Automatic Data and Computation Decomposition on Distributed Memory Parallel Computers[1]

PeiZong Lee
Institute of Information Science
Academia Sinica
Taipei, Taiwan, R.O.C.
Internet: leepe@iis.sinica.edu.tw
TEL: +886 (2) 2788-3799
FAX: +886 (2) 2782-4814

Zvi M. Kedem
Dept. of Computer Science
New York University
New York, NY, USA
Internet: kedem@cs.nyu.edu
TEL: +1 (212) 998-3101
FAX: +1 (212) 995-4123

Abstract

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 optimize the overall execution time. The method relies on a number of novel techniques, also presented in this paper. The core idea is to rank the "importance" of data arrays in a program and define 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, we are able 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 overlapping the computation and communication time. However, 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 method is illustrated on programs for the 2D heat equation and for the 2D fast Fourier transform both on a linear processor array.

Keywords: computation decomposition, data alignment, data distribution, distributed-memory computers, iteration space mapping vector, parallelizing compilers, spatial dependence vector, temporal dependence vector, tiling techniques.

[1]This work was partially supported by the NSC under Grant NSC 88-2213-E-001-008, 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 the second author was visiting the Center for Applied Sciences and Engineering Research and the Institute of Information Science, Academia Sinica, Nankang, Taiwan, July – August, 1998.

1 Introduction

Distributed memory parallel computers (DMPCs) have been playing an important role in solving computation-intensive problems, as they are relatively easily scalable, so that given a large number of processing elements (PEs), they are suited for solving large problems—such as Grand Challenge Problems [17]. However, program development form DMPCs is time-consuming and error-prone, as the programmer is forced to manage both parallelism and communication [5, 41, 49]. The tools generally used for these are the decomposition of computation and the decomposition of data. Our key contribution is a set of integrated techniques of simultaneously producing decomposition for computation and data, focusing on data distribution first, and specifying computation decomposition based on it.

In this Introduction, we start by briefly reviewing some relevant previous work and then providing an overview for our techniques. Early pioneering work dealt with mapping 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 [6, 15, 22, 23, 28, 29, 30, 34, 40, 42, 45].

As in general, the number of iterations of 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 a cyclic dependency among (the iterations in) the tiles. In [18], a sufficient condition for existence of tiles without size restriction was presented. Others concentrated on finding tiles with size restriction to minimize execution time [4, 9, 14, 39, 46, 48]. 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, e.g., 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, there exists a communication-free computation decomposition, whose partitioning hyperplanes are perpendicular to a basis of that null space, if read-only data can be replicated [7]. 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 [3, 16, 35, 38, 43, 46]. In [8], additional methods were proposed for determining non-trivial 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. Which implies that data arrays are not distributed independently along each dimension; for example, data arrays are stored among PEs in a skewed manner.

However, as mentioned above, data distributions may be ignored on shared memory model, but they are a crucial factor to gain performance on DMPCs. 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 [20], communication-free approaches only can be adopted with additional restrictions on DMPCs. All systolic algorithm approach, tiling approach, and communication-free approach 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 [11, 25, 32], became prominent. Data redistribution between program fragments can also be optimized by comparing the relative costs of different data distribution schemes [26]. For other approaches see [19, 21, 36]. For a complete survey of other data distribution techniques see [26].

In general, component alignment approaches are very promising for DMPCs, because dimensions on each data array can be distributed independently among one another, 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. This is the focus of our paper.

We will 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 possibly includes *several* nested Do-loops. Therefore, they are useful for determining data distributions for that program fragment. We will show how to integrate data alignment techniques and iteration space tiling techniques for optimizing both data and computation decompositions.

Our approach is different from previous work, which focus on determining the computation decomposition for a single nested Do-loop first, and thus implying a corresponding data decomposition for the data used in that nested Do-loop. Therefore, for different nested Do-loops, data distributions may be different, 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*. By focusing on such dominant arrays, we are able to produce novel techniques. Dominant arrays are those, which we do not want to migrate during the computation. We then establish correspondence between iteration space mapping vectors and distributed dimensions of the dominant data array in each nested Do-loop. We can thus design algorithms for determining data and computation decompositions at the same time. When 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 paper is organized as follows. Section 2 presents necessary definitions, models, assumptions, and background materials. Section 3 presents an overview of our proposed method. Section 4 demonstrates our methodology by analyzing different data distributions for the two-dimensional heat equation. Section 5 proposes algorithms to determine data and computation decompositions all at once. Section 6 illustrates our tiling techniques on DMPCs. Section 7 presents experimental studies on a 32-node nCUBE-2 computer and on four workstations connected by a fast Ethernet. Finally, some concluding remarks are given in Section 8.

2 Definitions, models, assumptions, and background materials

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 [13].

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 [11, 12, 31]. 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 will use the term "array" to stand for any dimensional array, including a 1-D array (vector) or 2-D array (matrix).

Subscripts

In this paper, we will 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.

2.1 Data distribution

Block, cyclic, and cyclic(b) data distributions

`cyclic`(b) distribution is the most general regular distribution, in which blocks of size b of a 1-D data array are distributed among the PEs of a 1-D PE array in a round-robin fashion. For example, let array $A(l : u)$ be indexed from l to u, where A is a 1-D array; or, in general, some specific dimension of a high-dimensional array. We will 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 $\mathtt{PE}_p$. Thus, the xth entry of A is stored in $\mathtt{PE}_p$, where $p = \lfloor (x-l)/b \rfloor \bmod N$. We will 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 focus on assigning elements of a k-D data array A to the elements of a g-D grid P. Since data distributions for different dimensions of A are independent, we can 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 $\mathtt{cyclic}(\mathtt{db}_i)$ along P_j, if and only if there exists a function f_{A_i} of the form

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

   where $\mathtt{doffset}_i$ is an offset, such that if $(a_1, ..., a_i, ..., a_k)$ is assigned to $(p_1, ..., p_j, ..., p_g)$, 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 $(p_1, ..., p_{j-1}, p_j, p_{j+1}, ..., p_g)$ and $(p_1, ..., p_{j-1}, p'_j, p_{j+1}, ..., p_g)$ 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 $(p_1, ..., p_{j-1}, p_j, p_{j+1}, ..., p_g)$, 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:

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

where $\mathtt{map}()$ is a one-to-one function, $1 \leq \mathtt{map}(A_i) \leq g$, and $f_{A_i}(x)$ returns that PE index along the dimension $\mathtt{map}(A_i)$ 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. In the sequel, we will also use "R" to indicate replication and "$\times$" to indicate non distribution.

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 $\mathtt{PE}_A$ of length g is defined by (j is the position in the vector):

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

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 will ignore the data-matching vector when there is no risk of confusion, or when P is a 1-D PE array.

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)

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

In order to specify the relation between the dimensions of the data space and the dimensions of the iteration space as depicted in Equation (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 $\mathtt{DT}_{g \times k}$, such that

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

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 convenience we use "$\circ$" as an operator for matrix/vector multiplication. If $A_{\alpha(\gamma)}$ is either distributed, replicated, or fixed along the γth dimension of the PE grid; then the γth row of $\mathtt{DT}$ is an elementary vector $\tilde{e}_{\alpha(\gamma)}$ with a functional operator $f_{A_{\alpha(\gamma)}}(a_{\alpha(\gamma)})$ in position $\alpha(\gamma)$, such that the γth row has $f_{A_{\alpha(\gamma)}}(a_{\alpha(\gamma)})$ in position $\alpha(\gamma)$ and has 0's in other positions. Then, we have $f_{A_{\alpha(\gamma)}}(a_{\alpha(\gamma)}) = p_\gamma$ or R or c. We will, however, ignore the relationship when no dimension of A is distributed along the γth dimension of the PE grid.

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

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

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 at 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 will try to minimize migration of data by finding those data arrays that are accessed most often (later referred to as "dominant") and try to assign them for as large fragments of computation as possible 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; and 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 their 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.

Axis alignment

The *axis alignment* technique has been introduced in [32], and further developed in e.g., [11, 26]. We briefly describe it here, 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 two subscripts of these two dimensions are affine functions of the same (single) loop control index variable of a Do-loop. It is better for these two dimensions of the two arrays to be aligned with each other, thus avoiding 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 (single) 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: $q(\texttt{block}, \times)$ and $u(\times, \texttt{block})$. It is easily seen, that during the computation, communication is required only for accessing 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: $q(\texttt{block}, \times)$ and $u(\texttt{block}, \times)$. Also, it is easily seen, that during the computation, data re-organization accesses among PEs are needed for performing a transpose operation.

In 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

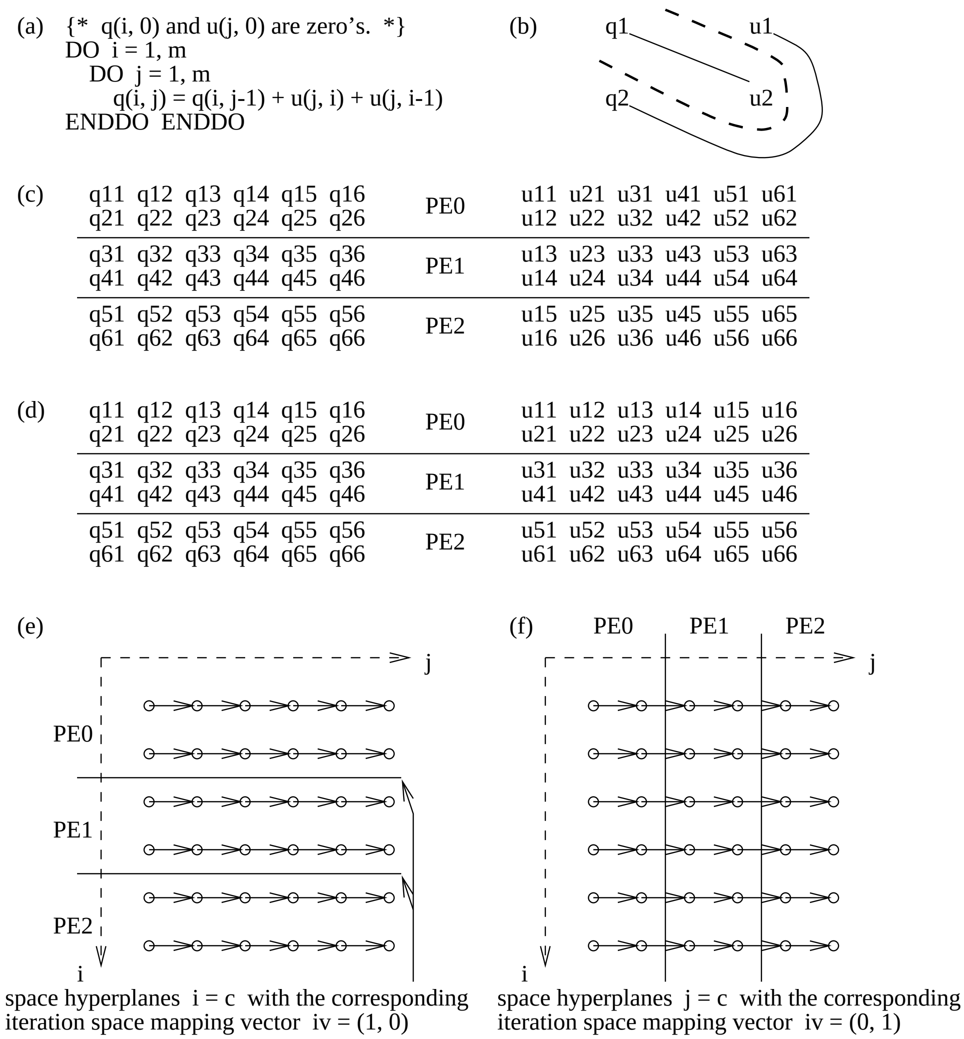


Figure 2: (a) A depth-two nested Do-loop, (b) component affinity graph representing alignment relations among dimensions of arrays q and u. When the problem size $\mathrm{m} = 6$ and the number of PEs $N = 3$, data layouts of arrays q and u under data distribution schema: (c) $q(\texttt{block}, \times)$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

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; and for this see [11, 26, 32]. The bold, dashed line partitions array dimensions into two groups according to 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 among iteration space and 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 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 many 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 the other 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 will use d_u to represent a temporal dependence vector and $d_u^{\mathtt{r}}$ to represent a temporal use vector both for array u. (Superscript "$\mathtt{r}$" stands for "read".) 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 $d_u^{\mathtt{r}} = (1, 0)$, both in the iteration space $\mathcal{I}^{(2)}$.

In this paper, we only consider nested Do-loops with constant (that is regular) temporal vectors, known as uniform dependence algorithms [29, 42, 48]. For cases when some temporal vector d_v is not constant (that is irregular), 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; or, we have to transform the original program to an equivalent one which contains only regular temporal vectors. These techniques, however, are beyond the scope of this papers.

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 will associate with u the spatial vector $(0,1)$. This indicates that for the same value of the first dimension, several elements 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 temporal vectors are irregular. For example, the pair $\langle u(j,i), u(i,i)\rangle$, arising from Gaussian elimination with partial pivot, induces irregular temporal vectors which cannot be represented by a constant number of temporal vectors. But it is easy to use one spatial vector $(1,0)$ to indicate that it is better not to distribute array u along the first dimension in order to avoid communication overhead due to irregular data accesses.

Formally, each pair of occurrences of the same data array defines a spatial vector. If two subscripts of the same ith dimension of the pair of occurrences are the same, then the value in position i of the spatial vector is 0; otherwise, it is 1. We will 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); it *induces a spatial use vector* if both occurrences are on the RHS. We will use s_u to represent a spatial dependence vector and $s_u^{\mathtt{r}}$ 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 $s_q = (0,1)$ for array q and the pair $\langle u(j,i), u(j,i-1)\rangle$ induces one spatial use vector $s_u^{\mathtt{r}} = (0,1)$ for array u.

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

(b)
```
DO  i = 1, m
  DO  j = 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
```

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

The respective roles of temporal and spatial dependence 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 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 that 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 $d_A = \{(0,\gamma,0),(1,-\delta,0)\}$, where $0 \le \gamma,\ \delta < m$; and also induces one spatial dependence vector $s_A = (1,0)$. 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 third dimension among PEs. As values in the third dimension of all temporal dependence vectors are all zeros, there is no dependence relation along the third 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 is no temporal dependence relation within this program fragment; however, there is a spatial dependence vector $s_A = (1,0)$ due to the pair $\langle A(m,i), A(i,i)\rangle$. Of course, computation decompositions for these two Do-loops *cannot* be determined based on the non-existent temporal dependence relation. However, based on the spatial dependence vector $s_A = (1,0)$, we *can* decide to distribute array A along its second dimension among PEs. Then, based on the owner computes rule, the iteration space of the first Do-loop is decomposed along its second dimension.

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 $\texttt{cyclic}(b)$, 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

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

where $\texttt{ioffset}_j$ is an offset, $\texttt{map}()$ is a one-to-one function, $1 \le \texttt{map}(I_j) \le g$, and $f_{I_j}(y)$ returns the PE index along the dimension $\texttt{map}(I_j)$ 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 Equation (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 $\texttt{IT}_{g\times n}$, such that

$$\texttt{IT} \circ (i_1, i_2, \ldots, i_n)^{\mathrm{T}} = (p_1, p2, \ldots, p_g)^{\mathrm{T}}, \tag{2}$$

where $(i_1, i_2, \ldots, i_n)$ is an index of an iteration of the depth-n nested Do-loop, $(p_1, p2, \ldots, p_g)$ is an index of a PE on the g-D grid, and, as before, "$\circ$" is an operator for matrix/vector multiplication. If $I_{\beta(\gamma)}$ is either distributed or replicated or fixed along the γth dimension of the PE grid, then the γth row of $\mathtt{IT}$ is an elementary vector $\tilde{e}_{\beta(\gamma)}$ with a functional operator $f_{I_{\beta(\gamma)}}(i_{\beta(\gamma)})$ in position $\beta(\gamma)$, such that the γth row has $f_{I_{\beta(\gamma)}}(i_{\beta(\gamma)})$ in position $\beta(\gamma)$ and has 0's in other positions. Then, we have $f_{I_{\beta(\gamma)}}(i_{\beta(\gamma)}) = p_\gamma$ or R or c. We will ignore the relationship when no dimension of $\mathcal{I}^{(n)}$ is distributed along the γth 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 α is an affine function of some single loop control index variable $i_{x(\alpha)}$. From Equations (1) and (2), we want to match both data and computation decompositions, such that

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

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

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 only involve 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 $u(\texttt{block}, \times)$, $\texttt{DT}_{1\times 2} = (\lfloor(\texttt{dpar})/2\rfloor, 0)$ and $\texttt{IT}_{1\times 2} = (0, \lfloor(\texttt{ipar})/2\rfloor)$ 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 will use the elementary vector $e^n_{x(\alpha(\gamma))}$, which has 1 in position $x(\alpha(\gamma))$ and has 0's in other positions, to represent $\tilde{e}^n_{x(\alpha(\gamma))}$ with a functional operator $f_{I_{x(\alpha(\gamma))}}(i_{x(\alpha(\gamma))})$ in position $x(\alpha(\gamma))$. Therefore, we will say that we want to find g *iteration space mapping vectors*, which are g elementary vectors corresponding to the g rows in $\texttt{IT}_{g\times n}$. 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 $\texttt{iv} = e_1 = (1, 0)$. Figure 2-(f) shows that the iteration space is partitioned by the iteration space mapping hyperplanes $j = c$, whose normal vector is $\texttt{iv} = e_2 = (0, 1)$.

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 will 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 based on 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 whatever 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 maybe 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 $(1, 0)$. That means, 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, $\texttt{PE}_p$ has to wait for data generated by its neighboring $\texttt{PE}_{p-1}$, 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, whatever we use the owner computes rule or the owner stores rule, significant communication cannot be avoided.

3 An overview of the proposed method

We briefly sketch our method. We have shown in Section 2.3 that 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.

Step 1: We apply the loop fission techniques according to the data dependence graph among statements [2], to make the original program more amenable to parallel execution.

Step 2: We construct a component affinity graph for each Do-loop, then we apply the dynamic programming algorithm for axis alignments to decide whether data redistribution is needed between adjacent program fragments [26]. 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 (excluding privatization arrays) based on finding iteration space mapping vectors from some of the most computation-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 will be given in Section 5, while an example is presented in Section 4.)

Step 4: While performing 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 temporal dependences, we find tiling vectors and determine tile sizes so that iterations can be executed in a coarse-grain pipelining fashion. Otherwise, it is a communication-free computation decomposition, provided that we can replicate the required remote read-only data. (A detailed algorithm will be 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 [2]. 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 this paper, 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.

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 will use a specific example: solving the 2D heat equation on a linear processor array with N PEs. Consider the program in Figure 4-(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 [44]. 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 4-(b) shows the structure of Do-loops included in statements from lines 13 through 38 and Figure 4-(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 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 4-(d).

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

(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
```

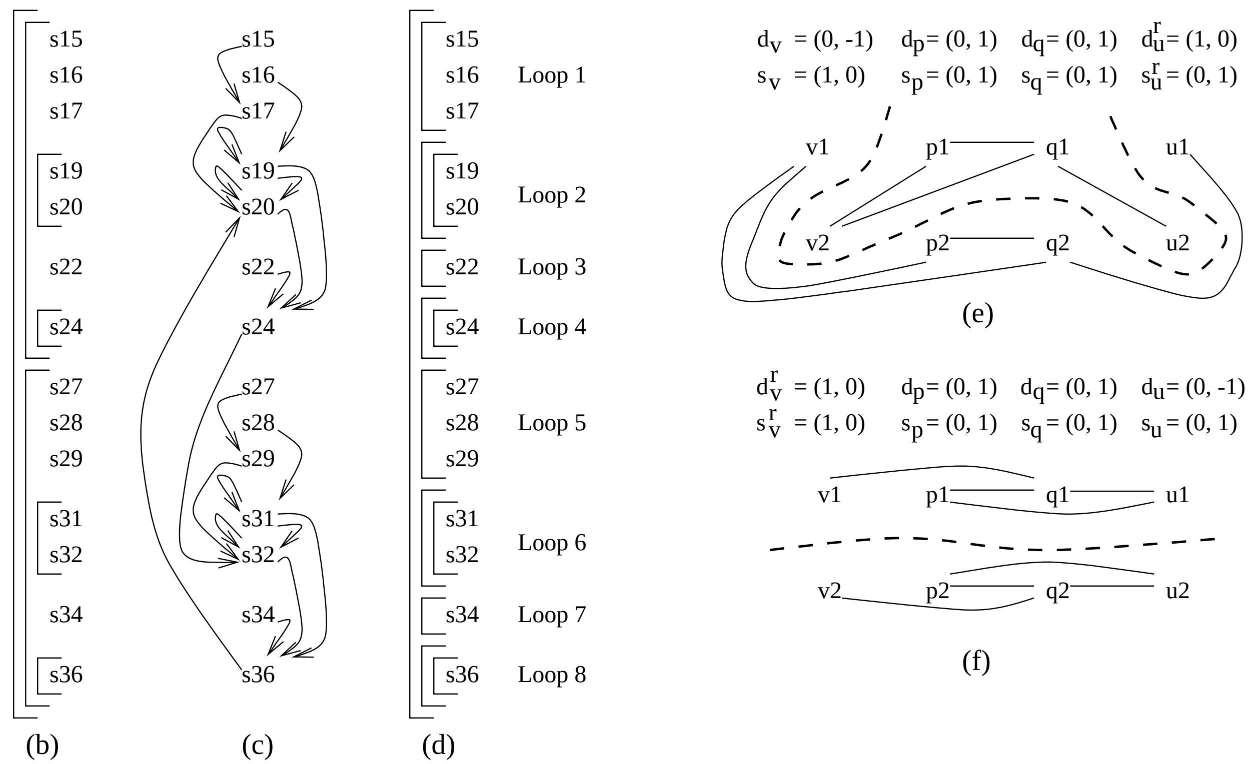


Figure 4: (a) A version of 2D heat equation program, (b) structure of Do-loops including statements from lines 13 to 38, (c) data dependence relations among statements, (d) an equivalent program after loop fission, (e) component affinity graph of lines from 14 to 25, (f) component affinity graph of lines from 26 to 37.

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 1-D arrays. But if p and q do induce dependence relations among PEs and tiling techniques are used, then p and q need to be maintained as two dimensional.

We now perform Step 2 to determine axis alignment for each program fragment. Figure 4-(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 4-(d), we obtain temporal and spatial vectors: $d_p = (0,1)$, $s_p = (0,1)$, $d_q = (0,1)$, $s_q = (0,1)$, $d_u^{\mathtt{r}} = (1,0)$, and $s_u^{\mathtt{r}} = (0,1)$; from Loop 4, we obtain $d_v = (0,-1)$ and $s_v = (1,0)$. (The second component of d_v is negative because the loop control index j is decreasing.)

Figure 4-(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 4-(d), we obtain temporal and spatial vectors: $d_p = (0,1)$, $s_p = (0,1)$, $d_q = (0,1)$, $s_q = (0,1)$, $d_v^{\mathtt{r}} = (1,0)$, and $s_v^{\mathtt{r}} = (1,0)$; from Loop 8, we can obtain $d_u = (0,-1)$ and $s_u = (0,1)$.

4.1 Different data distributions for column sweep and row sweep

We first consider the column sweep, and perform for it Steps 3 and 4. 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 vectors and the spatial vectors arising from this Do-loop. There is one such temporal dependence vector $d_v = (0, -1)$ and one such spatial dependence vector $s_v = (1, 0)$. Because of the spatial dependence vector $s_v = (1, 0)$, we assign an "*" to the first dimension of v, and write $v(*,\quad)$, indicating that we prefer not to distribute the first dimension of v.

Since the subscripts of the second dimension of v involve only the outermost loop control index variable i, the iteration space mapping vector **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 4-(e), we have the following data distributions (where, as before, "×" 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 5-(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 5-(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 5-(b) shows data layout of array v and Figure 5-(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

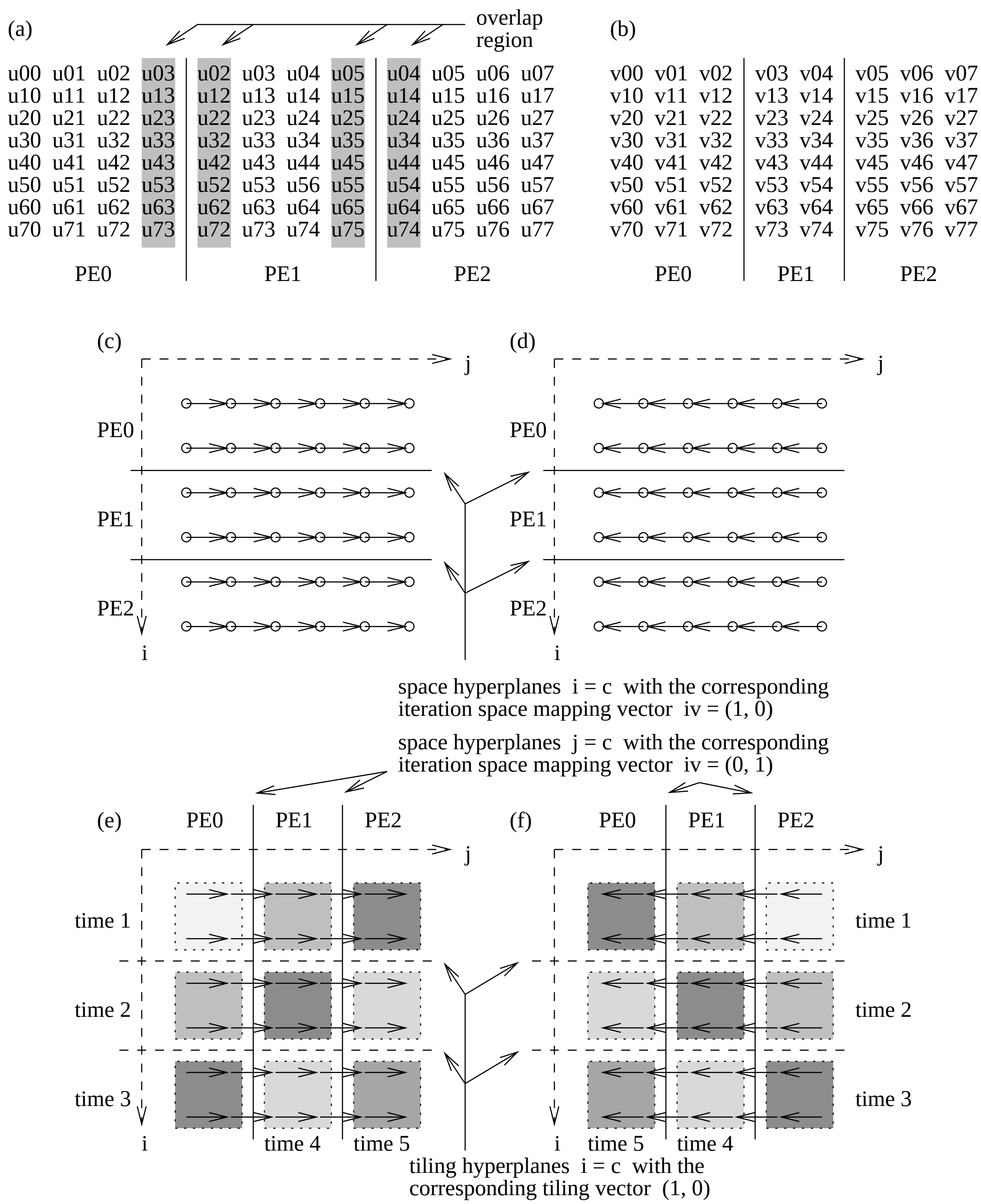


Figure 5: Data layout based on the schema in formula (4) when $N_x = N_y = 6$ and there are three PEs. (a) Data layout and the overlap region of array u, (b) data layout of array v, and temporal dependence among iterations in (c) Loop 2, (d) Loop 4, (e) Loop 6, and (f) Loop 8.

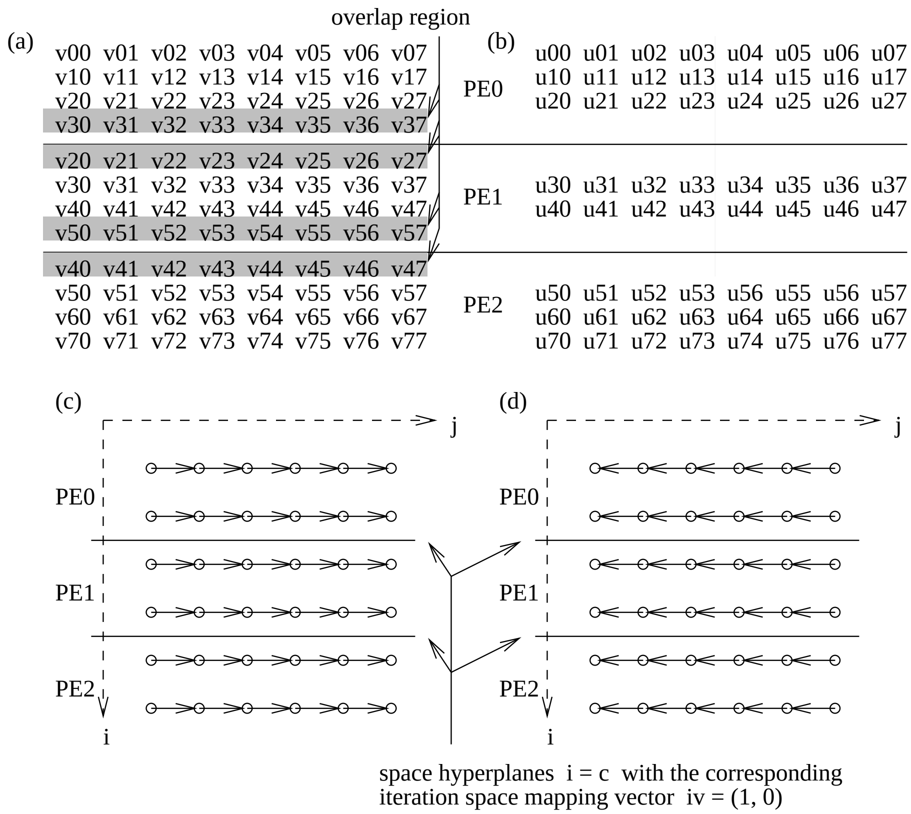


Figure 6: Data layout based on the schema in formula (5) when $N_x = N_y = 6$ and there are three PEs. (a) Data layout and the overlap region of array v, (b) data layout of array u, and temporal dependence among iterations in (c) Loop 6, and (d) Loop 8.

in Loop 8. In this Do-loop, there is one temporal dependence vector $d_u = (0, -1)$ and one spatial dependence vector $s_u = (0, 1)$. Because of s_u, we get $u(\quad, *)$.

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 4-(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 $\mathbf{iv}$ for Loop 6 is $(1,0)$ as illustrated in Figure 6-(c). As $\mathbf{iv} \cdot d_p = 0$, $\mathbf{iv} \cdot d_q = 0$, and $\mathbf{iv} \cdot d_v^r = (1,0) \cdot (1,0) = 1$, communication is needed only for accessing the read-only array v, and as depicted in Figure 6-(a), we can use an overlap region to maintain a consistent memory access of remote read-only data received from neighboring PEs. Third, there is no communication while executing Loop 8. Figure 6-(b) shows data layout of array u and Figure 6-(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 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 re-organization 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.

4.2 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 5-(e). We will 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 5-(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 paper, providing both theoretical and experimental results.

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 (excluding privatization arrays), in which g dimensions of the data array are to be distributed. That is, in the most computationally-intensive nested Do-loop, where the target data array variables are generated or are 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 will use four symbols "$\emptyset$", "$*$", "$\surd$", and "$\times$". 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 $A(*, \surd, \emptyset, *, \times)$. The meaning of the symbols is as follows. "$\emptyset$" indicates that we have not yet made any decision on how to handle the dimension, "$*$" indicates that we tentatively decided not to distribute on the dimension, "$\surd$" 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 $\emptyset$'s. During the execution of the algorithm, "$\emptyset$" can be replaced by "$*$", "$\surd$", or "$\times$"; and "$*$" can be replaced by "$\surd$" or "$\times$". When the algorithm terminates, all dimensions have "$\surd$" or "$\times$". We leave the special case of deciding whether a specific dimension of a data array should be replicated among PEs or should be stored within only a PE to a following optimization phase, which is beyond the scope of this paper.

Excluding all 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. $|\surd|$ will denote the number of $\surd$'s appearing in the dimensions of A, etc. Thus for $A(*, \surd, \emptyset, *, \times)$, we have $|*| = 2$ and $|\emptyset| = |\surd| = |\times| = 1$. Of course, $|\emptyset| + |*| + |\surd| + |\times| = |A|$ at every stage of the algorithm.

Substep 1.

Action: We replace some $\emptyset$'s by $*$'s as follows. We find all spatial dependence/use vectors of A. For each such vector, if its ith component is non-zero, we put an "$*$" in the ith dimension of A.

Explanation: The condition implies that different elements (positions) of A in the ith dimension will appear in the same iteration. Therefore, it is desirable to attempt not to distribute on the ith dimension of A.

Substep 2.

Action: We replace some $*$'s by $\times$'s as follows. 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. If there exist two distinct level-j and level-j' loop control index variables and each appears in dimension i of different occurrences of A in the Do-loop, we put a "$\times$" in the ith dimension of A.

Explanation: The condition implies that some temporal dependence vector or temporal use vector is irregular. Therefore, we decide not to distribute on the ith dimension of A.

Substep 3.

In the general case, all four symbols can appear in the dimensions of A (though the first time we reach this step, only $\emptyset$'s, $*$'s, and $\times$'s appear). Due to the relative complexity of this step, we present examples immediately following the description of the algorithm. 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 two candidate sets of elementary vectors, S and S', where $S \cap S' = \emptyset$. S will contain all elementary vectors of length n, where n is the depth of the Do-loop, satisfying the following condition. For each e_j in S, some dimension of A has been marked with an "$\emptyset$", and all the occurrences of A in the Do-loop refer to the level-j loop control index variable in that dimension. Note that, if some occurrence of A in the Do-loop refers to another level-j' loop control index variable also in that dimension, then that dimension has been marked with "$\times$" in Substep 2. S' will contain all the elementary vectors of length n which are not in S, satisfying the following condition. For each e_j in S', some dimension of A has been marked with an "$*$", and all the occurrences of A in the Do-loop refer to the level-j loop control index variable in that dimension.

We will consider four cases depending on the value of g with respect to the values of $|\surd| + |\emptyset|$ and $|\surd| + |\emptyset| + |*|$. The simplest cases are 2 and 4, but we will proceed in the order based on the relative value of g. Before that, we find all temporal dependence vectors and temporal use vectors for all data arrays generated or used in this Do-loop.

Case 1: $g < |\surd| + |\emptyset|$. We select additional $g - |\surd|$ elementary vectors from S for iteration distribution (and corresponding data distribution of A). We will select them so as to heuristically minimize their interference with temporal dependence vectors and temporal use vectors.

For each vector in S, we define a *rank* of length two. The first component is the number of temporal dependence vectors to which the vector is *not* orthogonal; the second component is the number of temporal use vectors to which the vector 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: S_1, S_2, S_3, ... Thus a vector is in S_i if and only if its rank is i. Let $|S_i|$ be the cardinality of S_i.

Let $r \geq 1$ be minimal such that $\sum_{i=1}^{r} |S_i| \geq g - |\surd|$. If $\sum_{i=1}^{r} |S_i| = g - |\surd|$, we remove from S the vectors not in $\cup_{i=1}^{r} S_i$ and add them to S'. We put $\surd$'s in dimensions of A corresponding to vectors in S, put $\times$'s in dimensions of A corresponding to vectors in S', and we are done. To obey the alignment constraints, other data arrays will inherit $\surd$'s and $\times$'s information.

Otherwise, if $\sum_{i=1}^{r} |S_i| \geq g - |\surd|$, we first remove from S the vectors not in $\cup_{i=1}^{r} S_i$ and add them to S'. Second, we put $\times$'s in dimensions of A corresponding to vectors in S'. Third, we remove from S the vectors in S_r, and put $\surd$'s in dimensions of A corresponding to vectors in S. Note that the corresponding $|S_r|$ dimensions of the $|S_r|$ vectors in S_r have been marked with $\emptyset$'s. It remains to decide which $g - |\surd|$ dimensions of A, from those corresponding to vectors in S_r, should be selected.

If data array variables of A are generated or used in other nested Do-loops that have not yet been considered, we repeat Step 2 with S_r playing the role of S. Otherwise, if there are additional generated-and-used data arrays, whose data distributions have not yet been completely determined, we repeat Step 1. If there are still remaining dimensions whose distributions need to be determined, we arbitrarily put $\surd$'s in the $g-|\surd|$ dimensions corresponding to vectors in S, 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 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.

Case 2: $g = |\surd| + |\emptyset|$. In addition to the original $|\surd|$ dimensions, which are marked with $\surd$'s, we put $\surd$'s in all of the $|\emptyset|$ dimensions which originally are marked with $\emptyset$'s. We also put $\times$'s in all of the $|*|$ dimensions which originally are marked with $*$'s. To obey the alignment constraints, other data arrays will inherit $\surd$'s and $\times$'s information.

Case 3: $|\surd| + |\emptyset| < g < |\surd| + |\emptyset| + |*|$. We take all the vectors from S and select $g - |\surd| - |\emptyset|$ vectors from S' for iteration distribution. To select the appropriate vectors from S', we use a heuristic similar to the one used in Case 1. For each vector e_i in S', 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 (6) in Section 6). At this point, we need to know that it is better to have the flexibility to choose arbitrary tile sizes. As described later, if Inequality (6) 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 (6); 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, ... Choosing $g - |\surd| - |\emptyset|$ vectors from S' is similar to that of choosing vectors from S in Case 1. We add these $g - |\surd| - |\emptyset|$ vectors into S, then we put $\surd$'s in dimensions of A corresponding to vectors in S, put $\times$'s in dimensions of A corresponding to vectors in S', and we are done.

Case 4: $|\surd| + |\emptyset| + |*| < g$. We put $\surd$'s in dimensions of A corresponding to vectors in both S and S'.

Data array A then is either fixed along $(g - |\surd| - |\emptyset| - |*|)$ dimensions of the PE grid or is replicated on those dimensions. We leave these special cases to a following optimization phase, which is beyond the scope of this paper.

Substep 4. /* 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 which are marked with "$\surd$", we have to decide block sizes for the corresponding `cyclic`($\mathtt{db}_i$) 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 which compromise both load balance and communication overhead [27]. 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`($\mathtt{b}_2$), and $b_1/s_1 = b_2/s_2$; then their stride alignments are matched. For details of generating communication sets, interested readers can refer to [27]. 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` (`cyclic`(1)) distribution is preferred. 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. □

Examples. The target machine is a linear processor array, therefore, $g = 1$.

Example of Case 1: Consider the matrix multiplication algorithm as specified in Figure 7-(a). The corresponding component affinity graph and temporal vectors are shown in Figure 7-(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. The subscripts of the first dimension of C only involve the level-2 loop control index variable i, therefore, e_2 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_3 is another candidate. Thus, $S = \{e_2, e_3\}$ and $g = 1 < |S| = 2$. $\mathrm{rank}(e_2) = \mathrm{rank}(e_3) = (0, 1)$. We have to sacrifice one candidate. The final iteration space mapping vector is arbitrarily chosen as e_2 for fixing array A in local memory. Because the iteration space is rectangular, `block` distribution is used. Since e_2 corresponds to the first dimension of C, according to the alignment relations, we have the following data distributions: $C(\texttt{block}, \times)$, $A(\texttt{block}, \times)$, and $B(\texttt{block}, \times)$.

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

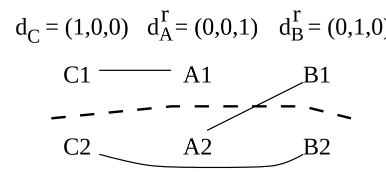
(b)


Figure 7: (a) A matrix multiplication algorithm, (b) the corresponding component affinity graph.

Note that, in Figure 7-(b), since the dashed partition line goes across an edge (connecting B1 and A2), communication overhead cannot be avoided if we only allow to store one copy of data array B in the PE array. But since array B is read-only, appropriately storing multiple copies of B can be used to avoid the need for communication [24]. This discussion of when it is worthwhile to replicate read-only data to reduce the communication cost is beyond the scope of this paper.

Example of Case 2: Consider the program fragment of the column sweep of the 2D heat equation in Figure 4-(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 $s_v = (1, 0)$. Therefore, we prefer not to distribute the first dimension of v, thus $v(*, \emptyset)$. Since the subscripts of the second dimension of v only involve the outermost loop control index variable i, e_1 is a candidate of the iteration space mapping vector, say $S = \{e_1\}$. Since $g = 1 = |S|$, the final iteration space mapping vector is chosen as e_1, and the final data distribution scheme is determined as shown in Equation (4).

Example of Case 3: Consider the depth-two nested Do-loop in Figure 8-(a). Suppose that data distribution for array C has not been determined, and we proceed to do it now. Array C has two spatial vectors $(0, 1)$ and $(1, 0)$, therefore, we put two "*"s in both dimensions of C, thus, $C(*, *)$.

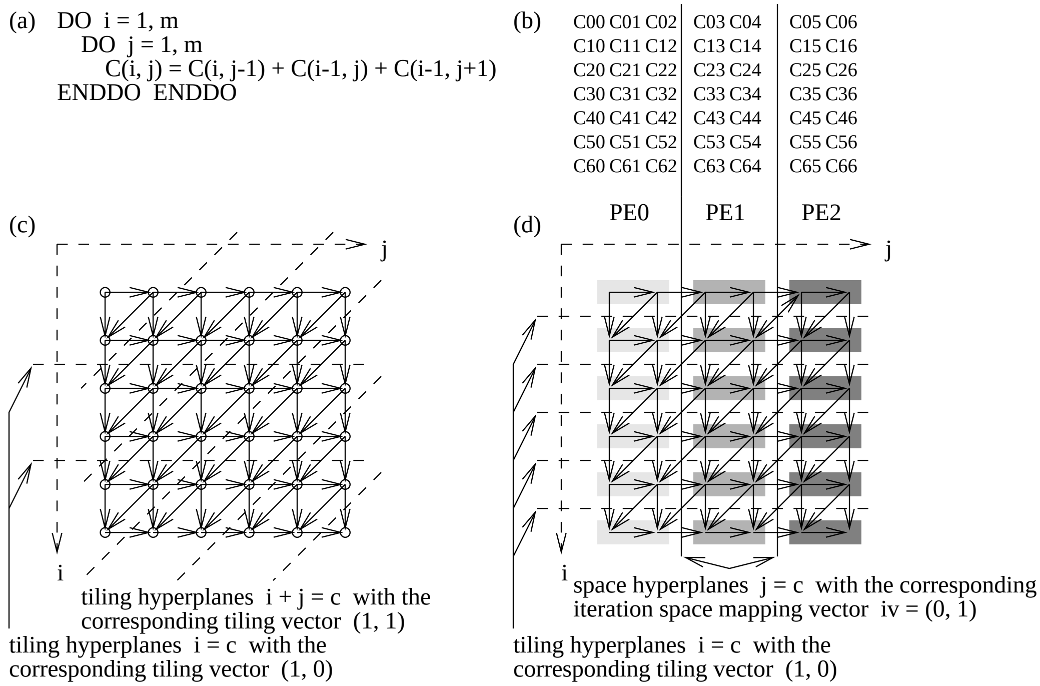


Figure 8: (a) A Do-loop has three temporal dependence vectors: $(0, 1)$, $(1, 0)$, and $(1, -1)$, (b) array C is distributed as $C(\times, \texttt{block})$, (c) two tiling vectors are $(1, 0)$ and $(1, 1)$, (d) an iteration space mapping vector is $(0, 1)$ and a tiling vector is $(1, 0)$.

Thus, $S = \emptyset$, $S' = \{e_1, e_2\}$, and $|S| = 0 < g = 1 < |S \cup S'| = 2$. We have to remove one "*" from C. Since C has three temporal dependence vectors: $(1, 0)$, $(0, 1)$, and $(1, -1)$, e_1 satisfies the Inequality (6) 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. 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 $C(\texttt{block}, \times)$.

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 [33, 47]. 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 4-(d)). We decide on data distribution first, and based on it 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 dependency 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 [4, 14, 30, 48] and ours, 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 cheap. 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 5-(e) and Figure 5-(f). While employing a shared memory machine model, data distribution is ignored. For data locality reasons, it is better that 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 [18]. For each tiling vector, say h,

$$\begin{array}{rl} \text{either} & h \cdot d_j \geq 0 \text{ for all temporal dependence vectors } d_j \\ \text{or} & h \cdot d_j \leq 0 \text{ for all temporal dependence vectors } d_j. \end{array} \tag{6}$$

Let D be the set of all temporal dependence/use vectors. If $\mathrm{rank}(D) < n$, 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, …, 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, …, d'_{n-1}). If h satisfies the constraint in Inequality (6), 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 [4, 14, 48]. We will refer to this method as *memory-oblivious tiling method.*

We consider now a one-dimensional distributed memory PE array, the Do-loop in Figure 8-(a), and the data distribution as in Figure 8-(b) (data array C is distributed along its second dimension, $C(\times, \texttt{block})$). We will attempt to apply 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 8-(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 8-(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 8, 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 8-(d). We can choose $e_1 = (1, 0)$ as the second tiling vector and the tile size as $1 \times$ `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* (6). 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* (6). We will 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 temporally 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 non-zero 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 (6) would be satisfied.

Let d_k be a temporal vector for which $e_i \cdot d_k < 0$. Let α be the position of the first non-zero entry of d_k, say $d_{k,\alpha}$. Then e_α is chosen as a tiling vector and the tile size along the αth dimension must be at most $d_{k,\alpha}$. 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* [37], therefore, all tiles satisfy the atomic computation constraint. For example, in a 2-D 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.

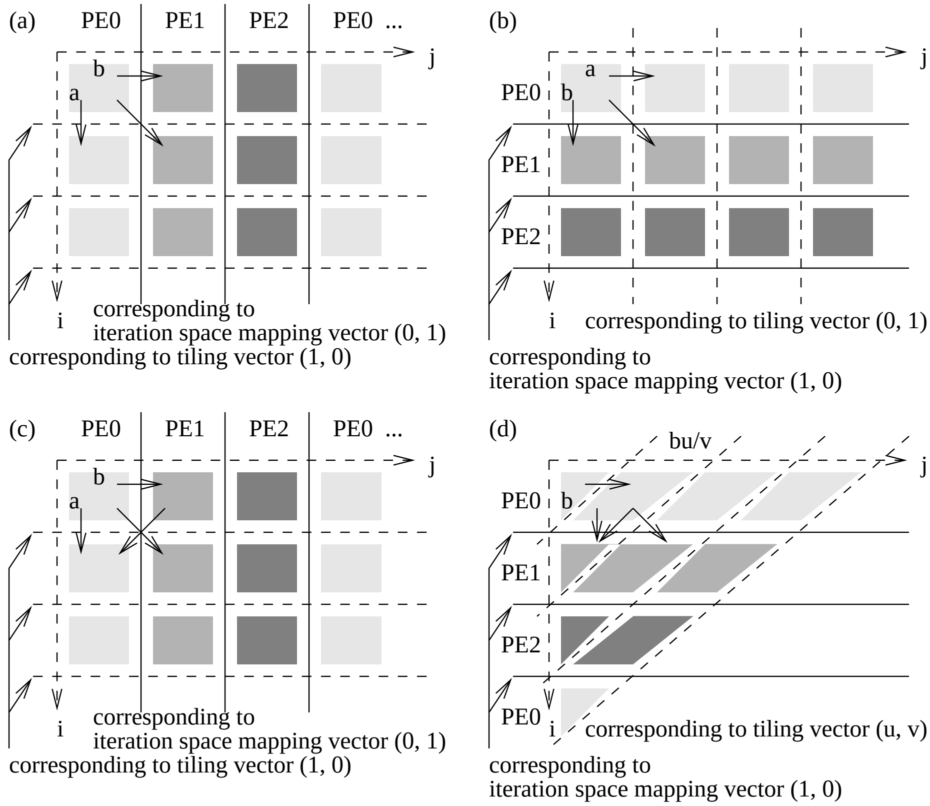


Figure 9: Four cases of tiling two-dimensional rectangular iteration space.

6.3 Optimizing tile sizes

To minimize total execution time, it may be useful to choose tile sizes which are smaller than those forced 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 are able to provide an analytical method to determine tile shapes and sizes only for a rectangular 2-D 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 non-negative* as depicted in Figure 9-(a). In this case, tiles are executed columnwise. 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:

$$T \leq (XY/(ZN) + N - 1)(Zt_{\mathrm{f}} + t_{\mathrm{s}} + at_{\mathrm{c}}), \tag{7}$$

where t_{f} is the execution time for performing an iteration, t_{s} is the set up time for 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 will ignore the term at_{c} in Equation (7). We will therefore minimize $T' = (XY/(ZN) + N - 1)(Zt_{\mathrm{f}} + t_{\mathrm{s}})$. Then the optimal Z is $(XYt_{\mathrm{s}}/(N(N-1)t_{\mathrm{f}}))^{1/2}$. (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 non-negative*, as depicted in Figure 9-(b). The derivation is similar to that of in Case 1. We get $Z = (XYt_{\mathrm{s}}/(N(N-1)t_{\mathrm{f}}))^{1/2}$. 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 9-(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, for the total execution time:

$$T \leq (XY/(ZN) + 2(N-1))(Zt_{\mathrm{f}} + t_{\mathrm{s}} + at_{\mathrm{c}}). \tag{8}$$

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 $Z \geq Y/N$, $b = Z$ otherwise. Then $a = \min\{Z/b, v\}$, where v is the smallest from all the positive entries appearing in the temporal vectors which 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 mutually prime positive integers*, as depicted in Figure 9-

(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 for the total execution time:

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

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 two applications: the two-dimensional heat equation and the two-dimensional Fast Fourier Transform, both on one-dimensional PE arrays. The actual experiments were conducted on two machines: a 32-node nCUBE-2 computer and a cluster of four UltraSPARC-II 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-II workstations, each with 64MB 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 [1]).

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 distribution, we can apply our tiling techniques to improve the execution time. Figure 10 shows the experiments on the nCUBE-2, using three algorithms: a dynamic data-layout algorithm (Aggregate) whose transpose operations are based on aggregate operations [10], 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 11 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 10 and Figure 11 show scalability and speed up; the execution time decreases when the number of PEs increases and the execution time increases when the size of the problem increases.

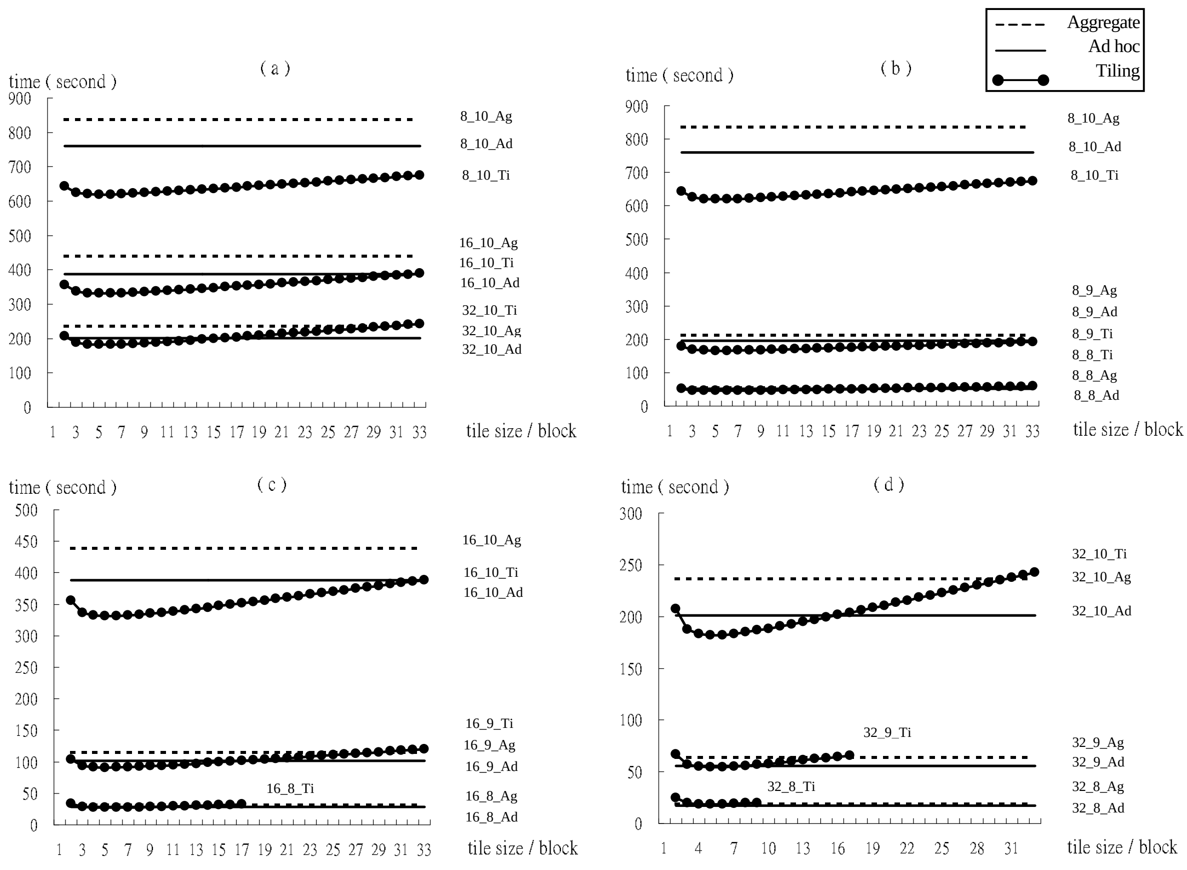


Figure 10: Execution time of three algorithms for solving the 2D heat equation on the nCUBE-2. (a) Data size is $2^{10} \times 2^{10}$ on 32 PEs, 16 PEs, and 8 PEs. Data sizes are $2^8 \times 2^8$, $2^9 \times 2^9$, and $2^{10} \times 2^{10}$ on (b) 8 PEs, (c) 16 PEs, and (d) 32 PEs.

The pure computation times for these three algorithms are basically the same. However, their

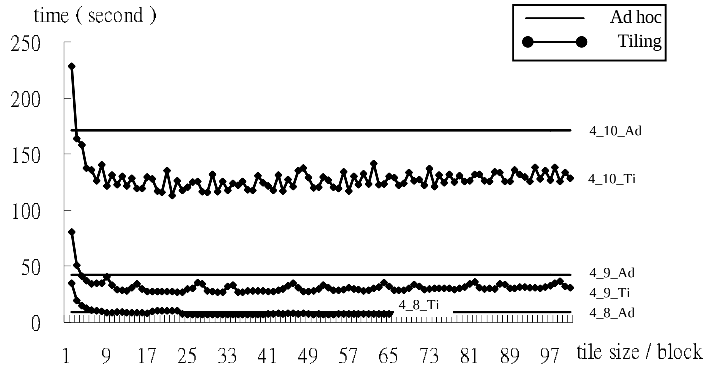


Figure 11: Execution time of two algorithms for solving the 2D heat equation on the cluster, where data sizes are $2^8 \times 2^8$, $2^9 \times 2^9$, and $2^{10} \times 2^{10}$.

communication times, which depend on the volume of data transferred, are quite different. Suppose that the data size is $N_\mathrm{x} \times N_\mathrm{y}$ partitioned among N PEs, with each containing $N_\mathrm{x}N_\mathrm{y}/N$ data. For "Aggregate," a PE sends $2(\log N)N_\mathrm{x}N_\mathrm{y}/(2N)$ data to other PEs; the factor 2 is due to one 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 $2(N-1)N_\mathrm{x}N_\mathrm{y}/N^2$ data to other PEs. For "Tiling," a PE only sends $3N_\mathrm{x}$ or $3N_\mathrm{y}$ 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 $\texttt{block} = N_\mathrm{x}/N$ or N_y/N. For the nCUBE-2, $t_\mathrm{s} = 350$ μs, $t_\mathrm{c} = 4.56$ μs, $t_\mathrm{f} = 17$ μs for executing an iteration of Loop 2 in Figure 4-(d), and $t_\mathrm{f} = 8$ μs for executing an iteration of Loop 4 in Figure 4-(d). Then for Loop 2, $Z/\texttt{block} = 4$ or 5, which matches the experimental results. The factor $Z/\texttt{block}$ for Loop 4 is 6 or 7. For the cluster, $t_\mathrm{s} = 819$ μs, $t_\mathrm{c} = 0.21$ μs, $t_\mathrm{f} = 1.1$ μs for executing an iteration of Loop 2, and $t_\mathrm{f} = 0.5$ μs for executing an iteration of Loop 4. Then for Loop 2, $Z/\texttt{block} = 32$, which matches the experimental studies. The factor $Z/\texttt{block}$ 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 1-D FFT for each row, Loop 2 computes a 1-D FFT for each column, Loop 3 computes an inverse 1-D FFT for each column, and Loop 4 computes an inverse 1-D 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 $s_{A_{\mathrm{R}}} = (0,1)$ and $s_{A_{\mathrm{I}}} = (0,1)$, respectively. By our method, the data distribution functions for A_{R} and A_{I} will be $A_{\mathrm{R}}(\texttt{block}, \times)$ and $A_{\mathrm{I}}(\texttt{block}, \times)$, 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 $s_{A_{\mathrm{R}}} = (1,0)$ and $s_{A_{\mathrm{I}}} = (1,0)$, respectively. The data distribution functions for A_{R} and A_{I} will be $A_{\mathrm{R}}(\times, \texttt{block})$ and $A_{\mathrm{I}}(\times, \texttt{block})$, 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 12 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.

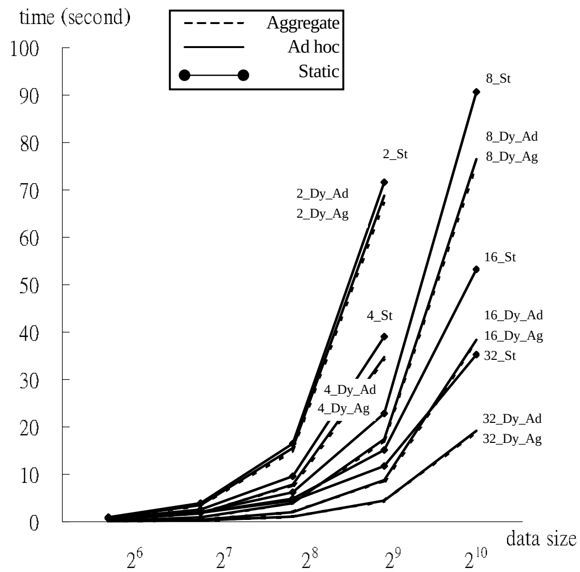


Figure 12: Execution time of three algorithms for solving the 2D FFT on the nCUBE-2.

8 Conclusions

Experienced programmers write scientific application programs following a good programming style. For example, while writing a program for the 2-D 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 invented 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 invented—to help determine which dimensions of the dominant data array are better not to be distributed. 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 which involve the dominant data array, starting from the most computationally-intensive Do-loop. We had 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 the computation and communication time. However, tiling the iteration space depends on data distributions, 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.

References

[1] MPICH, a portable implementation of MPI. Technical report, Argonne National Laboratory and University of Chicago, 1996.

[2] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[3] J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors*. PhD thesis, Dept. of EE and CS, Stanford Univ., Stanford, CA, Mar. 1997.

[4] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.

[5] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

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

[7] T. Chen and J. Sheu. Communication-free data allocation techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel Distributed Syst.*, 5(9):924–938, Sep. 1994.

[8] I. Couvertier-Reyes. *Automatic Data and Computation Mapping for Distributed Memory Machines*. PhD thesis, Louisiana State University, Baton Rouge, Louisiana, May 1996.

[9] F. Desprex, J. Dongarra, F. Rastello, and Y. Robert. Determining the idle time of a tiling: New results. *Journal of Information Science and Engineering*, 14(1):167–190, March 1998.

[10] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[11] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel Distributed Syst.*, 3(2):179–193, Mar. 1992.

[12] S. Hiranandani, K. Kennedy, and C-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[13] C. T. Ho. *Optimal Communication Primitives and Graph Embeddings on Hypercubes.* PhD thesis, Yale Univ., 1990.

[14] E. Hodzic and W. Shang. On supernode transformation with minimized total running time. *IEEE Trans. Parallel Distributed Syst.*, 9(5):417–428, May 1998.

[15] C. H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24:595–632, 1987.

[16] C. H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19:90–102, 1993.

[17] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., New York, 1993.

[18] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, pages 319–329, San Diego, California, January 1988.

[19] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM Trans. on Programming Languages and Systems*, 20(4):869–916, July 1998.

[20] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[21] U. Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Rice University, Houston, Texas, October 1995.

[22] H. T. Kung and C. E. Leiserson. *Introduction to VLSI Systems*, chapter 8.3 Algorithms for VLSI Processor Arrays. Edited by C. Mead and L. Conway. Addison-Wesley, Reading, MA, 1980.

[23] M. S. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1987.

[24] P.-Z. Lee. Parallel matrix multiplication algorithms on hypercube multicomputers. *International Journal of High Speed Computing*, 7(3):391–406, Sep. 1995.

[25] P.-Z. Lee. Techniques for compiling programs on distributed memory multicomputers. *Parallel Computing*, 21(12):1895–1923, 1995.

[26] P.-Z. Lee. Efficient algorithms for data distribution on distributed memory parallel computers. *IEEE Trans. Parallel Distributed Syst.*, 8(8):825–839, Aug. 1997.

[27] P.-Z. Lee and W. Y. Chen. Generating global name-space communication sets for array assignment statements. Technical Report TR-IIS-97-016, Institute of Information Science, Academia Sinica, Taipei, Taiwan, October 1997, on available via WWW http://www.iis.sinica.edu.tw/∼leepe/PAPER/tr97016.ps.

[28] P.-Z. Lee and Z. M. Kedem. Synthesizing linear-array algorithms from nested For loop algorithms. *IEEE Trans. Comput.*, C-37:1578–1598, December 1988.

[29] P.-Z. Lee and Z. M. Kedem. Mapping nested loop algorithms into multi-dimensional systolic arrays. *IEEE Trans. Parallel Distributed Syst.*, 1:64–76, Jan. 1990.

[30] P.-Z. Lee and Z. M. Kedem. On high-speed computing with a programmable linear array. *The Journal of Supercomputing*, 4(3):223–249, Sep. 1990.

[31] J. Li and M. Chen. Compiling communication-efficient problems for massively parallel machines. *IEEE Trans. Parallel Distributed Syst.*, 2(3):361–376, July 1991.

[32] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

[33] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Trans. on Programming Languages and Systems*, 18(4):424–453, July 1996.

[34] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Comput.*, C-35:1–12, Jan. 1986.

[35] Q. Ning, V. Van Dongen, and G. R. Gao. Automatic data and computation decomposition for distributed memory machines. *Parallel Processing Letters*, 5(4):539–550, 1995.

[36] D. J. Palermo. *Compiler Techniques for Optimizing Communication and Data Distribution for Distributed-Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1996.

[37] C. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Trans. Comput.*, C-37(8):991–1004, August 1988.

[38] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel Distributed Syst.*, 2(4):472–482, Oct. 1991.

[39] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[40] H. B. Ribas. *Automatic Generation of Systolic Programs From Nested Loops.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, June 1990.

[41] A. Rogers and K. Pingali. Compiling for distributed memory architectures. *IEEE Trans. Parallel Distributed Syst.*, 5(3):281–298, Mar. 1994.

[42] W. Shang and J. A. B. Fortes. On time mapping of uniform dependence algorithms into lower dimensional processor arrays. *IEEE Trans. Parallel Distributed Syst.*, 3(3):350–363, May 1992.

[43] K.-P. Shih, J.-P. Sheu, and C.-H. Huang. Statement-level communication-free partitioning techniques for parallelizing compilers. In D. Sehr et al., editor, *Lecture Notes in Computer Science 1239, Ninth International Workshop on Languages and Compilers for Parallel Computing*, pages 389–403, San Jose, California, Aug. 1996. Springer-Verlag.

[44] J. C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*, chapter 7.3 The Alternating Direction Implicit (ADI) Method, pages 142–153. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, CA, 1989.

[45] P. S. Tseng. *A Systolic Array Parallelizing Compiler.* Kluwer Academic Publishers, Boston, MA, 1990.

[46] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Syst.*, 2(4):452–471, Oct. 1991.

[47] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.

[48] J. Xue. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing*, 42:42–59, 1997.

[49] H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proc. of the IEEE*, 81(2):264–287, Feb. 1993.

Appendix: 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 [26]. As is shown in Step 1 of the proposed method in Section 3, we will 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 will 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 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 will 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 [32], which we adopt. For completeness, in Figure 13, we present a very brief version of the component alignment algorithm; for fuller details, see [32]. 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.

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 components (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.

Figure 13: Heuristic component alignment algorithm.