Vol. 39 No. 6 Transactions of Information Processing Society of Japan June 1998

Regular Paper

An Efficient Data Distribution Technique for Distributed Memory Parallel Computers

MINYI GUO,† YOSHIYUKI YAMASHITA†† and IKUO NAKATA†††

In this paper, we propose a linear data distribution technique, which extends the traditional BLOCK or CYCLIC distribution for intra-dimension as in HPF, to permit partitioning the array elements along slant lines. The array distribution patterns are determined by analyzing the array subscript references in loop nests. If the data are distributed along the slant lines, then we show the conversion algorithm between global address and local address, and the conversion algorithm from global iteration space to local iteration space. Some experimental results on a distributed memory parallel computer, CP-PACS, show the efficiency of our subscript analysis, and also show that the proposed data distribution technique can achieve better performance than the traditional data distribution for some scientific applications.

1. Introduction

For programming on massively parallel distributed memory machines, selecting an appropriate data decomposition is of critical importance to the performance of the data-parallel program for these machines. The important consideration for a good data distribution pattern is that it should maximize the system performance by balancing the computational load and by minimizing remote memory accesses. Languages, such as Fortran D [7] and HPF [13] support the functionalities of the distribution and alignment of data arrays through compiler directives. For example, in HPF [13], arrays are decomposed in two steps: the data alignment step, which deals with how data arrays should be aligned with respect to one another, and the data distribution step, which deals with how data arrays can then be distributed onto processors. Most of the existing data-parallel languages use the regular distribution, e.g., block, cyclic or block-cyclic distribution, as in Fortran D, Vienna Fortran [4] and HPF. The advantage of such a distribution method lies in the easiness to compute the local addresses for accessing the array elements and to generate the SPMD (Single Program Multiple Data) code for each processor. However, for some scientific applications, it is not always possible to minimize the communication cost.

In this paper, we propose a technique called the linear data distribution, which partitions array onto processors along parallel hyperplanes. We determine the array distribution pattern based on analysis of the array subscript references in the loop nests. We mainly concentrate our attention on two-dimensional arrays, since most of the arrays used in scientific computation application have less than three dimensions [10]. Moreover, two-dimensional arrays can be easily generalized to that of higher dimensions. A hyper-plane in two dimensions is a line; hence, we discuss the techniques to find the best method to partition data into parallel lines, such as rows, columns or slant lines, which will lead to a reduction in communication overhead. If data are distributed along slant lines, we will show the conversion algorithms between the global and local addresses, and the conversion algorithm from the global iteration space to the local iteration space.

The rest of this paper is organized as follows. Section 2 gives the overview of our linear data distribution. Section 3 describes the algorithms for conversion of the global address and the iteration space to the local ones. The experiment along with evaluation of the result are presented in Section 4. Related works and conclusion are given in Sections 5 and 6, respectively.

2. Linear Data Distribution

The traditional data distribution pattern used in most of the data-parallel languages is restricted to a method where arrays are distributed along intra-dimension with BLOCK or CYCLIC. However for some scientific applications, such a distribution fashion cannot guarantee the minimization of communication over-

† Doctoral Program in Engineering, University of Tsukuba
†† Institute of Information Sciences and Electronics, University of Tsukuba
††† University of Library and Information Science

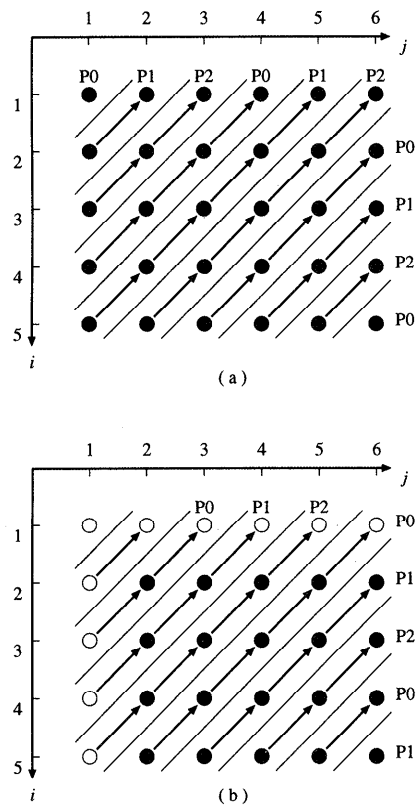

Fig. 1 Communication-free data distribution for array A, B in loop L_1, the number of processors p=3. (a) Array A. (b) Array B.

head.

Example 1 Consider the following loop nest L_1:

```
do i = 1, n1
  do j = 1, n2
    A(i,j) = F(A(i+1,j-1), B(i+1,j+1))
  enddo
enddo                                  L1
```

no matter how the array A is distributed along rows or columns over processors, the data communication among physical processors cannot be avoided when the SPMD program is executed. But if we extend the distribution pattern as anti-diagonal (**Fig. 1**), no communication will occur. We call this distribution pattern as partitioning along slant lines.

For this proposal, we suppose that, distribution of array elements over processors are not limited along intra-dimension, i.e., rows or/and columns, it can extend to inter-dimension, we call it linear data distribution. In other words, the array elements partitioned onto a certain processor satisfy linear equation $ai + bj = c_l$, where a, b and c_l are constants and i, j are array subscripts. For example, in Fig. 1 (a), $a = 1, b = 1$, and $c_l = 4, 7, 10, \ldots$ for the processor P_2.

In the following discussion, we suppose that array size is denoted as $n_1 \times n_2$ and the subscript starts from 1 while processors are numbered starting from 0.

2.1 Definition of Linear Distribution

Definition 1 Let $P = \{P_0, P_1, \ldots, P_{p-1}\}$ be a virtual processor set, p is the number of processors. Each P_k also has a set whose elements are all data to be distributed over it. For *linear distribution*, given the values of constant a and b, an array A distributed onto virtual processor P_k can be represented as

$$A_k = \{A(i,j) | ai + bj = c_l^k, 1 \le l \le L\}$$

where L is the number of linear lines distributed onto P_k.

Example 2 For loop nest appeared in Example 1, the communication-free data distribution is

$$\begin{aligned} A_k = \{&A(i,j) | i + j = c_A^k, \\ &c_A^k = 2 + k + (l-1) * p, \\ &1 \le l \le L \wedge (1,1) \le (i,j) \le (n_1, n_2)\}, \\ B_k = \{&B(i,j) | i + j = c_B^k, c_B^k = c_A^k + 2, \\ &(2,2) \le (i,j) \le (n_1, n_2)\}. \end{aligned}$$

Obviously, the traditional distribution such as BLOCK,CYCLIC can also be represented by linear distribution. For instance,
(BLOCK,*) distribution can be expressed as

$$\begin{aligned} A_k = \{&A(i,j) | i = c \wedge \\ &k * B + 1 \le c \le (k+1) * B\} \end{aligned}$$

where B is the block size.
(*,CYCLIC) distribution can be expressed as

$$\begin{aligned} A_k = \{&A(i,j) | j = c \wedge \\ &c = l * p + k + 1 \wedge 0 \le l < L\} \end{aligned}$$

where

$$L = \begin{cases} \lfloor \frac{n_2}{p} \rfloor, & k \ge (n_2 \bmod p) \\ \lfloor \frac{n_2}{p} \rfloor + 1, & k < (n_2 \bmod p). \end{cases}$$

2.2 Distribution Analysis

We distinguish two types of array references according to whether an array appears both on the left hand side and the right hand side of the loop body (refer to as the lhs and the rhs arrays respectively) or not. If an array is not only assigned its value (appearing as the lhs array), but also used as an operand (appear-

ing as the rhs array), data dependence (flow or anti dependence) will occur in this loop. If a set of array elements possesses the dependence relation each other is partitioned on the same processor, communication among processors is not required. This and the next subsection describe how to compute the coefficients a and b, based on an analysis of two types of array references. We call these *distribution analysis* and *alignment analysis* respectively.

For an array appearing in a loop which has loop-carried dependence, we denote the existence of the dependence relations between $A(i_1, j_1)$ and $A(i_2, j_2)$ in the loop as follows:

---

do $i = 1, n_1$
  do $j = 1, n_2$
    $\delta(A(i_1, j_1), A(i_2, j_2))$ L_2
  enddo
enddo

---

where i_1, j_1, i_2, j_2 are linear function of i and j, respectively, i.e.,

$$i_1 = \alpha_{10} + \alpha_{11} i + \alpha_{12} j \quad (1)$$
$$j_1 = \alpha_{20} + \alpha_{21} i + \alpha_{22} j \quad (2)$$
$$i_2 = \beta_{10} + \beta_{11} i + \beta_{12} j \quad (3)$$
$$j_2 = \beta_{20} + \beta_{21} i + \beta_{22} j. \quad (4)$$

We assume that the array A can be partitioned along

$$ai + bj = c. \quad (5)$$

In order to eliminate the communication, the lhs array element and the rhs array element referred in a loop iteration (i, j) should be partitioned onto the same processor, which is satisfied if

$$ai_1 + bj_1 = c, \quad ai_2 + bj_2 = c'$$

where c and c' are constants assigned for the same processor. If we assign $c = c'$ then

$$ai_1 + bj_1 = ai_2 + bj_2 \quad (6)$$

should be true. Since i_1, j_1, i_2, and j_2 are given by Eqs. (1), (2), (3), and (4), applying them to Eq. (6) we have

$$a(\alpha_{10} + \alpha_{11} i + \alpha_{12} j) + b(\alpha_{20} + \alpha_{21} i + \alpha_{22} j) = a(\beta_{10} + \beta_{11} i + \beta_{12} j) + b(\beta_{20} + \beta_{21} i + \beta_{22} j)$$

which implies,

$$a\alpha_{10} + b\alpha_{20} = a\beta_{10} + b\beta_{20}$$
$$a\alpha_{11} + b\alpha_{21} = a\beta_{11} + b\beta_{21}$$
$$a\alpha_{12} + b\alpha_{22} = a\beta_{12} + b\beta_{22}.$$

In matrix notation, we have,

$$\begin{pmatrix} \alpha_{10} & \alpha_{20} \\ \alpha_{11} & \alpha_{21} \\ \alpha_{12} & \alpha_{22} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \beta_{10} & \beta_{20} \\ \beta_{11} & \beta_{21} \\ \beta_{12} & \beta_{22} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}.$$

Let

$$\alpha = \begin{pmatrix} \alpha_{10} & \alpha_{20} \\ \alpha_{11} & \alpha_{21} \\ \alpha_{12} & \alpha_{22} \end{pmatrix} \quad \beta = \begin{pmatrix} \beta_{10} & \beta_{20} \\ \beta_{11} & \beta_{21} \\ \beta_{12} & \beta_{22} \end{pmatrix}$$

and

$$\mathbf{a} = \begin{pmatrix} a \\ b \end{pmatrix}.$$

Then the above system of equations can be abbreviated as

$$(\alpha - \beta)\mathbf{a} = \mathbf{0}. \quad (7)$$

A nontrivial solution $\mathbf{a}$ (at most one of a and b is zero) that satisfies Eq. (7) would imply zero communication. Such a linear distribution is known as a *nontrivial distribution*. We illustrate the use of the above sufficient conditions with the following examples.

Example 3 Reconsider the loop L_1 in Example 1. For array A referred to in L_1, we obtain the following using the subscript reference analysis,

$$\begin{pmatrix} -1 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \mathbf{0}$$

which implies

$$a = b.$$

We select $a = 1$, $b = 1$ as a solution. This implies that A should be partitioned by anti-diagonals to achieve communication-free parallel execution, as shown in Fig. 1 (a).

Example 4 Consider a more complicated nested loop L_3

---

do $i = 1, n_1$
  do $j = 1, n_2$
    $\delta(A(2i+j+1, 3i), A(2j, i+j-1))$ L_3
  enddo
enddo

For array A in L_3, we have

$$\alpha = \begin{pmatrix} 1 & 0 \\ 2 & 3 \\ 1 & 0 \end{pmatrix} \quad \beta = \begin{pmatrix} 0 & -1 \\ 0 & 1 \\ 2 & 1 \end{pmatrix}$$

applying them to Eq. (7), we have,

$$\begin{pmatrix} 1 & 1 \\ 2 & 2 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \mathbf{0}$$

that is

$$a = -b.$$

We select $a = -1$ and $b = 1$ as a solution. This shows that the diagonal partition of A are nontrivial distribution.

It is possible that there only exists trivial solution for a and b ($a = 0$ and $b = 0$), which implies that there is no communication-free partition for A in this loop. We do not illustrate such instance due to limited paper space.

2.3 Alignment Analysis

For a loop in which the assignment to array A uses values of array B, to ensure that the data elements are read in a statement residing on the same processor as the one whose data element is being written onto, the alignment between distributions of A and B must be specified. In the alignment analysis, although we can not determine how to distribute A and B onto the processors but we can specify the relationship of which elements of A and B get distributed on the same processor.

In this subsection, we will deal with the loop such as

---

do $i = 1, n_1$
  do $j = 1, n_2$
    $A(i_1, j_1) = F(B(i_2, j_2))$ L_4
  enddo
enddo

---

where i_1, j_1, i_2, and j_2 are of the same functions as in Section 2.2. Similar to the distribution analysis, we specify that the arrays A and B are distributed along

$$a_1 i_1 + b_1 j_1 = c_1$$

and

$$a_2 i_2 + b_2 j_2 = c_2$$

respectively. Because the relation of elements of A and B is an alignment relation, we need not limit that they have the same value of c when A and B are distributed along the above formulas. After analyzing the subscript references for A and B similarly to Section 2.2, we should find a solution for the following system of equations to achieve communication-free partitioning:

$$\begin{pmatrix} -\alpha_{10} & -\alpha_{20} & 1 \\ \alpha_{11} & \alpha_{21} & 0 \\ \alpha_{12} & \alpha_{22} & 0 \end{pmatrix} \begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} =$$

$$\begin{pmatrix} -\beta_{10} & -\beta_{20} & 1 \\ \beta_{11} & \beta_{21} & 0 \\ \beta_{12} & \beta_{22} & 0 \end{pmatrix} \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix}$$

Notice that there may be an infinite number of solutions, we are interested in the relationship between a_1, b_1, c_1 and a_2, b_2, c_2. Consider the loop in Example 5.

Example 5 Compute the alignment relations of A and B in following loop nest.

---

do $i = 1, n_1$
  do $j = 1, n_2$
    $A(i+j, i) = F(B(i-1, j))$ L_5
  enddo
enddo

Communication-free partitioning is possible if the system of equations

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} =$$

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix}$$

has a nontrivial solution. The system of equations are reduced to the following set of equations:

$$a_2 = a_1 + b_1$$
$$b_2 = a_1$$
$$c_1 = c_2 + a_2$$

which has a solution (1) $a_1 = 0$, $b_1 = 1$, $a_2 = 1$, and $b_2 = 0$. This means that A is partitioned into columns and B is partitioned into rows (**Fig. 2**). We also can select solution (2) $a_1 = 1$, $b_1 = -1$, $a_2 = 0$, and $b_2 = 1$, which implies that A is partitioned into diagonals and B is partitioned into columns (**Fig. 3**).

In practice, the type of partition selected will be determined through the distribution analysis of A (or B).

2.4 Data Distribution Strategy

For our linear distribution technique, if coefficients a and b have been obtained through the subscript reference analysis, we can simply describe the data distribution strategy as follows:

Data Distribution Strategy

Say n arrays are used in a loop. For $i = 1, 2, \ldots, n$

(1) If $a_i = 0 \wedge b_i \neq 0$, select (*,BLOCK) or (*,CYCLIC). Practically, which of these two schemes is determined by other factors of the loop nest. For instance, if the bound of the inner-loop is the function of the index of outer-loop, selecting (*, CYCLIC) can achieve a good load balance. The same reasons are valid for

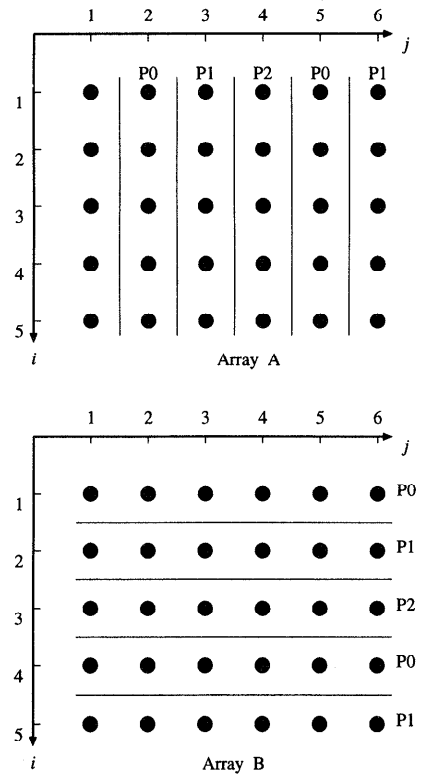

Fig. 2 Communication-free data distribution for solution (1) of Loop L_5.

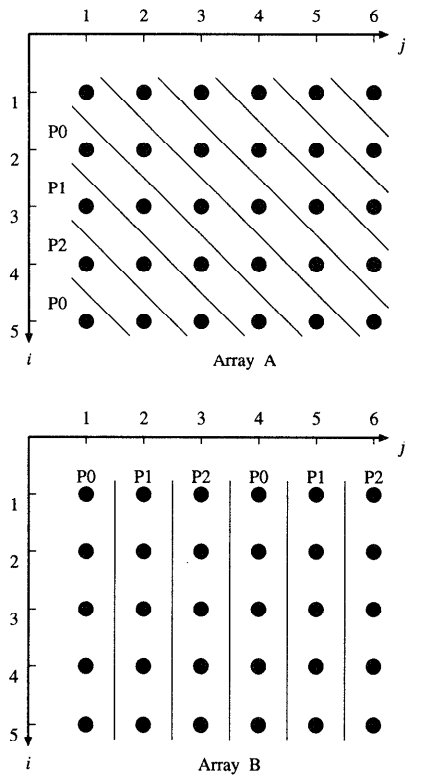

Fig. 3 Communication-free data distribution for solution (2) of Loop L_5.

(2).

(2) If $a_i \neq 0 \wedge b_i = 0$, select (BLOCK,*) or (CYCLIC,*).

(3) If $a_i \neq 0 \wedge b_i \neq 0$, select the linear distribution (a_i, b_i).

(4) If $a_i = 0 \wedge b_i = 0$, it only has *trivial distribution solution.* The good distribution scheme is determined by other factors of the loop nest.

(5) check the alignment relationship $[(a_i, b_i), (a_j, b_j)]$ so that no conflict occurs.

3. Index Conversion and Iteration Space Conversion

For the traditional regular BLOCK/CYCLIC distributions, there exists a set of direct algebraic formula for conversion between local and global indices. But if linear distribution is selected, no such algebraic formula can be applied when parallelizing compiler generates SPMD programs. Therefore we should consider the index conversion algorithm between local and global and we also implement the conversion algorithm from global iteration space to local iteration space.

Let us first consider the distribution method from a global array to local arrays. With respect to the traditional regular distribution, the local array spaces are usually allocated to nearly $1/p$ of the size of the global arrays and data are distributed across processors with the same size for the row or column of each local array, whenever the distribution scheme is along the row or column. But in linear distributions, the sizes of each row in local arrays are different, since the global arrays are distributed along slant lines, and cyclicly partitioned to each processor in order to get a good load balance. Hence, we should allocate the local array space as (number of slant lines)×(maximum size of line length). Reconsider the array A in Example 1. Let $p = 4$ and A is an 8×8 matrix, A is partitioned along slant lines $i + j = c$. The local data are distributed to each processor P_0, P_1, P_2, and P_3, as shown in **Fig. 4**. If we allocate 2 dimensional arrays for them, we must assign the local array space as 4×8, since it consumes the 2 times the space of global array and half of the local space is useless.

We therefore allocate the local array space as one-dimensional array when the linear dis-

P0 : A(1,1)
A(5,1), A(4,2), A(3,3), A(2,4), A(1,5)
A(8,2), A(7,3), A(6,4), A(5,5), A(4,6), A(3,7), A(2,8)
A(8,6), A(7,7), A(6,8)

P1 : A(2,1), A(1,2)
A(6,1), A(5,2), A(4,3), A(3,4), A(2,5), A(1,6)
A(8,3), A(7,4), A(6,5), A(5,6), A(4,7), A(3,8)
A(8,7), A(7,8)

P2 : A(3,1), A(2,2), A(1,3)
A(7,1), A(6,2), A(5,3), A(4,4), A(3,5), A(2,6), A(1,7)
A(8,4), A(7,5), A(6,6), A(5,7), A(4,8)
A(8,8)

P3 : A(4,1), A(3,2), A(2,3), A(1,4)
A(8,1), A(7,2), A(6,3), A(5,4), A(4,5), A(3,6), A(2,7), A(1,8)
A(8,5), A(7,6), A(6,7), A(5,8)

Fig. 4 Array A's elements distributed to each processor for loop L_1, where A (8,8) and p=4.

P0 : A'= A(1,1) A(5,1), A(4,2), A(3,3), A(2,4), A(1,5), A(8,2), A(7,3), A(6,4), A(5,5), A(4,6), A(3,7), A(2,8), A(8,6), A(7,7), A(6,8)
bound[4] = { 1, 2, 7, 14}

P1 : A'= A(2,1), A(1,2), A(6,1), A(5,2), A(4,3), A(3,4), A(2,5), A(1,6), A(8,3), A(7,4), A(6,5), A(5,6), A(4,7), A(3,8), A(8,7), A(7,8)
bound[4] = { 1, 3, 9, 15}

P2 : A'= A(3,1), A(2,2), A(1,3), A(7,1), A(6,2), A(5,3), A(4,4), A(3,5), A(2,6), A(1,7), A(8,4), A(7,5), A(6,6), A(5,7), A(4,8), A(8,8)
bound[4] = { 1, 4, 11, 16}

P3 : A'= A(4,1), A(3,2), A(2,3), A(1,4), A(8,1), A(7,2), A(6,3), A(5,4), A(4,5), A(3,6), A(2,7), A(1,8) , A(8,5), A(7,6), A(6,7), A(5,8)
bound[4] = { 1, 5, 13}

Fig. 5 The structure of the local arrays and their bound arrays for loop L_1.

tribution is applied. The global array elements distributed along $ai + bj = c$ are cyclicly partitioned by following the ascending order of the values of c, that is, the line whose value of c is *minc* (minimum c) is partitioned to P_0, next to P_1, and so on. For one processor, some parallel lines whose values are $c_1, c_2, \ldots, c_L$, where $c_i < c_j$, $i < j$, are partitioned over it, and the elements are also stored into local array following the ascending order of the local values of c. Attached to each local array is a bound array which records the first position of each slant line in the local array. For the current example, the local value of the array element and its bound array are shown in **Fig. 5**.

It is clear that there exists greatly different structures between the global and local arrays, thus one should develop a special conversion algorithms between the local and global indices

Algorithm 1 gi2li

Input: global index (i_g, j_g) and linear distribution coefficient (a, b)

Output: processor number P_k and local index i_l

1. calculate the minimum value of *c–minc* and maximum value of *c–maxc* for linear distribution (a, b) and value c for global index (i_g, j_g)

$$minc = \begin{cases} a+b & a*b>0 \\ a*n_1+b & a*b<0 \end{cases}$$

$$maxc = \begin{cases} a*n_1+b*n_2 & a*b>0 \\ a+b*n_2 & a*b<0 \end{cases}$$

$$c = a*i_g + b*j_g$$

2. calculate the processor number P_k:

$$P_k = (c - minc) \bmod p$$

3. calculate L: the number of slant lines of global array and l: the number of local lines distributed onto P_k

$$L = maxc - minc + 1$$

$$l = \begin{cases} L/p+1 & L \bmod p - P_k > 0 \\ L/p & \text{otherwise} \end{cases}$$

4. compute all values of c and local value c distributed onto P_k

$$global\_c = \{c_i | 1 \le i \le L \wedge c_i < c_j,\ if\ i<j\}$$
$$local\_c = \{c_i^k | c_i^k = c_{k+(i-1)*p+1} \wedge 1 \le i \le l\}$$

5. compute the bound of the one-dimensional local array A':

for $i = 1$ **to** l
$ij[i] = tuple(local\_c[i], a, b)$;

tuple() compute the number of legal tuples $< i, j >$ that satisfy $ai + bj = local\_c[\,]$

$$bound[u] = bound[1] + \sum_{v=1}^{n-1} ij[v],$$

where $bound[1] = 1$

6. get the local index i_l in processor P_k:

if $a*b<0$ **then**

$$i_l = \begin{cases} bound[u] + (j_g - 1)/|a| & c \le a+b \\ bound[u] + (i_g - 1)/b & c > a+b \end{cases}$$

if $a*b>0$ **then**

$$i_l = \begin{cases} bound[u] + (j_g - 1)/a & c \le a*n_1+b \\ bound[u] + (n_1 - i_g)/b & c > a*n_1+b \end{cases}$$

where $c = local\_c[u]$

Fig. 6 The conversion algorithm from the global index to the local index.

and a conversion algorithm from global iteration space to local iteration space for compiler generated SPMD programs. The algorithms gi2li, li2gi, and gs2ls in **Figs. 6~8** are the conversion from global to local index, the conversion from local to global index, and the conversion from global to local iteration space respectively. Here we assume $\text{sign}(a) = \text{sign}(a*b)$. Because, for example, if $a*b<0$, $ai+bj=c$ and $-ai-bj=-c$ are designated the same distribution.

Example 6 For the example in this section, the local array A' and its bound are given in Fig. 5, and

$$minc = 1+1 = 2$$
$$maxc = 8+8 = 16$$

Algorithm 2 li2gi

Input: local index i_l, processor number P_k, linear distribution coefficient (a, b)
Output: global index (i_g, j_g)

1. find out u such that
   $bound[u] \leq i_l < bound[u+1]$
2. get local value c of the line corresponding to i_l:
   $c = local\_c[u]$
3. **if** $a * b < 0$ **then**
   if $c \leq a + b$ bf then
   $j_g = (i_l - bound[u]) * |a| + 1$
   $i_g = (c - b * j_g)/a$
   else
   $i_g = (i_l - bound[u]) * b + 1$
   $j_g = (c - a * i_g)/b$
   endif
   else
   if $c \leq a * n_1 + b$ **then**
   $j_g = (i_l - bound[u]) * a + 1$
   $i_g = (c - b * j_g)/a$
   else
   $i_g = n_1 - (i_l - bound[u]) * b$
   $j_g = (c - a * i_g)/b$
   endif
   endif

Fig. 7 The conversion algorithm from the local index to the global index.

$L = (16 - 2 + 1)$.
Given the global index (3,6), the local index i_l is computed as follows:
$c = 3 + 6 = 9$
$P_k = (9 - 2) \bmod 4 = 3$
$i_l = bound[2] + (6 - 1)/1 = 5 + 5 = 10$
which means $A'(10) = A(3, 6)$ on processor P_3. Also, using Algorithm 3, we can obtain the SPMD node program of L_1 as follows:

do $j = 1, l$
  do $i = bound[j+1] - 1, bound[j], -1$
    $A'(i) = A'(i-1) + B'(i)$
  enddo
enddo

4. Experimental Results

In order to evaluate the ideas presented in this paper, we conducted some experiments implemented on CP-PACS, a 2048-processor MIMD distributed memory parallel computer developed at the University of Tsukuba. All the node programs are written in C, using PARALLELWARE☆ programming environment, a commercially available package that extends C and FORTRAN77 with a portable communication library. The experimental programs are the latter half part of ADI, namely an alternating direction implicit solution for the two dimensional diffusion equation, and a program which includes a matrix multiplication loop nest followed by a matrix transposition loop nest, called MULTRANS.

The sequential source program MULTRANS is shown in **Fig. 9**. We measured that the execution time of this sequential program is 201.76 sec (with data size 1024×1024). We also implemented several versions of parallelized programs through determining different data

Algorithm 3 gs2ls

Input: loop nest its type is as L_2 or L_4;
Output: SPMD program for input loop.

1. **if** the loop is L_2 **then**
2. compute the coefficient (a, b) according to i_1, j_1, i_2, j_2 as described in Section 2.2.
3. compute the distance vector d and local distance e:
   $d = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} = \begin{pmatrix} \alpha_{10} - \beta_{10} \\ \alpha_{20} - \beta_{20} \end{pmatrix}$
   $e = gcd(|d_1|, |d_2|)$
4. compute the upper and lower bound of local iteration space:
   $lower = bound[j]$ $upper = bound[j+1] - 1$
   $j = 1, 2, \ldots, l$
5. translate the local loop as
   do $j = 1, l$
   do $i = t_1, t_2, t_3$
   $\delta(A'(i), A'(i'))$
   enddo
   enddo
where determine the t_1, t_2, t_3 and i' through
   if $d_1 > 0 \wedge d_2 < 0$ **then**
   $t_1 = upper, t_2 = lower, t_3 = -1, i' = i + e;$
   if $d_1 < 0 \wedge d_2 > 0$ **then**
   $t_1 = upper, t_2 = lower, t_3 = -1, i' = i - e;$
   if $d_1 > 0 \wedge d_2 > 0$ **then**
   $t_1 = lower, t_2 = upper, t_3 = 1, i' = i - e;$
   if $d_1 < 0 \wedge d_2 < 0$ **then**
   $t_1 = lower, t_2 = upper, t_3 = 1, i' = i + e;$
6. **if** the input loop is L_4 **then** because elements $B(i_2, j_2)$ are aligned with $A(i_1, j_1)$ and distributed to the local array elements $B'(i)$, on the same processor with $A'(i)$, So the loop is simply translated to
   do $j = 1, l$
   do i = lower, upper
   $A'(i) = F(B'(i))$
   enddo
   enddo

Fig. 8 The algorithm conversion from the global iteration space to the local iteration space.

☆ PARALLELWARE is a trademark of Nippon Steel Corporation.

```
do i = 1,n
   do j = 1,n
      do k = 1,n
         C(i,j) = C(i,j) + A(i,k) * B(k,j);
      enddo
   enddo
enddo
do i = 1,n
   do j = 1,i
      T = A(i,j);
      A(i,j) = A(j,i);
      A(j,i) = T;
   enddo
enddo
```

Fig. 9 The sequential source experimental program MULTRANS.

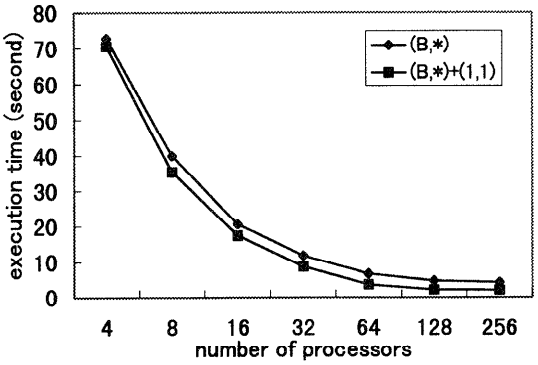

Fig. 10 Performance of the different versions of the experimental program MULTRANS on CP-PACS (data size: 1024×1024).

distribution schemes for each program, to observe how performance is influenced by different data distribution schemes for the same program, and to verify whether the executions and the analytical results are consistent.

As is widely known, based on the traditional BLOCK/CYCLIC distribution, the best distribution schemes for array A, B, and C in the program shown in Fig. 9 are (BLOCK,*), (*,BLOCK) and (BLOCK,*), respectively, for the matrix multiplication loop. However, in the matrix transposition loop, $A(i,j)$ and $A(j,i)$ have the data dependence relation in a loop iteration (i,j). Using the distribution analysis technique proposed in this paper, we obtained the coefficients of linear distribution (a,b) as $(1,1)$. That is, the linear distribution $(1,1)$ should be selected. Therefore, we implemented two types of distribution schemes for array A, where one is (BLOCK,*) for the entire program (denoted as (B,*) in **Fig. 10**), another is (BLOCK,*) for the multiplication loop and $(1,1)$ distribution for the transposition loop (denoted as (B,*) + (1,1) in Fig. 10). It requires redistribution of A between the two loops. We apply the naive redistribution approach [14] by using the global-local index conversion algorithms (Figs. 6, 7). The experimental results with array size $n = 1024\times1024$ on different processor nodes are shown in Fig. 10. Theoretically, the ideal speedup of a parallized program run on p processors is p times as much as the execution time of sequential program, but due to the affect of communication, the ideal speedup cannot be achieved in practical.

Although the linear distribution version incurred a higher computation and communication overheads for converting the index and redistributing the array elements, due to lack of communication execution of the transposition loop, it shows better performance, especially when the number of processors is numerous. However, in the linear distribution version, we also observed that the redistribution from the distribution scheme (BLOCK,*) to $(1,1)$ consumed most of the cost of executing the matrix transposition. In order to get more remarkable improvement, we must develop the optimal redistribution algorithms between the traditional and linear distributions [9].

To show our distribution strategy in Section 2.4 is valid, another program, the latter half part of ADI, is experimented. For this program, a loop-carried dependence occurs in the first dimension of arrays. We first measured the execution time of this sequential program is 0.38 sec (with data size 512×512). After analyzing the subscript references we can get $a = 0$ and $b = 1$, that is, the (*,CYCLIC) or (*,BLOCK) distribution scheme should be selected. We implemented three versions with the distribution (BLOCK,BLOCK), (BLOCK,*) and (*,CYCLIC). The performance where data size $n = 512\times512$ are shown in **Fig. 11**. We observed that if a suitable distribution scheme is selected, the practical speedup of the parallelized program is near to the ideal one, and inversely, if the distribution scheme is not properly selected, the performance of a parallelized code may even be slower than sequential code as reported by Blume and Eigenmann [3].

We also observed that for the (BLOCK,

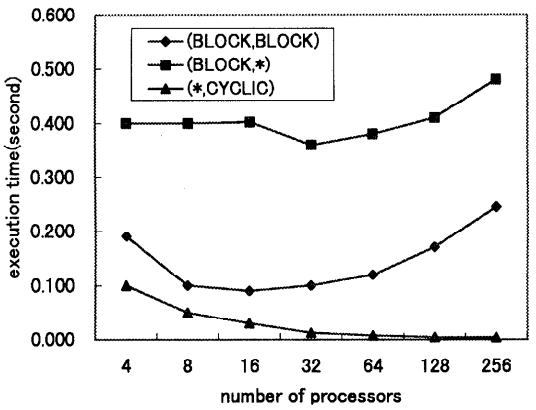

Fig. 11 Performance of the different versions of the latter part of ADI on CP-PACS (data size: 512×512).

BLOCK) and (BLOCK,*) distributions, the performance did not improve when the number of processors is increased, because the communication overhead has also increased. On the other hand, because of the communication-free execution under the (*,CYCLIC) distribution, the performance of this version is much better than the other two.

5. Related Work

Many researchers have tried to assist programmers in the difficult task of specifying a data distribution. Balasundram, Fox, Kennedy and Kremer[2] propose an interactive tool that allows programmers to select regions of a sequential program. This tool responds to a data decomposition scheme and diagnostic information for the selected region. The tool outputs a Fortran D program that can be translated by the Fortran D compiler[7),15)].

Gupta and Banerjee[10),11)] described the PARADIGM compiler that decomposes the data distribution problem into a number of subproblems, each dealing with a different distribution parameter for all the arrays of the input program (align pass, block-cyclic pass, block-size pass, and num-proc pass).

Anderson and Lam[1)] addressed the alignment and distribution problem in a linear algebraic framework. However, they often sacrifice parallelism to reduce communication. The tradeoff between communication and parallelism is intimately related to parameters of the target machine. Recently, Lim and Lam presented an algorithm to find the optimal affine transform that maximizes the degree of parallelism while minimizing the degree of synchronization in a program with arbitrary loop nestings and affine data accesses[16)]. Their algorithm is based on the concept of affine partitioning. They address that partition are used for two different purposes — space partition and time partition. It seems that their algorithm is more suitable for shared memory machines.

Dierstein, Hayer and Rauber[6)] discuss a branch-and-bound approach for distributing array data of the input program automatically. This algorithm incrementally constructs paths in a decision tree where each node of the path corresponds to the distribution of an array of the source program. For each path, a communication analysis tool computes the communication costs resulting from the way considered to distribute the array.

Ramanujan and Sadayappan[17)] and Chen and Sheu[5)] have worked on deriving data partitions for a restricted class of programs. They concentrate on the problems of transforming programs into the parallel form and reducing the interprocessor communication overhead.

Application which motivate the need of non-traditional BLOCK, CYCLIC distribution are shown in Ref. 12). In this reference some new data mapping techniques are proposed to overcome some limitations of the current HPF data mapping methods, from which we obtain some enlightenments.

6. Conclusions and Further Research

Our main focus in this paper has been on presenting an array distribution scheme by analyzing the subscript reference, which includes the linear distribution technique, as is distinguished from the traditional distribution scheme. This technique is suitable for optimizing array distributions for some areas in scientific programming. It can reduce the communication cost when the data dependence is carried along the slant lines in the loop nest. Another contribution of this technique is that the data distribution scheme for a loop nest can easily be determined by using the analysis of the subsrcipt reference. The main limitation is that we only deal with static array distribution in individual loops and do not consider dynamic distribution between several loops.

We are currently exploring several directions in which this work can be extended. We expect to embed this method to a practical parallelizing compiler and plan to examine other strate-

gies for searching through the space of possible data partitions. We have also developed some preliminary ideas on how to deal with the data redistribution problem which includes the linear distribution for individual loops [8),9)].

References

1) Anderson, J.M. and Lam, M.S.: Global Optimizations for Parallelism and Locality on Scalable Parallel Machines, *Proc. SIGPLAN '93 Conference on Program Language Design and Implementation*, pp.112–125, ACM (1993).
2) Balasundaram, V., Fox, G., Kennedy, K. and Kremer, U.: An interactive environment for data partitioning and distribution, *Proc. Fifth Distributed Memory Computing Conference* (1990).
3) Blume, W. and Eigenmann, R.: Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs, *IEEE Trans. Parallel and Distributed Systems*, Vol.3, No.6, pp.643–656 (1992).
4) Chapman, B., Mehrotra, P. and Zima., H.: Vienna Fortran – A Fortran Language Extension for Distributed Memory Multiprocessors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, North-Holland (1992).
5) Chen, T.-S. and Sheu, J.-P.: Communication-Free Data Allocation Techniques for Parallelizing Compilers on Multicomputers, *IEEE Trans. Parallel and Distributed Systems*, Vol.5, No.9, pp.924–938 (1994).
6) Dierstein, A., Hayer, R. and Rauber, T.: The ADDAP System on the iPSC/860: Automatic Data Distribution and Parallelization, *Journal of Parallel and Distributed Computing*, Vol.32, pp.1–10 (1996).
7) Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C. and Wu, M.: Fortran D language specification, Technical Report, TR90-141, Dept. of Computer Science, Rice University (1990).
8) Guo, M., Yamashita, Y. and Nakata, I.: Efficient Implementation of Multi-dimensional Array Redistribution, Technical Report, University of Tsukuba (1997). (submitted to *IEICE Trans. Informantion and Systems*).
9) Guo, M., Yamashita, Y. and Nakata, I.: Improving Performance of Multi-Dimensional Array Redistribution on Distributed Memory Machines, *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '98)*, Orlando, USA (1998).
10) Gupta, M.: Automatic data partitioning on distributed memory multicomputers, Ph.D. Thesis, University of Illinois, Urbana-Champaign (1992).
11) Gupta, M. and Banerjee, P.: PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers, *Proc. 1993 ACM International Conference on Supercomputing*, pp.87–96, ACM (1993).
12) HPF Forum: *HPF-2 Scope of Activities and Motivating Applications*, Rice University, Houston, Texas, version 0.8 edition (1994).
13) HPF Forum: *High performance Fortran Language Specification*, Rice University, Houston, Texas, version 2.0 edition (1996).
14) Kaushik, S.D., Huang, C.H. and Sadayappan, P.: Multi-phase Redistribution: A Communication-Efficient Approach to Array Redistribution, Technical Report, The Ohio State University (1995).
15) Kremer, U., Mellor-Crummey, J., Kennedy, K. and Carle, A.: Automatic Data Layout for Distribute-Memory Machines in the D Programming Environment, *Automatic Parallelization – New Approaches to Code Generation, Data Distribution, and Performance Prediction*, Kessler, C.W. (Ed.), Vieweg Advanced Studies in Computer Science, pp.136–147, Verlag Vieweg, Wiesbaden (1993).
16) Lim, A.W. and Lam, M.S.: Maximizing Parallelism and Minimizing Synchronization with Affine Transforms, *Proc. 24th Annual ACM SIGPALN-SIGART Symposium on Principles of Programming Languages*, Paris, France (1997).
17) Ramanujam, J. and Sadayappan, P.: Compile-time techniques for data distribution in distributed memory machines, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.4, pp.472–481 (1991).

(Received October 17, 1997)
(Accepted April 3, 1998)

Minyi Guo received the B.S. and M.E. degrees in Computer Science from Nanjing University, China in 1982 and 1986, respectively. From 1986 to 1994, he was a lecturer of the Department of Computer Science at Nanjing University. Since 1994, he is a doctor candidate of Engineering degree at University of Tsukuba, Japan. His research interests include CASE environment and tools, parallelizing compilers and data parallel languages. He is a member of the Information Processing Society of Japan (IPSJ) and the Japan Society for Software Science and Technology (JSSST).

Yoshiyuki Yamashita received the B.S. in Physics in 1982 from Osaka University. After working for four years at Hitachi Microcomputer Engineering Co., he received the M.E. and Ph.D. in Computer Science from University of Tsukuba in 1989 and 1991, respectively. He is currently an Associate Professor in the institute of Information Sciences and Electronics at University of Tsukuba. His current research interests are programming languages and scientific visualization.

Ikuo Nakata received the B.S. and M.S. degrees in mathematics and the D.Sc. degree in information science from the University of Tokyo, in 1958, 1960, and 1977, respectively. He joined the Central Research Laboratory, Hitachi, Ltd. in 1960 and the Systems Development Laboratory, Hitachi, Ltd. in 1974. From 1979 to 1997 he had been a Professor in the Institute of Information Sciences and Electronics, University of Tsukuba. Since 1997 he is a Professor of the University of Library and Information Science, Tsukuba, Japan. His current research interests include programming languages, language processors, software engineering, and parallel processing. Dr. Nakata is a member of the Information Processing Society of Japan, the Institute of Electronics and Communication Engineers of Japan, the Japan Society for Software Science and Technology, the IEEE Computer Society, and the Association for Computing Machinery.