Compiler Techniques for Determining Data Distribution and Generating Communication Sets on Distributed-Memory Multicomputers[1]

PeiZong Lee and Wen-Yao Chen
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

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. This approach is different from previous research works, which only allow programmers explicitly to specify the data distribution using language extensions. 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 block-cyclic fashion. Our result contributes towards automatic compilation of sequential programs to message-passing version programs running on distributed memory parallel computers. Our methods also 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.

Keywords: communication set, component alignment, data distribution, distributed memory computer, doall statement, dynamic programming algorithm for data distribution, parallelizing compiler.

*** A preliminary version of this technical report is accepted to be presented at the **29th IEEE Hawaii International Conference on System Sciences**, Maui, Hawaii, January 3-6, 1996.

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

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 [16]. 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 DO-loop program or 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 [4] [20]. 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 among PEs, 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 [15] [17]. Recently, a lot of research works are concentrated on the more general cases when an array's distribution is in a block-cyclic fashion [3] [6] [7] [8] [11] [12] [22]. Furthermore, methods to generate aggregate communication operations based on pattern matching techniques are also proposed [19]. We are interested in integrating previous research works and in formulating a complete set of closed forms of generating communication sets for each PE.

The rest of this paper is organized as follows. In Section 2, we introduce some background of compiling sequential programs on distributed memory multicomputers. In Section 3, we present algorithms to determine data distribution at compiling time. In Section 4, we derive formulas to represent communication sets for *doall statements* with arbitrary block sizes. In Section 5, we propose closed forms to represent communication sets for doall statements with restricted block sizes. Finally, some concluding remarks are given in Section 6.

2 Background

2.1 Nomenclature

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. For instance, $[1 : 102] = \{1, 2, 3, \cdots, 102\}$.
- $[a : e_1 : s_1]$ 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. For example, $[1 : 102 : 40] = \{1, 41, 81\}$.
- $[[a : e_1] : e_2 : s_2]$ specifies the set $\{[a : e_1], [a : e_1] + s_2, [a : e_1] + 2s_2, \cdots$, until not greater than $e_2\}$. Thus, $[[1 : 30] : 102 : 40] = \{1, 2, 3, \cdots, 30, 41, 42, 43, \cdots, 70, 81, 82, 83, \cdots, 102\}$.
- $[[a : e_1 : s_1] : e_2 : s_2]$ means 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 $e_2\}$. Thus, $[[1 : 30 : 10] : 102 : 40] = \{1, 11, 21, 41, 51, 61, 81, 91, 101\}$.
- $[[[a : e_1] : e_2 : s_2] : e_3 : s_3]$ stands for the set $\{[[a : e_1] : e_2 : s_2], [[a : e_1] : e_2 : s_2] + s_3, [[a : e_1] : e_2 : s_2] + 2s_3, \cdots$, until not greater than $e_3\}$. Thus, $[[[1 : 3] : 30 : 10] : 102 : 40] = \{1, 2, 3, 11, 12, 13, 21, 22, 23, 41, 42, 43, 51, 52, 53, 61, 62, 63, 81, 82, 83, 91, 92, 93, 101, 102\}$.
- $[[[a : e_1 : s_1] : e_2 : s_2] : e_3 : s_3]$ 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 $e_3\}$. For instance, [[1 :

$3 : 2] : 30 : 10] : 102 : 40] = \{1, 3, 11, 13, 21, 23, 41, 43, 51, 53, 61, 63, 81, 83, 91, 93, 101\}$.

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 $1 < b < \lceil (a_2 - a_1 + 1)/N \rceil$.

2.2 Distributed Memory Multicomputers

In this paper, we are concerned with distributed memory systems. The abstract target machine we adopt is a q-D grid of $N_1 \times N_2 \times \cdots \times N_q$ 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 $1 \leq i \leq q$. Such a topology can be easily embedded into almost all distributed memory machines, including massively configured parallel computers. For example, the q-D grid can be embedded into a hypercube computer using a binary reflected Gray code.

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 [9]. More precisely, in general, a source program has sequential parts (which must be executed sequentially) and concurrent parts (which can be executed concurrently). Each PE will execute the sequential parts individually; while all PEs will execute the concurrent parts altogether by using message passing communication primitives. In practice, scalar variables and small data arrays used in the program are replicated on all PEs in order to reduce communication costs; while large data arrays are partitioned and distributed among PEs. In this paper, we adopt a global name space for representing large data arrays among PEs. Therefore, our machine model can be regarded as a distributed shared memory model [15] [17] [21].

2.3 Compiling Sequential Programs on Distributed Memory Machines

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 [25]. 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 and Figure 2 show three programs: (a) a sequential program for solving a linear system AX = B, (b) its corresponding doall loop program, and (c) its corresponding SPMD program in which data arrays are distributed by $cyclic(b)$. 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. *}

(a)

```
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
```

(b)

```
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
```

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

```
(c)  {*  Hand compiled output SPMD program using the global name space for N processors.
         Matrix A is distributed by cyclic(b) along its rows;  Arrays X, Y, and B are distributed by cyclic(b);
         A working matrix T1 and a working array T2 are replicated in all processors.  *}

     # define my$p = who_am_i()   {*  return myself processor ID  *}
     REAL  A( [[my$p * b : my$p * b + b - 1] : m - 1 : N * b], [0 : m - 1] ),  T1( [0 : b - 1], [0 : m - 1] )
     REAL  X( [[my$p * b : my$p * b + b - 1] : m - 1 : N * b] ),  Y( [[my$p * b : my$p * b + b - 1] : m - 1 : N * b] )
     REAL  B( [[my$p * b : my$p * b + b - 1] : m - 1 : N * b] ),  T2( [0 : b - 1] )

             {*  A = LU.  *}
     do k$ = 0, m - 1, N * b
        do pivot$p = 0, N - 1
           start$k = k$ + pivot$p * b
           end$k = start$k + b - 1
           if (my$p = pivot$p)
              do k = start$k, end$k
                 do i = k + 1, end$k
                    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
              broadcast( A( [start$k : end$k], [start$k : m-1] ) )
           else
              receive( pivot$p, T1( [0 : b-1], [start$k : m-1] ) )
           endif

           if (my$p > pivot$p)
              start$i = k$ + my$p * b
           else
              start$i = k$ + (my$p + N) * b
           endif

           do i$ = start$i, m - 1, N * b
              do k = start$k, end$k
                 do i = i$, i$ + b - 1
                    A(i, k) = A(i, k) / T1(k-start$k, k)
                    do j = k + 1, m - 1
                       A(i, j) = A(i, j) - A(i, k) * T1(k-start$k, j)
     enddo  enddo  enddo  enddo  enddo enddo

             {*  LY = B.  *}
     do i$ = 0, m - 1, N * b
        do pivot$p = 0, N - 1
           start$i = i$ + pivot$p * b
           end$i = start$i + b - 1
           if (my$p = pivot$p)
              do i = start$i, end$i
                 Y(i) = B(i)
                 do j = i + 1, end$i
                    B(j) = B(j) - A(j, i) * Y(i)
              enddo  enddo
              broadcast( Y( [start$i : end$i] ) )
           else
              receive( pivot$p, T2( [0 : b-1] ) )
           endif

           if (my$p > pivot$p)
              start$j = i$ + my$p * b
           else
              start$j = i$ + (my$p + N) * b
           endif

           do j$ = start$j, m - 1, N * b
              do i = start$i, end$i
                 do j = j$, j$ + b - 1
                    B(j) = B(j) - A(j, i) * T2(i-start$i)
     enddo enddo enddo enddo enddo

             {*  UX = Y.  *}
     do i$ = m - N * b, 0, -(N * b)
        do pivot$p = N - 1, 0, -1
           start$i = i$ + pivot$p * b
           end$i = start$i + b - 1
           if (my$p = pivot$p)
              do i = end$i, start$i, -1
                 X(i) = Y(i) / A(i, i)
                 do j = start$i, i - 1
                    Y(j) = Y(j) - A(j, i) * X(i)
              enddo  enddo
              broadcast( X( [start$i : end$i] ) )
           else
              receive( pivot$p, T2( [0 : b-1] ) )
           endif

           if (my$p < pivot$p)
              end$j = i$ + my$p * b
           else
              end$j = i$ + (my$p - N) * b
           endif

           do j$ = my$p * b, end$j, N * b
              do i = end$i, start$i, -1
                 do j = j$, j$ + b - 1
                    Y(j) = Y(j) - A(j, i) * T2(i-start$i)
     enddo enddo enddo enddo enddo
```

Figure 2: (c) The corresponding hand compiled output SPMD program.

3 Determining Data Distribution at Compiling Time

In this section, we show how to use a component alignment algorithm to determine data distribution. This method is also adopted by other researchers [2, 5, 18, 20]. Because we will generalize previous methods to deal with a wider class of problems, in the following, we describe this method in a great detail.

We 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.

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.

Readers can find that Case 2 is a special case of Case 3. In Section 4 and Section 5, 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. As to the costs of Case 6 through Case 8, they are considerably higher than those of Case 1 through Case 5.

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. Edges are defined in two ways. First, if subscripts of dimensions of the array (or matrix) in the left-hand-side of '=' have affinity relations to the subscripts of dimensions of the array(s) in 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 some pairs of subscripts of dimensions of these two arrays have affinity relations, and in addition, none of subscripts in these two arrays have affinity relations to 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. 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: The communication cost required if the corresponding two dimensions of arrays of an edge in the component affinity graph are distributed along different dimensions of the processor grid. These costs depend on array's dimensionalities on both the tail of the edge and the head of that edge.

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

Note that, the component affinity graph defined in this paper is slightly different from the one introduced by Li and Chen [20]. First, their component affinity graph is undirected. Second, in their model, they only showed the definition of an affinity relation for some specific cases. For example, if a left-hand-side subscript and a right-hand-side subscript have an affinity relation, then the left-hand-side subscript must be an index variable of a Do-loop and the right-hand-side subscript must be an affine function of the form $i + c$, where i is the index variable and c is an integer. Third, in their graph, there are only edges between left-hand-side objects and right-hand-side objects. This is because their model follows the restricted owner computes rule. Therefore, the computation of assignment statements can be performed only after all right-hand-side objects are sent to the owner PEs of the left-hand-side objects. The owner computes paradigm simplifies code generation considerably. However, it does not provide an adequate solution when the right-hand side of assignment statements contain complex expressions, and whose operands must be sent to the owner of the left-hand-side objects.

For example, if two operands of a binary operator are aligned, it is better to compute this binary operator first and then send <u>one</u> intermediate result to the owner of the left-hand-side object, than to send these <u>two</u> operands independently to the owner of the left-hand-side object and then perform the computation due to this binary operator. For this purpose, we found that it is necessary to specify the direction of the data communication according to a relaxed owner computes rule based on the costs defined in Table 2. More precisely, right-hand-side objects are sent to the owner of the left-hand-side objects as the conventional owner computes rule; objects of lower dimensional arrays are sent to the owner of the higher dimensional arrays in order to reduce the communication cost, if these objects are the corresponding operands of binary operators.

Note that, although the component alignment problem is NP-complete, Li and Chen have proposed an efficient heuristic algorithm based on applying the optimal matching procedure to a bipartite graph constructed from the nodes corresponding to components (dimensions) of two data arrays [20]. In this paper, when dealing with component alignment problems, we adopt Li and Chen's heuristic algorithm by regarding our directed component affinity graphs as undirected ones. The direction of edges, however, are used in a code-generation phase and will be used to determine the direction of the data communication according to the owner computes rule. For completeness, in Figure 3, we only present a very brief version of the component alignment algorithm; however, interested readers can refer to the original paper for the details about this method [20].

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 a 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 new selected array;
- **Step 3.3** combine the matched nodes with the basic set as a new basic set.

Figure 3: Heuristic component alignment algorithm.

We now return to our example of the linear system. Figure 4 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.

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

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.

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* ($cyclic(1)$) 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

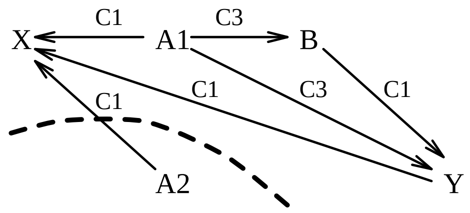

Figure 4: Component affinity graph and the suggested component alignment of the sample program which solves a linear system based on the LU decomposition.

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, however, 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, 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. Suppose that the time of executing the LU decomposition is T_{LU}; the time of executing two triangular linear systems is T_{2TLS}; and the total execution time is T. Then, from the SPMD program in Figure 2, we can formulate T, T_{LU}, and T_{2TLS} as follows.

$$
\begin{aligned}
T &= T_{LU} + T_{2TLS} \\
T_{LU} &= \sum_{i_1=1}^{m/(N*b)} \sum_{i_2=1}^{N} \Big\{ 9 * t_f + \sum_{i_3=1}^{b} \sum_{i_4=i_3+1}^{b} \Big(t_f + (m - ((i_1 - 1) * N * b + (i_2 - 1) * b + i_3) + 1) * t_x \Big)
\end{aligned}
$$

$$
\begin{aligned}
& + \sum_{i_5=i_1}^{m/(N*b)} \sum_{i_6=1}^{b} \sum_{i_7=1}^{b} \Big(t_f + (m - ((i_1-1)*N*b + (i_2-1)*b + i_6) + 1) * t_x \Big) \\
& + (\log N + 1) * b * (t_s + (m - ((i_1-1)*N*b + (i_2-1)*b)) * t_c) \Big\} \\
T_{2TLS} \;=\; & 2 * \sum_{i_1=1}^{m/(N*b)} \sum_{i_2=1}^{N} \Big\{ 9 * t_f + \sum_{i_3=1}^{b} \Big(t_f + (b - i_3) * t_x \Big) + \sum_{i_4=i_1}^{m/(N*b)} \sum_{i_5=1}^{b} \sum_{i_6=1}^{b} t_x \\
& + (\log N + 1) * (t_s + b * t_c) \Big\}
\end{aligned}
$$

The symbolic manipulations of the above formulas can be solved using a computer algebra system like "Derive" [24]. 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 5.

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. For instance, when computing a 2-D FFT for a data matrix, we usually 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 certain communication overhead due to requiring several "bit-reverse shuffle-exchange" and "butterfly-pattern" data communications. However, if we perform a transpose operation for the matrix between calculating 1-D FFTs for all rows and 1-D FFTs for all columns, then no communication operations are required during evaluating each 1-D FFT. 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

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 $A_{1024\times1024}X_{1024} = B_{1024}$ 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.

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 $1 \leq i \leq s$ and $1 \leq j \leq s - i + 1$. Define $T_{i,j}$ to be 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 $P_{i,j}$.

Heuristic algorithm for determining whether data redistribution is necessary:

A dynamic programming 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 $T_{1,i}$ $(= M_{1,i})$, where $1 \leq i \leq s$ and $1 \leq j \leq s - i + 1$.

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. $T_{i,j} := \text{MIN}_{1 \leq k < i}\{T_{i-k,\,k} + M_{i,\,j} + cost(P_{i-k,\,k},\, P_{i,\,j})\}$;
4. **end_for end_for**
5. $Minimum\_Cost := \text{MIN}_{1 \leq k \leq s}\{T_{s-k+1,\,k}\}$.

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

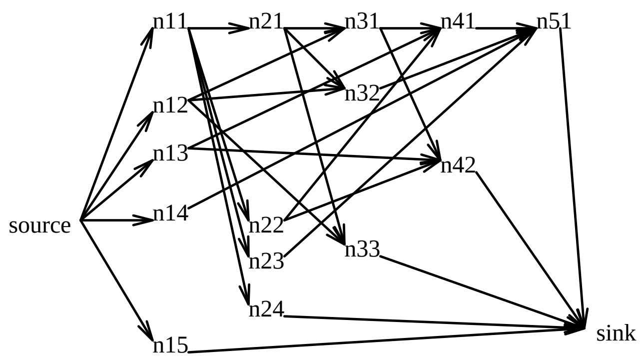

Figure 5: The corresponding single-source shortest paths problem for $s = 5$.

The data distribution scheme obtained from the above algorithm 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 describe how to improve this dynamic programming algorithm. It is clear that $M_{i,(\gamma_i+1)} \geq M_{i,\gamma_i}$, for $1 \leq \gamma_i < s-i+1$. 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 $1 \leq \beta < \gamma_i + 1$, 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}, ..., L_{i+j-1}, than to use only one distribution scheme $P_{i,j}$, for $\gamma_i + 1 < j \leq s - i + 1$. Therefore, we **need not** compute $M_{i,j}$. Based on this observation, we can show that $T_{i,(\gamma_i+1)} > T_{(i+\beta),(\gamma_i-\beta+1)}$ and $T_{i,j} > T_{(i+\gamma_i+1),(j-\gamma_i-1)}$. Therefore, we **need not** compute $T_{i,j}$, for $\gamma_i + 1 \leq j \leq s - i + 1$.

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 $1 \leq \beta \leq \gamma_i \leq s - i + 1$. 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, $\gamma = \max_{1 \leq i \leq s}\{\gamma_i\}$. Then the above dynamic programming algorithm can be improved as follows.

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

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}, ..., L_{i+j-1}, where $1 \leq i \leq s$ and $1 \leq j \leq \gamma_i + 1$. The total number of component alignment problems computed is thus no more than $s(\gamma + 1)$.

4 Generating Communication Sets for Doall Statements

Single-loop doall statements have the same power as *one-dimensional array statements.* For instance, the following table shows the equivalence relation between single-loop doall statements and one-dimensional array statements, where g is a function of array C.

doall statements	array statements
doall $i = 0,\ \lfloor \frac{u_1 - l_1}{s_1} \rfloor$ $A(l_1 + i * s_1) = g(C(l_2 + i * s_2))$	$A(l_1\ :\ u_1\ :\ s_1) = g(C(l_2\ :\ u_2\ :\ s_2))$
doall $i = l,\ u,\ s$ $A(l_1 + i * s_1) = g(C(l_2 + i * s_2))$	$A(l_1 + l * s_1\ :\ l_1 + l * s_1 + \lfloor \frac{u-l}{s} \rfloor * s * s_1\ :\ s * s_1) =$ $g(C(l_2 + l * s_2\ :\ l_2 + l * s_2 + \lfloor \frac{u-l}{s} \rfloor * s * s_2\ :\ s * s_2))$

In this section, we are interested in generating all necessary communication sets in each PE when a single-loop doall statement is executed by distributed memory machines. In the sequel, we will use *doall statements* to represent single-loop doall statements. In the following, we state the problem we want to solve in this section.

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

The case when s_1 or s_2 is negative can be treated analogously. This problem has been studied before. Koelbel and Mehrotra pioneeredly provided closed-form representations for the special cases when $l_1 = 0$, $s_1 = 1$, $a_1 = c_1 = 0$, $a_2 = c_2 = m - 1$, and arrays are distributed in *block* or *cyclic* distributions [15, 17]. The following researchers concerned with $block{-}cyclic$ ($cyclic(b)$) distributions. Although they only formulated "send sets" and "receive sets", none of them got closed-form representations. Stichnoth et al. pointed out that a $cyclic(b_i)$ distribution can be regarded as a union of b_i $cyclic(1)$ ($cyclic$) distributions. Since there exists closed forms to represent communication sets for *cyclic* distributions, communication sets for $block{-}cyclic$ distributions can thus be represented by a union of $b_1 * b_2$ closed forms [22]. Gupta et al. proposed closed forms for representing communication sets for arrays that are distributed using *block* or *cyclic* distributions. These closed forms are then used with a virtual processor approach to give a solution for arrays with $block{-}cyclic$ distributions [7]. The above two approaches did not discover periodic patterns in communication sets.

Chatterjee et al. enumerated the local memory access sequence of communication sets based on a finite-state machine [3]. Kennedy et al. also presented algorithms, which were based on a finite-state machine and an integer lattice method, for computing the local memory access sequence [8] [13] [14]. They also noticed that the data access patterns in the communication sets appeared periodically. They calculated communication sets based on a scanning technique similar to the merge sort for computing the intersection of two reference patterns corresponding to the left-hand-side and the right-hand-side array subscripts [8]. Their methods, however, would incur certain runtime overheads due to indirect addressing of data. Independently, Benkner et al. also proposed a similar technique and implemented in their Prepare HPF compiler [1].

Next, for the special case when the parameters $a_1 = c_1$; $a_2 = c_2$; $l_1 = l_2 = 0$; and $s_1 = s_2 = 1$, the mentioned problem is reduced to a data redistribution problem. Research on this data redistribution problem also have been reported [6] [10] [11] [12] [23].

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 6 shows a detailed outline of implementing a doall statement in each PE which is a generalization based on formulas presented in [15].

Step 1 of Figure 6 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.

4.2 The Derivation of Communication Sets

In this subsection, we derive communication sets and processor sets with arbitrary block sizes b_1 and b_2. Without loss of generality, we assume that $(a_2 - a_1 + 1)$ is a multiple of Nb_1 and $(c_2 - c_1 + 1)$ is a multiple of Nb_2. The data distribution function for array A is $f_A(j) = (\lfloor \frac{j - a_1}{b_1} \rfloor \mod N)$, 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

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, where $local_A(p) = [[a_1 + p * b_1 : a_1 + p * b_1 + b_1 - 1] : a_2 : N * b_1]$;

1.2 $send\_pe(p) = \{q \,|\, q \neq p$ and PE_p will send some data to $PE_q\}$;

1.3 $recv\_pe(p) = \{q \,|\, q \neq p$ 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, where $local_C(p) = [[c_1 + p * b_2 : c_1 + p * b_2 + b_2 - 1] : c_2 : N * b_2]$;

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 some message buffers whose contents are received from PE_q;

4.3 execute computations for iterations in $iter(p, q)$.

Figure 6: Implementing a doall statement on a distributed-shared-memory machine.

is $f_C(j') = (\lfloor \frac{j' - c_1}{b_2} \rfloor \mod N)$, 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 $nxt(x, y, z)$ we use here, is the smallest integer greater than x and is congruent to y modulo z, that is, $nxt(x, y, z) = x + (y - x) \bmod z$. In Table 5, we introduce some notations which will be used later.

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. Figure 7 shows an algorithm for computing j_{pf} and j_{pl}. k_{pf} and k_{pl} also can be computed similarly. The value $j_{start} = \lceil (l_1 - a_1 - pb_1 - b_1 + 1)/(Nb_1) \rceil$ is the first j such that $top_l(A, p, j) \geq l_1$. The value $j_{final} = \lfloor (u_1 - a_1 - pb_1)/(Nb_1) \rfloor$ is the last j such that $bot_l(A, p, j) \leq u_1$. If $s_1 \leq b_1$, then $j_{start} = j_{pf}$ and $j_{final} = j_{pl}$. If $s_1 > b_1$, we need to check other details.

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

Table 5: Notations which will be used in deriving sets.

We now return to the derivation. Because $exec(p)$ will be used for deriving other communication sets and processor sets, we formulate it first. We have the following relations.

$$
\begin{aligned}
local_A(p) &= \bigcup_{j=0}^{\frac{a_2-a_1+1}{Nb_1}-1} [bot_l(A,p,j) : top_l(A,p,j)] \\
exec(p) &= f_1^{-1}(local_A(p) \cap [l_1 : u_1 : s_1]) \\
&= f_1^{-1}\left(\bigcup_{j=j_{pf}}^{j_{pl}} [bot_a(A,p,j) : top_a(A,p,j) : s_1]\right) \\
&= \bigcup_{j=j_{pf}}^{j_{pl}} [bot_e(A,p,j) : top_e(A,p,j)].
\end{aligned}
$$

Note that, in the expression $[bot_e(A,p,j) : top_e(A,p,j)]$, it may occur that $bot_e(A,p,j) > top_e(A,p,j)$ when $s_1 > b_1$. Throughout this paper, if $\alpha > \beta$, then $[\alpha : \beta]$ is empty. Next, according to the order of appearance in Figure 6, after deriving $exec(p)$, we should present processor sets $send\_pe(p)$ and $recv\_pe(p)$. However, since exact solutions of these two sets are tedious, we prefer to present communication sets $send_C(p,q)$ and $recv_C(p,q)$ first. We now introduce a set $f_2(exec(q))$, which will

```
j_start = ⌈(l_1 − a_1 − pb_1 − b_1 + 1)/(Nb_1)⌉;
j_final = ⌊(u_1 − a_1 − pb_1)/(Nb_1)⌋;
if (s_1 ≤ b_1) then
  j_pf = j_start;
  j_pl = j_final;
else /* s_1 > b_1 */
  j = j_start;
  while (j ≤ j_final) do
    if (bot_a(A, p, j) ≤ top_a(A, p, j))
      j_pf = j;
      break;
    else
      j = j + 1;
    endif
  endwhile
  if (j > j_final) then
    exec(p) = φ;
  else /* j_pf ≤ j_final */
    j = j_final;
    while (j ≥ j_pf) do
      if (bot_a(A, p, j) ≤ top_a(A, p, j))
        j_pl = j;
        break;
      else
        j = j − 1;
      endif
    endwhile
  endif
endif
```

Figure 7: An algorithm for computing j_{pf} and j_{pl}.

be used in deriving $send_C(p,q)$.

$$\begin{aligned} f_2(exec(q)) &= \textstyle\bigcup_{j=j_{qf}}^{j_{ql}} f_2([bot_e(A,q,j) : top_e(A,q,j)]) \\ &= \textstyle\bigcup_{j=j_{qf}}^{j_{ql}} [bot_e(A,q,j)s_2 + l_2 : top_e(A,q,j)s_2 + l_2 : s_2] \\ &= \textstyle\bigcup_{j=j_{qf}}^{j_{ql}} [bot_f(A,q,j) : top_f(A,q,j) : s_2]. \end{aligned}$$

We now define the periodic coefficients of the communication set $send_C(p,q)$, which is equal to $local_C(p) \cap f_2(exec(q))$. Let $period_s$ be the period of the reference pattern of array C in $send_C(p,q)$ whose value is a multiple of Nb_2; $period_{sb}^C$ be the number of blocks of local elements of array C whose reference pattern in $send_C(p,q)$ appears periodically; and $period_{sb}^A$ be the number of blocks of local elements of array A, whose reference pattern of local elements of array C in $send_C(p,q)$ (based on $f_2(exec(q))$) appears periodically. Then, we have the following equations.

$$\begin{aligned} &period_s = \mathrm{lcm}(Nb_2,\ (\mathrm{lcm}(Nb_1, s_1)/s_1) * s_2) \\ &period_{sb}^C = period_s/(Nb_2) \\ &period_{sb}^A = (period_s * s_1)/(Nb_1 s_2). \end{aligned}$$

We now study the intersection of $local_C(p) \cap f_2(exec(q))$, which is equal to $\left(\bigcup_{k=k_{pf}}^{k_{pl}} [bot_l(C,p,k) : top_l(C,p,k)]\right) \cap \left(\bigcup_{j=j_{qf}}^{j_{ql}} [bot_f(A,q,j) : top_f(A,q,j) : s_2]\right)$. We found that if $\lceil \frac{b_1}{s_1} \rceil \leq \lceil \frac{(N-1)b_2+1}{s_2} \rceil$, then each referenced block of array A in PE_q ($[bot_f(A,q,j) : top_f(A,q,j) : s_2]$) will intersect to at most one local block of array C in PE_p ($[bot_l(C,p,k) : top_l(C,p,k)]$). Similarly, if $\lceil \frac{b_2}{s_2} \rceil \leq \lceil \frac{(N-1)b_1+1}{s_1} \rceil$, then each local block of array C in PE_p will also intersect to at most one referenced block of array A in PE_q.

Property 1 *When $N \geq 2$, at least one of the following two conditions is true:* (a) $\lceil \frac{b_1}{s_1} \rceil \leq \lceil \frac{(N-1)b_2+1}{s_2} \rceil$ *and* (b) $\lceil \frac{b_2}{s_2} \rceil \leq \lceil \frac{(N-1)b_1+1}{s_1} \rceil$.

Proof : First, we want to show that if (a) fails then (b) must be true. If (a) fails, then $\lceil \frac{b_1}{s_1} \rceil > \lceil \frac{(N-1)b_2+1}{s_2} \rceil$. We have $\lceil \frac{(N-1)b_1+1}{s_1} \rceil \geq \lceil \frac{b_1}{s_1} \rceil > \lceil \frac{(N-1)b_2+1}{s_2} \rceil \geq \lceil \frac{b_2}{s_2} \rceil$. Therefore, $\lceil \frac{b_2}{s_2} \rceil < \lceil \frac{(N-1)b_1+1}{s_1} \rceil$.

Similarly, we can show that if (b) fails then (a) must be true. $\square$

Property 2 *Let L and R be the left boundary and the right boundary of $[[a : a+b-1] : e : Nb] \cap [\alpha : \beta : \gamma]$, respectively. Suppose that $\lceil \frac{\beta-\alpha+1}{\gamma} \rceil \leq \lceil \frac{(N-1)b+1}{\gamma} \rceil$. Then,*

$$[[a : a+b-1] : e : Nb] \cap [\alpha : \beta : \gamma] \quad = \quad [L : R : \gamma],$$

where

$$L \quad = \quad \begin{cases} \alpha, & \textit{if } \alpha \in [[a : a+b-1] : e : Nb] \\ nxt(nxt(\max\{a,\alpha\}, a, Nb), \alpha, \gamma), & \textit{otherwise;} \end{cases}$$
$$R \quad = \quad \begin{cases} \beta, & \textit{if } \beta \in [[a : a+b-1] : e : Nb] \\ nxt(nxt(\min\{e,\beta\}, a, Nb) - Nb + b - \gamma, \alpha, \gamma), & \textit{otherwise.} \end{cases}$$

Proof : Let L' and R' be the left boundary and the right boundary of $[[a : a+b-1] : e : Nb] \cap [\alpha : \beta]$, respectively. Then,

$$L' \quad = \quad \begin{cases} \alpha, & \text{if } \alpha \in [[a : a+b-1] : e : Nb] \\ nxt(\max\{a,\alpha\}, a, Nb), & \text{otherwise;} \end{cases}$$
$$R' \quad = \quad \begin{cases} \beta, & \text{if } \beta \in [[a : a+b-1] : e : Nb] \\ nxt(\min\{e,\beta\}, a, Nb) - Nb + b - 1, & \text{otherwise.} \end{cases}$$

Since $\lceil \frac{\beta-\alpha+1}{\gamma} \rceil \leq \lceil \frac{(N-1)b+1}{\gamma} \rceil$, $[\alpha : \beta : \gamma]$ will intersect to at most one local block of $[[a : a+b-1] : e : Nb]$. Thus, $[[a : a+b-1] : e : Nb] \cap [\alpha : \beta : \gamma] = [nxt(L', \alpha, \gamma) : nxt(R' - \gamma + 1, \alpha, \gamma) : \gamma] = [L : R : \gamma]$. $\square$

Based on Properties 1 and 2, we can show that $send_C(p,q)$ can be represented by a union of a variable number of closed forms. First, if $\lceil \frac{b_1}{s_1} \rceil \leq \lceil \frac{(N-1)b_2+1}{s_2} \rceil$, $send_C(p,q)$ can be represented as follows.

$$send_C(p,q) \quad = \quad local_C(p) \cap f_2(exec(q))$$

$$
\begin{aligned}
&= [[c_1 + pb_2 \,:\, c_1 + pb_2 + b_2 - 1] \,:\, c_2 \,:\, Nb_2] \cap \left(\bigcup_{j=j_{qf}}^{j_{ql}} [bot_f(A,q,j) : top_f(A,q,j) : s_2]\right) \\
&= \bigcup_{j=j_{qf}}^{j_{ql}} \left([[c_1 + pb_2 \,:\, c_1 + pb_2 + b_2 - 1] \,:\, c_2 \,:\, Nb_2] \cap [bot_f(A,q,j) : top_f(A,q,j) : s_2]\right) \\
&= \bigcup_{j=j_{qf}}^{j_{ql}} [L(j) : R(j) : s_2] \\
&= [L(j_{qf}) : R(j_{qf}) : s_2] \cup \left(\bigcup_{j=j_{qf}+1}^{\min\{j_{ql}, j_{qf}+period_{sb}^{A}\}} [[L(j) : R(j) : s_2] : u_2 : period_s]\right),
\end{aligned}
$$

where

$$
L(j) = \begin{cases} bot_f(A,q,j), & \text{if } bot_f(A,q,j) \in local_C(p) \\ nxt(nxt(\max\{c_1+pb_2, bot_f(A,q,j)\}, c_1+pb_2, Nb_2), l_2, s_2), & \text{otherwise;} \end{cases}
$$

$$
R(j) = \begin{cases} top_f(A,q,j), & \text{if } top_f(A,q,j) \in local_C(p) \\ nxt(nxt(\min\{c_2, top_f(A,q,j)\}, c_1+pb_2, Nb_2) - Nb_2 + b_2 - s_2, l_2, s_2), & \text{otherwise.} \end{cases}
$$

Second, if $\lceil \frac{b_2}{s_2} \rceil \le \lceil \frac{(N-1)b_1+1}{s_1} \rceil$, $send_C(p,q)$ can be represented as follows.

$$
\begin{aligned}
&send_C(p,q) = f_2(exec(q)) \cap local_C(p) \\
&= f_2 f_1^{-1}\left(f_1 f_2^{-1}(f_2(exec(q)) \cap local_C(p))\right) \\
&= f_2 f_1^{-1}\left([[a_1 + qb_1 \,:\, a_1 + qb_1 + b_1 - 1] \,:\, a_2 \,:\, Nb_1] \cap \left(\bigcup_{k=k_{pf}}^{k_{pl}} [bot_f(C,p,k) : top_f(C,p,k) : s_1]\right)\right) \\
&= \bigcup_{k=k_{pf}}^{k_{pl}} f_2 f_1^{-1}\left([[a_1 + qb_1 \,:\, a_1 + qb_1 + b_1 - 1] \,:\, a_2 \,:\, Nb_1] \cap [bot_f(C,p,k) : top_f(C,p,k) : s_1]\right) \\
&= \bigcup_{k=k_{pf}}^{k_{pl}} [f_2 f_1^{-1}(L(k)) : f_2 f_1^{-1}(R(k)) : s_2] \\
&= [f_2 f_1^{-1}(L(k_{pf})) : f_2 f_1^{-1}(R(k_{pf})) : s_2] \cup \\
&\qquad \left(\bigcup_{k=k_{pf}+1}^{\min\{k_{pl}, k_{pf}+period_{sb}^{C}\}} [[f_2 f_1^{-1}(L(k)) : f_2 f_1^{-1}(R(k)) : s_2] : u_2 : period_s]\right),
\end{aligned}
$$

where

$$
L(k) = \begin{cases} bot_f(C,p,k), & \text{if } bot_f(C,p,k) \in local_A(q) \\ nxt(nxt(\max\{a_1+qb_1, bot_f(C,p,k)\}, a_1+qb_1, Nb_1), l_1, s_1), & \text{otherwise;} \end{cases}
$$

$$
R(k) = \begin{cases} top_f(C,p,k), & \text{if } top_f(C,p,k) \in local_A(q) \\ nxt(nxt(\min\{a_2, top_f(C,p,k)\}, a_1+qb_1, Nb_1) - Nb_1 + b_1 - s_1, l_1, s_1), & \text{otherwise.} \end{cases}
$$

Next, we handle $recv_C(p,q)$. Because $recv_C(p,q)$ is equal to $send_C(q,p)$, $recv_C(p,q)$ also can be represented by a union of a variable number of closed forms. Although $recv_C(p,q)$ specifies a set of indices of array C, in practice, we prefer that $recv_C(p,q)$ can be represented based on indices of array A. For instance, the loop body of the doall statement $A(f_1(i)) = g(C(f_2(i)))$ is equivalent to $A(f_1(i)) = g(C(f_2(f_1^{-1}(f_1(i)))))$. Thus, the doall statement can be executed efficiently after receiving data messages from other PEs once we fetch elements of array A. Therefore, our goal is to generate the set corresponding to indices of array A, which is equal to $f_1(f_2^{-1}(recv_C(p,q)))$, because $recv_C(p,q) = f_2 f_1^{-1}(f_1 f_2^{-1}(recv_C(p,q)))$. Since the derivation of $recv_C(p,q)$ is similar to that of $send_C(p,q)$, we omit all of the middle steps, and only present the final formulas.

First, if $\lceil \frac{b_1}{s_1} \rceil \leq \lceil \frac{(N-1)b_2+1}{s_2} \rceil$, $recv_C(p,q)$ can be represented as follows.

$$\begin{aligned} recv_C(p,q) &= f_2 f_1^{-1}(f_1 f_2^{-1}(recv_C(p,q))) = f_2 f_1^{-1}(f_1 f_2^{-1}(send_C(q,p))) \\ &= f_2 f_1^{-1}\Big([f_1 f_2^{-1}(L(j_{pf})) : f_1 f_2^{-1}(R(j_{pf})) : s_1] \cup \\ &\qquad \Big(\textstyle\bigcup_{j=j_{pf}+1}^{\min\{j_{pl}, j_{pf}+period_{sb}^A\}} [[f_1 f_2^{-1}(L(j)) : f_1 f_2^{-1}(R(j)) : s_1] : u_1 : period_s * s_1/s_2]\Big)\Big), \end{aligned}$$

where

$$\begin{aligned} L(j) &= \begin{cases} bot_f(A,p,j), & \text{if } bot_f(A,p,j) \in local_C(q) \\ nxt(nxt(\max\{c_1+qb_2, bot_f(A,p,j)\}, c_1+qb_2, Nb_2), l_2, s_2), & \text{otherwise;} \end{cases} \\ R(j) &= \begin{cases} top_f(A,p,j), & \text{if } top_f(A,p,j) \in local_C(q) \\ nxt(nxt(\min\{c_2, top_f(A,p,j)\}, c_1+qb_2, Nb_2) - Nb_2 + b_2 - s_2, l_2, s_2), & \text{otherwise.} \end{cases} \end{aligned}$$

Second, if $\lceil \frac{b_2}{s_2} \rceil \leq \lceil \frac{(N-1)b_1+1}{s_1} \rceil$, $recv_C(p,q)$ can be represented as follows.

$$\begin{aligned} recv_C(p,q) &= f_2 f_1^{-1}(f_1 f_2^{-1}(recv_C(p,q))) = f_2 f_1^{-1}(f_1 f_2^{-1}(send_C(q,p))) \\ &= f_2 f_1^{-1}\Big([L(k_{qf}) : R(k_{qf}) : s_1] \cup \\ &\qquad \Big(\textstyle\bigcup_{k=k_{qf}+1}^{\min\{k_{ql}, k_{qf}+period_{sb}^C\}} [[L(k) : R(k) : s_1] : u_1 : period_s * s_1/s_2]\Big)\Big), \end{aligned}$$

where

$$\begin{aligned} L(k) &= \begin{cases} bot_f(C,q,k), & \text{if } bot_f(C,q,k) \in local_A(p) \\ nxt(nxt(\max\{a_1+pb_1, bot_f(C,q,k)\}, a_1+pb_1, Nb_1), l_1, s_1), & \text{otherwise;} \end{cases} \\ R(k) &= \begin{cases} top_f(C,q,k), & \text{if } top_f(C,q,k) \in local_A(p) \\ nxt(nxt(\min\{a_2, top_f(C,q,k)\}, a_1+pb_1, Nb_1) - Nb_1 + b_1 - s_1, l_1, s_1), & \text{otherwise.} \end{cases} \end{aligned}$$

We now formulate $send\_pe(p)$ and $recv\_pe(p)$. It is possible to derive exact solutions for $send\_pe(p)$ and $recv\_pe(p)$. However, the computation cost is very expensive in a general case. This is because to test whether q is in $send\_pe(p)$ or whether q is in $recv\_pe(p)$ is equivalent to test whether $send_C(p,q) \neq \phi$ or whether $send_C(q,p) \neq \phi$, respectively. Because of this reason, we consider inexact solutions for $send\_pe(p)$ and $recv\_pe(p)$. We now introduce a property, which will be used to derive $send\_pe(p)$ and $recv\_pe(p)$.

Property 3 *Suppose that array A is distributed by $cyclic(b_1)$; $f_A(i)$, which specifies the PE that stores $A(i)$, is the data distribution function of array A; x and y are two indices of array A, where $x < y$. Then, we have*

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

Property 3 also holds for array C with its corresponding distribution by $cyclic(b_2)$ and its data distribution function f_C. We now process $send\_pe(p)$, which is equal to $f_A(f_1(f_2^{-1}(local_C(p) \cap [l_2:u_2:s_2])))$.

$$\begin{aligned} send\_pe(p) &= f_A(f_1(f_2^{-1}(local_C(p) \cap [l_2:u_2:s_2]))) \\ &= \bigcup_{k=k_{pf}}^{k_{pl}} f_A(f_1([bot_e(C,p,k):top_e(C,p,k)])) \\ &= \bigcup_{k=k_{pf}}^{\min\{k_{pl},k_{pf}+period_{sb}^{C}\}} f_A([bot_f(C,p,k):top_f(C,p,k):s_1]) \\ &\subseteq \bigcup_{k=k_{pf}}^{\min\{k_{pl},k_{pf}+period_{sb}^{C}\}} f_A([bot_f(C,p,k):top_f(C,p,k)]). \end{aligned}$$

Note that, the above formula is an equation only when $s_1 \leq b_1$. Next, we are concerned with $recv\_pe(p)$, which is equal to $f_C(f_2(exec(p)))$.

$$\begin{aligned} recv\_pe(p) &= f_C(f_2(exec(p))) \\ &= \bigcup_{j=j_{pf}}^{j_{pl}} f_C(f_2([bot_e(A,p,j):top_e(A,p,j)])) \\ &= \bigcup_{j=j_{pf}}^{\min\{j_{pl},j_{pf}+period_{sb}^{A}\}} f_C([bot_f(A,p,j):top_f(A,p,j):s_2]) \\ &\subseteq \bigcup_{j=j_{pf}}^{\min\{j_{pl},j_{pf}+period_{sb}^{A}\}} f_C([bot_f(A,p,j):top_f(A,p,j)]). \end{aligned}$$

Note that, the above formula is an equation also only when $s_2 \leq b_2$.

5 Using Closed Forms to Represent Communication Sets

In the last section, we derived communication sets and processor sets with arbitrary block sizes b_1 and b_2. These sets, however, cannot be represented by a constant number of closed forms. For instance, each of these sets only can be represented by a union of $(period_{sb}^{A}+1)$ or $(period_{sb}^{C}+1)$ closed forms. Since the number of boundary indices of these closed forms which we need to calculate is proportional to the corresponding $period_{sb}^{A}$ or $period_{sb}^{C}$, the computation overhead becomes serious if the corresponding $period_{sb}^{A}$ or $period_{sb}^{C}$ is large. 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 processor sets and communication sets can be represented by a constant number of closed forms. In the sequel, we will use *closed forms* to represent *a constant number of closed forms.*

5.1 Determining Suitable Block Sizes b_1 and b_2

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 computation and communication overheads due to random access patterns whose costs are relatively very expensive. Table 6 summarizes certain conditions where processor sets and communication sets have closed forms.

conditions	$send\_pe_C(p)$	$recv\_pe_C(p)$	$send_C(p,q)$	$recv_C(p,q)$
arbitrary b_1 and b_2				
b_1/s_1 is a factor of b_2/s_2	$\surd$		$\surd$	$\surd$
b_1/s_1 is a multiple of b_2/s_2		$\surd$	$\surd$	$\surd$
all-closed-forms condition*	$\surd$	$\surd$	$\surd$	$\surd$

Table 6: Conditions when processor sets and communication sets have closed forms. All-closed-forms condition is when 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.

If these sets can be represented by closed forms, then they can be implemented efficiently. Otherwise, we only can use *ad hoc* methods to enumerate these sets or 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

$$A(l_1 + i * s_1) = A(l_1 + i * s_1) + C(l_2 + i * s_2) \quad \text{and}$$
$$A(l_1 + i * s_1) = A(l_1 + i * s_1) - D(l_3 + i * s_3).$$

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: $send\_pe_C(p)$, $recv\_pe_C(p)$, $send_C(p,q)$, and $recv_C(p,q)$ for the first doall statement, as well as $send\_pe_D(p)$, $recv\_pe_D(p)$, $send_D(p,q)$, and $recv_D(p,q)$ for the second doall statement all have closed forms. □

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

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

First, we may naively 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. Second, we can choose $b_1 = s_1 * (s_4 * h/\gcd(s_2, s_4))$, $b_2 = (s_2 * s_4 * h/\gcd(s_2, s_4))$, and $b_3 = s_3 * (s_2 * h/\gcd(s_2, s_4))$. Then all sets for the above two doall statements have closed forms. Of course, the second choice is better than the first choice. □

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

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

First, we may naively 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. Second, we can choose $b_1 = (s_1 * s_4 * h/\gcd(s_1, s_4))$, $b_2 = s_2 * (s_4 * h/\gcd(s_1, s_4))$, and $b_3 = s_3 * (s_1 * h/\gcd(s_1, s_4))$. Then all sets for the above two doall statements have closed forms. Certainly, the second choice is better than the first choice. □

In the following, we derive processor sets and communication sets for three cases in Table 6.

5.2 The Case When $b_1 = s_1 * h_1$ and $b_2 = s_2 * h_1 * h_2$

In this case, b_1/s_1 is a factor of b_2/s_2. Therefore, $send\_pe_C(p)$, $send_C(p,q)$, and $recv_C(p,q)$ have closed forms. 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 $period_s = Nb_2$ and $period_{sb}^C = period_s/(Nb_2) = 1$, 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 3, we have the following closed form.

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

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 $period_{sb}^A = (period_s * s_1)/(Nb_1 s_2) = h_2$, $recv\_pe(p)$ can be represented by a union of at most $h_2 + 1$ closed forms.

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

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 processors 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 8 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, $sbody^1_C(0,2) = \phi$. $\square$

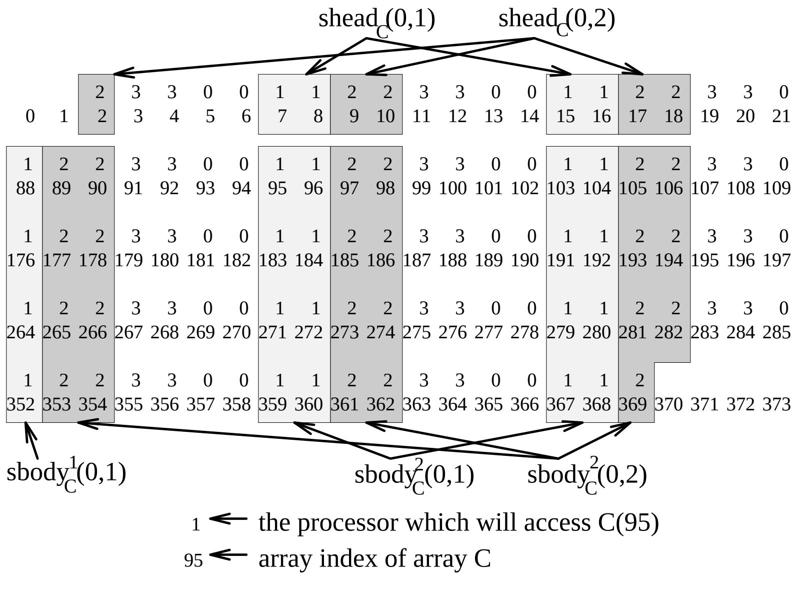

Figure 8: Elements of array C in PE_0, where array C is distributed by $cyclic(22)$ over four processors. In addition, $send_C(0,q) = shead_C(0,q) \cup sbody^1_C(0,q) \cup sbody^2_C(0,q)$, for $1 \le q \le 3$.

We notice that $shead_C(p,q)$ is not empty if $nxt(bot_l(C,p,k_{pf}), l_2, s_2) < l_2$; $sbody^1_C(p,q)$ 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 $sbody^2_C(p,q)$ will be evaluated without any conditions. In addition, the period of $f_2(exec(q))$ is

$(\mathrm{lcm}(Nb_1, s_1)/s_1) * s_2 = Ns_2h_1$ and $period_s = Nb_2$. 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

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

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

$$\begin{aligned} sbody^2_C(p, q) &= [[[nxt(bot_a(C, p, k'_{pf}), bot_f(A, q, j_{qf}+1), Ns_2h_1) : \\ &\quad\; nxt(bot_a(C, p, k'_{pf}), bot_f(A, q, j_{qf}+1), Ns_2h_1) + s_2(h_1 - 1) : s_2] : \\ &\quad\; top_a(C, p, k'_{pf}) : Ns_2h_1] : u_2 : Nb_2]. \\ send_C(p, q) &= shead_C(p, q) \cup sbody^1_C(p, q) \cup sbody^2_C(p, q). \end{aligned}$$

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. As indicated in Section 4, we prefer that $recv_C(p, q)$ can be represented based on indices of array A. In addition, there is a one-to-one correspondence between $rhead_C(p, q) \cup rbody^1_C(p, q) \cup rbody^2_C(p, q)$ and $f_2 f_1^{-1}\Big(f_1(f_2^{-1}(rhead_C(p, q))) \cup f_1(f_2^{-1}(rbody^1_C(p, q))) \cup f_1(f_2^{-1}(rbody^2_C(p, q)))\Big)$. Let $k'_{qf} = k_{qf} + 1$ if $nxt(bot_l(C, q, k_{qf}), l_2, s_2) < l_2$, $k'_{qf} = k_{qf}$ otherwise. Then, $recv_C(p, q)$ can be represented as follows.

$$rhead_C(p, q) = \begin{cases} f_2 f_1^{-1}\Big([bot_a(A, p, j_{pf}) : \min\{top_a(A, p, j_{pf}), top_f(C, q, k_{qf})\} : s_1] \; \cup \\ \quad [[bot_a(A, p, j_{pf}+1) : top_a(A, p, j_{pf}+1) : s_1] : top_f(C, q, k_{qf}) : Nb_1]\Big), \\ \qquad \text{if } nxt(bot_l(C, q, k_{qf}), l_2, s_2) < l_2; \\ \phi, \quad \text{otherwise.} \end{cases}$$

$$rbody^1_C(p, q) = \begin{cases} f_2 f_1^{-1}\Big([[bot_f(C, q, k'_{qf}) : nxt(bot_f(C, q, k'_{qf}), top_a(A, p, j_{pf}), Nb_1) : s_1] : \\ \quad u_1 : Nb_1h_2]\Big), \\ \qquad \text{if } bot_f(A, p, j_{pf}) < bot_a(C, q, k'_{qf}) \le top_f(A, p, j_{pf}) \text{ or} \\ \qquad \quad nxt(bot_a(C, q, k'_{qf}), top_f(A, p, j_{pf}), Ns_2h_1) - s_2(h_1 - 1) < bot_a(C, q, k'_{qf}) \\ \qquad \quad \le \min\{nxt(bot_a(C, q, k'_{qf}), top_f(A, p, j_{pf}), Ns_2h_1), top_f(A, p, j_{pl})\}; \\ \phi, \quad \text{otherwise.} \end{cases}$$

$$\begin{aligned} rbody^2_C(p, q) &= f_2 f_1^{-1}\Big([[[nxt(bot_f(C, q, k'_{qf}), bot_a(A, p, j_{pf}+1), Nb_1) : \\ &\quad\; nxt(bot_f(C, q, k'_{qf}), bot_a(A, p, j_{pf}+1), Nb_1) + s_1(h_1 - 1) : s_1] : \\ &\quad\; top_f(C, q, k'_{qf}) : Nb_1] : u_1 : Nb_1h_2]\Big). \end{aligned}$$

$$recv_C(p,q) \;=\; rhead_C(p,q) \cup rbody^1_C(p,q) \cup rbody^2_C(p,q).$$

In the following, we show an example to explain how to relate indices of array A to $recv_C(p,q)$.

Example 5: We continue Example 4. Figure 9 shows elements of array A in PE_1 through PE_3, and the corresponding PEs that store elements of array C, which will be used to modify elements of array A. Among them, $f_1(f_2^{-1}(recv_C(1,0))) = f_1(f_2^{-1}(rhead_C(1,0))) \cup f_1(f_2^{-1}(rbody^1_C(1,0))) \cup f_1(f_2^{-1}(rbody^2_C(1,0)))$, where $f_1(f_2^{-1}(rhead_C(1,0))) = [21:23:2] \cup [[37:39:2]:49:16]$; $f_1(f_2^{-1}(rbody^1_C(1,0))) = [[183:183:2]:745:176]$; and $f_1(f_2^{-1}(rbody^2_C(1,0))) = [[[197:199:2]:225:16]:745:176]$. $f_1(f_2^{-1}(recv_C(2,0))) = f_1(f_2^{-1}(rhead_C(2,0))) \cup f_1(f_2^{-1}(rbody^2_C(2,0)))$, where $f_1(f_2^{-1}(rhead_C(2,0))) = [11:11:2] \cup [[25:27:2]:49:16]$ and $f_1(f_2^{-1}(rbody^2_C(2,0))) = [[[185:187:2]:225:16]:745:176]$. $\square$

5.3 The Case When $b_1 = s_1 * h_1 * h_2$ and $b_2 = s_2 * h_2$

This case has a symmetrical scene as the case in the last subsection because b_1/s_1 is a multiple of b_2/s_2. Therefore, $recv\_pe_C(p)$, $send_C(p,q)$, and $recv_C(p,q)$ have closed forms. 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 $period^C_{sb} = period_s/(Nb_2) = h_1$, $send\_pe(p)$ can be represented by a union of at most h_1+1 closed forms.

$$send\_pe(p) = \begin{cases} [0:N-1], \quad \text{if } u_1 - l_1 + 1 \geq Nb_1 \text{ and } h_1 \geq N; \\ \bigcup_{k=k_{pf}}^{k_{pf}+h_1-1} f_A([bot_f(C,p,k) : top_f(C,p,k)]), \\ \qquad \text{if } u_1 - l_1 + 1 \geq Nb_1,\ h_1 < N, \text{ and } nxt(bot_l(C,p,k_{pf}), l_2, s_2) \geq l_2; \\ \bigcup_{k=k_{pf}}^{k_{pf}+h_1-1} f_A([bot_f(C,p,k) : top_f(C,p,k)]) \cup \\ \quad f_A([bot_f(C,p,k_{pf}+h_1) : f_1((nxt(l_2 + period_s - s_2, l_2, s_2) - l_2)/s_2)]), \\ \qquad \text{if } u_1 - l_1 + 1 \geq Nb_1,\ h_1 < N, \text{ and } nxt(bot_l(C,p,k_{pf}), l_2, s_2) < l_2; \\ \bigcup_{k=k_{pf}}^{k_{pl}} f_A([bot_f(C,p,k) : top_f(C,p,k)]), \quad \text{if } u_1 - l_1 + 1 < Nb_1. \end{cases}$$

Note that, $send\_pe(p)$ cannot be represented by a constant number of closed forms independent of h_1.

Second, we formulate $recv\_pe(p)$, which is equal to $f_C(f_2(exec(p)))$. Since $period^A_{sb} = (period_s * s_1)/(Nb_1 s_2) = 1$, it is enough to analyze the set of PEs, which store elements of array C that will be

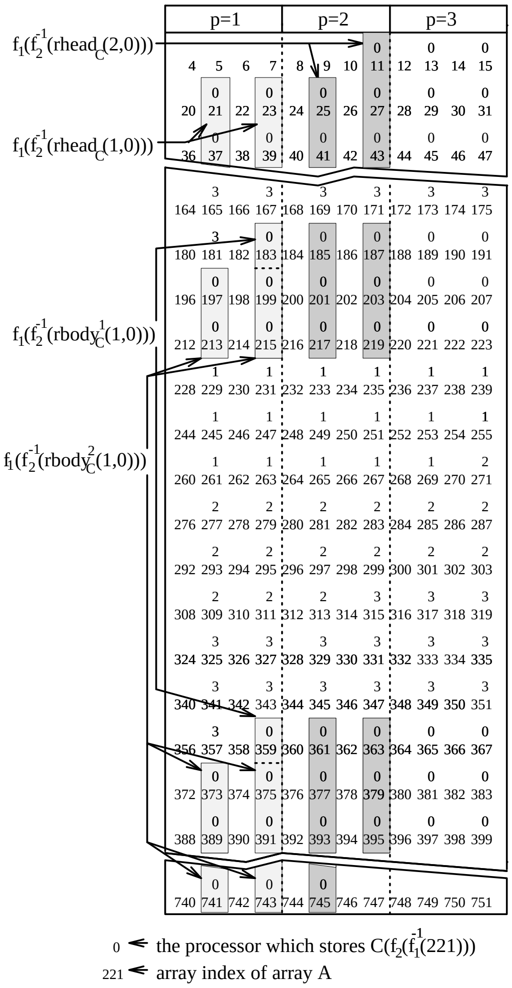

Figure 9: Elements of array A in PE_1 through PE_3, where array A is distributed by $cyclic(4)$ over four processors. In addition, $f_1(f_2^{-1}(recv_C(p,0))) = f_1(f_2^{-1}(rhead_C(p,0))) \cup f_1(f_2^{-1}(rbody_C^1(p,0))) \cup f_1(f_2^{-1}(rbody_C^2(p,0)))$, for $1 \leq p \leq 3$.

accessed by elements of array A within a block of size b_1.

$$recv\_pe(p) \;=\; \begin{cases} [0:N-1], \quad \text{if } u_1 - l_1 + 1 \geq Nb_1 \text{ and } h_1 \geq N; \\ [f_C(bot_f(A,p,j_{pl})) : f_C(top_f(A,p,j_{pf}))], \\ \qquad \text{if } u_1 - l_1 + 1 \geq Nb_1,\ h_1 < N, \text{ and } f_C(bot_f(A,p,j_{pl})) \leq f_C(top_f(A,p,j_{pf})); \\ [0 : f_C(top_f(A,p,j_{pf}))] \cup [f_C(bot_f(A,p,j_{pl})) : N-1], \\ \qquad \text{if } u_1 - l_1 + 1 \geq Nb_1,\ h_1 < N, \text{ and } f_C(bot_f(A,p,j_{pl})) > f_C(top_f(A,p,j_{pf})); \\ f_C([bot_f(A,p,j_{pf}) : top_f(A,p,j_{pf})]) \cup f_C([bot_f(A,p,j_{pl}) : top_f(A,p,j_{pl})]), \\ \qquad \text{if } u_1 - l_1 + 1 < Nb_1. \end{cases}$$

Third, we deal with $send_C(p,q)$, which is equal to $local_C(p) \cap f_2(exec(q))$. This set can be represented by a union of three closed forms: $shead_C(p,q)$, $sbody_C^1(p,q)$, and $sbody_C^2(p,q)$. Let $j_{qf}^{'} = j_{qf} + 1$

if $nxt(bot_l(A,q,j_{qf}),l_1,s_1) < l_1$, $j'_{qf} = j_{qf}$ otherwise.

$$
shead_C(p,q) = \begin{cases} [bot_a(C,p,k_{pf}) : \min\{top_a(C,p,k_{pf}), top_f(A,q,j_{qf})\} : s_2] \ \cup \\ \quad [[bot_a(C,p,k_{pf}+1) : top_a(C,p,k_{pf}+1) : s_2] : top_f(A,q,j_{qf}) : Nb_2], \\ \qquad \text{if } nxt(bot_l(A,q,j_{qf}),l_1,s_1) < l_1; \\ \phi, \quad \text{otherwise.} \end{cases}
$$

$$
sbody^1_C(p,q) = \begin{cases} [[bot_f(A,q,j'_{qf}) : nxt(bot_f(A,q,j'_{qf}), top_a(C,p,k_{pf}), Nb_2) : s_2] : u_2 : Nb_2h_1], \\ \qquad \text{if } bot_a(C,p,k_{pf}) < bot_f(A,q,j'_{qf}) \le top_a(C,p,k_{pf}) \text{ or} \\ \qquad\quad nxt(bot_f(A,q,j'_{qf}), top_a(C,p,k_{pf}), Nb_2) - s_2(h_2-1) < bot_f(A,q,j'_{qf}) \\ \qquad\quad \le \min\{nxt(bot_f(A,q,j'_{qf}), top_a(C,p,k_{pf}), Nb_2), top_a(C,p,k_{pl})\}; \\ \phi, \quad \text{otherwise.} \end{cases}
$$

$$
\begin{aligned}
sbody^2_C(p,q) &= [[[nxt(bot_f(A,q,j'_{qf}), bot_a(C,p,k_{pf}+1), Nb_2) : \\
&\qquad nxt(bot_f(A,q,j'_{qf}), bot_a(C,p,k_{pf}+1), Nb_2) + s_2(h_2-1) : s_2] : \\
&\qquad top_f(A,q,j'_{qf}) : Nb_2] : u_2 : Nb_2h_1]. \\
send_C(p,q) &= shead_C(p,q) \cup sbody^1_C(p,q) \cup sbody^2_C(p,q).
\end{aligned}
$$

Fourth, we manage $recv_C(p,q)$, which is equal to $send_C(q,p)$. Hence, it also can be represented by a union of three closed forms. As stated before, we prefer that $recv_C(p,q)$ can be represented based on indices of array A. Let $j'_{pf} = j_{pf}+1$ if $nxt(bot_l(A,p,j_{pf}),l_1,s_1) < l_1$, $j'_{pf} = j_{pf}$ otherwise. Then, we have

$$
rhead_C(p,q) = \begin{cases} f_2f_1^{-1}\Big([bot_f(C,q,k_{qf}) : \min\{top_f(C,q,k_{qf}), top_a(A,p,j_{pf})\} : s_1] \ \cup \\ \quad [[bot_f(C,q,k_{qf}+1) : top_f(C,q,k_{qf}+1) : s_1] : top_a(A,p,j_{pf}) : Ns_1h_2]\Big), \\ \qquad \text{if } nxt(bot_l(A,p,j_{pf}),l_1,s_1) < l_1; \\ \phi, \quad \text{otherwise.} \end{cases}
$$

$$
rbody^1_C(p,q) = \begin{cases} f_2f_1^{-1}\Big([[bot_a(A,p,j'_{pf}) : nxt(bot_a(A,p,j'_{pf}), top_f(C,q,k_{qf}), Ns_1h_2) : s_1] : \\ \quad u_1 : Nb_1]\Big), \\ \qquad \text{if } bot_a(C,q,k_{qf}) < bot_f(A,p,j'_{pf}), \le top_a(C,q,k_{qf}) \text{ or} \\ \qquad\quad nxt(bot_f(A,p,j'_{pf}), top_a(C,q,k_{qf}), Nb_2) - s_2(h_2-1) < bot_f(A,p,j'_{pf}) \\ \qquad\quad \le \min\{nxt(bot_f(A,p,j'_{pf}), top_a(C,q,k_{qf}), Nb_2), top_a(C,q,k_{ql})\}; \\ \phi, \quad \text{otherwise.} \end{cases}
$$

$$
\begin{aligned}
rbody^2_C(p,q) &= f_2f_1^{-1}\Big([[[nxt(bot_a(A,p,j'_{pf}), bot_f(C,q,k_{qf}+1), Ns_1h_2) : \\
&\qquad nxt(bot_a(A,p,j'_{pf}), bot_f(C,q,k_{qf}+1), Ns_1h_2) + s_1(h_2-1) : s_1] : \\
&\qquad top_a(A,p,j'_{pf}) : Ns_1h_2] : u_1 : Nb_1]\Big). \\
recv_C(p,q) &= rhead_C(p,q) \cup rbody^1_C(p,q) \cup rbody^2_C(p,q).
\end{aligned}
$$

5.4 The Case When Both $send\_pe(p)$ and $recv\_pe(p)$ Have Closed Forms

When 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 both $send\_pe(p)$ and $recv\_pe(p)$ have closed forms.

In the first case, let $b_1 = s_1 * h_1$, $b_2 = s_2 * h_1 * h_2$, and h_2 is either a factor of N or a multiple of N. In this case, $send\_pe(p)$ can be represented by closed forms as presented in Section 5.2. In the following, we show that $recv\_pe(p)$ also can be represented by closed forms.

$$recv\_pe(p) = \begin{cases} [0 : N-1], \quad \text{if } u_2 - l_2 + 1 \geq Nb_2 \text{ and } h_2 \geq N; \\ [f_C(bot_f(A,p,j_{pf})) : \\ \qquad f_C(bot_f(A,p,j_{pf})) + \min\{N-1, (j_{pl} - j_{pf})N/h_2\} : N/h_2] \bmod N, \\ \quad \text{if } h_2 < N \text{ and } f_C(bot_f(A,p,j)) = f_C(top_f(A,p,j)), \text{ for all } j_{pf} \leq j \leq j_{pf}+1; \\ [[f_C(top_f(A,p,j_{pf})) - 1 : f_C(top_f(A,p,j_{pf}))] : \\ \qquad f_C(top_f(A,p,j_{pf})) + \min\{N-2, (j_{pl} - j_{pf})N/h_2\} : N/h_2] \bmod N, \\ \quad \text{if } h_2 < N \text{ and } f_C(bot_f(A,p,j)) \neq f_C(top_f(A,p,j)), \text{ for some } j_{pf} \leq j \leq j_{pf}+1. \end{cases}$$

Note that, the above closed form has two exceptions. First, when $f_C(bot_f(A,p,j_{pf})) = f_C(top_f(A,p,j_{pf}))$, then $(f_C(top_f(A,p,j_{pf})) - 1) \bmod N$ is not in $recv\_pe(p)$. Second, when $u_2 - l_2 + 1 < Nb_2$ and $f_C(bot_f(A,p,j_{pl})) = f_C(top_f(A,p,j_{pl}))$, then $(f_C(top_f(A,p,j_{pf})) + (j_{pl} - j_{pf})N/h_2) \bmod N$ is not in $recv\_pe(p)$.

In the second case, let $b_1 = s_1 * h_1 * h_2$, $b_2 = s_2 * h_2$, and h_1 is either a factor of N or a multiple of N. In this case, $recv\_pe(p)$ can be represented by closed forms as presented in Section 5.3. In the following, we show that $send\_pe(p)$ also can be represented by closed forms.

$$send\_pe(p) = \begin{cases} [0 : N-1], \quad \text{if } u_1 - l_1 + 1 \geq Nb_1 \text{ and } h_1 \geq N; \\ [f_A(bot_f(C,p,k_{pf})) : \\ \qquad f_A(bot_f(C,p,k_{pf})) + \min\{N-1, (k_{pl} - k_{pf})N/h_1\} : N/h_1] \bmod N, \\ \quad \text{if } h_1 < N \text{ and } f_A(bot_f(C,p,k)) = f_A(top_f(C,p,k)), \text{ for all } k_{pf} \leq k \leq k_{pf}+1; \\ [[f_A(top_f(C,p,k_{pf})) - 1 : f_A(top_f(C,p,k_{pf}))] : \\ \qquad f_A(top_f(C,p,k_{pf})) + \min\{N-2, (k_{pl} - k_{pf})N/h_1 : N/h_1] \bmod N, \\ \quad \text{if } h_1 < N \text{ and } f_A(bot_f(C,p,k)) \neq f_A(top_f(C,p,k)), \text{ for some } k_{pf} \leq k \leq k_{pf}+1. \end{cases}$$

Note that, the above closed form also has two exceptions. First, when $f_A(bot_f(C,p,k_{pf})) = f_A(top_f(C,p,k_{pf}))$, then $(f_A(top_f(C,p,k_{pf})) - 1) \bmod N$ is not in $send\_pe(p)$. Second, when $u_1 - l_1 + 1 < Nb_1$ and $f_A(bot_f(C,p,k_{pl})) = f_A(top_f(C,p,k_{pl}))$, then $(f_A(top_f(C,p,k_{pf})) + (k_{pl} - k_{pf})N/h_1) \bmod N$ is not in $send\_pe(p)$.

5.5 Experimental Studies

In this subsection, we present two experimental studies on a nCUBE-2 computer. For each experimental study, the execution time required by each processor to execute the node program was measured and the maximum finish time was reported. The first experimental study calculates a saxpy operation on two data arrays; the second experimental study performs a data redistribution operation on a specific data array. In effect, the data redistribution operation can be seen as a special case of the saxpy operation.

Example 6: Consider the following saxpy operation:

$$\textbf{doall}\ i = 0,\ 80639$$
$$A(1997 + 3 * i) = A(1997 + 3 * i) + saxpy\_con * C(5 + 2 * i),$$

where $saxpy\_con$ 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 7 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 block of array C in PE_p ($[bot_l(C, p, k) : top_l(C, p, k)]$) will intersect to at most one referenced block of array A in PE_q ($[bot_f(A, q, j) : top_f(A, q, j) : s_2]$), and *vice versa*. Therefore, certain optimization can be taken by using two-nested closed forms to represent $send_C(p, q)$ and $recv_C(p, q)$ instead of using the mentioned formulas which use three-nested closed forms to represent the above two sets: $send_C(p, q)$ and $recv_C(p, q)$. In addition, 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

b_2	b_1	3	9	63	315	945	3780	7560	15120
	2 PE	0.876	1.241	0.879	0.829	0.821	0.821	0.821	0.821
2	4 PE	0.438	0.412	0.476	0.428	0.420	0.417	0.417	0.417
	8 PE	0.220	0.207	0.274	0.225	0.217	0.214	0.213	0.213
	16 PE	0.110	0.108	0.173	0.125	0.117	0.113	0.113	0.295
	2 PE	1.238	0.541	0.564	0.516	0.508	0.508	0.506	0.506
6	4 PE	0.407	0.271	0.318	0.270	0.262	0.259	0.259	0.258
	8 PE	0.205	0.136	0.125	0.145	0.137	0.134	0.134	0.133
	16 PE	0.106	0.068	0.065	0.085	0.077	0.074	0.073	0.134
	2 PE	0.875	0.582	0.349	0.380	0.371	0.369	0.368	0.367
42	4 PE	0.472	0.326	0.175	0.199	0.193	0.189	0.189	0.189
	8 PE	0.270	0.125	0.088	0.089	0.102	0.099	0.099	0.098
	16 PE	0.172	0.065	0.045	0.047	0.051	0.057	0.056	0.064
	2 PE	0.826	0.534	0.395	0.309	0.347	0.349	0.349	0.348
210	4 PE	0.426	0.278	0.209	0.169	0.161	0.179	0.179	0.179
	8 PE	0.223	0.149	0.089	0.085	0.084	0.094	0.094	0.094
	16 PE	0.124	0.087	0.047	0.043	0.044	0.048	0.054	0.055
	2 PE	0.819	0.527	0.387	0.357	0.297	0.344	0.345	0.345
630	4 PE	0.419	0.271	0.201	0.162	0.184	0.177	0.177	0.177
	8 PE	0.216	0.141	0.107	0.084	0.093	0.083	0.093	0.093
	16 PE	0.116	0.079	0.052	0.044	0.047	0.044	0.046	0.053
	2 PE	0.819	0.525	0.384	0.363	0.359	0.307	0.336	0.337
2520	4 PE	0.417	0.268	0.198	0.188	0.187	0.155	0.160	0.175
	8 PE	0.213	0.139	0.103	0.099	0.084	0.078	0.082	0.083
	16 PE	0.113	0.076	0.059	0.049	0.044	0.040	0.043	0.044
	2 PE	0.818	0.524	0.383	0.362	0.358	0.361	0.299	0.335
5040	4 PE	0.416	0.268	0.197	0.187	0.185	0.163	0.151	0.160
	8 PE	0.213	0.138	0.103	0.098	0.099	0.083	0.076	0.085
	16 PE	0.113	0.076	0.058	0.056	0.046	0.042	0.039	0.045
	2 PE	0.817	0.522	0.380	0.361	0.357	0.358	0.358	0.295
10080	4 PE	0.415	0.268	0.196	0.186	0.184	0.187	0.164	0.149
	8 PE	0.212	0.138	0.102	0.097	0.097	0.084	0.086	0.075
	16 PE	0.293	0.135	0.066	0.057	0.056	0.044	0.044	0.038

Table 7: Execution time (second) of computing the saxpy operation in Example 6 using 2 PEs, 4 PEs, 8 PEs, and 16 PEs, respectively. Array A is distributed by a $cyclic(b_1)$ distribution; array C is distributed by a $cyclic(b_2)$ distribution.

cases when block sizes are large, because in these cases, the indexing overhead for packing data messages is not significant.

4. All cases except three show scalable improvements when the number of PEs grows. Three exception cases are when the number of PEs is 16, $b_1 = 15120$ and $b_2 = 2$ or $b_2 = 6$, and $b_1 = 3$ and $b_2 = 10080$. This is because in these 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 Figure 10, which is drawn based on Table 7, we can summarize that it is preferable to choose block sizes $b_1 \geq 63$ and $b_2 \geq 42$ for this saxpy operation. □

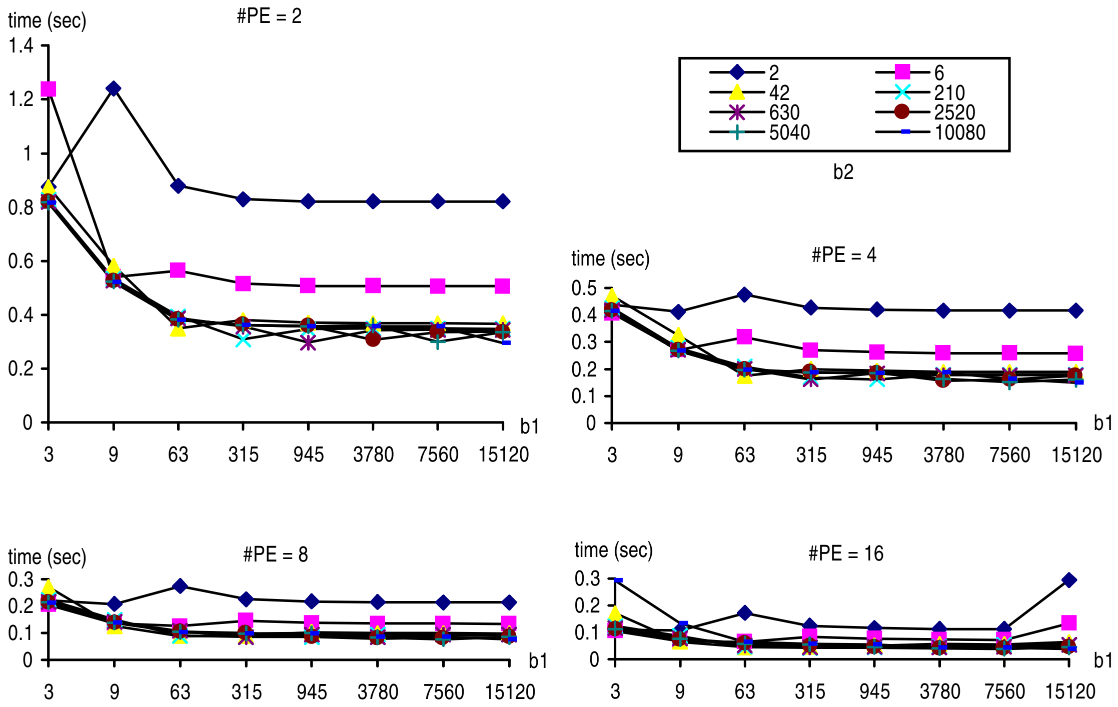

Figure 10: Execution time (second) of performing the saxpy operation in Example 6 using 2 PEs; 4 PEs; 8 PEs; and 16 PEs, respectively. Array A is distributed by a $cyclic(b_1)$ distribution; array C is distributed by a $cyclic(b_2)$ distribution.

Example 7: Consider the following data redistribution operation:

$$\textbf{doall}\ i = 0,\ 241919$$
$$A(i) = OLD\_A(i),$$

where array A is distributed by a $cyclic(b_1)$ distribution; array OLD_A is distributed by a $cyclic(b_2)$ distribution. Table 8 lists experimental results of implementing this data redistribution operation with various block sizes b_1 and b_2. Experimental results show that the behavior of the execution time of this data redistribution operation is similar to that of the saxpy operation. From Figure 11, which is drawn based on Table 8, we can summarize that it is preferable to choose block sizes $b_1 \geq 63$ and $b_2 \geq 63$ for this data redistribution operation. $\square$

b_2	b_1	3	9	63	315	945	3780	7560	15120
3	2 PE	0.000	1.652	1.292	1.244	1.236	1.233	1.233	1.233
	4 PE	0.000	0.627	0.688	0.639	0.631	0.630	0.628	0.629
	8 PE	0.000	0.322	0.378	0.331	0.323	0.320	0.319	0.319
	16 PE	0.000	0.170	0.225	0.177	0.169	0.166	0.165	0.165
9	2 PE	1.660	0.000	0.930	0.892	0.885	0.883	0.882	0.882
	4 PE	0.642	0.000	0.505	0.462	0.455	0.452	0.452	0.452
	8 PE	0.331	0.000	0.223	0.241	0.234	0.232	0.231	0.231
	16 PE	0.173	0.000	0.115	0.132	0.124	0.121	0.121	0.121
63	2 PE	1.299	0.938	0.000	0.712	0.714	0.716	0.716	0.716
	4 PE	0.692	0.509	0.000	0.366	0.369	0.369	0.368	0.368
	8 PE	0.382	0.228	0.000	0.181	0.190	0.189	0.189	0.189
	16 PE	0.227	0.118	0.000	0.090	0.099	0.100	0.100	0.100
315	2 PE	1.251	0.898	0.718	0.000	0.673	0.692	0.693	0.694
	4 PE	0.644	0.464	0.371	0.000	0.343	0.355	0.356	0.357
	8 PE	0.333	0.243	0.186	0.000	0.182	0.182	0.182	0.183
	16 PE	0.178	0.132	0.096	0.000	0.107	0.093	0.096	0.097
945	2 PE	1.243	0.891	0.721	0.678	0.000	0.680	0.686	0.688
	4 PE	0.636	0.458	0.372	0.353	0.000	0.345	0.352	0.353
	8 PE	0.325	0.236	0.192	0.185	0.000	0.178	0.178	0.180
	16 PE	0.170	0.124	0.099	0.099	0.000	0.112	0.092	0.094
3780	2 PE	1.240	0.888	0.723	0.699	0.688	0.000	0.668	0.679
	4 PE	0.633	0.455	0.372	0.359	0.352	0.000	0.367	0.346
	8 PE	0.322	0.233	0.191	0.184	0.183	0.000	0.240	0.178
	16 PE	0.167	0.122	0.101	0.096	0.096	0.000	0.093	0.111
7560	2 PE	1.240	0.888	0.723	0.700	0.693	0.677	0.000	0.667
	4 PE	0.633	0.455	0.372	0.360	0.355	0.358	0.000	0.366
	8 PE	0.322	0.232	0.191	0.184	0.181	0.203	0.000	0.239
	16 PE	0.167	0.122	0.101	0.097	0.094	0.103	0.000	0.119
15120	2 PE	1.239	0.887	0.722	0.701	0.695	0.686	0.675	0.000
	4 PE	0.632	0.454	0.371	0.361	0.357	0.350	0.356	0.000
	8 PE	0.321	0.232	0.190	0.185	0.183	0.182	0.205	0.000
	16 PE	0.167	0.122	0.101	0.098	0.096	0.094	0.103	0.000

Table 8: Execution time (second) of performing the data redistribution operation in Example 7 using 2 PEs, 4 PEs, 8 PEs, and 16 PEs, respectively. Array A is distributed by a $cyclic(b_1)$ distribution; array OLD_A is distributed by a $cyclic(b_2)$ distribution.

In the above two experimental studies, we assumed that the problem variables and the number of PEs were given at run time. Therefore, each node had to compute all boundary indices of closed forms at run time. In practice, for many applications, problem variables and the number of PEs are known at compiling time. Then, boundary indices of closed forms can be computed in advance at compiling time, and the resulting execution time can thus be even better as expectation.

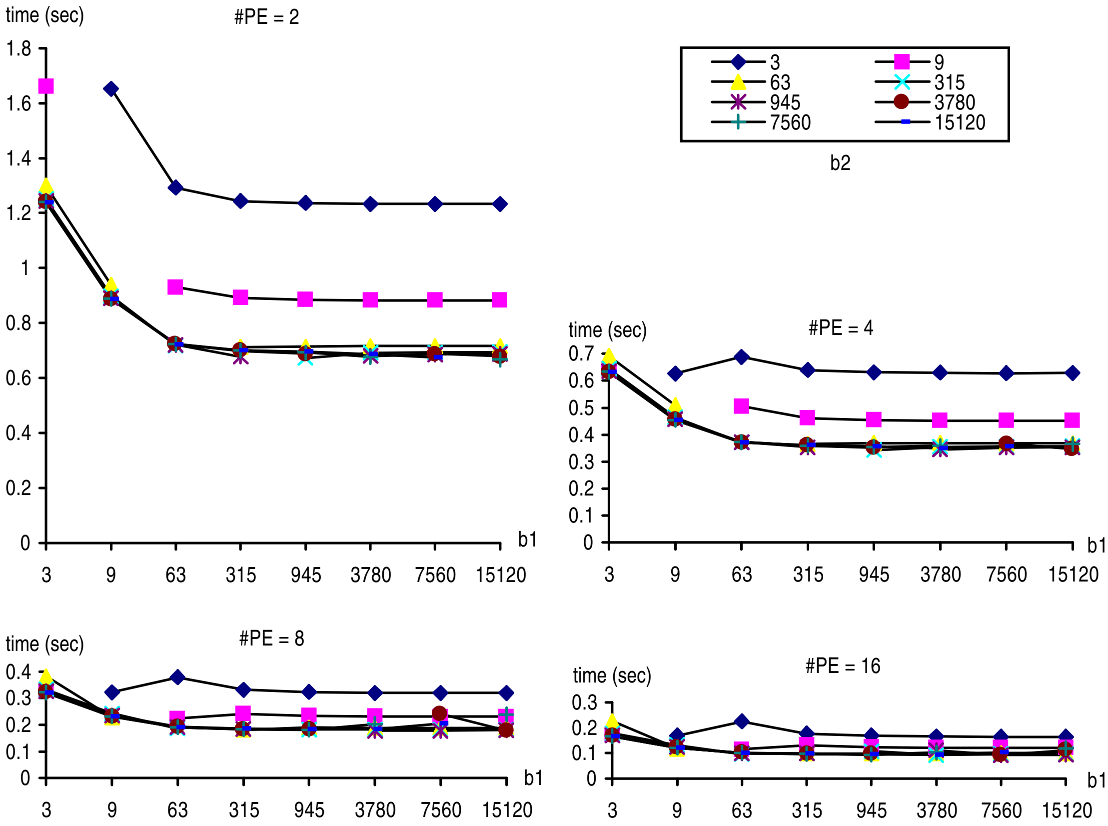

Figure 11: Execution time (second) of performing the data redistribution operation in Example 7 using 2 PEs; 4 PEs; 8 PEs; and 16 PEs, respectively. Array A is distributed by a $cyclic(b_1)$ distribution; array OLD_A is distributed by a $cyclic(b_2)$ distribution.

6 Conclusions

We have presented in this paper several techniques for determining data distribution and generating communication sets on distributed memory multicomputers. In Section 3, 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.

In Section 4 and Section 5, we derived formulas to represent communication sets of executing single-loop doall statements. In Section 4, we found that there were no simple formulas to represent communication sets when data arrays were distributed arbitrarily. However, in Section 5, 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 PEs increased. We have studied doall statements with one-dimensional arrays in this paper. If alignments and distributions of each dimension in a multi-dimensional array are independent of one another, the extension of our approach to multi-dimensional arrays is straightforward.

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] 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.

[3] 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. *Journal of Parallel and Distributed Computing*, 26:72–84, 1995.

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

[5] 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.

[6] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C. H. Huang, and P. Sadayappan. On the generation of efficient data communication for distributed-memory machines. In *Proc. International Computer Symposium*, pages 504–513, Taichung, Taiwan, Dec. 1992.

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

[8] 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.

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

[10] E. T. Kalns and L. M. Ni. Processor mapping techniques toward efficient data redistribution. *IEEE Trans. Parallel Distributed Syst.*, 1995, to appear.

[11] S. D. Kaushik, C. H. Huang, R. W. Johnson, and P. Sadayappan. An approach to communication-efficient data redistribution. In *Proc. of ACM International Conf. on Supercomputing*, pages 364–373, Manchester, U.K., July 1994.

[12] S. D. Kaushik, C. H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase array redistribution: Modeling and evaluation. Technical Report OSU-CISRC-9/94-52, Department of Computer and Information Science, The Ohio State University, 1994.

[13] K. Kennedy, N. Nedeljkoviá, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proc. of ACM International Conf. on Supercomputing*, pages 180–184, Barcelona, Spain, July 1995.

[14] K. Kennedy, N. Nedeljkoviá, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proc. ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, Santa Barbara, CA, July 1995.

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

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

[17] 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.

[18] 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.

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

[20] 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.

[21] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, 1989.

[22] 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.

[23] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF program. In *Proc. of Scalable High Performance Computing Conference*, pages 309–316, May 1994.

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

[25] 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.