Techniques for Compiling Programs on Distributed Memory Multicomputers[1]

PeiZong Lee
Institute of Information Science, Academia Sinica
Taipei, Taiwan, R.O.C.
Internet: leepe@iis.sinica.edu.tw
TEL: +886 (2) 788-3799
FAX: +886 (2) 782-4814

Abstract

It is widely accepted that distributed memory parallel computers will play an important role in solving computation-intensive problems. However, the design of an algorithm in a distributed memory system is time-consuming and error-prone, because a programmer is forced to manage both parallelism and communication. In this paper, we present techniques for compiling programs on distributed memory parallel computers. We will study the storage management of data arrays and the execution schedule arrangement of Do-loop programs on distributed memory parallel computers. First, we introduce formulas for representing data distribution of specific data arrays across processors. Then, we define communication cost for some message-passing communication operations. Next, we derive a dynamic programming algorithm for data distribution. After that, we show how to improve the communication time by pipelining data, and illustrate how to use data-dependence information for pipelining data. Jacobi's iterative algorithm and the Gauss elimination algorithm for linear systems are used to illustrate our method. We also present experimental results on a 32-node nCUBE-2 computer.

Keywords: Data-dependence vector, data partition and distribution, distribution function, distributed memory computer, dynamic programming algorithm for data distribution, parallelizing compiler, pipelining.

[1]This work was partially supported by the NSC under Grant NSC 82-0408-E-001-016 and NSC 84-2213-E-001-003.

.

Keywords: Data-dependence vector, data partition and distribution, distribution function, distributed memory computer, dynamic programming algorithm for data distribution, parallelizing compiler, pipelining.

1 Introduction

This paper is concerned with compiling programs efficiently for distributed memory parallel computers. It is widely accepted that distributed memory parallel computers will play an important role in solving computation-intensive problems, because a distributed memory system can be easily scaled up to a very large number of processors to solve larger problems, such as Grand Challenge Problems [22]. However, the design of an algorithm in a distributed memory system is time-consuming and error-prone, because a programmer is forced to manage both parallelism and communication [5] [39] [48].

It is our goal in this paper to present techniques for compiling programs on distributed memory parallel computers. We will study the storage management of data arrays and the execution schedule arrangement of Do-loop programs on distributed memory parallel computers. If dependent data only influence neighboring data, an efficient component-alignment algorithm can be used to partition and distribute data arrays in the distributed memory. If dependent data influence a large number of data, then broadcasting techniques or pipelining techniques are used.

In the following, we briefly sketch the scope of this paper. First, we want to derive a dynamic programming algorithm for data distribution. Previously, Li and Chen [35], Gupta and Banerjee [16] formulated the component alignment problem from the whole source program. The fixed data distribution schema they derived may result in a larger communication overhead. Unlike their methods, we deal with each nested Do-loop independently. Data distribution schema between two nested Do-loops may be different and may require some data communication between them. A dynamic programming algorithm can heuristically determine whether the data re-distribution is necessary.

Second, we want to improve the communication time by pipelining data. Because many scientific computation algorithms are regular, they can be compiled as an iterative loop, whose body includes the following four steps: (1) parallel computation step, (2) reduction step, (3) updating step, and (4) broadcast step. The reduction step and broadcast step normally use a lot of communication time and result in the idleness of processors. However, it is possible to pipeline data for implementing a reduction operation or a broadcast operation, and thus, the communication time can be saved. In addition, if the hardware supports overlaying the computation and the communication, the total execution time may reduce further.

Third, we want to use data-dependence information for pipelining data. In many cases, the method of using the component alignment algorithm for distributing data is not sufficient enough to provide enough information for generating efficient communication operations. However, if compilers can detect all data-dependence vectors, it is possible to distribute data according to the iterative space. It also allows compilers to generate efficient communication codes for pipelining data.

Our research work is influenced by the pioneering efforts of compiling programs on distributed memory systems. Mace first showed that a class of dynamic data layout problems for interleaved memory machines are NP-complete [36]. Anderson and Lam then presented another formulation of the dynamic data layout problem to be NP-hard [1]. Kremer also identified that the problem of dynamic data remapping (the inter-phase data layout problem) is NP-complete [29]. Li and Chen, in addition, proved that the problem of determining an optimal static alignment between the dimensions of distinct arrays is NP-complete [35]. Thus, early parallelizing compiler research emphasized on allowing programmers to specify the data distribution using language extensions.

Callahan and Kennedy showed that message passing programs on distributed memory systems can be derived from sequential shared memory programs along with directions on how elements of shared arrays are distributed to processors [5]. Hiranandani, Kennedy, and Tseng developed an enhanced version of Fortran called Fortran D, which allowed programmers to specify data decomposition [18]. Koelbel and Mehrotra proposed the global name-space in which the compiler then permitted programmers to express their algorithms using the global name-space and to specify them at a high level [28]. Balasundaram, Fox, Kennedy, and Kremer designed an interactive environment for guiding data partition and distribution, and for estimating performance [2, 3]. Other similar works, which allow programmers explicitly to specify the data decomposition using language extensions and can generate all the communication instructions by compilers, include ASPAR [23], ParaScope [26], AL [42], SUPERB [49], and others.

There also have been research works to use compiler techniques to automatically determine sequential programs' data distribution on distributed memory systems. Chen, Choo, and Li studied compiling a functional programming language — Crystal — on various distributed memory systems, such as the iPSC, the nCUBE, and the Connection Machine [12]. They showed that it is possible to use compiler techniques for automatically transforming shared memory programs to message passing programs on distributed memory systems. They invented a component-alignment algorithm for data distribution [35], and derived an algorithm for matching syntactic reference patterns with appropriate aggregate communication primitives [34]. Gupta and Banerjee, on the other hand, have generalized the above compiler techniques for dealing with Fortran. They especially discussed the contiguous or cyclic data partitioning method and emphasized finding optimal communication costs at compile time [15, 16]. Su, Palermo, and Banerjee, in addition, developed a compiler, PARADIGM, for automating parallelization of regular computations on distributed-memory computers [41].

Chapman, Fahringer, Mehrotra, Moritsch, and Zima adopted Li and Chen's component alignment algorithm [35] for handling distributed data in Vienna Fortran [6]; in addition, they used language extension for handling dynamic data distribution [7]. Kennedy, Kremer, Meller-Crummey, and Carle proposed an automatic data layout strategy which is implemented in their D programming tools. For their strategy, they first explored several possible data layouts for each program phase; they then defined communication cost between candidate data layouts of adjacent phases. The problem of finding dynamic data layouts for the entire program is thus reduced to a single-source shortest paths problem [25, 31]. Kremer also developed techniques for using 0-1 integer programming for automatic data layout in the inter-phase data layout problem [30]. Other papers, which addressed the problem of determining initial data distributions and/or distributions for temporaries, include [9] [11] [43].

In addition, there are other research works related to the compilation of programs on distributed-memory computers. Kalns, Xu, and Ni suggested a cost model for determining a small set of appropriate data distribution patterns among many possible choices [24]. Hovland and Ni determined data distribution using augmented data access descriptors [19]. Chen and Sheu [13], Huang and Sadayappan [20], Ramanujam and Sadayappan [37] [38], and Wolf and Lam [44, 45] determined data distribution and/or degree of parallelism based on the hyperplane method. Furthermore, Gong, Gupta, and Melhem [14] and Hudak and Abraham [21] developed compile-time techniques for optimizing communication overhead.

The rest of this paper is organized as follows. In Section 2, we introduce the schema of data partition and distribution, and the communication primitives. In Section 3, we show an example of how to distribute data using the component alignment algorithm. In Section 4, we derive a dynamic programming algorithm for data distribution. In Section 5, we show how to improve the communication time by pipelining data and how to use data-dependence information for pipelining data. In Section 6, we apply our method to two applications and present experimental results on a 32-node nCUBE-2 computer. Finally, some concluding remarks are given in Section 7.

2 Background

In this paper, we are concerned with distributed memory systems. The abstract target machine we adopt is a q-D grid of $N_1 \times N_2 \times \cdots \times N_q$ processors, where D stands for dimensional and q is less than or equal to the deepest level of the Do-loop program. A processor on the q-D 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 almost all distributed memory machines, including configurably massively parallel computers. For example, the q-D grid can be embedded into a hypercube computer using a binary reflected Gray code. Gupta and Banerjee suggested that q is the maximum dimensionality of any data array used in the program [16]. However, it is possible to use higher dimensional grids for achieving faster computation. For example, we can use a 3-D grid for computing the 3-nested-loop matrix multiplication algorithm, although each data array used in the algorithm is 2-D.

The parallel program generated from a sequential program for a grid corresponds to the SPMD (Single Program Multiple Data) model, in which each processor executes the same program but operates on distinct data items [16] [18] [34] [42]. More precisely, in general, a source program has sequential parts (which must be executed sequentially) and concurrent parts (which can be executed concurrently). Each processor will execute the sequential parts individually; while all processors will execute the concurrent parts altogether by using message passing communication primitives. In practice, scalar variables and small data arrays used in the program are replicated on all processors in order to reduce communication costs; while large data arrays are partitioned and distributed among processors.

2.1 Data Partition and Distribution

In this subsection, we show how to represent the data partition and distribution of a large data array among processors. We first present a simple example which shows the idea of how to distribute a 1-D data array on a 1-D processor array with N processing elements. Consider the following program fragment.

```
DO 10 i = lo, hi
        :
-------- A(s * i + c) --------
        :
```

Suppose that $(hi - lo + 1)$ is a multiple of $s * b * N$. If we assign the entry $A(j)$ to processing element $p = (\lfloor \frac{j-(s*lo+c)}{s*b} \rfloor \bmod N)$, then the entries of the data array A are evenly distributed among N processing elements. In addition, for $i \in \{lo, \ldots, lo+b-1\}$, $A(s*i+c)$ is in processing element 0; for $i \in \{lo+b, \ldots, lo+2b-1\}$, $A(s*i+c)$ is in processing element 1; $\cdots$; and for $i \in \{hi-b+1, \ldots, hi\}$, $A(s*i+c)$ is in processing element $N-1$.

In the following, we define formulas to represent data distribution of specific data arrays among processing elements. Let the kth dimension of a data array A be A_k. Each array dimension A_k will be mapped to a unique dimension $map(A_k)$ of the processor grid, where $1 \leq map(A_k) \leq q$. The partition and distribution of each data array can be represented by a distribution function. As in this paper we only consider 1-D or 2-D data arrays, in the following, we show two distribution functions for distributing 1-D or 2-D data arrays, respectively. As to distribution functions for high dimensional data arrays, they can be generalized in a similar way. Suppose that the problem size is m.

Case 1: 1-D data array. The distribution function of a 1-D data array entry $A(i)$ is of the form

$$f_A(i) = \begin{cases} \lfloor \frac{d*i-offset}{block} \rfloor [\bmod N_{map(A)}] & \text{if } A \text{ is distributed} \\ X & \text{if } A \text{ is replicated} \\ \text{constant or } * & \text{if } A \text{ is not distributed,} \end{cases}$$

where $d \in \{-1,\ 1\}$ and the square parentheses surrounding "mod $N_{map(A)}$" indicate that the appearance of this part in the expression is optional. $f_A(i)$ returns the processor index along the dimension $map(A)$ of the processor grid where $A(i)$ is stored. This distribution function, which is a generalization based on [16], can specify the following parameters: (1) method of distribution — whether the data array is distributed across processors or replicated or not distributed; (2) method of partition — contiguous (if $block = \lceil \frac{m}{N_{map(A)}} \rceil$) or cyclic (if $block = 1$) or contiguous-cyclic (if $1 < block < \lceil \frac{m}{N_{map(A)}} \rceil$); (3) method of indexing — increasing or decreasing (which depends on whether d is 1 or -1); (4) the grid dimension ($map(A)$) to which the data array is mapped; (5) the block size ($block$) for distribution;

and (6) the displacement ($offset$) applied to the subscript value for mapping.

Case 2: 2-D data array. The distribution function of a 2-D data array entry $A(i,j)$ is of the form

$$f_A(i,j) = \begin{cases} (x_1,\ x_2) & \text{if the distributions of } A_1 \text{ and } A_2 \text{ are independent} \\ (x_1,\ (d_1x_1 + d_2x_2) \bmod N_{map(A_2)}) & \text{if } A_2 \text{ is rotated according to } A_1 \\ ((d_1x_1 + d_2x_2) \bmod N_{map(A_1)},\ x_2) & \text{if } A_1 \text{ is rotated according to } A_2, \end{cases}$$

where x_1 and x_2 are obtained from the 1-D distribution function, and d_1 and d_2 are in $\{-1,\ 1\}$. $f_A(i,j)$ returns the processor indices along the dimensions $map(A_1)$ and $map(A_2)$ of the processor grid where $A(i,j)$ is stored. This 2-D distribution function is a generalization from the 1-D distribution function, and therefore, it is more general than the one in [16], in which the distributions of A_1 and A_2 must be independent. For example, the data distribution schema of A, B, and C in a version of Cannon's matrix multiplication algorithm $A = B \times C$ can be represented by the schema in Fig. 1-(a), (b), and (c), respectively [4]; however, the data distribution schema of B and C in this algorithm cannot be represented by [16].

In Fig. 1, (a) – (d) show some data distribution schema possible for a 16×16 data array on a 4×4 processor grid, and (e) – (h) show some schema on a four-processor machine. As in Fortran, the data array subscripts are assumed to start with the value 1. For convenience, we assume $map(A_1) = 1$ and $map(A_2) = 2$. Their distribution functions are as follows.

$$\begin{array}{lll}
\text{(a)} & N_1 = 4,\ N_2 = 4: & f_A(i,j) = (\lfloor \frac{i-1}{4} \rfloor,\ \lfloor \frac{j-1}{4} \rfloor) \\
\text{(b)} & N_1 = 4,\ N_2 = 4: & f_A(i,j) = (\lfloor \frac{i-1}{4} \rfloor,\ (-\lfloor \frac{i-1}{4} \rfloor - \lfloor \frac{j-1}{4} \rfloor) \bmod 4) \\
\text{(c)} & N_1 = 4,\ N_2 = 4: & f_A(i,j) = ((-\lfloor \frac{i-1}{4} \rfloor - \lfloor \frac{j-1}{4} \rfloor) \bmod 4,\ \lfloor \frac{j-1}{4} \rfloor) \\
\text{(d)} & N_1 = 4,\ N_2 = 4: & f_A(i,j) = (\lfloor \frac{i-1}{4} \rfloor,\ X) \\
\text{(e)} & N_1 = 1,\ N_2 = 4: & f_A(i,j) = (*,\ \lfloor \frac{-j+16}{4} \rfloor) = (\lfloor \frac{-j+16}{4} \rfloor) \\
\text{(f)} & N_1 = 4,\ N_2 = 1: & f_A(i,j) = (\lfloor \frac{i-9}{4} \rfloor \bmod 4,\ *) = (\lfloor \frac{i-9}{4} \rfloor \bmod 4) \\
\text{(g)} & N_1 = 4,\ N_2 = 1: & f_A(i,j) = (\lfloor \frac{i-1}{2} \rfloor \bmod 4,\ *) = (\lfloor \frac{i-1}{2} \rfloor \bmod 4) \\
\text{(h)} & N_1 = 2,\ N_2 = 2: & f_A(i,j) = (\lfloor \frac{i-1}{4} \rfloor \bmod 2,\ \lfloor \frac{j-1}{4} \rfloor \bmod 2)
\end{array}$$

Note that, in (e) – (g), a 2-D data array is mapped to a 1-D processor array, therefore, one dimension of the 2-D data array is not distributed. Next, if the dimensionality q of the processor grid is greater than the dimensionality r of a data array, we need to specify the distribution across the remaining $(q - r)$ dimensions. We restrict the distribution in each remaining dimension either to a specific location or to be replicated along that dimension.

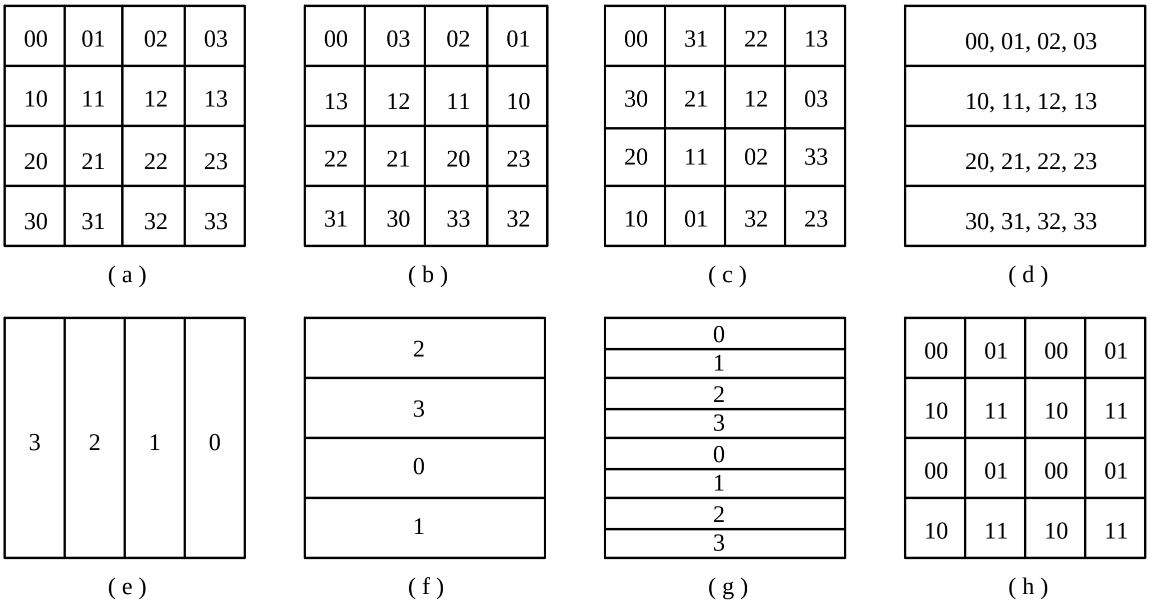

Figure 1: Various data distribution schema for a 16×16 data array.

2.2 Interdependence of Communication Generation and Data Distribution

Assume that the following set of communication primitives are supported by the run-time library.

- Transfer: send a message from a processor to the other processor.
- Shift: circular shift of data among neighboring processors along the specified grid dimension.
- OneToManyMulticast: send a message to all processors on the specified dimension(s) of the processor grid.
- Reduction: reduce data using a simple associative and commutative operator over all the processors lying on the specified grid dimension(s).
- AffineTransform: send data from each processor on the specified grid dimension(s) to a distinct processor according to an affine transform.
- Scatter: send a different message to each processor lying on the specified grid dimension(s).
- Gather: receive a message from each processor lying on the specified grid dimension(s).

- ManyToManyMulticast: replicate data from all processors on the specified grid dimension(s) to themselves.

Table 1 shows the communication costs of these primitives on the hypercube computer. The parameter m denotes the message size in words, seq is a sequence of identifiers representing the processors in various dimensions over which the collective communication primitive is carried out. The function num applied to such a sequence simply returns the total number of processors involved.

Primitive	Cost on Hypercube
Transfer(m)	$O(m)$
Shift(m)	$O(m)$
OneToManyMulticast(m, seq)	$O(m * \log num(seq))$
Reduction(m, seq)	$O(m * \log num(seq))$
AffineTransform(m, seq)	$O(m * \log num(seq))$
Scatter(m, seq)	$O(m * num(seq))$
Gather(m, seq)	$O(m * num(seq))$
ManyToManyMulticast(m, seq)	$O(m * num(seq))$

Table 1: Costs of communication primitives on the hypercube computer.

Li and Chen noticed that a data distribution scheme must be given before analyzing communication costs [34]. However, to examine whether a data distribution scheme is good or not, really depends on which communication primitives are involved. Gupta and Banerjee suggested the following two steps to break this cyclical dependency [16]. First, assume $N_1 = N_2 = \cdots = N_q$, then a data distribution scheme can be determined by using the component-alignment algorithm. Second, formulate the total execution time including the computation time and the communication time; then the values of N_1, N_2, ..., N_q can be determined by requiring the optimal total execution time.

3 Distributing Data Using the Component Alignment Algorithm

In this section, we show an example of how to distribute data arrays using the component alignment algorithm. Given a program, we first construct a component affinity graph from the source program. It is a directed, and weighted graph, whose nodes represent dimensions (components) of arrays and whose edges specify affinity relations between nodes. Two dimensions of arrays are said to have an affinity relation if two subscripts of these two dimensions are affine functions of the same (single) index variable of a Do-loop. For example, for $A[s_1 * i + c_1, j]$ and $B[k, s_2 * i + c_2]$, the first dimension of A and the second dimension of B have an affinity relation, because the two subscripts of these two dimensions are affine functions ($s_1 * i + c_1$ and $s_2 * i + c_2$) of the same index variable i of a Do-loop. The weight with an edge is equal to the communication cost and is necessary if two dimensions of arrays are distributed along different dimensions of the processor grid. The direction of an edge specifies the direction of the data communication according to the "owner computes" rule.

The component alignment problem is defined as partitioning the node set of the component affinity graph into q disjointed subsets (q is the dimension of the abstract target grid and q may be larger than the dimension of the physical target grid) so that the total weight of edges across nodes in different subsets is minimized, with the restriction that no two nodes corresponding to the same array are in the same subset. These q disjointed subsets will be use to determine data distributions for all data arrays.

Note that, the component affinity graph defined in this paper is slightly different from the one introduced by Li and Chen [35]. First, their component affinity graph is undirected. Second, in their model, they only showed the definition of an affinity relation for some specific cases. For example, if a left-hand-side subscript and a right-hand-side subscript have an affinity relation, then the left-hand-side subscript must be an index variable of a Do-loop and the right-hand-side subscript must be an affine function of the form $i + c$, where i is the index variable and c is an integer. Third, in their graph, there are only edges between left-hand-side objects and right-hand-side objects. This is because their model follows the restricted owner computes rule. Therefore, the computation of assignment statements can be performed only after all right-hand-side objects are sent to the owner processing elements of the left-hand-side objects. The owner computes paradigm simplifies code generation considerably. However, it does not provide an adequate solution when the right-hand side of assignment statements contain complex expressions, and whose operands must be sent to the owner of the left-hand-side objects.

In our model, we additionally add an edge between each pair of dimensions of two arrays if these two arrays are the corresponding two operands of a binary operator and these dimensions have an affinity relation (thus, these two operands of a binary operator are in the right-hand side of '='). This is because, if these two operands of a binary operator are aligned, it is better to compute this binary operator first and then send one intermediate result to the owner of the left-hand-side object, than to send these <u>two</u> operands independently to the owner of the left-hand-side object and then perform the computation due to this binary operator. However, if both operands have affinity relations to the left-hand-side object, then this edge is not necessary. We also specify the direction of each edge for the purpose of specifying the direction of the data communication according to a relaxing owner computes rule. For example, right-hand-side objects are sent to the owner of the left-hand-side objects as the conventional owner computes rule; objects of lower dimensional arrays are sent to the owner of the higher dimensional arrays in order to reduce the communication cost, if these objects are the corresponding operands of binary operators.

Note that, although the component alignment problem is NP-complete, Li and Chen have proposed an efficient heuristic algorithm based on applying the optimal matching procedure to a bipartite graph constructed from the nodes corresponding to components (dimensions) of two data arrays [35]. In this paper, when dealing with component alignment problems, we adopt Li and Chen's heuristic algorithm by regarding our directed component affinity graphs as undirected ones. The direction of edges, however, are used in a code-generation phase and will be used to determine the direction of the data communication according to the owner computes rule. Next, in this paper, we assume that the abstract target grid is 2-dimensional, and therefore, q is equal to 2.

3.1 Defining Approximate Communication Cost

We now define approximate communication costs if the corresponding two dimensions of arrays of an edge in the component affinity graph are distributed along different dimensions of the processor grid. Since we assume that the abstract target grid is a 2-D grid of $N = N_1 \times N_2$ processors, we only consider the following four cases depending on array's dimensionalities on both the tail of an edge and the head of that edge in Table 2. Assume that the problem size is m.

We now specify communication primitives we used when right-hand-side objects are sent to the owner of the left-hand-side objects. We use pattern matching techniques similar to those proposed by Li and Chen [34], and which are based on the relationship between left-hand-side and right-hand-side array subscript reference patterns. Table 3 shows the communication primitives used when left-hand-side and right-hand-side array subscripts have some specific patterns.

Note that, a naive implementation of case 9 can adopt a ManyToManyMulticast communication

edge head	edge tail	approximate communication cost
2-D	2-D	ManyToManyMulticast($\frac{m^2}{N}$, $\{N$ PEs$\}$) or $m^2 *$ Transfer(1)
1-D	2-D	ManyToManyMulticast($\frac{m^2}{N}$, $\{N$ PEs$\}$) or $N *$ OneToManyMulticast($\frac{m^2}{N}$, $\{N_2$ PEs$\}$)
2-D	1-D	$N_1 *$ OneToManyMulticast($\frac{m}{N_1}$, $\{N$ PEs$\}$)
1-D	1-D	$N_1 *$ OneToManyMulticast($\frac{m}{N_1}$, $\{N_2$ PEs$\}$) or $m *$ Transfer(1)

Table 2: The communication cost required if the corresponding two dimensions of arrays of an edge in the component affinity graph are distributed along different dimensions of the processor grid. These costs depend on array's dimensionalities on both the tail of the edge and the head of that edge.

case	LHS	RHS	communication primitive
1	c_1	c_2	Transfer
2	i	i	no communication
3	i	$i \pm c$	Shift
4	i	c	OneToManyMulticast
5	i	j	ManyToManyMulticast
6	i	unknown	Gather
7	unknown	i	Scatter
8	(i, j)	$(i, j)\, A + (c_1, c_2)$	AffineTransform
9	$f_1(i)$	$f_2(i)$	*need additional analysis

Table 3: Communication primitives used when left-hand-side and right-hand-side array subscripts have some specific patterns. Let i and j be loop indexing variables; c, c_1, and c_2 be constants at compile time; "unknown" means that the value is unknown at compile time; A be a 2×2 integer matrix; and $f_1(i)$ and $f_2(i)$ be two affine functions of the form $s_1 * i + c_1$ and $s_2 * i + c_2$, respectively.

primitive. However, this can be improved by applying other techniques of generating communication for array expressions [8] [10] [17] [27] [40].

3.2 Example of Jacobi's Iterative Algorithm

Consider the following Jacobi's iterative algorithm for linear systems $A_{m \times m} X_m = B_m$.

```
{* X(i) has been assigned an initial value before the computation.  *}
DO 10 k = 1, MAX_ITERATION
   DO 6 i = 1, m
      V(i) = 0.0
      DO 6 j = 1, m
```

```
5            V(i) = V(i) + A(i,j) * X(j)
6        CONTINUE
7        DO 9 i = 1, m
8           X(i) = X(i) + (B(i) - V(i)) / A(i,i)
9        CONTINUE
10    CONTINUE
```

There is an iterative loop from line 1 to line 10, whose body is from line 2 to line 9. Fig. 2 shows the corresponding component affinity graph. The edge weights of the graph are as follows.

$$
\begin{aligned}
c_1 &= \text{ManyToManyMulticast}(\tfrac{m^2}{N}, \{N \text{ PEs}\}) \text{ (line 5)} \\
c_2 &= N_1 * \text{OneToManyMulticast}(\tfrac{m}{N_1}, \{N \text{ PEs}\}) \text{ (line 5)} \\
c_3 &= N_1 * \text{OneToManyMulticast}(\tfrac{m}{N_1}, \{N_2 \text{ PEs}\}) \text{ (line 8)}
\end{aligned}
$$

Along with each term, we indicate the line number in the program to which the affinity relation appears. The data size for array A is m^2, the data size for arrays V, B, and X each is m. The total number of processors is denoted by N, while N_1 and N_2 refer to the number of processors along which various array dimensions are initially assumed to be distributed. Note that c_1 and c_2 are greater than c_3. Therefore, applying the component alignment algorithm on this graph, we get the following disjointed sets of dimensions: set 1 includes A_1 and V; set 2 includes A_2, B, and X. These two sets are mapped to dimensions 1 and 2, respectively, of the processor grid.

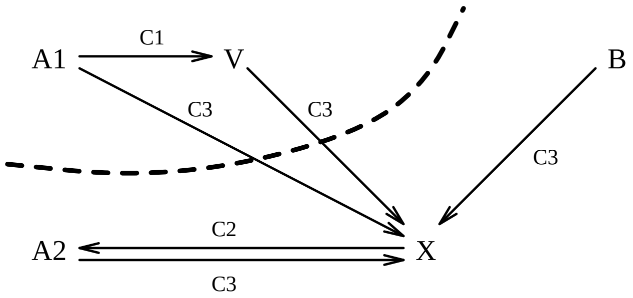

Figure 2: Component affinity graph of Jacobi's iterative algorithm

Next, we determine the partition strategy. As the iteration space is rectangular, for the sake of requiring load balance, the distribution functions for all array dimensions are determined to be contiguous:

$$f_A(i,j) = (\lfloor\frac{i-1}{m/N_1}\rfloor,\ \lfloor\frac{j-1}{m/N_2}\rfloor);\quad f_V(i) = (\lfloor\frac{i-1}{m/N_1}\rfloor,\ X);\quad f_X(j) = f_B(j) = (X,\ \lfloor\frac{j-1}{m/N_2}\rfloor).$$

Note that, the dimensionality, which is 2, of the abstract processor grid is greater than the dimensionality, which is 1, of data arrays V, X, and B. Data array V is distributed across the first dimension of the processor grid, and is replicated along the second dimension of the processor grid; data arrays X and B are distributed across the second dimension of the processor grid, and are replicated along the first dimension of the processor grid.

We now determine the value of N_1 and N_2. The total execution time including both the computation time and the communication time of an iteration from line 2 to line 9 is formulated as follows.

$$\begin{aligned}\text{Time} \quad = \quad & 2 * \tfrac{m^2}{N_1 * N_2} * t_f + \text{Reduction}(\tfrac{m}{N_1}, \{N_2 \text{ PEs}\}) \text{ (line 5)} \\ & + 3 * \tfrac{m}{N_2} * t_f + N_1 * \text{OneToManyMulticast}(\tfrac{m}{N_1}, \{N_2 \text{ PEs}\}) \\ & \quad \text{(or } N_1 * \text{Transfer}(\tfrac{m}{N_1}) \text{ if } N_2 = 1\text{) (line 8)} \\ & \quad \text{(communication cost because arrays } V \text{ and } X \text{ are distributed} \\ & \quad \text{ along different dimensions of the processor grid)} \\ & + \text{OneToManyMulticast}(\tfrac{m}{N_2}, \{N_1 \text{ PEs}\}) \text{ (line 5 and line 8)} \\ & \quad \text{(communication cost because of the loop-carried dependence of } X\text{).}\end{aligned}$$

We assume that the average time of a floating point operation is t_f and the average time of transferring a word is t_c. The optimal execution time can be obtained by substituting all possible N_1 and N_2 into the formula, where $N = N_1 * N_2$. Table 4 shows the execution time on three processor grids. The costs of communication primitives are based on Tables 1–3. The case when $N_1 = 1$ and $N_2 = N$ is better than the other two cases for the computation time. However, this distribution scheme cannot be satisfied, as it requires more communication time than the other two cases. Therefore, we will show a better approach in the next section.

$N_1 \times N_2$	Computation Time	Communication Time
$N_1 = 1,\ N_2 = N$	$(2 * \frac{m^2}{N} + 3 * \frac{m}{N}) * t_f$	$(2 * m * \log N) * t_c$
$N_1 = N,\ N_2 = 1$	$(2 * \frac{m^2}{N} + 3 * m) * t_f$	$(m + m * \log N) * t_c$
$N_1 = \sqrt{N},\ N_2 = \sqrt{N}$	$(2 * \frac{m^2}{N} + 3 * \frac{m}{\sqrt{N}}) * t_f$	$(\frac{1}{2} * m * \log N * (\frac{2}{\sqrt{N}} + 1)) * t_c$

Table 4: Computation time and communication time on three processor grids

4 A Dynamic Programming Algorithm for Data Distribution

Consider Jacobi's iterative algorithm again. The body of the iterative loop contains two Do-loops: L_1 and L_2. L_1 is from line 2 to line 6; L_2 is from line 7 to line 9. Suppose that we compute L_1 and L_2 separately using the component alignment algorithm. Then the total execution time for computing an iteration should include not only the execution time for L_1 and L_2 but also the communication time for changing data layouts for L_1 to L_2 and the communication time because of the loop-carried dependence, see Fig. 3.

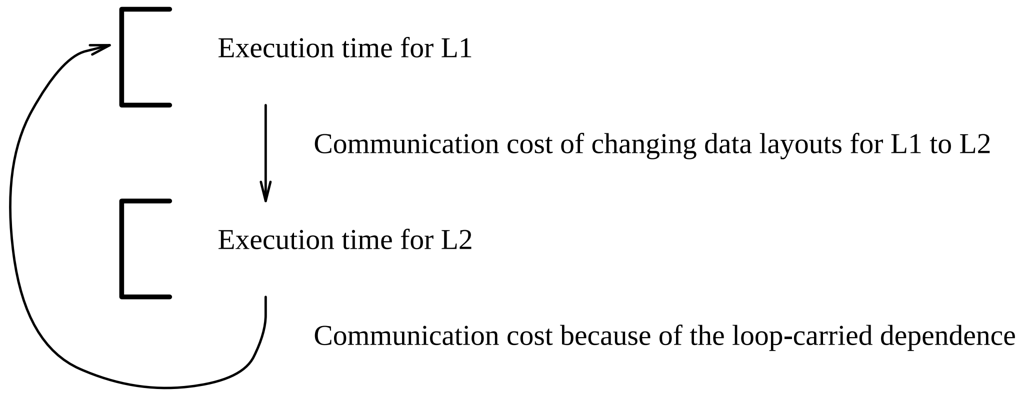

Figure 3: The total execution time for computing two Do-loops in an iteration.

Fig. 4 shows the component alignment for L_1 and L_2. In L_1, A_1 and V and B are mapped to dimension 1 of the processor grid; A_2 and X are mapped to dimension 2. In L_2, A_1 and V and B and X are mapped to dimension 1; A_2 is mapped to dimension 2. Suppose that the execution time for L_1 is Time_1 and for L_2 it is Time_2; the communication time for changing data layouts for L_1 to L_2 is CTime_1 and the communication time because of the loop-carried dependence is CTime_2. Then,

$$\begin{aligned}
\mathrm{Time}_1 &= 2 * \frac{m^2}{N_1 * N_2} * t_f + \mathrm{Reduction}(\tfrac{m}{N_1}, \{N_2 \text{ PEs}\}) \text{ (line 5)} \\
\mathrm{Time}_2 &= 3 * \frac{m}{N_1} * t_f \text{ (line 8)} \\
\mathrm{CTime}_1 &= 0 \text{ (anti-dependence of } X \text{ can be eliminated)} \\
\mathrm{CTime}_2 &= N_1 * \mathrm{OneToManyMulticast}(\tfrac{m}{N_1}, \{N_1 \text{ PEs}\}) \\
&\quad \text{(loop-carried dependence of } X \text{ from line 8 to line 5).}
\end{aligned}$$

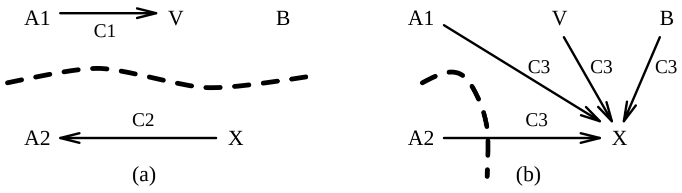

Figure 4: Component alignment for (a) lines 2–6 and (b) lines 7–9 of Jacobi's iterative algorithm

Note that, it is not necessary for sending or receiving data among processors for changing data layouts for L_1 to L_2 in this algorithm, because the anti-dependence relation of array X from line 5 to line 8 can be eliminated using variable renaming techniques. However, because the loop-carried dependence of array X from line 8 to line 5 must be considered, it is necessary for sending or receiving data among processors for changing data layouts for L_2 to L_1. The optimal execution time for L_1 is $2 * \frac{m^2}{N} * t_f$ when $N_1 = N$ and $N_2 = 1$. The optimal execution time for L_2 is $3 * \frac{m}{N} * t_f$ when $N_1 = N$ and $N_2 = 1$. The communication cost for updating array X because of the loop-carried dependence is $m * \log N * t_c$. Therefore, the total execution time for computing an iteration is $(2 * \frac{m^2}{N} + 3 * \frac{m}{N}) * t_f$ + $m * \log N * t_c$, which is better than the case using a different data distribution scheme in the last section. In the following, we list data distribution functions for each data array.

The data layouts for Do-loop L_1:

$$f_A(i,j) = (\lfloor\frac{i-1}{m/N}\rfloor);\ \ f_V(i) = (\lfloor\frac{i-1}{m/N}\rfloor);\ \ f_X(j) = (X).$$

The data layouts for Do-loop L_2:

$$f_A(i,j) = (\lfloor\frac{i-1}{m/N}\rfloor);\ \ f_V(i) = f_X(i) = f_B(i) = (\lfloor\frac{i-1}{m/N}\rfloor).$$

Note that, matrix A is of two dimensions and the physical target processor ring is of one dimension. The first dimension of A is distributed across the processor ring; the second dimension of A is not distributed. Next, because data array B is only used in Do-loop L_2, we only list a data distribution function of array B for Do-loop L_2; the data distribution function of array B for Do-loop L_1 is implicitly defined to be equal to that for Do-loop L_2.

Since this is our preferred implementation, we briefly describe it below. Table 5 shows the data layouts of the corresponding parallel Jacobi's iterative algorithm for implementing linear systems $A_{4\times4}X_4 = B_4$ on a four-processor linear array. The ith row of data array A and the ith elements of data arrays V, B, and X are stored in processor $i-1$. $V(i)$ and $X(i)$ are computed in processor $i-1$. After computing a new version of data array X, it broadcasts to all processors for further computation.

processor 0	$A_{11}\ A_{12}\ A_{13}\ A_{14}$	V_1	B_1	X_1	$(X_1\ X_2\ X_3\ X_4)$
processor 1	$A_{21}\ A_{22}\ A_{23}\ A_{24}$	V_2	B_2	X_2	$(X_1\ X_2\ X_3\ X_4)$
processor 2	$A_{31}\ A_{32}\ A_{33}\ A_{34}$	V_3	B_3	X_3	$(X_1\ X_2\ X_3\ X_4)$
processor 3	$A_{41}\ A_{42}\ A_{43}\ A_{44}$	V_4	B_4	X_4	$(X_1\ X_2\ X_3\ X_4)$

Table 5: Data layouts of the parallel Jacobi's iterative algorithm for implementing linear systems $A_{4\times4}X_4 = B_4$ on a four-processor linear array.

In general, a program contains s Do-loops or an iterative loop contains s Do-loops, and the problem of heuristically determining whether the data re-distribution is necessary can be solved by the following dynamic programming algorithm. Let L_1, L_2, ..., L_s be s Do-loops in sequence in the program. Let $M_{i,j}$ be the cost of computing the sequence of loops L_i, L_{i+1}, ..., L_{i+j-1} using the component-alignment algorithm, and $P_{i,j}$ be the distribution scheme, for $1 \le i \le s$ and $1 \le j \le s-i+1$. Define $T_{i,j}$ to be the cost of computing the sequence of loops L_1, L_2, ..., L_{i+j-1} with the restriction that the final data distribution scheme after computing $T_{i,j}$ is $P_{i,j}$. Initially, $T_{1,i}$ is equal to $M_{1,i}$.

Algorithm 1. A dynamic programming algorithm for computing the cost of data distribution schema of executing a sequence of s Do-loops on the distributed memory computer is presented.

Input: $M_{i,j}$, $P_{i,j}$, and $T_{1,i}$ $(= M_{1,i})$, where $1 \le i \le s$ and $1 \le j \le s-i+1$.

Output: The cost of executing s Do-loops on the distributed memory computer.

1. **for** $i := 2$ to s **do**
2. **for** $j := 1$ to $s-i+1$ **do**
3. $T_{i,j} := \mathrm{MIN}_{1 \le k < i}\{T_{i-k,k} + M_{i,j} + cost(P_{i-k,k},\, P_{i,j})\}$;
4. **end_for end_for**
5. $Minimum\_Cost := \mathrm{MIN}_{1 \le k \le s}\{T_{s-k+1,k} + loop\_carried\_dependence(T_{s-k+1,k})\}$.

Note: $cost(P_{i-k,k},\, P_{i,j})$ returns the communication cost of changing data layouts from $P_{i-k,k}$ to $P_{i,j}$. $loop\_carried\_dependence(T_{s-k+1,k})$ returns the communication cost incurred by the loop-carried dependence, if a sequence of distribution schema are used for computing $T_{s-k+1,k}$. For example, if a sequence of distribution schema P_{α_1,β_1}, P_{α_2,β_2}, ..., and $P_{s-k+1,k}$ are used for computing $T_{s-k+1,k}$, then $loop\_carried\_dependence(T_{s-k+1,k})$ returns the communication cost of changing data layouts from $P_{s-k+1,k}$ to P_{α_1,β_1}.

The data distribution schema obtained from Algorithm 1 are at least as good as any static data distribution schema, because the cost of the latter ones is equal to $T_{1,s}$. We now briefly analyze the algorithm. The time complexity of this dynamic programming algorithm is $O(s^3)$. However, before applying this dynamic programming algorithm, we need to compute $s(s+1)/2$ component alignment problems for the consecutive loops L_i, L_{i+1}, ..., L_{i+j-1}, where $1 \le i,\ j \le i+j-1 \le s$. Since, this is done at compiling time, it is still worth to implement this algorithm in order to improve the communication overhead at run time. We will show a complete example by applying this dynamic programming algorithm for determining data distribution in Section 6.

5 Improving the Communication Time by Pipelining Data

In general, because many scientific computation algorithms are regular, they can be compiled as an iterative loop, whose body includes the following four steps: (1) parallel computation step, (2) reduction step, (3) updating step, and (4) broadcast step. The reduction step and broadcast step normally use a lot of communication time and result in the idleness of processors. However, it is possible to pipeline data for implementing a reduction operation or a broadcast operation, and thus, the communication time can be saved.

Consider the parallel version of Jacobi's iterative algorithm in Section 4 again. In each iteration, it includes three steps: (1) a parallel computation step for calculating $V(i)$ (corresponding to line 5); (2) an updating step for modifying $X(i)$ (corresponding to line 8); and (3) a broadcast step for revising $X(j)$ (due to the loop-carried dependence of X from line 8 to line 5). Since, after each iteration, $X(i)$ is updated and will be sent to each processing element and will be used in the next iteration. In this case, it is possible to improve the communication time by pipelining variables $X(i)$ for all $1 \leq i \leq m$. Fig. 5 shows the pipelined implementation for solving linear systems $A_{16\times 16}X_{16} = B_{16}$ on a four-processor ring. Fig. 6 shows the parallel program in a processor. The total communication time in each iteration becomes $N_1 * \text{Transfer}(\frac{m}{N_1}) * t_c$ which is equal to $m * t_c$. And the total execution time in each iteration is $(2 * \frac{m^2}{N} + 3 * \frac{m}{N}) * t_f + m * t_c$ which is better than that of the parallel algorithm mentioned in Section 4. If the hardware supports overlaying the computation and the communication, the total execution time may reduce further.

5.1 Using Data-Dependence Information for Pipelining Data

We have shown that it is possible to generate a pipelined version of Jacobi's iterative algorithm. In effect, its rationale depends on investigating data-dependence information of the original sequential algorithm. Then, based on the data-dependence information, we can find a feasible iteration space mapping function which maps iterations to be executed at specific processing elements. This iteration space mapping function also can determine the direction of pipelining data.

In general, in a nested-loop program, data-dependence vectors can be defined in four ways: (1) modify-modify dependency (output dependence), which is the relation that a variable is generated

step	PROCESSOR 0	PROCESSOR 1	PROCESSOR 2	PROCESSOR 3
	V(1..4) B(1..4)	V(5..8) B(5..8)	V(9..12) B(9..12)	V(13..16) B(13..16)
1	A(1..4, 1..4) X(1..4)	A(5..8, 5..8) X(5..8)	A(9..12, 9..12) X(9..12)	A(13..16, 13..16) X(13..16)
2	A(1..4, 13..16) X(13..16)	A(5..8, 1..4) X(1..4)	A(9..12, 5..8) X(5..8)	A(13..16, 9..12) X(9..12)
3	A(1..4, 9..12) X(9..12)	A(5..8, 13..16) X(13..16)	A(9..12, 1..4) X(1..4)	A(13..16, 5..8) X(5..8)
4	A(1..4, 5..8) X(5..8)	A(5..8, 9..12) X(9..12)	A(9..12, 13..16) X(13..16)	A(13..16, 1..4) X(1..4)
5	update X(1..4)	update X(5..8)	update X(9..12)	update X(13..16)
	next iteration	next iteration	next iteration	next iteration

Figure 5: Variables $X(i)$ are pipelined during the computation of the Jacobi's iterative algorithm for solving linear systems $A_{16\times16}X_{16} = B_{16}$ on a four-processor ring. The first four steps accumulate $V(i)$, the fifth step updates $X(i)$.

```
{* Let m be the problem size, N be the number  *}
{*       of  processors, and block = m / N. *}
REAL A(block, m), V(block), X(block), B(block)
me = who_am_i()   {* Return current processor's ID.  *}
do 20 k = 1, MAX_ITERATION
    do 8 i = 1, block
        V(i) = 0.0
    continue
    do 17 ip = 0, N - 1
        send_to_right( X(1..block) )
        current_base = ((me - ip) mod N) * block
        do 15 i = 1, block
            do 15 j = 1, block
                V(i) = V(i) + A(i, current_base + j) * X(j)
        continue
        receive_from_left( X(1..block) )
    continue
    do 20 i = 1, block
        X(i) = X(i) + (B(i) - V(i)) / A(i, me * block + i)
    continue
```

Figure 6: Generated parallel codes for the Jacobi's iterative algorithm.

(modified) in different indexes; (2) use-use dependency (input dependence), which is the relation that a variable is used in different indexes; (3) modify-use dependency (flow dependence), which is the relation that a variable is generated in one index and will be used in the other index; and (4) use-modify dependency (anti-dependence), which is the relation that a variable is used in one index and will be generated in the other index.

Each different variable token symbol in the loop body causes one data-dependence vector of either modify-modify dependency, if the variable token symbol is on the left-hand-side of ':=', or use-use dependency, if the variable token symbol is on the right-hand-side of ':='. On the other hand, each pair of a modified token symbol (on the left-hand-side of :=) and a used token symbol (on the right-hand-side of :=) or each pair of a modified token symbol and the other modified token symbol of the same array variable also causes one data-dependence vector of either: modify-modify dependency; or modify-use dependency; or use-modify dependency. We will say that a nested-loop algorithm has regular data-dependence vectors, if all of the data-dependence vectors are constant. In our model, we only consider algorithms which have regular data-dependence vectors.

More precisely, in a p-nested-loop algorithm, the index of a variable token symbol $V(f_1, f_2, \ldots, f_e)$ can be represented by a linear function:

$$(f_1, f_2, \ldots, f_e)^t = R_v(i_1, i_2, \ldots, i_p)^t + \vec{g}_v,$$

where R_v is an $(e \times p)$ matrix, $\vec{g}_v$ is an $(e \times 1)$ vector, and entries of R_v and $\vec{g}_v$ are integers. First, the data-dependence vector with variable token symbol $V(f_1, f_2, \ldots, f_e)$ is the null space of R_v, $null(R_v)$. In this paper, we assume that the dimension of the null space of R_v is either 0 or 1. If the dimension of the null space of R_v is 0, then the trivial data-dependence vector with $V(f_1, f_2, \ldots, f_e)$ is $\vec{0}_p$, which means either $V(f_1, f_2, \ldots, f_e)$ is generated only once or $V(f_1, f_2, \ldots, f_e)$ is used only once. If the dimension of the null space of R_v is 1, then the data-dependence vector with $V(f_1, f_2, \ldots, f_e)$ is $null(R_v)$, which means that $V(f_1, f_2, \ldots, f_e)$ is first used or generated in some iteration $\bar{I}_0$, and will be periodically used or generated in all iterations $\bar{I}_0 + k * null(R_v)$, where k is a scalar variable, and $\bar{I}_0$ and $\bar{I}_0 + k * null(R_v)$ are in the iteration space. The case when the dimension of the null space of R_v is greater than 1, can be dealt with by using scalar expansion techniques to introduce new auxiliary variable token symbols.

Second, the data-dependence vector $\vec{z}$ between two distinct variable token symbols $V(f_1, f_2, \ldots, f_e)$ and $V(f'_1, f'_2, \ldots, f'_e)$ can be determined as follows. Let $(f_1, f_2, \ldots, f_e)^t = R_v(i_1, i_2, \ldots, i_p)^t + \vec{g}_v$ and $(f'_1, f'_2, \ldots, f'_e)^t = R'_v(i_1, i_2, \ldots, i_p)^t + \vec{g}'_v$. Assume that $(f_1, f_2, \ldots, f_e) = (f'_1, f'_2, \ldots, f'_e)$. Then, $\vec{z}$ $(= (z_1, z_2, \ldots, z_p)^t)$ is regular if and only if $R_v = R'_v$; $\vec{z}$ satisfies $R_v\vec{z} = \vec{g}_v - \vec{g}'_v$, where $0 \leq z_i \leq u_i - l_i$ for all $1 \leq i \leq p$, and l_i and u_i are respectively the lower bound and the upper bound for each level i of the loop; and in addition, $\vec{z}$ is the smallest integer vector which is larger than $\vec{0}_p$ in lexicographical order.

For Jacobi's iterative algorithm, each outmost iteration contains a two-dimensional iteration space, which is indexed by $(i, j)^t$. There are two non-trivial data-dependence vectors: $d_1 = (0, 1)^t$, which is with $V(i)$ in line 5; and $d_2 = (1, 0)^t$, which is with $X(j)$ in line 5. Note that, there is also one trivial data-dependence vector: $d_3 = (0, 0)^t$, which is with $A(i, j)$ in line 5. Let $e_1 = (1, 0)$ and $e_2 = (0, 1)$. Because our physical target machine is a 1-D processor array, the iteration space mapping function,

$\mathbf{S}_{1\times 2}$, which is a 1×2 row vector, is chosen to be either e_1 or e_2. In addition, either all $\mathbf{S}d_j$ are greater than or equal to zero or all $\mathbf{S}d_j$ are less than or equal to zero, for every data-dependence vector d_j. This is for avoiding communication deadlock. $e_1 = (1,0)$, which represents a set of parallel hyperplane $i = constant$; $e_2 = (0,1)$, which represents a set of parallel hyperplane $j = constant$. These two sets of hyperplanes divide the whole iteration space into disjoint tiles. In addition, each of these tiles which contains several iterations will be assigned to be executed at a specific processing element. Because the set of parallel hyperplanes $i = constant$ is more consistent with data distribution functions, as most data arrays are distributed along the first dimension of the abstract processor grid; we thus choose the transposition of its normal direction vector, $e_1 = (1,0)$, to represent the iteration space mapping function.

The iteration space mapping function, $(1,0)$, maps the iteration $(i,j)^t$ to the processing element $(\lfloor\frac{i-1}{m/N}\rfloor)$ according to data distribution functions, which means that the iteration $(i,j)^t$ will be executed at processing element $(\lfloor\frac{i-1}{m/N}\rfloor)$. $(1,0)$ maps the data-dependence vector d_1 $(= (0,1)^t)$ to 0, which means that $V(i)$ will be used or will be generated in a dedicated processing element throughout the computation. $(1,0)$ maps the data-dependence vector d_2 $(= (1,0)^t)$ to 1, which means that $X(j)$ will be used in the neighboring processing element in the next consecutive step. Therefore, $X(j)$ can be sent by pipelining to the neighboring processing element instead of using a broadcast operation to the neighboring processing element. Readers can refer to Fig. 6, in the pipelined version program, and broadcast operations are substituted by Shift operations (send and receive operations).

We can now summarize Section 5. Suppose that the target machine is a q-D grid. In general, in a p-nested loop program, the iteration space mapping function, $\mathbf{S}_{q\times p}$, which is a $q \times p$ matrix, consists of q different row vectors choosing from $\{e_1, e_2, \ldots, e_p\}$, where e_i is a p-tuple row vector, whose entries are all zero except that the i-th entry is equal to 1. In addition, for each i where $1 \le i \le q$, either all i-th entries of the vectors $\mathbf{S}d_j$ are greater than or equal to zero, or all i-th entries of the vectors $\mathbf{S}d_j$ are less than or equal to zero, for all data-dependence vector d_j. For example, suppose that e_{s_i} is the i-th row of $\mathbf{S}$. Then, either $e_{s_i}d_j \ge 0$ for all d_j, or $e_{s_i}d_j \le 0$ for all d_j. This is for avoiding communication deadlock.

The iteration space mapping function, $\mathbf{S} = (e_{s_1}, e_{s_2}, \ldots, e_{s_q})^t$, maps the iteration $(i_1, i_2, \ldots, i_p)^t$ to be executed at the processing element $(g_1(i_{s_1}), g_2(i_{s_2}), \ldots, g_q(i_{s_q}))$, where $g_j(i_{s_j})$ is the data distribution function along the j-th dimension of the q-D physical target processor grid. Then, based on the data-dependence information, we can determine whether or not a variable token will be used or will be generated in a dedicated processing element throughout the computation; or whether or not a variable token needs to be pipelined to a neighboring processing element. Note that, the iteration space mapping function should be consistent with data distribution functions generated by the method described in Section 4, otherwise it may incur additional communication overhead.

The idea of using iteration space mapping functions comes from mapping Do-loops into systolic arrays [32]. In the process of preparing the final manuscript, the author has learned that Ramanujam and Sadayappan have independently derived a method for tiling iterative space [38]. Their method is more general than the method presented in this paper. In [38], in order to minimize communication volume, the iteration space mapping functions may be not chosen from $\{e_1, e_2, \ldots, e_p\}$. However, this may incur additional efforts for mapping tiles into grid-connected machines. If the iteration space mapping functions are chosen from $\{e_1, e_2, \ldots, e_p\}$, their results are similar to ours.

6 Experimental Studies

In this section, we apply our method and implement two applications on a 32-node nCUBE-2 computer currently in Academia Sinica. In this computer, each node has 4 Mega bytes memory, runs at a modest clock rate of 20 MHz, and is rated at 7.5 MIPS (Mega Instructions Per Second), and 3.5 MFLOPS (Mega FLOating-point operations Per Second) in single precision arithmetic. Because of the memory limitation, each application only can be implemented up to a certain size.

6.1 Solving a Set of Linear Systems by Using Jacobi's Iterative Algorithm

In this section, we use an example to illustrate the dynamic programming algorithm. This example is to solve a set of linear system equations by using Jacobi's iterative algorithm.

$$\begin{cases} \left(a^n A + \sum_{h=0}^{n-1} a^h D\right) X_n = B & \text{for n = 1, 2, .., W} \\ \left(a^n A + \sum_{h=0}^{n-1} a^h D\right)^T Y_n = C & \text{for n = 1, 2, .., W} \end{cases}$$

where a and W are constant; h and n are scalar variables; A and D are two $m \times m$ matrices; and B and C are two $m \times 1$ vectors. The program is listed below.

```
1    DO 30 n = 1, W
2       DO 5 i = 1, m
3          DO 5 j = 1, m
4             A(i,j) = a * A(i,j) + D(i,j)
5       CONTINUE
6       DO 15 k = 1, MAX_ITERATION
7          DO 11 i = 1, m
8             U(i) = 0.0
9             DO 11 j = 1, m
10               U(i) = U(i) + A(i,j) * X(j)
11         CONTINUE
12         DO 14 i = 1, m
13            X(i) = X(i) + (B(i) - U(i)) / A(i,i)
14         CONTINUE
15      CONTINUE
16      DO 25 k = 1, MAX_ITERATION
17         DO 21 i = 1, m
18            V(i) = 0.0
19            DO 21 j = 1, m
20               V(i) = V(i) + A(j,i) * Y(j)
21         CONTINUE
22         DO 24 i = 1, m
23            Y(i) = Y(i) + (C(i) - V(i)) / A(i,i)
24         CONTINUE
25      CONTINUE
26      DO 29 i = 1, m
27         XD(n,i) = X(i)
28         YD(n,i) = Y(i)
29      CONTINUE
30   CONTINUE
```

We let line 2 to line 5 be loop L_1; line 6 to line 15 be loop L_2; line 16 to line 25 be loop L_3; and line 26 to line 29 be loop L_4. The component affinity graphs and their corresponding component alignment of these four loops are shown in Fig. 7. The communication cost of L_1 and L_4 each is zero. The communication cost of L_2 and L_3 each is C_m, where $C_m = K * m * t_c$ based on the pipelined version Jacobi's iterative algorithm mentioned in Section 5, and K is equal to the constant MAX_ITERATION. Suppose that the cost for performing a matrix transpose operation is C_T. Then, the communication cost $M'_{i,j}$ required for computing different sequences of consecutive loops by the component-alignment algorithm are shown in Table 6. Note that, $M_{i,j}$ is the total execution time of computing the sequence of loops L_i, L_{i+1}, ..., L_{i+j-1}. Since the computation of this parallel algorithm is load-balanced, we only need to consider the communication time $M'_{i,j}$ here, because the computation time for all different data distribution schema are the same. For simplicity, we use $M_{i,j}$ to represent $M'_{i,j}$.

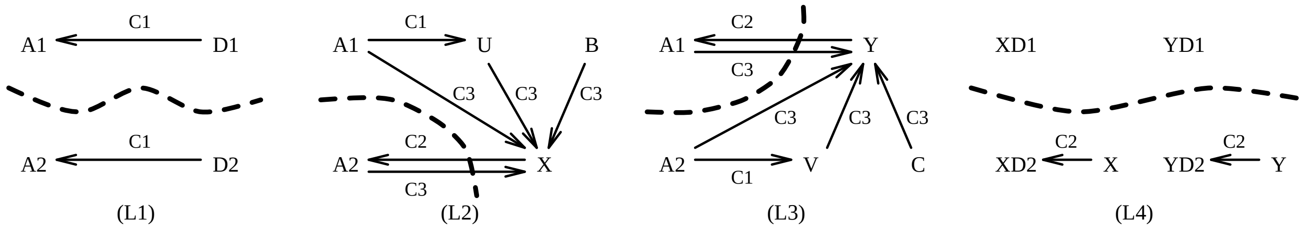

Figure 7: Component alignment graphs for L_1, L_2, L_3, and L_4, respectively.

M_{ij}	j = 1	j = 2	j = 3	j = 4
i = 1	0	C_m	$2C_m + K * C_T$	$2C_m + K * C_T$
i = 2	C_m	$2C_m + K * C_T$	$2C_m + K * C_T$	
i = 3	C_m	C_m		
i = 4	0			

Table 6: The communication cost required for computing various sequences of consecutive loops using the component-alignment algorithm. K is the constant MAX_ITERATION.

Similarly, $T_{i,j}$ will only represent the total communication time in this example. Thus, the communication cost required for computing the sequence of loops using our dynamic programming algorithm can be solved as shown in Table 7. From Table 7, we conclude that, in an outmost iteration, first, data layouts between L_1 and L_2 are not changed; next, a matrix transpose operation for matrix A is necessary before executing L_3; then, data layouts between L_3 and L_4 are not changed; finally, another matrix transpose operation for matrix A^T is necessary before the next iteration. In the following, we list data distribution functions for each data array.

The data layouts for Do-loops L_1 and L_2:

$$f_A(i,j) = f_D(i,j) = (\lfloor \frac{i-1}{m/N} \rfloor); \quad f_X(i) = f_U(i) = f_B(i) = (\lfloor \frac{i-1}{m/N} \rfloor).$$

The data layouts for Do-loops L_3 and L_4:

$$f_A(i,j) = f_{XD}(i,j) = f_{YD}(i,j) = (\lfloor \frac{j-1}{m/N} \rfloor); \quad f_X(i) = f_Y(i) = f_V(i) = f_C(i) = (\lfloor \frac{i-1}{m/N} \rfloor).$$

$$
\begin{aligned}
T_{11} &= M_{11} = 0\\
T_{12} &= M_{12} = C_m\\
T_{13} &= M_{13} = 2C_m + K * C_T\\
T_{14} &= M_{14} = 2C_m + K * C_T\\
T_{21} &= T_{11} + M_{21} + (cost(P_{11}, P_{21}) = 0) = C_m\\
T_{22} &= T_{11} + M_{22} + (cost(P_{11}, P_{22}) = 0) = 2C_m + K * C_T\\
T_{23} &= T_{11} + M_{23} + (cost(P_{11}, P_{23}) = 0) = 2C_m + K * C_T\\
T_{31} &= \mathrm{MIN}\{T_{21} + M_{31} + (cost(P_{21}, P_{31}) = C_T), T_{12} + M_{31} + (cost(P_{12}, P_{31}) = C_T)\}\\
&= 2C_m + C_T\\
T_{32} &= \mathrm{MIN}\{T_{21} + M_{32} + (cost(P_{21}, P_{32}) = C_T), T_{12} + M_{32} + (cost(P_{12}, P_{32}) = C_T)\}\\
&= 2C_m + C_T\\
T_{41} &= \mathrm{MIN}\{T_{31} + M_{41} + (cost(P_{31}, P_{41}) = 0), T_{22} + M_{41} + (cost(P_{22}, P_{41}) = 0),\\
&\qquad T_{13} + M_{41} + (cost(P_{13}, P_{41}) = 0)\}\\
&= 2C_m + C_T.
\end{aligned}
$$

T_{ij}	j = 1	j = 2	j = 3	j = 4
i = 1	0	C_m	$2C_m + K * C_T$	$2C_m + K * C_T$
i = 2	C_m	$2C_m + K * C_T$	$2C_m + K * C_T$	
i = 3	$2C_m + C_T$	$2C_m + C_T$		
i = 4	$2C_m + C_T$			

$$
\begin{aligned}
Minimum\_Cost &= \mathrm{MIN}_{1 \le k \le 4}\{T_{4-k+1,\,k} + loop\_carried\_dependence(T_{4-k+1,\,k})\}\\
&= \mathrm{MIN}\{T_{41} + C_T, T_{32} + C_T, T_{23} + 0, T_{14} + 0\}\\
&= 2C_m + 2C_T.
\end{aligned}
$$

Table 7: The communication cost required for computing various sequences of consecutive loops using our dynamic programming algorithm.

Table 8 lists experimental results of implementing the mentioned application with various problem sizes on a 32-node nCUBE-2 computer. In this experimental study, we implement two versions of parallel implementations: (1) based on a dynamic data distribution scheme; (2) based on a static data distribution scheme. We let the constant $W = 10$, and the constant MAX_ITERATION $= 20 * \log m$, where m is the problem size. It is clear that to use dynamic data distribution schema is better than to use static data distribution schema.

Note that, as mentioned in Section 3, we proposed a technique which, by adding an additional edge (in a component affinity graph) between two dimensions of arrays both in the right-hand side of an assignment statement if they have an affinity relation, will reduce the sending cost of right-hand-side operands to the owner of the left-hand side. We have applied this optimization technique (to lines from 7 to 11 or to lines from 17 to 21 of the sequential program) to implement the parallel version of the program based on a static data distribution scheme. On the other hand, for the parallel version of

matrix size	#PE = 2	#PE = 4	#PE = 8	#PE = 16	#PE = 32
$2^5 \times 2^5$	6 (6)	4 (4)	5 (6)	8 (10)	13 (21)
$2^6 \times 2^6$	24 (24)	14 (14)	11 (11)	12 (15)	18 (27)
$2^7 \times 2^7$	107 (109)	57 (58)	32 (35)	24 (29)	26 (38)
$2^8 \times 2^8$	481 (485)	244 (249)	128 (134)	74 (83)	55 (72)
$2^9 \times 2^9$	2142 (2162)	1079 (1092)	548 (562)	288 (305)	167 (193)

Table 8: The simulation time (in units of seconds) for solving a set of linear systems. The data that are not in parentheses are based on a dynamic data distribution scheme; the data in parentheses are based on a static data distribution scheme.

the program based on a dynamic data distribution scheme, most of the performance improvement is due to using a dynamic data distribution scheme, and the performance improvement due to pipelining data by using data-dependence information is less significant.

6.2 Gauss Elimination Algorithm for Linear Systems

Consider the following Gauss elimination algorithm for linear systems $A_{m\times m} X_m = B_m$.

```
1                 {* Matrix triangularization.  *}
2      DO 8 k = 1, m
3         DO 8 i = k + 1, m
4            L(i,k) = A(i,k) / A(k,k)
5            B(i) = B(i) − L(i,k) * B(k)
6            DO 8 j = k + 1, m
7               A(i,j) = A(i,j) − L(i,k) * A(k,j)
8      CONTINUE
9                 {* Triangular linear system UX = Y. *}
10     DO 12 i = m, 1, −1
11        V(i) = 0.0
12     CONTINUE
13     DO 17 k = m, 1, −1
14        X(k) = (B(k) - V(k)) / A(k,k)
15        DO 17 i = k − 1, 1, −1
16           V(i) = V(i) + A(i,k) * X(k)
17     CONTINUE
```

Fig. 8 shows the corresponding component affinity graph and the suggested component alignment. Although the program fragment from line 2 to line 8 prefers using a 2-D processor grid, the program fragment from line 13 to line 17 prefers to use a processor ring. In order to achieve a better load balance among processors, a processor ring is used. In addition, data arrays are partitioned along the first dimension. Because the index space includes an oblique pyramid and a triangle, for the sake of requiring load balance, contiguous-cyclical data distribution schema are thus used.

$$f_A(i,j) = f_L(i,j) = (\lfloor \frac{i-1}{block} \rfloor \bmod N); \quad f_V(i) = f_B(i) = f_X(i) = (\lfloor \frac{i-1}{block} \rfloor \bmod N).$$

Note that, matrix A is of two dimensions and the target processor ring is of one dimension. The first dimension of A is distributed across the processor ring; the second dimension of A is not distributed.

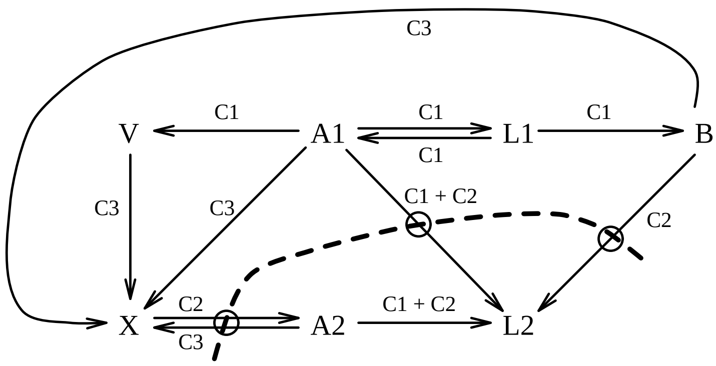

Figure 8: Component affinity graph and the suggested component alignment of the Gauss elimination algorithm

A naive compiler may generate a lot of OneToManyMulticast operations for broadcasting $B(k)$ in line 5, $A(k,j)$ in line 7, and $X(k)$ in line 16 to all processors in the ring for each distinct k and j. It will certainly incur excessive communication overhead. In effect, data communication for $B(k)$, $A(k,j)$, and $X(k)$ is due to the loop-carried dependence. A better method for generating efficient codes relies on investigating the data-dependence information of each data token.

Table 9 shows non-zero data-dependence vectors with their corresponding variable token symbols in either depth-two nested-loop program fragments (indexed by $(k,i)^t$) or depth-three nested-loop program fragments (indexed by $(k,i,j)^t$). Let the iteration space mapping function for depth-two nested-loop program fragments be $(0,1)$ and the iteration space mapping function for depth-three nested-loop program fragments be $(0,1,0)$. Then, for all variable tokens $B(i)$, $A(i,j)$, $L(i,k)$, and $V(i)$ will be used (or generated) in dedicated processing elements throughout the computation. In addition, for all variable tokens $B(k)$, $A(k,j)$, and $X(k)$ will be used in the neighboring processing element in the next consecutive step. However, they can be sent by pipelining to the neighboring processing element instead of using broadcast operations to the neighboring processing element.

token	line	dependence vector	mapping dependence-vector
$B(i)$	5	$(1,0)^t$	$(0,1)(1,0)^t = 0$
$B(k)$	5	$(0,1)^t$	$(0,1)(0,1)^t = 1$
$A(i,j)$	7	$(1,0,0)^t$	$(0,1,0)(1,0,0)^t = 0$
$L(i,k)$	7	$(0,0,1)^t$	$(0,1,0)(0,0,1)^t = 0$
$A(k,j)$	7	$(0,1,0)^t$	$(0,1,0)(0,1,0)^t = 1$
$V(i)$	16	$(1,0)^t$	$(0,1)(1,0)^t = 0$
$X(k)$	16	$(0,1)^t$	$(0,1)(0,1)^t = 1$

Table 9: Data-dependence vectors with each data token and the suggested iteration space mapping functions $(0,1)$ and $(0,1,0)$ respectively for depth-two and depth-three nested-loop program fragments in the Gauss elimination algorithm.

Table 10 lists experimental results of implementing the Gauss elimination algorithm for linear systems with various problem sizes and block sizes on a 32-node nCUBE-2 computer. In this experimental study, we implement two programs: (1) a pipelined version algorithm; (2) an algorithm using broadcast operations. It is clear that the pipelined version program runs faster than the program using broadcast operations.

(a) matrix size $2^9 \times 2^9$

block size	#PE = 2	#PE = 4	#PE = 8	#PE = 16	#PE = 32
$2^1 \times 2^1$	226 (222)	117 (118)	63 (68)	35 (46)	22 (36)
$2^2 \times 2^2$	134 (134)	69 (70)	36 (39)	20 (24)	12 (17)
$2^3 \times 2^3$	104 (105)	54 (56)	28 (31)	15 (19)	10 (13)
$2^4 \times 2^4$	92 (95)	48 (51)	27 (31)	15 (20)	9 (13)
$2^5 \times 2^5$	88 (92)	48 (54)	28 (34)	17 (24)	**

(b) matrix size $2^{10} \times 2^{10}$

block size	#PE = 2	#PE = 4	#PE = 8	#PE = 16	#PE = 32
$2^1 \times 2^1$	1767 (1744)	900 (898)	466 (485)	250 (286)	141 (194)
$2^2 \times 2^2$	1061 (1058)	537 (542)	276 (285)	144 (159)	79 (99)
$2^3 \times 2^3$	824 (826)	418 (424)	214 (224)	113 (124)	63 (75)
$2^4 \times 2^4$	726 (733)	372 (384)	195 (209)	105 (122)	59 (78)
$2^5 \times 2^5$	686 (701)	359 (381)	195 (222)	111 (140)	65 (95)

Table 10: The simulation time (in units of seconds) of running a Gauss elimination algorithm for linear systems. The data that are not in parentheses are obtained from a pipelined version algorithm; the data that are in parentheses are obtained from an algorithm using broadcast operations. Data are distributed in a contiguous-cyclic fashion.

Note that, in this experimental study, two versions of parallel implementations are based on the same static data distribution scheme. The proposed optimization technique mentioned in Section 3 is not applied in this example. Thus, the whole performance improvement is due to pipelining data by using data-dependence information.

7 Conclusions

We have presented in this paper several techniques for compiling Do-loop programs on distributed memory parallel computers. In Section 2, we have generalized Gupta and Banerjee's data distribution functions for 1-D and 2-D data arrays by additionally allowing distributed data to be indexed increasingly or decreasingly, and allowing the distributions of different data dimensions to be dependent or independent. Thus, the distribution schema are more rich than previously proposed. In Section 3, we extended Li and Chen's model for constructing component affinity graphs. First, we extended the definition of an affinity relation of two dimensions of arrays if two subscripts of these two dimensions are affine functions of the same (single) index variable of a Do-loop. Second, we added an additional edge between two dimensions of arrays both in the right-hand side of an assignment statement if they have an affinity relation. This modification will reduce sending cost of right-hand-side operands to the owner of the left-hand side. Third, we specified a direction for each edge which will be used to specify the direction of the data communication according to the owner computes rule. After that, we introduced communication primitives and defined approximate communication cost if data were sent from one data distribution to another data distribution.

In Section 4, we developed a dynamic programming algorithm for heuristically determining whether the data re-distribution is necessary. In Section 5, we considered the improvement of communication time by pipelining data. Especially, the data-dependence information which can be used to map iterations to be executed in specific processors also can provide enough information for pipelining data. Experimental studies have shown that our approach improves conventional methods.

From our experience, the codes generated by parallel compilers are influenced by the paradigms and techniques of designing parallel algorithms (for example, reduction, prefix sum, and pipelining), which are also influenced by the target parallel computer architectures. In general, advanced compiler techniques, such as loop interchanging, loop distribution, skewing, reversal, data blocking (strip miling), or data interleaving, can improve extracting parallelism in a sequential program [45, 46, 47]. A correct parallel program generated from a compiler must preserve the data-dependence relations of the original sequential program. Data-dependence relations will influence the strategies of selecting a good data distribution scheme, which will consequently influence the balanced load and the communication costs among processors.

Acknowledgements

The author would like to thank Prof. Y. Oyanagi and anonymous referees for their valuable comments on this paper.

References

[1] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. of ACM-SIGPLAN PLDI*, pages 112–125, Albuquerque, N.M., June 1993.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partition and distribution. In *Proc. of Fifth Distributed Memory Comput. Conf.*, pages 1160–1170, Charleston, SC, Apr. 1990.

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proc. ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 213–223, Williamsburg, VA, Apr. 1991.

[4] Jarle Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12:335–342, 1989.

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

[6] B. Chapman, T. Fahringer, and H. Zima. Automatic support for data distribution on distributed memory multiprocessor systems. In *Lecture Notes in Computer Science 768, Sixth International Workshop on Languages and Compilers for Parallel Computing*, pages 184–199, Portland, Oregon, Aug. 1993.

[7] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic data distributions in Vienna Fortran. In *Proc. of Supercomputing '93*, pages 284–293, Portland, Oregon, Nov. 1993.

[8] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S. H. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proc. ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 149–158, May 1993.

[9] S. Chatterjee, J. R. Gilbert, and R. Schreiber. Mobile and replicated alignment of arrays in data-parallel programs. In *Proc. of Supercomputing '93*, Nov. 1993.

[10] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng. Optimal evaluation of array expressions on massively parallel machines. Technical Report CSL-92-13, Xerox PARC, 1992.

[11] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. H. Teng. Automatic array alignment in data-parallel programs. In *Proc. of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 1993.

[12] M. Chen, Y. I. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 2:171–207, 1988.

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

[14] C. Gong, R. Gupta, and R. Melhem. Compilation techniques for optimizing communication on distributed-memory systems. In *Proc. International Conf. on Parallel Processing*, pages II–39–46, St. Charles, IL, Aug. 1993.

[15] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proc. International Parallel Processing Symposium*, pages 470–475, Beverly Hills, CA, Mar. 1992.

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

[17] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C. H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *Proc. International Conf. on Parallel Processing*, pages II–301–305, St. Charles, IL, Aug. 1993.

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

[19] P. D. Hovland and L. M. Ni. A model for automatic data partitioning. In *Proc. International Conf. on Parallel Processing*, pages II–251–259, St. Charles, IL, Aug. 1993.

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

[21] D. E. Hudak and S. G. Abraham. *Compiling Parallel Loops for High Performance Computers.* Kluwer Academic Publishers, Norwell, Massachusetts, 1993.

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

[23] K. Ikudome, G. C. Fox, A. Kolawa, and J. W. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proc. of Fifth Distributed Memory Comput. Conf.*, pages 1105–1114, Charleston, SC, Apr. 1990.

[24] E. T. Kalns, H. Xu, and L. M. Ni. Evaluation of data distribution patterns in distributed-memory machines. In *Proc. International Conf. on Parallel Processing*, pages II–175–183, St. Charles, IL, Aug. 1993.

[25] K. Kennedy and U. Kremer. Initial framework for automatic data layout in Fortran D: A short update on a case study. Technical Report CRPC-TR93324-S, Center for Research on Parallel Computation, Rice University, July 1993.

[26] K. Kennedy, K. S. Mckinley, and C. W. Tseng. Interactive parallel programming using the ParaScope editor. *IEEE Trans. Parallel Distributed Syst.*, 2(3):329–341, July 1991.

[27] C. Koelbel. Compile-time generation of regular communications patterns. In *Proc. of Supercomputing '91*, pages 101–110, Nov. 1991.

[28] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parallel Distributed Syst.*, 2(4):440–451, Oct. 1991.

[29] U. Kremer. NP-completeness of dynamic remapping. In *Proc. of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, Dec. 1993.

[30] U. Kremer. Automatic data layout using 0–1 integer programming. In *Proc. of International Conf. on Parallel Architectures and Compilation Techniques*, Montréal, Canada, Aug. 1994.

[31] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 136–152, Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993.

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

[33] P.-Z. Lee and T. B. Tsai. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proc. International Conf. on Parallel Processing*, pages II–161–165, St. Charles, IL, Aug. 1993, also Technical Report TR-93-004, Institute of Information Science, Academia Sinica.

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

[35] 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.

[36] M. Mace. *Memory Storage Patterns in Parallel Processing.* Kluwer Academic Publishers, Boston, MA, 1987.

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

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

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

[40] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21:150–159, 1994.

[41] E. Su, D. J. Palermo, and P. Banerjee. Automating parallelization of regular computations for distributed-memory multicomputers in the paradigm compiler. In *Proc. International Conf. on Parallel Processing*, pages II–30–38, St. Charles, IL, Aug. 1993.

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

[43] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proc. of International Conf. on Supercomputing*, July 1992.

[44] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. of ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, June 1991.

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

[46] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[47] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, Reading, MA, 1990.

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

[49] H. P. Zima, H-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.