TR-93-004

Compiling Efficient Programs for Tightly-Coupled Distributed Memory Computers

PeiZong Lee & Tzung-Bow Tsai

Compiling Efficient Programs for Tightly-Coupled Distributed Memory Computers[*]

PeiZong Lee[†] Institute of Information Science, Academia Sinica, Taipei, Taiwan, R.O.C.
Tzung-Bow Tsai, Dept. of EE, Chung-Cheng Univ., Chia-Yi, Taiwan, R.O.C.

April 8, 1993

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 a systematic method 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 derive a dynamic programming algorithm for data distribution. Second, we improve the communication time by pipelining data. Third, we use data-dependence information for pipelining data. Jacobi's iterative algorithm, a successive over-relaxation iterative algorithm, and the Gauss elimination algorithm for linear systems are used to illustrate our method.

*** A preliminary version of this technical report is accepted to be presented at the 22nd International Conference on Parallel Processing, St. Charles, IL, U.S.A., Aug. 16–20, 1993.

[*]This work was partially supported by the NSC under Grant NSC 81-0408-E-001-505 and NSC 82-0408-E-001-016.
[†]PeiZong Lee: leepe@iis.sinica.edu.tw, TEL: +886 (2) 788-3799, FAX: +886 (2) 782-4814.

1 Introduction

This paper is concerned with compiling efficient programs for tightly-coupled 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 [6]. 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.

It is our goal in this paper to present a systematic method 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 [14]. 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 [14], Gupta and Banerjee [8] formulated the component alignment problem from the whole source program. The 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 compute the minimum cost order of data distribution schema for executing a sequence of nested Do-loops in distributed memory computers.

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 three steps: (1) parallel computation step; (2) reduction step; and (3) updating step. The reduction step normally uses a lot of communication time and results in the idleness of processors. However, it is possible to pipeline data for implementing a reduction 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 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. 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 [4]. Hiranandani, Kennedy, and Tseng developed an enhanced version of Fortran called Fortran D, which allowed programmers to specify data decomposition [9]. 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 [13]. Balasundaram, Fox, Kennedy, and Kremer designed an interactive environment for guiding data partition and distribution, and for estimating performance [1, 2]. 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 [11], ParaScope [12], AL [16], SUPERB [19], and others.

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 [5]. 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 [14], and derived an algorithm for matching syntactic reference patterns with appropriate aggregate communication primitives [15]. 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 [7, 8].

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. In Section 6, we show how to use data-dependence information for pipelining data. 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 any distributed memory machine. For example, the q-D grid can be embedded into a hypercube computer using a binary reflected Gray code [10]. Gupta and Banerjee suggested that q is the maximum dimensionality of any data array used in the program [8]. 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 [8] [9] [15] [16]. 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. 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.

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

$$f_A(i) = \begin{cases} \lfloor \frac{d*i-offset}{block} \rfloor [\text{mod } N_{map(A)}] & \text{if } A \text{ is partitioned} \\ X & \text{if } A \text{ is replicated,} \end{cases}$$

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 location in the dimension $map(A)$ of the processor grid where $A(i)$ is stored. This distribution function, which is a generalization based on [8], can specify the following parameters: (1) method of distribution — whether the data array is partitioned across processors or replicated; (2) method of partition — contiguous or cyclic; (3) method of indexing — increasing or decreasing; (4) the grid dimension to which the data array is mapped; (5) the block size for distribution; and (6) the displacement 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

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

where x_1 and x_2 are obtained from the 1-D distribution function, d_1 and d_2 are in $\{-1,\ 1\}$. $f_A(i,j)$ returns the locations in 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 [8], 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 [3].

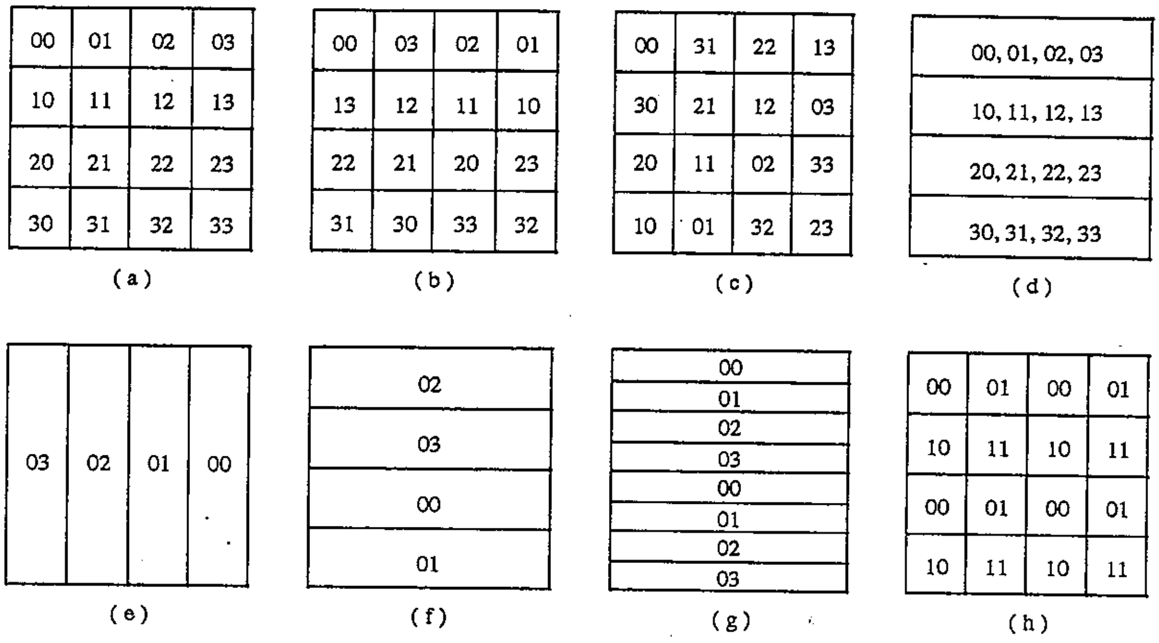

Figure 1: Data layouts for various data distribution schema

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 $map(A_2) = 2$.

(a)	$N_1 = 4,\ N_2 = 4$:	$f_A(i,j) = (\lfloor \frac{i-1}{4} \rfloor,\ \lfloor \frac{j-1}{4} \rfloor)$
(b)	$N_1 = 4,\ N_2 = 4$:	$f_A(i,j) = (\lfloor \frac{i-1}{4} \rfloor,\ (-\lfloor \frac{i-1}{4} \rfloor - \lfloor \frac{j-1}{4} \rfloor) \bmod 4)$
(c)	$N_1 = 4,\ N_2 = 4$:	$f_A(i,j) = ((-\lfloor \frac{i-1}{4} \rfloor - \lfloor \frac{j-1}{4} \rfloor) \bmod 4,\ \lfloor \frac{i-1}{4} \rfloor)$
(d)	$N_1 = 4,\ N_2 = 4$:	$f_A(i,j) = (\lfloor \frac{i-1}{4} \rfloor,\ X)$
(e)	$N_1 = 1,\ N_2 = 4$:	$f_A(i,j) = (0,\ \lfloor \frac{-j+16}{4} \rfloor)$
(f)	$N_1 = 4,\ N_2 = 1$:	$f_A(i,j) = (\lfloor \frac{i-9}{4} \rfloor \bmod 4,\ 0)$
(g)	$N_1 = 4,\ N_2 = 1$:	$f_A(i,j) = (\lfloor \frac{i-1}{2} \rfloor \bmod 4,\ 0)$
(h)	$N_1 = 2,\ N_2 = 2$:	$f_A(i,j) = (\lfloor \frac{i-1}{4} \rfloor \bmod 2,\ \lfloor \frac{j-1}{4} \rfloor \bmod 2)$

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. As in [8], we restrict the distribution in each remaining dimension either to a specific location or to be replicated along that dimension.

2.2 Interdependence of Communication Generation and Data Distribution

The following set of communication primitives is borrowed from [7] and [15].

- 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.

Li and Chen noticed that a data distribution scheme must be given before analyzing communication costs [15]. 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. First, assume $N_1 = N_2 = \cdots = N_q$, then a data distribution scheme can be determined by using the component-alignment algorithm. Second,

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.

formulate the total execution time including the computation time and the communication time; then the values of $N_1, N_2, \ldots, N_q$ 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 [14]. It is a directed 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 the difference of the two subscripts of these two dimensions is a constant value. 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 (q is the dimension of the abstract target grid) disjointed subsets 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 [8] [14]. In this paper, we assume that the abstract target grid is 2-dimensional, therefore, q is equal to 2.

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

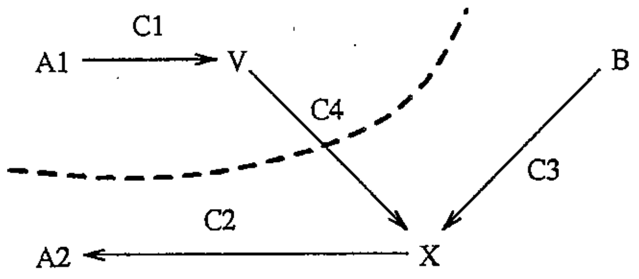

Figure 2: Component affinity graph of Jacobi's iterative algorithm

```
      {* X(i) has been assigned an initial value before the computation.  *}
1     DO 10 k = 1, MAX_ITERATION
2        DO 6 i = 1, m
3           V(i) = 0.0
4           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.

$$\begin{aligned}
c_1 &= \text{ManyToManyMulticast}(\tfrac{m^2}{N}, N) \text{ (line 5)} \\
c_2 &= \text{ManyToManyMulticast}(\tfrac{m}{N_1}, N_1) + \text{OneToManyMulticast}(m, N_2) \text{ (line 5)} \\
c_3 &= N_1 * \text{OneToManyMulticast}(\tfrac{m}{N_1}, N_2) \text{ (line 8)} \\
c_4 &= N_1 * \text{OneToManyMulticast}(\tfrac{m}{N_1}, N_2) \text{ (line 8)}
\end{aligned}$$

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_2 is greater than c_4. 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.

Next, we determine the partition strategy. As the iteration space is rectangular, the distribution

$N_1 \times N_2$	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 2: Computation time and communication time on three processor grids

functions for all array dimensions are determined to be contiguous:

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

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.

$$\begin{aligned} \text{Time} \;\; = \;\; & 2 * \frac{m^2}{N_1 * N_2} * t_f + \text{Reduction}(\tfrac{m}{N_1}, N_2) \text{ (line 5)} \\ & + 3 * \tfrac{m}{N_2} * t_f + N_1 * \text{OneToManyMulticast}(\tfrac{m}{N_1}, N_2) \\ & \text{(or } N_1 * \text{Transfer}(\tfrac{m}{N_1}) \text{ if } N_2 = 1) \text{ (line 8)} \\ & + \text{OneToManyMulticast}(\tfrac{m}{N_2}, N_1) \text{ (line 5 and line 8)} \\ & \text{(communication cost because of the loop-carried dependence of } X). \end{aligned}$$

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 2 shows the execution time on three processor grids. The costs of communication primitives are based on Table 1. 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 one in the next section.

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

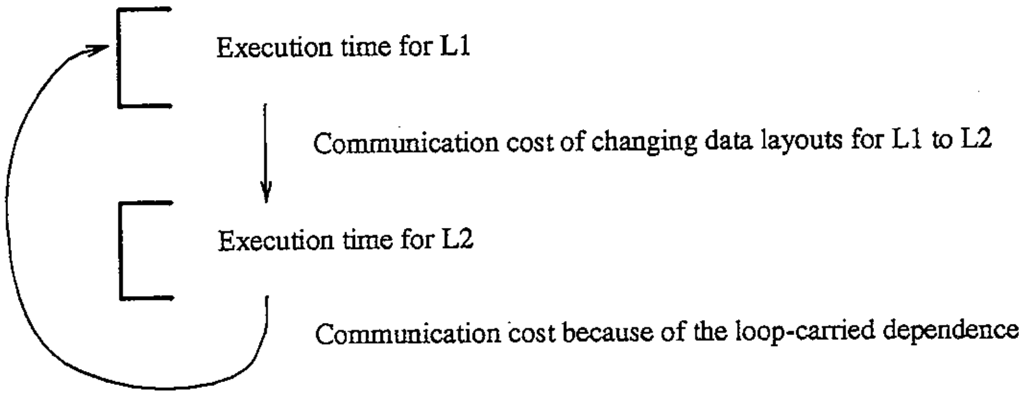
Execution time for L1
Communication cost of changing data layouts for L1 to L2
Execution time for L2
Communication cost because of the loop-carried dependence

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

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.

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,

$$\begin{array}{rcl} \text{Time}_1 & = & 2 * \frac{m^2}{N_1 * N_2} * t_f + \text{Reduction}(\frac{m}{N_1}, N_2) \text{ (line 5)} \\ \text{Time}_2 & = & 3 * \frac{m}{N_1} * t_f \text{ (line 8)} \\ \text{CTime}_1 & = & 0 \text{ (in this algorithm)} \\ \text{CTime}_2 & = & \text{ManyToManyMulticast}(\frac{m}{N_1}, N_1) + \text{OneToManyMulticast}(m, N_2) \\ & & \text{(loop-carried dependence of } X \text{ from line 8 to line 5).} \end{array}$$

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. 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 $m * t_c$. Therefore, the total execution time for computing an iteration is $(2 * \frac{m^2}{N} + 3 * \frac{m}{N}) * t_f + m * t_c$, which is better than the one using a different data distribution scheme in the last section.

Since this is our preferred implementation, we briefly describe it in below. Table 3 shows

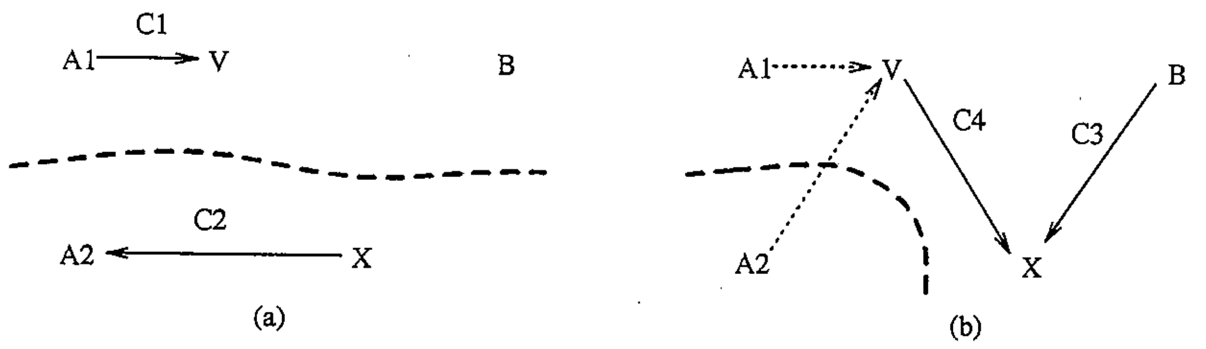

Figure 4: Component alignment for (a) lines 2–6 and (b) lines 7–9 of Jacobi's iterative algorithm

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 3: 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.

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.

In general, a program contains s Do-loops or an iterative loop contains s Do-loops, and the problem of finding the optimal execution time or the minimum cost order of data distribution schema can be obtained 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 minimum 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 $P_{i,j}$. Clearly, $T_{1,i}$ is equal to $M_{1,i}$.

Algorithm 1. A dynamic programming algorithm for computing the minimum cost order of data

distribution schema of executing a sequence of s Do-loops on the distributed memory computer.

Input: $M_{i,j}$ and $P_{i,j}$, where $1 \leq i \leq s$ and $1 \leq j \leq s-i+1$.

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

```
for $i := 2$ to $s$ do
    for $j := 1$ to $s - i + 1$ do
        $T_{i,j} := \mathrm{MIN}_{1 \leq k < i}(T_{i-k,k} + M_{i,j} + cost(P_{i-k,k}, P_{i,j}))$ ;
                /* $cost(P_{i-k,k}, P_{i,j})$ returns the communication cost of changing
                    data layouts from $P_{i-k,k}$ to $P_{i,j}$. */
$Minimum\_Cost := \mathrm{MIN}_{1 \leq k \leq s}(T_{s-k+1,k} + loop\_carried\_dependence(T_{s-k+1,k}))$ .
                /* $loop\_carried\_dependence(T_{s-k+1,k})$ returns the communication cost
                    incurred by the loop-carried dependence, if a sequence of
                    distribution schema is used for computing $T_{s-k+1,k}$. */
```

5 Improving the Communication Time by Pipelining Data

Consider the following successive over-relaxation (SOR) iterative algorithm, which can converge faster than 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. *}
1      DO 9 k = 1, MAX_ITERATION
2        DO 8 i = 1, m
3            V(i) = 0.0
4            DO 6 j = 1, m
5               V(i) = V(i) + A(i,j) * X(j)
6            CONTINUE
7            X(i) = X(i) + OMEGA * (B(i) - V(i)) / A(i,i)
8         CONTINUE
9      CONTINUE
```

The corresponding component affinity graph of this algorithm is the same as the one of Jacobi's iterative algorithm in Fig. 2, although the edge weights as shown below are different.

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

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

$$\begin{aligned}
c_1 &= m^2 * \text{Transfer}(1) \text{ (line 5)}\\
c_2 &= m * \text{OneToManyMulticast}(1, N) \text{ (line 5)}\\
c_3 &= m * \text{Transfer}(1) \text{ (line 7)}\\
c_4 &= m * \text{Transfer}(1) \text{ (line 7)}
\end{aligned}$$

Applying the component alignment algorithm to 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 the dimensions 1 and 2, respectively, of the processor grid. The partition strategy is the same as the one in Equation (1) in Section 3. 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 8 is formulated as follows.

$$\begin{aligned}
\text{Time} = m * \ & [2 * \tfrac{m}{N_2} * t_f + \text{Reduction}(1, N_2) \text{ (line5)}\\
& + 4 * t_f + \text{Transfer}(1) \text{ (line 7)}\\
& + \text{OneToManyMulticast}(1, N_1)] \text{ (line 5 and line 7)}\\
& \text{(communication cost because of the loop-carried dependence of } X).
\end{aligned}$$

When $N_1 = 1$ and $N_2 = N$, the execution time, $(2 * \frac{m^2}{N} + 4 * m) * t_f + m * (\log N + 1) * t_c$, is minimal. Table 4 shows the data layouts of the corresponding parallel SOR iterative algorithm for implementing linear systems $A_{4\times4}X_4 = B_4$ on a four-processor linear array. The jth column of data array A and the jth elements of data arrays B and X are stored in processor $j-1$. Data array V is replicated on all processors. A naive implementation can be as follows: at step i in an iteration, each processor computes $V(i)$ using local data; then, a reduction operation is used to compute the complete $V(i)$; finally, $X(i)$ is updated according to $B(i)$ and $V(i)$ in processor $i-1$.

The above naive algorithm reveals a clue that at step i in each iteration, $X(i)$ is updated and will be used in the continuous steps to update $V(j)$, for $i < j \leq m$ then $1 \leq j \leq i$, until at step i in the next iteration. In this case, it is possible to improve the communication time by pipelining variable $V(j)$ for all $1 \leq j \leq m$. Fig. 5 shows the pipelining implementation for solving

	PROCESSOR 0				PROCESSOR 1				PROCESSOR 2				PROCESSOR 3			
step	B(1) X(1)	B(2) X(2)	B(3) X(3)	B(4) X(4)	B(5) X(5)	B(6) X(6)	B(7) X(7)	B(8) X(8)	B(9) X(9)	B(10) X(10)	B(11) X(11)	B(12) X(12)	B(13) X(13)	B(14) X(14)	B(15) X(15)	B(16) X(16)
1	A(1,1)	A(1,2)	A(1,3)	A(1,4)												
2		A(2,2)	A(2,3)	A(2,4)	A(1,5)	A(1,6)	A(1,7)	A(1,8)								
3			A(3,3)	A(3,4)	A(2,5)	A(2,6)	A(2,7)	A(2,8)	A(1,9)	A(1,10)	A(1,11)	A(1,12)				
4				A(4,4)	A(3,5)	A(3,6)	A(3,7)	A(3,8)	A(2,9)	A(2,10)	A(2,11)	A(2,12)	A(1,13)	A(1,14)	A(1,15)	A(1,16)
5	X(1)				A(4,5)	A(4,6)	A(4,7)	A(4,8)	A(3,9)	A(3,10)	A(3,11)	A(3,12)	A(2,13)	A(2,14)	A(2,15)	A(2,16)
6	A(2,1)	X(2)			A(5,5)	A(5,6)	A(5,7)	A(5,8)	A(4,9)	A(4,10)	A(4,11)	A(4,12)	A(3,13)	A(3,14)	A(3,15)	A(3,16)
7	A(3,1)	A(3,2)	X(3)			A(6,6)	A(6,7)	A(6,8)	A(5,9)	A(5,10)	A(5,11)	A(5,12)	A(4,13)	A(4,14)	A(4,15)	A(4,16)
8	A(4,1)	A(4,2)	A(4,3)	X(4)			A(7,7)	A(7,8)	A(6,9)	A(6,10)	A(6,11)	A(6,12)	A(5,13)	A(5,14)	A(5,15)	A(5,16)
9	A(5,1)	A(5,2)	A(5,3)	A(5,4)				A(8,8)	A(7,9)	A(7,10)	A(7,11)	A(7,12)	A(6,13)	A(6,14)	A(6,15)	A(6,16)
10	A(6,1)	A(6,2)	A(6,3)	A(6,4)	X(5)				A(8,9)	A(8,10)	A(8,11)	A(8,12)	A(7,13)	A(7,14)	A(7,15)	A(7,16)
11	A(7,1)	A(7,2)	A(7,3)	A(7,4)	A(6,5)	X(6)			A(9,9)	A(9,10)	A(9,11)	A(9,12)	A(8,13)	A(8,14)	A(8,15)	A(8,16)
12	A(8,1)	A(8,2)	A(8,3)	A(8,4)	A(7,5)	A(7,6)	X(7)			A(10,10)	A(10,11)	A(10,12)	A(9,13)	A(9,14)	A(9,15)	A(9,16)
13	A(9,1)	A(9,2)	A(9,3)	A(9,4)	A(8,5)	A(8,6)	A(8,7)	X(8)			A(11,11)	A(11,12)	A(10,13)	A(10,14)	A(10,15)	A(10,16)
14	A(10,1)	A(10,2)	A(10,3)	A(10,4)	A(9,5)	A(9,6)	A(9,7)	A(9,8)				A(12,12)	A(11,13)	A(11,14)	A(11,15)	A(11,16)
15	A(11,1)	A(11,2)	A(11,3)	A(11,4)	A(10,5)	A(10,6)	A(10,7)	A(10,8)	X(9)				A(12,13)	A(12,14)	A(12,15)	A(12,16)
16	A(12,1)	A(12,2)	A(12,3)	A(12,4)	A(11,5)	A(11,6)	A(11,7)	A(11,8)	A(10,9)	X(10)			A(13,13)	A(13,14)	A(13,15)	A(13,16)
17	A(13,1)	A(13,2)	A(13,3)	A(13,4)	A(12,5)	A(12,6)	A(12,7)	A(12,8)	A(11,9)	A(11,10)	X(11)			A(14,14)	A(14,15)	A(14,16)
18	A(14,1)	A(14,2)	A(14,3)	A(14,4)	A(13,5)	A(13,6)	A(13,7)	A(13,8)	A(12,9)	A(12,10)	A(12,11)	X(12)			A(15,15)	A(15,16)
19	A(15,1)	A(15,2)	A(15,3)	A(15,4)	A(14,5)	A(14,6)	A(14,7)	A(14,8)	A(13,9)	A(13,10)	A(13,11)	A(13,12)				A(16,16)
20	A(16,1)	A(16,2)	A(16,3)	A(16,4)	A(15,5)	A(15,6)	A(15,7)	A(15,8)	A(14,9)	A(14,10)	A(14,11)	A(14,12)	X(13)			
21		The next iteration			A(16,5)	A(16,6)	A(16,7)	A(16,8)	A(15,9)	A(15,10)	A(15,11)	A(15,12)	A(14,13)	X(14)		
22						The next iteration			A(16,9)	A(16,10)	A(16,11)	A(16,12)	A(15,13)	A(15,14)	X(15)	
23										The next iteration			A(16,13)	A(16,14)	A(16,15)	X(16)
24														The next iteration		

Figure 5: The pipelining implementation of the SOR iterative algorithm for solving linear systems $A_{16\times 16}X_{16} = B_{16}$ on a four-processor ring.

linear systems $A_{16\times 16}X_{16} = B_{16}$ on a four-processor ring. Fig. 6 shows the parallel program in a processor. Note that the computation of $V(i)$ and the updating of $X(i)$ can be implemented in one computation step. After a careful study of the total execution time, we find that the average execution time of an iteration is at most $(m+N)*(2*\frac{m}{N}*t_f+2*t_c)$, which is equal to $(2*\frac{m^2}{N}+2*m)*t_f+2*(m+N)*t_c$ and is better than the above naive algorithm. If the hardware supports overlaying the computation and the communication, the total execution time may reduce further.

6 Using Data-Dependence Information for Pipelining Data

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
```

```
{* Let m be the problem size, N be the number *}
{*     of processors, and block = m / N. *}
REAL A(m, block), X(block), B(block), V(m)
me = who_am_i()   {* Return current processor's ID. *}
before = me * block
do 44 k = 1, MAX_ITERATION
   do 15 i = 1, before
      temp = 0.0
      do 11 j = 1, block
         temp = temp + A(i, j) * X(j)
      continue
      receive_from_left( V(i) )
      V(i) = V(i) + temp
      send_to_right( V(i) )
   continue
   do 23 i = 1, block
      current = before + i
      V(current) = 0.0
      do 21 j = i, block
         V(current) = V(current) + A(current, j) * X(j)
      continue
      send_to_right( V(current) )
   continue
   do 34 i = 1, block
      current = before + i
      temp = 0.0
      do 29 j = 1, i - 1
         temp = temp + A(current, j) * X(j)
      continue
      receive_from_left( V(current) )
      V(current) = V(current) + temp
      X(i) = X(i) + omega *
               ( B(i) - V(current) ) / A(current, i)
   continue
   do 43 i = (me + 1) * block + 1, m
      temp = 0.0
      do 39 j = 1, block
         temp = temp + A(i, j) * X(j)
      continue
      receive_from_left( V(i) )
      V(i) = V(i) + temp
      send_to_right( V(i) )
   continue
continue
```

Figure 6: Generated parallel codes for the SOR iterative algorithm.

```
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 j = m, 1, -1
14         X(j) = (B(j) - V(j)) / A(j,j)
15         DO 17 i = j - 1, 1, -1
16            V(i) = V(i) + A(i,j) * X(j)
17      CONTINUE
```

Fig. 7 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

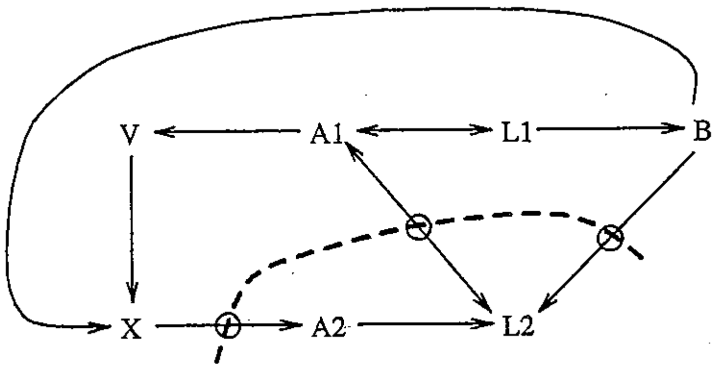

Figure 7: Component affinity graph and the suggested component alignment of the Gauss elimination algorithm

triangle, cyclical data distribution schema will be used.

$$f_A(i,j) = f_L(i,j) = ((i-1) \bmod N,\ 0);\ \ f_V(i) = f_B(i) = f_X(i) = (i-1) \bmod N.$$

A naive compiler may generate a lot of OneToManyMulticast operations for broadcasting $B(k)$ in line 5, $A(k,j)$ in line 7, and $X(j)$ in line 16 to all processors in the ring for each distinct k and j. It will certainly incurs excessive communication overhead.

In effect, data communication for $B(k)$, $A(k,j)$, and $X(j)$ is due to the loop-carried dependence. A better method for generating efficient codes relies on the data-dependence information of each data token. For instance, line 5 is in the body of a two-nested loop, token $B(k)$ was generated in index $(k-1,k)^t$ and is used in indices $(k,0)^t + i(0,1)^t$, for all $k+1 \leq i \leq m$. The data-dependence vector corresponding to token $B(k)$ is $(0,1)^t$. As we want to map index $(k,i)^t$ to be executed in the virtual processor i, the index-processor mapping is $(0,1)$. $(0,1)$ will map the data-dependence vector $(0,1)^t$ to 1, which means that $B(k)$ will be used in the neighboring processor in the next consecutive step. Therefore, $B(k)$ can be arranged by pipelining to the neighboring processor instead of broadcasting to the neighboring processor. The detailed data-dependence information of each data token and the suggested index-processor mapping can be seen in Table 5. Fig. 8 shows the parallel program in a processor. In the program, OneToManyMulticast operations are substituted by Shift operations (send and receive operations).

token	line	used in indices	virtual-PE mapping	dependence-vector mapping	used in PEs
$B(i)$	5	$(0,i)^t + k(1,0)^t$	$(0,1)((k,i)^t = i$	$(0,1)(1,0)^t = 0$	$(i-1) \bmod N$
$B(k)$	5	$(k,0)^t + i(0,1)^t$	$(0,1)((k,i)^t = i$	$(0,1)(0,1)^t = 1$	all PEs
$A(i,j)$	7	$(0,i,j)^t + k(1,0,0)^t$	$(0,1,0)(k,i,j)^t = i$	$(0,1,0)(1,0,0)^t = 0$	$(i-1) \bmod N$
$L(i,k)$	7	$(k,i,0)^t + j(0,0,1)^t$	$(0,1,0)(k,i,j)^t = i$	$(0,1,0)(0,0,1)^t = 0$	$(i-1) \bmod N$
$A(k,j)$	7	$(k,0,j)^t + i(0,1,0)^t$	$(0,1,0)(k,i,j)^t = i$	$(0,1,0)(0,1,0)^t = 1$	all PEs
$V(i)$	16	$(0,i)^t + j(1,0)^t$	$(0,1)(j,i)^t = i$	$(0,1)(1,0)^t = 0$	$(i-1) \bmod N$
$X(j)$	16	$(j,0)^t + i(0,1)^t$	$(0,1)(j,i)^t = i$	$(0,1)(0,1)^t = 1$	all PEs

Table 5: Data-dependence information of each data token and the suggested index-processor mapping of the Gauss elimination algorithm

```
1    {* Let m be the problem size, N be the number  *}
2    {*      of  processors, and block = m / N. *}
3    REAL A(block, m), L(block, m), X(block), B(block)
4    REAL V(block), Apipeline(m), Xpipeline, Bpipeline
5    me = who_am_i() {* Return  current processor's ID. *}
6               {* Matrix triangularization.  *}
7    do 15  k = 1, me
8       receive_from_left( Apipeline(k..m), Bpipeline )
9       send_to_right( Apipeline(k..m), Bpipeline )
10      do 15  i  = 1, block
11         L(i, k) = A(i, k) / Apipeline(k)
12         B(i) = B(i) - L(i, k) * Bpipeline
13         do 15  j  = (k + 1),  m
14            A(i, j) = A(i, j) - L(i, k) * Apipeline(j)
15   continue
16   do 34  k = (me + 1), m, N
17      pivot = ceiling(k / N)
18      send_to_right( A(pivot, k..m), B(pivot) )
19      do 24  i  = (pivot + 1),  block
20         L(i, k) = A(i, k) / A(pivot, k)
21         B(i) = B(i) - L(i, k) * B(pivot)
22         do 24  j = (k + 1), m
23            A(i, j) = A(i, j) - L(i, k) * A(pivot, j)
24      continue
25      receive_from_left( A(pivot, k..m), B(pivot) )
26      do 34  k1  = (k + 1), min(m, (k + N - 1))
27         receive_from_left( Apipeline(k1..m), Bpipeline )
28         send_to_right( Apipeline(k1..m), Bpipeline )
29         do 34  i = (pivot + 1), block
30            L(i, k1) = A(i, k1) / Apipeline(k1)
31            B(i) = B(i) - L(i, k1) * Bpipeline
32            do 34  j = (k1 + 1), m
33               A(i, j) = A(i, j) - L(i, k1) * Bpipeline
34   continue
35               {* Triangular linear system UX = Y. *}
36   do 38  i = block, 1, -1
37      V(i) = 0.0
38   continue
39   do 44   j = m, ((m - N + 1) + (me + 1)), -1
40      receive_from_right( Xpipeline )
41      send_to_left( Xpipeline )
42      do 44  i = block, 1, -1
43         V(i) = V(i) + A(i, j) * Xpipeline
44   continue
45   do 58  j = (m - N + me + 1), 1, -N
46      pivot = ceiling(j / N)
47      X(pivot) = (B(pivot) - V(pivot)) / A(pivot, j)
48      send_to_left( X(pivot) )
49      do 51 i = (pivot - 1), 1 -1
50         V(i) = V(i) + A(i, j) * X(pivot)
51      continue
52      receive_from_right( X(pivot) )
53      do 58  j1 = (j - 1), max(1, (j - N + 1)), -1
54         receive_from_right( Xpipeline )
55         send_to_left( Xpipeline )
56         do 58  i = (pivot - 1), 1, -1
57            V(i) = V(i) + A(i, j1) * Xpipeline
58   continue
```

Figure 8: Generated parallel codes for the Gauss elimination algorithm.

7 Conclusions

We have presented in this paper a systematic method for compiling Do-loop programs on distributed memory parallel computers. In Section 2, we have generalized 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 4, we developed a dynamic programming algorithm for distributing data which can compute the minimum cost order of data distribution schema for executing a sequence of nested Do-loops in distributed memory computers. In Section 5 and Section 6, we considered the improvement of communication time by pipelining data. Especially, in Section 6, 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.

From our experience, the codes generated by parallel compilers are influenced by the paradigms and techniques of designing parallel algorithms, which are also influenced by the target parallel computer architectures. In general, advanced compiler techniques, such as loop interchanging, loop distribution, data blocking (strip miling), or data interleaving, can improve extracting parallelism in an algorithm [17, 18]. A correct parallel algorithm generated from a compiler must preserve the data-dependence relations of the original sequential algorithm. 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.

References

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

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

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

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

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

[6] G. C. Fox. Prospects for parallel computing. SCCS 265, Syracuse University, 1992.

[7] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Int. Parallel Processing Symp.*, pages 179–193, Beverly Hills, CA, Mar. 1992. IEEE.

[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(2):179–193, Mar. 1992.

[9] 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.

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

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

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

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

[14] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: 3rd Symp. Frontiers Massively Parallel Computat.*, pages 424–433, College Park, MD, Oct. 1990. IEEE.

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

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

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

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

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