The Journal of Supercomputing, 21, 191–211, 2002

Skewed Data Partition and Alignment Techniques for Compiling Programs on Distributed Memory Multicomputers*

TZUNG-SHI CHEN† chents@mail.cju.edu.tw

Department of Information Management, Chang Jung University, Tainan 711, Taiwan, R.O.C.

CHIH-YUNG CHANG changcy@email.au.edu.tw

Department of Computer and Information Science, Aletheia University, Tamsui, Taipei, Taiwan, R.O.C.

Abstract. Minimizing data communication over processors is the key to compile programs for distributed memory multicomputers. In this paper, we propose new data partition and alignment techniques for partitioning and aligning data arrays with a program in a way of minimizing communication over processors. We use skewed alignment instead of the dimension-ordered alignment techniques to align data arrays. By developing the skewed scheme, we can solve more complex programs with minimized data communication than that of the dimension-ordered scheme. Finally, we compare the proposed scheme with the dimension-ordered alignment one by experimental results. The experimental results show that our proposed scheme has more opportunities to align data arrays such that data communications over processors can be minimized.

Keywords: array alignment, data distribution, data partition, distributed memory multicomputers, parallelizing compilers

1. Introduction

Parallel processing is the most promising approach for designing and establishing high performance computers. Distributed memory multicomputers are commercially available and are being used to solve various scientific problems. They often offer significant advantages over the shared memory ones in terms of cost and scalability. However, they are much more difficult to program on shared memory machines. One major reason is the absence of a single global address space. As a result, programmers or compilers are responsible for distributing code and data over processors, and managing communication among tasks.

Over the last decade, a great number of researchers paid their attention on maximizing parallelism and minimizing communication for a given program executed

* This work is supported by the National Science Council of Republic of China under grants NSC-89-2213-E-309-005, NSC-89-2218-E-309-003 and NSC-89-2745-P-309-003.
† To whom all correspondence should be addressed.

on a parallel machine [1, 4, 5, 13, 14, 16–18, 21]. Chen and Sheu [4], Lim et al. [13, 14], Ramanujam and Sadayappan [16], and Shih, Sheu, and Huang [18] presented approaches to analyze data reference patterns on a program with structures of nested loops so that the parallelized program can be run on a parallel machine in a communication-free manner with some constraints. Furthermore, Lim et al. [13, 14] tried to maximize parallelism and minimize communication on a scalable parallel machine by using affine transformations when a program cannot be partitioned in a communication-free manner. For a program running on a distributed memory multicomputer, it is not easy to distribute and manage partitioned data and computations over processors when affine transformation methods addressed in [13, 14] are used. In general, their methods are suitable for use in shared memory or distributed shared memory multiprocessor systems. This is because affine transformation methods are not easy to handle regular data distribution, such as block or cyclic block data distribution, and to consider the situation of the workload balancing.

The distribution of data across processors is of critical importance to the efficiency of the parallel program in a distributed memory machine. For a good data distribution pattern, we should consider the case that it has to allow the workload to be evenly distributed over processors so that we can maximize parallelism and minimize communication. Recently, a number of researchers developed parallelizing compilers that take a sequential program based on automatic partitioning of data arrays and computations and generate the target parallelized program for a parallel machine [8, 10, 11, 15, 19, 20]. The PARADIGM project [8, 19] developed a fully automated technique for translating serial programs for efficient execution on distributed memory multicomputers. Lee [10, 11] proposed a dynamic programming technique to efficiently solve the data redistribution problem among program segments based on the approaches proposed in [8, 12]. Ayguadé, Garcia, and Kremer [2], Prakash and Srikant [15], and Tandri and Abdelrahman [20] also addressed data redistribution techniques for parallel programs with explicitly specifying **do**/**doall** loop constructs.

There are numerous researchers concentrated on the alignment and distribution problem [2, 3, 8, 10, 12, 20]. Li and Chen [12] studied the problem of aligning data arrays in order to minimize communications. Meanwhile, they also showed the alignment problem is NP-complete in the number of data arrays with a loop nest. Gupta and Banerjee [8] used the constraint-based method to extend the alignment method [12] for the distributed memory multicomputers. Lee [10] solved the data redistribution problem among loop nests using a dynamic programming technique based on Gupta and Banerjee's method [8]. Previous investigations [2, 3, 20] addressed some approaches to solve the alignment problem by considering how to distribute and align data arrays as well as how to preserve parallelism on a given program. Recently, Garcia, Ayguadé, and Labarta [7] proposed a framework for automatically integrating data alignment, distribution and redistribution together in distributed memory machines.

As we know, a large number of researchers as mentioned above have paid their attention in aligning multiple arrays by using array dimensions to match the other array dimensions of different arrays. In contrast to dimension-ordered data layouts,

Kandemir et al. [9] addressed a linear algebra framework to automatically determine the optimal data layouts expressed by hyperplanes for each array referenced in a program. That is, determining skewed data layouts is important to a program for analyzing data access behavior and exploiting parallelism. The skewed data layouts are useful for banded-matrix operations such as in BLAS library [6]. In addition, most data arrays are referenced in a skewed manner after applying loop skewing or unimodular transformations [23] to the program for extracting maximum parallelism.

The main aim of this paper is to propose a framework for determining the skewed data partition and distribution as well as skewed data alignment for each data array accessed on a given source program. This makes the partitioned data and program efficient while they are distributed and performed on a multicomputer system. Here the source program is assumed to be a loop nest with explicit **doall** and **do** constructs [23]. First, we show how to identify a perfect skewed data alignment relation between data arrays and the loop nest. Based on these relations, a skewed alignment scheme is proposed to align data arrays with a loop nest program so as to minimize communication. The experimental results show that our skewed alignment scheme is more efficient than the dimension-ordered matching scheme.

The rest of this paper is organized as follows. Section 2 states the machine model and program model used here as well as the strategies of the skewed data partition and data distribution. In Section 3, we explore the alignment relations between a data array and a loop nest. Furthermore, we propose a skewed array alignment scheme to optimize the data alignment problem in Section 4. Experimental results are presented in Section 5. Conclusions are summarized in Section 6.

2. Preliminaries

In this section, we first describe the distributed memory system model and program model used in this work. Next, the skewed data partition and data distribution are addressed.

2.1. Machine model and program model

Here we give the abstract target machine, a q-dimensional grid of $N_1 \times N_2 \times \cdots \times N_q (= N)$ processors, where q is the maximum dimensionality of any array used in the source program, and is less than or equal to the deepest level of the loop nest program appeared in the source program. A processor in the q-D (D stands for dimensional) grid is represented by the tuple $(p_1, p_2, \ldots, p_q)$, where $0 \leq p_i \leq N_i - 1$ for $1 \leq i \leq q$. Such a topology can be easily embedded into most of distributed memory machines, such as hypercubes and tori.

The program model used here is assumed to be an l-deep non-perfect loop nest containing the explicit **doall** and **do** loop constructs. The normalized loop model L [23] used in this paper is introduced below.

```
do i_1 = lb_1 to ub_1
   {code_1}
   do i_2 = lb_2 to ub_2
        {code_2}
            ⋮
            {code_{l-1}}
        do i_l = lb_l to ub_l
            {code_l}
        enddo /* for loop i_l */
            {code_{l+1}}
            ⋮
        {code_{2l-2}}
   enddo /* for loop i_2 */
        {code_{2l-1}}
enddo /* for loop i_1 */
```

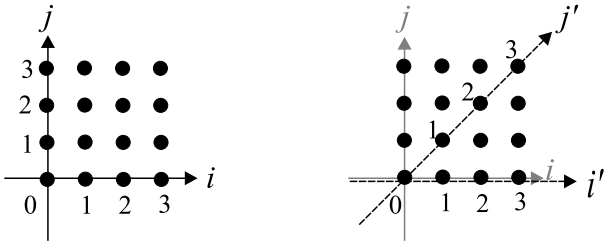
Here each loop i_j, $1 \leq j \leq l$, may be a **do** loop or **doall** loop construct. The expressions of lb_j and ub_j are integer-valued linear expressions possibly involving $i_1, i_2, \ldots, i_{j-1}$ for $1 < j \leq l$. The $code_j$, $1 \leq j \leq 2l-1$, is a basic block possibly including none of the program codes, some of statements, or loop nests. Programs belonging to this loop model can be obtained via a sequence of loop transformations used in [23]. The parallel program generated from a sequential program corresponds to the SPMD (simple program-multiple-data) model, in which each processor performs the same program code but operates on distinct data items [8, 10, 23]. In addition, the owner-computes rule is used here. Processor that owns left-hand side variable of a statement computes the whole statement [23].

2.2. Skewed data partition and distribution

In this subsection, we introduce how to express the skewed data partition and data distribution over processors. We first state the difference of data representation between traditional data space and skewed data space for a data array. Consider a 2-dimensional array $A[i, j]$ with $0 \leq i \leq 3$ and $0 \leq j \leq 3$. From the point view of traditional data space representation, we know that the data space of array A has two axes i and j as shown in Figure 1(a). In other words, we can say that the data space is spanned with two base vectors $(1, 0)$ and $(0, 1)$ corresponding to axes i and j, respectively. Therefore, an element $A[i, j]$ of array A can be represented by the matrix representation

$$\overline{I} = D\overline{I}, \text{ where } \overline{I} = \begin{bmatrix} i \\ j \end{bmatrix} \text{ and } D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

which uses the two base vectors as the basis of the data space. Now we intend to derive a new data space representation, named skewed data space representation, used in designing our proposed skewed alignment scheme. To illustrate the



(a) Data space of array A. (b) Skewed data space of array A.

Figure 1. Data space and its skewed data space for array A.

idea of skewed data alignment technique, we use two new vectors $\bar{s}_1 = (1, 0)$ and $\bar{s}_2 = (1, 1)$, which are linearly independent, as the basis of a new data space. The transformation from the original data space to the new data space can be derived as follows. We start evaluating all components of the vectors $\bar{s}_1$ and $\bar{s}_2$. Let g_1 and g_2 be their greatest common divisors (gcd) of all components in $\bar{s}_1$ and $\bar{s}_2$, respectively. Then, we divide each component by g_1 in $\bar{s}_1$ to obtain the new vector $\bar{s}'_1$ as well as divide each component by g_2 in $\bar{s}_2$ to obtain the new vector $\bar{s}'_2$. In this example, we have, $\bar{s}_1 = \bar{s}'_1$ and $\bar{s}_2 = \bar{s}'_2$. Now we put the two new vectors $(1, 0)$ and $(1, 1)$ as columns to form a transformation matrix $D' = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ so that we have two corresponding new axes i' and j'. Detailed derivations are shown in below.

$$
\begin{aligned}
D\overline{I'} \;&=\; D'\overline{I'}, \text{ where } \overline{I'} = \begin{bmatrix} i' \\ j' \end{bmatrix} \\
&\Longrightarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i' \\ j' \end{bmatrix} \\
&\Longrightarrow D'^{-1} D\overline{I} = D'^{-1} D'\overline{I'} \\
&\Longrightarrow D'^{-1} D\overline{I} = I_{2\times 2}\overline{I'} \\
&\Longrightarrow \overline{I'} = D'^{-1} D\overline{I}
\end{aligned}
$$

where

$$
D'^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \text{(an inverse matrix of } D')
$$

$$
\text{and} \qquad I_{2\times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{(an identity matrix).}
$$

That is, we have the following index transformation

$$
\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} i - j \\ j \end{bmatrix},
$$

where $-3 \le i' \le 3$ and $\max(0, -i') \le j' \le \min(3, 3 - i')$. Thus, its transformed data space is shown in Figure 1(b). Hence, i' is referred to as the axis of the vector $\bar{s}_1 = (1, 0)$ and j' is referred to as the axis of the vector $\bar{s}_2 = (1, 1)$. For example,

$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ while $\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. For general cases of the above transformation derivations, we can refer to reference [4] for details.

With these transformed axes of data arrays, we will discuss how to perform skewed data partition and data distribution among processors on a q-D grid below. For traditional data distribution, the k-th dimension of an n-dimensional data array A is denoted as A_k, $1 \le k \le n$. Here we use vectors to denote the dimensions of a data array. Let the original data space be spanned by the base vectors $\bar{e}_i$ for $1 \le i \le n$, where $\bar{e}_i$ is a $1 \times n$ vector whose components are set to 0 except for the i-th position set to 1. Each element of array A can be represented by such n base vectors. Thus, each array dimension $\bar{e}_k$ will be mapped to a unique dimension $map(\bar{e}_k)$ of the processor grid, where $1 \le map(\bar{e}_k) \le q$. For skewed data partition and distribution, a data array A in general can be represented by n row vectors $\bar{m}_k$, $1 \le k \le n$, where these n vectors are linearly independent. Thus, each array dimension $\bar{m}_k$ will be mapped into a unique dimension $map(\bar{m}_k)$ of the processor grid, where $1 \le map(\bar{m}_k) \le q$. Here we suppose that each array dimension $\bar{m}_k$ has the corresponding transformed index axis i'_k. The skewed data distribution for dimension $\bar{m}_k$ of a data array A is of the form

$$f_A^{\bar{m}_k}(i'_k) = \begin{cases} \left\lfloor \frac{d \times i'_k - offset}{block} \right\rfloor [\mathrm{mod} N_{map(\bar{m}_k)}] & \text{if } A \text{ is distributed} \\ X & \text{if } A \text{ is replicated} \\ \text{constant or } * & \text{if } A \text{ is not distributed,} \end{cases}$$

where $d \in \{-1, 1\}$ and the part of square parentheses surrounding is optional in this expression. Symbol d stands for index increasing or decreasing along the direction $\bar{m}_k$, which depends on whether d is 1 or -1, respectively. Symbol *offset* indicates displacement for the mapping of the dimension $\bar{m}_k$. Symbol *block* indicates the block size for distribution in this dimension. Function $f_A^{\bar{m}_k}(i'_k)$ returns the processor index along the dimension $map(\bar{m}_k)$ of the processor grid. This distribution function extended with skewed data distribution is generalized based on the proposed method [10]. Here we show different data distributions possible for a 16×16 data array $A[0..15, 0..15]$ on a 4×4 processor grid as in Figure 2(a) and Figure 2(b) and on a four-processor multicomputer as in Figure 2(c). The block size in these examples is 4. Their corresponding data distribution functions are shown below.

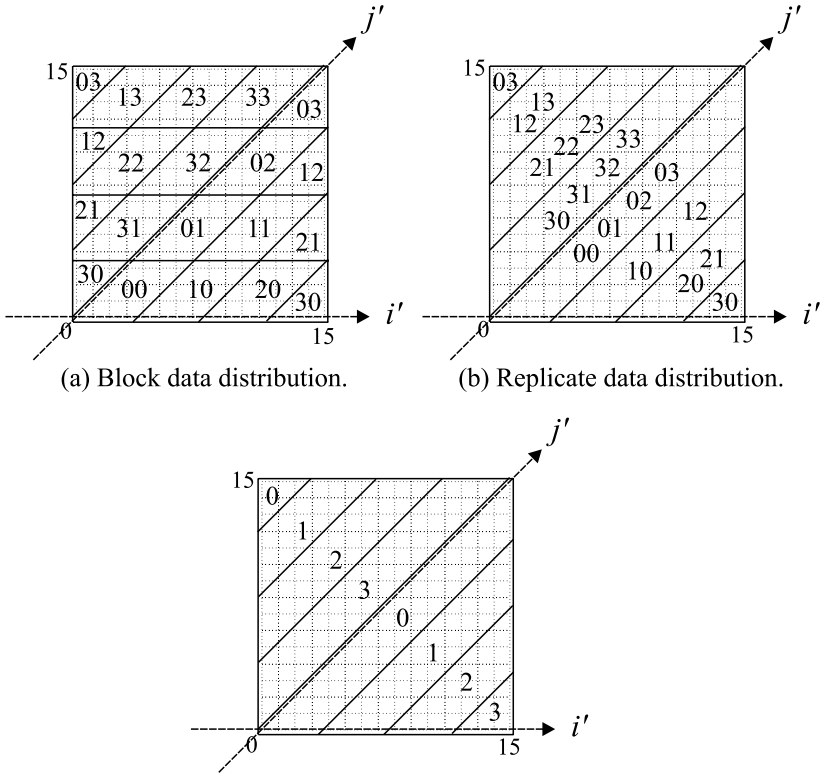
(a) $f_A^{(1,0)}(i') = \left\lfloor \frac{i'}{4} \right\rfloor \bmod 4$ $\quad f_A^{(1,1)}(j') = \left\lfloor \frac{j'}{4} \right\rfloor \bmod 4$

(b) $f_A^{(1,0)}(i') = \left\lfloor \frac{i'}{4} \right\rfloor \bmod 4$ $\quad f_A^{(1,1)}(j') = X$

(c) $f_A^{(1,0)}(i') = \left\lfloor \frac{i'}{4} \right\rfloor \bmod 4$ $\quad f_A^{(1,1)}(j') = *$

3. Data alignment relations

Some terms used in this paper are introduced in this section. An iteration space of an l-deep loop nest as an l-dimensional polyhedron where each point (iteration) is denoted by an $l \times 1$ column vector $\overline{I} = (i_1, i_2, \ldots, i_l)^t$, where t is denoted as the



(a) Block data distribution.

(b) Replicate data distribution.

(c) Cyclic data distribution.

Figure 2. Some data distribution schema for a 16×16 data array A.

operation of matrix transpose and each i_k denotes a loop index, $1 \leq k \leq l$. Here i_1 is the outmost loop while i_l is the innermost loop from outer to inner. Herein, $\bar{\mathbf{0}}^l$ represents an $l \times 1$ column vector where all of each component (element) are zero.

In the following, we define a data reference for an n-dimensional array A accessed by a statement surrounded by an l-deep non-perfect loop nest as defined in the previous section. The data reference to array A is expressed by $A[\exp_1, \exp_2, \ldots, \exp_n]$ where $\exp_j$, $1 \leq j \leq n$ is an integer-valued linear expression possibly involving loop index variables $i_1, i_2, \ldots, i_l$. This data reference can also be expressed by $M_A\overline{I} + \bar{o}$, where M_A is defined as an $n \times l$ access matrix, $\overline{I}$ is the iteration vector, and $\bar{o}$ is an offset (constant vector), an $n \times 1$ column vector [4]. For example, to the reference $A[i-1, i+j]$ surrounded by a 2-deep loop nest with index variables i and j, we have the access representation

$$M_A \begin{bmatrix} i \\ j \end{bmatrix} + \bar{o}, \text{where } M_A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \text{and} \quad \bar{o} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

Next, we will explore alignment relations between a certain data array surrounded by an l-deep loop nest and the k-th outermost loop, $1 \leq k \leq l$.

Definition 1 [Perfect Data Alignment]**.** Suppose that all the iterations on the k-th outermost loop in an l-deep loop nest access the elements of array A along a certain direction (dimension) $\bar{m}$. We call the dimension $\bar{m}$ of array A as *perfect data alignment* which is aligned with the k-th outermost loop.

By Definition 1, it turns out that if the property of perfect data alignment is held, no communication is incurred while distributing all iterations along the k-th outermost loop as well as distributing the elements of array A accessed by these iterations along dimension $\bar{m}$ over processors. This is because that the reference data and its corresponding computations (iterations) are distributed to the same processor. Hence, we have the following theorem based on the concept of skewed data layouts presented in [9].

Theorem 1 *Suppose we have an n-dimensional array reference R surrounded by an l-deep loop nest and there exists an $n \times l$ access matrix and an offset vector $\bar{o}$ to the array reference R. If there exists the k-th column vector $\bar{m}_k^t \neq \bar{\mathbf{0}}^l$ in M_R, $1 \leq k \leq l$, aligning the array direction $\bar{m}_k$ with the k-th outermost loop is referred to as a perfect data alignment.*

Proof: Assume $\bar{m}_k^t \neq \bar{\mathbf{0}}^l$ in M_R, $1 \leq k \leq l$. We will prove that aligning the array direction $\bar{m}_k$ with the k-th outermost loop is a perfect data alignment. Suppose that we have two contiguous iterations $\overline{I}_1$ and $\overline{I}_2$ in the k-th outermost loop. Let us consider two contiguous iterations $\overline{I}_1$ and $\overline{I}_2$ on the k-th outermost loop of a given l-deep loop nest; that is,

$$\overline{I}_1 = \begin{bmatrix} i_1 \\ \vdots \\ i_{k-1} \\ i_k \\ i_{k+1} \\ \vdots \\ i_l \end{bmatrix} \quad \text{and} \quad \overline{I}_2 = \begin{bmatrix} i_1 \\ \vdots \\ i_{k-1} \\ i_k + c \\ i_{k+1} \\ \vdots \\ i_l \end{bmatrix}, \text{ where } c \text{ is an integer constant.}$$

Both of them access their respective data points $M_R\overline{I}_1 + \bar{o}$ and $M_R\overline{I}_2 + \bar{o}$. Thus, we have the difference between two data points derived as given below.

$$(M_R\overline{I}_2 + \bar{o}) - (M_R\overline{I}_1 + \bar{o}) = M_R \begin{bmatrix} i_1 - i_1 \\ \vdots \\ i_{k-1} - i_{k-1} \\ (i_k + c) - i_k \\ i_{k+1} - i_{k+1} \\ \vdots \\ i_l - i_l \end{bmatrix}_{l\times 1} = M_R \begin{bmatrix} 0 \\ \vdots \\ 0 \\ c \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{l\times 1} = c\bar{m}_k^t$$

Therefore, the two data points accessed by the two contiguous iterations, respectively, in k-th outermost loop are lied on the same data reference direction $\bar{m}_k$ for

$1 \le k \le l$. By Definition 1, we can prove that aligning the array direction $\bar{m}_k$ with the k-th outermost loop is a perfect data alignment. □

It is noted that $\bar{m}_k^t = \bar{\mathbf{0}}^l$ means only a specific element of this array is repeatedly accessed by the iterations in the k-th outermost loop.

For example, consider the following 2-deep loop nest $L1$.

```
do i = 0 to m − 1 /* doall loop */
   do j = 0 to m − 1
      B[j + 1, i] = B[j, i] * A[i + j, j]          (L1)
   enddo
enddo
```

We have the access matrix to the data reference $A[i + j, j]$

$$M_A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = [\bar{m}_1^t \bar{m}_2^t].$$

According to Theorem 1, array A along $\bar{m}_1 = (1, 0)$ is perfectly aligned with the loop i. For instance, elements of array A accessed by the two consecutive iterations $\overline{I}_1 = (1, 1)$ and $\overline{I}_1' = (2, 1)$ to loop i have the relation of the data reference direction $\bar{m}_1^t = (1, 0)^t$; that is,

$$M_A \begin{bmatrix} 2 \\ 1 \end{bmatrix} - M_A \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Apparently, we do gain a good reference pattern, by distributing array A along the direction $\bar{m}_1 = (1, 0)$, while distributing the iterations of loop i over processors. As a result, we are able to make a profit on reducing communication while array A is distributed along $(1, 0)$ and thus is aligned with loop i. At the same time, we do also minimize data communication by distributing array A along the direction $\bar{m}_2 = (1, 1)$ while the iterations of loop j is distributed over processors. In contrast to the perfect data alignment, we have the non-perfect data alignment either aligning the array A along direction $(0, 1)$ with loop i or aligning the array A along direction $(0, 1)$ with loop j. Applying the non-perfect data alignment to distributing data and iterations generally incurs a lot of data communication among processors.

4. Skewed array alignment

In this section, we address a heuristic alignment scheme for efficiently aligning data arrays with a program so as to optimize and minimize data communication across processors.

We first define a directed graph, called *computation-communication alignment graph*, CCAG, to extract and express the characteristics of a program and data arrays accessed by the program.

Definition 2 [Computation-Communication Alignment Graph]. A *computation-communication alignment graph*, CCAG, is a directed graph $G = \{V, E\}$ to a program with an l-deep loop nest, where V is a set of vertices and E is a set of edges, defined below. V is composed of three different types of vertices:

(1) array dimension vertex $A_{\bar{m}_k}$ with respect to an array dimension $\bar{m}_k$ whose transpose column vector $\bar{m}_k^t \neq \bar{\mathbf{0}}^l$ is the k-th column vector of the $n \times l$ access matrix M_R for reference R of an n-dimensional array A, on a statement inside an l-deep loop nest, $1 \leq k \leq l$,
(2) loop vertex with respect to loop nest with **do** (sequential loop) construct, and
(3) loop vertex with respect to loop nest with **doall** (parallel loop) construct.

E is composed of three different types of edges:

(1) a read edge, $(A_{\bar{m}_k}, \text{loop } k)$, links array dimension $\bar{m}_k$ of array A to the incurred k-th outermost loop if this reference R is on the right-hand side of the statement,
(2) a write edge, $(\text{loop } k, A_{\bar{m}_k})$, links an incurred k-th outermost loop to array dimension $\bar{m}_k$ of array A if this reference R is on the left-hand side of the statement, and
(3) a **do** edge links loop nesting between two consecutive loops from outer to inner if there exists such a loop structure in a program.

There are weights on read and write edges which, respectively, indicate the number of edges $(A_{\bar{m}_k}, \text{loop } k)$ and $(\text{loop } k, A_{\bar{m}_k})$.

Note that if there is no loop nest or more than one consecutive loop nests in a program, we assume that there exists an outermost loop surrounding the original program. According to Definition 2, if there exists such a loop structure, the root of this loop structure with loop level 1 is the outermost loop vertex. The constructed loop structure can be labeled from root down to leaves one by one numbered with loop levels 1 to l.

As in loop nest $L1$, we have a CCAG with 4 array dimension vertices, $A_{(1,0)}$, $A_{(1,1)}$ $B_{(0,1)}$, and $B_{(1,0)}$, and 2 loop vertices, where loop vertex i, indicated by the gray point, is a **doall** loop and loop vertex j, indicated by the empty point, is a **do** loop as shown in Figure 3. An edge of solid arrow line with weight one links perfect data alignment dimension to its incurred loop, and vice versa. A **do** edge with dashed arrow line is represented by the loop structure linked from outer loop i to inner loop j.

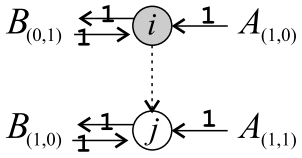


Figure 3. CCAG for loop nest $L1$.

Apparently, we have a lot of information to be extracted under the construction of a CCAG. These include what loop (iterations) can be parallelized, what array dimension needs to be aligned with some loop, and what array dimension needs to be aligned or matched with the other array dimensions. Each partitioned array dimensions can be identified while all of each dimension of the array are determined as well.

We know that the data alignment problem is an NP-complete problem [2, 12]. However, in this paper, we use a new kind of graphs to capture the characteristics of a program and data arrays as well as intent to optimize the data communication overhead. Thus, we will address a heuristic alignment scheme in the following. To the best of our knowledge, this is the first discussion to explore the skewed data alignment problem.

In the following subsections, we first address how to pick up the alignment directions from a given data array to a loop nest. Next, we shall present how to align multiple data arrays with the loop nest for minimizing data communication overhead.

4.1. Alignment for a data array

Based on the CCAG definition mentioned above, iterations (computations) along a loop are needed to be aligned with data arrays along specific dimensions. Here we first study how to pick up one of dimensions of some data array for minimizing the communication cost when the source program is performed. In this subsection, we shall address a better approach to select an array dimension to a specific data array aligned with the source program in order to optimize the communication overhead.

We know that a source program is modeled with doall-loops, performed in parallel, as well as do-loops, performed in sequence. We have to take into account examining the doall-loops first, then do-loops as the main issue. The other key issue is on the data access types with write references and read references. Here we first consider to align program with the data array with write references, then to consider the read references.

A heuristic algorithm to efficiently align a data array with a loop nest to minimize communication over processors based on a given CCAG is described below. In the following, we shall discuss how to select data alignment dimensions from a given data array. Here two main phases are designed to align multiple data references to an array: the doall-loops phase followed by the do-loops phase. Based on a constructed CCAG, we first examine the loop structure with existing **doall** loops from loop level 1 to the deepest loop level l in sequence in the doall-loops phase. That is, we give high priority to distributing the most external parallel loop from outer to inner while distributing computations over processors. This will lead to the largest granularity of this program that is distributed to processors. That is, we expect to explore the coarse-grained parallelism to a program. This consideration will be benefited to the distributed memory machines in general. In the former phase, we are concerned with minimizing communication while distributing computations along **doall** loops and distributing data arrays along the selected skewed dimensions. This

is because we require to distribute the computations along **doall** loops such that these computations can be executed in a parallel manner.

After that, we use the same operation as the doall-loops phase does to proceed with the processing of the **do** loops in the latter phase. In this phase, we examine the loop structure with the existing **do** loops from loop level 1 to the deepest loop level l as well. In the second phase, we are concerned with minimizing the residual communication while partitioning and distributing data arrays is performed along the selected skewed dimensions in this phase.

For each phase, two sub-phases are included; the former is for aligning write data references with the loop nest, whereas the latter is for aligning read data references with the loop nest. That is, we give higher priority to align write references with the loop nest than the read ones. It turns out that the computation distribution for this loop can meet the owner-computes rule for the generated parallel code.

The algorithm of aligning an n-dimensional data array A with l-deep loop nest is described below. Initially, let a set of alignment vectors for array A, S_A, be empty, ϕ, i.e., $S_A = \phi$.

Algorithm Skewed-Alignment(A, S_A)
Begin
Doall-loops phase:
    Alignment(Type of **doall** loop, Type of write edge, A, S_A);
    Alignment(Type of **doall** loop, Type of read edge, A, S_A);
Do-loops phase:
    Alignment(Type of **do** loop, Type of write edge, A, S_A);
    Alignment(Type of **do** loop, Type of read edge, A, S_A);
End

Procedure Alignment(*loop-type*, *edge-type*, A, S_A)

Step 1: Set p to 1.

Step 2: Examine loop level p, $1 \leq p \leq l$, on loop structure in CCAG of the l-deep loop nest.

Step 3: Assume there are r reference dimensions to array A connected to loop vertices of *loop-type* in loop level p. For each reference dimension $\bar{m}_k$, $1 \leq k \leq r$, we sum up the weights, which are on the edges connecting $\bar{m}_k$ with all of loops of *loop-type* in loop level p. We sort them according to weights and obtain the results, each aligned array dimension $\bar{m}_k$, with the weight w_k, $1 \leq k \leq r$, in a decreasing sequence; that is, $w_1 \geq w_2 \geq \cdots \geq w_r$. Suppose when $w_a = w_{a+1} = \cdots = w_b$, we have $w'_a \geq w'_{a+1} \geq \cdots \geq w'_b$ where w'_k is the total weight of summing up the weights, which are on the edges connecting $\bar{m}_k$ with all the loops of *loop-type* in loop level $p+1$ to loop level l, $1 \leq a \leq k \leq b \leq r$.

Step 4: We add the alignment direction one by one from $\bar{m}_1$ to $\bar{m}_r$ into the set of S_A if the vector is not the linear combination of the vectors in the predecessor set of S_A.

Step 5: We examine the set of S_A. If the cardinality of the set is not equal to n, dimensionality of array A, and $p < l$, add one to p and goto Step 2; otherwise, goto Step 6.

Step 6: Complete the construction of skewed data alignment dimensions associated with the loops of *loop-type* in the loop nest.

We know that it is possible that the construction of aligned dimension set S_A for array A is incomplete when the above algorithm is applied. That is, the cardinality of the constructed S_A is less than n, the dimensionality of array A. If the cardinality of S_A is still less that n, we add suitable base vectors to the alignment set S_A to form n vectors, linearly independent, as a complete set of dimensions for skewed data representation. This is because we have to use n linearly independent vectors to represent the data space of array A. Using these heuristic criteria of selection, the larger the sum of weights to an aligned dimension of an array, the more the number of references accessed to the aligned dimension of the array is. Therefore, we obtain the best set of aligned dimensions to an array in order to minimize communication under such a selection scheme.

We consider the Cholesky factorization algorithm appeared in [3] shown in the following loop nest $L2$.

```
do k = 1 to m
   S1: A[k, k] = sqrt(A[k, k])
   do h = k + 1 to m /* doall loop */
      S2: A[k, h] = A[k, h]/A[k, k]
   enddo
   do i = k + 1 to m − 1 /* doall loop */
      do j = i to m − 1 /* doall loop */
         S3: A[i, j] = A[i, j] − A[k, i] * A[k, j]          (L2)
      enddo
   enddo
enddo
```

In the loop nest $L2$, loop k is a **do** loop and all other loops h, i, and j are **doall** loops. The CCAG for $L2$ is shown in Figure 4. We apply our proposed scheme to

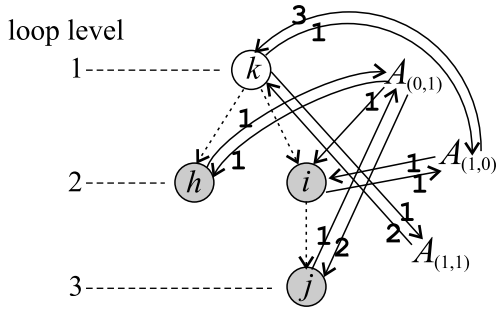


Figure 4. CCAG for loop nest $L2$.

loop nest $L2$ for selecting a perfect alignment to array A. We apply this method to **doall** loops and write references to array A in the loop structure using the alignment procedure in doall-loops phase. First, there is no **doall** loop in the loop depth level 1. Next, we are followed by examining the loop level 2. For aligned direction $\bar{m}_1 = (0, 1)$, we have its corresponding value of weight, 1, incurred by the write edge (loop h, $A_{(0,1)}$). For aligned direction $\bar{m}_2 = (1, 0)$, we also have its corresponding value of weight, 1, incurred by the write edge (loop i, $A_{(1,0)}$). Since these two values of weights are the same, we have to examine the loops below the loop level 2 in CCAG. Here, we only obtain the one write edge (loop j, $A_{(0,1)}$) with weight 1 for the aligned direction $\bar{m}_1 = (0, 1)$. Hence, we first select $\bar{m}_1$ with the largest priority in the loop level 2 to be added to the set of S_A; that is, $S_A = \{\bar{m}_1\}$. Next, we add $\bar{m}_2$ with the second larger priority to the set of S_A such that $S_A = \{\bar{m}_1, \bar{m}_2\}$. We complete the alignment process since the cardinality of elements in S_A is equal to 2, the dimensionality of array A, and both vectors are linearly independent. Thus, both of vectors $\bar{m}_1$ and $\bar{m}_2$ are the perfect alignment directions to array A in loop nest $L2$ based on our proposed scheme.

Note that Li and Chen's CAG [12] for Cholesky factorization algorithm has no edge because these is only a single array A in this loop nest. Hence, it is impossible to determine the kind of alignment to a certain data array which is good. Compared to their method, we can handle the skewed data partition along with alignment together by our proposed CCAG as using our proposed scheme.

4.2. Aligning multiple data arrays for a program

As mentioned above subsection, we have described how to efficiently align a data array individually with a source program. After that, we have to take into account how to determine data alignment with these multiple data arrays accessed by the program. Here we first transform the CCAG derived from the previous subsection into a CAG (component affinity graph) in [12]. Next, the proposed alignment algorithm [12] used here is applied to the CAG to solve our alignment problem.

In the following, we introduce how to align multiple data arrays together in details. Assume we have K data arrays, $A_1, A_2, \ldots, A_K$, accessed by the source program. For each data array A_i, we can obtain the corresponding set S_{A_i} of selected skewed alignment vectors, $1 \leq i \leq K$, by applying our proposed skewed-alignment algorithm. First, we transform and reduce the CCAG into a reduced CCAG, an undirected graph, called RCCAG. Each vertex in the RCCAG is a vertex including all loop vertices and array dimension vertices which was obtained from the set of S_{A_i} for each array A_i. The set of edges in RCCAG is composed of edges connecting loop vertices to the array dimension vertices in S_{A_i} for each array A_i.

Below we demonstrate how to get the RCCAG with the example loop nest $L2$ described in the previous subsection. The CCAG of loop nest $L2$ is shown in Figure 4. We know that we have the skewed alignment set $S_A = \{(0, 1), (1, 0)\}$ for data array A. Hence, we have the loop vertices, loops i, j, k, h, and the array dimension vertices $A_{(0,1)}$ and $A_{(1,0)}$. The RCCAG for loop nest $L2$ is shown in Figure 5.

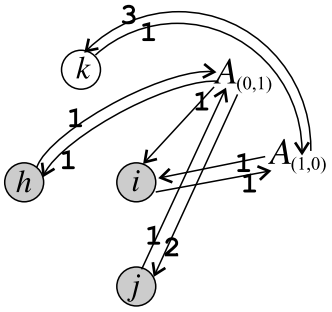


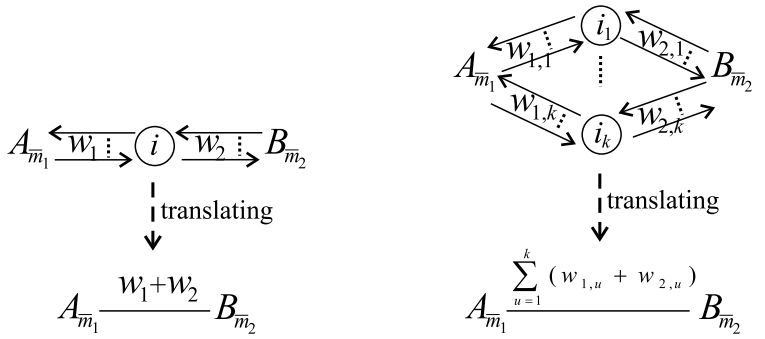
Figure 5. RCCAG for loop nest $L2$.

Next, we will transform the RCCAG into a CAG. Finally, we use the alignment algorithm [12], maximum-weight bipartite graph matching, to solve our perfect alignment problem to the transformed CAG. If there exists only one data array in loop nest, it is not necessary to transform the RCCAG to CAG. This is because we have already obtained the final results as in the loop nest $L2$.

In what follows, we present how to translate the RCCAG into CAG graph. The vertices in RCCAG are as the vertices in the transformed CAG. The undirected edges in the transformed CAG are generated by one of the following two cases.

Case 1. Suppose there is a weight w_1 which is the summation of weights on all edges connecting array dimension vertex $A_{\bar{m}_1}$ with a loop vertex i, whatever **do** or **doall** loops for array A. Also, suppose there is a weight w_2 which is the summation of weights on all edges connecting array dimension vertex $B_{\bar{m}_2}$ with the loop vertex i for the other array B. There exists an undirected edge, connecting $A_{\bar{m}_1}$ with $B_{\bar{m}_2}$, with having weight $w_1 + w_2$ on the edge as shown in Figure 6(a).

Case 2. Suppose there are weights $w_{1,1}, \ldots w_{1,k}$ which are the summation of weights on all edges connecting array dimension vertex $A_{\bar{m}_1}$ with the corresponding



(a) Case 1 of transforming RCCAG to CAG.

(b) Case 2 of transforming RCCAG to CAG.

Figure 6. Two cases of transforming RCCAG to CAG.

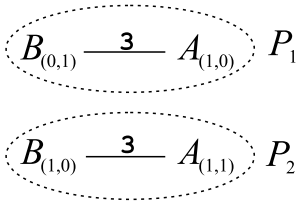


Figure 7. The transformed CAG for RCCGA and alignment between arrays A and B.

loop vertices $i_1, \ldots, i_k$, whatever **do** or **doall** loops for array A. Also, suppose there are weights $w_{2,1}, \ldots, w_{2,k}$ which are the summation of weights on all edges connecting array dimension vertex $B_{\bar{m}_2}$ with the corresponding loop vertices $i_1, \ldots, i_k$, whatever **do** or **doall** loops for the other array B. There exists an undirected edge, connecting $A_{\bar{m}_1}$ with $B_{\bar{m}_2}$, with having weight $\sum_{u=1}^{k}(w_{1,u} + w_{2,u})$ on the edge as shown in Figure 6(b).

The weight on each edge in the transformed CAG is represented as the unaligned penalty. That is, the larger the weight of an edge is, the more both the array dimensions need to be aligned. Therefore, we can use the approach, maximum-weight bipartite graph matching [12], to solve our alignment problem. For a given program, we can perform this scheme to transform the data arrays with the new axes and to explore the skewed alignment relations among these data arrays.

Now we demonstrate how to align multiple arrays together with the example, loop nest $L1$. The relations, CCAG for $L1$, among program segment and arrays A and B are shown in Figure 3. Via selecting the perfect data alignments for each array, we have the sets $S_A = \{\bar{m}_1^A = (1, 0), \bar{m}_2^A = (1, 1)\}$ and $S_B = \{\bar{m}_1^B = (0, 1), \bar{m}_2^B = (1, 0)\}$. Thus, the RCCAG is constructed from CCAG of loop nest $L1$ while all of directed edges were transformed to undirected edges. Therefore, we have the transformed CAG and two matched dimension partitions P_1 and P_2 as shown in Figure 7. Apparently, $A_{(1,0)}$ is aligned with $B_{(0,1)}$ and $A_{(1,1)}$ is aligned with $B_{(1,0)}$ when the proposed approach [12] is applied to this transformed CGA.

5. Experimental results

In this section, the experimental results are presented and discussed. We compare our proposed skewed scheme with the traditional dimension-ordered alignment scheme.

Based on the skewed data alignment technique, we are going to discuss how to determine the data distribution and block size for each data array with two examples as follows. These two examples are written by hand to the parallelized versions. Here assume N processors with 1-D grid machine used in our experiments. We first demonstrate how to determine data partition and block size with an example as loop nest $L1$ here. By applying our proposed scheme, $A_{(1,0)}$ is aligned with $B_{(0,1)}$ while $A_{(1,1)}$ is aligned with $B_{(1,0)}$ for $L1$. In $L1$, loop i is a **doall** loop, which can be executed in parallel. Clearly, we distribute loop i to N processors in block while loop j is sequentially executed without the need of distribution. As derived from

Section 2, we have the following index transformations.

$$\begin{bmatrix} i'_A \\ j'_A \end{bmatrix} = \begin{bmatrix} i-j \\ j \end{bmatrix} \text{ for array } A[i,j],\ 0 \le i \le 2m-1 \text{ and } 0 \le j \le m-1,$$

$$\text{and} \qquad \begin{bmatrix} i'_B \\ j'_B \end{bmatrix} = \begin{bmatrix} j \\ i \end{bmatrix} \text{ for array } B[i,j],\ 0 \le i \le m \text{ and } 0 \le j \le m-1.$$

Thus, we have the following skewed data distribution for array A as shown in Figure 8(a) and for array B as shown in Figure 8(b) with block size of $\frac{m}{N}$, provided the assumption that m is divisible by N.

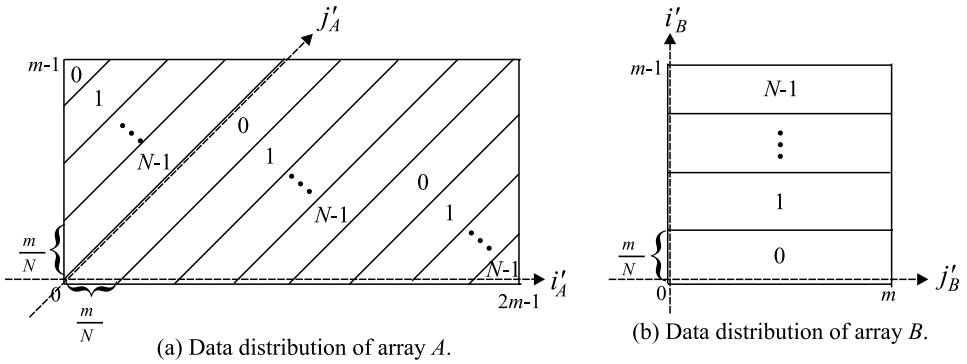
$$\text{(a) } f_A^{(1,0)}(i'_A) = \left\lfloor \frac{i'_A}{m/N} \right\rfloor \bmod N \qquad \text{(b) } f_B^{(0,1)}(i'_B) = \left\lfloor \frac{i'_B}{m/N} \right\rfloor \bmod N$$

$$f_A^{(1,1)}(j'_A) = * \qquad\qquad f_B^{(1,0)}(j'_B) = *$$

With such data and computation distributions over processors, the loop nest $L1$ can be executed in parallel in a communication-free manner.

For the loop nest $L1$, the experimental results are shown in Table 1. In this experimental study, we implement two versions of parallel codes on a 32-node nCUBE-2 multicomputer: the first, called SA, designed by our proposed skewed alignment scheme and the second, called CAG, designed by the CAG's alignment scheme where $A_{(1,0)}$ is aligned with $B_{(0,1)}$ while $A_{(0,1)}$ is aligned with $B_{(1,0)}$. In Table 1, the problem size is represented by m and the number of processors is represented by N. Here, the block size of data distribution for arrays A and B is $\frac{m}{N}$. Obviously, the running performance using our proposed skewed alignment scheme is superior to the traditional dimension-ordered alignment scheme. The major reason is that the parallel version implemented by the skewed alignment scheme is running on processors in a communication-free manner for this sample example.

We can see that data arrays A and B are perfectly aligned with each other. Assume we align arrays A and B using the traditional dimension-ordered alignment. That is, either $A_{(1,0)}$ is aligned with $B_{(1,0)}$ while $A_{(0,1)}$ is aligned with $B_{(0,1)}$, or $A_{(1,0)}$



(a) Data distribution of array A. (b) Data distribution of array B.

Figure 8. Optimum skewed data partition and distribution for loop nest $L1$ on 1-D grid.

Table 1. Execution time in seconds for loop nest $L1$ running in parallel on a 32-node nCUBE-2 multicomputer

	$N = 2$		$N = 4$		$N = 8$		$N = 16$		$N = 32$	
m	SA	CAG	SA	CAG	SA	CAG	SA	CAG	SA	CAG
2^8	0.43	1.02	0.23	1.13	0.13	1.15	0.06	1.88	0.03	2.43
2^9	0.81	1.82	0.45	1.03	0.27	1.36	0.14	1.57	0.05	1.89
2^{10}	1.65	3.85	0.87	2.91	0.47	2.46	0.28	2.97	0.16	3.08
2^{11}	3.23	8.76	1.65	5.54	0.88	5.02	0.46	4.89	0.27	4.76
2^{12}	6.43	15.87	3.23	12.11	1.65	11.89	0.87	11.03	0.46	10.55

is aligned with $B_{(0,1)}$ while $A_{(0,1)}$ is aligned with $B_{(1,0)}$. Both of them might incur a great amount of communication while these data arrays are distributed among processors whatever block or cyclic data distribution are used. This is because there does not exist any perfect data alignment between arrays A and B.

In most of parallelizing compiling systems, they use some powerful compiling techniques as tools to explore loop parallelism and data locality for improving the performance of executing programs. The techniques used, in general, include loop skewing, loop interchange, loop tiling, unimodular transformation, and more [22, 23]. Next, we consider the other example, the SOR loop program borrowed from [22] as shown in the following.

do $i = 1$ to $n - 2$
  do $j = 1$ to $m - 2$
    $A[i, j] = 0.2 * (A[i - 1, j] + A[i, j - 1] + A[i, j]$
      $+A[i + 1, j] + A[i, j + 1])$
  enddo
enddo

For exploring the parallelism of this SOR loop, we can apply loop skewing [23] to this sequential loop to produce the following parallelized loop.

do $i = 2$ to $n + m - 4$
  do $j = \max(1, i - n + 2)$ to $\min(m - 2, i - 1)$/* **doall** loop */
    $A[i - j, j] = 0.2 * (A[i - j - 1, j] + A[i - j, j - 1]$
      $+A[i - j, j] + A[i - j + 1, j] + A[i - j, j + 1])$
  enddo
enddo

Apparently, the subscripts of array A of the generated loop are linear expressions involving i and j. We can not use the CAG to analyze this parallelized loop because there is only one data array in this loop program. However, we can apply our proposed skewed scheme to this example. In this parallelized loop, loop j is a **doall** loop, which can be executed in parallel. We distribute loop j to N processors in block. Derived from previous sections, the data array A is spanned along both directions $(1, 0)$ and $(-1, 1)$ with the corresponding axes i'_A and j'_A, respectively.

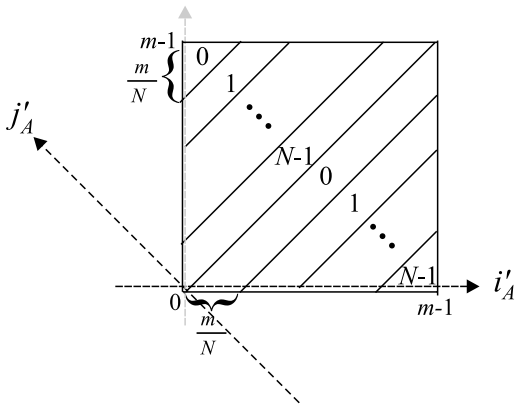


Figure 9. Data distribution of array A.

Here we assume $m = n$ and m is divisible by N in the SOR program. We also obtain the following index transformations.

$$\begin{bmatrix} i'_A \\ j'_A \end{bmatrix} = \begin{bmatrix} i + j \\ j \end{bmatrix} \text{for array } A[i, j],\ 0 \le i \le m-1 \quad \text{and} \quad 0 \le j \le m-1$$

Thus, we have the data partition for array A with block size $\frac{m}{N}$ along $(-1, 1)$ below as shown in Figure 9.

$$f_A^{(1,0)}(i'_A) = *$$
$$f_A^{(-1,1)}(j'_A) = \left\lfloor \frac{j'_A}{m/N} \right\rfloor \bmod N$$

In this example, we partition the data array A with the consideration of load balancing among processors. For the parallelized version of SOR program, we show the experimental results as depicted in Table 2. In this experimental study, we implement the parallel code on a 32-node nCUBE-2 multicomputer. The parallelized program is executed in a pipelined and concurrent manner. Obviously, to

Table 2. Execution time in seconds for the parallelized version of SOR program on a 32-node nCUBE-2 multicomputer

$m = n$	$N = 2$	$N = 4$	$N = 8$	$N = 16$	$N = 32$
2^8	1.43	1.32	1.26	1.13	1.11
2^9	1.85	1.77	1.59	1.31	1.23
2^{10}	2.87	2.12	2.01	1.88	1.64
2^{11}	4.52	3.22	2.86	2.23	1.92
2^{12}	8.02	5.27	4.02	3.16	2.54

this example, both shape and orientation of partitioning array A are matched to that of the distributed code. Thus, we can obtain the good running performance on nCUBE-2. From the above demonstration, we know that our skewed scheme can be combined with most of the powerful compiling techniques together, such as loop skewing, loop interchange, and so on. The proposed scheme can automatically detect and extract program parallelism as well as distribute data arrays over processors so that the incurred communication can be minimized.

6. Conclusions

Data alignment and distribution in distributed memory machines is of crucial important issue to the efficiency of parallelized programs since local memory accesses are much faster than those involving interprocessor communication. If no attention is paid to these data allocation problem, a large amount of time cost in data communication may seriously undermine the benefits of parallelism. In order for minimizing the interprocessor communication, it is critical for parallelized compilers to analyze the pattern of references among data arrays of a program and determine how to align and distribute these data.

In this paper, we proposed a heuristic alignment technique for efficiently aligning data arrays so as to minimize communication over multicomputers. We used skewed alignment instead of the traditional techniques with dimension-ordered alignment to align data arrays. With this development of skewed alignment scheme, we can solve complex programs having minimized data communication more efficiently than that of the traditional dimension-ordered scheme. Finally, we show that our proposed scheme is outperformed over the scheme proposed by the previous work as dimension-ordered data distribution used.

References

1. J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, pp. 112–125, 1993.
2. E. Ayguadé, J. Garcia, and U. Kremer. Tools and techniques for automatic data layout: a case study. *Parallel Computing*, 24:557–578, 1998.
3. V. Boudet, F. Rastello, and Y. Robert. Alignment and distribution is not (always) NP-hard. *Journal of Parallel and Distributed Computing*, 61:501–519, 2001.
4. T.-S. Chen and J.-P. Sheu. Communication-free data allocation techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 5:924–938, 1994.
5. M. Dion and Y. Robert. Mapping affine loop nests. *Parallel Computing*, 22:1372–1397, 1996.
6. J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–7, 1990.
7. J. Garcia, E. Ayguadé, and J. Labarta. A framework for integrating data alignment, distribution, and redistribution in distributed memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 12:416–431, 2001.
8. M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3:179–193, 1992.

9. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam, A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10:115–135, 1999.
10. P.-Z. Lee. Techniques for compiling parallel programs on distributed memory multicomputers. *Parallel Computing*, 21:1895–1923, 1995.
11. P.-Z. Lee. Efficient algorithms for data distribution on distributed memory parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 8:825–839, 1997.
12. J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.
13. A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24:445–475, 1998.
14. A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *13th ACM International Conference on Supercomputing*, Rhodes, Greece, pp. 228–237, June 1999.
15. S. R. Prakash and Y. N. Srikant. Communication cost estimation and global data partitioning for distributed memory machines. In *Proceedings of 1997 International Conference on High-Performance Computing*, India, December 1997.
16. J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2:472–482, 1991.
17. J.-P. Sheu and T.-S. Chen. Partitioning and mapping of nested loops for linear array multicomputers. *The Journal of Supercomputing*, 9:185–204, 1995.
18. K.-P. Shih, J.-P. Sheu, and C.-H. Huang. Statement-level communication-free partitioning techniques for parallelizing compilers. *The Journal of Supercomputing*, 15:243–269, 2000.
19. E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of 1995 International Conference on Supercomputing*, Barcelona, Spain, pp. 424–433, July 1995.
20. S. Tandri and T. S. Abdelrahman. Automatic partitioning of data and computations on scalable shared memory multiprocessors. In *Proceedings of 1997 International Conference on Parallel Processing*, Bloomingdale, IL, pp. 64–73, August 1997.
21. P. Tsanakas, N. Koziris, and G. Papakonstantinou. Chain grouping: A method for partitioning loops onto mesh-connected processor arrays. *IEEE Transactions on Parallel and Distributed Systems*, 11:941–955, 2000.
22. M. Wolfe. More iteration space tiling. In *Proceedings of the ACM International Conference on Supercomputing*, Crete, Greece, pp. 655–664, 1989.
23. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Inc., Reading, MA, 1996.