Communication-Free Data Allocation Techniques for Parallelizing Compilers on Multicomputers

Tzung-Shi Chen and Jang-Ping Sheu, *Member, IEEE*

Abstract—**In distributed memory multicomputers, local memory accesses are much faster than those involving interprocessor communication. For the sake of reducing or even eliminating the interprocessor communication, the array elements in programs must be carefully distributed to local memory of processors for parallel execution. In this paper, we devote our efforts to the techniques of allocating array elements of nested loops onto multicomputers in a communication-free fashion for parallelizing compilers. We first analyze the pattern of references among all arrays referenced by a nested loop, and then partition the iteration space into blocks without interblock communication. The arrays can be partitioned under the communication-free criteria with nonduplicate or duplicate data. Finally, a heuristic method for mapping the partitioned array elements and iterations onto the fixed-size multicomputers under the consideration of load balancing is proposed. Based on our methods, the nested loops can execute without any communication overhead on the distributed memory multicomputers. Moreover, the performance of the strategies with nonduplicate and duplicate data for matrix multiplication is studied.**

Index Terms— **Data allocation, data dependence, distributed memory multicomputers, interprocessor communication, parallelizing compilers**

I. Introduction

DEVELOPMENT of parallelizing compilers that extract parallelism of sequential programs, generate parallel code, and map them onto various parallel machines has been the recent focus of researches. For distributed memory multicomputers, the memory access time from a processor to its own local memory is much faster than the time to local memory of the other processors. An efficient parallel executing program thus requires low communication overhead. Various compiler techniques have therefore been developed for the data allocation problem in order to reduce, or even eliminate, communication traffic on multicomputers.

Previously, a number of researchers [1], [12], [13], [15], [22], [23] paid their attention to exploiting a large amount of parallelism in sequential programs. However, exploiting a large amount of parallelism in sequential programs may not promise that executing the parallelized programs can obtain more efficiency on distributed memory multicomputers. The main reason is that the extracted parallelism may possibly give rise to additional communication overhead during parallel execution. Both factors, parallelism and communication overhead, that affect the execution efficiency should be considered together such that the parallelized programs can obtain better performance. Several researchers thus developed parallelizing compilers in which programmers must explicitly specify data allocation and in which the codes can then be generated with appropriate communication constructs [2], [10], [11], [19].

Achieving automatic data management in designing parallelizing compilers is nevertheless difficult, because the data must be attentively distributed so that communication traffic is minimized in parallel execution of programs. Several researchers [4]–[7], [14], [21] focus the data allocation problem on automatically allocating the data or restructuring the programs in order to improve the efficiency of usage of memory hierarchy or to reduce the interprocessor communication overhead in parallel machines. For shared memory multiprocessor systems, researchers Gannon, Jalby, and Gallivan [4], [5], Hudak and Abraham [7], Irigoin and Triolet [8], Wolf and Lam [21], and Wolfe [24] proposed several approaches to automatically transform programs and partition data for improving data locality and enhancing cache-hit ratio on the complex memory hierarchy. In addition, large amounts of communication overhead for distributed memory multicomputers may seriously degrade performance during parallel execution of programs. Some researchers, such as Gallivan, Jalby, and Gannon [4], King, Chou, and Ni [9], Ramanaujam and Sadayappan [17], and Sheu and Tai [20], studied the problems of transforming programs into the parallel form and reducing the interprocessor communication overhead. Furthermore, Ramanaujam and Sadayappan [18] focused on analyzing the For-all loops and partitioning these loops and the corresponding data such that the partitioned programs can be executed without communication overhead in distributed memory multicomputers.

In this paper, we concentrate on automatically allocating the array elements of nested loops with uniformly generated references [4] for communication-free execution on distributed memory multicomputers for parallelizing compilers. First, we analyze the pattern of references among all arrays referenced in a nested loop and derive the sufficient conditions for communication-free partitioning of arrays. Two communication-free partitioning strategies, nonduplicate data and duplicate data strategies, are discussed. Based on the consideration of no interblock communication existing, those sufficient conditions can be used to exploit the parallelism of the nested loop as much as possible. Our method can obtain

Manuscript received November 23, 1992; revised September 22, 1993. This work was supported by the National Science Council of the Republic of China under Grant NSC 82-0408-E-008-010.

The authors are with the Department of Computer Science and Information Engineering, National Central University, Chung-Li 32054 Taiwan, Republic of China; e-mail: sheujp@mbox.ee.ncu.edu.tw.

IEEE Log Number 9403071.

more parallelism than the method proposed by Ramanaujam and Sadayappan [18] in For-all loops with uniformly generated references. Then the communication-free code can be transformed to a parallel execution form. Finally, the parallelized nested loop and the corresponding array elements can be allocated to the fixed-size multicomputers under the consideration of load balancing. The performance of the data allocation with nonduplicate and duplicate data strategies is also discussed.

The rest of this paper is organized as follows. The nested-loop model, basic concept, and assumptions used in presenting the data allocation strategies are introduced in Section II. In Section III, the sufficient conditions for communication-free partitioning of array elements of nested loops are derived based on nonduplicate data and duplicate data strategies. Eliminating redundant computations is also considered in this section. Section IV states how to transform the partitioned nested loops into the parallel form and map the parallelized programs and the corresponding data onto the fixed-size multicomputers such that the processor workload is as balanced as possible. The performance of the strategies with nonduplicate and duplicate data for matrix multiplication is also studied. Finally, conclusions are summarized in Section V.

II. Basic Concept and Assumptions

Nested loops including a large number of referenced arrays are commonly used in scientific programs. In programs, nested loops usually provide a large amount of parallelism and are the most time-consuming parts. A normalized n-nested loop [23] with the following form is considered in this paper:

```
for I1 = 1 to u1
    for I2 = 1 to u2
      ⋮
      for In = 1 to un
          [loop body]
      end
      ⋮
    end
end
```

where u_j are integer-valued linear expressions possibly involving $I_1, I_2, \cdots, I_{j-1}$ for $1 < j \leq n$. Let $\mathbf{Z}$ and $\mathbf{R}$ denote the set of integers and the set of real numbers, respectively. The symbols $\mathbf{Z}^n$ and $\mathbf{R}^n$ represent the set of n-tuple of integers and the set of n-tuple of real numbers, respectively. The *iteration space* [23] of an n-nested loop is a subset of $\mathbf{Z}^n$ and is defined as $I^n = \{(I_1, I_2, \cdots, I_n) \mid 1 \leq I_j \leq u_j, \text{ for } 1 \leq j \leq n\}$. The vector $\bar{i} = (i_1, i_2, \cdots, i_n)$ in I^n is represented as an iteration of the nested loop. The *lexicographical order* of the iteration $\bar{i} = (i_1, i_2, \cdots, i_n)$ is before that of the iteration $\bar{i}' = (i'_1, i'_2, \cdots, i'_n)$ if $i_1 = i'_1, i_2 = i'_2, \cdots, i_{j-1} = i'_{j-1}$, and $i_j < i'_j$ for $1 \leq j \leq n$.

In the nested loop, there may exist *input*, *output*, *flow dependences*, and *antidependence* [16], denoted as the respective symbols δ^i, δ^o, δ^f, and δ^a, which are referred to as *data dependence* δ in the following discussions. The symbol $S(\bar{i})$ denotes a computation that statement S is performed at iteration $\bar{i} \in I^n$. That a computation $S'(\bar{j})$ is data dependent on a computation $S(\bar{i})$ is denoted as $S(\bar{i})\ \delta\ S'(\bar{j})$. Let the linear function $h : \mathbf{Z}^n \longrightarrow \mathbf{Z}^d$ be defined as a *reference function* $h(I_1, \cdots, I_n) = (a_{1,1}I_1 + \cdots + a_{1,n}I_n, \cdots, a_{d,1}I_1 + \cdots + a_{d,n}I_n)$ and be represented by the following matrix:

$$H = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots \\ a_{d,1} & \cdots & a_{d,n} \end{bmatrix}_{d \times n},$$

where $a_{i,j} \in \mathbf{Z}$, for $1 \leq i \leq d$ and $1 \leq j \leq n$. In the loop body, a d-dimensional array element $A[h(i_1, i_2, \cdots, i_n) + \bar{c}]$ may be referenced by the reference function h at iteration $(i_1, i_2, \cdots, i_n)$ in I^n, where $\bar{c}$ is known as the constant offset vector in $\mathbf{Z}^d$ [21]. The *data space* of array A is a subset of $\mathbf{Z}^d$ and is defined over the user-defined array subscript index set. For array A, all of s referenced array variables $A[H_p\bar{i} + \bar{c}_p]$, for $1 \leq p \leq s$, are called *uniformly generated references* [5], [21] if:

$$H_1 = H_2 = \cdots = H_s,$$

where H_p is the linear transformation function from $\mathbf{Z}^n$ to $\mathbf{Z}^d$, $\bar{i} \in I^n$, and $\bar{c}_p$ is the constant offset vector in $\mathbf{Z}^d$. Since little exploitable data dependence exists between nonuniformly generated references, we focus the data allocation to each array on the same reference function in a nested loop. The different arrays may have different reference functions.

Example 1: Consider a two-nested loop $L1$.

```
for i = 1 to 4
  for j = 1 to 4
    S1 : A[2i, j] := C[i, j] * 7 ;
    S2 : B[j, i + 1] := A[2i − 2, j − 1]
                        +C[i−1, j−1];    (L1)
  end
end
```

In this example, the iteration space is $I^2 = \{(i,j) \mid 1 \leq i, j \leq 4\}$. In loop $L1$, with three arrays A, B, and C, the following are the respective reference functions:

$$H_A = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix},\ H_B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},\ \text{and } H_C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

There exists a flow dependence between the variables $A[2i, j]$ at statement S_1 and $A[2i-2, j-1]$ at statement S_2, with the different offset vectors $(0,0)$ and $(-2,-1)$, respectively. For array C, only read by loop $L1$, there exists an input dependence between the variables $C[i,j]$ at statement S_1 and $C[i-1, j-1]$ at statement S_2, with the different offset vectors $(0,0)$ and $(-1,-1)$, respectively. The array variable $B[j, i+1]$ is generated only at statement S_2, and its offset vector is $(0,1)$. Loop $L1$ thus has uniformly generated references on arrays A, B, and C. □

Definition 1 [Data-Referenced Vector]: In an n-nested loop L with uniformly generated references, if there exist two referenced array variables $A[H\bar{i} + \bar{c}_1]$ and $A[H\bar{i} + \bar{c}_2]$ for array A, then the vector $\bar{r} = \bar{c}_1 - \bar{c}_2$ is called *data-referenced vector* of array A. □

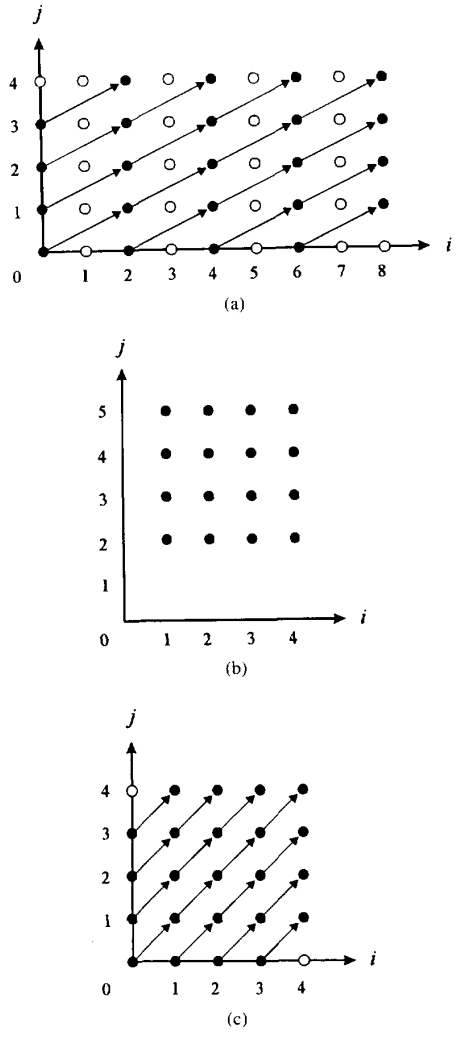


Fig. 1. Data spaces and their corresponding data-referenced vectors for arrays A, B, and C in loop $L1$. (a) Array $A[0:8,0:4]$. (b) Array $B[1:4,2:5]$. (c) Array $C[0:4,0:4]$.

The data-referenced vector $\bar{r}$ represents the vector difference between two array elements $A[H\bar{i}+\bar{c}_1]$ and $A[H\bar{i}+\bar{c}_2]$, which are referenced by an iteration $\bar{i}$. Note that any data dependence in loop L exists between two distinct referenced array variables $A[H\bar{i}+\bar{c}_1]$ and $A[H\bar{i}+\bar{c}_2]$; i.e., two iterations $\bar{i}_1$ and $\bar{i}_2$ can reference the same array element if and only if $H\bar{i}_1+\bar{c}_1 = H\bar{i}_2+\bar{c}_2$; i.e., $H(\bar{i}_2-\bar{i}_1)=\bar{r}$. Communication overhead is therefore not to be incurred if the iteration space is partitioned along the direction $\bar{i}_2-\bar{i}_1$ into iteration blocks and if the data space of array A is partitioned along the direction $\bar{r}$ into data blocks.

Example 1 is presented here to illustrate the ideas of communication-free data allocation strategy. The arrays A, B, and C of loop $L1$ have the referenced array variables $A[2i,j]$, $A[2i-2,j-1]$, $B[j,i+1]$, $C[i,j]$, and $C[i-1,j-1]$, respectively. The data-referenced vectors of arrays A and C are $\bar{r}_1=(2,1)$ and $\bar{r}_2=(1,1)$, respectively. There exists no data-referenced vector of array B, because there only exists one referenced array variable on array B. All of the data spaces of arrays A, B, and C, and their data-referenced vectors of each array element are shown in Fig. 1(a), 1(b), and 1(c), respectively. In Fig. 1, solid points represent array elements that are used in loop $L1$, and empty points represent that array elements are not used in loop $L1$.

At iteration $(1,1)$, the array element $A[2,1]$ is generated by S_1, and $A[0,0]$ is used in S_2. Then, at iteration $(2,2)$, the array element $A[4,2]$ is generated by S_1, and $A[2,1]$ is used in S_2, and so on. Restated, two iterations, $\bar{i}_1=(1,1)$ and $\bar{i}_2=(2,2)$, satisfying the condition $H_A(\bar{i}_2-\bar{i}_1)=\bar{r}_1$ can access the same array element $A[2,1]$. The data space of array A is therefore partitioned along the data-referenced vector $\bar{r}_1$ into the data blocks B_j^A for $1\le j\le 7$, enclosing the points with lines, as shown in Fig. 2(a). These used and generated array elements grouped in the same data block are then to be allocated to the same processor. Similarly, the array C is also partitioned along data-referenced vector $\bar{r}_2$ into their corresponding data blocks, B_j^C for $1\le j\le 7$, as shown in Fig. 2(c). It is easy to show that if the iteration space is partitioned along the direction $(1,1)$, as shown in Fig. 3, there exists no interblock communication for arrays A and C. Therefore, array B must be partitioned along the direction $(1,1)$ into the corresponding data blocks B_j^B, $1\le j\le 7$, as shown in Fig. 2(b), such that the partitioned iteration blocks B_j, $1\le j\le 7$, can be executed in parallel without interblock communication.

It is not hard to see from Fig. 2 that there exists no data transfer between processors while the corresponding data blocks B_j^A, B_j^B, and B_j^C are assigned to the processor PE_j for $1\le j\le 7$. For cases where the number of iteration blocks is larger than the number of processors, see Section IV for discussion of how to make the workload as balanced as possible among processors. In the next section, given a nested loop, we analyze the relations of references among array elements and derive the sufficient conditions for communication-free partitioning of arrays in the nested loop on multicomputers.

III. Communication-Free Array Partitioning

In this section, we describe the communication-free array partitioning schemes that analyze the data usage of each array and derive the sufficient conditions for determining the partition patterns of array elements in nested loops.

A. Communication-Free Array Partitioning Without Duplicate Data

In this subsection, we discuss the communication-free array partitioning without duplicate data; i.e., there exists exactly one copy of each array element during execution of the program. No data transfer existing during parallel executing programs will obtain better efficiency in distributed memory multicomputers. However, having no interprocessor communication is impossible if a certain data dependence exists between partitioned programs. As long as those related data can be found and then grouped together, assigned into one

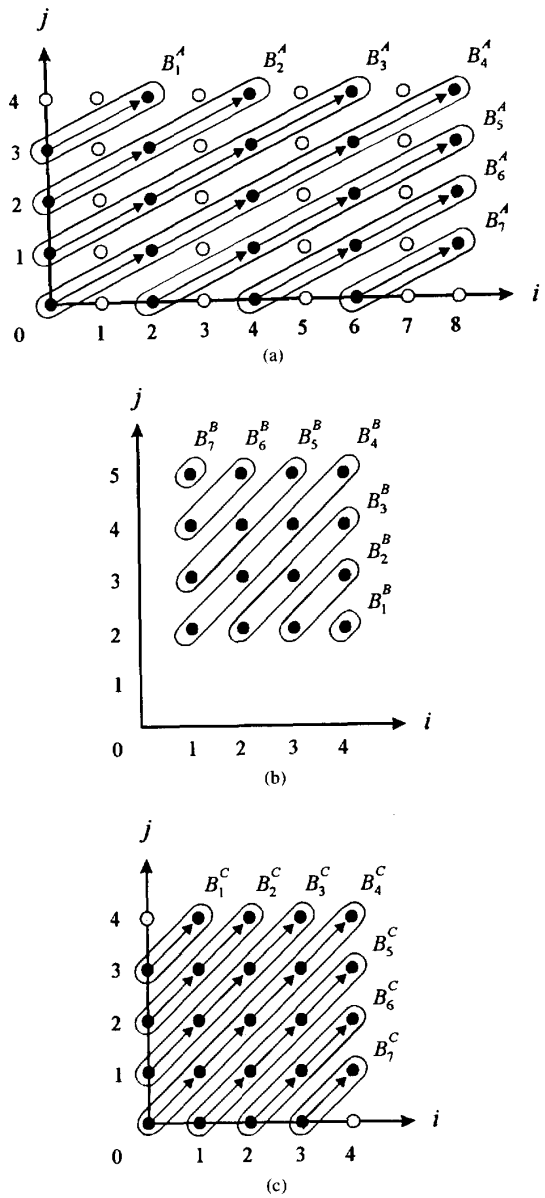


Fig. 2. Partitioning arrays A, B, and C of loop $L1$ into their corresponding data blocks. (a) Array $A[0:8,0:4]$. (b) Array $B[1:4,2:5]$. (c) Array $C[0:4,0:4]$.

processor, the partitioned data and programs may be executed in a communication-free fashion.

Given an n-nested loop L, the problem is how to partition the data referenced in loop L such that not only the communication overhead is not necessary but also the degree of parallelism can be extracted as large as possible. We first analyze the relations among all array variables of loop L. The iteration space is then partitioned into iteration blocks such that no interblock communication exists. For each partitioned iteration block, all data, referenced by those iterations, must be grouped into their corresponding data block for each array. On the other hand, a partitioned iteration block and the corresponding partitioned data block of each array must be allocated to the same processor so that no data transfer during parallel execution is incurred. Our methods proposed in this paper can make the size of partitioned iteration blocks as small as possible to achieve a higher degree of parallelism.

From the definition of a vector space, an n-dimensional vector space V over $\mathbf{R}$ can be generated using exactly n linearly independent vectors. Let X be a set of p linearly independent vectors, where $p \leq n$. These p vectors form a basis of a p-dimensional subspace, denoted by span(X), of V over $\mathbf{R}$. The dimension of a vector space V is denoted by dim(V). In the following, a formal definition of partition of iteration space is given.

Definition 2 [Iteration Partition]: The *iteration partition* of an n-nested loop L partitioned by the space $\Psi = \text{span}(\{\bar{t}_1, \bar{t}_2, \cdots, \bar{t}_u\})$, where $\bar{t}_l \in \mathbf{R}^n$, $1 \leq l \leq u$, denoted as $P_\Psi(I^n)$, is to partition the iteration space I^n into disjoint iteration blocks $B_1, B_2, \cdots, B_q$, where q is the total number of partitioned blocks. For each iteration block B_j, there exists a base point $\bar{b}_j \in \mathbf{R}^n$ and the following:

$$B_j = \{\bar{i} \in I^n | \bar{i} = \bar{b}_j + a_1\bar{t}_1 + a_2\bar{t}_2 + \cdots + a_u\bar{t}_u, a_l \in \mathbf{R}, 1 \leq l \leq u\},$$

for $1 \leq j \leq q$, where:

$$I^n = \bigcup_{1 \leq j \leq q} B_j. \qquad \square$$

Note that if $\dim(\Psi) = n$, there exists only one iteration block, the whole iteration space I^n, while we apply the iteration partition $P_\Psi(I^n)$ to loop L. If $\dim(\Psi) = 0$, one iteration is an iteration block while we apply the iteration partition $P_\Psi(I^n)$ to loop L.

Definition 3 [Data Partition]: Given an iteration partition $P_\Psi(I^n)$, the *data partition* of array A with all s referenced array variables $A[H_A\bar{i} + \bar{c}_1]$, $\cdots$, $A[H_A\bar{i} + \bar{c}_s]$, denoted as $P_\Psi(A)$, is the partition of data space of array A into q data blocks $B_1^A, B_2^A, \cdots, B_q^A$. For each data block B_j^A corresponding to one iteration block B_j of $P_\Psi(I^n)$ for $1 \leq j \leq q$, there exists the following condition:

$$B_j^A = \{A[\bar{a}] | \bar{a} = H_A\bar{i} + \bar{c}_l, \bar{i} \in B_j, 1 \leq l \leq s\}. \qquad \square$$

Consider Example 1. If $\Psi = \text{span}(\{(1,1)\})$ is chosen as the space of the iteration partition $P_\Psi(I^2)$ in loop $L1$, the iteration space can be partitioned into seven iteration blocks as shown in Fig. 3. Points enclosed by a line form an iteration block, and the dotted points represent the base points of the corresponding iteration blocks. For example, the base point $\bar{b}_5$ of iteration block $B_5 = \{\bar{i} \in I^2 | \bar{i} = \bar{b}_5 + a(1,1), 0 \leq a \leq 2\}$ is $(2,1)$. Based on the iteration partition $P_\Psi(I^2)$, the arrays A, B, and C are partitioned into the corresponding data blocks by using the respective data partition $P_\Psi(A)$, $P_\Psi(B)$, and $P_\Psi(C)$, as shown in Fig. 2.

Example 2: Consider a two-nested loop $L2$.

```
for i= 1 to 4
  for j = 1 to 4
      S1: A[i + j, i + j] := B[2i, j] * A[i + j - 1, i + j] ;
```

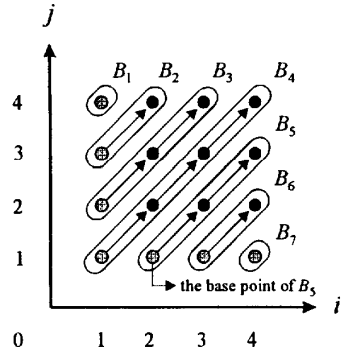


Fig. 3. Partitioning the iteration space of loop $L1$ into the corresponding iteration blocks.

S_2: $A[i+j-1, i+j-1] := B[2i-1, j-1]/3;$ $(L2)$
end
end

In loop $L2$, the following are the respective reference functions of arrays A and B:

$$H_A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \text{and } H_B = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}.$$

The data-referenced vectors $\bar{r}_1$, between $A[i+j, i+j]$ and $A[i+j-1, i+j-1]$, $\bar{r}_2$, between $A[i+j-1, i+j-1]$ and $A[i+j-1, i+j]$, and $\bar{r}_3$, between $A[i+j-1, i+j]$ and $A[i+j, i+j]$, of array A are $(1,1)$, $(0,-1)$, and $(-1,0)$, respectively. The data-referenced vector $\bar{r}_4$ of array B is $(1,1)$. Consider the equation $H_A\bar{t}_2 = \bar{r}_2$. Two iterations $\bar{i}_1$ and $\bar{i}_2$ can access the same element of array A if the equation $\bar{i}_2 - \bar{i}_1 = \bar{t}_2$ is satisfied. Because there exists no solution in the equation $H_A\bar{t}_2 = \bar{r}_2$, there exists no data dependence between $A[i+j-1, i+j-1]$ and $A[i+j-1, i+j]$. However, solving the equation $H_B\bar{t}_4 = \bar{r}_4$ can exactly obtain a solution $\bar{t}_4 = (\frac{1}{2}, 1)$. It is impossible for the data dependence vector $\bar{t}_4$ between two iterations, because $\bar{t}_4$ does not belong to $\mathbf{Z}^2$. Also there exists no data dependence on array B. Let the symbol $\mathbf{0}^d \in \mathbf{Z}^d$ be denoted as a zero-vector where each component is equal to 0. Consider the equation $H\bar{t} = \bar{r}$. In the special case that $\bar{r} = \mathbf{0}^d$, the set of solutions $\bar{t}$ of equation $H\bar{t} = \mathbf{0}^d$ is Ker(H), the null space of H. The vector $\bar{t}$ indicates the difference of two iterations accessing the same element of a certain array variable. For example, Ker(H_A) is span($\{(1,-1)\}$) in loop $L2$. On variable $A[i+j, i+j]$, the array element $A[4,4]$, referenced by the iteration $(1,3)$, can be referenced again by iterations $(1,3)$ + span($\{(1,-1)\}$), i.e., $(2,2)$ and $(3,1)$, of loop $L2$. □

In the following, we discuss how to choose the better space to partition the iteration space and data spaces without duplicate data such that there exists no interblock communication and parallelism is extracted as large as possible. The following definition is given for discussing the dependency among elements of an array.

Definition 4 [Reference Space]: In an n-nested loop L, if a reference function H_A and s variables $A[H_A\bar{i} + \bar{c}_1]$, $\cdots$, $A[H_A\bar{i} + \bar{c}_s]$ for array A exist, the data-referenced vectors are $\bar{r}_p = \bar{c}_j - \bar{c}_k$ for all $1 \le j < k \le s$ and $1 \le p \le \frac{s(s-1)}{2}$, then the *reference space* of array A is as follows:

$$\Psi_A = \text{span}(\beta \cup \{\bar{t}_1, \bar{t}_2, \cdots, \bar{t}_{\frac{s(s-1)}{2}}\}),$$

where β is the basis of Ker(H_A), and $\bar{t}_j \in \mathbf{R}^n$, $1 \le j \le \frac{s(s-1)}{2}$, must satisfy the following conditions:

1) $\bar{t}_j$ is a particular solution of equation $H_A\bar{t} = \bar{r}_j$.
2) A solution $\bar{t}' \in \bar{t}_j +$ Ker(H_A) exists such that $\bar{t}' \in \mathbf{Z}^n$ and $\bar{t}' = \bar{i}_2 - \bar{i}_1$ where $\bar{i}_1, \bar{i}_2 \in I^n$.

□.

The *reference space* used here is similar to the *group-temporal reuse vector space* previously defined by Wolf and Lam [21]. The *reference space* represents the relations of all data references between iterations. For array A, there exists no data dependence between iteration blocks when the iteration space I^n is partitioned with the *reference space* Ψ_A. This is because all data dependences are considered in Ψ_A such that data accesses are not needed between iteration blocks. In each iteration block, iterations according to the lexicographical order are executed to preserve the dependency in the loop.

Consider Example 2. In loop $L2$, the *reference space* Ψ_A of array A is span($\{(1,-1), (\frac{1}{2}, \frac{1}{2})\}$), because Ker($H_A$) = span($\{(1,-1)\}$) and there exists a particular solution $\bar{t}_1 = (\frac{1}{2}, \frac{1}{2})$ of equation $H_A\bar{t} = \bar{r}_1$ that satisfies conditions (1) and (2) of Definition 4. The *reference space* Ψ_B of array B is span(ϕ), because Ker(H_B) = $\{\mathbf{0}^2\}$, and the only solution $\bar{t}_4 = (\frac{1}{2}, 1) \notin \mathbf{Z}^2$ that does not satisfy condition (2) of Definition 4.

In the above discussions, only the communication-free iteration partition $P_{\Psi_A}(I^n)$ and data partition $P_{\Psi_A}(A)$ of an array A are considered in the nested loop. Among these iteration blocks, several data references may, however, exist in the other arrays. Whenever partitioning the iteration space, not only the data references that occur in an array must be considered but also the other data references that occur among arrays in a nested loop. Given an n-nested loop L with k array variables, let the *reference space* Ψ_{A_j} be span(X_j) of array A_j for $1 \le j \le k$. Then Ψ = span($X_1 \cup X_2 \cup \cdots \cup X_k$) is the *partitioning space* for communication-free partition of arrays in loop L without duplicate data. All iteration blocks partitioned by $P_\Psi(I^n)$ can be correctly executed in parallel. This is proven in Theorem 1 of the Appendix.

By Theorem 1, when dim(Ψ) < n, this means that there exists parallelism in loop L for the iteration partition $P_\Psi(I^n)$. By Definition 2, the smaller the value of dim(Ψ) is, the higher the degree of parallelism has. In general, when dim(Ψ) < $n-1$, our method can exploit more parallelism than Ramanaujam and Sadayappan's method [18] in For-all loops with uniformly generated references. This is because Ramanaujam and Sadayappan's method uses only $(n-1)$-dimensional hyperplanes to partition the arrays in For-all loops. To illustrate the communication-free array partitioning without duplicate data, consider the following example. In Example 1, the *reference spaces* are $\Psi_A = \Psi_C$ = span($\{(1,1)\}$), and $\Psi_B = \{\mathbf{0}^2\}$ for respective arrays A, C, and B. Therefore, by Theorem 1, the *partitioning space* is Ψ = span($\{(1,1)\} \cup \{(1,1)\} \cup \phi$) for communication-free iteration partition $P_\Psi(I^2)$ of loop $L1$.

Because $\dim(\Psi) = 1 (< 2)$, there exists a large amount of parallelism in loop $L1$. The overall results of partitioned data and iteration blocks in loop $L1$ have been shown in Fig. 2 and Fig. 3, respectively. Because loop $L1$ is not a For-all loop, Ramanaujam and Sadayappan's method cannot solve it in parallel execution.

The strategy allowing nonduplicate data for communication-free array partitioning in nested loops has been described in this subsection. Allowing duplicate data for some array elements can actually make it possible that several loops may exist with a great amount of parallelism for communication-free array partitioning. The communication-free array partitioning by duplicate data strategy for extracting more parallelism than the nonduplicate one is discussed in the next subsection.

B. Communication-Free Array Partitioning with Duplicate Data

In this subsection, we consider the communication-free array partitioning with duplicate data; i.e., there may exist more than one copy of an array element allocated onto local memory of processors. Because of communication overhead being most time-consuming in parallel executing programs, it is worthwhile to duplicate referenced data onto processors such that a high degree of parallelism can be exploited; meanwhile, the computations should be correctly performed in a communication-free fashion. Duplicate data strategy, in comparison with nonduplicate one, may extract more parallelism of programs based on communication-free array partitioning. Before describing our duplicate data strategy, we shall give the following definition for data arrays.

Definition 5 [Fully and Partially Duplicable Arrays]: If there exists no flow dependence on an array A, then the array A is called a *fully duplicable array*; otherwise, the array A is called a *partially duplicable array*. □

Note that the fully duplicable arrays may incur antidependence, output, or input dependence; however, the partially duplicable arrays can incur flow dependence. For the two kinds of arrays, we next discuss how to choose the better space to partition the iteration space and arrays with duplicate data such that there exists no interblock communication.

First, we examine the fully duplicable arrays in loop L. Because there exists no flow dependence on array A, any iteration will not use the elements of array A generated by other iterations; therefore, the data can be arbitrarily distributed onto each processor, and the original loop can be correctly executed in parallel. Therefore, the *reference space* Ψ_A can be reduced into $\text{span}(\phi)$ denoted as the *reduced reference space* Ψ_A^r. That is, Ψ_A^r is the subspace of Ψ_A.

Next the partially duplicable arrays are to be examined. Assume that there exist p flow dependences on a partially duplicable array A in loop L. The *reference space* Ψ_A of array A can be reduced into the *reduced reference space* $\Psi_A^r = (\beta \cup \{\bar{t}_1, \bar{t}_2, \cdots, \bar{t}_p\})$, where β is the basis of $\text{Ker}(H_A)$ and $\bar{t}_j$, $1 \leq j \leq p$, which lead to flow dependences, are particular solutions satisfying the conditions (1) and (2) in Definition 4. The reducible reason for the *reference space* is that only the flow dependences can actually cause the data transfer between execution of iterations. That is, only flow dependence is necessary to be considered during execution of programs; however, input, output dependences, and antidependence merely determine the precedence of executing iterations so that they cannot make any data transfer.

As for partitioning the iteration space, data references that occur among all arrays in a nested loop must be considered. Given an n-nested loop L with k array variables, let the *reduced reference space* $\Psi_{A_j}^r$ be $\text{span}(X_j^r)$ of each either fully or partially duplicable array A_j for $1 \leq j \leq k$. It is proven in Theorem 2 of the Appendix that $\Psi^r = \text{span}(X_1^r \cup X_2^r \cup \cdots \cup X_k^r)$ is the *partitioning space* for communication-free partitioning with duplicate data by using the iteration partition $P_{\Psi^r}(I^n)$.

To illustrate the communication-free array partitioning with duplicate data, consider the following two examples. First, Example 1 is considered again. The *reduced reference spaces* are $\Psi_A^r = \text{span}(\{(1,1)\})$ and $\Psi_B^r = \Psi_C^r = \text{span}(\phi)$ for respective arrays A, B, and C. Therefore, by Theorem 2, the *partitioning space* of loop $L1$ is $\Psi^r = \text{span}(\{(1,1)\} \cup \phi \cup \phi)$ for communication-free iteration partition $P_{\Psi^r}(I^2)$. For loop $L1$, the duplicate data strategy obtains the same results as the nonduplicate one. That is, loop $L1$ does not need to duplicate data for enhancing the parallelism.

Next consider Example 2. By Theorem 1, while we apply the iteration partition $P_\Psi(I^2)$ to loop $L2$, where $\Psi = \text{span}(\{(1,-1), (\frac{1}{2}, \frac{1}{2})\})$, loop $L2$ needs to be executed sequentially based on the nonduplicate data strategy. Because both arrays A and B in loop $L2$ are fully duplicable arrays, the *partitioning space* Ψ^r is $\text{span}(\phi)$ by Theorem 2. While we apply the iteration partition $P_{\Psi^r}(I^2)$ to loop $L2$, it can be executed fully in parallel.

Clearly, using duplicate data strategy can obtain more parallelism than using the nonduplicate one in loop $L2$. By duplicate data strategy, the overall results of partitioned data and iteration blocks in loop $L2$ are shown in Fig. 4 and Fig. 5, respectively. Note that the data blocks $B_{i,j}^A$ and $B_{i,j}^B$ and iteration block $B_{i,j}$, where $1 \leq i, j \leq 4$, will be assigned to the same processor.

However, several redundant computations possibly exist in the nested loops such that the *partitioning spaces* proven in Theorem 1 and Theorem 2 cannot achieve *minimum*; namely, the dimensions of *partitioning spaces* are not *minimal*. Given some sufficient conditions, the *minimal partitioning spaces* can be obtained. The details are discussed in the next subsection.

C. Eliminating Redundant Computations

Suppose each computation of nested loops is meaningful for programmers. However, there still possibly exist several redundant computations in programs. If two computations can generate values to an identical array element and the array element referenced by the first computation cannot be referenced until the second computation is executed, then the first computation is *redundant*. Thus, eliminating the redundant computations cannot affect the final results after executing a nested loop. For simplicity, we assume that the reference function H_A of each array A in an n-nested loop L is nonsingular, i.e., $\text{Ker}(H_A) = \{\mathbf{0}^n\}$.

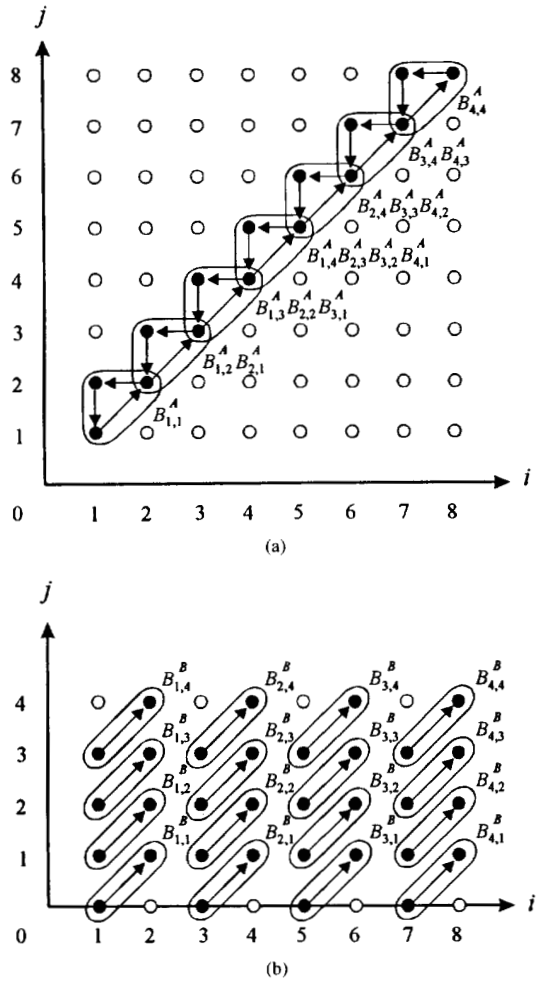


Fig. 4. Partition of arrays A and B in loop $L2$ using the data partition $P_{\Psi r}(A)$ and $P_{\Psi r}(B)$, respectively. (a) Partition of data space of array $A[1:8, 1:8]$. (b) Partition of data space of array $B[1:8, 0:4]$.

In this subsection, the main goals are stated as follows. First, all redundant computations in a nested loop can be eliminated by our proposed approach. Next the *minimal partitioning spaces* can be derived for communication-free partitioning without and with duplicate data after eliminating those redundant computations from the nested loop. The following definition is first given for describing the reference relationship among elements of an array.

Definition 6 [Data Reference Graph]: For an array A in loop L, a directed *data reference graph* is defined as $G^A = (V^A, E^A)$. The set of vertices, $V^A = W^A \cup R^A$, consists of the sets W^A and R^A of referenced array variables appearing in the respective left-hand side (performed by write operations to these variables) and right-hand side (performed by read operations from these variables) of the assignment statements. The set of edges, E^A, is the set of data dependences between two referenced array variables. □

Let the sets W^A and R^A in V^A consist of vertices w_i, $1 \le i \le m$, and r_j, $1 \le j \le v$, respectively. Because of

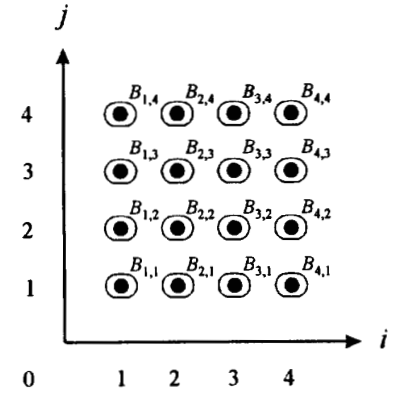


Fig. 5. Partition of iteration space of loop $L2$ using the iteration partition $P_{\Psi r}(I^2)$.

data dependences with transitive relation, if data dependences exist between any two referenced array variables, then the connection of G^A can be connected as:

1) the edges (w_i, w_j) with output dependences for all $1 \le i < j \le m$,
2) the edges (r_i, r_j) with input dependences for all $1 \le i < j \le v$,
3) the edges (w_1, r_j), (w_2, r_j), $\cdots$, (w_{τ_j}, r_j) with flow dependences, and
4) the edges (r_j, w_{τ_j+1}), (r_j, w_{τ_j+2}), $\cdots$, (r_j, w_m) with antidependences, $0 \le \tau_j \le m$, for each vertex $r_j \in R^A$, $1 \le j \le v$.

In general, if all of the above four types of connections appear in array A, the data reference graph G^A of array A will be as shown in Fig. 6.

Example 3: Consider a two-nested loop $L3$.

```
for i = 1 to 4
  for j = 1 to 4
    S1: A[i, j] := A[i − 1, j − 1] * 3 ;
    S2: A[i, j − 1] := A[i + 1, j − 2]/7 ;   (L3)
  end
end
```

The set of vertices for array A is $V^A = W^A \cup R^A = \{w_1, w_2\} \cup \{r_1, r_2\}$, where the vertices w_1, w_2, r_1, and r_2 are the variables $A[i, j]$, $A[i, j-1]$, $A[i+1, j-2]$, and $A[i-1, j-1]$, respectively. There are edges (w_1, w_2) with output dependence δ^o, and (r_1, r_2) with input dependence δ^i. For vertex r_1, $\tau_1 = 0$, so that there are edges (r_1, w_1) and (r_1, w_2) with antidependences δ^a. For vertex r_2, $\tau_2 = 2$, so that there are edges (w_1, r_2) and (w_2, r_2) with flow dependences δ^f. As a consequence, the data reference graph G^A of array A for loop $L3$ is shown in Fig. 7. □

In what follows, we describe what redundant computations are. An array element that is in the left-hand side of a computation is called a *write*. Two cases of redundant computations can be identified. First, for any two contiguous *writes* writing to the same identical array element of a nested loop, if the first *write* is not read by any computation until the second *write* is performed, then the first *write* is a *redundant write*. Next, if the

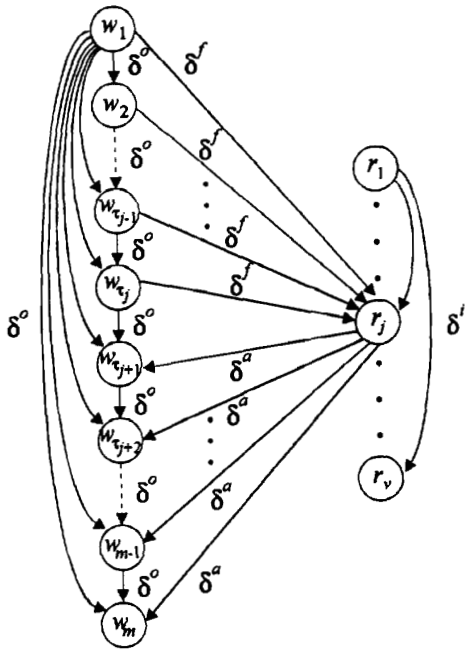


Fig. 6. Data reference graph G^A of array A for loop L.

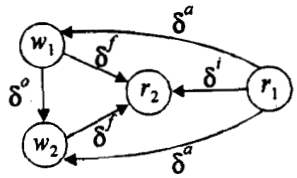


Fig. 7. Data reference graph G^A of array A for loop $L3$.

first *write* is read only by the redundant computations until the second *write* is performed, the first *write* is also a *redundant write*. Thus, any computation $S(\bar{i})$ for $\bar{i} \in I^n$ is a redundant computation if it will generate a *redundant write*. For clearly illustrating the redundant computation, consider the following example that we substitute the two statements S_1 and S_2 of loop $L3$ into the following four statements:

S_1': $A[i,j] := C[i,j] * 3$;
S_2': $B[i,j] := A[i,j-1]/D$;
S_3': $A[i-1,j-1] := E[i,j-1]/F + 11$;
S_4': $B[i,j-1] := G * 5 - K$;

For the first case, $S_2'(\bar{i}_1)$ for $\bar{i}_1 = (2,2)$ is a redundant computation. This is because the array element $B[2,2]$ written by the computation $S_2'(\bar{i}_1)$ is immediately overwritten by the computation $S_4'(\bar{i}_2)$ for $\bar{i}_2 = (2,3)$ in the next iteration, without being read during these two computations. For the next case, the array element $A[2,1]$ written by the computation $S_1'(\bar{i}_3)$ for $\bar{i}_3 = (2,1)$ is read only by the computation $S_2'(\bar{i}_1)$ until it is overwritten by $S_3'(\bar{i}_4)$ for $\bar{i}_4 = (3,2)$. Because $S_2'(\bar{i}_1)$ is a redundant computation from the above analysis, $S_1'(\bar{i}_3)$ is also a redundant computation.

We formalize the method of eliminating redundant computations of the nested loop L based on the above described concept. Assume that there are α assignment statements S_j for $1 \le j \le \alpha$ in the nested loop L. By Definition 6, we assume that vertex w_k in W^A corresponds to statement S_k, $1 \le k \le m$, and that r_j in R^A corresponds to statement S_{x_j}, $1 \le x_j \le \alpha$ and $1 \le j \le v$, for G^A of array A in the nested loop L. Assume that two contiguous *writes* are generated by the respective computations $S_k(\bar{i})$ and $S_p(\bar{i}+\bar{t})$ for $\bar{i} \in I^n$. That is, $S_p(\bar{i}+\bar{t})$ is output-dependent on $S_k(\bar{i})$, with the output dependence vector $\bar{t}$ contributed by $(w_k, w_p) \in E^A$ and p is the smallest index between $k+1$ and m. Then the computation $S_k(\bar{i})$ is a *redundant computation* if one of the following two cases is held.

Case 1: The data generated by $S_k(\bar{i})$ are not read by any computation until the computation $S_p(\bar{i}+\bar{t})$ is executed.

Case 2: The data generated by $S_k(\bar{i})$ are read only by the *redundant computations* $S_{x_j}(\bar{i}+\bar{t}_j)$, which are flow-dependent on $S_k(\bar{i})$, with the respective flow dependence vectors $\bar{t}_j$ contributed by $(w_k, r_j) \in E^A$, $1 \le j \le v$, until the computation $S_p(\bar{i}+\bar{t})$ is executed.

Obviously, all redundant computations in the nested loop L can be recursively examined by the above two cases. Let the set $N(S_k) = \{\bar{i} \in I^n \mid S_k(\bar{i})$ is not redundant computation$\}$ be the set of all iterations without redundant computations on statement S_k, $1 \le k \le \alpha$. For example, considering Example 3, the sets $N(S_1) = \{(i,4) \mid 1 \le i \le 4\}$ and $N(S_2) = \{(i,j) \mid 1 \le i,j \le 4\}$ of all iterations without redundant computations on the respective statements S_1 and S_2 can be derived from the method of eliminating redundant computations.

After eliminating redundant computations, several data dependences possibly can be deleted in the nested loops. Let the symbol $Val(A[H_A\bar{i}+\bar{c}], S_k)$ denote the set $\{A[H_A\bar{i}+\bar{c}] \mid \bar{i} \in N(S_k)\}$, where the array variable $A[H_A\bar{i}+\bar{c}]$ appears in statement S_k. That is, these nonredundant computations on statement S_k with the array variable $A[H_A\bar{i}+\bar{c}]$ actually need to access the set of array elements $Val(A[H_A\bar{i}+\bar{c}], S_k)$. Therefore, the data dependence, corresponding to an edge (a,b) in E^A, is a *false data dependence* if $Val(a,S) \cap Val(b,S') = \phi$. That is, those redundant computations can result in the false data dependence between two vertices a and b. In contrast, a *useful data dependence*, not a false data dependence, can actually make dependence relations between variables in the nested loop L. After eliminating the redundant computations and the false data dependences from the nested loop L, the degree of parallelism can be increased. Based on the above analysis, considering Example 3, since $Val(w_1, S_1) \cap Val(r_2, S_1) = \phi$, the flow dependence (w_1, r_2) in E^A is a false flow dependence. This is because the computations of the array elements that are generated by w_1 ($A[i,j]$) and then used in r_2 ($A[i-1,j-1]$) are redundant on statement S_1. Similarly, the output dependence (w_1, w_2), the antidependence (r_1, w_1), and the input dependence (r_1, r_2) are all false. Thus, the useful data dependences contain only the flow dependence (w_2, r_2) contributing the flow dependence vector $\bar{t}_1 = (1,0)$, and the antidependence (r_1, w_2) contributing the antidependence vector $\bar{t}_2 = (1,-1)$.

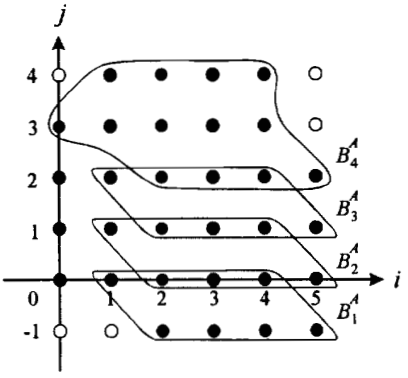


Fig. 8. Partition of array A in loop $L3$ using the data partition $P_{\Psi^{min^r}}(A)$.

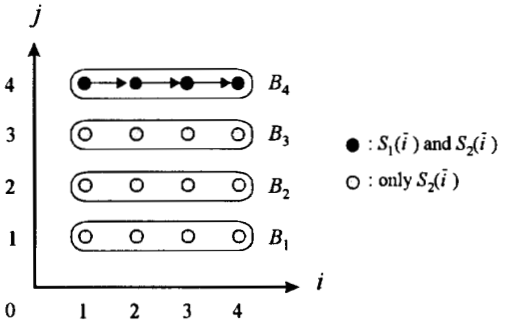


Fig. 9. Partition of iteration space of loop $L3$ using the iteration partition $P_{\Psi^{min^r}}(I^2)$.

In the following, the communication-free partitioning strategies with *minimal partitioning spaces* are discussed. Since only useful data dependences can actually cause data transfer between iterations, the space $\Psi_A^{min} = \text{span}(\{\bar{t} \mid \bar{t}$ is contributed by a useful data dependence$\})$ is the *minimal reference space* of array A without duplicate data. This is because if any vector $\bar{t} \in X$, where $\Psi_A^{min} = \text{span}(X)$ is removed to form $\Psi' = \text{span}(X - \{\bar{t}\})$ such that $\Psi' \neq \Psi_A^{min}$, then the vector $\bar{t}$, which can lead to a useful data dependence, must exist between two iterations that are allocated on different iteration blocks such that communication-free partitioning cannot be achieved by using the iteration partition $P_{\Psi'}(I^n)$. Only the data accessed by the nonredundant computations must be considered here to form the data partition $P_{\Psi_A^{min}}(A)$.

Given an n-nested loop L with k array variables, let the *minimal reference space* $\Psi_{A_j}^{min}$ be span(X_j) of each array A_j for $1 \leq j \leq k$. It is proven in Theorem 3 of the Appendix that the *partitioning space* $\Psi^{min} = \text{span}(X_1 \cup X_2 \cup \cdots \cup X_k)$ for communication-free partitioning of arrays A_j, $1 \leq j \leq k$, without duplicate data is *minimal*.

Similarly, since only useful flow dependences can actually cause data transfer between iterations, the space $\Psi_A^{min^r} = \text{span}(\{\bar{t} \mid \bar{t}$ is contributed by a useful flow dependence$\})$ is the *minimal reduced reference space* of array A with duplicate data. Given an n-nested loop L with k array variables, let the *minimal reduced reference space* $\Psi_{A_j}^{min^r}$ be span(X_j^r) of each array A_j for $1 \leq j \leq k$. It is proven in Theorem 4 of the Appendix that the *partitioning space* $\Psi^{min^r} = \text{span}(X_1^r \cup X_2^r \cup \cdots \cup X_k^r)$ for communication-free partitioning of arrays A_j, $1 \leq j \leq k$, with duplicate data is *minimal*.

Example 3 is considered again for illustrating how to obtain the *minimal partitioning spaces* for communication-free partitioning without and with duplicate data. For communication-free partitioning without duplicate data of loop $L3$, by Theorem 3, the *minimal partitioning space* is $\Psi^{min} = \text{span}(\{\bar{t}_1, \bar{t}_2\}) = \text{span}(\{(1,0), (1,-1)\})$. Thus, loop $L3$ must be executed sequentially. The *minimal partitioning space* of loop $L3$ with duplicate data is $\Psi^{min^r} = \text{span}(\{\bar{t}_1\}) = \text{span}(\{(1,0)\})$; therefore, the overall results of partitioned data and iteration blocks of loop $L3$ are shown in Fig. 8 and Fig. 9, respectively.

In Fig. 9, both the computations $S_1(\bar{i})$ and $S_2(\bar{i})$ executed at iteration $\bar{i}$ are denoted by a solid point. Only computation $S_2(\bar{i})$ executed at $\bar{i}$ is denoted by a dotted point. Because the *partitioning space* Ψ^r of loop $L3$ is span$(\{(1,0),(1,1)\})$, it must also be executed sequentially under the duplicate data strategy. Therefore, removing the redundant computations not only can reduce the computation time but also can increase the degree of parallelism.

For obtaining the *minimal partitioning spaces*, the approach of removing redundant computations, designed in parallel compilers, is complex and more time-consuming. The trade-off depends on whether users need to obtain large amounts of parallelism for some particular programs. Suppose there does not exist any redundant computation in a program. Then the *partitioning spaces* proven in Theorems 1 and 2 can achieve *minimum* for communication-free partitioning of loop L with nonduplicate and duplicate data, respectively.

The sufficient conditions for communication-free array partitioning have been discussed and derived here without duplicate data corresponding to Theorems 1 and 3 and with duplicate data corresponding to Theorems 2 and 4. However, transforming the partitioned iterations on the basis of the above two schemes into a parallel execution form is important in designing parallel compilers. In practical applications, the number of processors is fixed on multicomputers. How to map the partitioned program onto a fixed-size multicomputers in the consideration of load balancing must be considered if the number of partitioned iteration blocks is larger than the number of processors. The problems of program transformation and processor assignment are discussed in the next section.

IV. Program Transformation and Processor Assignment

In this section, the program transformation of a partitioned nested loop without regard to the number of processors is first described, and the processor assignment with a fixed-size number of processors is then discussed. Mapping a parallel code onto processors may cause inefficiency if the parallelizing compiler cannot generate an appropriate parallelized program of the parallel code. Automatically transforming the partitioned nested loop into a parallel execution form therefore becomes the focus.

By Theorems $1-4$, we can obtain an n-nested loop with the *partitioning space* $\Psi = \text{span}(X)$, where X consists of g linearly independent vectors; i.e., $\dim(\Psi) = g$. In the

following, the partitioned nested loop is to be transformed into a parallel execution form with $k\ (=n-g)$ **forall** loops as using the iteration partition $P_\Psi(I^n)$. By *orthogonal projection* [3], each iteration block can be projected to the subspace Ker(Ψ). This implies that the basis Q of Ker(Ψ) can therefore be used to represent each transformed point of k **forall** loops; namely, each point indicates one partitioned iteration block. Since dim(Ker(Ψ)) $= k$ and dim(Ψ) $= g$, there are k outermost **forall** loop index variables and g innermost loop index variables, respectively, in the transformed nested loop.

First, we derive the basis $Q = \{\bar{a}_i = (a_{i,1}, a_{i,2}, \cdots, a_{i,n}) \in \mathbf{Z}^n \mid \gcd(a_{i,1}, a_{i,2}, \cdots, a_{i,n}) = 1,\ 1 \le i \le k\}$ of Ker(Ψ). The elementary row operations are next used on the matrix $\begin{bmatrix} \bar{a}_1 \\ \vdots \\ \bar{a}_k \end{bmatrix}_{k\times n}$ to derive the row echelon form $\begin{bmatrix} \bar{a}'_1 \\ \vdots \\ \bar{a}'_k \end{bmatrix}_{k\times n}$, where $\bar{a}'_j$ is derived from $\bar{a}_i$, $j = \sigma(i)$, and the function $\sigma : i \to j$ is a permutation for $1 \le i, j \le k$ [3]. The first position of nonzero component of $\bar{a}'_j$ is y_j for $1 \le j \le k$, and $y_j < y_{j+1}$ for $1 \le j < k$. The new index variables $I'_1, I'_2, \cdots, I'_n$ can be obtained from the original loop index variables $I_1, I_2, \cdots, I_n$ with the following equation:

$$\begin{aligned} &I'_i = a_{\sigma^{-1}(j),1}I_1 + a_{\sigma^{-1}(j),2}I_2 + \cdots \\ &\qquad + a_{\sigma^{-1}(j),n}I_n, \\ &\text{if } i = y_j \text{ for } 1 \le i \le n \text{ and } 1 \le j \le k, \\ &I'_i = I_i, \\ &\qquad \text{if } i \ne y_j \text{ for } 1 \le i \le n \text{ and } 1 \le j \le k. \end{aligned} \tag{1}$$

The inverse relations of (1) are derived as follows:

$$\begin{aligned} &I_i = b_{j,1}I'_1 + b_{j,2}I'_2 + \cdots + b_{j,n}I'_n, \\ &\qquad \text{if } i = y_j \text{ for } 1 \le i \le n \text{ and } 1 \le j \le k, \\ &\qquad \text{where } b_{j,l} \in \mathbf{R} \text{ for } 1 \le l \le n; \\ &I_i = I'_i, \\ &\qquad \text{if } i \ne y_j \text{ for } 1 \le i \le n \text{ and } 1 \le j \le k. \end{aligned} \tag{2}$$

The lower bounds l'_j and the upper bounds u'_j of k outermost **forall** loop index variables I'_{y_j}, $1 \le j \le k$, can be calculated on the basis of the the ranges of original loop index variables and (1) and (2). In order to execute all iterations within one iteration block according to the lexicographical order, the first g index variables I_{z_i} for $1 \le z_i \le n$ and $1 \le i \le g$ are chosen as the innermost loop indexes. The I_{z_i} cannot be expressed linearly by the variables $I'_{y_1}, \cdots, I'_{y_k}, I_{z_1}, \cdots, I_{z_{i-1}}$ for $1 \le i \le g$, and $z_j < z_{j+1}$ for $1 \le j < g$.

The main reason for such a selection for the g index variables is to make a one-to-one mapping from the original iteration space to the new transformed space. Similarly, the respective lower bounds l'_j and upper bounds u'_j, $k+1 \le j \le n$, of the innermost loop index variables I_{z_i}, $1 \le i \le g$, can be derived. All of the above lower and upper bounds can be determined by the method of transforming the loop bound proposed in [22]. Besides, in order to determine the value of original loop index variables, except for g innermost loop index variables, the *extended statements* can be derived by (1) and (2) as follows:

$$I_i = c_{i,1}I'_{y_1} + \cdots + c_{i,k}I'_{y_k} + c_{i,k+1}I_1 + \cdots + c_{i,k+i-1}I_{i-1},$$

where $i \ne z_j$ and $c_{i,l} \in \mathbf{R}$, $1 \le l \le k+i-1$, for $1 \le i \le n$ and $1 \le j \le g$.

Based on the above transformation, the whole transformed loop L' is as follows:

```
forall I'_{y1} = l'_1 to u'_1
  forall I'_{y2} = l'_2 to u'_2
    :
    forall I'_{yk} = l'_k to u'_k
      for I_{z1} = l'_{k+1} to u'_{k+1}
        :
        for I_{zg} = l'_n to u'_n
          [modified loop body]
        end
        :
      end
    end-forall
    :
  end-forall
end-forall.
```

How to transform a partitioned nested loop into a parallel execution form is to be illustrated with the following example.

Example 4: Consider a three-nested loop $L4$.

```
for i1 = 1 to 4
  for i2 = 1 to 4
    for i3 = 1 to 4
      A[i1, i2, i3] := A[i1-1, i2+1, i3-1]+B[i1, i2, i3]
;   (L4)
    end
  end
end
```

By applying any one of Theorems 1 – 4, the *minimal partitioning space* of loop $L4$ is $\Psi = \text{span}(\{(1,-1,1)\})$. That is, loop $L4$ does not necessarily duplicate data to enhance the parallelism. First, we can obtain that the basis of Ker(Ψ) is $Q = \{(a_1, a_2, a_3), (b_1, b_2, b_3)\}$, where $(a_1, a_2, a_3) = (1, 1, 0)$ and $(b_1, b_2, b_3) = (-1, 0, 1)$, so that $\gcd(a_1, a_2, a_3) = 1$ and $\gcd(b_1, b_2, b_3) = 1$.

Let the new index variables i'_1, i'_2, and i'_3 be $i'_1 = a_1i_1 + a_2i_2 + a_3i_3 = i_1 + i_2$, $i'_2 = b_1i_1 + b_2i_2 + b_3i_3 = -i_1 + i_3$, and $i'_3 = i_3$. The inverse relations are derived as $i_1 = -i'_2 + i'_3$, $i_2 = i'_1 + i'_2 - i'_3$, and $i_3 = i'_3$.

Since dim(Ker(Ψ)) $= 2$ and dim(Ψ) $= 1$, there are two outermost **forall** loop index variables i'_1 and i'_2 and one innermost loop index variable i_1, respectively. The following transformed loop $L4'$ can be obtained through usage of the above transformation strategy.

```
forall i'_1 = 2 to 8
  forall i'_2 = max(-3, -i'_1 + 2) to min(3, -i'_1 + 8)
    for i_1 = max(1, i'_1 - 4, -i'_2 + 1) to min(4, i'_1
              -1, -i'_2 + 4)
```

$E_1 : i_2 := i'_1 - i_1$;
$E_2 : i_3 := i'_2 + i_1$; $(L4')$
$A[i_1, i_2, i_3] := A[i_1 - 1, i_2 + 1, i_3 - 1]$
$+B[i_1, i_2, i_3]$;
end
end-forall
end-forall

The statements E_1 and E_2 in loop $L4'$ are the extended statements. Each point of the set $\{(i'_1, i'_2) \mid 2 \le i'_1 \le 8$ and $\max(-3, -i'_1 + 2) \le i'_2 \le \min(3, -i'_1 + 8)\}$ represents one iteration block, and all of the points can be independently executed in parallel without interblock communication. □

Although the number of processors is larger than the number of partitioned iteration blocks, each partitioned iteration block corresponding to one element of the set $\{(I'_{y_1}, I'_{y_2}, \cdots, I'_{y_k}) \mid l'_j \le I'_{y_j} \le u'_j$, for $1 \le j \le k\}$, and its partitioned data blocks of each array can be distributed into local memory of the corresponding processor. That is, the execution time is dominated by the partitioned iteration block with the maximum number of iterations. However, in the following, we discuss the assignment problem of iteration blocks on multicomputers with the fixed-size number of processors. The following example illustrates how to map the parallel execution form on multicomputers such that the workload of processors is as balanced as possible. Assume that there are four processors, $PE_{0,0}$, $PE_{0,1}$, $PE_{1,0}$, and $PE_{1,1}$. The transformed loop $L4'$ is considered. The following code can be obtained for parallel execution of processor PE_{a_1,a_2} for $0 \le a_1 \le p_1 - 1$ and $0 \le a_2 \le p_2 - 1$, where $p_1 = p_2 = 2$.

forall $i'_1 = (2 + (a_1 - (2 \bmod p_1)) \bmod p_1)$ to 8 step p_1
forall $i'_2 = (\max(-3, -i'_1+2) + (a_2 - (\max(-3, -i'_1+2) \bmod p_2)) \bmod p_2)$
to $\min(3, -i'_1 + 8)$ step p_2
for $i_1 = \max(1, i'_1 - 4, -i'_2 + 1)$ to $\min(4, i'_1 - 1, -i'_2 + 4)$
$E_1 : i_2 := i'_1 - i_1$;
$E_2 : i_3 := i'_2 + i_1$;
$A[i_1, i_2, i_3] := A[i_1 - 1, i_2 + 1, i_3 - 1]$
$+B[i_1, i_2, i_3]$;
end
end-forall
end-forall

The above processor assignment is shown in Fig. 10, where the value within each point represents the number of iterations within the corresponding partitioned iteration block. Around the dashed line with four points, the left-down, left-up, right-down, and right-up points are assigned to processors $PE_{0,0}$, $PE_{0,1}$, $PE_{1,0}$, and $PE_{1,1}$, respectively, according to the transformed loop $L4'$. By examining the above allocation, the workloads of the four processors are the same and equal to 16 iterations. The main reason for such allocation is that the neighboring iteration blocks of each partitioned iteration block through the iteration partition have almost the same number of iterations, except for the boundary-partitioned iteration blocks when the iteration space is very large. For example, four points around the dashed line in Fig. 10 have almost the same number

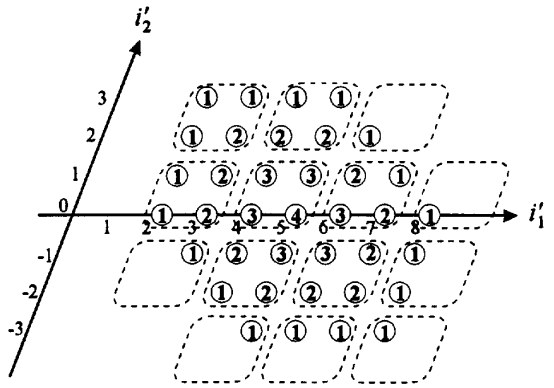


Fig. 10. Processor assignment of loop $L4'$.

of iterations. Hence, the balanced workload on each processor can possibly be obtained if the neighboring iteration blocks are distributed to their corresponding processors based on the mod operation.

Such a simple processor assignment strategy can be easily extended to the k dimensions of nested **forall** loops. Assume that the number of processors is $p = p_1 \times p_2 \times \cdots \times p_k$ for numbering the p processors. Let $p_i = \lfloor \sqrt[k]{p} \rfloor$ for $1 \le i \le k-1$, and let $p_k = \lfloor \frac{p}{(\lfloor \sqrt[k]{p} \rfloor)^{k-1}} \rfloor$. Then processor $PE_{a_1,a_2,\cdots,a_k}$, $0 \le a_1 \le p_1 - 1$, $0 \le a_2 \le p_2 - 1$, $\cdots$, $0 \le a_k \le p_k - 1$, is assigned to execute the following code, and the corresponding data need to be allocated onto its local memory.

forall $I'_{y_1} = (l'_1 + (a_1 - (l'_1 \bmod p_1)) \bmod p_1)$ to u'_1 step p_1
forall $I'_{y_2} = (l'_2 + (a_2 - (l'_2 \bmod p_2)) \bmod p_2)$ to u'_2 step p_2
⋮
forall $I'_{y_k} = (l'_k + (a_k - (l'_k \bmod p_k)) \bmod p_k)$ to u'_k step p_k
for $I_{z_1} = l'_{k+1}$ to u'_{k+1}
⋮
for $I_{z_g} = l'_n$ to u'_n
[modified loop body]
end
⋮
end
end-forall
⋮
end-forall
end-forall

Now we compare the performance of nonduplicate and duplicate data strategies. In distributed memory multicomputers, assume that the time required to perform one iteration is t_{comp}; the time required to communicate including two parts is t_{start}, the *startup* time for communication; and t_{comm} is the time required to transmit a single datum from one processor to the neighboring one. Therefore, the time required to transmit x data between two neighboring processors is $t_{start} + xt_{comm}$.

Consider the following matrix multiplication algorithm.

```
for i = 1 to M
  for j = 1 to M
    for k = 1 to M
      C[i,j] := C[i,j] + A[i,k] * B[k,j] ;   (L5)
    end
  end
end
```

The iteration space of loop $L5$ is $I^3 = \{(i,j,k) \mid 1 \leq i,j,k \leq M\}$. The *reference spaces* $\Psi_A = \text{span}(\{(0,1,0)\})$, $\Psi_B = \text{span}(\{(1,0,0)\})$, and $\Psi_C = \text{span}(\{(0,0,1)\})$ of respective arrays A, B, and C. By Theorem 1, the *partitioning space* Ψ is $\text{span}(\{(0,1,0)\} \cup \{(1,0,0)\} \cup \{(0,0,1)\})$. That is, the matrix multiplication algorithm needs to be executed sequentially, as when using the nonduplicate data strategy.

Consider a $p_1 \times p_2$–mesh multicomputer as the target machine. Therefore, the time complexity including the computation time and the communication time of allocating the whole arrays A and B from host processor to one node processor is computed as follows:

$$T_1 = O(M^3\, t_{comp} + 2(t_{start} + M^2\, t_{comm})).$$

Next considered is the fact that if only some of fully or partially duplicable arrays are duplicated, they may sacrifice less parallelism than all of them. Note that both arrays A and B are fully duplicable arrays, and that array C is a partially duplicable array. Thus, the *reduced reference spaces* $\Psi_A^r = \text{span}(\phi)$, $\Psi_B^r = \text{span}(\phi)$, and $\Psi_C^r = \text{span}(\{(0,0,1)\})$ for respective arrays A, B, and C. Demonstrated in the following is the fact that only the array B is duplicated in loop $L5$. Because of array A not replicating data, let $\Psi' = \text{span}(\{(0,1,0)\} \cup \{(0,0,1)\})$ such that the communication-free iteration partition $P_{\Psi'}(I^3)$ can be obtained. Assume that the number of processors on mesh is $p = p_1 \times p_2$, that $\sqrt{p} = p_1 = p_2$, and that M is a multiple of p. The processor PE_a for $0 \leq a \leq p-1$ will execute the following loop $L5'$ by the processor assignment strategy:

```
forall i' = (1 + (a − (1 mod p)) mod p) to M step p
    for j = 1 to M
      for k = 1 to M
      E1 : i := i' ;
        C[i,j] := C[i,j] + A[i,k] * B[k,j] ;   (L5')
      end
    end
end-forall
```

Because the index variable i' is equivalent to i, the extended statement E_1 can be eliminated by using the index variable i instead of i'. Because we do not replicate the data of array A to each processor, the whole array B must be duplicated to each processor for parallel execution without interprocessor communication.

We initially allocate the referenced elements of arrays A and B from host processor to each node processor on mesh. Because the processor PE_a, $0 \leq a \leq p-1$, requires accessing the following array elements:

$$A[\alpha, 1:M], \text{ for } \alpha = (1 + (a-1) \bmod p) + lp, l \in \mathbf{Z}, 1 \leq \alpha \leq M,$$

the host processor must send these data to the corresponding processor in a pipelined fashion. In addition, because all processors require accessing the same array elements,

$$B[1:M, 1:M],$$

the host processor must broadcast the whole array B to each node processor. Thus, the communication time complexities of distributing the initial referenced elements of arrays A and B are $O(p(t_{start} + \frac{M}{p} M\ t_{comm}))$ and $O(t_{start} + 2\sqrt{p}\, M^2\, t_{comm})$, respectively. Since there exists no communication among processors during execution, the computation time complexity is $O(\frac{M^3}{p}\ t_{comp})$. Therefore, the total time complexity including the computation and communication time under the duplication of array B is as follows:

$$T_2 = O(\frac{M^3}{p}\, t_{comp} + (p\, t_{start} + M^2\, t_{comm}) + (t_{start} + 2\sqrt{p}\, M^2\, t_{comm})).$$

Nevertheless, if only the array A, not array B, is duplicated, the similar discussions and the same total time complexity can be obtained.

In the following, both arrays A and B in loop $L5$ are to be duplicated. By Theorem 2, the communication-free iteration partition $P_{\Psi''}(I^3)$ can be obtained, where the *partitioning space* $\Psi'' = \text{span}(\{(0,0,1)\})$. By the processor assignment strategy, the following results can thus be obtained. The processor PE_{a_1,a_2} for $0 \leq a_1 \leq p_1 - 1$ and $0 \leq a_2 \leq p_2 - 1$ is to execute the following loop $L5''$:

```
forall i' = (1 + (a1 − (1 mod p1)) mod p1) to M step p1
    forall j' = (1 + (a2 − (1 mod p2)) mod p2) to M
step p2
        for k = 1 to M
        E1 : i := i' ;
        E2 : j := j' ;
          C[i,j] := C[i,j] + A[i,k] * B[k,j] ;   (L5'')
        end
    end-forall
end-forall
```

Assume that M is a multiple of $\sqrt{p}$. We initially allocate the referenced elements of arrays A and B from the host processor to each node processor on mesh. Because the processor PE_{a_1,a_2}, $0 \leq a_1 \leq \sqrt{p} - 1$, requires accessing the same array elements as follows:

$$A[\alpha, 1:M], \text{ for } \alpha = (1 + (a_2 - 1) \bmod \sqrt{p}) + l\sqrt{p}, l \in \mathbf{Z}, 1 \leq \alpha \leq M, \text{ and}$$

$0 \leq a_2 \leq \sqrt{p} - 1$, the host processor must send the same data to the corresponding row processors by multicasting in a pipelined fashion. Similarly, because the processor PE_{a_1,a_2},

$0 \le a_2 \le \sqrt{p} - 1$, requires accessing the same array elements

$$B[1 : M, \alpha], \text{ for } \alpha = (1 + (a_1 - 1) \bmod \sqrt{p}) + l\sqrt{p}, l \in \mathbf{Z}, 1 \le \alpha \le M,$$

and

$$0 \le a_1 \le \sqrt{p} - 1,$$

the host processor must send the same data to the corresponding column processors by multicasting in a pipelined fashion. Thus, distributing the initial referenced elements of arrays A and B in a pipelined fashion has the same communication time complexity $O(\sqrt{p}\ t_{start} + 2\sqrt{p}\ \frac{M^2}{\sqrt{p}}\ t_{comm})$. Because we replicate only the partial data of both arrays A and B to processors for loop $L5''$, the communication cost of distributing the initial data to each processor is less than that of loop $L5'$. Since there exists no communication between processors during execution, the computation time complexity is $O(\frac{M^3}{p}\ t_{comp})$. Therefore, the total time complexity under duplicate data strategy is as follows:

$$T_3 = O(\frac{M^3}{p}\ t_{comp} + 2(\sqrt{p}\, t_{start} + 2\, M^2\, t_{comm})).$$

The overall execution results for loops $L5$, $L5'$, and $L5''$ are executed on the Transputer multicomputer with 16 processors, as shown in Tables I and II. The execution time of loops $L5$, $L5'$, and $L5''$ are illustrated in Table I with problem sizes $M = 16, 32, 64, 128$, and 256. The speedup derived from Table I is illustrated in Table II. When the number of processors is equal to 1, we consider only the computation time, not including the time of allocating arrays A and B. Although duplicating data seems to waste the time of allocating initial data, it can increase great amounts of parallelism and incur no communication overhead during parallel execution of programs. Therefore, the time of parallel execution is less than that of sequential execution, as shown in Table I. However, because data locality in loop $L5$ is not exploited during sequential execution, the speedup becomes better and better whenever the problem size becomes larger and larger, as shown in Table II. This implies that exploiting data locality is also important during program execution in each processor [21]. Because of large existing amounts of communication overhead in loop $L5'$ while distributing whole array B, the speedup of loop $L5''$ is more efficient than that of loop $L5'$. By the above analysis, the communication time of distributing the initial referenced elements of arrays must be as small as possible in order to obtain better efficiency during parallel execution. In addition, determining which kind of duplication of array is suitable for replicating their referenced data can be appropriately estimated such that parallelized programs can gain better performance during parallel execution.

TABLE I
EXECUTION TIME OF LOOPS $L5$, $L5'$, AND $L5''$ (in s)

Number of processors	Loop	Problem size (M) 16	32	64	128	256
$p = 1$	$L5$	0.0399	0.3162	2.5241	20.1691	161.2546
$p = 4$	$L5'$	0.0144	0.0956	0.6961	5.2895	41.3058
	$L5''$	0.0127	0.0855	0.6467	5.1405	40.7988
$p = 16$	$L5'$	0.0135	0.0543	0.2869	1.7908	12.3584
	$L5''$	0.0080	0.0326	0.2043	1.4326	10.6513

TABLE II
SPEEDUP OF LOOPS $L5'$ AND $L5''$

Number of processors	Loop	Problem size (M) 16	32	64	128	256
$p = 4$	$L5'$	2.77	3.31	3.63	3.81	3.89
	$L5''$	3.14	3.70	3.90	3.92	3.95
$p = 16$	$L5'$	2.96	5.82	8.80	11.26	13.05
	$L5''$	4.99	9.70	12.35	14.08	15.14

V. CONCLUSION

Data distribution in the distributed memory multicomputers is of crucial importance for the efficiency of parallelized programs, because local memory accesses are much faster than those involving interprocessor communication. If no attention is paid to the data allocation problem, a large amount of time spent in data communication and synchronization may seriously undermine the benefits of parallelism. In order to reduce or even eliminate the interprocessor communication, it is important for parallelizing compilers to analyze the pattern of references among arrays of a nested loop and to determine how to allocate these data to local memory of processors.

Two automatic array partitioning strategies with nonduplicate and duplicate data have been proposed in this paper, such that no data transfer during parallel execution is incurred and the parallelism of nested loops can be exploited as large as possible. Under the duplicate data strategy, more parallelism can be extracted than for the nonduplicate one. Moreover, the *minimal partitioning spaces* with a high degree of parallelism can be obtained when we eliminate the redundant computations. A method for transforming the nested loop into a parallel execution form is also proposed on the basis of the two partitioning strategies. Finally, a method is proposed to distribute the parallelized program and the corresponding array elements into the fixed-size multicomputers under the consideration of load balancing. For the matrix multiplication algorithm, the performance of the strategies with nonduplicate and duplicate data is discussed, and the overall results are executed on the Transputer multicomputer with 16 processors. In addition, the communication-free partitioning strategies proposed in this paper can also prevent cache-thrashing problem in shared memory multiprocessor systems [14].

Although our compilation techniques consider each nested loop independently in a program, there still exist several

benefits for the communication-free data allocation strategies. First, the data arrays can be efficiently distributed to each processor in a pipelined fashion. Second, a large amount of *startup* time is reduced, because we transmit a large number of data arrays at a time. Finally, there exists no data synchronization during parallel execution. Currently, we are implementing the proposed data allocation strategies in our UPPER project: A User-interactive Parallel Programming EnviRonment (UPPER). In this project, the performance of several scientific programs, such as matrix multiplication, discrete Fourier transform, convolution, some basic linear algebra programs, and so forth, are evaluated under the cases with and without using the communication-free data allocation strategies.

APPENDIX

In this Appendix, we prove Theorems 1 through 4.

Theorem 1: Given an n-nested loop L with k array variables, let the *reference space* Ψ_{A_j} be span(X_j) of each array A_j for $1 \leq j \leq k$. If $\Psi = \text{span}(X_1 \cup X_2 \cup \cdots \cup X_k)$, then Ψ is the *partitioning space* for communication-free partitioning of arrays A_j for $1 \leq j \leq k$ without duplicate data by using the iteration partition $P_\Psi(I^n)$.

Proof: Theorem 1 of this Appendix shall be proved by induction as follows.

Basic $k = 1$: Clearly, $\Psi = \Psi_{A_1}$, which is correct because Ψ_{A_1} is the *partitioning space* for communication-free partitioning of array A_1 without duplicate data.

Induction hypothesis $k = x$: The space $\Psi_{1,x} = \text{span}(X_1 \cup X_2 \cup \cdots \cup X_x)$ is the *partitioning space* for communication-free partitioning of arrays A_j for $1 \leq j \leq x$ without duplicate data.

Induction $k = x + 1$: The space $\Psi_{1,x+1} = \text{span}(X_1 \cup X_2 \cup \cdots \cup X_{x+1})$ can be rewritten by the form $\Psi_{1,x+1} = \text{span}(X_{1,x} \cup X_{x+1})$, where $X_{1,x} = X_1 \cup X_2 \cup \cdots \cup X_x$. By induction hypothesis, $\Psi_{1,x} = \text{span}(X_{1,x})$ is the communication-free *partitioning space* of arrays A_j for $1 \leq j \leq x$. Moreover, $\Psi_{A_{x+1}} = \text{span}(X_{x+1})$ is the communication-free *partitioning space* of array A_{x+1}. In the following, we prove that $\Psi_{1,x+1}$ is the *partitioning space* of arrays A_j for $1 \leq j \leq x + 1$ by contradiction.

Let $X_{1,x} = \{\bar{s}_1, \bar{s}_2, \cdots, \bar{s}_p\}$ and $X_{x+1} = \{\bar{t}_1, \bar{t}_2, \cdots, \bar{t}_q\}$, where $\bar{s}_i \in \mathbf{R}^n$ for $1 \leq i \leq p$ and $\bar{t}_j \in \mathbf{R}^n$ for $1 \leq j \leq q$. Suppose the iteration partition $P_{\Psi_{1,x+1}}(I^n)$ can cause interblock communication. Without loss of generality, assume that there exists a data dependence vector $\bar{t}'$ between two iterations $\bar{i}_1 \in B_1$ and $\bar{i}_2 \in B_2$; i.e., $\bar{t}' = \bar{i}_2 - \bar{i}_1$. Let $\bar{b}_1$ and $\bar{b}_2$ be the base points of blocks B_1 and B_2, respectively. Then the iterations $\bar{i}_1 = \bar{b}_1 + a_1\bar{s}_1 + \cdots + a_p\bar{s}_p + a_{p+1}\bar{t}_1 + \cdots + a_{p+q}\bar{t}_q$ for $a_l \in \mathbf{R}$, $1 \leq l \leq p+q$, and $\bar{i}_2 = \bar{b}_2 + e_1\bar{s}_1 + \cdots + e_p\bar{s}_p + e_{p+1}\bar{t}_1 + \cdots + e_{p+q}\bar{t}_q$, for $e_l \in \mathbf{R}$, $1 \leq l \leq p+q$. Because the data dependence vector $\bar{t}'$ is caused by one of $x + 1$ arrays, it must be of the form $\bar{t}' = f_1\bar{s}_1 + \cdots + f_p\bar{s}_p$ for $f_l \in \mathbf{R}$, $1 \leq l \leq p$, or $\bar{t}' = g_1\bar{t}_1 + \cdots + g_q\bar{t}_q$ for $g_l \in \mathbf{R}$, $1 \leq l \leq q$. The iteration $\bar{i}_2$ therefore becomes one of the following forms:

$$\bar{i}_2 = \bar{i}_1 + \bar{t}' = (\bar{b}_1 + a_1\bar{s}_1 + \cdots + a_p\bar{s}_p + a_{p+1}\bar{t}_1 + \cdots + a_{p+q}\bar{t}_q) + (f_1\bar{s}_1 + \cdots + f_p\bar{s}_p) = \bar{b}_1 + (a_1 + f_1)\bar{s}_1 + \cdots + (a_p + f_p)\bar{s}_p + a_{p+1}\bar{t}_1 + \cdots + a_{p+q}\bar{t}_q$$

and

$$\bar{i}_2 = \bar{i}_1 + \bar{t}' = (\bar{b}_1 + a_1\bar{s}_1 + \cdots + a_p\bar{s}_p + a_{p+1}\bar{t}_1 + \cdots + a_{p+q}\bar{t}_q) + (g_1\bar{t}_1 + \cdots + g_q\bar{t}_q) = \bar{b}_1 + a_1\bar{s}_1 + \cdots + a_p\bar{s}_p + (a_{p+1} + g_1)\bar{t}_1 + \cdots + (a_{p+q} + g_q)\bar{t}_q.$$

By Definition 2, the iteration $\bar{i}_2$ belongs to B_1. But $\bar{i}_2$ must belong to block B_2; this is in contradiction with the iteration $\bar{i}_2$ belonging to B_1. Because of the above incorrect assumption, in which there exists a data dependence vector between iteration blocks, the iteration partition $P_{\Psi_{1,x+1}}(I^n)$ incurs no interblock communication for arrays A_i, $1 \leq i \leq x+1$, without duplicate data. □

Theorem 2: Given an n-nested loop L with k array variables, let the *reduced reference space* $\Psi^r_{A_j}$ be span(X^r_j) of each array A_j for $1 \leq j \leq k$. If $\Psi^r = \text{span}(X^r_1 \cup X^r_2 \cup \cdots \cup X^r_k)$, then Ψ^r is the *partitioning space* for communication-free partitioning of arrays A_j for $1 \leq j \leq k$ without duplicate data by using the iteration partition $P_{\Psi^r}(I^n)$.

Proof: This proof is similar to the proof of Theorem 1. It can be completely proved through usage of the *reduced reference space* $\Psi^r_{A_i}$ with duplicate data instead of the *reference space* Ψ_{A_i}, without duplicate data for each array A_i for $1 \leq i \leq k$. □

Theorem 3: Given an n-nested loop L with k array variables, let the *minimal reference space* $\Psi^{min}_{A_j}$ be span(X_j) of each array A_j for $1 \leq j \leq k$. If $\Psi^{min} = \text{span}(X_1 \cup X_2 \cup \cdots \cup X_k)$, then Ψ^{min} is the *minimal partitioning space* for communication-free partitioning of arrays A_j for $1 \leq j \leq k$, without duplicate data, by using the iteration partition $P_{\Psi^{min}}(I^n)$.

Proof: It shall be proved by induction as follows.

Basic $k = 1$: Clearly, $\Psi^{min} = \Psi^{min}_{A_1}$, which is correct because $\Psi^{min}_{A_1}$ is the *minimal partitioning space* for communication-free partitioning of array A_1 without duplicate data.

Induction hypothesis $k = x$: The space $\Psi^{min}_{1,x} = \text{span}(X_1 \cup X_2 \cup \cdots \cup X_x)$ is the *minimal partitioning space* for communication-free partitioning of arrays A_j for $1 \leq j \leq x$ without duplicate data.

Induction $k = x + 1$: The space $\Psi^{min}_{1,x+1} = \text{span}(X_1 \cup X_2 \cup \cdots \cup X_{x+1})$ can be rewritten by the form $\Psi^{min}_{1,x+1} = \text{span}(X_{1,x} \cup X_{x+1})$, where $X_{1,x} = X_1 \cup X_2 \cup \cdots \cup X_x$. By induction hypothesis, $\Psi^{min}_{1,x} = \text{span}(X_{1,x})$ is the *minimal partitioning space*, such that there exists no interblock communication for arrays $A_1, A_2, \cdots, A_x$ without duplicate data, by using the iteration partition $P_{\Psi^{min}_{1,x}}(I^n)$. Moreover, $\Psi^{min}_{A_{x+1}} = \text{span}(X_{x+1})$ is the *minimal partitioning space* such that there exists no interblock communication for array A_{x+1} without duplicate data by using the iteration partition $P_{\Psi^{min}_{A_{x+1}}}(I^n)$. Therefore, by Theorem 1, the space $\Psi^{min}_{1,x+1} = \text{span}(X_{1,x} \cup X_{x+1})$ is the communication-free *partitioning space* of arrays A_j for $1 \leq j \leq x+1$ without duplicate data. However, we prove that the *partitioning space* $\Psi^{min}_{1,x+1}$ is *minimal*, as follows.

Let $X_{1,x+1}$ be $X_{1,x} \cup X_{x+1}$. Assume that a vector $\bar{t} \in \mathbf{Z}^n$ in $X_{1,x+1}$ is removed to form the space $\Psi' = \text{span}(X_{1,x+1} - \{\bar{t}\})$, such that $\Psi' \neq \Psi^{min}_{1,x+1}$. Without loss of generality, the vector $\bar{t}$, which can lead to a useful data dependence, is assumed to be a

particular solution of array A_j, $1 \leq j \leq x+1$. Therefore, there exists interblock communication while applying the iteration partition $P_{\Psi'}(I^n)$ to array A_j. It can be proved that $\Psi^{min}_{1,x+1}$ is the *minimal partitioning space* for communication-free partitioning of arrays A_i, $1 \leq i \leq x+1$, without duplicate data.□

Theorem 4: Given an n-nested loop L with k array variables, let the *minimal reduced reference space* $\Psi^{min^r}_{A_j}$ be span(X^r_j) of each array A_j for $1 \leq j \leq k$. If $\Psi^{min^r} =$ span($X^r_1 \cup X^r_2 \cup \cdots \cup X^r_k$), then Ψ^{min^r} is the *minimal partitioning space* for communication-free partitioning of arrays A_j for $1 \leq j \leq k$ with duplicate data by using the iteration partition $P_{\Psi^{min^r}}(I^n)$.

Proof: This proof is similar to the proof of Theorem 3. It can be completely proved through use of the *minimal reduced reference space* $\Psi^{min^r}_{A_i}$ with duplicate data instead of the *minimal reference space* $\Psi^{min}_{A_i}$ without duplicate data for each array A_i for $1 \leq i \leq k$. □

REFERENCES

[1] U. Banerjee, "Unimodular transformations of double loops," in *3rd Workshop on Languages and Compilers for Parallel Computing*, 1990, pp. 192–219.

[2] D. Callahan and K. Kennedy, "Compiling programs for distributed-memory multiprocessors," *J. Supercomputing*, vol. 2, pp. 151–169, Oct. 1988.

[3] S. H. Friedberg, A. J. Insel, and L. E. Spence, *Linear Algebra.* Englewood Cliffs, NJ: Prentice-Hall, 1979.

[4] J. Gallivan, W. Jalby, and D. Gannon, "On the problem of optimizing data transfers for complex memory systems," *Proc. ACM Int. Conf. Supercomputing*, 1988, pp. 238–253.

[5] D. Gannon, W. Jalby, and J. Gallivan, "Strategies for cache and local memory management by global program transformations," *J. Parallel Distrib. Computing*, vol. 5, pp. 587–616, Oct. 1988.

[6] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 179–193, Mar. 1992.

[7] D. Hudak and S. Abraham, "Compiler techniques for data partitioning of sequentially iterated parallel loops," *Proc. ACM Int. Conf. Supercomputing* , 1990, pp. 187–200.

[8] F. Irigoin and R. Triolet, "Supernode partitioning," *Proc. 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, 1988, pp. 319–329.

[9] C. T. King, W. H. Chou, and L. M. Ni, "Pipelined data-parallel algorithms—Part II: Design," *IEEE Trans. Parallel Distrib. Syst.* , vol. 1, pp. 486–499, Oct. 1990.

[10] C. Koelbel, P. Mehrotra, and J. V. Rosendale, "Semi-automatic process partitioning for parallel computation," *Int. J. Parallel Programming*, vol. 16, no. 5, pp. 365–382, 1987.

[11] C. Koelbel and P. Mehrotra, "Compiling global name-space parallel loops for distributed execution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 440–451, Oct. 1991.

[12] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[13] L. S. Liu, C. W. Ho, and J. P. Sheu, "On the parallelism of nested for-loops using index shift method," *Proc. 1990 Int. Conf. Parallel Processing*, vol. II, 1990, pp. 119–123.

[14] M. Lu and J. Z. Fang, "A solution of the cache ping-pong problem in multiprocessor systems," *J. Parallel Distrib. Computing*, 1992, pp. 158–171.

[15] D. A. Padua, D. J. Kuck, and D. H. Lawrie, "High-speed multiprocessors and compilation techniques," *IEEE Trans. Comput.*, vol. C-29, no. 9, pp. 763–776, Sept. 1980.

[16] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, 1986, pp. 1184–1201.

[17] J. Ramanujam and P. Sadayappan, "A methodology for parallelizing programs for multicomputers and complex memory multiprocessors," *Proc. ACM Int. Conf. Supercomputing*, 1989, pp. 637–646.

[18] ——, "Compile-time techniques for data distribution in distributed memory machines," *IEEE Trans. Parallel Distrib. Syst.* , vol. 2, pp. 472–482, Oct. 1991.

[19] A. Rogers and K. Pingali, "Process decomposition through locality of reference," *Proc. ACM SIGPLAN'89 Conf. Programming Language Design and Implementation*, 1989, pp. 69–80, 1989.

[20] J. P. Sheu and T. H. Tai, "Partitioning and mapping nested loops on multiprocessor systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 430–439, Oct. 1991.

[21] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *Proc. ACM SIGPLAN'91 Conf. Programming Language Design and Implementation*, 1991, pp. 30–44.

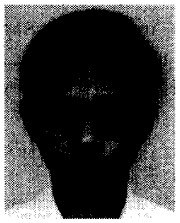
[22] ——, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 452–471, Oct. 1991.

[23] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers.* Cambridge, MA: MIT Press, 1989.

[24] ——, "More iteration space tiling," *Proc. ACM Int. Conf. Supercomputing*, 1989, pp. 655–664.

T.-S. Chen received the B.S. degree in computer science from Tamkang University, Taiwan, Republic of China, in 1989.

He is currently working toward the Ph.D. degree in the Department of Computer Science and Information Engineering, National Central University, Taiwan, Republic of China. His research interests include parallelizing compilers and parallel algorithms.



J.-P. Sheu (S'85–M'86) received the B.S. degree in computer science from Tamkang University, Taiwan, Republic of China, in 1981, and the M.S. and Ph.D. degrees in computer science from the National Tsing Hua University, Taiwan, Republic of China, in 1983 and 1987, respectively.

He joined the faculty of the Department of Electrical Engineering, National Central University, Taiwan, Republic of China, as an Associate Professor in 1987. Since 1992, he has been a Full Professor at the Department of Computer Science and Information Engineering, National Central University. His current research interests include parallel processing and distributed computing systems.

Dr. Sheu is a member of the IEEE Computer Society and Phi Tau Phi Society.