Compiler Techniques for Determining Data Distribution and Generating Communication Sets on Distributed-Memory Machines

PeiZong Lee and Wen-Yao Chen
Institute of Information Science, Academia Sinica
Taipei, Taiwan, R.O.C.

Abstract

This paper is concerned with designing efficient algorithms for determining data distribution and generating communication sets on distributed memory multicomputers. First, we propose a dynamic programming algorithm to automatically determine data distribution at compiling time. The proposed algorithm also can determine whether data redistribution is necessary between two consecutive DO-loop program fragments. Second, we propose closed forms to represent communication sets among processing elements for executing doall statements, when data arrays are distributed in a restricted block-cyclic fashion. Our methods can be included in current compilers and used when programmers fail to provide any data distribution directives. Experimental studies on a nCUBE-2 multicomputer are also presented.

1 Introduction

Arrays distribution and communication sets generation are two problems we must solve when dealing with the compilation of DO-loop program fragments for distributed memory multicomputers. For instance, in High Performance Fortran (HPF), programmers have obligations to provide *TEMPLATE*, *ALIGN*, and *DISTRIBUTE* directives to specify data distribution [8]. Then, based on these directives, compilers can generate all communication instructions. In this paper, however, we try to determine data distribution automatically by compilers in contrast with previous research works, which previously only allowed programmers explicitly to specify the data distribution using language extensions. We show systematic methods for determining data distributions and for generating communication sets for each processing element (PE). Thus, the proposed algorithms can be included in compilers for automatically transforming sequential DO-loop program fragments into parallel version programs with message-passing communication primitives. For instance, our methods can be included in HPF compilers and used when programmers fail to provide any data distribution directives.

In the following, we state the problems we will address in this paper. First, given a sequence of DO-loop programs, we are interested in how to align data arrays, so that data communication incurred due to the resulting data distribution will be minimized. Conventionally, this problem can be solved by using a component alignment algorithm to determine a static data distribution scheme for the whole program [3] [12]. In contrast to giving a static solution, we will present a dynamic programming algorithm to determine whether data redistribution is necessary between two consecutive DO-loop program fragments.

Second, after determining data alignments among data arrays, we are interested in how to distribute data arrays among PEs. In order to do this, compilers must include an analytical model, which can formulate communication time and computation time. In addition, this analytical model can help to determine grain and granularity of execution space, it also can help to determine whether data arrays are distributed among PEs by a *block* fashion, or a *cyclic* fashion, or a *block-cyclic* fashion.

Third, after determining data distribution, we focus our attentions on generating communication sets among PEs. Previous research works have provided closed forms of generating communication sets for the special cases when an array's distribution is either in a block fashion or in a cyclic fashion [7] [9]. Recently, a lot of research works are concentrated on the more general cases when an array's distribution is in a block-cyclic fashion [2] [4] [5] [13]. However, for the general cases, none of them provided formulas which could be represented by a constant number of closed forms. We are interested in integrating previous research works and in formulating a complete set of closed forms of generating communication sets for each PE.

*This work was partially supported by the NSC under Grant NSC 85-2213-E-001-006.

2 Background

The following closed forms (regular sections) will be used in this paper.

- $[a : e_1]$ represents the set of consecutive integers from a to e_1.
- $[a : e_1 : s_1]$ is in behalf of the set of integers from a with a stride (period) s_1 until to a maximum integer which is not greater than e_1.
- $[[a : e_1] : e_2 : s_2]$ is the set $\{[a : e_1], [a : e_1] + s_2, [a : e_1] + 2s_2, \cdots,$ until not greater than $e_2\}$.
- $[[a : e_1 : s_1] : e_2 : s_2]$ specifies the set $\{[a : e_1 : s_1], [a : e_1 : s_1] + s_2, [a : e_1 : s_1] + 2s_2, \cdots,$ until not greater than $e_2\}$.
- $[[[a : e_1 : s_1] : e_2 : s_2] : e_3 : s_3]$ illustrates the set $\{[[a : e_1 : s_1] : e_2 : s_2], [[a : e_1 : s_1] : e_2 : s_2] + s_3, [[a : e_1 : s_1] : e_2 : s_2] + 2s_3, \cdots,$ until not greater than $e_3\}$.

Suppose that array $A([a_1 : a_2])$ is indexed from a_1 to a_2 and there are in total N PEs numbered from 0 to $N-1$. Then, if we adopt *cyclic*(b) distribution, the set $A([[a_1 + p * b : a_1 + p * b + b - 1] : a_2 : N * b])$ is stored in PE p (PE_p). We will say that array A is distributed by a *cyclic* fashion if $b = 1$; by a *block* fashion if $b = \lceil (a_2 - a_1 + 1)/N \rceil$; and by a *block−cyclic* fashion if $1 < b < \lceil (a_2 - a_1 + 1)/N \rceil$.

2.1 Compiling Sequential Programs on Distributed Memory Machines

The abstract target machine we adopt is a q-D grid of $N_1 \times N_2 \times \cdots \times N_q$ PEs, where D stands for dimensional. A PE 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 $1 \leq i \leq q$. The parallel program generated from a sequential program for a grid corresponds to the SPMD (Single Program Multiple Data) model, in which each PE executes the same program but operates on distinct data items [6]. In this paper, we adopt a global name space for representing data arrays among PEs. Therefore, our machine model can be regarded as a distributed shared memory model [7] [9].

When dealing with the compilation of a sequential program on a distributed memory computer, we must decide on a suitable data distribution for each data array, so that a computation load balance can be achieved; in addition, overhead due to communication can be minimized. We also must provide efficient algorithms for generating communication sets, so that performance gained due to parallel computing will not be degraded by software overhead. Previously, researchers have shown that after applying loop transformation techniques such as loop interchange; loop reversal; and loop skewing, a sequential Do-loop program fragment can be transformed into an equivalent program fragment either with *doall* loops in all levels, or with an outmost *doserial* loop in which all its inner loops are doall loops [15]. Doall loops guarantee that statements in different iterations (loop bodies) can be executed independently even in different PEs. Therefore, we can group different sets of iterations into PEs, and execute each set of iterations in different PEs independently.

Figure 1 shows two programs: (a) a sequential program for solving a linear system AX = B, and (b) its corresponding doall loop program. The corresponding SPMD program in which data arrays are distributed by *cyclic*(b) can be found in [10]. Readers can find that there is a one-to-one correspondence between statements in the original sequential program (which have been rewritten after performing loop transformations) and its corresponding doall-version program. For this reason, without any confusion, in the sequel we will frequently apply compiler techniques directly on the sequential programs. As to compile a doall loop version program to a SPMD program, it is straightforward if data distributions for all arrays (or matrices) are determined.

{* Solving a linear system AX = B based on the LU decomposition. *}

```
REAL  A(m, m), B(m), X(m), Y(m)

do k = 0, m - 1                {*  A = LU.  *}
  do i = k + 1, m - 1
    A(i, k) = A(i, k) / A(k, k)
    do j = k + 1, m - 1
      A(i, j) = A(i, j) - A(i, k) * A(k, j)
enddo  enddo  enddo

do i = 0, m - 1                {*  LY = B.  *}
  Y(i) = B(i)
  do j = i + 1, m - 1
    B(j) = B(j) - A(j, i) * Y(i)
enddo  enddo

do i = m - 1, 0, -1            {*  UX = Y.  *}
  X(i) = Y(i) / A(i, i)
  do j = 0, i - 1
    Y(j) = Y(j) - A(j, i) * X(i)
enddo  enddo
```

(a)

```
REAL  A(m, m), B(m), X(m), Y(m)

doserial k = 0, m - 1
  doall i = k + 1, m - 1
    A(i, k) = A(i, k) / A(k, k)
    doall j = k + 1, m - 1
      A(i, j) = A(i, j) - A(i, k) * A(k, j)
enddo  enddo  enddo

doserial i = 0, m - 1
  Y(i) = B(i)
  doall j = i + 1, m - 1
    B(j) = B(j) - A(j, i) * Y(i)
enddo  enddo

doserial i = m - 1, 0, -1
  X(i) = Y(i) / A(i, i)
  doall j = 0, i - 1
    Y(j) = Y(j) - A(j, i) * X(i)
enddo  enddo
```

(b)

Figure 1: Solving a linear system AX = B based on the LU decomposition: (a) the original sequential program, (b) the corresponding doall loop version.

3 Determining Data Distribution

We now show how to use a component alignment algorithm to determine data distribution. First, we analyze the relationship between left-hand-side and right-hand-side array subscript reference patterns in the

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	$f_1(i)$	$f_2(i)$	*need additional analysis	*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)	$O(m * num(seq))$

Table 1: Communication primitives used in the SPMD program when left-hand-side and right-hand-side array subscripts have some specific patterns. i and j are loop indexing variables; c, c_1, and c_2 are constants at compile time; "unknown" means that the value is unknown at compile 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.

original sequential program. Based on pattern matching techniques [11], 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. Readers can find that Case 2 is a special case of Case 3. In Section 4, we will show how to use closed forms to represent communication sets of Case 3. Thus, we can generate communication sets of Case 3 efficiently. 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.

3.1 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 the Case 3 of Table 1.

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

case	edge head	edge tail	approximate communication cost
$C1$	1-D	1-D	N_1 * OneToManyMulticast($\frac{m}{N_1}$, $\{N_2$ PEs$\}$) or m * Transfer(1)
$C2$	2-D	1-D	N_1 * OneToManyMulticast($\frac{m}{N_1}$, $\{N$ PEs$\}$)
$C3$	1-D	2-D	ManyToManyMulticast($\frac{m^2}{N}$, $\{N$ PEs$\}$) or N * OneToManyMulticast($\frac{m^2}{N}$, $\{N_2$ PEs$\}$)
$C4$	2-D	2-D	ManyToManyMulticast($\frac{m^2}{N}$, $\{N$ PEs$\}$) or m^2 * Transfer(1)

Table 2: Approximate communication costs.

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

We now return to our example of the linear system. Figure 2 shows the component affinity graph and the suggested component alignment of the sample program mentioned in Figure 1. Suppose that our target machine is a linear processor array with N PEs. For the purpose of parallelism, based on the suggested component alignment, matrix A will be distributed by $cyclic(b)$ along its rows; arrays B, X, and Y will also be distributed by $cyclic(b)$. The data distribution functions of A, B, X, and Y are listed in below.

$$f_A(i,j) = (\lfloor \tfrac{i}{b} \rfloor \bmod N);$$
$$f_B(i) = f_X(i) = f_Y(i) = (\lfloor \tfrac{i}{b} \rfloor \bmod N).$$

Note that, 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. In the next subsection, we will show how to decide the block size b.

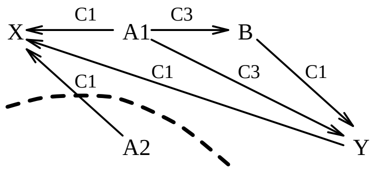

Figure 2: Component affinity graph and the suggested component alignment of the sample program which solves a linear system based on the LU decomposition.

3.2 Determining the Granularity of Data Distribution

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

We can formulate the total execution time from the SPMD program which includes both the computation time and the communication time. For each arithmetical operation or logical operation, we assume that the computation time is t_f; for each *saxpy* operation which executes a multiplication then follows an addition, and we assume that the computation time is t_x; for each message passing operation, we assume that the communication cost is $t_s + k * t_c$, where t_s is the start-up time for sending a message; t_c is the communication time of transferring a word; and k is the message size in words. Table 3 shows the parameters t_f, t_x, t_s, and t_c on a 32-node nCUBE-2 computer.

parameter	t_f	t_x	t_s	t_c
mean (in μsec)	4.56	7.15	171.34	2.31
variance (in μsec)	0.18	0.14	0.94	0.05

Table 3: Parameters used in describing the execution time on the nCUBE-2 computer.

We now continue our sample example of the linear system. Let the time of executing the LU decomposition be T_{LU}; the time of executing two triangular linear systems be T_{2TLS}; and the total execution time be T. Then, from the corresponding SPMD program in [10], we can formulate T, T_{LU}, and T_{2TLS} as follows.

$T = T_{LU} + T_{2TLS}$,

$T_{LU} = \sum_{i_1=1}^{m/(N*b)} \sum_{i_2=1}^{N} \{ 9 * t_f + \sum_{i_3=1}^{b} \sum_{i_4=i_3+1}^{b} (t_f + (m - ((i_1 - 1) * N * b + (i_2 - 1) * b + i_3) + 1) * t_x) + \sum_{i_5=i_1}^{m/(N*b)} \sum_{i_6=1}^{b} \sum_{i_7=1}^{b} (t_f + (m - ((i_1 - 1) * N * b + (i_2 - 1) * b + i_6) + 1) * t_x) + (\log N + 1) * b * (t_s + (m - ((i_1 - 1) * N * b + (i_2 - 1) * b)) * t_c) \}$,

$T_{2TLS} = 2 * \sum_{i_1=1}^{m/(N*b)} \sum_{i_2=1}^{N} \{ 9 * t_f + \sum_{i_3=1}^{b} (t_f + (b - i_3) * t_x) + \sum_{i_4=i_1}^{m/(N*b)} \sum_{i_5=1}^{b} \sum_{i_6=1}^{b} t_x + (\log N + 1) * (t_s + b * t_c) \}$.

The symbolic manipulations of the above formulas can be solved using a computer algebra system like "Derive" [14]. The total execution time 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 $\frac{\partial T}{\partial b} = 0$, or by substituting all possible b into the formula. Table 4 shows T_{LU}, T_{2TLS}, and T for various block size b ranging from 1 to 64, and for various numbers of PEs N ranging from 2 to 32, when the problem size m is 1024. We also list the real execution time on a 32-node nCUBE-2 computer for a comparison.

It is interesting to point out that both the optimal execution time of the LU decomposition and the whole program is achieved when the block size is 1; however, the optimal execution time of two triangular linear systems is achieved when the block size is 8 or 16. We will discuss other details of choosing a block size b again in Section 4.

3.3 Determining Whether Data Redistribution is Necessary

It is feasible to assume that the optimal data distributions for each single Do-loop may be different among one another in a sequence of Do-loops which perform computation-intensive scientific applications. In effect, we found that under load balance constraints, the communication overhead due to mismatch arrays' component alignments is much higher than the communication overhead due to select a different block size b. In the following, we introduce a simple dynamic programming algorithm to determine whether the data redistribution is necessary.

Suppose that a program contains s Do-loops: L_1, L_2, ..., L_s in sequence. 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 $P_{i,j}$ be the distribution scheme, for $1 \le i \le s$ and $1 \le j \le s-i+1$. Define $T_{i,j}$ to be the cost of comput-

block size		#PE = 2		#PE = 4		#PE = 8		#PE = 16		#PE = 32	
1	T_{LU}	1315.1	(1286.4)	661.4	(646.9)	334.9	(327.9)	172.1	(169.1)	91.1	(90.3)
	T_{2TLS}	2.84	(4.56)	2.09	(3.04)	1.87	(2.46)	1.91	(2.35)	2.09	(2.47)
	T	1318.0	(1290.9)	663.5	(650.0)	336.8	(330.3)	174.0	(171.4)	93.2	(92.8)
2	T_{LU}	1316.8	(1290.1)	663.5	(650.7)	337.1	(331.6)	174.2	(172.8)	93.0	(94.0)
	T_{2TLS}	2.59	(4.18)	1.64	(2.49)	1.25	(1.73)	1.12	(1.44)	1.14	(1.39)
	T	1319.4	(1294.3)	665.1	(653.1)	338.3	(333.3)	175.3	(174.2)	94.2	(95.4)
4	T_{LU}	1320.5	(1297.6)	668.8	(658.1)	343.2	(339.1)	180.6	(180.2)	99.5	(101.3)
	T_{2TLS}	2.47	(4.02)	1.43	(2.23)	0.94	(1.39)	0.74	(1.01)	0.68	(0.87)
	T	1323.0	(1301.6)	670.3	(660.4)	344.2	(340.5)	181.4	(181.2)	100.1	(102.2)
8	T_{LU}	1328.2	(1312.6)	680.2	(673.1)	356.3	(353.9)	194.4	(194.8)	113.2	(115.5)
	T_{2TLS}	2.43	(3.98)	1.34	(2.15)	0.82	(1.26)	0.58	(0.84)	0.49	(0.66)
	T	1330.6	(1316.6)	681.5	(675.2)	357.1	(355.2)	195.0	(195.6)	113.6	(116.1)
16	T_{LU}	1343.7	(1342.6)	703.2	(702.8)	382.8	(383.2)	221.8	(223.1)	139.6	(141.9)
	T_{2TLS}	2.43	(4.04)	1.34	(2.20)	0.81	(1.29)	0.55	(0.84)	0.44	(0.64)
	T	1346.1	(1346.6)	704.6	(705.0)	383.6	(384.5)	222.4	(224.0)	140.1	(142.6)
32	T_{LU}	1375.2	(1402.2)	749.0	(761.6)	435.5	(440.1)	274.7	(276.3)	186.9	(187.6)
	T_{2TLS}	2.49	(4.25)	1.43	(2.40)	0.90	(1.47)	0.64	(1.02)	0.52	(0.80)
	T	1377.7	(1406.5)	751.3	(764.0)	436.4	(441.6)	275.4	(277.3)	187.4	(188.4)
64	T_{LU}	1440.2	(1520.7)	844.6	(876.3)	539.4	(547.3)	371.7	(368.6)	not implement	
	T_{2TLS}	2.63	(4.71)	1.63	(2.85)	1.14	(1.92)	0.89	(1.46)		
	T	1442.8	(1525.4)	846.3	(879.1)	540.5	(549.3)	372.6	(370.1)		

Table 4: The simulation time in units of seconds for solving a linear system $A_{1024\times 1024}X_{1024} = B_{1024}$ based on the LU decomposition and two triangular linear systems. The data that are not in parentheses are obtained by running a 32-node nCUBE-2 computer; the data in parentheses are based on an analytical model.

ing the sequence of Do-loops $L_1, L_2, \ldots, L_{i+j-1}$ with the restriction that it uses the distribution scheme $P_{i,j}$ to compute Do-loops $L_i, L_{i+1}, \ldots, L_{i+j-1}$. 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 $P_{i,j}$.

Heuristic algorithm for determining whether data redistribution is necessary:

An algorithm for computing the cost of data distribution schema of executing a sequence of s Do-loops on distributed memory computers is presented.

Input: $M_{i,j}$, $P_{i,j}$, and $T_{1,i}$ ($= M_{1,i}$).

Output: The cost of executing s Do-loops.

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

The data distribution scheme obtained from the above algorithm is at least as good as any static data distribution scheme, whose cost is equal to $T_{1,s}$. We now briefly describe how to improve this algorithm. It is clear that $M_{i,(\gamma_i+1)} \ge M_{i,\gamma_i}$, for $1 \le \gamma_i < s-i+1$. We can show that if $M_{i,(\gamma_i+1)}$ is larger than $M_{i,\beta}$ plus $M_{(i+\beta),(\gamma_i-\beta+1)}$ and plus a threshold value which is equal to three times of the maximal communication cost between any two distribution schema, for some β where $1 \le \beta < \gamma_i + 1$, then it is better to use three distribution schema $P_{i,\beta}$, $P_{(i+\beta),(\gamma_i-\beta+1)}$, and $P_{(i+\gamma_i+1),(j-\gamma_i-1)}$ to compute the sequence of Do-loops $L_i, L_{i+1}, \ldots, L_{i+j-1}$, than to use only one distribution scheme $P_{i,j}$, for $\gamma_i+1 < j \le s-i+1$. Therefore, we **need not** compute $M_{i,j}$. Based on this observation, we can show that $T_{i,(\gamma_i+1)} > T_{(i+\beta),(\gamma_i-\beta+1)}$ and $T_{i,j} > T_{(i+\gamma_i+1),(j-\gamma_i-1)}$. Therefore, we **need not** compute $T_{i,j}$, for $\gamma_i + 1 \le j \le s-i+1$.

Let γ_i be the minimum integer such that $M_{i,(\gamma_i+1)} > M_{i,\beta} + M_{(i+\beta),(\gamma_i-\beta+1)} +$ (a threshold value), for some β where $1 \le \beta \le \gamma_i \le s-i+1$. 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 $1 \le i \le s$. For example, $\gamma = \max_{1\le i\le s}\{\gamma_i\}$. Then the above algorithm can be improved as follows.

1'. **for** $i := 2$ to s **do**
2'. **for** $j := 1$ to γ_i **do**
3'. $T_{i,j} := \mathrm{MIN}_{1\le k<\min\{i,\gamma+1\}}\{T_{i-k,k} + M_{i,j} + cost(P_{i-k,k}, P_{i,j}),\ \text{if } k \le \gamma_{i-k}\}$;
4'. **end_for end_for**
5'. $Minimum\_Cost := \mathrm{MIN}_{1\le k\le\gamma}\{T_{s-k+1,k},\ \text{if } k \le \gamma_{s-k+1}\}$.

The time complexity of this improved 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 this algorithm, we need to compute at most $\gamma_1 + \gamma_2 + \cdots + \gamma_s + s$ reasonable-size component alignment problems for the consecutive Do-loops $L_i, L_{i+1}, \ldots, L_{i+j-1}$, 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 $s(\gamma+1)$.

4 Generating Communication Sets for Doall Statements

Suppose that in a distributed-shared-memory machine, processors are numbered from 0 to $N-1$. Arrays $A([a_1 : a_2])$ and $C([c_1 : c_2])$ are distributed in $cyclic(b_1)$ and $cyclic(b_2)$, respectively. We now show how to compute necessary communication sets in each processor due to execute the following doall statement, where $s_1 > 0$, $s_2 > 0$, and g is a function.

doall $i = 0,\ \lfloor \frac{u_1 - l_1}{s_1} \rfloor$
$A(l_1 + i * s_1) = g(C(l_2 + i * s_2))$.

4.1 The Structure of Generated Code

We now analyze the problem. We will say that $f_k(i) = l_k + i * s_k$ and the inverse functions $f_k^{-1}(l_k + i * s_k) = i$, for $k = 1$ or 2. Figure 3 shows a detailed outline of implementing a doall statement in each PE which is a generalization based on formulas in [7].

Step 1 of Figure 3 generates an iteration set which specifies iterations to be performed on PE_p, and two processor sets which represent PEs that PE_p will send data to or receive data from. Step 2 calculates communication sets and sends them to other PEs. Step 3 performs computations for iterations which access only local data. Step 4 receives data message from other PEs and executes computations for iterations which access local data and some message buffers. Note that, $exec(p)$ in substep 1.1 is only formulated for deriving other communication sets and processor sets. Since $exec(p) = \bigcup_{q \in recv\_pe(q)} iter(p,q)$ and $iter(p,q) = f_2^{-1}(recv_C(p,q))$, we can combine substep 1.1 and three substeps in Step 4 into a receive-execute loop. Therefore, in practice, iteration sets $exec(p)$ and $iter(p,q)$ need not to be calculated. It is also interesting to point out that in order to gain efficiency by allowing overlapped execution, we have arranged communication and computation tasks interleavedly.

Code on processing element p (PE_p):

1. Generate iteration sets and processor sets:
1.1 $exec(p) = f_1^{-1}(local_A(p) \cap [l_1 : u_1 : s_1])$, which specifies iterations to be performed on PE_p;
1.2 $send\_pe(p) = \{q \mid q \neq p$ and PE_p will send some data to $PE_q\}$;
1.3 $recv\_pe(p) = \{q \mid q \neq p$ and PE_p will receive some data from $PE_q\}$;

2. forall $q \in send\_pe(p)$, **do**
2.1 $send_C(p,q) = local_C(p) \cap f_2(exec(q))$, which represents elements sent from PE_p to PE_q;
2.2 send message containing $send_C(p,q)$ to PE_q;

3. perform computations for iterations in $iter(p,p)$, where $iter(p,p) = f_2^{-1}(local_C(p) \cap [l_2 : u_2 : s_2]) \cap exec(p) = f_2^{-1}(send_C(p,p))$, which stands for iterations on PE_p that access only local data, where $u_2 = l_2 + \lfloor (u_1 - l_1)/s_1 \rfloor * s_2$;

4. forall $q \in recv\_pe(p)$, **do**
4.1 receive message containing $recv_C(p,q)$ from PE_q, where $recv_C(p,q) = send_C(q,p)$, which speaks for elements sent from PE_q to PE_p;
4.2 $iter(p,q) = f_2^{-1}(local_C(q) \cap [l_2 : u_2 : s_2]) \cap exec(p) = f_2^{-1}(recv_C(p,q))$, which indicates iterations on PE_p that access local data and message buffers whose contents are received from PE_q;
4.3 execute computations for iterations in $iter(p,q)$.

Figure 3: Implementing a doall statement on a distributed-shared-memory machine.

4.2 Determining Suitable Block Sizes

In [10], we derived communication sets and processor sets with arbitrary block sizes b_1 and b_2. These sets, however, cannot be represented by closed forms. In this section, we return to analyze the block sizes of b_1 and b_2. Our goal is to choose reasonable block sizes b_1 and b_2, so that communication sets can be represented by closed forms.

Consider the target doall statement again. We first present an ideal case. Suppose that we assign the entry $A(j)$ to PE $p = (\lfloor \frac{j-l_1}{s_1 * h} \rfloor \bmod N)$ and the entry $C(j')$ to PE $p' = (\lfloor \frac{j'-l_2}{s_2 * h} \rfloor \bmod N)$, Then, for $i \in \{0, 1, \ldots, h-1\}$, $A(l_1 + i * s_1)$ and $C(l_2 + i * s_2)$ are in PE 0; for $i \in \{h, h+1, \ldots, 2*h-1\}$, $A(l_1 + i * s_1)$ and $C(l_2 + i * s_2)$ are in PE 1; and so on. In addition, there is no communication overhead to perform the target doall statement. In this ideal case, we notice that $b_1 = s_1 * h$ and $b_2 = s_2 * h$.

We now consider the general case. Suppose that the data distribution functions for arrays A and C are $f_A(j) = (\lfloor \frac{j - offset_1}{b_1} \rfloor \bmod N)$ and $f_C(j') = (\lfloor \frac{j' - offset_2}{b_2} \rfloor \bmod N)$, respectively. We found that, even if we don't care about the values of $offset_1$ and $offset_2$, if b_1/s_1 is a factor of b_2/s_2, or b_1/s_1 is a multiple of b_2/s_2, then the communication sets can be represented by closed forms. However, if the condition fails, it will incur communication overhead due to random access patterns whose costs are very expensive. Table 5 summarizes certain conditions where processor sets and communication sets have closed forms.

If these sets can be represented by closed forms,

conditions	A	B	C	D
$send\_pe_C(p)$		$\surd$		$\surd$
$recv\_pe_C(p)$			$\surd$	$\surd$
$send_C(p,q)$		$\surd$	$\surd$	$\surd$
$recv_C(p,q)$		$\surd$	$\surd$	$\surd$

Table 5: Conditions when processor sets and communication sets have closed forms. (A) arbitrary b_1 and b_2, (B) b_1/s_1 is a factor of b_2/s_2, (C) b_1/s_1 is a multiple of b_2/s_2, and (D) b_1/s_1 is a factor of b_2/s_2 and $(b_2 * s_1)/(b_1 * s_2)$ is a factor or a multiple of N, or when b_1/s_1 is a multiple of b_2/s_2 and $(b_1 * s_2)/(b_2 * s_1)$ is a factor or a multiple of N.

then they can be implemented efficiently. Otherwise, we only can use *ad hoc* methods to enumerate these sets and use indirectly memory access methods to get their corresponding data. The latter case, of course, will incur certain computation overhead. Therefore, our goal is to determine suitable block sizes such that the more sets can be represented by closed forms the better. In the following, we show examples to illustrate the flavor of choosing block sizes. We assume that the iteration space of a doall statement is large enough such that each PE has to execute roughly the same amount of iterations.

Example 1: Suppose that the loop bodies of two consecutive doall statements are

$A(l_1 + i * s_1) = A(l_1 + i * s_1) + C(l_2 + i * s_2)$ and
$A(l_1 + i * s_1) = A(l_1 + i * s_1) - D(l_3 + i * s_3)$.

In this case, we choose $b_1 = s_1 * h$, $b_2 = s_2 * h$, and $b_3 = s_3 * h$, where block sizes b_1, b_2, and b_3 are for arrays A, C, and D, respectively. Then all sets: $send_C(p,q)$, $recv_C(p,q)$, $send\_pe_C(p)$, and $recv\_pe_C(p)$ for the first doall statement, as well as $send_D(p,q)$, $recv_D(p,q)$, $send\_pe_D(p)$, and $recv\_pe_D(p)$ for the second doall statement all have closed forms. □

Example 2: Suppose that the loop bodies of two consecutive doall statements are

$A(l_1 + i * s_1) = A(l_1 + i * s_1) * C(l_2 + i * s_2)$ and
$C(l_4 + i * s_4) = C(l_4 + i * s_4) + D(l_3 + i * s_3)$.

In this case, we choose $b_1 = s_1 * h$, $b_2 = \mathrm{lcm}(s_2, s_4) * h$, and $b_3 = s_3 * h$. Then, except $recv\_pe_C(p)$ for the first doall statement and $send\_pe_D(p)$ for the second doall statement, all other sets have closed forms. □

Example 3: Suppose that the loop bodies of two consecutive doall statements are

$A(l_1 + i * s_1) = A(l_1 + i * s_1) - C(l_2 + i * s_2)$ and
$A(l_4 + i * s_4) = A(l_4 + i * s_4) * D(l_3 + i * s_3)$.

In this case, we choose $b_1 = \mathrm{lcm}(s_1, s_4) * h$, $b_2 = s_2 * h$, and $b_3 = s_3 * h$. Then, except $send\_pe_C(p)$ for the first doall statement and $send\_pe_D(p)$ for the second doall statement, all other sets have closed forms. □

4.3 Using Closed Forms to Represent Communication Sets

In this subsection, we derive processor sets and communication sets for cases when $b_1 = s_1 * h_1$ and $b_2 = s_2 * h_1 * h_2$. Thus, b_1/s_1 is a factor of b_2/s_2. The derivation of closed forms of other cases can be seen in [10]. We assume that the data distribution function for array A is $f_A(j) = (\lfloor \frac{j-a_1}{b_1} \rfloor \bmod N)$, thus, $local_A(p) = [[a_1 + pb_1 : a_1 + pb_1 + b_1 - 1] : a_2 : Nb_1]$. The data distribution function for array C is $f_C(j') = (\lfloor \frac{j'-c_1}{b_2} \rfloor \bmod N)$, thus, $local_C(p) = [[c_1 + pb_2 : c_1 + pb_2 + b_2 - 1] : c_2 : Nb_2]$. We also assume that $(u_1 - l_1)$ is a multiple of s_1 and $u_2 = l_2 + ((u_1 - l_1)/s_1) * s_2$.

The function $nxt(a,b,c)$ we use here, is the smallest integer greater than a and is congruent to b *modulo* c, that is, $nxt(a,b,c) = a + (b-a) \bmod c$. We now introduce some notations which will be used later.

$$
\begin{aligned}
bot_l(A,p,j) &= a_1 + pb_1 + jNb_1 \\
top_l(A,p,j) &= a_1 + pb_1 + b_1 - 1 + jNb_1 \\
bot_a(A,p,j) &= nxt(\max\{bot_l(A,p,j),\, l_1\},\ l_1,\ s_1) \\
top_a(A,p,j) &= nxt(\min\{top_l(A,p,j),\, u_1\} - s_1 + 1, \\
&\qquad\qquad l_1,\ s_1) \\
bot_e(A,p,j) &= (bot_a(A,p,j) - l_1)/s_1 \\
top_e(A,p,j) &= (top_a(A,p,j) - l_1)/s_1 \\
bot_f(A,p,j) &= bot_e(A,p,j)s_2 + l_2 \\
top_f(A,p,j) &= top_e(A,p,j)s_2 + l_2 \\
bot_l(C,p,k) &= c_1 + pb_2 + kNb_2 \\
top_l(C,p,k) &= c_1 + pb_2 + b_2 - 1 + kNb_2 \\
bot_a(C,p,k) &= nxt(\max\{bot_l(C,p,k),\, l_2\},\ l_2,\ s_2) \\
top_a(C,p,k) &= nxt(\min\{top_l(C,p,k),\, u_2\} - s_2 + 1, \\
&\qquad\qquad l_2,\ s_2) \\
bot_f(C,p,k) &= bot_e(C,p,k)s_1 + l_1 \\
top_f(C,p,k) &= top_e(C,p,k)s_1 + l_1.
\end{aligned}
$$

Let j_{pf} and j_{pl} be the first j and the last j such that $[bot_l(A,p,j) : top_l(A,p,j)] \cap [l_1 : u_1 : s_1] \neq \phi$, respectively; and k_{pf} and k_{pl} be the first k and the last k such that $[bot_l(C,p,k) : top_l(C,p,k)] \cap [l_2 : u_2 : s_2] \neq \phi$, respectively. We now introduce a property, which will be used in $send\_pe(p)$ and $recv\_pe(p)$.

Property 1 *Suppose that array A is distributed by cyclic(b_1); $f_A(i)$ is the data distribution function of array A; x and y are two indices of array A, where $x < y$. Then, we have* $f_A([x:y]) =$

$$
\begin{cases}
[0 : N-1], & \textit{if } y - x + 1 > (N-1) * b_1; \\
[f_A(x) : f_A(y)], & \\
\quad \textit{if } y - x + 1 \le (N-1) * b_1 \textit{ and } f_A(x) \le f_A(y); & \\
[0 : f_A(y)] \cup [f_A(x) : N-1], & \\
\quad \textit{if } y - x + 1 \le (N-1) * b_1 \textit{ and } f_A(x) > f_A(y). &
\end{cases}
$$

In [10], we investigated the periodic property of communication sets. Let $period_e$ be the period of the iteration pattern in $exec(p)$ such that $period_e * s_1$ is a multiple of Nb_1; $period_{eb}$ be the number of blocks of local elements of array A whose access pattern appears periodically; $period_s$ be the period of the reference pattern of array C in $send_C(p,q)$ whose value is a multiple of Nb_2; $period^C_{sb}$ be the number of blocks of local elements of array C whose reference pattern appears periodically; and $period^A_{sb}$ be the number of blocks of local elements of array A, whose reference pattern of local elements of array C (based on $f_2(exec(q))$) appears periodically. We have the following equations.

$$period_e = (\text{lcm}(Nb_1, s_1))/s_1$$
$$period_{eb} = (period_e * s_1)/(Nb_1)$$
$$period_s = \text{lcm}(Nb_2, period_e * s_2)$$
$$period^C_{sb} = period_s/(Nb_2)$$
$$period^A_{sb} = (period_s * s_1)/(Nb_1 s_2).$$

We now formulate $exec(p)$. Since $period_{eb} = (period_e * s_1)/(Nb_1) = \text{lcm}(Nb_1, s_1)/(Nb_1) = 1$ and $Nb_1/s_1 = Nh_1$, we have $bot_e(A,p,j+1) = bot_e(A,p,j) + Nh_1$ for $j_{pf} < j < j_{pl}$. Thus, we have

$$\begin{aligned} exec(p) &= f_1^{-1}(local_A(p) \cap [l_1 : u_1 : s_1]) \\ &= f_1^{-1}\left(\textstyle\bigcup_{j=j_{pf}}^{j_{pl}} [bot_a(A,p,j) : top_a(A,p,j) : s_1]\right) \\ &= \textstyle\bigcup_{j=j_{pf}}^{j_{pl}} [bot_e(A,p,j) : top_e(A,p,j)]. \end{aligned}$$

Now, we derive processor sets and communication sets. First, we process $send\_pe(p)$, which is equal to $f_A(f_1(f_2^{-1}(local_C(p) \cap [l_2 : u_2 : s_2])))$. Since $period^C_{sb} = period_s/(Nb_2) = \text{lcm}(Nb_2, period_e * s_2)/(Nb_2) = 1$, it is enough to analyze the set of PEs which use elements of array C within a block of size b_2. We found that if $h_2 \geq N$, then every PE will use some elements of array C within a block of size b_2. If $h_2 < N$, then the left boundary element and the right boundary element of array C within a block of size b_2 are referred by $f_A(bot_f(C,p,k_{pl}))$ and $f_A(top_f(C,p,k_{pf}))$, respectively. Note that, if $nxt(bot_l(C,p,k_{pf}), l_2, s_2) < l_2$, then $f_A(bot_f(C,p,k_{pf}))$ maybe is not equal to $f_A(bot_f(C,p,k_{pl}))$. Based on Property 1, we have the following closed form.

$send\_pe(p) =$
$$\begin{cases} [0 : N-1], \quad \text{if } u_2 - l_2 + 1 \geq Nb_2 \text{ and } h_2 \geq N; \\ [f_A(bot_f(C,p,k_{pl})) : f_A(top_f(C,p,k_{pf}))], \\ \quad \text{if } u_2 - l_2 + 1 \geq Nb_2,\ h_2 < N, \text{ and} \\ \quad f_A(bot_f(C,p,k_{pl})) \leq f_A(top_f(C,p,k_{pf})); \\ [0 : f_A(top_f(C,p,k_{pf}))] \cup \\ \qquad [f_A(bot_f(C,p,k_{pl})) : N-1], \\ \quad \text{if } u_2 - l_2 + 1 \geq Nb_2,\ h_2 < N, \text{ and} \\ \quad f_A(bot_f(C,p,k_{pl})) > f_A(top_f(C,p,k_{pf})); \\ f_A([bot_f(C,p,k_{pf}) : top_f(C,p,k_{pf})]) \cup \\ \qquad f_A([bot_f(C,p,k_{pl}) : top_f(C,p,k_{pl})]), \\ \quad \text{if } u_2 - l_2 + 1 < Nb_2. \end{cases}$$

Second, we formulate $recv\_pe(p)$, which is equal to $f_C(f_2(exec(p)))$. We start from $exec(p)$ and check the elements of array C that these iterations will refer to. Recall that $exec(p) = \bigcup_{j=j_{pf}}^{j_{pl}} [bot_e(A,p,j) : top_e(A,p,j)]$. Then, $f_2(exec(p)) = \bigcup_{j=j_{pf}}^{j_{pl}} [bot_f(A,p,j) : top_f(A,p,j) : s_2]$, which represents the elements of array C that are referred by iterations executed in PE_p; and $f_C(f_2(exec(p)))$ indicates the set of PEs that store these elements of array C. Since $period^A_{sb} = (period_s * s_1)/(Nb_1 s_2) = h_2$, $recv\_pe(p)$ can be represented by a union of at most $h_2 + 1$ closed forms.

$recv\_pe(p) =$
$$\begin{cases} [0 : N-1], \ \text{if } u_2 - l_2 + 1 \geq Nb_2 \text{ and } h_2 \geq N; \\ \bigcup_{j=j_{pf}}^{j_{pf}+h_2-1} f_C([bot_f(A,p,j) : top_f(A,p,j)]), \\ \quad \text{if } u_2 - l_2 + 1 \geq Nb_2,\ h_2 < N, \text{ and} \\ \quad nxt(bot_l(A,p,j_{pf}), l_1, s_1) \geq l_1; \\ \left(\bigcup_{j=j_{pf}}^{j_{pf}+h_2-1} f_C([bot_f(A,p,j) : top_f(A,p,j)])\right) \cup \\ \qquad f_C([bot_f(A,p,j_{pf}+h_2) : \\ \qquad\quad nxt(l_2 + period_s - s_2, l_2, s_2)]), \\ \quad \text{if } u_2 - l_2 + 1 \geq Nb_2,\ h_2 < N, \text{ and} \\ \quad nxt(bot_l(A,p,j_{pf}), l_1, s_1) < l_1; \\ \bigcup_{j=j_{pf}}^{j_{pl}} f_C([bot_f(A,p,j) : top_f(A,p,j)]), \\ \quad \text{if } u_2 - l_2 + 1 < Nb_2. \end{cases}$$

Note that, in the above formula, the set $f_C([bot_f(A,p,j) : top_f(A,p,j)])$ consists of only one or two PEs. In addition, all these PEs are distinct. However, in spite of these facts, $recv\_pe(p)$ still cannot be represented by a constant number of closed forms independent of h_2.

Third, we deal with $send_C(p,q)$, which is equal to $local_C(p) \cap f_2(exec(q))$. This set will be represented by a union of three closed forms: $shead_C(p,q)$, $sbody^1_C(p,q)$, and $sbody^2_C(p,q)$. Before deriving $send_C(p,q)$, we show an example to explain where these three closed forms come from.

Example 4: Suppose that the number of PEs is 4; the loop body of a doall statement is $A(11+2i) = g(C(2+i))$, where g is a function; and $u_1 = 745$. Then, $l_1 = 11$; $s_1 = 2$; $l_2 = 2$; $s_2 = 1$; and $u_2 = 369$. If we let $h_1 = 2$ and $h_2 = 11$, then $b_1 = s_1 * h_1 = 4$ and $b_2 = s_2 * h_1 * h_2 = 22$.

Figure 4 shows elements of array C in PE_0 and the corresponding PEs which will refer to these elements. Among them, $send_C(0,1) = shead_C(0,1) \cup sbody^1_C(0,1) \cup sbody^2_C(0,1)$, where $shead_C(0,1) = [7 : 8 : 1] \cup [[15 : 16 : 1] : 21 : 8]$; $sbody^1_C(0,1) = [[88 : 88 : 1] : 369 : 88]$; and $sbody^2_C(0,1) = [[[95 : 96 : 1] : 109 : 8] : 369 : 88]$. $send_C(0,2) = shead_C(0,2) \cup sbody^2_C(0,2)$, where $shead_C(0,2) = [2 : 2 : 1] \cup [[9 : 10 : 1] : 21 : 8]$ and $sbody^2_C(0,2) = [[[89 : 90 : 1] : 109 : 8] : 369 : 88]$. Note that, $shead_C(0,1)$ is on purpose written by a union of two closed forms, as we will derive a unified formula to represent $shead(p,q)$. Next, $sbody^1_C(0,2) = \phi$. $\square$

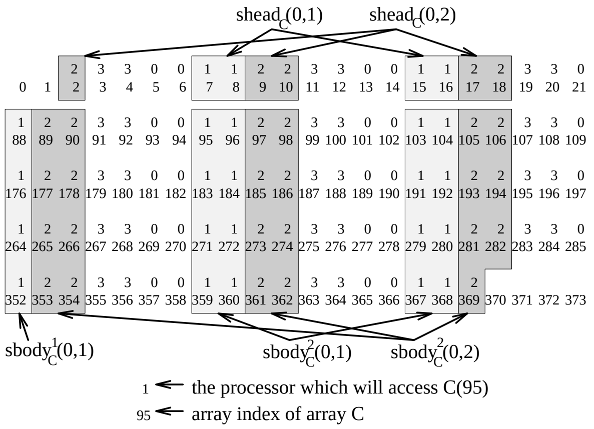

Figure 4: Elements of array C in PE_0, where array C is distributed by $cyclic(22)$ over four processors. In addition, $send_C(0,q) = shead_C(0,q) \cup sbody_C^1(0,q) \cup sbody_C^2(0,q)$, for $1 \le q \le 3$.

We notice that $shead_C(p,q)$ is not empty if $nxt(bot_l(C,p,k_{pf}), l_2, s_2) < l_2$; $sbody_C^1(p,q)$ includes some elements if $bot_l(C,p,k)$ is in between $bot_f(A,q,j)+1$ and $top_f(A,q,j)$ for some j and k; and $sbody_C^2(p,q)$ will be evaluated without any conditions. In addition, the period of $f_2(exec(q))$ is $period_e * s_2 = Ns_2h_1$ and $period_s = \mathrm{lcm}(Nb_2, period_e * s_2) = Nb_2$. Let $k'_{pf} = k_{pf} + 1$ if $nxt(bot_l(C,p,k_{pf}), l_2, s_2) < l_2$, $k'_{pf} = k_{pf}$ otherwise. Then, we have

$$shead_C(p,q) = \begin{cases} [bot_f(A,q,j_{qf}) : \min\{top_f(A,q,j_{qf}), top_a(C,p, \\ \quad k_{pf})\} : s_2] \cup [[bot_f(A,q,j_{qf}+1) : top_f(A,q, \\ \quad j_{qf}+1) : s_2] : top_a(C,p,k_{pf}) : Ns_2h_1], \\ \quad \text{if } nxt(bot_l(C,p,k_{pf}), l_2, s_2) < l_2; \\ \phi, \quad \text{otherwise.} \end{cases}$$

$$sbody_C^1(p,q) = \begin{cases} [[bot_a(C,p,k'_{pf}) : nxt(bot_a(C,p,k'_{pf}), \\ \quad top_f(A,q,j_{qf}), Ns_2h_1) : s_2] : u_2 : Nb_2], \\ \text{if } bot_f(A,q,j_{qf}) < bot_a(C,p,k'_{pf}) \le top_f(A,q, \\ j_{qf}) \text{ or } nxt(bot_a(C,p,k'_{pf}), top_f(A,q,j_{qf}), \\ Ns_2h_1) - s_2(h_1-1) < bot_a(C,p,k'_{pf}) \le \\ \min\{nxt(bot_a(C,p,k'_{pf}), top_f(A,q,j_{qf}), Ns_2h_1), \\ top_f(A,q,j_{ql})\}; \\ \phi, \quad \text{otherwise.} \end{cases}$$

$$sbody_C^2(p,q) = [[[nxt(bot_a(C,p,k'_{pf}), bot_f(A,q,j_{qf}+1), Ns_2h_1) : nxt(bot_a(C,p,k'_{pf}), bot_f(A,q,j_{qf}+1), Ns_2h_1) + s_2(h_1-1) : s_2] : top_a(C,p,k'_{pf}) : Ns_2h_1] : u_2 : Nb_2].$$

$$send_C(p,q) = shead_C(p,q) \cup sbody_C^1(p,q) \cup sbody_C^2(p,q).$$

Fourth, we are concerned with $recv_C(p,q)$, which is equal to $send_C(q,p)$. Hence, $recv_C(p,q)$ also can be represented by a union of three closed forms [10].

4.4 Experimental Studies

In this subsection, we present experimental studies of calculating a saxpy operation on two data arrays on a nCUBE-2 computer.

Example 5: Consider the following saxpy operation:

doall $i = 0,\ 80639$

$$A(1997 + 3*i) = A(1997 + 3*i) + CON * C(5 + 2*i),$$

where CON is a floating-point constant. In addition, array A is distributed by a $cyclic(b_1)$ distribution; array C is distributed by a $cyclic(b_2)$ distribution. Table 6 lists experimental results of implementing this saxpy operation with various block sizes b_1 and b_2. Experimental results can be distilled as follows.

1. The execution time of computing the cases when $b_1 = s_1 * h$ and $b_2 = s_2 * h * h'$ is close to that of the cases when $b_1 = s_1 * h * h'$ and $b_2 = s_2 * h$.

2. When h' is less than the number of PEs, then the execution time becomes better when h' is close to 1. This is because, in these cases, each PE needs to send data messages to at most $(h'+1)$ PEs. Therefore, the communication time reduces when h' becomes smaller.

3. When h' is larger than or equal to the number of PEs, then the execution time improves when block sizes b_1 and b_2 are increasingly larger. This may illustrate that our algorithm favors the cases when block sizes are large, because in these cases, the indexing overhead for packing data messages is not significant.

4. All cases except one show scalable improvements when the number of PEs grows. One exception case is when the number of PEs is 16 and $b_1 = 3$ and $b_2 = 10080$. This is because in the extreme block to cyclic cases or cyclic to block cases, the indexing overhead for packing data messages is significant; in addition, the communication overhead also becomes worse when the number of PEs grows because of involving certain *all-to-all* communications.

5. Because the iteration space is linear and each PE executes roughly the same number of iterations, there is no load unbalance problem. Therefore, according to the communication oracle, it is preferable to choose large block sizes b_1 and b_2.

From Table 6, we can summarize that it is preferable to choose block sizes $b_1 \geq 63$ and $b_2 \geq 42$ for this saxpy operation. □

b_2	b_1 / #PE	3	9	63	315	945	3780
2	2	1.334	1.241	0.879	0.829	0.821	0.821
	4	0.668	0.702	0.476	0.428	0.420	0.417
	8	0.334	0.353	0.274	0.225	0.217	0.214
	16	0.168	0.186	0.173	0.125	0.117	0.113
6	2	1.238	0.783	0.564	0.516	0.508	0.508
	4	0.707	0.392	0.318	0.270	0.262	0.259
	8	0.356	0.197	0.186	0.145	0.137	0.134
	16	0.181	0.099	0.095	0.085	0.077	0.074
42	2	0.875	0.582	0.404	0.380	0.371	0.369
	4	0.472	0.326	0.203	0.199	0.193	0.189
	8	0.270	0.186	0.102	0.105	0.102	0.099
	16	0.172	0.096	0.052	0.055	0.060	0.057
210	2	0.826	0.534	0.395	0.344	0.347	0.349
	4	0.426	0.278	0.209	0.186	0.180	0.179
	8	0.223	0.149	0.110	0.094	0.094	0.094
	16	0.124	0.087	0.057	0.048	0.049	0.053
630	2	0.819	0.527	0.387	0.357	0.331	0.344
	4	0.419	0.271	0.201	0.187	0.201	0.177
	8	0.216	0.141	0.107	0.097	0.101	0.092
	16	0.116	0.079	0.063	0.050	0.052	0.049
2520	2	0.819	0.525	0.384	0.363	0.359	0.332
	4	0.417	0.268	0.198	0.188	0.187	0.168
	8	0.213	0.139	0.103	0.099	0.098	0.084
	16	0.113	0.076	0.059	0.056	0.050	0.043
5040	2	0.818	0.524	0.383	0.362	0.358	0.361
	4	0.416	0.268	0.197	0.187	0.185	0.188
	8	0.213	0.138	0.103	0.098	0.099	0.096
	16	0.113	0.076	0.058	0.056	0.053	0.049
10080	2	0.817	0.522	0.380	0.361	0.357	0.358
	4	0.415	0.268	0.196	0.186	0.184	0.187
	8	0.212	0.138	0.102	0.097	0.097	0.098
	16	0.293	0.135	0.066	0.057	0.056	0.050

Table 6: Execution time (second) of computing the saxpy operation in Example 5 using 2 PEs, 4 PEs, 8 PEs, and 16 PEs, respectively.

5 Conclusions

Several techniques for determining data distribution and generating communication sets on distributed memory machines have been presented in this paper. First, we proposed a cost model which emphasized that the total execution time should include both the computation time and the communication time. This cost model was then used to determine the granularity of data distribution. We also extended Li and Chen's component alignment algorithm and developed a dynamic programming algorithm for heuristically determining whether data redistribution was necessary.

Second, we derived formulas to represent communication sets of executing doall statements. We found that there existed closed forms to represent communication sets if data arrays were distributed according to certain restrictions. Experimental studies also showed that the indexing overhead of the proposed closed forms was not significant and the approach scaled well as the number of processors increased.

References

[1] S. Benkner, P. Brezany, and H. Zima. Processing array statements and procedure interfaces in the PREPARE high performance Fortran compiler. In *Lecture Notes in Computer Science 786*, pages 324–338, 1993.

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

[3] 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.

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

[5] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. In *Proc. of ACM International Conf. on Supercomputing*, pages 392–403, Manchester, U.K., July 1994.

[6] 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.

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

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

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

[10] P.-Z. Lee and W. Y. Chen. Compiler techniques for determining data distribution and generating communication sets on distributed-memory multicomputers. Technical Report TR-IIS-95-007, Institute of Information Science, Academia Sinica, 1995.

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

[12] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

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

[14] Ellis Wade Jr. and Ed Lodi. *A Tutorial Introduction to Derive*. Brooks/Cole Publishing Company, Pacific Grove, California, 1991.

[15] 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.