Efficient Algorithms for Data Distribution on Distributed Memory Parallel Computers

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

Data distribution has been one of the most important research topics in parallelizing compilers for distributed memory parallel computers. Good data distribution schema should consider both the computation load balance and the communication overhead. In this paper, we show that data re-distribution is necessary for executing a sequence of Do-loops if the communication cost due to performing this sequence of Do-loops is larger than a threshold value. Based on this observation, we can prune the searching space and derive efficient dynamic programming algorithms for determining effective data distribution schema to execute a sequence of Do-loops with a general structure. Experimental studies on a 32-node nCUBE-2 computer are also presented.

Keywords: component alignment, data distribution, distributed memory computer, Do-loops, dynamic programming algorithm for data distribution, parallelizing compiler.

1 Introduction

This paper is concerned with designing efficient algorithms for data distribution on distributed memory parallel computers. 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. A processor on the q-D grid is represented by the tuple $(p_1, p_2, \ldots, p_q)$, where $0 \le p_i \le N_i - 1$ for $1 \le i \le q$. Such a topology can be easily embedded into almost all distributed memory machines; for example, the q-D grid can be embedded into a hypercube computer using binary reflected Gray code encoding. The parallel program generated from a sequential program for a grid corresponds to the SPMD (Single Program Multiple Data) model, in which large data arrays are partitioned and distributed among processors, and in which each processor executes the same program but operates on distinct data items [11, 13, 30, 31, 38].

Given a sequence of s Do-loops with a general structure, we want to determine an effective data distribution schema for executing this sequence of Do-loops. This problem can be classified into three cases as shown in Fig. 1: (a) a sequence of s Do-loops; (b) a sequence of s Do-loops which are enclosed by an iterative loop; and (c) a sequence of s Do-loops with a general structure; among them, some consecutive Do-loops may be enclosed by iterative loops, which, again, with adjacent Do-loops, may be enclosed by other iterative loops, and so on.

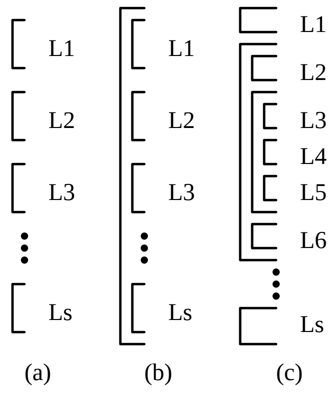

Figure 1: (a) a sequence of s Do-loops; (b) a sequence of s Do-loops which are enclosed by an iterative loop; and (c) a sequence of s Do-loops with a general structure.

This problem is quite important because many scientific programs are comprised of a sequence of

Do-loops or iterative loops, which may contain other sequences of Do-loops with a general structure. Thus, a naive data distribution scheme may result in excessive communication overhead on distributed memory parallel computers. For instance, when computing a 2-D fast Fourier transform (FFT) for a data matrix, we calculate a 1-D FFT for each row first, and then we evaluate a 1-D FFT for each column. If we adopt a fixed data distribution throughout the computation on a linear processor array, it will incur a certain communication overhead due to the requirement of several "bit-reverse shuffle-exchange" and "butterfly-pattern" data communications. However, if a compiler can perform a transpose operation for the matrix between calculating 1-D FFTs for all the rows and 1-D FFTs for all the columns, then no communication operations are required while each 1-D FFT is evaluated. A similar situation happens while partial differential equations are computed based on the 3-D FFT as included in the NASA/Ames numerical aerodynamics simulation benchmarks.

Data distribution has been one of the most important research topics in parallelizing compilers for distributed memory parallel computers. Mace first showed theoretically that a class of dynamic data layout problems for interleaved memory machines are NP-complete [33]. Anderson and Lam then presented another formulation of the dynamic data layout problem which was NP-hard [1]. Kremer also identified that the problem of dynamic data re-mapping (the inter-phase data layout problem) is NP-complete [23]. Li and Chen, in addition, proved that the problem of determining an optimal static alignment between the dimensions of distinct arrays is NP-complete [32].

Thus, in practice, previous parallelizing compiler research has emphasized allowing programmers to specify the data distribution using language extensions, such that compilers can then generate all the communication instructions according to these language extensions [3, 34, 42]. For instance, in High Performance Fortran (HPF), programmers have the obligation to provide *TEMPLATE*, *ALIGN*, and *DISTRIBUTE* directives to specify data distribution [22]. It is also possible to use compiler techniques to automatically determine data distribution of sequential programs on distributed memory systems. Li and Chen [32] and Gupta and Banerjee [11] formulated the component alignment problem from the whole source program and used it to determine data distribution. However, the fixed data distribution schema they derived may result in a larger communication overhead.

In addition, there have been other research works related to the compilation of programs on distributed-memory computers. Knobe, Lukas, Steele, and Natarajan provided algorithms for automatic alignment of arrays on SIMD machines [20, 21]. Chapman, Fahringer, Mehrotra, Moritsch, and Zima adopted Li and Chen's component alignment algorithm [32] for handling distributed data in Vienna Fortran [4]; in addition, they used a language extension to handle dynamic data distribution [5]. Kennedy, Kremer, Meller-Crummey, and Carle proposed an automatic data layout strategy which was implemented in their D programming tools [25]. Kremer also developed techniques for using 0-1 integer programming for automatic data layout in the inter-phase data layout problem [24]. Other papers, which addressed the problem of determining initial data distributions or distributions for temporaries, include [6] [7] [39].

Furthermore, Hovland and Ni determined data distribution using augmented data access descriptors [14]. Kalns, Xu, and Ni suggested a cost model for determining a small set of appropriate data distribution patterns among many possible choices [18]. Kalns and Ni proposed techniques for logical processor mapping that minimizes the total amount of data that must be communicated among processors [17]. Chen and Sheu [8], Huang and Sadayappan [15], Ramanujam and Sadayappan [35, 36], and Wolf and Lam [40, 41] determined the data distribution and/or degree of parallelism of a single nested loop based on the hyperplane method. In addition, Gong, Gupta, and Melhem [10] and Hudak and Abraham [16] developed compile-time techniques for optimizing communication overhead.

Like previous works in [11, 32], we will deal with the whole source program altogether; however, unlike them, we will deal with each Do-loop independently. Data distribution schema between two Do-loops may be different and may require some data communication between them. We found that if compilers adopt the *owner computes rule: the owner of the left-hand side element executes the assignment for that element*, to generate codes running on distributed memory machines, then data distribution schema determine both the computation load and the communication overhead among the processing elements (PEs). Because data re-distribution is expensive, it is a compromise to let several consecutive Do-loops share a common data distribution scheme. We will show that data re-distribution is necessary for executing a sequence of Do-loops if the communication cost due to performing this sequence of Do-loops is larger than a threshold value. Based on this observation, we can prune the searching space and derive dynamic programming algorithms which can determine effective data distribution schema for executing a sequence of Do-loops having a general structure.

The rest of this paper is organized as follows. In Section 2, we illustrate techniques for determining data distribution at compiling time and introduce a primitive dynamic programming algorithm for data distribution. In Section 3, we analyze two tables used and generated by the proposed dynamic programming algorithm. We prove that data re-distribution is necessary for executing a sequence of Do-loops if the communication cost due to performing this sequence of Do-loops is larger than a threshold value. In Section 4, we propose efficient algorithms for determining data distribution schema for executing a sequence of Do-loops having a general structure. In Section 5, we present experimental studies on a 32-node nCUBE-2 computer. Finally, some concluding remarks are given in Section 6.

2 Determining Data Distribution at Compiling Time

In this section, we will show how a component alignment algorithm can be used to determine data distribution. This method has also been adopted by other researchers [4, 11, 25, 32]. Because we will generalize previous methods to deal with a wider class of problems, in the following, we will describe this method in great detail.

We will first analyze the relationship between left-hand-side and right-hand-side array subscript reference patterns in the original sequential program. Based on pattern matching techniques, in Table 1, we specify communication primitives used in the SPMD program when right-hand-side objects are sent to the owner of the left-hand-side objects. These communication primitives were also adopted by [11, 31].

In Table 1, the communication primitive *Transfer* specifies that a message be sent from one processor to the other processor. *Shift* means a circular shift of data among neighboring processors along the specified grid dimension. *AffineTransform* indicates sending data from each processor on the specified grid dimension(s) to a distinct processor according to an affine transform. *OneToManyMulticast* represents sending a message to all processors on the specified dimension(s) of the processor grid. *Reduction* stands for reducing data using a simple associative and commutative operator over all the processors lying on the specified grid dimension(s). *Gather* means to receive a message from each processor lying on the specified grid dimension(s). *Scatter* means sending a different message to each processor lying

case	LHS	RHS	communication primitive	cost on hypercube
1	c_1	c_2	Transfer(m)	$O(m)$
2	i	$i \pm c$	†Shift(m)	$O(m)$
3	$f_1(i)$	$f_2(i)$	AffineTransform(m, seq)	‡need additional analysis
4	i	c	OneToManyMulticast(m, seq)	$O(m * \log num(seq))$
5	c	i	Reduction(m, seq)	$O(m * \log num(seq))$
6	i	unknown	Gather(m, seq)	$O(m * num(seq))$
7	unknown	i	Scatter(m, seq)	$O(m * num(seq))$
8	i or $f_3(i)$	j or $f_4(j)$	ManyToManyMulticast(m, seq)	$O(m * num(seq))$

† Only when data arrays in both sides have the same data distribution.

‡ [12, 19, 28] have presented algorithms for generating communication sets.

Table 1: Communication primitives used in the SPMD program when left-hand-side and right-hand-side array subscripts have certain specific patterns. i and j are loop indexing variables; c, c_1, and c_2 are constants at compiling time; "unknown" means that the value is unknown at compiling time; $f_1(i)$ and $f_2(i)$ are two affine functions of the form $s_1 * i + c_1$ and $s_2 * i + c_2$, respectively; $f_3(i)$ and $f_4(j)$ are two functions of i and j, respectively. 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.

on the specified grid dimension(s). Finally, *ManyToManyMulticast* represents replication of data from all processors on the specified grid dimension(s) to themselves.

Readers can find that Case 2 is a special case of Case 3. There exist algorithms for generating communication sets for Case 3 [12, 19, 28]. Therefore, in the following, we will say that two array subscripts have an *affinity* relation if these two subscripts are affine functions of the same (single) index variable of a Do-loop. As to the costs of Case 6 through Case 8, they are considerably higher than those of Case 1 through Case 5.

2.1 Determining Alignments of Arrays' Dimensions

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 as shown in Case 3 of Table 1. Edges are defined in two ways. First, if the subscripts of the dimensions of the array (or matrix) on the left-hand-side of '=' have affinity relations with the subscripts of the dimensions of the array(s) on the right-hand-side of '=', then there are edges between corresponding pairs of dimensions. Second, if two right-hand-side arrays (or matrices) are the corresponding two operands of a binary operator, and if some pairs of subscripts of dimensions of these two arrays have affinity relations, and if in addition, none of subscripts in these two arrays have affinity relations with those of the left-hand-side array (or matrix), then there are edges between corresponding pairs of dimensions of these two arrays.

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 may be larger than the dimension of the physical target grid), so that the total weight of the 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.

Although the component alignment problem is NP-complete, Li and Chen have proposed an efficient heuristic algorithm [32]. 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 being undirected. The directions of edges, which indicate parent-child relations, can be used to determine block sizes of data distribution so that communication sets can be represented by closed forms [29]. The directions of edges also 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. For completeness, in Fig. 2, we present a very brief version of the component alignment algorithm; however, interested readers can refer to the original paper for details about this method [32].

The above mentioned q disjointed subsets is used to determine data distributions for all data arrays. For each subset, all matched nodes (dimensions of arrays) are assigned the same data distribution, such as *block*, *cyclic*, *block-cyclic* (*cyclic(b)*), *replicated* (the data array is replicated across processors), or *not distributed* (the data array is stored in a specific processor) [11, 27].

A heuristic component alignment algorithm:

Step 1: Construct a component affinity graph from the source program;

Step 2: choose a (high-dimensional) array with a highest dimensionality; thus, this array has the maximum number of nodes in the graph, and let its corresponding nodes in the graph become the initial basic set;

Step 3: **while** the remaining graph is not empty, **do**

- **Step 3.1** choose an array with highest dimensionality from the remaining graph;
- **Step 3.2** apply the optimal matching procedure to a bipartite graph constructed from the basic set and the nodes corresponding to components (dimensions) of the newly selected array;
  /* All disjointed subsets of matched nodes represent a partition. */
- **Step 3.3** combine the matched nodes with the basic set as a new basic set.

Figure 2: Heuristic component alignment algorithm.

There are two oracles to help decide the block size b. The load balance oracle suggests using *cyclic* (*cyclic*(1)) distribution if the iteration space is a pyramid (such as the iteration space of an LU decomposition), a triangle (such as the iteration space of a triangular linear system), or any other non-rectangular space. The communication oracle emphasizes not making the block size too small; otherwise it will incur a high communication overhead and a high indexing overhead. These two oracles, unfortunately, are inconsistent.

Based on *cyclic*(b) distribution, we can, however, formulate the total execution time from the SPMD program which includes both the computation time and the communication time [28]. The total execution time T is a function of the problem size m, the number of PEs N, and the block size b. When the problem size m and the number of PEs N are fixed, the optimal execution time can be obtained by requiring that $\frac{\partial T}{\partial b} = 0$ or by substituting all possible b into the formula. Alternatively, compilers may include a knowledge base, which contains an analytical model and certain experienced data distributions, and which can help determine the grain and granularity of execution space [2].

2.2 Determining Whether Data Re-distribution is Necessary

Like the case of computing a 2-D FFT as mentioned in Section 1, it is reasonable to assume that the optimal data distributions for single Do-loops may be different from one another in a sequence of Do-loops which perform computation-intensive scientific applications. In the following, we will introduce a dynamic programming algorithm to determine whether data re-distribution is necessary.

Suppose that a program contains s Do-loops, L_1, L_2, ..., L_s, in sequence as shown in Fig. 1-(a). Let $M_{i,j}$ be the cost of computing the sequence of Do-loops L_i, L_{i+1}, ..., L_{i+j-1} using the component-alignment algorithm, and let $P_{i,j}$ be the distribution scheme, for $1 \le i \le s$ and $1 \le j \le s-i+1$. Define $T_{i,j}$ as the cost of computing the sequence of Do-loops L_1, L_2, ..., L_{i+j-1} with the restriction that it uses the distribution scheme $P_{i,j}$ to compute Do-loops L_i, L_{i+1}, ..., L_{i+j-1}. Thus, the final data distribution scheme after computing $T_{i,j}$ is $P_{i,j}$. Initially, $T_{1,j}$ is equal to $M_{1,j}$. $cost(P_{i-k,\,k},\, P_{i,\,j})$ returns the communication cost of changing data layouts from $P_{i-k,\,k}$ to $P_{i,\,j}$.

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

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. $Minimum\_Cost := \mathrm{MIN}_{1 \le k \le s}\{T_{s-k+1,\,k}\}$.

If a sequence of s Do-loops is enclosed by an iterative loop as shown in Fig. 1-(b), then line 5 in Algorithm 1 can be modified in the following as Algorithm 1′:

$$5'.\ \ Minimum\_Cost' := \mathrm{MIN}_{1 \le k \le s}\{T_{s-k+1,\,k} + loop\_carried\_dependence(T_{s-k+1,\,k})\},$$

where $loop\_carried\_dependence(T_{s-k+1,\,k})$ returns the communication cost incurred by the loop-carried dependence. For example, in an iterative loop, if a sequence of distribution schema $P_{\lambda_1,\,\mu_1}$, $P_{\lambda_2,\,\mu_2}$, ..., and $P_{s-k+1,\,k}$ are used in computing $T_{s-k+1,\,k}$, then $loop\_carried\_dependence(T_{s-k+1,\,k})$ returns the communication cost of changing the data layouts from $P_{s-k+1,\,k}$ to $P_{\lambda_1,\,\mu_1}$.

Algorithm 1 (and Algorithm 1$'$) can be regarded as finding a single-source shortest path in a weighted graph. In this weighted graph, there are two virtual nodes and $\frac{s(s+1)}{2}$ physical nodes. Two virtual nodes include one source and one sink. $\frac{s(s+1)}{2}$ physical nodes $n_{i,j}$ are numbered as i and j, where $1 \le i \le s$ and $1 \le j \le s-i+1$. The node weight, edges, and edge weight of this graph are defined as follows. (1) The weight of each of two virtual nodes is zero. (2) The weight of node $n_{i,j}$ is $M_{i,j}$. (3) The source has s edges connected to nodes $n_{1,j}$, and the weight of each of these edges is zero, for $1 \le j \le s$, respectively. (4) The sink, which also has s edges, is connected by nodes $n_{i,(s-i+1)}$, and the weight of each of these edges is also zero, for $1 \le i \le s$, respectively. Also, (5) node $n_{i,j}$ has $s-(i+j)+1$ edges connected to nodes $n_{(i+j),k}$, and the weight of each of these edges is $cost(P_{i,j},\ P_{(i+j),\,k})$, for $(i+j) \le s$ and $1 \le k \le s-(i+j)+1$, respectively. Then, Algorithm 1 is equivalent to finding the shortest path from source to sink such that the sum of the node weight and edge weight in each of these paths is a minimum. Fig. 3 shows the corresponding single-source shortest path problem for $s=5$.

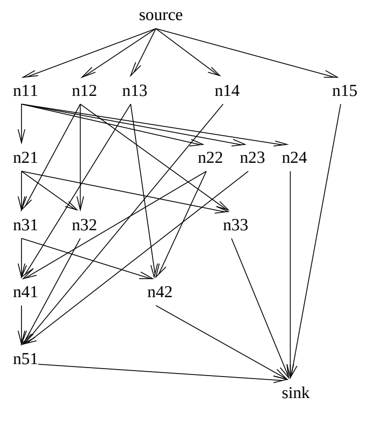

Figure 3: The corresponding single-source shortest path problem for $s=5$.

The sequence of data distribution schema obtained from Algorithm 1 is at least as good as any static data distribution scheme because the cost of any static data distribution scheme is equal to $T_{1,s}$. We now briefly analyze Algorithm 1. 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 Do-loops $L_i,\ L_{i+1},\ \ldots,\ L_{i+j-1}$, where $1 \le i,\ j \le i+j-1 \le s$.

It is instructive to compare our method with the one in [25]. In [25], the authors first explored several possible data layouts for each program phase, and they then defined the communication cost between candidate data layouts of adjacent phases. The problem of finding dynamic data layouts for the entire program is, thus, also reduced to a single-source shortest path problem. However, they did not show how to decide the length of each program phase. In our experience, the communication overhead due to component alignments of mismatched arrays is much higher than the communication overhead due to selecting a different block size from $cyclic(b)$. The way to choose a block size b can be determined by an analytical model or by certain experienced data distributions. If b_1 is close to b_2, the difference between $cyclic(b_1)$ and $cyclic(b_2)$ is not significant. Thus, if each program phase has only one data layout, then their method corresponds to the leftmost path in our solution space (as shown in Fig. 3).

2.3 An Example

In the following, we will use a complete example to illustrate how the above dynamic programming algorithm can be applied to determine data distribution. Suppose that the problem size is m, and that the number of PEs used is N. Consider the program in Fig. 4, which will be executed on a linear processor array. Let line 2 to line 6 be loop L_1, line 7 to line 19 be loop L_2, line 20 to line 24 be loop L_3, line 25 to line 37 be loop L_4, and line 38 to line 41 be loop L_5. The component affinity graphs and the corresponding component alignment of the loops from L_i to L_j, where $1 \leq i \leq j \leq 5$, are shown in Fig. 5. The weight of an edge is defined as follows. Because the topology of our target machine is a linear array, if the corresponding array's dimensionality on the tail of an edge is 1, then the weight of that edge is defined as ManyToManyMulticast(m/N, $\{N$ PEs$\}$). If the corresponding array's dimensionality on the tail of an edge is 2, then the weight of that edge is defined as ManyToManyMulticast(m^2/N, $\{N$ PEs$\}$).

Suppose that the average time of computing a floating point operation is t_f, and that the average time of transferring a word is t_c. Then, depending on whether matrix A is distributed row by row or distributed column by column, Table 2 shows the approximate computation time and the communication time of executing various consecutive loops, where certain cost coefficients are defined below:

$$
\begin{array}{ll}
K = MAX\_ITERATION & C_1 = m * t_c \\
C_a = (4m/N) * t_f & C_2 = K * C_1 \\
C_b = 4m * t_f & C_3 = m^2 * t_c \\
C_c = (m^2/N) * t_f & C_4 = C_2 + C_3 \\
C_d = m^2 * t_f & C_5 = K * C_3 \\
C_e = K * ((5m^2 + 3m)/N) * t_f & C_6 = 3m * t_c \\
C_f = K * (2m^2 + (3m^2/N) + 3m) * t_f & C_7 = K * (m^2 + 2m * (\log N) + m) * t_c.
\end{array}
$$

	matrix A is distributed row by row		matrix A is distributed column by column	
loops	computation time	communication time	computation time	communication time
L_1	C_c	0	C_d	C_3
L_2	C_e	C_2	C_f	C_7
L_3	C_d	C_3	C_c	0
L_4	C_f	C_7	C_e	C_2
L_5	C_a	0	C_b	C_6
$L_1 - L_2$	$C_c + C_e$	C_2	$C_d + C_f$	$C_3 + C_7$
$L_2 - L_3$	$C_e + C_d$	$C_2 + C_3$	$C_f + C_c$	C_7
$L_3 - L_4$	$C_d + C_f$	$C_3 + C_7$	$C_c + C_e$	C_2
$L_4 - L_5$	$C_f + C_a$	$C_7 + C_1$	$C_e + C_b$	$C_2 + C_6$
$L_1 - L_3$	$C_c + C_e + C_d$	$C_2 + C_3$	$C_d + C_f + C_c$	$C_3 + C_7$
$L_2 - L_4$	$C_e + C_d + C_f$	$C_2 + C_3 + C_7$	$C_f + C_c + C_e$	$C_7 + C_2$
$L_3 - L_5$	$C_d + C_f + C_a$	$C_3 + C_7 + C_1$	$C_c + C_e + C_b$	$C_2 + C_6$
$L_1 - L_4$	$C_c + C_e + C_d + C_f$	$C_2 + C_3 + C_7$	$C_d + C_f + C_c + C_e$	$C_3 + C_7 + C_2$
$L_2 - L_5$	$C_e + C_d + C_f + C_a$	$C_2 + C_3 + C_7 + C_1$	$C_f + C_c + C_e + C_b$	$C_7 + C_2 + C_6$
$L_1 - L_5$	$C_c + C_e + C_d + C_f + C_a$	$C_2 + C_3 + C_7 + C_1$	$C_d + C_f + C_c + C_e + C_b$	$C_3 + C_7 + C_2 + C_6$

Table 2: Computation time and communication time for various consecutive loops.

Data distributions are determined based on choosing a smaller total execution time. Table 3 lists the data distribution functions of each data array for various consecutive loops. As the iteration space is rectangular, to keep load balance and avoid calculating the index overhead, "block" distributions are chosen for all array dimensions. The data distribution function $f_X(i) = p$ means that the entry i of the one-dimensional data array X, $X(i)$, is stored in PE p. The data distribution function $f_A(i, j) = p$ means that the entry (i, j) of the two-dimensional data matrix A, $A(i, j)$, is stored in PE p.

Suppose that the cost of performing a matrix transpose operation based on the cascade sum algorithm [9], C_T, is $(m^2/N) * (\log N) * t_c$; in addition, C_T is very small in comparison with C_7 and $C_6 < C_T < C_7$. Then, $M_{i,j}$, $T_{i,j}$, and the expected execution time required to compute the sequence of s $(= 5)$ Do-loops can be solved by Algorithm 1 as shown in Table 4.

loops	$P_{i,j}$	data distribution functions
L_1	$P_{1,1}$	$f_A(i,j) = f_X(i) = (\lfloor \frac{i-1}{m/N} \rfloor)$
L_2	$P_{2,1}$	$f_A(i,j) = f_X(i) = f_B(i) = f_C(i) = (\lfloor \frac{i-1}{m/N} \rfloor)$
L_3	$P_{3,1}$	$f_A(i,j) = f_Y(j) = (\lfloor \frac{j-1}{m/N} \rfloor)$
L_4	$P_{4,1}$	$f_A(i,j) = f_Y(j) = f_U(j) = f_V(j) = (\lfloor \frac{j-1}{m/N} \rfloor)$
L_5	$P_{5,1}$	$f_A(i,j) = f_{XD}(k,i) = f_{YD}(k,i) = f_X(i) = f_Y(i) = (\lfloor \frac{i-1}{m/N} \rfloor)$
$L_1 - L_2$	$P_{1,2}$	$f_A(i,j) = f_X(i) = f_B(i) = f_C(i) = (\lfloor \frac{i-1}{m/N} \rfloor)$
$L_2 - L_3$	$P_{2,2}$	$f_A(i,j) = f_X(i) = f_B(i) = f_C(i) = (\lfloor \frac{i-1}{m/N} \rfloor)$; $f_Y(j) = 1$
$L_3 - L_4$	$P_{3,2}$	$f_A(i,j) = f_Y(j) = f_U(j) = f_V(j) = (\lfloor \frac{j-1}{m/N} \rfloor)$
$L_4 - L_5$	$P_{4,2}$	$f_A(i,j) = f_{XD}(j,k) = f_{YD}(j,k) = f_Y(j) = f_U(j) = f_V(j) = (\lfloor \frac{j-1}{m/N} \rfloor)$; $f_X(i) = 1$
$L_1 - L_3$	$P_{1,3}$	$f_A(i,j) = f_X(i) = f_B(i) = f_C(i) = (\lfloor \frac{i-1}{m/N} \rfloor)$; $f_Y(j) = 1$
$L_2 - L_4$	$P_{2,3}$	$f_A(i,j) = f_Y(j) = f_U(j) = f_V(j) = (\lfloor \frac{j-1}{m/N} \rfloor)$; $f_X(i) = f_B(i) = f_C(i) = 1$
$L_3 - L_5$	$P_{3,3}$	$f_A(i,j) = f_{XD}(j,k) = f_{YD}(j,k) = f_Y(j) = f_U(j) = f_V(j) = (\lfloor \frac{j-1}{m/N} \rfloor)$; $f_X(i) = 1$
$L_1 - L_4$	$P_{1,4}$	$f_A(i,j) = f_X(i) = f_B(i) = f_C(i) = (\lfloor \frac{i-1}{m/N} \rfloor)$; $f_Y(j) = f_U(j) = f_V(j) = 1$
$L_2 - L_5$	$P_{2,4}$	$f_A(i,j) = f_{XD}(j,k) = f_{YD}(j,k) = f_Y(j) = f_U(j) = f_V(j) = (\lfloor \frac{j-1}{m/N} \rfloor)$; $f_X(i) = f_B(i) = f_C(i) = 1$
$L_1 - L_5$	$P_{1,5}$	$f_A(i,j) = f_{XD}(k,i) = f_{YD}(k,i) = f_X(i) = f_B(i) = f_C(i) = (\lfloor \frac{i-1}{m/N} \rfloor)$; $f_Y(j) = f_U(j) = f_V(j) = 1$

Table 3: Data distributions for various consecutive loops.

From Table 4, based on Algorithm 1, in total, $C_b + 2C_c + 2C_e + C_1 + 2C_2 + C_6 + C_T$ time is required to execute an iteration of the outermost loop. In addition, $(P_{1,2}, P_{3,3})$ is a candidate sequence of the data distribution schema for an outermost iteration. That is, 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, L_4, and L_5 are not changed. However, under this sequence of data layouts, another matrix transpose operation for matrix A^T which requires C_T communication time is necessary before the next iteration.

Alternatively, based on Algorithm 1$'$, in total, $C_a + 2C_c + 2C_e + 2C_2 + 2C_T$ time is required to execute an iteration of the outermost loop. In addition, $(P_{1,2}, P_{3,2}, P_{5,1})$ is a candidate sequence of the data distribution schema for an outermost iteration. That is, 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; after that, another matrix transpose operation for matrix A^T is necessary before executing L_5. This result is better than that of Algorithm 1 because no data communication is necessary for transferring data layouts from $P_{5,1}$ to $P_{1,2}$.

$$
\begin{array}{lll}
M_{11} = C_c = \beta & M_{12} = C_c + C_e + C_2 = \epsilon & M_{23} = C_c + C_e + C_f + C_2 + C_7 = \nu \\
M_{21} = C_e + C_2 = \gamma & M_{22} = C_d + C_e + C_2 + C_3 = \theta & M_{33} = C_b + C_c + C_e + C_2 + C_6 = \zeta \\
M_{31} = C_c = \beta & M_{32} = C_c + C_e + C_2 = \epsilon & M_{14} = C_c + C_d + C_e + C_f + C_2 + C_3 + C_7 = \rho \\
M_{41} = C_e + C_2 = \gamma & M_{42} = C_b + C_e + C_2 + C_6 = \delta & M_{24} = C_b + C_c + C_e + C_f + C_2 + C_6 + C_7 = \xi \\
M_{51} = C_a = \alpha & M_{13} = C_c + C_d + C_e + C_2 + C_3 = \iota & M_{15} = C_a + C_c + C_d + C_e + C_f + C_1 \\
 & & \qquad + C_2 + C_3 + C_7 = \sigma
\end{array}
$$

$$
\begin{aligned}
T_{11} &= M_{11} = \beta \\
T_{12} &= M_{12} = \epsilon \\
T_{13} &= M_{13} = \iota \\
T_{14} &= M_{14} = \rho \\
T_{15} &= M_{15} = \sigma \\
T_{21} &= T_{11} + M_{21} + (cost(P_{11}, P_{21}) = 0) = C_c + C_e + C_2 = \epsilon \\
T_{22} &= T_{11} + M_{22} + (cost(P_{11}, P_{22}) = 0) = C_c + C_d + C_e + C_2 + C_3 = \iota \\
T_{23} &= T_{11} + M_{23} + (cost(P_{11}, P_{23}) = C_T + C_1) = 2C_c + C_e + C_f + C_2 + C_7 + C_T + C_1 = o \\
T_{24} &= T_{11} + M_{24} + (cost(P_{11}, P_{24}) = C_T + C_1) = C_b + 2C_c + C_e + C_f + C_2 + C_6 + C_7 + C_T + C_1 = \pi \\
T_{31} &= \mathrm{MIN}\{T_{21} + M_{31} + (cost(P_{21}, P_{31}) = C_T), T_{12} + M_{31} + (cost(P_{12}, P_{31}) = C_T)\} = 2C_c + C_e + C_2 + C_T = \eta \\
T_{32} &= \mathrm{MIN}\{T_{21} + M_{32} + (cost(P_{21}, P_{32}) = C_T), T_{12} + M_{32} + (cost(P_{12}, P_{32}) = C_T)\} = 2C_c + 2C_e + 2C_2 + C_T = \kappa \\
T_{33} &= \mathrm{MIN}\{T_{21} + M_{33} + (cost(P_{21}, P_{33}) = C_T + C_1), T_{12} + M_{33} + (cost(P_{12}, P_{33}) = C_T + C_1)\} \\
&= C_b + 2C_c + 2C_e + 2C_2 + C_6 + C_T + C_1 = \lambda \\
T_{41} &= \mathrm{MIN}\{T_{31} + M_{41} + (cost(P_{31}, P_{41}) = 0), T_{22} + M_{41} + (cost(P_{22}, P_{41}) = C_T + C_1), \\
&\qquad T_{13} + M_{41} + (cost(P_{13}, P_{41}) = C_T + C_1)\} = 2C_c + 2C_e + 2C_2 + C_T = \kappa \\
T_{42} &= \mathrm{MIN}\{T_{31} + M_{42} + (cost(P_{31}, P_{42}) = C_1), T_{22} + M_{42} + (cost(P_{22}, P_{42}) = C_T + 2C_1), \\
&\qquad T_{13} + M_{42} + (cost(P_{13}, P_{42}) = C_T + 2C_1)\} = C_b + 2C_c + 2C_e + 2C_2 + C_6 + C_T + C_1 = \lambda \\
T_{51} &= \mathrm{MIN}\{T_{41} + M_{51} + (cost(P_{41}, P_{51}) = C_T), T_{32} + M_{51} + (cost(P_{32}, P_{51}) = C_T), \\
&\qquad T_{23} + M_{51} + (cost(P_{23}, P_{51}) = C_T + C_1), T_{14} + M_{51} + (cost(P_{14}, P_{51}) = C_1)\} \\
&= C_a + 2C_c + 2C_e + 2C_2 + 2C_T = \mu.
\end{aligned}
$$

$$
\begin{aligned}
Minimum\_Cost &= \mathrm{MIN}_{1 \le k \le 5}\{T_{5-k+1,\,k}\} \\
&= \mathrm{MIN}\{T_{51}, T_{42}, T_{33}, T_{24}, T_{15}\} \\
&= T_{33} = C_b + 2C_c + 2C_e + C_1 + 2C_2 + C_6 + C_T = \lambda.
\end{aligned}
$$

$$
\begin{aligned}
Minimum\_Cost' &= \mathrm{MIN}_{1 \le k \le 5}\{T_{5-k+1,\,k} + loop\_carried\_dependence(T_{5-k+1,\,k})\} \\
&= \mathrm{MIN}\{T_{51} + 0, T_{42} + C_T + C_1, T_{33} + C_T + C_1, T_{24} + C_T + C_1, T_{15} + 0\} \\
&= T_{51} = C_a + 2C_c + 2C_e + 2C_2 + 2C_T = \mu.
\end{aligned}
$$

Table 4: Apply Algorithm 1 and Algorithm 1′ to the sample program. The values of M_{ij} and T_{ij} are represented by Greek letters and will be used again in Table 5.

2.4 More Details about Data Distribution

This subsection could appear immediately after introducing Algorithm 1; however, we think that it is more appropriate to present an example first. In this subsection, we will describe the data distribution for each data array in $P_{i,j}$ in detail.

As readers can see from Fig. 5, the component affinity graph and the corresponding component alignment for each Do-loop only deal with data arrays which are used in that Do-loop. Therefore, if a data array is used in L_i, L_{i+1}, ..., L_{i+j-1}, then its data distribution can be determined based on the results derived by the component alignment algorithm and is defined in $P_{i,j}$. However, if a data array is not used in L_i, L_{i+1}, ..., L_{i+j-1}, then, after applying the component alignment algorithm, its data distribution in $P_{i,j}$ is not defined. In the following, we use a heuristic method to assign a data distribution in $P_{i,j}$ for each data array if this data array is not used in L_i, L_{i+1}, ..., L_{i+j-1}.

This heuristic method includes two phases. The first phase is applied during constructing the $(P_{i,j})$-table. Suppose that a data array is not used in the first $e-1$ Do-loops, and that it is used in the e-th Do-loop, for $e > 1$. First, we implicitly assume that its data distribution while computing the first $e-1$ Do-loops is the same as that defined in the e-th Do-loop. Therefore, if $i+j-1 < e$, then the data distribution in $P_{i,j}$ for this data array is defined as being the same as the one defined in $P_{e,1}$. Second, if this data array is not used in a Do-loop, we also implicitly assume that its data distribution is not changed during the computation. Therefore, if $i > e$ and this data array is not used in L_i, L_{i+1}, ..., L_{i+j-1}, then the data distribution in $P_{i,j}$ for this data array is defined as being the same as the one defined in $P_{(i-1),1}$. For instance, suppose that in the first three Do-loops, a data array is only used in the second Do-loop. Then, its data distribution while computing the first Do-loop and the third Do-loop is the same as that defined in the second Do-loop.

The second phase is applied after performing Algorithm 1. After performing Algorithm 1, we have found a sequence of distribution schema P_{λ_1,μ_1}, P_{λ_2,μ_2}, ..., P_{λ_ξ,μ_ξ}, for computing a sequence of s Do-loops. Suppose that a data array is first used in L_{λ_f}, L_{λ_f+1}, ..., $L_{\lambda_f+\mu_f-1}$; then, its data distribution in P_{λ_f,μ_f} is determined based on the results derived by the component alignment algorithm. First, for $i < f$, this data array is not used in L_{λ_i}, L_{λ_i+1}, ..., $L_{\lambda_i+\mu_i-1}$; thus, we can let its data distribution in P_{λ_i,μ_i} be the same as that defined in P_{λ_f,μ_f}. Second, for $i > f$, if this data array is not used in L_{λ_i},

L_{λ_i+1}, ..., $L_{\lambda_i+\mu_i-1}$, then its data distribution in P_{λ_i,μ_i} is defined as being the same as that defined in $P_{\lambda_{i-1},\mu_{i-1}}$.

For instance, if we adopt $(P_{1,2}, P_{3,3})$ as the sequence of data distribution schema for computing the sample program, although data arrays B and C are not used in L_3, L_4, and L_5, their data distributions while computing L_3, L_4, and L_5 are the same as those defined in the first two Do-loops. Similarly, although data arrays XD, YD, Y, U, and V are not used in L_1 and L_2, their data distributions while computing L_1 and L_2 are the same as those defined in the last three Do-loops.

3 Behavior of the $(M_{i,j})$-table and $(T_{i,j})$-table

We will now analyze the behavior of the $(M_{i,j})$-table and $(T_{i,j})$-table. We have found that $M_{i,(\gamma+1)} \geq M_{i,\gamma}$ and $T_{i,(\gamma+1)} \geq T_{i,\gamma}$, for $1 \leq \gamma < s-i+1$. We define *THRESHOLD* as a value that is equal to four times the maximal communication cost between any two distribution schema. The reason why we define *THRESHOLD* as this value will be made clear in Theorem 2 and will be discussed again in Section 3.2.

3.1 Main Theorems

We want to show that if $M_{i,(\gamma+1)} > (M_{i,\beta} + M_{(i+\beta),(\gamma-\beta+1)} + \textit{THRESHOLD})$, for some β where $1 \leq \beta < \gamma+1$, then it is better to use three distribution schema $P_{i,\beta}$, $P_{(i+\beta),(\gamma-\beta+1)}$, and $P_{(i+\gamma+1),(j-\gamma-1)}$ to compute the sequence of Do-loops L_i, L_{i+1}, ..., L_{i+j-1}, than to use only one distribution scheme $P_{i,j}$ for $\gamma+1 < j \leq s-i+1$. Therefore, we **need not** compute $M_{i,j}$. Based on this observation, we can show that $T_{i,(\gamma+1)} > T_{(i+\beta),(\gamma-\beta+1)}$ and $T_{i,j} > T_{(i+\gamma+1),(j-\gamma-1)}$. Therefore, we **need not** compute $T_{i,j}$, for $\gamma+1 \leq j \leq s-i+1$.

Theorem 1 : *If $M_{i,(\gamma+1)} > M_{i,\beta} + M_{(i+\beta),(\gamma-\beta+1)} + \textit{THRESHOLD}$, for some β where $1 \leq \beta < \gamma+1$, then the following three cases are true for $\gamma+1 < j \leq s-i+1$.*

1. $M_{i,j} > M_{i,\beta} + M_{(i+\beta),(\gamma-\beta+1)} + M_{(i+\gamma+1),(j-\gamma-1)} + cost(P_{i,\beta}, P_{(i+\beta),(\gamma-\beta+1)})$
   $+ cost(P_{(i+\beta),(\gamma-\beta+1)}, P_{(i+\gamma+1),(j-\gamma-1)})$;
2. $T_{i,(\gamma+1)} > T_{(i+\beta),(\gamma-\beta+1)}$;

3. $T_{i,j} > T_{(i+\gamma+1),\,(j-\gamma-1)}$.

Proof: **1**. Consider the computation of Do-loops L_i, L_{i+1}, ..., $L_{(i+\beta-1)}$, $L_{(i+\beta)}$, ..., $L_{(i+\gamma)}$, $L_{(i+\gamma+1)}$, ..., L_{i+j-1}. Let $cost\_\delta_1$ be the cost of computing the sequence of Do-loops L_i, L_{i+1}, ..., $L_{(i+\gamma)}$ using the distribution scheme $P_{i,j}$. Then, $cost\_\delta_1$ is at least as large as $M_{i,\,(\gamma+1)}$. Thus, $cost\_\delta_1 > M_{i,\beta} + M_{(i+\beta),\,(\gamma-\beta+1)} + \mathit{THRESHOLD}$. Let $cost\_\delta_2$ be the cost of computing the sequence of Do-loops $L_{(i+\gamma+1)}$, ..., L_{i+j-1} using the distribution scheme $P_{i,j}$. Then, $cost\_\delta_2$ is at least as large as $M_{(i+\gamma+1),\,(j-\gamma-1)}$. Therefore,

$$\begin{aligned}
M_{i,j} &= cost_\delta_1 + cost_\delta_2 \\
&> M_{i,\beta} + M_{(i+\beta),\,(\gamma-\beta+1)} + M_{(i+\gamma+1),\,(j-\gamma-1)} + \mathit{THRESHOLD} \\
&\geq M_{i,\beta} + M_{(i+\beta),\,(\gamma-\beta+1)} + M_{(i+\gamma+1),\,(j-\gamma-1)} + cost(P_{i,\beta},\, P_{(i+\beta),\,(\gamma-\beta+1)}) \\
&\quad + \; cost(P_{(i+\beta),\,(\gamma-\beta+1)},\, P_{(i+\gamma+1),\,(j-\gamma-1)}).
\end{aligned}$$

2.

$$\begin{aligned}
& T_{i,\,(\gamma+1)} \\
= \;& \mathrm{MIN}_{1\leq k<i}\{T_{i-k,\,k} + M_{i,\,(\gamma+1)} + cost(P_{i-k,\,k},\, P_{i,\,(\gamma+1)})\} \\
> \;& \mathrm{MIN}_{1\leq k<i}\{T_{i-k,\,k} + M_{i,\beta} + M_{(i+\beta),\,(\gamma-\beta+1)} + \mathit{THRESHOLD}\} \\
\geq \;& \mathrm{MIN}_{1\leq k<i}\{T_{i-k,\,k} + M_{i,\beta} + M_{(i+\beta),\,(\gamma-\beta+1)} + cost(P_{i-k,\,k},\, P_{i,\beta}) + cost(P_{i,\beta},\, P_{(i+\beta),\,(\gamma-\beta+1)})\} \\
\geq \;& T_{(i+\beta),\,(\gamma-\beta+1)}.
\end{aligned}$$

3. Let $cost\_\delta_1$ and $cost\_\delta_2$ be the same as those defined in the first case. Then,

$$\begin{aligned}
T_{i,j} &= \mathrm{MIN}_{1\leq k<i}\{T_{i-k,\,k} + M_{i,j} + cost(P_{i-k,\,k},\, P_{i,j})\} \\
&\geq \mathrm{MIN}_{1\leq k<i}\{T_{i-k,\,k} + cost_\delta_1 + cost_\delta_2\} \\
&> \mathrm{MIN}_{1\leq k<i}\{T_{i-k,\,k} + M_{i,\beta} + M_{(i+\beta),\,(\gamma-\beta+1)} + M_{(i+\gamma+1),\,(j-\gamma-1)} + \mathit{THRESHOLD}\} \\
&\geq \mathrm{MIN}_{1\leq k<i}\{T_{i-k,\,k} + M_{i,\beta} + M_{(i+\beta),\,(\gamma-\beta+1)} + M_{(i+\gamma+1),\,(j-\gamma-1)} + cost(P_{i-k,\,k},\, P_{i,\beta}) \\
&\quad + \; cost(P_{i,\beta},\, P_{(i+\beta),\,(\gamma-\beta+1)}) + cost(P_{(i+\beta),\,(\gamma-\beta+1)},\, P_{(i+\gamma+1),\,(j-\gamma-1)})\} \\
&\geq T_{(i+\gamma+1),\,(j-\gamma-1)}. \qquad \square
\end{aligned}$$

Under the condition in Theorem 1, we can further prove that it is better to use several distribution schema to compute the sequence of Do-loops $L_{i-\alpha}$, $L_{i-\alpha+1}$, ..., L_{i+j-1} than to use only one distribution scheme $P_{(i-\alpha),(j+\alpha)}$, for $1 \le \alpha < i$ and $\gamma + 1 \le j \le s - i + 1$. Therefore, we **need not** compute $M_{(i-\alpha),(j+\alpha)}$. Based on this observation, we can show that $T_{(i-\alpha),(\gamma+1+\alpha)} > T_{(i+\beta),(\gamma-\beta+1)}$ and $T_{(i-\alpha),(j+\alpha)} > T_{(i+\gamma+1),(j-\gamma-1)}$, for $1 \le \alpha < i$ and $1 \le \beta < \gamma + 1 < j \le s - i + 1$. Therefore, we **need not** compute $T_{(i-\alpha),(j+\alpha)}$, for $1 \le \alpha < i$ and $\gamma + 1 \le j \le s - i + 1$.

Theorem 2 : *If* $M_{i,(\gamma+1)} > M_{i,\beta} + M_{(i+\beta),(\gamma-\beta+1)} + \mathit{THRESHOLD}$*, for some* β *where* $1 \le \beta < \gamma + 1$*, then the following four cases are true for* $1 \le \alpha < i$ *and* $\gamma + 1 < j \le s - i + 1$.

1. $M_{(i-\alpha),(\gamma+1+\alpha)} > M_{(i-\alpha),\alpha} + M_{i,\beta} + M_{(i+\beta),(\gamma-\beta+1)} + cost(P_{(i-\alpha),\alpha},\, P_{i,\beta}) + cost(P_{i,\beta},\, P_{(i+\beta),(\gamma-\beta+1)})$;
2. $M_{(i-\alpha),(j+\alpha)} > M_{(i-\alpha),\alpha} + M_{i,\beta} + M_{(i+\beta),(\gamma-\beta+1)} + M_{(i+\gamma+1),(j-\gamma-1)} + cost(P_{(i-\alpha),\alpha},\, P_{i,\beta}) + cost(P_{i,\beta},\, P_{(i+\beta),(\gamma-\beta+1)}) + cost(P_{(i+\beta),(\gamma-\beta+1)},\, P_{(i+\gamma+1),(j-\gamma-1)})$;
3. $T_{(i-\alpha),(\gamma+1+\alpha)} > T_{(i+\beta),(\gamma-\beta+1)}$;
4. $T_{(i-\alpha),(j+\alpha)} > T_{(i+\gamma+1),(j-\gamma-1)}$.

Proof: We will only prove the fourth case in this presentation; the other cases can be dealt with using a similar technique. $T_{(i-\alpha),(j+\alpha)}$ is the cost of computing the sequence of Do-loops $L_1, L_2, \ldots, L_{i-\alpha-1}, L_{i-\alpha}, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_{(i+\beta-1)}, L_{(i+\beta)}, \ldots, L_{(i+\gamma)}$, and $L_{(i+\gamma+1)}, \ldots, L_{i+j-1}$ with the restriction that it uses the distribution scheme $P_{(i-\alpha),(j+\alpha)}$ to compute Do-loops $L_{i-\alpha}, L_{i-\alpha+1}, \ldots, L_{i+j-1}$. Let $cost\_\delta_3$ be the cost of computing the sequence of Do-loops $L_{i-\alpha}, L_{i-\alpha+1}, \ldots, L_{i-1}$ using the distribution scheme $P_{(i-\alpha),(j+\alpha)}$. Then, $cost\_\delta_3$ is at least as large as $M_{(i-\alpha),\alpha}$. Let $cost\_\delta_4$ be the cost of computing the sequence of Do-loops $L_i, L_{i+1}, \ldots, L_{(i+\gamma)}$ using the distribution scheme $P_{(i-\alpha),(j+\alpha)}$. Then, $cost\_\delta_4$ is at least as large as $M_{i,(\gamma+1)}$. Thus, $cost\_\delta_4 > M_{i,\beta} + M_{(i+\beta),(\gamma-\beta+1)} + \mathit{THRESHOLD}$. Let $cost\_\delta_5$ be the cost of computing the sequence of Do-loops $L_{(i+\gamma+1)}, \ldots, L_{i+j-1}$ using the distribution scheme $P_{(i-\alpha),(j+\alpha)}$. Then, $cost\_\delta_5$ is at least as large as $M_{(i+\gamma+1),(j-\gamma-1)}$. Therefore,

$$\begin{aligned}
& T_{(i-\alpha),(j+\alpha)} \\
= \;& \mathrm{MIN}_{1 \le k < (i-\alpha)}\{T_{(i-\alpha)-k,k} + M_{(i-\alpha),(j+\alpha)} + cost(P_{(i-\alpha-k),k},\, P_{(i-\alpha),(j+\alpha)})\} \\
\ge \;& \mathrm{MIN}_{1 \le k < (i-\alpha)}\{T_{(i-\alpha)-k,k} + cost_\delta_3 + cost_\delta_4 + cost_\delta_5\} \\
> \;& \mathrm{MIN}_{1 \le k < (i-\alpha)}\{T_{(i-\alpha)-k,k} + M_{(i-\alpha),\alpha} + M_{i,\beta} + M_{(i+\beta),(\gamma-\beta+1)} + M_{(i+\gamma+1),(j-\gamma-1)} \\
& + \mathit{THRESHOLD}\}
\end{aligned}$$

$$
\begin{aligned}
&\geq \mathrm{MIN}_{1 \leq k < (i-\alpha)} \{ T_{(i-\alpha)-k,\,k} + M_{(i-\alpha),\,\alpha} + M_{i,\,\beta} + M_{(i+\beta),\,(\gamma-\beta+1)} + M_{(i+\gamma+1),\,(j-\gamma-1)} \\
&\quad + \mathit{cost}(P_{(i-\alpha-k),\,k},\, P_{(i-\alpha),\,\alpha}) + \mathit{cost}(P_{(i-\alpha),\,\alpha},\, P_{i,\,\beta}) + \mathit{cost}(P_{i,\,\beta},\, P_{(i+\beta),\,(\gamma-\beta+1)}) \\
&\quad + \mathit{cost}(P_{(i+\beta),\,(\gamma-\beta+1)},\, P_{(i+\gamma+1),\,(j-\gamma-1)}) \} \\
&\geq T_{(i+\gamma+1),\,(j-\gamma-1)}. \qquad \square
\end{aligned}
$$

From Theorem $2-(4)$, it is clear why *THRESHOLD* is chosen to be equal to four times the maximal communication cost between any two distribution schema. Note that Theorems $1-(1)$, $1-(2)$, and $2-(1)$ (and $1-(3)$, $2-(2)$, and $2-(3)$) will be true when *THRESHOLD* is only equal to two times (and three times, respectively) the maximal communication cost between any two distribution schema. Theorem 1 and Theorem 2 also suggest a sequence for computing $(M_{i,j})$-table and $(T_{i,j})$-table as shown in Fig. 6. For instance, if $M_{3,2} > (M_{3,1} + M_{4,1} +$ a threshold value), then from Theorem $1-(1)$, we need not compute $M_{3,3}$; from Theorem $1-(2)$, we need not compute $T_{3,2}$; from Theorem $1-(3)$, we need not compute $T_{3,3}$; from Theorem 2-(1), we need not compute $M_{2,3}$ and $M_{1,4}$; from Theorem 2-(2), we need not compute $M_{2,4}$ and $M_{1,5}$; from Theorem 2-(3), we need not compute $T_{2,3}$ and $T_{1,4}$; and from Theorem 2-(4), we need not compute $T_{2,4}$ and $T_{1,5}$.

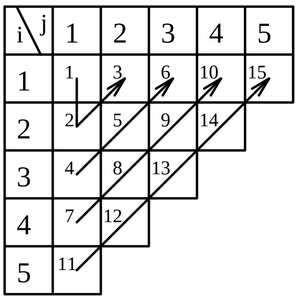

Figure 6: A sequence for computing $(M_{i,j})$-table and $(T_{i,j})$-table.

3.2 A Heuristic Method for Choosing a Threshold Value

If a data matrix or a data array is generated and used within more than one Do-loop, it may require a certain communication cost while executing these two Do-loops. For a 2-D data matrix, the communication cost of a transpose operation is C_T. For a 1-D data array, the communication cost of a data re-distribution operation is bounded by $m * t_c$, where m is the problem size. Therefore,

$$\begin{aligned} \mathit{THRESHOLD} &= \sum_{\forall \text{ matrix } A} \min\{4,\ \text{No. of Do-loops where matrix } A \text{ appears}\} * C_T + \\ &\quad \sum_{\forall \text{ array } B} \min\{4,\ \text{No. of Do-loops where array } B \text{ appears}\} * m * t_c, \end{aligned}$$

where each data matrix or each data array appears within at least two Do-loops. The constant 4 in the above formula is used because *THRESHOLD* is bounded by *four* times the maximal communication cost between any two distribution schema.

For example, in the sample program, matrix A appears within five Do-loops, each of arrays X and Y appears within three Do-loops, and each of other matrices and arrays only appears once in some do-loop. Therefore, *THRESHOLD* can be chosen as $4 * C_T + 2 * (3 * m * t_c)$.

4 Efficient Algorithms for Data Distribution

Based on Theorem 1 and Theorem 2, we can improve Algorithm 1 to deal with three cases as shown in Fig. 1.

4.1 The Case When a Program Segment Contains a Sequence of s Do-loops

This section discusses the first two cases shown in Fig. 1-(a) and Fig. 1-(b). Let γ_i be the minimum integer such that $M_{i,(\gamma_i+1)} > M_{i,\beta} + M_{(i+\beta),(\gamma_i-\beta+1)} + \mathit{THRESHOLD}$ for some β where $1 \leq \beta \leq \gamma_i \leq s-i+1$. Note that, for the boundary cases when $\gamma_i = s-i+1$ or $\beta = s-i+1$, we define dummy values $M_{i,s-i+2}$, $M_{s+1,1}$, and $M_{(i+\beta),(s-i-\beta+2)}$, so that the above assumption is satisfied. Let γ be the maximal value among γ_i, for $1 \leq i \leq s$. For example, $\gamma = \max_{1 \leq i \leq s}\{\gamma_i\}$.

Algorithm 2: A new dynamic programming algorithm for computing the cost of the data distribution schema for executing a sequence of s Do-loops on distributed memory computers is presented.

Input: $M_{i,j}$, $P_{i,j}$, and γ_i, where $1 \leq i \leq s$ and $1 \leq j \leq \gamma_i$; $T_{1,j}$ $(= M_{1,j})$, where $1 \leq j \leq \gamma_1$; and γ.

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

1. **for** $i := 2$ to s **do**
2.   **for** $j := 1$ to γ_i **do**

3. $T_{i,j} := \text{MIN}_{1 \le k < \min\{i,\gamma+1\}}\{T_{i-k,\,k} + M_{i,\,j} + cost(P_{i-k,\,k},\, P_{i,\,j}),\ \text{ if } k \le \gamma_{i-k}\}$;

4. **end_for end_for**

5. $Minimum\_Cost := \text{MIN}_{1 \le k \le \gamma}\{T_{s-k+1,\,k},\ \text{if } k \le \gamma_{s-k+1}\}$.

If a sequence of s Do-loops is enclosed by an iterative loop, then line 5 in Algorithm 2 can be modified below as Algorithm $2'$:

$5'$. $Minimum\_Cost' := \text{MIN}_{1 \le k \le \gamma}\{T_{s-k+1,\,k} + loop\_carried\_dependence(T_{s-k+1,\,k}), \text{if } k \le \gamma_{s-k+1}\}$.

We will now analyze Algorithm 2. The time complexity of this new dynamic programming algorithm is $O((\sum_{i=2}^{s}\gamma_i)\gamma + \gamma)$, which is bounded by $O(s\gamma^2)$. In addition, before applying Algorithm 2, we need to compute at most $\gamma_1 + \gamma_2 + \cdots + \gamma_s + s$ component alignment problems for the consecutive Do-loops L_i, L_{i+1}, ..., L_{i+j-1}, where $1 \le i \le s$ and $1 \le j \le \gamma_i + 1$. The total number of component alignment problems computed is, thus, no more than $s(\gamma + 1)$.

Table 5 shows a complete example by applying Algorithm 2 to the sample program mentioned in Section 2.3 for determining data distribution. In this example, we let *THRESHOLD* be $4 * C_T + 6 * m * t_c$, and we also assume that $4 * C_T + 6 * m * t_c$ is very small in comparison with C_7 $(= K * (m^2 + 2m * (\log N) + m) * t_c)$. Thus, $\gamma_1 = 3$; $\gamma_2 = 2$; $\gamma_3 = 3$; $\gamma_4 = 2$; $\gamma_5 = 1$; and $\gamma = 3$. We can see that the result computed from Algorithm 2 is the same as that computed from Algorithm 1. However, the computation for $M_{1,4}$; $M_{1,5}$; $M_{2,4}$; $T_{1,4}$; $T_{1,5}$; $T_{2,3}$; and $T_{2,4}$ is saved. In addition, the computation for $T_{5,1}$ and $Minimum\_Cost$ is simplified.

4.2 The Case When a Program Segment Contains s Do-loops with a General Structure

This section discusses the most general case as shown in Fig. 1-(c). In general, some Do-loops in a sequence of Do-loops may be iterative loops which contain other sequences of Do-loops with a general structure. In other words, some consecutive Do-loops may be enclosed by iterative loops which, again, with adjacent Do-loops, may be enclosed by other iterative loops, and so on. This enclosure relation can be naturally represented by trees (or a forest). A Do-loop may be a simple Do-loop or an iterative loop. Suppose that an iterative loop encloses at least two Do-loops. Then, simple Do-loops are leaf nodes in the trees, and iterative loops are internal nodes in the trees. If ν Do-loops are immediately

M_{ij}	j = 1	j = 2	j = 3	j = 4	j = 5
i = 1	β	ϵ	ι	save	save
i = 2	γ	θ	ν	save	
i = 3	β	ϵ	ζ		
i = 4	γ	δ			
i = 5	α				

T_{ij}	j = 1	j = 2	j = 3	j = 4	j = 5
i = 1	β	ϵ	ι	save	save
i = 2	ϵ	ι	save	save	
i = 3	η	κ	λ		
i = 4	κ	λ			
i = 5	μ				

$$
\begin{aligned}
T_{11} &= M_{11} = \beta \\
T_{12} &= M_{12} = \epsilon \\
T_{13} &= M_{13} = \iota \\
T_{21} &= T_{11} + M_{21} + (cost(P_{11}, P_{21}) = 0) = \epsilon \\
T_{22} &= T_{11} + M_{22} + (cost(P_{11}, P_{22}) = 0) = \iota \\
T_{31} &= \mathrm{MIN}\{T_{21} + M_{31} + (cost(P_{21}, P_{31}) = C_T), T_{12} + M_{31} + (cost(P_{12}, P_{31}) = C_T)\} = \eta \\
T_{32} &= \mathrm{MIN}\{T_{21} + M_{32} + (cost(P_{21}, P_{32}) = C_T), T_{12} + M_{32} + (cost(P_{12}, P_{32}) = C_T)\} = \kappa \\
T_{33} &= \mathrm{MIN}\{T_{21} + M_{33} + (cost(P_{21}, P_{33}) = C_T + C_1), T_{12} + M_{33} + (cost(P_{12}, P_{33}) = C_T + C_1)\} = \lambda \\
T_{41} &= \mathrm{MIN}\{T_{31} + M_{41} + (cost(P_{31}, P_{41}) = 0), T_{22} + M_{41} + (cost(P_{22}, P_{41}) = C_T + C_1), \\
&\qquad T_{13} + M_{41} + (cost(P_{13}, P_{41}) = C_T + C_1)\} = \kappa \\
T_{42} &= \mathrm{MIN}\{T_{31} + M_{42} + (cost(P_{31}, P_{42}) = C_1), T_{22} + M_{42} + (cost(P_{22}, P_{42}) = C_T + 2C_1), \\
&\qquad T_{13} + M_{42} + (cost(P_{13}, P_{42}) = C_T + 2C_1)\} = \lambda \\
T_{51} &= \mathrm{MIN}\{T_{41} + M_{51} + (cost(P_{41}, P_{51}) = C_T), T_{32} + M_{51} + (cost(P_{32}, P_{51}) = C_T)\} = \mu.
\end{aligned}
$$

$$
\begin{aligned}
Minimum\_Cost &= \mathrm{MIN}_{1 \le k \le 3}\{T_{5-k+1,\,k}\} \\
&= \mathrm{MIN}\{T_{51}, T_{42}, T_{33}\} \\
&= \lambda
\end{aligned}
$$

$$
\begin{aligned}
Minimum\_Cost' &= \mathrm{MIN}_{1 \le k \le 3}\{T_{5-k+1,\,k} + loop\_carried\_dependence(T_{5-k+1,\,k})\} \\
&= \mathrm{MIN}\{T_{51} + 0, T_{42} + C_T + C_1, T_{33} + C_T + C_1\} \\
&= \mu.
\end{aligned}
$$

Table 5: Apply Algorithm 2 and Algorithm 2$'$ to the sample program.

enclosed by an iterative loop, then this iterative loop is the parent node of these ν Do-loops, and these ν Do-loops are the ν corresponding child nodes of this iterative loop.

For instance, in the sample program mentioned in Section 2.3, Do-loops L_1, L_2, L_3, L_4, and L_5 are five child nodes of the outermost iterative loop. If we further elaborate the sample program, we can see that Do-loops L_2 and L_4 are two iterative loops, and that each of them contains three small Do-loops. L_2 contains $L_{2,1}$, which is from line 8 to line 12; $L_{2,2}$, which is from line 13 to line 16; and $L_{2,3}$, which is from line 17 to line 19. L_4 contains $L_{4,1}$, which is from line 26 to line 30; $L_{4,2}$, which is from line 31 to line 34; and $L_{4,3}$, which is from line 35 to line 37. Fig. 7 shows the family tree of the sample program. For convenience, we will say that L_1, L_2, L_3, L_4, and L_5 are the *first-level* Do-loops in the outermost iterative loop; in addition, they are *siblings* in this tree representation. Similarly, $L_{2,1}$, $L_{2,2}$, and $L_{2,3}$ are the first-level Do-loops in L_2, and they are siblings; $L_{4,1}$, $L_{4,2}$, and $L_{4,3}$ are the first-level Do-loops in L_4, and they are siblings.

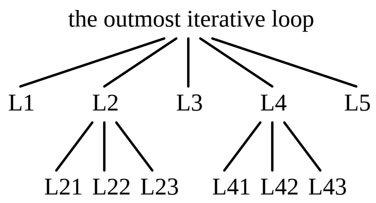

Figure 7: The family tree of the sample program.

Algorithm 3: An algorithm for computing the cost of the data distribution schema for executing a program segment which contains a sequence of s simple Do-loops with a general structure on distributed memory computers is constructed.

Input: A program segment which contains a sequence of s simple Do-loops with a general structure.

Output: The cost of executing this sequence of Do-loops on distributed memory computers.

1. Suppose that there are $\psi(s)$ first-level Do-loops in this input program segment;

2. scan these $\psi(s)$ first-level Do-loops one by one, **while** there exists an iterative loop, L_σ, which contains a sequence of ω simple Do-loops: $L_{\sigma 1}$, $L_{\sigma 2}$, ..., $L_{\sigma\omega}$, with a general structure, **do** recursively apply Algorithm 3 to the iterative loop L_σ;

3. construct $(M_{i,j})$-table, $(P_{i,j})$-table, γ_i, and γ for these $\psi(s)$ first-level Do-loops, where $1 \le i \le \psi(s)$, $1 \le j \le \gamma_i \le \psi(s) - i + 1$, and $\gamma = \max_{1 \le i \le \psi(s)}\{\gamma_i\}$;
4. apply Algorithm 2 to these $\psi(s)$ first-level Do-loops, and multiply the number of iterations by the resulting sequence of distribution schema to obtain their weight.

We will now briefly illustrate Algorithm 3. The first three steps in Algorithm 3 are quite straightforward; in the following, we will only explain the fourth step. We notice that, after applying Algorithm $2'$ to an iterative loop, if the resulting sequence of distribution schema contains more than one distribution scheme, then these distribution schema cannot be combined with any other distribution scheme in the sequel. For convenience, we use a dummy distribution scheme to represent the resulting distribution schema in the following. However, if there is only one distribution scheme obtained from Algorithm $2'$, then this distribution scheme may be combined with schema obtained from adjacent Do-loops.

We will now use the sample program again to go through Algorithm 3. First, because the sample program contains one iterative loop, Algorithm 3 recursively calls itself to handle this iterative loop. Then, because the outermost iterative loop contains five Do-loops L_1; L_2; L_3; L_4; and L_5, it (Algorithm 3) scans these five Do-loops one by one. Since L_2 and L_4 are iterative loops, it recursively applies itself to these two loops. When dealing with L_2, because $L_{2,1}$, $L_{2,2}$, and $L_{2,3}$ are simple Do-loops, it constructs $(M_{i,j})$-table, $(P_{i,j})$-table, γ_i, and γ for these three Do-loops, where $1 \le i \le 3$; $1 \le j \le \gamma_i \le 3 - i + 1$; $\gamma_1 = 3$; $\gamma_2 = 2$; $\gamma_3 = 1$; and $\gamma = 3$. After that, it can apply Algorithm $2'$ to L_2 and obtain a single data distribution scheme which illustrates that matrix A is distributed row by row as mentioned in Table 2. Similarly, when dealing with L_4, because $L_{4,1}$, $L_{4,2}$, and $L_{4,3}$ are simple Do-loops, it constructs $(M_{i,j})$-table, $(P_{i,j})$-table, γ_i, and γ for these three Do-loops, where $1 \le i \le 3$; $1 \le j \le \gamma_i \le 3 - i + 1$; $\gamma_1 = 3$; $\gamma_2 = 2$; $\gamma_3 = 1$; and $\gamma = 3$. After that, it can apply Algorithm $2'$ to L_4 and obtain a single data distribution scheme which illustrates that matrix A is distributed column by column as also mentioned in Table 2.

After handling L_2 and L_4 (both of which are iterative loops), Algorithm 3 constructs $(M_{i,j})$-table, $(P_{i,j})$-table, γ_i, and γ for the five Do-loops L_1; L_2; L_3; L_4; and L_5, where $1 \le i \le 5$; $1 \le j \le \gamma_i \le 5 - i + 1$; $\gamma_1 = 3$; $\gamma_2 = 2$; $\gamma_3 = 3$; $\gamma_4 = 2$; $\gamma_5 = 1$; and $\gamma = 3$. It then applies Algorithm $2'$ to these five first-level Do-loops and obtains a sequence of three data distribution schema as shown in Section

2.3. Then, it returns to Algorithm 3. Because there is only one outermost iterative loop in the sample program and whose data distribution schema have been obtained, Step 3 and Step 4 in Algorithm 3 are not applied in this case.

We will now analyze the time complexity of Algorithm 3. Suppose that a sequence of s simple Do-loops with a general structure is enclosed by $\psi(s)$ disjointed first-level iterative loops or simple Do-loops, and in addition, that the i-th iterative loop contains s_i simple Do-loops with a general structure. Then, $s = \sum_{i=1}^{\psi(s)} s_i$. We will first analyze how many component alignment problems are required to be computed in Algorithm 3. Based on the analysis of Algorithm 2, if an iterative loop contains s simple Do-loops, it requires computing $O(s\gamma)$ component alignment problems. Therefore, from Step 3 of Algorithm 3, we can formulate the recursive formula of the number of component alignment problems that must be computed in Algorithm 3 as follows:

$$\begin{cases} C(1) = 1; \\ C(s) = \sum_{i=1}^{\psi(s)} C(s_i) + O(\psi(s)\gamma), \ \text{where } s = \sum_{i=1}^{\psi(s)} s_i. \end{cases}$$

This recursion formula is similar to the one that counts the number of nodes in an arbitrary tree in which each internal node has at least two child nodes, and is bounded by the order of the number of its leaf nodes. Therefore, $C(s)$ is bounded by $O(s\gamma)$; in addition, the constant factor is less than 2.

We will now analyze the other computation time required for Algorithm 3. First, based on the analysis of Algorithm 2, if an iterative loop contains s simple Do-loops, Algorithm 2 can deal with this iterative loop within $O(s\gamma^2)$ time units. Therefore, from Step 4 of Algorithm 3, the recursion formula of the other computation time $T(s)$ can be formulated as follows:

$$\begin{cases} T(1) = 1; \\ T(s) = \sum_{i=1}^{\psi(s)} T(s_i) + O(\psi(s)\gamma^2), \ \text{where } s = \sum_{i=1}^{\psi(s)} s_i. \end{cases}$$

Similar to computing $C(s)$, $T(s)$ is bounded by $O(s\gamma^2)$.

5 Experimental Studies

In this section we will present experimental studies and show why it is important to determine whether data re-distribution is necessary. The target machine we used was a 32-node nCUBE-2 computer located at Academia Sinica. In this computer, each node has 4 Mega bytes of memory, runs at a modest clock rate of 20 MHz, and is rated at 7.5 MIPS (Million Instructions Per Second) and 3.5 MFLOPS (Million FLOating-point operations Per Second) in single precision arithmetic.

5.1 The Sample Program

Table 6 lists experimental results for implementing the sample application in Section 2.3 with various problem sizes. In this experimental study, we implemented two versions of parallel programs: (1) one based on a dynamic data distribution scheme; (2) one based on a static data distribution scheme. We let the constant OUT_ITERATION = 10 and the constant MAX_ITERATION = $20 * \log m$, where m is the problem size. Experimental results show that the proposed dynamic data distribution scheme outperformed a static data distribution scheme. Note that the computation time of these two parallel algorithms was not exactly the same because the second algorithm, which implements several message-passing data communication operations during the computation, requires more indexing operations than does the first algorithm, which is based on the original sequential computation.

matrix size	#PE = 2	#PE = 4	#PE = 8	#PE = 16	#PE = 32
$2^5 \times 2^5$	11.1 (0.1) 12.0 (0.8)	5.6 (0.1) 7.3 (1.6)	2.8 (0.1) 5.9 (2.5)	1.6 (0.3) 6.1 (3.8)	1.2 (0.5) 8.2 (5.9)
$2^6 \times 2^6$	51.7 (0.3) 53.3 (1.3)	25.9 (0.2) 28.9 (2.4)	13.1 (0.2) 17.8 (3.8)	6.7 (0.3) 13.6 (5.7)	3.7 (0.6) 14.0 (8.5)
$2^7 \times 2^7$	238.2 (1.2) 241.3 (2.0)	119.3 (0.8) 124.8 (4.0)	59.8 (0.5) 67.8 (6.3)	30.1 (0.5) 41.6 (9.2)	15.5 (0.7) 31.5 (13.2)
$2^8 \times 2^8$	1083 (4.9) 1091 (3.9)	542.1 (2.9) 553.2 (7.6)	271.3 (1.7) 287.0 (11.8)	136.0 (1.2) 156.5 (16.3)	68.5 (1.1) 96.4 (22.7)
$2^9 \times 2^9$	4869 (20.3) 4890 (7.8)	2435 (11.3) 2462 (15.9)	1218 (6.8) 1251 (23.4)	610.0 (3.9) 651.3 (32.1)	305.6 (2.6) 358.0 (42.4)
$2^{10} \times 2^{10}$	****	10841 (45.2) 10911 (32.4)	5427 (29.8) 5499 (50.6)	2714 (15.7) 2801 (67.0)	1358 (9.0) 1464 (85.2)
$2^{11} \times 2^{11}$	****	****	****	12129 (68.5) 12322 (149.0)	6072 (46.8) 6291 (190.4)

Table 6: The simulation time, "execution time (communication time)," for solving the sample program is expressed in units of seconds: (1) based on a dynamic data distribution scheme; (2) based on a static data distribution scheme. (The net computation time) = (execution time) − (communication time). "*****" means "not implemented" because of memory limitations.

5.2 Two-dimensional Fast Fourier Transform (2-D FFT)

In this experimental study, we implemented a 2-D FFT, immediately followed by an inverse 2-D FFT using the conventional row-column method. Therefore, the input data matrix was equal to the output data matrix. This program contains four loops:

L_1: loop 1 performs a 1-D FFT for each row;
L_2: loop 2 evaluates a 1-D FFT for each column;
L_3: loop 3 calculates an inverse 1-D FFT for each column; and
L_4: loop 4 computes an inverse 1-D FFT for each row.

Table 7 shows the approximate computation time and communication time of these four loops depending on whether the input data matrix A is distributed row by row or distributed column by column. A static data distribution scheme which distributes data either row by row or column by column will incur a $2C_h$ communication overhead due to the requirement of several "bit-reverse shuffle-exchange" and "butterfly-pattern" data communications, where $C_h = 2*(m^2/N)*(\log N)*t_c$. However, $C_h > C_T$, where C_T is the cost of performing a matrix transpose operation; thus, by applying our algorithms, we can show that data re-distribution is required between L_1 and L_2, and between L_3 and L_4. Table 8 lists the experimental results of implementing this 2-D FFT program based on both a dynamic data distribution scheme and a static data distribution scheme. The experimental results also show that using the above mentioned dynamic data distribution scheme is better than using a static data distribution scheme.

	matrix A is distributed row by row		matrix A is distributed column by column	
	computation time	communication time	computation time	communication time
L_1	C_g	0	C_g	C_h
L_2	C_g	C_h	C_g	0
L_3	C_g	C_h	C_g	0
L_4	C_g	0	C_g	C_h

Table 7: Computation time and communication time of four loops. $C_g = c*(m^2*(\log m)/N)*t_f$, where c is a constant; $C_h = 2*(m^2/N)*(\log N)*t_c$.

matrix size	#PE = 2	#PE = 4	#PE = 8	#PE = 16	#PE = 32
$2^5 \times 2^5$	0.163 (0.017)	0.085 (0.012)	0.049 (0.012)	0.035 (0.017)	0.039 (0.030)
	0.253 (0.088)	0.255 (0.165)	0.295 (0.243)	0.357 (0.322)	0.275 (0.251)
$2^6 \times 2^6$	0.745 (0.062)	0.380 (0.039)	0.198 (0.027)	0.111 (0.026)	0.077 (0.034)
	0.949 (0.205)	0.743 (0.351)	0.712 (0.499)	0.773 (0.651)	0.887 (0.812)
$2^7 \times 2^7$	3.378 (0.243)	1.711 (0.144)	0.871 (0.087)	0.452 (0.060)	0.250 (0.054)
	3.876 (0.523)	2.515 (0.785)	1.964 (1.055)	1.831 (1.339)	1.918 (1.636)
$2^8 \times 2^8$	15.140 (0.962)	7.651 (0.561)	3.871 (0.325)	1.967 (0.194)	1.016 (0.129)
	16.521 (1.499)	9.571 (1.892)	6.281 (2.326)	4.892 (2.819)	4.473 (3.354)
$2^9 \times 2^9$	67.139 (3.832)	33.878 (2.223)	17.101 (1.272)	8.643 (0.727)	4.385 (0.427)
	71.599 (4.932)	39.055 (5.128)	22.878 (5.567)	15.096 (6.193)	11.697 (7.051)
$2^{10} \times 2^{10}$	****	****	74.994 (5.045)	37.819 (2.843)	19.096 (1.608)
			90.648 (15.056)	53.242 (14.703)	35.265 (15.477)
$2^{11} \times 2^{11}$	****	****	****	****	82.884 (6.302)
					121.508 (36.663)

Table 8: The simulation time, "execution time (communication time)," for solving the 2-D FFT program is expressed in units of seconds: (1) based on a dynamic data distribution scheme; (2) based on a static data distribution scheme. "*****" means "not implemented" because of memory limitations.

5.3 Two-dimensional Heat Equation

In this experimental study, we implemented a 2-D heat equation using the alternating direction implicit (ADI) method, which is a way to reduce two-dimensional problems to a succession of many one-dimensional problems. Consider $u_t = b_1 u_{xx} + b_2 u_{yy}$ on a rectangle. We adopted the Peaceman-Rachford algorithm to formulate the partial differential equation as a second-order approximation of solving two sets of tridiagonal systems of linear equations, where variables of one set of tridiagonal systems corresponds to elements from each column of an intermediate matrix, and variables of the other set of tridiagonal systems corresponds to elements from each row of a target matrix [37]. According to the Thomas algorithm, we can reduce a tridiagonal system of linear equations to three sets of first-order recurrence equations, which can be solved by fast parallel algorithms based on either recursive-doubling techniques or cyclic-reduction techniques.

Thus, a static data distribution scheme by distributing data either row by row or column by column will incur a communication overhead due to the requirement of either "recursive-doubling-pattern" or "cyclic-reduction-pattern" data communication. By applying our algorithms, we have been able to show that data re-distribution is required between performing these two sets of tridiagonal systems. Table 9 lists the experimental results of implementing this 2-D heat equation based on both a dynamic data distribution scheme and a static data distribution scheme. The experimental results also show that using the above mentioned dynamic data distribution scheme is better than using a static data distribution scheme.

resolution	#PE = 2		#PE = 4		#PE = 8		#PE = 16		#PE = 32	
$2^6 \times 2^6$	16.086	(5.543)	9.393	(4.106)	6.068	(3.411)	4.640	(3.299)	4.360	(3.676)
	39.612	(6.193)	33.824	(12.515)	34.639	(18.773)	39.572	(25.016)	46.752	(31.548)
$2^7 \times 2^7$	55.828	(17.714)	30.909	(11.842)	18.194	(8.637)	11.839	(7.047)	9.093	(6.683)
	139.893	(16.784)	97.070	(24.178)	83.613	(37.905)	86.504	(48.964)	97.234	(63.195)
$2^8 \times 2^8$	206.873	(62.178)	111.079	(38.718)	61.709	(25.511)	36.321	(18.200)	23.677	(14.603)
	520.590	(52.849)	315.673	(47.546)	227.535	(74.700)	203.104	(98.028)	209.332	(126.395)
$2^9 \times 2^9$	793.865	(231.285)	418.917	(137.613)	225.146	(84.465)	124.967	(54.607)	73.619	(38.424)
	2014.049	(186.911)	1113.913	(96.638)	699.733	(152.807)	526.407	(196.044)	478.899	(250.583)
$2^{10} \times 2^{10}$	****		****		864.962	(310.443)	461.217	(183.924)	255.943	(117.280)
					2360.988	(296.590)	1539.387	(388.614)	1197.972	(505.510)
$2^{11} \times 2^{11}$	****		****		****		****		956.192	(405.710)
									3372.475	(1009.577)

Table 9: The simulation time, "execution time (communication time)," of the first 100 iterations for solving the 2-D heat equation is expressed in units of seconds: (1) based on a dynamic data distribution scheme; (2) based on a static data distribution scheme. "****" means "not implemented" because of memory limitations.

6 Conclusions

A scientific application program is naturally written using a sequence of Do-loops and subroutines, and each subroutine can be treated as another sequence of Do-loops. Because it is not practical to implement an exponential time algorithm to deal with optimal data distributions at compiling time, it is helpful to implement a heuristic algorithm with polynomial time complexity. In this paper, we have proposed efficient dynamic programming algorithms which allow variable length program segments to use different data distribution schema in order to improve the computation load balance and to avoid communication overhead. The searching space of the algorithms was reduced dramatically after we proved that data re-distribution was necessary for executing a sequence of Do-loops if the communication cost due to performing this sequence of Do-loops was larger than a threshold value, and if this threshold value was equal to four times the maximal communication cost between any two distribution schema.

Suppose that we must use at least two distribution schema to compute any $(\gamma + 1)$ consecutive Do-loops. Normally γ is a small integer, for example, $\gamma = 5$. Then, we can find a sequence of distribution schema for executing a sequence of s Do-loops having a general structure (as shown in Fig. 1-(c)) in $O(s\gamma^2)$ time units. In addition, while applying these algorithms, we only need to compute at most $s(\gamma+1)$ component alignment problems, each having a reasonable problem size. In practice, our method can be used in parallelizing compilers to automatically determine data distribution for distributed memory systems.

Acknowledgments

This work was partially supported by the NSC under Grants NSC 84-2213-E-001-003, NSC-85-2213-E-001-006 and NSC 86-2213-E-001-006. Part of this paper was presented at the IEEE International Conf. on Parallel and Distributed Systems, HsinChu, Taiwan, Dec. 1994 [26]. The author would like to thank Prof. M. C. Chen and the anonymous referees for their valuable comments on this paper, Dr. S.-K. Ueng, Dr. C.-M. Wang and Dr. J.-J. Wu for early reading 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. 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors, Volumn I: General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[10] 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.

[11] 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.

[12] S. K. S. Gupta, S. D. Kaushik, C. H. Huang, and P. Sadayappan. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32:155–172, 1996.

[13] 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.

[14] 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.

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

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

[17] E. T. Kalns and L. M. Ni. Processor mapping techniques toward efficient data redistribution. Technical Report MSU-CPS-ACS-86, Department of Computer Science, Michigan State University, Jan. 1994.

[18] 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.

[19] K. Kennedy, N. Nedeljković, and A. Sethi. Communication generation for cyclic(k) distributions. In *Proc. Third Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, pages 185–197, Troy, New York, May 1995.

[20] K. Knobe, J. D. Lukas, and G. L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, Feb. 1990.

[21] K. Knobe and V. Natarajan. Automatic data allocation to minimize communication on SIMD machines. *The Journal of Supercomputing*, 7:387–415, 1993.

[22] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

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

[24] 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.

[25] 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.

[26] P.-Z. Lee. Efficient algorithms for data distribution on distributed memory multicomputers. In *Proc. International Conf. on Parallel and Distributed Systems*, pages 573–579, Hsin-Chu, Taiwan, Dec. 1994, also Technical Report TR-94-005, Institute of Information Science, Academia Sinica.

[27] P.-Z. Lee. Techniques for compiling programs on distributed memory multicomputers. *Parallel Computing*, 21(12):1895–1923, 1995.

[28] P.-Z. Lee and W. Y. Chen. Compiler techniques for determining data distribution and generating communication sets on distributed-memory multicomputers. In *Proc. 29th Hawaii International Conference on System Sciences*, volume 1, pages 537–546, Maui, Hawaii, Jan. 1996, also Technical Report TR-95-016, Institute of Information Science, Academia Sinica.

[29] P.-Z. Lee and W. Y. Chen. Generating global name-space communication sets for doall statements. *Submitted for publication*, on available via WWW http://www.iis.sinica.edu.tw/~leepe/PAPER/comm97.ps.

[30] 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.

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

[32] 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.

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

[34] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Computing*, pages 364–384. Pitman/MIT-Press, 1991.

[35] 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.

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

[37] J. C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*, chapter 7.3 The Alternating Direction Implicit (ADI) Method, pages 142–153. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, CA, 1989.

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

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

[40] 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.

[41] 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.

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

```
DO 42 n = 1, OUT_ITERATION
  DO 6 i = 1, m
    X(i) = 0.0
    DO 6 j = 1, m
      X(i) = X(i) + A(i,j)
  CONTINUE
  DO 19 k = 1, MAX_ITERATION
    DO 12 i = 1, m
      C(i) = 0.0
      DO 12 j = 1, m
        C(i) = C(i) + A(i,j) * B(i)
    CONTINUE
    DO 16 i = 1, m
      DO 16 j = 1, m
        A(i,j) = (A(i,j) + C(i) - X(i)) / (m * m)
    CONTINUE
    DO 19 i = 1, m
      X(i) = X(i) + (B(i) - C(i)) / A(i,i)
  CONTINUE
  DO 24 i = 1, m
    Y(i) = 0.0
    DO 24 j = 1, m
      Y(i) = Y(i) + A(j,i)
  CONTINUE
  DO 37 k = 1, MAX_ITERATION
    DO 30 i = 1, m
      V(i) = 0.0
      DO 30 j = 1, m
        V(i) = V(i) + A(j,i) * U(i)
    CONTINUE
    DO 34 i = 1, m
      DO 34 j = 1, m
        A(j,i) = (A(j,i) + V(i) - Y(i)) / (m * m)
    CONTINUE
    DO 37 i = 1, m
      Y(i) = Y(i) + (U(i) - V(i)) / A(i,i)
  CONTINUE
  DO 41 i = 1, m
    XD(n,i) = X(i) + A(i,1) * Y(i)
    YD(n,i) = Y(i) + A(i,1) * X(i)
  CONTINUE
CONTINUE
```

Figure 4: The sample program.

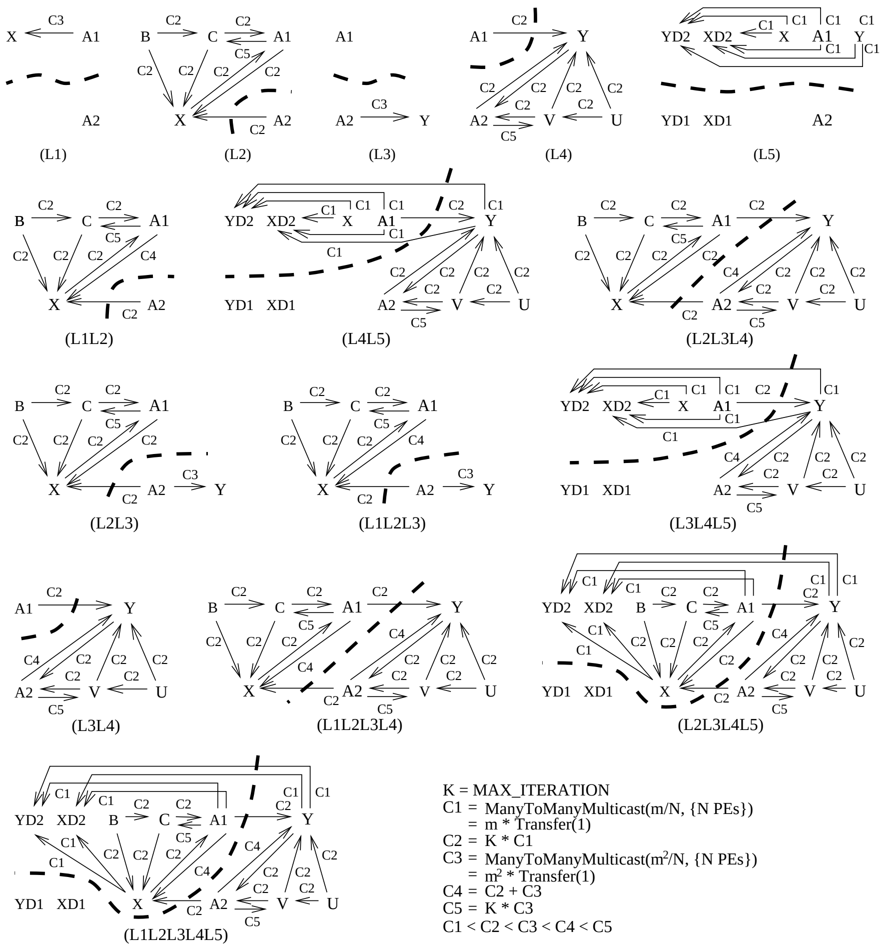

Figure 5: The component affinity graphs and the corresponding component alignment for various consecutive loops.