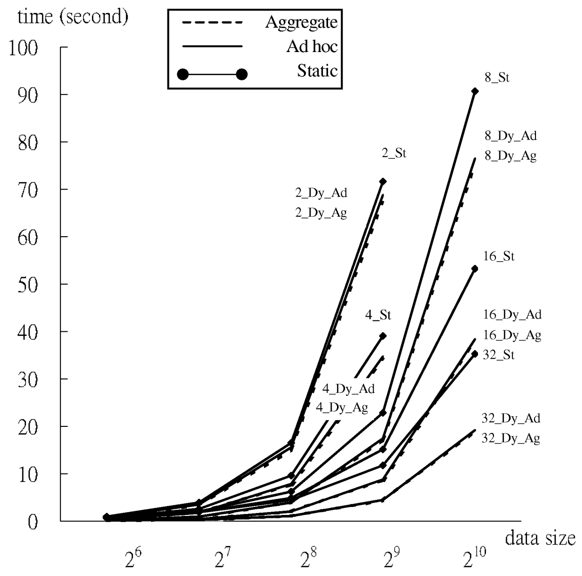


Fig. 14. Execution time of three algorithms for solving the 2D FFT on the nCUBE-2.

A_R(`block`, $\times$) and A_I(`block`, $\times$), respectively. In the column sweep, data among different columns are independent; however, there is irregular data dependence along the first dimension of A_R and A_I. Thus, the spatial dependence vectors of A_R and A_I are $s_{A_R} = (c_4, 0)$ and $s_{A_I} = (c_4, 0)$, respectively. The data distribution functions for A_R and A_I will be A_R($\times$, `block`) and A_I($\times$, `block`), respectively.

When static data distribution scheme which distributes A_R and A_I either row by row or column by column is adopted, communication will be required to execute several "bit-reverse shuffle-exchange" and "butterfly-pattern." As temporal dependence vectors are irregular, we cannot use tiling. Thus, each PE sends data of size $4(\log N)m^2/N$ to other PEs; the factor 4 is due to the "bit-reverse shuffle-exchange" and "butterfly-pattern" data communications for both A_R and A_I. When a dynamic data distribution scheme for the row sweep and for the column sweep is adopted, communication will be required to execute four matrix transposes, with two transposes for both A_R and A_I. If a transpose is implemented using aggregate operations, each PE sends a total of $4(\log N)m^2/(2N)$ data to other PEs; if a transpose is implemented using an ad hoc method, each PE sends a total of $4(N-1)m^2/N^2$ data to other PEs. Figure 14 shows the results for the nCUBE-2 for three algorithms: a static data distribution (Static) and two dynamic data distributions (Aggregate and Ad-hoc). `x_Dy_Pq` stands for using x PEs to run dynamic data-layout algorithm `Pq` (which is `Ag` or `Ad`, as before); `x_St` stands for the static data-layout. For this problem, dynamic data distribution is superior to the static one.

7.3 Gaussian Elimination with Pivoting

In this experimental study, the input is a matrix $A(i, j)$ for $1 \le i, j \le m$; the output includes an upper triangular matrix $A(i, j)$ for $1 \le i \le j \le m$, a lower

Table II. Execution Time (Second) for Solving the Gauss Elimination on the Cluster

A \ b		1	2	4	8	16	32	64	128	256
$2^7 \times 2^7$	1 PE	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	
	2 PE	0.22	0.21	0.21	0.21	0.21	0.22	0.24		
	4 PE	0.20	0.22	0.23	0.23	0.23	0.24			
$2^8 \times 2^8$	1 PE	2.20	2.20	2.20	2.20	2.20	2.20	2.20	2.20	2.20
	2 PE	1.22	1.22	1.23	1.23	1.24	1.28	1.36	1.53	
	4 PE	0.82	0.85	0.87	0.88	0.89	0.90	0.97		
$2^9 \times 2^9$	1 PE	18.3	18.3	18.3	18.3	18.3	18.3	18.3	18.3	18.3
	2 PE	9.20	9.15	9.19	9.26	9.32	9.35	9.59	10.27	11.68
	4 PE	4.78	4.98	5.06	5.12	5.15	5.26	5.66	6.40	
$2^{10} \times 2^{10}$	1 PE	145.4	145.4	145.4	145.4	145.4	145.4	145.4	145.4	145.4
	2 PE	71.0	71.0	72.0	72.7	73.1	73.3	74.2	76.6	83.8
	4 PE	35.1	36.2	36.8	37.2	37.8	37.9	39.5	43.1	49.6

Note: Matrix A is distributed along the second dimension by a `cyclic`(b) distribution, data sizes are $2^7 \times 2^7$, $2^8 \times 2^8$, $2^9 \times 2^9$, and $2^{10} \times 2^{10}$, respectively.

triangular matrix for multipliers $A(i, j)$ for $1 \leq j < i \leq m$, and a permutation array `ipvt`(i) for $1 \leq i \leq m$, as shown in Figure 10. Applying our algorithm, matrix A is distributed along the second dimension by a `cyclic`(b) distribution, say $A(\times, \texttt{cyclic}(b))$. Table II lists the results on the cluster when data sizes are $2^7 \times 2^7$, $2^8 \times 2^8$, $2^9 \times 2^9$, and $2^{10} \times 2^{10}$, respectively. It shows scalability and speedup; the execution time decreases when the number of PEs increases, and the execution time increases when the size of the problem increases. It is instructive to mention that because the set-up time t_s of a message is much larger than the execution time t_f for performing an iteration, when data size is small, this is a communication-bound problem. Therefore, load balancing is not a sensitive factor. However, when data size is large, this becomes a computation-bound problem, and therefore load balancing is a crucial factor for gaining performance. Because the iteration space is a pyramid, when block sizes are $b = 1$ or $b = 2$, the execution time is minimized, as expected.

8. CONCLUSIONS

Experienced programmers write scientific application programs following a good programming style. For example, while writing a program for the 2D heat equation, they write a column sweep then followed by a row sweep. Therefore, data references exhibit localities and fixed patterns. Since optimizing data distribution and maintaining locality are crucial for performance on DMPCs, we introduced the concept of the dominant array to account for this. A dominant array was one whose migration would be very expensive and therefore minimized by appropriate data distribution, with other data arrays aligned with it as appropriate and feasible. As in different program fragments optimal data alignments may be different, we proposed an algorithm to decide whether consecutive program fragments should share the same data alignment.

While determining a static data distribution scheme within one program fragment, which may include several Do-loops, we proposed to use spatial dependence/use vectors—another concept we introduced—to help determine which dimensions of the dominant data array is better not to distribute. Spatial dependence/use vectors independently represent a superset of irregular temporal dependence/use relations, and thus they implicitly help determine mapping of data with irregular data dependence relations into a fixed PE. We then used regular temporal dependence/use vectors to determine whether the remaining dimensions of the dominant data array should be distributed or not. For this, we examined one by one the Do-loops that involve the dominant data array, starting from the most computationally intensive Do-loop. We found correspondences between iteration space mapping vectors and distributed dimensions of the dominant data array in each nested Do-loop, which allowed us to design algorithms for determining data and computation decompositions at the same time.

After data distributions are determined, computation decomposition for each nested Do-loop is determined based on either the owner computes rule or the owner stores rule with respect to the dominant data array. If all temporal dependence relations across iteration partitions are regular, tiling techniques are used to allow pipelining and overlapping of the computation and the communication. However, tiling the iteration space must account for data distributions, as otherwise communication costs will be incurred due to data redistribution. We have proposed algorithms to determine tiling vectors and constraints of tile sizes for arbitrary nested Do-loops and to determine optimal tile sizes for the depth-two nested Do-loops.

APPENDIX

A. DETERMINING AXIS ALIGNMENT

For completeness, in the following, we describe how to construct component affinity graphs and how to determine axis alignment based on the component alignment algorithm, as presented in Lee [1997]. As is shown in Step 1 of the proposed method in Section 3, we apply the loop fission technique so that the original program is more suitable for parallel execution and we can execute nested Do-loops in sequence.

For each nested Do-loop, we construct a *component affinity graph*. The *composite* component affinity graph for a sequence of consecutive nested Do-loops is the union of the graphs for individual nested Do-loops. If an iterative Do-loop contains j nested Do-loops, the component affinity graph for this iterative Do-loop is identical to the composite graph of the j inner nested Do-loops, except that the weight of each edge becomes m times of the original one, where m is the problem size of the iterative Do-loop.

We now describe how given a nested Do-loop, we construct a component affinity graph for it. The graph is undirected and weighted. Its nodes represent dimensions (components) of arrays and its edges specify affinity relations between nodes. Edges are defined in two ways. First, if the subscripts of the dimensions of the dominant data array in that nested Do-loop have affinity relations with the subscripts of the dimensions of other arrays generated or used in that nested Do-loop, then there are edges between corresponding pairs of dimensions. We need these edges, because later iteration partitioning due to

```
A heuristic component alignment algorithm:
Step 1: construct a component affinity graph from the source program;
Step 2: choose a (high-dimensional) array with a highest dimensionality;
    thus, this array has the maximum number of nodes in the graph, and let
    its corresponding nodes in the graph become the initial basic set;
Step 3: while the remaining graph is not empty, do
    Step 3.1: choose an array with highest dimensionality from the remaining
        graph;
    Step 3.2: apply the optimal matching procedure to a bipartite graph
        constructed from the basic set and the nodes corresponding to
        dimensions of the newly selected array;
        /* All disjointed subsets of matched nodes represent a partition. */
    Step 3.3: combine the matched nodes with the basic set as a new basic set.
```

Fig. 15. Heuristic component alignment algorithm.

computation decomposition is based on the data distribution of the dominant data array. It is advantageous, to align other data arrays with this dominant data array. Second, if two right-hand-side arrays correspond to the two operands of a binary operator, and if some pairs of subscripts of dimensions of these two arrays have affinity relations, then there are edges between corresponding pairs of dimensions of these two arrays. It is advantageous for these two operands to be aligned. We use the higher ranked data array to represent an intermediate result of the operation for considering alignments with the operands of other binary operations.

The weight of an edge is an estimate of the communication that is required if dimensions of two arrays are distributed along different dimensions of P. The component alignment problem is defined as an optimal partitioning of the node set of the component affinity graph into k disjointed subsets, where k is the dimension of the highest dimensional data array. The objective is to minimize the total weight of the edges across nodes in different subsets, under the constraint that no two nodes corresponding to the same array are in the same subset.

Although the component alignment problem is NP-hard, Li and Chen have proposed an efficient heuristic algorithm [Li and Chen 1991b], which we adopt. For completeness, in Figure 15, we present a very brief version of the component alignment algorithm; for fuller details, see Li and Chen [1991b]. Array dimensions within each of the above mentioned k disjointed subsets are then aligned together. Data distributions of these array dimensions will share the same pattern, as discussed in Section 5.

ACKNOWLEDGMENTS

The authors would like to thank Prof. S. Chatterjee and the anonymous referees for their valuable comments on this article.

REFERENCES

ADVE, V., JIN, G., MELLOR-CRUMMEY, J., AND YI, Q. 1998. High performance Fortran compilation techniques for parallelizing scientific codes. In *Proceedings of Supercomputing '98* (Orlando, Fla.).

AGARWAL, A., KRANZ, D., AND NATARAJAN, V. 1993. Automatic partitioning of parallel loops for cache-coherent multiprocessors. In *Proceedings of the International Conference on Parallel Processing* (St. Charles, Ill.). I–2–11.

AGARWAL, A., KRANZ, D., AND NATARAJAN, V. 1995. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Paral. Distrib. Syst. 6*, 9 (Sept.), 943–962.

ALLEN, J. R. AND KENNEDY, K. 1987. Automatic translation of Fortran programs to vector form. *ACM Trans. Program. Lang. Syst. 9*, 4 (Oct.), 491–542.

ANDERSON, J. 1997. Automatic computation and data decomposition for multiprocessors. Ph.D. dissertation. Dept. of EE and CS, Stanford Univ., Stanford, Calif.

BARUA, R., KRANZ, D., AND AGARWAL, A. 1996. Communication-minimal partitioning of parallel loops and data arrays for cache-coherent distributed-memory multiprocessors. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing* (Berlin, Germany). Lecture Notes in Computer Science, vol. 1239. Springer-Verlag, New York, pp. 350–368.

BOULET, P., DARTE, A., RISSET, T., AND ROBERT, Y. 1994. (Pen)-ultimate tiling? *Integration, the VLSI Journal 17*, 33–51.

CALLAHAN, D. AND KENNEDY, K. 1988. Compiling programs for distributed-memory multiprocessors. *J. Supercomput. 2*, 151–169.

CHATTERJEE, S., GILBERT, J. R., SCHREIBER, R., AND SHEFFLER, T. J. 1994a. Array distribution in data-parallel programs. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 892. Springer-Verlag, New York, pp. 76–91.

CHATTERJEE, S., GILBERT, J. R., SCHREIBER, R., AND SHEFFLER, T. J. 1994b. Modelling data-parallel programs with the alignment-distribution graph. *J. Prog. Lang. 2*, 227–258.

CHATTERJEE, S., GILBERT, J. R., SCHREIBER, R., AND TENG, S. H. 1993. Automatic array alignment in data-parallel programs. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Charleston, S.C.). ACM, New York.

CHEN, M. C. 1988. The generation of a class of multipliers: Synthesizing highly parallel algorithms in VLSI. *IEEE Trans. Comput. C-37*, 3 (Mar.), 329–338.

CHEN, T. AND SHEU, J. 1994. Communication-free data allocation techniques for parallelizing compilers on multicomputers. *IEEE Trans. Paral. Distrib. Syst. 5*, 9 (Sept.), 924–938.

COUVERTIER-REYES, I. 1996. Automatic data and computation mapping for distributed memory machines. Ph.D. dissertation, Louisiana State University, Baton Rouge, La.

DESPREX, F., DONGARRA, J., RASTELLO, F., AND ROBERT, Y. 1998. Determining the idle time of a tiling: New results. *J. Inf. Sci. Eng. 14*, 1 (Mar.), 167–190.

FOX, G. C., JOHNSON, M. A., LYZENGA, G. A., OTTO, S. W., SALMON, J. K., AND WALKER, D. W. 1988. *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems*. Prentice-Hall, Englewood Cliffs, N.J.

GUPTA, M. 1997. On privatization of variables for data-parallel execution. In *Proceedings of the International Parallel Processing Symposium*. 533–541.

GUPTA, M. AND BANERJEE, P. 1992a. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Paral. Distrib. Syst. 3*, 2 (Mar.), 179–193.

GUPTA, M. AND BANERJEE, P. 1992b. A methodology for high-level synthesis of communication on multicomputers. In *Proceedings of the ACM International Conference on Supercomputing* (Washington D.C.). ACM, New York.

GUPTA, M. AND BANERJEE, P. 1994. Compile-time estimation of communication costs of programs. *J. Prog. Lang. 2*, 191–225.

GUPTA, S. K. S. AND KRISHNAMURTHY, S. 1998. An interprocedural framework for determining efficient array data redistributions. *J. Inf. Sci. Eng. 14*, 1 (Mar.), 27–51.

HIRANANDANI, S., KENNEDY, K., AND TSENG, C.-W. 1992. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM 35*, 8 (Aug.), 66–80.

HO, C. T. 1990. Optimal communication primitives and graph embeddings on hypercubes. Ph.D. dissertation, Yale Univ.

HODZIC, E. AND SHANG, W. 1998. On supernode transformation with minimized total running time. *IEEE Trans. Paral. Distrib. Syst. 9*, 5 (May), 417–428.

HOGSTEDT, K., CARTER, L., AND FERRANTE, J. 1999. Selecting tile shape for minimal execution time. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (Saint Malo, France). ACM, New York, pp. 201–211.

HUANG, C. H. AND LENGAUER, C. 1987. The derivation of systolic implementations of programs. *Acta Inf. 24*, 595–632.

HUANG, C. H. AND SADAYAPPAN, P. 1993. Communication-free hyperplane partitioning of nested loops. *J. Paral. Distrb. Comput. 19*, 90–102.

HWANG, K. 1993. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., New York.

IRIGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Diego, Calif.). ACM, New York, pp. 319–329.

KANDEMIR, M. AND RAMANUJAM, J. 2000. Data relation vectors: A new abstraction for data optimizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (*PACT'00*) (Philadelphia, Pa).

KENNEDY, K. AND KREMER, U. 1998. Automatic data layout for distributed-memory machines. *ACM Trans. Program. Lang. Syst. 20*, 4 (July), 869–916.

KOELBEL, C., LOVEMAN, D., SCHREIBER, R., STEELE, G., JR., AND ZOSEL, M. 1994. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA.

KREMER, U. 1995. Automatic data layout for distributed memory machines. Ph.D. dissertation, Rice Univ., Houston, Tex.

KREMER, U. 1998. Fortran RED—A retargetable environment for automatic data layout. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 1656. Springer-Verlag, New York, pp. 148–165.

KUNG, H. T. AND LEISERSON, C. E. 1980. Chapter 8.3, Algorithms for VLSI processor arrays. In *Introduction to VLSI Systems*. C. Mead and L. Conway, Eds. Addison-Wesley, Reading, Mass.

LAM, M. S. 1987. A systolic array optimizing compiler. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, Pa.

LEE, P.-Z. 1995a. Parallel matrix multiplication algorithms on hypercube multicomputers. *Int. J. High-Speed Comput. 7*, 3 (Sept.), 391–406.

LEE, P.-Z. 1995b. Techniques for compiling programs on distributed memory multicomputers. *Paral. Comput. 21*, 12, 1895–1923.

LEE, P.-Z. 1997. Efficient algorithms for data distribution on distributed memory parallel computers. *IEEE Trans. Paral. Distrib. Syst. 8*, 8 (Aug.), 825–839.

LEE, P.-Z. AND CHEN, W. Y. 1997. Generating global name-space communication sets for array assignment statements. Tech. Rep. TR-IIS-97-016, Institute of Information Science, Academia Sinica, Taipei, Taiwan, available via WWW at http://www.iis.sinica.edu.tw/~leepe/PAPER/tr97016.ps.

LEE, P.-Z. AND KEDEM, Z. M. 1988. Synthesizing linear-array algorithms from nested For loop algorithms. *IEEE Trans. Comput. C-37*, 1578–1598.

LEE, P.-Z. AND KEDEM, Z. M. 1990a. Mapping nested loop algorithms into multi-dimensional systolic arrays. *IEEE Trans. Paral. Distrib. Syst. 1*, 64–76.

LEE, P.-Z. AND KEDEM, Z. M. 1990b. On high-speed computing with a programmable linear array. *J. Supercomput. 4*, 3 (Sept.), 223–249.

LI, J. AND CHEN, M. 1990. Generating explicit communication from shared-memory program references. In *Proceedings of the Supercomputing '90*. 865–876.

LI, J. AND CHEN, M. 1991a. Compiling communication-efficient problems for massively parallel machines. *IEEE Trans. Paral. Distrib. Syst. 2*, 3 (July), 361–376.

LI, J. AND CHEN, M. 1991b. The data alignment phase in compiling programs for distributed-memory machines. *J Paral. Distrib. Comput. 13*, 213–221.

LIM, A. W., CHEONG, G. I., AND LAM, M. S. 1999. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the ACM International Conference on Supercomputing* (Rhodes, Greece).

LIM, A. W. AND LAM, M. S. 1994. Communication-free parallelization via affine transformations. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing* (Ithaca, N.Y.). Lecture Notes in Computer Science, vol. 892. Springer-Verlag, New York.

LIM, A. W. AND LAM, M. S. 1998. Maximizing parallelism and minimizing synchronization with affine partitions. *Paral. Comput. 24*, 445–475.

MCKINLEY, K. S., CARR, S., AND TSENG, C.-W. 1996. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst. 18*, 4 (July), 424–453.

MOLDOVAN, D. I. AND FORTES, J. A. B. 1986. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Comput. C-35*, 1–12.

MPICH. 1996. MPICH, a portable implementation of MPI. Tech. rep., Argonne National Labortory and University of Chicago.

NING, Q., DONGEN, V. V., AND GAO, G. R. 1995. Automatic data and computation decomposition for distributed memory machines. *Paral. Proc. Lett. 5*, 4, 539–550.

PALERMO, D. J. 1996. Compiler techniques for optimizing communication and data distribution for distributed-memory multicomputers. Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, Urbana, Illinois.

POLYCHRONOPOULOS, C. 1988. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Trans. Comput. C-37*, 8 (Aug.), 991–1004.

RAMANUJAM, J. AND SADAYAPPAN, P. 1991. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Paral. Distrib. Syst. 2*, 4 (Oct.), 472–482.

RAMANUJAM, J. AND SADAYAPPAN, P. 1992. Tiling multidimensional iteration spaces for multicomputers. *J. Paral. Distrib. Comput. 16*, 108–120.

RIBAS, H. B. 1990. Automatic generation of systolic programs from nested loops. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, Pa.

ROGERS, A. AND PINGALI, K. 1994. Compiling for distributed memory architectures. *IEEE Trans. Paral. Distrib. Syst. 5*, 3 (Mar.), 281–298.

SHANG, W. AND FORTES, J. A. B. 1992. On time mapping of uniform dependence algorithms into lower dimensional processor arrays. *IEEE Trans. Paral. Distrib. Syst. 3*, 3 (May), 350–363.

SHIH, K.-P., SHEU, J.-P., AND HUANG, C.-H. 1996. Statement-level communication-free partitioning techniques for parallelizing compilers. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, D. Sehr, U. Banerjee, D. Gelernter, A. Nicolu, and D. Padua, Eds. Lecture Notes in Computer Science, vol. 1239. Springer-Verlag, New York. pp. 389–403.

STRIKWERDA, J. C. 1989. Chapter 7.3 The Alternating Direction Implicit (ADI) Method. In *Finite Difference Schemes and Partial Differential Equations*. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, Calif., pp. 142–153.

TSENG, P. S. 1990. *A Systolic Array Parallelizing Compiler*. Kluwer Academic Publishers, Boston, Mass.

WOLF, M. E. AND LAM, M. S. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Paral. Distrib. Syst. 2*, 4 (Oct.), 452–471.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA.

WU, J.-J. 1995. Optimization and transformation techniques for high performance fortran. Ph.D. disseration, Yale Univ.

WU, J., DAS, R., SALTZ, J., BERRYMAN, H., AND HIRANDAN, S. 1995. Distributed memory compiler design for sparse problems. *IEEE Trans. Comput. 44*, 6 (June), 737–753.

XUE, J. 1997. Communication-minimal tiling of uniform dependence loops. *J. Paral. Distrib. Comput. 42*, 42–59.

ZIMA, H. AND CHAPMAN, B. 1990. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, Redwood City, CA.

ZIMA, H. AND CHAPMAN, B. 1993. Compiling for distributed-memory systems. *Proc. IEEE 81*, 2 (Feb.), 264–287.

Received April 1999; revised January and November 2000; accepted August 2001