Generating Global Name-Space Communication Sets for Array Assignment Statements[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 the design of efficient algorithms for generating global name-space communication sets based on execution of array assignment statements on distributed-memory parallel computers. For general cases, although the communication sets can be represented by the union of a functional number of closed forms, these sets cannot be represented by a fixed number of closed forms. Closed-form expressions for communication sets would reduce the associated packing overhead at the sending processor and unpacking overhead at the receiving processor. In this paper, we will first present a method using row-wise block-to-block intersections and an integer lattice method to generate communication sets when data arrays are distributed in any arbitrary *block-cyclic* fashion. After that, we will show that compiler or run-time support itself is more suitable for determining the block sizes of the array distributions. We will also derive closed forms to represent communication sets when data arrays are distributed in a restricted *block-cyclic* fashion, which can be determined at compiling time. Our methods can be included in current compilers and used when programmers don't know how to use data distribution directives to assign suitable block sizes. Experimental studies on a 16-node nCUBE/2E parallel computer are also presented.

Keywords: array assignment statements, closed forms, communication sets, distributed-memory computers, forall statements, global name space, parallelizing compilers, run-time support.

[1]This work was partially supported by the NSC under Grants NSC 86-2213-E-001-006 and NSC 87-2213-E-001-005.

1 Introduction

This paper is concerned with the design of efficient algorithms for generating *global name-space* communication sets based on execution of *array assignment statements* on distributed-memory parallel computers. Data-parallel languages which adopt a (single) global name space allow programmers to express their algorithms as is done on a shared memory architecture [25]. Array assignment statements are used to express data-parallelism in scientific languages such as Fortran 90D/HPF [5], Fortran D [15] and High Performance Fortran (HPF) [24]. Because an array assignment statement is equivalent to a special form of a *forall* statement as shown in Table 1, different iterations (loop bodies) can be executed independently. Since data arrays are distributed among processing elements (PEs) in some fashion, according to the *owner computes rule*: *the owner of the left-hand side element executes the assignment for that element*, compiler or run-time support can group different sets of iterations into PEs, and PEs can execute their corresponding set of iterations independently. However, compiler or run-time support has to provide efficient algorithms for generating communication sets if the generated data, which are on the left-hand side (LHS) of the assignment statement, are not stored in the same PE as the used data, which are on the right-hand side (RHS) of the assignment statement. Otherwise, the performance gain due to parallel computing will be degraded by software overhead.

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

Table 1: An array assignment statement is equivalent to a special form of a forall statement, where g is a function of array C.

In this paper, we are interested in generating all the necessary communication sets in each PE when an array assignment statement is executed on a distributed-memory machine. Let $cyclic(b)$ distribution be the most general regular distribution in which blocks of size b of the array are distributed among PEs in a round-robin fashion. In the following, we will state the problem we want to solve in this paper. For convenience, throughout this paper, we will use forall statements to represent array assignment statements without confusion.

Problem: In a distributed-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 the necessary communication sets in each processor due to execution of the array assignment statement $A(l_1 : u_1 : s_1) = g(C(l_2 : u_2 : s_2))$, which is equivalent to the following forall statement, where $s_1 > 0$, $s_2 > 0$, and g is a function:

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

The case where s_1 or s_2 is negative can be treated analogously. The degenerate case where $s_1 = 0$ (reduction) or $s_2 = 0$ (broadcast) can be handled by other optimization method. For general cases where b_1, b_2, s_1 and s_2 are arbitrary numbers, although the communication sets due to execution of forall statements in each PE can be represented by the union of a functional number of closed forms, these sets cannot be represented by a fixed number of closed forms. For these cases, we will present an efficient algorithm based on row-wise block-to-block intersections and an integer lattice method to generate communication sets.

While $cyclic(b)$ (*block-cyclic*) distributions are important from an algorithmic standpoint [9, 26], the complicated arithmetical formulations of communication sets which result in the difficulty of efficiently compiling for such distributions has delayed the inclusion of this feature in commercial HPF compilers. Indeed, there has ever been some discussion of removing $cyclic(b)$ distribution from HPF altogether [46]. This is all because communication sets cannot be represented by a fixed number of closed forms for the general cases.

However, we believe that block sizes b_1 and b_2 should be determined by compilers, and that programmers only need to concentrate on implementing their sequential programs. The way to determine data alignment and data distribution can be implemented in compilers [11, 27, 28, 31]. The way to choose the grain and granularity of a block size b for a specific array distribution also can be determined by an analytical model [29] or by certain experienced data distributions from a knowledge base [2]. The following two oracles help decide the block size b. The load balance oracle suggests use of $cyclic$ ($cyclic(1)$) distribution if the iteration space is a pyramid (such as the iteration space of an LU decomposition), a triangle (such as the iteration space of a triangular linear system), or any other non-rectangular space. The communication oracle emphasizes not making the block size too small if the computation in each iteration involves shift operations or if data of each array element depend on data of neighboring array elements; otherwise, it will incur a high communication overhead, a high buffering overhead and a high indexing overhead. These two oracles, unfortunately, are inconsistent.

For instance, Table 2 shows comparisons of using different block sizes to execute a five-stencil problem with a triangular iteration space on a linear processor array, where the problem size $m = 2^{11}$ and the number of PEs $N = 16$ or $m = 2^{20}$ and $N = 64$. Suppose that arrays A and C are both distributed along rows by $cyclic(b)$. In an analytical model, we can formulate the total execution time from the SPMD (Single Program Multiple Data) program which includes both the computation time and the communication time. The total execution time T is a function of the problem size m, the number of PEs N, and the block size b. When the problem size m and the number of PEs N are fixed, the optimal execution time can be obtained by requiring that $\frac{\partial T}{\partial b} = 0$ or by substituting all possible b into the formula. Alternatively, from experience, choosing a block size $b = \frac{m}{N*2^5}$ or $b = \frac{m}{N*2^4}$ is also an acceptable compromise for both load balance and communication overhead. Because the cost of data re-distribution is high, in practice, block sizes are chosen not only for one statement but also for a segment of a program, which includes a lot of statements. Thus, block sizes should be a compromise for many statements. Therefore, it seems suitable to choose block sizes ranging from $\frac{m}{N*2^5}$ or $\frac{m}{N*2^4}$ for a non-rectangular iteration space to $\frac{m}{N}$ for a rectangular iteration space.

We now continue to state the problem. If b_1 is close to b_1' and b_2 is close to b_2', then the difference due to the load balance requirement between using ($cyclic(b_1)$ and $cyclic(b_2)$) and using ($cyclic(b_1')$ and $cyclic(b_2')$) is not significant, but the difference due to the software overhead incurred in generating communication sets may be significant. When strides s_1 and s_2 are given, we will show how block sizes b_1 and b_2 can be obtained, such that communication sets can be represented by closed forms. Closed-form expressions for communication sets would reduce the associated packing overhead at the sending PE and unpacking overhead at the receiving PE.

This paper is a continuation of our earlier work on compiling high-level languages to distributed-memory parallel computers. The trend of currently parallelizing compiler research has emphasized allowing programmers to specify the data distribution using language extensions, such that compilers can then generate all the communication instructions according to these language extensions [5, 8, 15, 24]. For instance, in HPF, programmers have the obligation to provide *TEMPLATE*, *ALIGN*, and *DISTRIBUTE* directives to specify data distribution. However, in order to use these three directives

(do $i=1,\ m$) (do $j=1,\ i$) $A(i,j) = \alpha * (C(i,j) + C(i-1,j) + C(i+1,j) + C(i,j-1) + C(i,j+1))$;
(do $i=1,\ m$) (do $j=1,\ i$) $C(i,j) = \beta * (A(i,j) + A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))$;

(a) A five-stencil program with a triangular iteration space.

$diff = \frac{2mb(N-1)}{N}$, which means the difference in the work load between PE_0 and PE_{N-1};
$ratio = \frac{m+bN-b+1}{m-bN+b+1}$, which means the ratio of the work load between PE_0 and PE_{N-1};
$comm = 2m(\frac{m}{bN} + 1 - \frac{1}{N} + \frac{1}{bN})$, which means the communication overhead in PE_0.

b	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
$diff$	$1.88 \cdot 2^{11}$	$1.88 \cdot 2^{12}$	$1.88 \cdot 2^{13}$	$1.88 \cdot 2^{14}$	$1.88 \cdot 2^{15}$	$1.88 \cdot 2^{16}$	$1.88 \cdot 2^{17}$	$1.88 \cdot 2^{18}$
$ratio$	1.01	1.03	1.06	1.12	1.27	1.61	2.76	30.77
$comm$	$1.01 \cdot 2^{19}$	$1.02 \cdot 2^{18}$	$1.03 \cdot 2^{17}$	$1.06 \cdot 2^{16}$	$1.12 \cdot 2^{15}$	$1.23 \cdot 2^{14}$	$1.47 \cdot 2^{13}$	$1.94 \cdot 2^{12}$

(b) The case where $m = 2^{11}$ and $N = 2^4$.

b	2^0	2^1	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}
$diff$	$1.97 \cdot 2^{20}$	$1.97 \cdot 2^{21}$	$1.97 \cdot 2^{29}$	$1.97 \cdot 2^{30}$	$1.97 \cdot 2^{31}$	$1.97 \cdot 2^{32}$	$1.97 \cdot 2^{33}$	$1.97 \cdot 2^{34}$
$ratio$	1.00	1.00	1.06	1.13	1.28	1.65	2.94	126.99
$comm$	$1.00 \cdot 2^{35}$	$1.00 \cdot 2^{34}$	$1.03 \cdot 2^{26}$	$1.06 \cdot 2^{25}$	$1.12 \cdot 2^{24}$	$1.25 \cdot 2^{23}$	$1.49 \cdot 2^{22}$	$1.98 \cdot 2^{21}$

(c) The case where $m = 2^{20}$ and $N = 2^6$.

Table 2: Comparisons of executing a five-stencil problem using different block sizes.

efficiently, programmers have to know both architectures used and possible parallelism in the program in advance. Unfortunately, many programmers maybe don't know how to use these three directives to assign suitable data distributions for the whole program because users of such multiprocessor systems generally are non-computer scientists, who seek the maximum possible performance of their applications but don't want to be involved in the parallelization process. In [28], we showed that it is possible to use compiler techniques to automatically determine data alignment and dynamic data distributions of sequential programs on distributed-memory systems. In this paper, we will further show that compiler or run-time support itself is more suitable for determining block sizes of array distributions.

The rest of this paper is organized as follows. In Section 2, we define notations which will be used later. In Section 3, we derive formulas to represent communication sets with arbitrary block sizes b_i. In Section 4, we present an integer lattice method to generate communication sets also with arbitrary block sizes b_i. In Section 5, we propose algorithms to determine block sizes b_i while giving strides s_i, and we also derive closed forms to represent communication sets with these restricted block sizes. In Section 6, experimental studies on a 16-node nCUBE/2E parallel computer are presented. Section 7 discusses related work in this area and illustrates that based on the two-level mapping model, there has no closed-form expressions for communication sets for arbitrary strides s_1 and s_2. Finally, some concluding remarks are given in Section 8.

2 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, $[1 : 102] = \{1, 2, 3, \cdots, 102\}$.
- $[a : e_1 : s_1]$ means the set of integers from a with stride (period) s_1 until 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 that 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 in a *cyclic* fashion if $b = 1$; in a *block* fashion if $b = \lceil (a_2 - a_1 + 1)/N \rceil$; and in a *block-cyclic* fashion if $1 < b < \lceil (a_2 - a_1 + 1)/N \rceil$.

The function $nxt(x, y, z)$ which we use here is the smallest integer greater than x and is congruent with y modulo z; that is, $nxt(x, y, z) = x + ((y - x) \bmod z)$.

3 Generation of Communication Sets for Array Assignments

We will now analyze the problem. Let $f_k(i) = l_k + i * s_k$, and let the inverse functions $f_k^{-1}(l_k + i * s_k) = i$, for $k = 1$ or 2.

3.1 Structure of Generated Code

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 $send\_pe(p) = \{q \mid q \neq p$ and PE_p will send some data to $PE_q\}$;
  1.3 $recv\_pe(p) = \{q \mid q \neq p$ and PE_p will receive some data from $PE_q\}$;

2. $\forall\ q \in send\_pe(p)$, **do**
  2.1 $send_C(p, q) = local_C(p) \cap f_2(exec(q))$, which represents elements sent from PE_p to PE_q, 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 $iter(p, p) = f_2^{-1}(local_C(p) \cap [l_2 : u_2 : s_2]) \cap exec(p) = f_2^{-1}(send_C(p, p)) = f_2^{-1}(recv_C(p, p))$, which stands for iterations on PE_p that access only local data;

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 1: Outline of implementing an array assignment statement.

Fig. 1 shows a detailed outline of the implementation of an array assignment statement (forall statement) in each PE, which is a generalization based on formulas presented in [23]. Step 1 of Fig. 1 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 messages 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 to derive other communication sets and processor sets. Since $exec(p) = iter(p,p) \cup (\bigcup_{q \in recv\_pe(p)} iter(p,q))$ and $iter(p,q) = f_2^{-1}(recv_C(p,q))$, we can combine Substep 1.1 and Step 3 as well as 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 be calculated. It is also instructive to point out that, in order to gain efficiency by allowing overlapping execution, we have arranged communication and computation tasks in an interleaved manner.

3.2 Derivation of Communication Sets

We now 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 that $(c_2 - c_1 + 1)$ is a multiple of Nb_2. Since array A adopts $cyclic(b_1)$ distribution, $local_A(p) = [[a_1 + pb_1 : a_1 + pb_1 + b_1 - 1] : a_2 : Nb_1]$. Since array C adopts $cyclic(b_2)$ distribution, $local_C(p) = [[c_1 + pb_2 : c_1 + pb_2 + b_2 - 1] : c_2 : Nb_2]$. We also assume that $(u_1 - l_1)$ is a multiple of s_1, and that $u_2 = l_2 + ((u_1 - l_1)/s_1) * s_2$. In Table 3, we introduce some notations which will be used later. The function name $'bot'$ means the first element in a block; $'top'$ means the last element in a block. The triple $'(A,p,j)'$ means the jth block data of array A in PE_p. The subscript $'l'$ means **l**ocal data; $'a'$ means **a**ccessed data; $'e'$ means iterations to be **e**xecuted; and $'f'$ means the corresponding re**f**erenced data between array A and array C.

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 let 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. Fig. 2 shows an algorithm for computing j_{pf} and j_{pl}. k_{pf} and k_{pl} also can be computed similarly. In Fig. 2, 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. Because the access pattern of $A(l_1 : u_1 : s_1)$ in PE_p appears periodically, the worst case complexity of computing j_{pf} and j_{pl} in Fig. 2 is $O(s_1/\gcd(Nb_1, s_1))$. Alternatively, in Section 4.2.4, we will give another algorithm for computing the first element of $A(l_1 : u_1 : s_1)$ stored in PE_p based on solving $O(b_1/\gcd(Nb_1, s_1))$ linear Diophantine equations. According to our experiments, in a majority of cases, the algorithm in Fig. 2 was more efficient than was solving $O(b_1/\gcd(Nb_1, s_1))$ linear Diophantine equations.

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

Table 3: Notations which will be used to derive communication sets.

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

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

Note that, in the expression $[bot_e(A,p,j) : top_e(A,p,j)]$, it may happen 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 Fig. 1, after deriving $exec(p)$, we should present the processor sets $send\_pe(p)$ and $recv\_pe(p)$. However, since exact solutions of these two sets are tedious, we prefer to present the communication sets $send_C(p,q)$ and $recv_C(p,q)$ first.

```
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 2: An algorithm for computing j_{pf} and j_{pl}.

3.2.1 Derivation of $send_C(p, q)$ and $recv_C(p, q)$

We now introduce a set $f_2(exec(q))$, which will be used in deriving $send_C(p, q)$ $(= local_C(p) \cap f_2(exec(q)))$:

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

We now define the periodic coefficients of the communication set $send_C(p, q)$. Let $period_e^A$ be the period of the iteration pattern in $exec(p)$ such that $period_e^A * s_1$ is a multiple of Nb_1; let $period_{eb}^A$ be the number of blocks of local elements of array A whose access pattern appears periodically; let $period_{eb}^C$ be the number of blocks of local elements of array C whose access pattern appears periodically; 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; let $period_{sb}^C$ be the number of blocks of local elements of array C whose reference pattern in $send_C(p, q)$ appears periodically; and let $period_{sb}^A$ 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:

$$period_e^A = (\mathrm{lcm}(Nb_1,\ s_1))/s_1; \qquad period_s = \mathrm{lcm}(Nb_2,\ period_e^A * s_2);$$
$$period_{eb}^A = (\mathrm{lcm}(Nb_1,\ s_1))/(Nb_1); \qquad period_{sb}^C = period_s/(Nb_2);$$
$$period_{eb}^C = (\mathrm{lcm}(Nb_2,\ s_2))/(Nb_2); \qquad period_{sb}^A = (period_s * s_1)/(Nb_1 s_2).$$

We will 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 $\lceil \frac{b_1}{s_1} \rceil \le \lceil \frac{(N-1)b_2+1}{s_2} \rceil$, then each referenced block of array A in PE_q ($[bot_f(A,q,j) : top_f(A,q,j) : s_2]$) will intersect with at most one local block of array C in PE_p ($[bot_l(C,p,k) : top_l(C,p,k)]$). Similarly, if $\lceil \frac{b_2}{s_2} \rceil \le \lceil \frac{(N-1)b_1+1}{s_1} \rceil$, then each local block of array C in PE_p will also intersect with at most one referenced block of array A in PE_q. The following two properties are used to generate $send_C(p,q)$ and $recv_C(p,q)$.

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

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 $\lceil \frac{(N-1)b_1+1}{s_1} \rceil \ge \lceil \frac{b_1}{s_1} \rceil > \lceil \frac{(N-1)b_2+1}{s_2} \rceil \ge \lceil \frac{b_2}{s_2} \rceil$. 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 $\lceil \frac{\beta-\alpha+1}{\gamma} \rceil \le \lceil \frac{(N-1)b+1}{\gamma} \rceil$. 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 $\lceil\frac{\beta-\alpha+1}{\gamma}\rceil \le \lceil\frac{(N-1)b+1}{\gamma}\rceil$, $[\alpha:\beta:\gamma]$ will intersect with 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 the union of a functional number of closed forms. First, if $\lceil\frac{b_1}{s_1}\rceil \le \lceil\frac{(N-1)b_2+1}{s_2}\rceil$, $send_C(p,q)$ can be represented as follows:

$$
\begin{aligned}
send_C(p,q) &= local_C(p)\cap f_2(exec(q))\\
&= [[c_1+pb_2\ :\ c_1+pb_2+b_2-1]\ :\ c_2\ :\ Nb_2]\cap\left(\bigcup_{j=j_{qf}}^{j_{ql}}[bot_f(A,q,j):top_f(A,q,j):s_2]\right)\\
&= \bigcup_{j=j_{qf}}^{j_{ql}}\left([[c_1+pb_2\ :\ c_1+pb_2+b_2-1]\ :\ c_2\ :\ Nb_2]\cap[bot_f(A,q,j):top_f(A,q,j):s_2]\right)\\
&= \bigcup_{j=j_{qf}}^{j_{ql}}[L(j):R(j):s_2]\\
&= [L(j_{qf}):R(j_{qf}):s_2]\cup\left(\bigcup_{j=j_{qf}+1}^{\min\{j_{ql},j_{qf}+period_{sb}^A\}}[[L(j):R(j):s_2]:u_2:period_s]\right)\\
&= [L(j_{qf}):R(j_{qf}):s_2]\cup[\left(\bigcup_{j=j_{qf}+1}^{\min\{j_{ql},j_{qf}+period_{sb}^A\}}[L(j):R(j):s_2]\right):u_2:period_s],
\end{aligned}
$$

where

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

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:

$$
\begin{aligned}
&send_C(p,q) = f_2(exec(q))\cap local_C(p)\\
=\ & f_2f_1^{-1}\left(f_1f_2^{-1}(f_2(exec(q))\cap local_C(p))\right)\\
=\ & f_2f_1^{-1}\left([[a_1+qb_1\ :\ a_1+qb_1+b_1-1]\ :\ a_2\ :\ Nb_1]\cap\left(\bigcup_{k=k_{pf}}^{k_{pl}}[bot_f(C,p,k):top_f(C,p,k):s_1]\right)\right)\\
=\ & \bigcup_{k=k_{pf}}^{k_{pl}} f_2f_1^{-1}\left([[a_1+qb_1\ :\ a_1+qb_1+b_1-1]\ :\ a_2\ :\ Nb_1]\cap[bot_f(C,p,k):top_f(C,p,k):s_1]\right)\\
=\ & \bigcup_{k=k_{pf}}^{k_{pl}}[f_2f_1^{-1}(L(k)):f_2f_1^{-1}(R(k)):s_2]\\
=\ & [f_2f_1^{-1}(L(k_{pf})):f_2f_1^{-1}(R(k_{pf})):s_2]\cup\\
&\qquad\left(\bigcup_{k=k_{pf}+1}^{\min\{k_{pl},k_{pf}+period_{sb}^C\}}[[f_2f_1^{-1}(L(k)):f_2f_1^{-1}(R(k)):s_2]:u_2:period_s]\right)\\
=\ & [f_2f_1^{-1}(L(k_{pf})):f_2f_1^{-1}(R(k_{pf})):s_2]\cup\\
&\qquad[\left(\bigcup_{k=k_{pf}+1}^{\min\{k_{pl},k_{pf}+period_{sb}^C\}}[f_2f_1^{-1}(L(k)):f_2f_1^{-1}(R(k)):s_2]\right):u_2:period_s],
\end{aligned}
$$

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 deal with $recv_C(p,q)$. Because $recv_C(p,q)$ is equal to $send_C(q,p)$, $recv_C(p,q)$ also can be represented by the union of a functional number of closed forms. Although $recv_C(p,q)$ specifies a set of indices of array C, in practice, we prefer that $recv_C(p,q)$ be represented based on indices of array A. For instance, the loop body of the forall 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 forall 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 $\lceil \frac{b_1}{s_1} \rceil \le \lceil \frac{(N-1)b_2+1}{s_2} \rceil$, $recv_C(p,q)$ can be represented as follows:

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

where

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

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

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

$$\begin{aligned} recv_C(p,q) &= f_2 f_1^{-1}(f_1 f_2^{-1}(recv_C(p,q))) = f_2 f_1^{-1}(f_1 f_2^{-1}(send_C(q,p))) \\ &= f_2 f_1^{-1}\Big([L(k_{qf}) : R(k_{qf}) : s_1] \cup \end{aligned}$$

$$\Big(\bigcup_{k=k_{qf}+1}^{\min\{k_{ql},k_{qf}+period_{sb}^C\}}[[L(k):R(k):s_1]:u_1:period_s*s_1/s_2]\Big)\Big)$$

$$= \; f_2 f_1^{-1}\Big([L(k_{qf}):R(k_{qf}):s_1] \cup [\Big(\bigcup_{k=k_{qf}+1}^{\min\{k_{ql},k_{qf}+period_{sb}^C\}}[L(k):R(k):s_1]\Big):u_1:period_s*s_1/s_2]\Big),$$

where

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

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

3.2.2 Derivation of $send\_pe(p)$ and $recv\_pe(p)$

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 testing whether q is in $send\_pe(p)$ or whether q is in $recv\_pe(p)$ is equivalent to testing whether $send_C(p,q) \neq \phi$ or whether $send_C(q,p) \neq \phi$, respectively. For this reason, we consider inexact solutions for $send\_pe(p)$ and $recv\_pe(p)$. The following property will be used to derive $send\_pe(p)$ and $recv\_pe(p)$.

Property 3 *Suppose that array A is distributed by cyclic*(b_1)*;* $f_A(i) = (\lfloor \frac{i-a_1}{b_1} \rfloor \bmod N)$*, 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*

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

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 also an equation only when $s_2 \leq b_2$.

4 Integer Lattice Method for Generating Communication Sets

In the last section, we derived communication sets which can be represented by the union of $(period^A_{sb}+1)$ or $(period^C_{sb}+1)$ closed forms. However, as one can see from a preliminary example in Figure 5, for many cases, $L(j) > R(j)$ for some $j \in [j_{qf}+1 : \min\{j_{ql}, j_{qf}+period^A_{sb}\}]$; therefore, $[[L(j) : R(j) : s_2] : u_2 : period_s]$ is an empty set. Similarly, for many cases, $L(k) > R(k)$ for some $k \in [k_{pf}+1 : \min\{k_{pl}, k_{pf}+period^C_{sb}\}]$; therefore, $[[f_2 f_1^{-1}(L(k)) : f_2 f_1^{-1}(R(k)) : s_2] : u_2 : period_s]$ is an empty set. In these cases, we actually need not compute $L(j)$, $R(j)$, $L(k)$, and $R(k)$. In the following, we present an integer lattice method, which adopts a variant of Kennedy, Nedeljković and Sethi's algorithm [21, 22] as a subroutine to generate communication sets.

4.1 A Result by Kennedy *et al.* and Its Variations

Let $A(a_1 : a_2)$ be an array distributed over N processing elements with $cyclic(b_1)$ distribution. Kennedy *et al.* treated each array element as a point (x, y) in Z^2 space [21, 22], such that the value x is the number of the row to which an index belongs, and the value y is its offset within that row. For instance, a one-dimensional array index i corresponds to a two-dimensional index (x, y) in processing element PE_p; then, $x = (i - a_1)/(Nb_1)$, $y = ((i - a_1) \bmod (Nb_1))$, and $p = (((i - a_1)/b_1) \bmod N)$. Figure 3 presents an example when $a_1 = 0$, $N = 4$, and $b_1 = 5$. Kennedy *et al.* show that regular section indices $A(l_1 : u_1 : s_1)$ within a processing element PE_p form a lattice which can be enumerated in increasing order by a specific pair of basis vectors $R_v = (a_r, b_r)$ and $L_v = (a_l, b_l)$ (assuming that stride s_1 is positive).

Vectors R_v and L_v can be found from the initial cycle of memory accesses in processing element PE_0 when $a_1 = 0$ and the lower bound l_1 is 0. Vector R_v is the distance between index 0 and the next smallest index accessed by PE_0; vector L_v is the distance between the largest index in the initial cycle and the index that starts the next cycle, both accessed by PE_0. For instance, in Figure 3-(a), $R_v = (0, 3)$ and $L_v = (1, -2)$; in Figure 3-(b), $R_v = (3, 3)$ and $L_v = (1, -2)$. They also have the following result.

Theorem 4 [21, 22] *Given an array element indexed by (x, y) that belongs to processing element PE_p,*

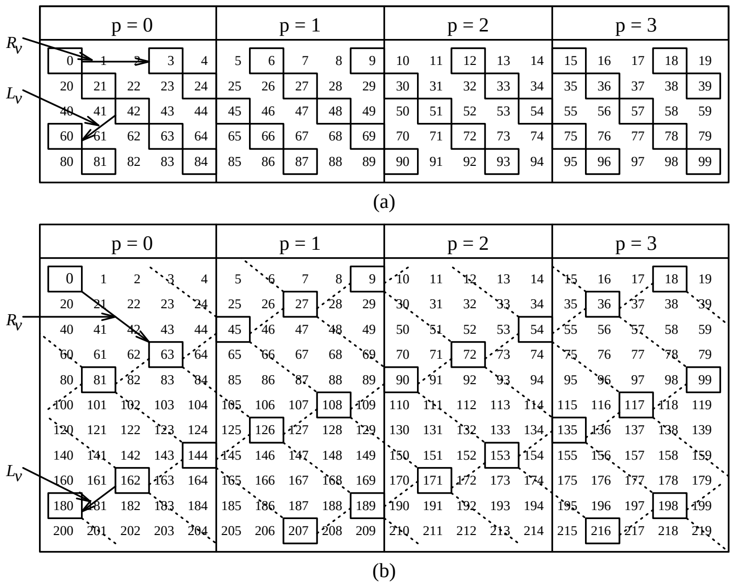


Figure 3: Array $A(0 : a_2)$ with $cyclic(5)$ distribution on 4 processing elements, in which a one-dimensional array index i of $A(i)$ in processing element $((i/5) \bmod 4)$ corresponds to a two-dimensional index $(i/20, i \bmod 20) \in Z^2$. (a) Rectangles mark elements $A(j * 3)$ for $j \in [0 : 33]$. In this case, stride $(s_1 = 3) \leq$ block size $(b_1 = 5)$. (b) Rectangles mark elements $A(j * 9)$ for $j \in [0 : 24]$. In this case, stride $(s_1 = 9) >$ block size $(b_1 = 5)$.

the next element accessed by the same processing element must have one of the following three distances:

$$\begin{array}{rl} R_v & \textit{if} \quad y + b_r \leq pb_1 + b_1 - 1; \\ L_v & \textit{if} \quad y + b_r > pb_1 + b_1 - 1 \quad \textit{and} \quad y + b_l \geq pb_1; \\ R_v + L_v & \textit{otherwise.} \quad \Box \end{array}$$

Because we need to generate global addresses in the global name space, we have to modify their algorithm, which only generates local addresses. We have found that it is enough to use two scales, R and L, to represent two distance vectors (basis vectors), R_v and L_v. For instance, in Figure 3-(a), $R = 3$ and $L = 18$; in Figure 3-(b), $R = 63$ and $L = 18$. Therefore, Theorem 4 can be modified into the following equivalent theorem.

Theorem 5 *Let an array $A(a_1 : a_2)$ be distributed over N processing elements with cyclic(b_1) distribution. Suppose that under the constraint that $a_1 = 0$, and that elements $A(0 : u_1 : s_1)$ are accessed, we let R be the distance between index 0 and the next smallest index accessed by PE_0; let L be the distance between the largest index in the initial cycle and the index that starts the next cycle, both accessed by PE_0. Then, for arbitrary a_1 and for an arbitrary access pattern $A(l_1 : u_1 : s_1)$, given an array element indexed by i that belongs to processing element PE_p, the next element accessed by the same processing* *element must have one of the following three distances:*

$$\begin{array}{rl} R & \textit{if} \;\; pb_1 \le go\_right \le pb_1 + b_1 - 1; \\ L & \textit{if} \;\; (\textit{not}\; (pb_1 \le go\_right \le pb_1 + b_1 - 1)) \;\; \textit{and} \;\; (pb_1 \le go\_left \le pb_1 + b_1 - 1); \\ R + L & \textit{otherwise}, \end{array}$$

where $go\_right = ((i - a_1 + R) \bmod (Nb_1))$*; and* $go\_left = ((i - a_1 + L) \bmod (Nb_1))$. □

Theorem 4 and Theorem 5 also can be applied to the following variant case, which we will use to derive communication sets. Suppose that an array $A(a_1 : a_2)$ is stored in a two-dimensional table according to a row-major rule; in addition, the size of the second dimension of the table is Nb_1. If we wrap-around connect the right boundary and the left boundary of the table so that elements $A(a_1 + xNb_1 - 1)$ are neighbors of elements $A(a_1 + xNb_1)$, then this table becomes a spiral cylinder. Figure 7 shows an example of how to wrap-around connect the left boundary and the right boundary when $a_1 = 0$ and $Nb_1 = 15$. On a spiral cylinder, between any two columns, Theorem 4 and Theorem 5 are also true.

Corollary 6 *Let an array* $A(a_1 : a_2)$ *be stored in a two-dimensional table according to a row-major rule; in addition, let the size of the second dimension of the table be* Nb_1*. Then, on a spiral cylinder, among the columns from* lb *to* rb*, the following two cases are true.*

(1) *Suppose that* $lb < rb$*. Then, the access pattern of* $A(l_1 : u_1 : s_1)$ *among the columns from* lb *to* rb *forms a lattice. Suppose again that, under the constraint that* $a_1 = 0$*, and that elements* $A(0 : u_1 : s_1)$ *are accessed, we let* R *be the distance between index 0 and the next smallest index accessed among the columns from 0 to* $rb - lb$*; let* L *be the distance between the largest index in the initial cycle and the index that starts the next cycle, both accessed among the columns from 0 to* $rb - lb$*. Then, for arbitrary* a_1 *and for an arbitrary access pattern* $A(l_1 : u_1 : s_1)$*, given an index* i *located among the columns from* lb *to* rb*, the next index accessed among the columns from* lb *to* rb *must have one of the following three distances:*

$$\begin{array}{rl} R & \textit{if} \;\; (lb \le go\_right \le rb); \\ L & \textit{if} \;\; (\textit{not}\; (lb \le go\_right \le rb)) \;\; \textit{and} \;\; (lb \le go\_left \le rb); \\ R + L & \textit{otherwise}, \end{array}$$

where $go\_right = ((i - a_1 + R) \bmod (Nb_1))$*; and* $go\_left = ((i - a_1 + L) \bmod (Nb_1))$.

(2) *Suppose that* $lb > rb$*. Then, the access pattern of* $A(l_1 : u_1 : s_1)$ *among the columns from* lb *to* $Nb_1 - 1$ *and 0 to* rb *forms a lattice. Suppose again that, under the constraint that* $a_1 = 0$*, and* *that elements $A(0 : u_1 : s_1)$ are accessed, we let R be the distance between index 0 and the next smallest index accessed among the columns from 0 to $Nb_1 + rb - lb$; let L be the distance between the largest index in the initial cycle and the index that starts the next cycle both accessed among the columns from 0 to $Nb_1 + rb - lb$. Then, for arbitrary a_1 and for an arbitrary access pattern $A(l_1 : u_1 : s_1)$, given an index i located among the columns from lb to $Nb_1 - 1$ and 0 to rb, the next index accessed among the columns from lb to $Nb_1 - 1$ and 0 to rb must have one of the following three distances:*

$$\begin{array}{rl} R & \textit{if} \;\; (\textit{not}\,(rb < go\_right < lb)); \\ L & \textit{if} \;\; (rb < go\_right < lb) \;\; \textit{and} \;\; (\textit{not}\,(rb < go\_left < lb)); \\ R + L & \textit{otherwise}, \end{array}$$

where $go\_right = ((i - a_1 + R) \bmod (Nb_1))$; and $go\_left = ((i - a_1 + L) \bmod (Nb_1))$. □

4.2 Algorithms for Calculating the Memory Access Sequence

In order to find a starting accessed element and two distance vectors, R_v and L_v, Kennedy *et al.* solved $2*(b_1/\gcd(Nb_1, s_1)) - 1$ linear Diophantine equations. However, we notice that when $s_1 \le b_1$, each block contains at least one accessed address. In addition, the memory access sequence can be represented by a union of $(period_{eb}^A + 1)$ closed forms, where $period_{eb}^A = (s_1/\gcd(Nb_1, s_1)) \le (b_1/\gcd(Nb_1, s_1))$. Thus, in this case, it is better to use $(period_{eb}^A + 1)$ closed forms to represent the memory access sequence. On the other hand, when $s_1 > b_1$, each block may not contain any accessed address; thus, it is better to find the distance vectors for generating the memory access sequence in this case. We will show that, for two especially interesting cases, the distance vectors R and L can be found in constant time.

4.2.1 Cases Where $s_1 \le b_1$

$$\begin{aligned} & local_A(p) \cap [l_1 : u_1 : s_1] \\ = \; & \textstyle\bigcup_{j=j_{pf}}^{j_{pl}} [bot_a(A,p,j) : top_a(A,p,j) : s_1] \\ = \; & [bot_a(A,p,j_{pf}) : top_a(A,p,j_{pf}) : s_1] \;\cup \\ & \quad \Big(\textstyle\bigcup_{j=j_{pf}+1}^{\min\{j_{pl},\, j_{pf}+period_{eb}^A\}} [[bot_a(A,p,j) : top_a(A,p,j) : s_1] : u_1 : period_e^A * s_1]\Big) \\ = \; & [bot_a(A,p,j_{pf}) : top_a(A,p,j_{pf}) : s_1] \;\cup \\ & \quad [\Big(\textstyle\bigcup_{j=j_{pf}+1}^{\min\{j_{pl},\, j_{pf}+period_{eb}^A\}} [bot_a(A,p,j) : top_a(A,p,j) : s_1]\Big) : u_1 : period_e^A * s_1]. \end{aligned}$$

When $s_1 \leq b_1$, each local block of array A contains at least one element referenced by $A(l_1 : u_1 : s_1)$; in addition, using the algorithm in Figure 2, both j_{pf} and j_{pl} can be computed in constant time. Since the memory access sequence can be represented by a union of $(period_{eb}^A + 1)$ closed forms, the number of time units of the calculating boundary coefficients is $O(period_{eb}^A) = O(s_1/\gcd(Nb_1, s_1))$.

4.2.2 Cases Where $(N-1)b_1 < (s_1 \bmod Nb_1) < Nb_1$

These cases include interesting cases where $s_1 = yNb_1 - 1$ for every integer $y \geq 1$. First, the next smallest index x in the initial cycle accessed by PE_0 can be computed using an extrapolation method. Since $(N-1)b_1 < (s_1 \bmod Nb_1) < Nb_1$, index s_1 appears in column $(s_1 \bmod Nb_1)$ in PE_{N-1}; index $2s_1$ appears in column $2(s_1 \bmod Nb_1) - Nb_1$; index $3s_1$ appears in column $3(s_1 \bmod Nb_1) - 2Nb_1$; and so on. Suppose that y is the smallest integer such that $y*(s_1 \bmod Nb_1) - (y-1)*Nb_1 < b_1$; then, $x = y*s_1$. We have $y = \lceil (Nb_1 - b_1 + 1)/(Nb_1 - (s_1 \bmod Nb_1)) \rceil$ and $R = x = \lceil (Nb_1 - b_1 + 1)/(Nb_1 - (s_1 \bmod Nb_1)) \rceil * s_1$.

Second, since $(N-1)b_1 < (s_1 \bmod Nb_1) < Nb_1$, index $(period_e^A - 1)*s_1$ appears in PE_0. Therefore, the largest index in the initial cycle accessed by PE_0 is $(period_e^A - 1) * s_1$. Thus, we have $L = period_e^A * s_1 - (period_e^A - 1) * s_1 = s_1$.

4.2.3 Cases Where $0 < (s_1 \bmod Nb_1) < b_1$

These cases are dual cases where $(N-1)b_1 < (s_1 \bmod Nb_1) < Nb_1$, and they include interesting cases where $s_1 = yNb_1 + 1$ for every integer $y \geq 1$. First, since $0 < (s_1 \bmod Nb_1) < b_1$, index s_1 appears in PE_0. Therefore, the next smallest index in the initial cycle accessed by PE_0 is s_1. Thus, we have $R = s_1$.

Second, the largest index x in the initial cycle accessed by PE_0 can be computed using an extrapolation method. Since $0 < (s_1 \bmod Nb_1) < b_1$, index $(period_e^A - 1) * s_1$ appears in column $Nb_1 - (s_1 \bmod Nb_1)$ in PE_{N-1}; index $(period_e^A - 2) * s_1$ appears in column $Nb_1 - 2(s_1 \bmod Nb_1)$; and so on. Suppose that y is the smallest integer such that $Nb_1 - y * (s_1 \bmod Nb_1) < b_1$; then, $x = (period_e^A - y) * s_1$. We have $y = \lceil (Nb_1 - b_1 + 1)/(s_1 \bmod Nb_1) \rceil$ and $L = period_e^A * s_1 - x = y * s_1$ $= \lceil (Nb_1 - b_1 + 1)/(s_1 \bmod Nb_1) \rceil * s_1$.

Input: a_1, a_2 (the range of an array $A(a_1 : a_2)$), l_1, u_1, s_1 (the parameters of the access pattern $A(l_1 : u_1 : s_1)$), N (number of PEs), b_1 (block size), and p (a processing element ID).

Output: The $\Delta\mathcal{M}$ table.

1. #define $nxt(x,\ y,\ z) = x + ((y - x) \bmod z)$;
2. $(d, x, y) \longleftarrow$ EXTENDED-EUCLID(Nb_1, s_1);
   {* $d = \gcd(Nb_1, s_1) = x * Nb_1 + y * s_1$. *}
3. $period_e^A = Nb_1/d$;
4. $lp = a_1 + pb_1$; $rp = a_1 + pb_1 + b_1 - 1$;

Step 1: {* Handle the special cases where $s_1 \leq b_1$. *}

5. **if** $(s_1 \leq b_1)$ **return** $\Delta\mathcal{M} =$
6. $[bot_a(A, p, j_{pf}) : top_a(A, p, j_{pf}) : s_1] \cup$
7. $[\Big(\bigcup_{j=j_{pf}+1}^{\min\{j_{pl}, j_{pf}+period_{eb}^A\}} [bot_a(A, p, j) :$
8. $top_a(A, p, j) : s_1]\Big) : u_1 : period_e^A * s_1]$;

Step 2: {* Check whether $\Delta\mathcal{M}$ is empty or not. *}

9. **if** $(nxt(lp - l_1, d, d) > rp - l_1)$ **return** $\Delta\mathcal{M} = \phi$;

Step 3: {* Find the starting point accessed by PE_p. *}

10. $start = \infty$; $length = 0$;
11. **for** $i = nxt(lp - l_1,\ d,\ d),\ rp - l_1,\ d$
12. $loc = l_1 + (s_1/d)(iy + Nb_1\lceil -iy/(Nb_1)\rceil)$;
13. $start = \min(start, loc)$;
14. $length = length + 1$;
15. **endfor**

Step 4: {* Derive distance vectors R and L. *}

16. **if** $((N - 1)b_1 < (s_1 \bmod Nb_1) < Nb_1)$ **then**
17. $R = \lceil (Nb_1 - b_1 + 1)/(Nb_1 - (s_1 \bmod Nb_1))\rceil * s_1$;
18. $L = s_1$;
19. **else if** $(0 < (s_1 \bmod Nb_1) < b_1)$ **then**
20. $R = s_1$;
21. $L = \lceil (Nb_1 - b_1 + 1)/(s_1 \bmod Nb_1)\rceil * s_1$;
22. **else**
23. $R = \infty$; $L' = 0$;
24. **for** $i = d,\ b_1 - 1,\ d$
25. $loc = (s_1/d)(iy + Nb_1\lceil -iy/(Nb_1)\rceil)$;
26. $R = \min(R, loc)$;
27. $L' = \max(L', loc)$;
28. **endfor**
29. $L = Nb_1 * s_1/d - L'$;
30. **endif endif**

Step 5: {* Calculate the first cycle of the memory access sequence $\delta\mathcal{M}$. *}

31. $now = start$;
32. $\delta\mathcal{M}[0] = now$; $i = 1$;
33. **while** $(i < length)$ **do**
34. **if** $(pb_1 \leq ((now - a_1 + R) \bmod (Nb_1)) \leq pb_1 + b_1 - 1)$ **then**
35. $now = now + R$;
36. **else if** $(pb_1 \leq ((now - a_1 + L) \bmod (Nb_1)) \leq pb_1 + b_1 - 1)$ **then**
37. $now = now + L$;
38. **else** $now = now + R + L$;
39. **endif endif**
40. $\delta\mathcal{M}[i] = now$; $i = i + 1$;
41. **endwhile**

Step 6: {* Formulate the memory access sequence $\Delta\mathcal{M}$. *}

42. **return** $\Delta\mathcal{M} = [\delta\mathcal{M} : u_1 : period_e^A * s_1]$. $\square$

Figure 4: An algorithm for deriving the memory access sequence based on Theorem 5.

4.2.4 An Algorithm for Deriving the Memory Access Sequence

Let $\Delta\mathcal{M}$ represent the memory access sequence of $A(l_1 : u_1 : s_1)$ in PE_p and $\delta\mathcal{M}$ represent the first cycle of the memory access sequence of $A(l_1 : u_1 : s_1)$ in PE_p. Figure 4 presents an algorithm for deriving $\Delta\mathcal{M}$. This algorithm contains six steps as follows.

Step 1: {* Lines 5 to 8. *}

Deal with special cases where $s_1 \leq b_1$ as follows:

if s_1 (stride) $\leq b_1$ (block size) **then** $\Delta\mathcal{M} = [bot_a(A, p, j_{pf}) : top_a(A, p, j_{pf}) : s_1] \ \cup$ $[\Big(\bigcup_{j=j_{pf}+1}^{\min\{j_{pl}, j_{pf}+period_{eb}^A\}} [bot_a(A, p, j) : top_a(A, p, j) : s_1]\Big) : u_1 : period_e^A * s_1]$, **and then** STOP;

{* In the following, $s_1 > b_1$. *}

Step 2: {* Line 9. *}

check whether $\Delta\mathcal{M}$ is empty or not in constant time;

if $\Delta\mathcal{M} = \phi$ **then** STOP;

Step 3: {* Lines 10 to 15. *}

find the starting point accessed by PE_p;

Step 4: {* Lines 16 to 30. *}

derive the distance vectors R and L using a variant algorithm proposed by Kennedy *et al.* [21, 22];

Step 5: {* Lines 31 to 41. *}

calculate the first cycle of the memory access sequence $\delta\mathcal{M}$ according to Theorem 5;

Step 6: {* Line 42. *}

formulate the memory access sequence $\Delta\mathcal{M} = [\delta\mathcal{M} : u_1 : period_e^A * s_1]$;   STOP. □

4.3 Relation Between Memory Access Sequence Generation and Communication Set Generation

We will now analyze the set $send_C(p,q)$ again:

$$\begin{aligned} & send_C(p,q) \\ = \ & local_C(p) \cap f_2(exec(q)) \\ = \ & local_C(p) \cap \Big([bot_f(A,q,j_{qf}) : top_f(A,q,j_{qf}) : s_2] \cup \\ & \quad \Big(\bigcup_{j=j_{qf}+1}^{\min\{j_{ql}, j_{qf}+period_{eb}^A\}} [[bot_f(A,q,j) : top_f(A,q,j) : s_2] : u_2 : period_e^A * s_2]\Big)\Big). \end{aligned}$$

Since for every $j \in [j_{qf}+1 : \min\{j_{ql}, j_{qf}+period_{eb}^A\}]$, the set $[bot_f(A,q,j) : u_2 : period_e^A * s_2]$ forms a lattice, the problem of solving $local_C(p) \cap [bot_f(A,q,j) : u_2 : period_e^A * s_2]$ is reduced to a variant problem of generating the memory access sequence. However, this new variant problem is different from the original one because, even if index $bot_f(A,q,j) + i * period_e^A * s_2$ is not in $local_C(p)$ for some j and i, it is still possible that $local_C(p) \cap [bot_f(A,q,j) + i * period_e^A * s_2 : top_f(A,q,j) + i * period_e^A * s_2 : s_2]$

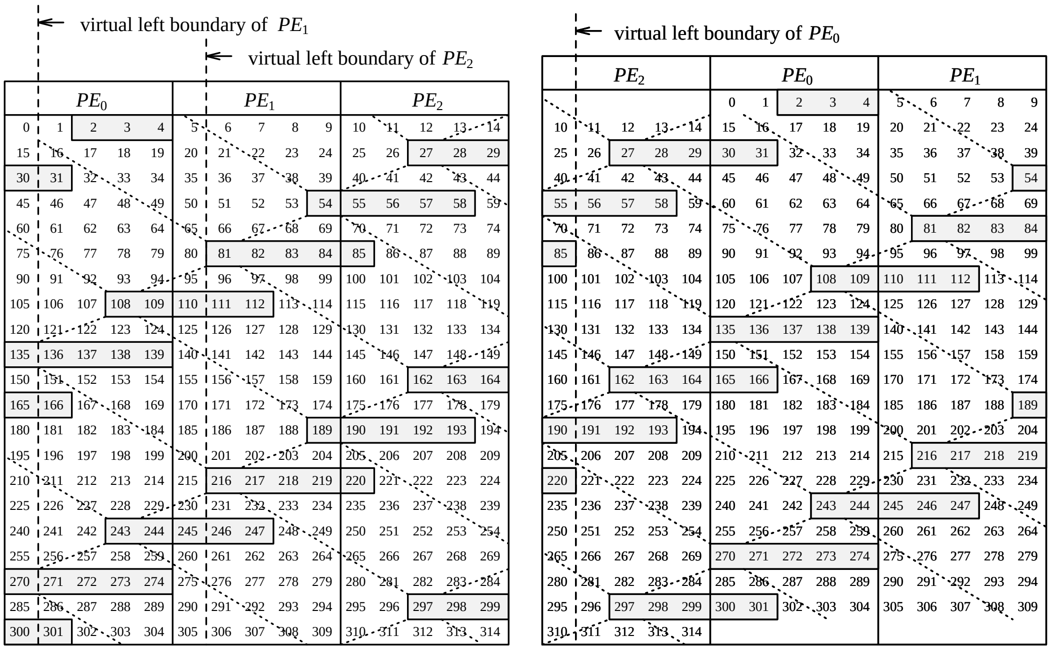
is not empty. In this case, we need to consider index $bot_f(A,q,j) + i * period_e^A * s_2$ for the further process of $send_C(p,q)$.

Similar to the discussion in Section 3.2.1, in order to guarantee that the regular section $[bot_f(A,q,j)+ i*period_e^A*s_2 : top_f(A,q,j)+i*period_e^A*s_2 : s_2]$ will intersect with at most one local block of $local_C(p)$, we will deal with the two cases, $\lceil \frac{b_1}{s_1} \rceil \leq \lceil \frac{(N-1)b_2+1}{s_2} \rceil$ and $\lceil \frac{b_2}{s_2} \rceil \leq \lceil \frac{(N-1)b_1+1}{s_1} \rceil$, separately as described in Property 1. In this presentation, we will only present the case where $\lceil \frac{b_1}{s_1} \rceil \leq \lceil \frac{(N-1)b_2+1}{s_2} \rceil$; the other case where $\lceil \frac{b_2}{s_2} \rceil \leq \lceil \frac{(N-1)b_1+1}{s_1} \rceil$ can be solved in a similar way. Under this constraint, the condition $Nb_2 + pb_2 - top_f(A,q,j) + bot_f(A,q,j) > pb_2 + b_2 - 1$ is always true. The reason why we need this constraint will become clear in the next paragraph.

We will extend the left boundary of PE_p from column pb_2 to a virtual left boundary $lb = pb_2 - top_f(A,q,j) + bot_f(A,q,j)$ if $pb_2 - top_f(A,q,j) + bot_f(A,q,j) \geq 0$ or to a virtual left boundary $lb = Nb_2+pb_2-top_f(A,q,j)+bot_f(A,q,j)$ if $pb_2-top_f(A,q,j)+bot_f(A,q,j) < 0$. Let $local'_C(p)$ contain data from column lb to column pb_2+b_2-1 on a spiral cylinder. Then, all the lattice points in the set $local'_C(p)$ $\cap$ $[bot_f(A,q,j) : u_2 : period_e^A * s_2]$ can be enumerated according to Corollary 6; in addition, for each element $bot_f(A,q,j)+i*period_e^A*s_2$ in the mentioned set, $local_C(p) \cap [bot_f(A,q,j)+i*period_e^A*s_2 : top_f(A,q,j) + i * period_e^A * s_2 : s_2] \neq \phi$.

Example 1: Suppose that $A(0:a_2)$ and $C(0:c_2)$ are distributed over three processing elements with $cyclic(9)$ and $cyclic(5)$ distributions, respectively; the loop body of a doall statement is $A(4+i*2) = g(C(2+i))$, where g is a function, and $u_1 = 628$. Then, $a_1 = 0$; $c_1 = 0$; $N = 3$; $b_1 = 9$; $b_2 = 5$; $l_1 = 4$; $s_1 = 2$; $l_2 = 2$; $s_2 = 1$; and $u_2 = 314$. Figure 5-(a) shows the memory access sequence of $A(4+i*2)$ by PE_0. Figure 5-(b) illustrates $send_C(p,0)$ for $0 \leq p \leq 2$, which represents elements of array C and will be sent to PE_0. Readers can check that for every p, $send_C(p,0)$ cannot be represented by a closed form in this case.

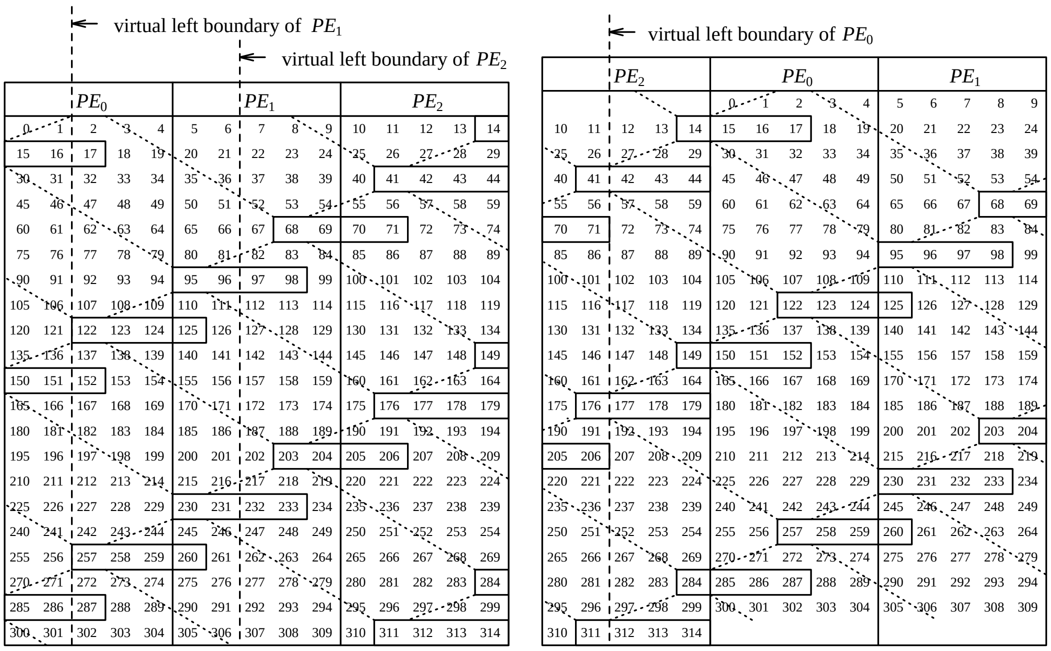
However, $period_{eb}^A = s_1/\gcd(Nb_1, s_1) = 2$. First, the set $[bot_f(A,0,1) : u_2 : period_e^A * s_2] = [14 : 314 : 27]$ forms a lattice as shown in Figure $6-(a)$. Although index $bot_f(A,0,1)+2*period_e^A*s_2 = 68$ is not in $local_C(2)$, $local_C(2) \cap [bot_f(A,0,1) + 2 * period_e^A * s_2 : top_f(A,0,1) + 2 * period_e^A * s_2 : s_2]$ $= local_C(2) \cap [68 : 71 : 1] = [70 : 71 : 1]$. Therefore, we need to consider the regular section $[68 : 71 : 1]$ for the process of $send_C(2,0)$. In this case, for processing elements $p = 1$ and 2, the virtual left boundary of PE_p is $lb = pb_2 - top_f(A,0,1) + bot_f(A,0,1) = 2$ and 7, respectively, as shown in Figure 7-(a). For processing element $p = 0$, the virtual left boundary of PE_p is $lb = Nb_2 + pb_2 - top_f(A, 0, 1) + bot_f(A, 0, 1) = 12$, as shown in Figure 7-(b).



(a) (b)

Figure 7: When dealing with the lattice $[bot_f(A, 0, 1) : u_2 : period_e^A * s_2]$, (a) for processing elements $p = 1$ and 2, the virtual left boundary of PE_p is $pb_2 - top_f(A, 0, 1) + bot_f(A, 0, 1) = 2$ and 7, respectively; (b) for processing element $p = 0$, the virtual left boundary of PE_p is $Nb_2 + pb_2 - top_f(A, 0, 1) + bot_f(A, 0, 1) = 12$.

Similarly, the set $[bot_f(A, 0, 2) : u_2 : period_e^A * s_2] = [27 : 314 : 27]$ forms a lattice as shown in Figure $6 - (b)$. Although index $bot_f(A, 0, 2) + 2 * period_e^A * s_2 = 81$ is not in $local_C(2)$, $local_C(2) \cap [bot_f(A, 0, 2) + 2 * period_e^A * s_2 : top_f(A, 0, 2) + 2 * period_e^A * s_2 : s_2] = local_C(2) \cap [81 : 85 : 1] = [85 : 85 : 1]$. Therefore, we need to consider the regular section $[81 : 85 : 1]$ for the process of $send_C(2, 0)$. In this case, for processing elements $p = 1$ and 2, the virtual left boundary of PE_p is $lb = pb_2 - top_f(A, 0, 2) + bot_f(A, 0, 2) = 1$ and 6, respectively, as shown in Figure 8-(a). For processing element $p = 0$, the virtual left boundary of PE_p is $lb = Nb_2 + pb_2 - top_f(A, 0, 2) + bot_f(A, 0, 2) = 11$, as shown in Figure 8-(b). $\square$



(a) (b)

Figure 8: When dealing with the lattice $[bot_f(A,0,2) : u_2 : period_e^A * s_2]$, (a) for processing elements $p = 1$ and 2, the virtual left boundary of PE_p is $pb_2 - top_f(A,0,2) + bot_f(A,0,2) = 1$ and 6, respectively; (b) for processing element $p = 0$, the virtual left boundary of PE_p is $Nb_2 + pb_2 - top_f(A,0,2) + bot_f(A,0,2) = 11$.

4.3.1 The Cases When $period_e^A * s_2 \leq b_2 + top_f(A,q,j) - bot_f(A,q,j)$

Suppose that $local'_C(p)$ includes elements among the columns from lb to rb on a spiral cylinder.

Define

$$
\begin{array}{l}
l = bot_f(A,q,j);\;\; u = u_2;\;\; s = period_e^A * s_2; \\
d = \gcd(Nb_2,\ s);\;\; period_{sb}^C = s/d;\;\; period_s = Nb_2 * s/d; \\
len = top_f(A,q,j) - bot_f(A,q,j);\;\; b = b_2 + len; \\
lb = \begin{cases} pb_2 - len, & \text{if } pb_2 - len \geq 0; \\ Nb_2 + pb_2 - len, & \text{otherwise;} \end{cases} \quad rb = pb_2 + b_2 - 1; \\
lp = c_1 + lb;\;\; rp = c_1 + rb; \\
k'_{pf} = \lceil (l - rp)/(Nb_2) \rceil;\;\; k'_{pl} = \lfloor (u - lp)/(Nb_2) \rfloor; \\
bot'_a(C,p,k) = nxt(\max\{lp + kNb_2, l\},\ l,\ s); \\
top'_a(C,p,k) = nxt(\min\{rp + kNb_2, u\} - s + 1,\ l,\ s); \\
\alpha_k = \begin{cases} k, & \text{if } lb < rb; \\ k-1, & \text{if } lb > rb; \end{cases} \quad \beta_k = \begin{cases} k, & \text{if } lb < rb; \\ k+1, & \text{if } lb > rb; \end{cases} \\
local_C(p) \cap [bot'_a(C,p,k) : bot'_a(C,p,k) + len : s_2] = \\
\qquad [bot_a(C,p,\beta_k) : \min\{top_a(C,p,\beta_k),\ bot'_a(C,p,k) + len\} : s_2].
\end{array}
$$

Since $s \leq b$, similar to the derivation in Section 4.2.1, each local block of $local'_C(p)$ contains at least one lattice point in $[l : u : s]$. Thus, we have

$$local'_C(p) \cap [l : u : s] = [bot'_a(C,p,\alpha_{k'_{pf}}) : top'_a(C,p,k'_{pf}) : s] \cup$$
$$[\Big(\bigcup_{k=k'_{pf}+1}^{\min\{\beta_{k'_{pl}},k'_{pf}+period^C_{sb}\}} [bot'_a(C,p,\alpha_k) : top'_a(C,p,k) : s]\Big) : u : period_s].$$

Therefore,

$$\begin{aligned}
& local_C(p) \cap [[bot_f(A,q,j) : top_f(A,q,j) : s_2] : u_2 : period^A_e * s_2] \\
= \;& local_C(p) \cap [[l : l + len : s_2] : u : s] \\
= \;& local_C(p) \cap [(local'_C(p) \cap [l : u : s]) : (local'_C(p) \cap [l : u : s]) + len : s_2] \\
= \;& [bot_a(C,p,k'_{pf}) : \min\{top_a(C,p,k'_{pf}),\ bot'_a(C,p,\alpha_{k'_{pf}}) + len\} : s_2] \ \cup \\
& [[bot'_a(C,p,\alpha_{k'_{pf}}) + s : \min\{top_a(C,p,k'_{pf}),\ bot'_a(C,p,\alpha_{k'_{pf}}) + s + len\} : s_2] : \\
& \quad \min\{top_a(C,p,k'_{pf}),\ top'_a(C,p,k'_{pf}) + len\} : s] \ \cup \\
& [\Big(\bigcup_{k=k'_{pf}+1}^{\min\{\beta_{k'_{pl}},k'_{pf}+period^C_{sb}\}} \Big([bot_a(C,p,k) : \min\{top_a(C,p,k),\ bot'_a(C,p,\alpha_k) + len\} : s_2] \ \cup \\
& \quad [[bot'_a(C,p,\alpha_k) + s : \min\{top_a(C,p,k),\ bot'_a(C,p,\alpha_k) + s + len\} : s_2] \ : \\
& \quad\ \min\{top_a(C,p,k),\ top'_a(C,p,k) + len\} : s]\Big)\Big) : u : period_s].
\end{aligned}$$

4.3.2 Other Interesting Cases

The following two cases correspond to two cases in Sections 4.2.2 and 4.2.3.

1. When $Nb_2 - b < (s \bmod Nb_2) < Nb_2$: Similar to the derivation in Section 4.2.2, we have $R = \lceil (Nb_2 - b + 1)/(Nb_2 - (s \bmod Nb_2)) \rceil * s$ and $L = s$.

2. When $0 < (s \bmod Nb_2) < b$: Similar to the derivation in Section 4.2.3, we have $R = s$ and $L = \lceil (Nb_2 - b + 1)/(s \bmod Nb_2) \rceil * s$.

4.4 An Algorithm for Deriving the Communication Sets

In this subsection, we will present an algorithm for calculating $local_C(p) \cap [[bot_f(A,q,j) : top_f(A,q,j) : s_2] : u_2 : period^A_e * s_2]$. We will first use the algorithm described in Section 4.2.4, according to Corollary 6, to enumerate all indices $bot_f(A,q,j) + i * period^A_e * s_2$ such that $local_C(p) \cap [bot_f(A,q,j) + i * period^A_e * s_2 : top_f(A,q,j) + i * period^A_e * s_2 : s_2] \neq \phi$. Then, the communication set $local_C(p) \cap$

$[[bot_f(A,q,j) : top_f(A,q,j) : s_2] : u_2 : period_e^A * s_2]$ can be derived incrementally according to these indices. We will now present the cases where $lb = pb_2 - top_f(A,q,j) + bot_f(A,q,j) \geq 0$, thus $lb \leq pb_2 + b_2 - 1 = rb$. We will illustrate certain modified code in Figure 10 so that the modified algorithm can handle the cases where $pb_2 - top_f(A,q,j) + bot_f(A,q,j) < 0$ and $lb = Nb_2 + pb_2 - top_f(A,q,j) + bot_f(A,q,j) > pb_2 + b_2 - 1 = rb$ at the end of this section.

Let $\delta\mathcal{M}$ represent the first cycle of the memory access sequence of $C(bot_f(A,q,j) : u_2 : period_e^A * s_2)$ in $local'_C(p)$; let $\Delta\mathcal{Z}$ represent the communication set of $local_C(p) \cap [[bot_f(A,q,j) : top_f(A,q,j) : s_2] : u_2 : period_e^A * s_2]$ in PE_p; and let $\delta\mathcal{Z}$ represent the first cycle of the communication set of $local_C(p) \cap [[bot_f(A,q,j) : top_f(A,q,j) : s_2] : u_2 : period_e^A * s_2]$ in PE_p. Figure 9 presents an algorithm for deriving $\Delta\mathcal{Z}$. This algorithm also contains six steps as follows.

Step 1: {* Line 9. *}

Handle the special cases where $period_e^A * s_2 \leq b_2 + top_f(A,q,j) - bot_f(A,q,j)$ as follows:

if $period_e^A * s_2 \leq b_2 + top_f(A,q,j) - bot_f(A,q,j)$ **then** $\Delta\mathcal{Z}$ can be represented by the formula mentioned in Section 4.3.1, **and then** STOP;

{* In the following, $period_e^A * s_2 > b_2 + top_f(A,q,j) - bot_f(A,q,j)$. *}

Step 2: {* Line 10. *}

check whether $\Delta\mathcal{Z}$ is empty or not in constant time;

if $\Delta\mathcal{Z} = \phi$ **then** STOP;

Step 3: {* Lines 11 to 16. *}

find the starting point accessed among the columns from lb to rb;

Step 4: {* Lines 17 to 31. *}

derive distance vectors R and L using a variant algorithm proposed by Kennedy *et al.* [21, 22];

Step 5: {* Lines 32 to 44. *}

calculate the first cycle of the memory access sequence $\delta\mathcal{M}$ according to Corollary 6;

compute the first cycle of the communication set $\delta\mathcal{Z}$ according to $\delta\mathcal{M}$;

Step 6: {* Line 45. *}

formulate the communication set $\Delta\mathcal{Z} = [\delta\mathcal{Z} : u_2 : period_s]$; STOP. □

Input: a_1, a_2 (the range of a generated array $A(a_1 : a_2)$), l_1, u_1, s_1 (parameters of the access pattern $A(l_1 : u_1 : s_1)$), c_1, c_2 (the range of a used array $C(c_1 : c_2)$), l_2, u_2, s_2 (parameters of the access pattern $C(l_2 : u_2 : s_2)$), N (number of PEs), b_1, b_2 (block sizes of A and C), p and q (processing element ID).
Output: The $\Delta\mathcal{Z}$ table. {* $\Delta\mathcal{Z} = local_C(p) \cap [[bot_f(A,q,j) : top_f(A,q,j) : s_2] : u_2 : period_e^A * s_2]$. *}

1. #define $nxt(x,\ y,\ z) = x + ((y-x) \bmod z)$;
2. **if** $(bot_f(A,q,j) > top_f(A,q,j))$ **return** $\Delta\mathcal{Z} = \phi$;
3. $l = bot_f(A,q,j)$; $s = period_e^A * s_2$;
4. $(d,x,y) \longleftarrow$ Extended-Euclid(Nb_2, s);
   {* $d = \gcd(Nb_2, s) = x * Nb_2 + y * s$. *}
5. $period_e^{C'} = Nb_2/d$; $period_s = period_e^{C'} * s$;
6. $len = top_f(A,q,j) - bot_f(A,q,j)$; $b = b_2 + len$;
7. $lb = pb_2 - len$; $rb = pb_2 + b_2 - 1$;
8. $lp = c_1 + lb$; $rp = c_1 + rb$;

Step 1: {* Handle the special cases where $s \le b$. *}
9. **if** $(s \le b)$ **return** $\Delta\mathcal{Z} =$ the formula in Section 4.3.1;

Step 2: {* Check whether $\Delta\mathcal{Z}$ is empty or not. *}
10. **if** $(nxt(lp-l,\ d,\ d) > rp - l)$ **return** $\Delta\mathcal{Z} = \phi$;

Step 3: {* Find the starting point accessed among the columns from lb to rb. *}
11. $start = \infty$; $length = 0$;
12. **for** $i = nxt(lp-l,\ d,\ d),\ rp-l,\ d$
13. $\quad loc = l + (s/d)(iy + Nb_2\lceil -iy/(Nb_2)\rceil)$;
14. $\quad start = \min(start, loc)$;
15. $\quad length = length + 1$;
16. **endfor**

Step 4: {* Derive distance vectors R and L. *}
17. **if** $(Nb_2 - b < (s \bmod Nb_2) < Nb_2)$ **then**
18. $\quad R = \lceil (Nb_2 - b + 1)/(Nb_2 - (s \bmod Nb_2))\rceil * s$;
19. $\quad L = s$;
20. **else if** $(0 < (s \bmod Nb_2) < b)$ **then**
21. $\quad R = s$;
22. $\quad L = \lceil (Nb_2 - b + 1)/(s \bmod Nb_2)\rceil * s$;
23. **else**
24. $\quad R = \infty$; $L' = 0$;
25. $\quad$**for** $i = d,\ b-1,\ d$
26. $\quad\quad loc = (s/d)(iy + Nb_2\lceil -iy/(Nb_2)\rceil)$;
27. $\quad\quad R = \min(R, loc)$;
28. $\quad\quad L' = \max(L', loc)$;
29. $\quad$**endfor**
30. $\quad L = Nb_2 * s/d - L'$;
31. **endif endif**

Step 5: {* Calculate the first cycle of both the memory access sequence and the communication set $\delta\mathcal{Z}$. *}
32. $\delta\mathcal{Z} = \phi$; $now = start$;
33. $k = (now - c_1)/(Nb_2)$; $i = 1$;
34. $\delta\mathcal{Z} = \delta\mathcal{Z} \cup [\max(bot_a(C,p,k),\ now) : \min(top_a(C,p,k),\ now + len) : s_2]$;
35. **while** $(i < length)$ **do**
36. $\quad$**if** $(lb \le ((now - c_1 + R) \bmod (Nb_2)) \le rb)$ **then**
37. $\quad\quad now = now + R$;
38. $\quad$**else if** $(lb \le ((now - c_1 + L) \bmod (Nb_2)) \le rb)$ **then**
39. $\quad\quad now = now + L$;
40. $\quad$**else** $now = now + R + L$;
41. $\quad$**endif endif**
42. $\quad k = (now - c_1)/(Nb_2)$; $i = i + 1$;
43. $\quad \delta\mathcal{Z} = \delta\mathcal{Z} \cup [\max(bot_a(C,p,k),\ now) : \min(top_a(C,p,k),\ now + len) : s_2]$;
44. **endwhile**

Step 6: {* Formulate the communication set $\Delta\mathcal{Z}$. *}
45. **return** $\Delta\mathcal{Z} = [\delta\mathcal{Z} : u_2 : period_s]$. □

Figure 9: An algorithm for deriving the communication set based on Corollary 6 where $0 \le lb \le rb$.

7′. $lb = Nb_2 + pb_2 - len$; $rb = pb_2 + b_2 - 1$;
10′. **if** $(nxt(lp-l,\ d,\ d) > Nb_2 + rp - l)$ **return** $\Delta\mathcal{Z} = \phi$;
12′. **for** $i' = nxt(lp-l,\ d,\ d),\ Nb_2 + rp - l,\ d$
$\quad i = (i' \bmod Nb_2)$;
33′. **if** $((now \bmod Nb_2) \le rb)$ **then**
$\quad k = (now - c_1)/(Nb_2)$;
else $k = (now - c_1)/(Nb_2) + 1$;
endif
$i = 1$;
36′. **if** (not $(rb < ((now - c_1 + R) \bmod (Nb_2)) < lb)$) **then**
38′. **else if** (not $(rb < ((now - c_1 + L) \bmod (Nb_2)) < lb)$) **then**
42′. **if** $((now \bmod Nb_2) \le rb)$ **then**
$\quad k = (now - c_1)/(Nb_2)$;
else $k = (now - c_1)/(Nb_2) + 1$;
endif
$i = i + 1$;

Figure 10: Corresponding modified code for deriving the communication set where $lb > rb$.

5 Representation of Communication Sets by Closed Forms

In Sections 3 and 4, 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 $(period^A_{sb}+1)$ or $(period^C_{sb}+1)$ closed forms in Section 3; otherwise, we must apply an efficient algorithm $period^A_{eb}$ or $period^C_{eb}$ times in order to generate the communication set $send_C(p,q)$ for some p and q in Section 4. Since the number of boundary indices of these closed forms or the number of times that we apply an efficient algorithm to generate communication sets which we need to calculate is proportional to the corresponding variables, $period^A_{sb}$; $period^C_{sb}$; $period^A_{eb}$; or $period^C_{eb}$, the computation overhead becomes serious if the corresponding $period^A_{sb}$; $period^C_{sb}$; $period^A_{eb}$; or $period^C_{eb}$ is large.

In this section, we will return to analysis of 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 following, we will use *closed forms* to represent *a constant number of closed forms.*

5.1 Determination of Suitable Block Sizes

Consider the target forall statement again. We will first present an ideal case. Suppose that we assign the entry $A(j)$ to PE_p, where $p = (\lfloor \frac{j-l_1}{s_1*h} \rfloor \bmod N)$, and the entry $C(j')$ to $PE_{p'}$, where $p' = (\lfloor \frac{j'-l_2}{s_2*h} \rfloor \bmod N)$. 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 in implementing the target forall statement. In this ideal case, we notice that $b_1 = s_1*h$ and $b_2 = s_2*h$.

We will now consider the general case. Suppose that the data distribution functions (defined in Property 3 in Section 3.2.2) 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 find that, 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, computation and communication overheads will be incurred due to random access patterns whose costs are relatively high. Table 4 summarizes certain conditions where processor sets and communication sets have closed forms.

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

Table 4: Conditions where processor sets and communication sets have closed forms. All-closed-forms condition occurs 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. In addition, only a constant time is required to test whether any one of these sets is empty or not. Otherwise, we only can use *ad hoc* methods to enumerate these sets or use indirect memory access methods to get their corresponding data. The latter case, of course, will incur a 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.

Since optimal data distribution schema between two Do-loops may be different, some data communication between them may be required. However, frequent data re-distribution is expensive. Thus, it is a compromise to let several consecutive Do-loops share a common data distribution scheme if arrays in these Do-loops are aligned together [28]. We will now present an algorithm for determining suitable block sizes. We will consider the following general cases: Suppose that in the loop bodies of a consecutive forall statements, there are a total of n different arrays, among which m different generated arrays appear in both the LHS and the RHS of the assignment (=); and $n - m$ different used arrays only appear in the RHS of the assignment as follows, where statements can appear in any permuted order (because we are only concerned with strides and block sizes):

$$
\begin{aligned}
A_1(l_{11} + i * s_{11}) &= f_{11}(\cdots); \\
A_1(l_{12} + i * s_{12}) &= f_{12}(\cdots); \\
&\vdots \\
A_1(l_{1x_1} + i * s_{1x_1}) &= f_{1x_1}(\cdots); \\
A_2(l_{21} + i * s_{21}) &= f_{21}(\cdots); \\
&\vdots \\
A_m(l_{m1} + i * s_{m1}) &= f_{m1}(\cdots); \\
&\vdots \\
A_m(l_{mx_m} + i * s_{mx_m}) &= f_{mx_m}(\cdots).
\end{aligned}
$$

In the above statements, $f_{ij}()$ is a function of $\Big(A_1(l_{11} + i * s_{11}),\ A_1(l_{12} + i * s_{12}),\ \ldots,\ A_1(l_{1x_1} + i * s_{1x_1}),$

$A_2(l_{21}+i{*}s_{21}), \ldots, A_m(l_{m1}+i{*}s_{m1}), \ldots, A_m(l_{mx_m}+i{*}s_{mx_m}), \ldots, A_n(l_{n1}+i{*}s_{n1}), \ldots, A_n(l_{nx_n}+i{*}s_{nx_n})\Big)$.

It is reasonable to assume that each stride s_{ij} is a small integer [34] [35]. We find that the block size b_i of array A_i is a multiple of $\mathrm{lcm}(s_{i1},\ s_{i2}, \ldots, s_{ix_i})$ because b_i must be a multiple of all s_{ij}, for $1 \le j \le x_i$. The following algorithm can determine suitable block sizes.

An Algorithm for determining suitable block sizes:

Step 1: {* Assign an initial block size. *}

We first construct a directed graph. Each node A_i represents a one-dimensional array, whose initial block size b_i is $\mathrm{lcm}(s_{i1},\ s_{i2}, \ldots, s_{ix_i})$. Each edge (A_i, A_j) specifies that in a statement, the variables of array A_i are in the LHS of the assignment, and that the variables of array A_j are in the RHS of the assignment.

Step 2: {* Determine block sizes so that they satisfy Case 2 or Case 3 in Table 4. *}

Each maximal strongly connected component in the graph is treated as a unit or a π-block. The graph then includes an acyclic partial ordering on the π-blocks. We will now define a new level for each π-block below, where Π is a π-block; Ψ is a source π-block; and Ω is a sink π-block:

$$nlevel(\Pi) = \min\{\max_{(\forall \text{ source } \Psi)}\{\mathrm{distance}(\Psi,\Pi)+1\},\ \max_{(\forall \text{ sink } \Omega)}\{\mathrm{distance}(\Pi,\Omega)+1\}\}.$$

Suppose that the *nlevel* of the acyclic π-blocks graph (arising from the directed graph) is nl. Then, we determine block sizes in this order:

for $i = 1$ to nl **do**

if a π-block in *nlevel* i contains more than one node, that is, these nodes form a strongly connected component, **then** we break the cycle by randomly letting a node A_j be a child of its neighboring node A_k (by changing the direction of all the edges "from A_k to A_j" to "from A_j to A_k") and recursively apply Step 2 again;

According to an ordering from child to parent by topological sorting in *nlevel* i:

for each A_j in *nlevel* i **do**

for all edges (A_k, A_j) and (A_j, A_k), where A_k are in *nlevel* i' and $i' < i$, and all edges (A_j, A_k) from A_k to A_j, where A_k are also in *nlevel* i **do**

$b_j = \mathrm{lcm}_{(\forall k)}(b_j,\ (b_k/s_{ky})s_{jx})$, where $A_j(l_{jx}+s_{jx})$ and $A_k(l_{ky}+s_{ky})$ appear in the same statement;

Step 3: {* Adjust block sizes so that as many block sizes can satisfy Case 4 in Table 4 as possible. *}

We now adjust each block size according to the reverse direction in Step 2 as follows:

for $i = nl$ down to 1 **do**

According to an ordering from parent to child by topological sorting in *nlevel* i:

for each A_k in *nlevel* i **do**

for all edges (A_j, A_k) from A_k to A_j, where A_j are also in *nlevel* i, and all edges (A_k, A_j) and (A_j, A_k), where A_j are in *nlevel* i' and $i' > i$ **do**

$b_k = \gcd_{(\forall j)}((b_j/s_{jx})s_{ky})$, where $A_k(l_{ky} + s_{ky})$ and $A_j(l_{jx} + s_{jx})$ appear in the same statement. {* Note that, $\gcd(x) = x$. *}

Step 4: {* Add a granularity factor. *}

for each b_i **do** $b_i = b_i * h$, where h is a granularity factor. □

In the above algorithm, the constructed directed graph is identical to the component affinity graph [28], which is used to determine data alignment and data distribution, if we ignore the weight of each edge in the component affinity graph. After Step 2, block sizes guarantee satisfaction of Case 2 or Case 3 in Table 4. This is because b_j/s_{jx} is a multiple of b_k/s_{ky}, where $A_j(l_{jx}+s_{jx})$ and $A_k(l_{ky}+s_{ky})$ appear in the same statement. The purpose of Step 3 is to adjust the block sizes so that they can satisfy Case 4 in Table 4. If there is only one edge from A_k to A_j or from A_j to A_k, and $b_j/s_{jx} = b_k/s_{ky}$, then block sizes b_j and b_k satisfy Case 4 in Table 4 for calculating the forall statement which involves $A_j(l_{jx} + s_{jx})$ and $A_k(l_{ky} + s_{ky})$. The granularity factor h in Step 4 can be determined by using an analytical model [29] or some knowledge bases. In the following, we will give an example to illustrate the idea of choosing block sizes. We assume that the iteration space of a forall statement is large enough such that each PE has to execute roughly the same number of iterations.

Example 2: Suppose that the loop bodies of nine consecutive forall statements are those shown below, where statements can appear in any permuted order:

(1) $A_1(l_{11} + i * s_{11}) = A_1(l_{11} + i * s_{11}) + A_2(l_{21} + i * s_{21});$

(2) $A_1(l_{12} + i * s_{12}) = A_1(l_{12} + i * s_{12}) - A_3(l_{31} + i * s_{31});$

(3) $A_2(l_{22} + i * s_{22}) = A_2(l_{22} + i * s_{22}) * A_4(l_{41} + i * s_{41});$

(4) $A_2(l_{23} + i * s_{23}) = A_2(l_{23} + i * s_{23}) + A_5(l_{51} + i * s_{51});$

(5) $A_3(l_{32} + i * s_{32}) = A_3(l_{32} + i * s_{32}) - A_5(l_{52} + i * s_{52});$

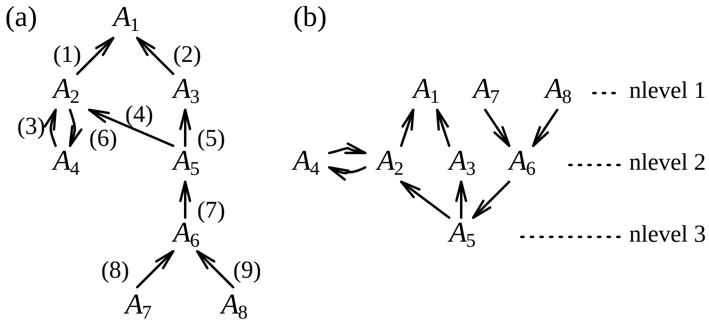
(6) $A_4(l_{42} + i * s_{42}) = A_4(l_{42} + i * s_{42}) * A_2(l_{24} + i * s_{24});$

(7) $A_5(l_{53} + i * s_{53}) = A_5(l_{53} + i * s_{53}) + A_6(l_{61} + i * s_{61});$

(8) $A_6(l_{62} + i * s_{62}) = A_6(l_{62} + i * s_{62}) - A_7(l_{71} + i * s_{71});$

(9) $A_6(l_{63} + i * s_{63}) = A_6(l_{63} + i * s_{63}) * A_8(l_{81} + i * s_{81}).$

Fig. 11-(a) shows the corresponding directed graph of these nine statements. In the graph, A_2 and A_4 form a maximal strongly connected component, and each of the remaining A_i forms a maximal strongly connected component. Fig. 11-(b) presents the *nlevel* of each node. When we break the cycle of the strongly connected component by A_2 and A_4, we randomly let A_4 be a child of A_2 in this example. Fig. 11-(c) also traces the mentioned algorithm, which includes four steps. Finally, statements 1, 2, and 6 satisfy Case 2 in Table 4; statement 3 satisfies Case 3 in Table 4; and statements 4, 5, 7, 8 and 9 satisfy Case 4 in Table 4. □



(c)

Step 1
$b_1^1 = \mathrm{lcm}(s_{11},\ s_{12})$
$b_2^1 = \mathrm{lcm}(s_{21},\ s_{22},\ s_{23},\ s_{24})$
$b_3^1 = \mathrm{lcm}(s_{31},\ s_{32})$
$b_4^1 = \mathrm{lcm}(s_{41},\ s_{42})$
$b_5^1 = \mathrm{lcm}(s_{51},\ s_{52},\ s_{53})$
$b_6^1 = \mathrm{lcm}(s_{61},\ s_{62},\ s_{63})$
$b_7^1 = s_{71}$
$b_8^1 = s_{81}$

Step 2	Step 3	Step 4
$b_1^2 = b_1^1$	$b_5^3 = b_5^2$	$b_1^4 = b_1^3 h$
$b_7^2 = b_7^1$	$b_6^3 = (b_5^3/s_{53})s_{61}$	$b_2^4 = b_2^3 h$
$b_8^2 = b_8^1$	$b_3^3 = (b_5^3/s_{52})s_{32}$	$b_3^4 = b_3^3 h$
$b_4^2 = b_4^1$	$b_2^3 = (b_5^3/s_{51})s_{23}$	$b_4^4 = b_4^3 h$
$b_2^2 = \mathrm{lcm}(b_2^1,\ (b_1^2/s_{11})s_{21},\ (b_4^2/s_{41})s_{22},\ (b_4^2/s_{42})s_{24})$	$b_4^3 = \gcd((b_2/s_{22})s_{41}, (b_2/s_{24})s_{42})$	$b_5^4 = b_5^3 h$
$b_3^2 = \mathrm{lcm}(b_3^1,\ (b_1^2/s_{12})s_{31})$	$b_8^3 = (b_6^3/s_{63})s_{81}$	$b_6^4 = b_6^3 h$
$b_6^2 = \mathrm{lcm}(b_6^1,\ (b_7^2/s_{71})s_{62},\ (b_8^2/s_{81})s_{63})$	$b_7^3 = (b_6^3/s_{62})s_{71}$	$b_7^4 = b_7^3 h$
$b_5^2 = \mathrm{lcm}(b_5^1,\ (b_2^2/s_{23})s_{51},\ (b_3^2/s_{32})s_{52},\ (b_6^2/s_{61})s_{53})$	$b_1^3 = \gcd((b_2/s_{21})s_{11}, (b_3/s_{31})s_{12})$	$b_8^4 = b_8^3 h$

Figure 11: (a) The corresponding directed graph of the nine statements, where (i) in each edge represents the i-th statement. (b) The corresponding *nlevel* for each node. (c) Block sizes b_i of A_i which are determined by four steps in sequence. b_i^j means the temporary value of b_i after Step j, and the final b_i is equal to b_i^4.

In the following subsections, we will derive processor sets and communication sets for Cases 2, 3, and 4 in Table 4.

5.2 The Case Where $b_1 = s_1 * h_1$ 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 sufficient to analyze the set of PEs which use elements of array C within a block of size b_2. We find 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 referenced 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 $f_A(bot_f(C,p,k_{pf}))$ may not be 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 referenced 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 $period_{sb}^A = (period_s * s_1)/(Nb_1s_2) = h_2$, $recv\_pe(p)$ can be represented by a union of at most $h_2 + 1$ closed forms:

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

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 will give an example to explain where these three closed forms come from.

Example 3: Suppose that the number of PEs is 4; that $a_1 = c_1 = 0$; that the loop body of a forall statement is $A(11+i*2) = g(C(2+i))$, where g is a function; and that $u_1 = 745$. 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 $b_2 = s_2 * h_1 * h_2 = 22$.

Fig. 12 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 deliberately written as a union of two closed forms, as we will derive a unified formula to represent $shead(p,q)$. Next, $sbody^1_C(0,2) = \phi$. $\square$

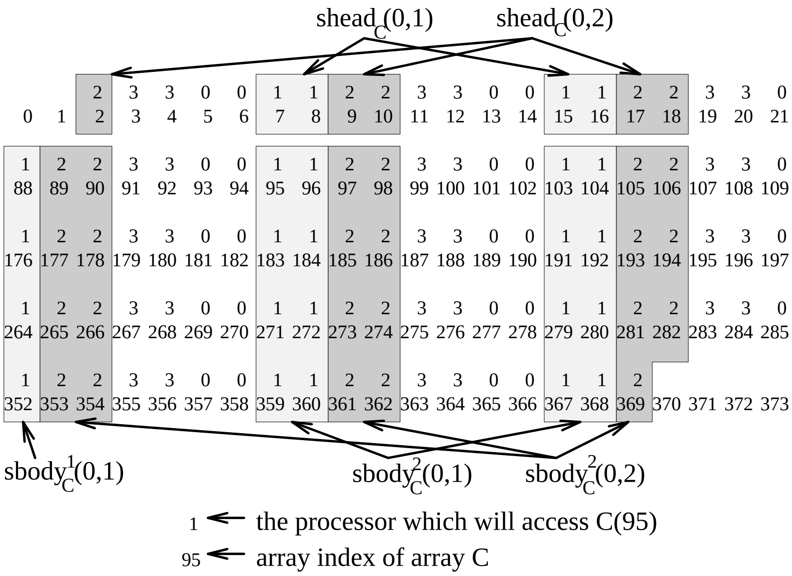


Figure 12: Elements of array C in PE_0, where array C is distributed by $cyclic(22)$ over four processors. In addition, $send_C(0,q) = shead_C(0,q) \cup sbody^1_C(0,q) \cup sbody^2_C(0,q)$, for $1 \le q \le 3$.

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

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

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

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

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

In summary, it is only necessary to determine two conditions and at most eight boundary variables to calculate $send_C(p,q)$ as shown in Table 5. Similarly, $recv_C(p,q)$ also has this property, and we will not repeat another Table 5 again in the following.

Conditions	$send_C(p,q)$
λ_1 and λ_2	$[\mu_1 : \mu_2 : s_2] \cup [[\mu_3 : \mu_3 + s_2(h_1-1) : s_2] : \mu_4 : Ns_2h_1] \cup$ $[([\mu_5 : \mu_6 : s_2] \cup [[\mu_7 : \mu_7 + s_2(h_1-1) : s_2] : \mu_8 : Ns_2h_1]) : u_2 : Nb_2]$
λ_1 and (not λ_2)	$[\mu_1 : \mu_2 : s_2] \cup [[\mu_3 : \mu_3 + s_2(h_1-1) : s_2] : \mu_4 : Ns_2h_1] \cup$ $[[[\mu_7 : \mu_7 + s_2(h_1-1) : s_2] : \mu_8 : Ns_2h_1] : u_2 : Nb_2]$
(not λ_1) and λ_2	$[([\mu_5 : \mu_6 : s_2] \cup [[\mu_7 : \mu_7 + s_2(h_1-1) : s_2] : \mu_8 : Ns_2h_1]) : u_2 : Nb_2]$
(not λ_1) and (not λ_2)	$[[[\mu_7 : \mu_7 + s_2(h_1-1) : s_2] : \mu_8 : Ns_2h_1] : u_2 : Nb_2]$

$\lambda_1 = (nxt(bot_l(C,p,k_{pf}),\ l_2,\ s_2) < l_2)$; $\lambda_2 = ((\mu_1 < \mu_5 \le top_f(A,q,j_{qf}))$ or $(\mu_6 - s_2(h_1 - 1) < \mu_5 \le \min\{\mu_6,\ top_f(A,q,j_{ql})\}))$; $\mu_1 = bot_f(A,q,j_{qf})$; $\mu_2 = \min\{top_f(A,q,j_{qf}),\ \mu_4\}$; $\mu_3 = bot_f(A,q,j_{qf}+1)$; $\mu_4 = top_a(C,p,k_{pf})$; $\mu_5 = bot_a(C,p,k'_{pf})$; $\mu_6 = nxt(\mu_5,\ top_f(A,q,j_{qf}),\ Ns_2h_1)$; $\mu_7 = nxt(\mu_5,\ \mu_3,\ Ns_2h_1)$; and $\mu_8 = top_a(C,p,k'_{pf})$.

Table 5: It is only necessary to determine two conditions and at most eight boundary variables to calculate $send_C(p,q)$.

Fourth, we are concerned with $recv_C(p,q)$, which is equal to $send_C(q,p)$. Hence, $recv_C(p,q)$ also can be represented by a union of three closed forms. As indicated in Section 3, we prefer that $recv_C(p,q)$

be represented based on indices of array A. In addition, there is a one-to-one correspondence between $rhead_C(p,q) \cup rbody_C^1(p,q) \cup rbody_C^2(p,q)$ and $f_2 f_1^{-1}\Big(f_1(f_2^{-1}(rhead_C(p,q))) \cup f_1(f_2^{-1}(rbody_C^1(p,q))) \cup f_1(f_2^{-1}(rbody_C^2(p,q)))\Big)$. 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:

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

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

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

$$
recv_C(p,q) = rhead_C(p,q) \cup rbody_C^1(p,q) \cup rbody_C^2(p,q).
$$

In the following, we will give an example to explain how indices of array A can be related to $recv_C(p,q)$.

Example 4: We continue with Example 3. Fig. 13 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, $f_1(f_2^{-1}(recv_C(1,0))) = f_1(f_2^{-1}(rhead_C(1,0))) \cup f_1(f_2^{-1}(rbody_C^1(1,0))) \cup f_1(f_2^{-1}(rbody_C^2(1,0)))$, where $f_1(f_2^{-1}(rhead_C(1,0))) = [21:23:2] \cup [[37:39:2]:49:16]$; $f_1(f_2^{-1}(rbody_C^1(1,0))) = [[183:183:2]:745:176]$; and $f_1(f_2^{-1}(rbody_C^2(1,0))) = [[[197:199:2]:225:16]:745:176]$. $f_1(f_2^{-1}(recv_C(2,0))) = f_1(f_2^{-1}(rhead_C(2,0))) \cup f_1(f_2^{-1}(rbody_C^2(2,0)))$, where $f_1(f_2^{-1}(rhead_C(2,0))) = [11:11:2] \cup [[25:27:2]:49:16]$ and $f_1(f_2^{-1}(rbody_C^2(2,0))) = [[[185:187:2]:225:16]:745:176]$. □

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

This case is a symmetrical case as did 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 $send\_pe(p)$,

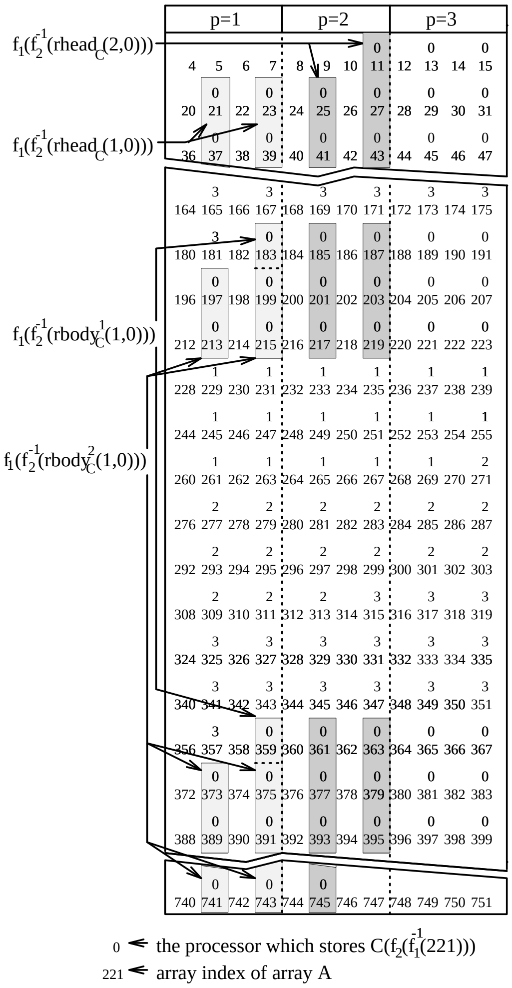


Figure 13: Elements of array A in PE_1 through PE_3, where array A is distributed by $cyclic(4)$ over four processors. In addition, $f_1(f_2^{-1}(recv_C(p,0))) = f_1(f_2^{-1}(rhead_C(p,0))) \cup f_1(f_2^{-1}(rbody_C^1(p,0))) \cup f_1(f_2^{-1}(rbody_C^2(p,0)))$, for $1 \le p \le 3$.

which is equal to $f_A(f_1(f_2^{-1}(local_C(p) \cap [l_2 : u_2 : s_2])))$. Since $period_{sb}^C = period_s/(Nb_2) = h_1$, $send\_pe(p)$ can be represented by a union of at most $h_1 + 1$ closed forms:

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

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 $period^A_{sb} = (period_s * s_1)/(Nb_1s_2) = 1$, it is sufficient to analyze the set of PEs which store elements of array C that will be 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: $shead_C(p,q)$, $sbody^1_C(p,q)$, and $sbody^2_C(p,q)$. 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.

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

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

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

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

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 $recv_C(p,q)$ 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

$$rhead_C(p,q) = \begin{cases} f_2 f_1^{-1}\Big([bot_f(C,q,k_{qf}) : \min\{top_f(C,q,k_{qf}), top_a(A,p,j_{pf})\} : s_1] \ \cup \\ \quad [[bot_f(C,q,k_{qf}+1) : top_f(C,q,k_{qf}+1) : s_1] : top_a(A,p,j_{pf}) : Ns_1h_2]\Big), \\ \qquad \text{if } nxt(bot_l(A,p,j_{pf}), l_1, s_1) < l_1; \\ \phi, \quad \text{otherwise.} \end{cases}$$

$$rbody_C^1(p,q) = \begin{cases} f_2 f_1^{-1}\Big([[bot_a(A,p,j'_{pf}) : nxt(bot_a(A,p,j'_{pf}), top_f(C,q,k_{qf}), Ns_1h_2) : s_1] : \\ \quad u_1 : Nb_1]\Big), \\ \qquad \text{if } bot_a(C,q,k_{qf}) < bot_f(A,p,j'_{pf}), \leq top_a(C,q,k_{qf}) \text{ or} \\ \qquad\quad nxt(bot_f(A,p,j'_{pf}), top_a(C,q,k_{qf}), Nb_2) - s_2(h_2-1) < bot_f(A,p,j'_{pf}) \\ \qquad\quad \leq \min\{nxt(bot_f(A,p,j'_{pf}), top_a(C,q,k_{qf}), Nb_2), top_a(C,q,k_{ql})\}; \\ \phi, \quad \text{otherwise.} \end{cases}$$

$$rbody_C^2(p,q) = f_2 f_1^{-1}\Big([[[nxt(bot_a(A,p,j'_{pf}), bot_f(C,q,k_{qf}+1), Ns_1h_2) : \\ nxt(bot_a(A,p,j'_{pf}), bot_f(C,q,k_{qf}+1), Ns_1h_2) + s_1(h_2-1) : s_1] : \\ top_a(A,p,j'_{pf}) : Ns_1h_2] : u_1 : Nb_1]\Big).$$

$$recv_C(p,q) = rhead_C(p,q) \cup rbody_C^1(p,q) \cup rbody_C^2(p,q).$$

5.4 The Case Where 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, 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 let h_2 be 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 will 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 $f_C(bot_f(A,p,j_{pf})) = f_C(top_f(A,p,j_{pf}))$, $((f_C(top_f(A,p,j_{pf})) - 1) \bmod N)$ 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 let h_1 be 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 will 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 $f_A(bot_f(C,p,k_{pf})) = f_A(top_f(C, p,k_{pf}))$, $((f_A(top_f(C,p,k_{pf})) - 1) \bmod N)$ is not in $send\_pe(p)$. Second, when $u_1 - l_1 + 1 < Nb_1$ and $f_A(bot_f(C,p,k_{pl})) = f_A(top_f(C,p,k_{pl}))$, $((f_A(top_f(C,p,k_{pf})) + (k_{pl} - k_{pf})N/h_1) \bmod N)$ is not in $send\_pe(p)$.

6 Experimental Studies

In this section, we will present three experimental studies implemented on a 16-node nCUBE/2E parallel computer. In 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 compared pros and cons of three proposed algorithms: the row-wise version described in Section 3, the lattice method in Section 4, and the closed-form version in Section 5. We adopted two communication models: first, a *conventional model* that only packs data values of RHS array elements into send buffers and generates corresponding addresses of LHS array entries at the receiving end. Second, a *deposit model*, which was also suggested by [6] [36], that packs elements using an *address-value* pair before sending, where *value* is the value of a RHS array element and *address* is the corresponding address of a LHS array entry. After that, at the receiving end, there is no need to unpack messages, and PEs use message buffers for the combined received-execute phase. This method, however, will incur additional communication time because the size of each message is doubled. The second experimental study calculated a saxpy operation on two data arrays, and the third experimental study performed a data re-distribution operation on a specific data array, both based on the closed-form version algorithm using the conventional communication model. In effect, the data re-distribution operation can be seen as a special case of the saxpy operation.

6.1 Comparisons of Three Proposed Algorithms

We compare the three proposed algorithms using the following benchmark code:

> **forall** $i = 0,\ 80639$
>
> $\quad A(1997 + i * s_1) = C(5 + i * s_2),$

where array A is distributed by a $cyclic(b_1)$ distribution and array C is distributed by a $cyclic(b_2)$ distribution. Table 6 and Table 7 list experimental results of implementing this forall statement with various block sizes b_1 and b_2 as well as strides s_1 and s_2 on 16 PEs. Note that in this experimental study, we only present the cases where $\lceil \frac{b_1}{s_1} \rceil \leq \lceil \frac{(N-1)b_2+1}{s_2} \rceil$, and where N is the number of PEs; the other cases where $\lceil \frac{b_2}{s_2} \rceil \leq \lceil \frac{(N-1)b_1+1}{s_1} \rceil$ can be presented in a similar way. The experimental results can be summarized as follows.

				row-wise method		lattice method		closed-form method	
b_1	b_2	(I)	(II)	conventional	deposit	conventional	deposit	conventional	deposit
8	5	15	3	116 (110)	126 (116)	97 (90)	112 (105)		
†9	6	1	1	63 (48)	67 (45)	60 (49)	68 (48)	59 (47)	71 (49)
10	7	21	3	115 (109)	123 (112)	86 (80)	101 (94)		
62	43	129	9	292 (287)	212 (200)	91 (85)	103 (94)		
†63	42	1	1	35 (34)	52 (49)	35 (34)	52 (50)	34 (32)	53 (50)
64	41	123	9	282 (276)	206 (195)	92 (86)	104 (95)		
314	209	47	42	54 (51)	56 (49)	70 (67)	66 (59)		
†315	210	1	1	33 (29)	43 (35)	33 (29)	43 (37)	32 (28)	43 (35)
316	211	47	42	54 (50)	57 (49)	71 (68)	67 (60)		
314	2519	47	474	124 (118)	127 (117)	1113 (1108)	622 (617)		
†315	2520	12	1	58 (52)	86 (76)	43 (38)	75 (68)	37 (32)	71 (62)
316	2521	47	474	127 (121)	129 (118)	1145 (1140)	670 (660)		
3779	209	3	42	56 (50)	84 (73)	1338 (1333)	774 (767)		
†3780	210	1	1	49 (44)	79 (68)	50 (45)	79 (72)	37 (31)	71 (60)
3781	211	3	42	56 (50)	84 (73)	1286 (1280)	742 (736)		
3779	2521	3	57	39 (37)	62 (58)	305 (303)	205 (203)		
†3780	2520	1	1	38 (36)	61 (57)	38 (36)	61 (59)	28 (26)	55 (51)
3781	2519	3	57	38 (36)	61 (57)	294 (292)	199 (197)		

Table 6: Experimental study 1 when $s_1 = 3$ and $s_2 = 2$. The simulation time, "execution time (pure computation time)," of solving the forall statement on 16 PEs is expressed in units of milliseconds. *communication overhead = execution time − pure computation time.* Factor (I) = $\min\{j_{pl} - j_{pf}, period^A_{sb}\}$ and factor (II) = $\min\{j_{pl} - j_{pf}, period^A_{eb}\} * \lceil period_s/(period^A_e * N)\rceil$. Cases indicated by † have closed-form expressions.

1. As shown in Table 6, all the studies computed the same forall statement with strides $s_1 = 3$ and $s_2 = 2$. However, the execution time varied quite a bit for different block sizes b_i. In

addition, total execution time is dominated by generating indices and by packing and unpacking of messages.

2. If block sizes satisfy conditions having closed-form expressions, then the execution time is better than in cases with similar block sizes but without closed-form expressions. In addition, it is better to use the conventional communication model and to generate LHS indices at the receiving end in these cases; this is due to the simplicity of closed-form expressions. We will study how to choose a suitable granularity size if block sizes satisfy conditions having closed-form expressions for Experimental studies 2 and 3 again.

3. In the following, we will study cases without closed-form expressions; therefore, these cases are implemented using the row-wise version algorithm and the lattice method. Factor (I) = $\min\{j_{pl} - j_{pf}, period^A_{sb}\}$, which represents the complexity of the row-wise version algorithm, indicates that $send_C(p, q)$ can be represented by a union of Factor (I) number of closed forms. Factor (II) = $\min\{j_{pl} - j_{pf}, period^A_{eb}\} * \lceil period_s/(period^A_e * N)\rceil$, which represents the complexity of the lattice method, indicates that the algorithm in Fig. 9 or 10 has to be run $\min\{j_{pl} - j_{pf}, period^A_{eb}\}$ times for computing $send_C(p, q)$, and for each time roughly $\lceil period_s/(period^A_e * N)\rceil$ lattice points will be retrieved in the first period. We find that, if factor (I) $>$ 1.5 $*$ factor (II), then the lattice method is more effective than the row-wise version algorithm; otherwise, if factor (I) $<$ 1.5 $*$ factor (II), then the row-wise version algorithm is more effective. The other observation is that, when block sizes are small, the lattice method is better because factor (II) is relatively small; when block sizes are large, the row-wise version algorithm is better because factor (I) is relatively small.

4. If factor (I) or factor (II) is large, which means that the cost of generating indices is high, then the deposit communication model to pack messages by address-value pairs is more effective; on the other hand, if factor (I) or factor (II) is small, then the conventional communication model is more effective. The threshold value depends on problem sizes, strides, and block sizes. In this experimental study, the threshold value of the row-wise method is around factor (I) = 50; the threshold value of the lattice method is around factor (II) = 40. The other observation is that, when strides $<$ block sizes and block sizes are small, the deposit communication model should not be used because a lot of block-boundary indices of RHS array entries have to be changed to

corresponding indices of LHS array entries in the code generation phase, which need to compute

$$\text{Indices of } A = g2l_A\{f_1 f_2^{-1}[l2g_C(\text{indices of } C \text{ at } PE_p)]\},$$

where $l2g_C(i,p) = (\lfloor i/b_2 \rfloor * N + p) * b_2 + (i \bmod b_2) + c_1$ means the function of transforming an index of array C at PE_p from a local name space to a global name space; $g2l_A(i) = \lfloor (i - a_1)/(N * b_1) \rfloor * b_1 + ((i - a_1) \bmod b_1)$ means the function of transforming an index of array A from a global name space to a local name space.

5. In Table 7, for cases where strides > block sizes, the lattice method is always better than the row-wise version algorithm; in addition, the deposit communication model is more effective than the conventional communication model for almost all cases. This is because elements in the sets $send_C(p,q)$ and $recv_C(p,q)$ are sparse; thus, it is better to combine the computation of both the indices of the RHS array entries and the corresponding indices of the the LHS array entries at the sending end.

						row-wise method				lattice method			
s_1	b_1	s_2	b_2	(I)	(II)	conventional		deposit		conventional		deposit	
3	2	3	2	3	3	105	(93)	98	(76)	104	(91)	97	(75)
7	4	9	7	49	7	104	(101)	85	(79)	52	(48)	50	(45)
7	5	5	3	21	7	88	(85)	82	(75)	62	(58)	63	(56)
9	5	7	6	54	9	104	(101)	83	(78)	52	(49)	50	(46)
9	7	7	4	36	9	88	(85)	73	(68)	54	(50)	51	(47)
11	7	8	5	55	11	127	(124)	116	(110)	62	(58)	63	(57)
23	19	11	7	161	23	330	(327)	201	(193)	81	(77)	66	(59)

Table 7: Experimental study 1 when strides s_i > block sizes b_i.

6.2 Saxpy Operation

We study the effectiveness of different block sizes using the following benchmark code, which performs a saxpy operation:

forall $i = 0,\ 80639$

$$A(1997 + i * 3) = A(1997 + i * 3) + saxpy\_con * C(5 + i * 2),$$

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 8 lists the experimental results for implementing this saxpy operation with various block sizes, b_1 and b_2, using the closed-form version algorithm. The experimental results can be summarized as follows.

b_2	b_1	3	9	63	315	945	3780	7560	15120
2	2 PE	876	1241	879	829	821	821	821	821
	4 PE	438	412	476	428	420	417	417	417
	8 PE	220	207	274	225	217	214	213	213
	16 PE	110	108	173	125	117	113	113	295
6	2 PE	1238	541	564	516	508	508	506	506
	4 PE	407	271	318	270	262	259	259	258
	8 PE	205	136	125	145	137	134	134	133
	16 PE	106	68	65	85	77	74	73	134
42	2 PE	875	582	349	380	371	369	368	367
	4 PE	472	326	175	199	193	189	189	189
	8 PE	270	125	88	89	102	99	99	98
	16 PE	172	65	45	47	51	57	56	64
210	2 PE	826	534	395	309	347	349	349	348
	4 PE	426	278	209	169	161	179	179	179
	8 PE	223	149	89	85	84	94	94	94
	16 PE	124	87	47	43	44	48	54	55
630	2 PE	819	527	387	357	297	344	345	345
	4 PE	419	271	201	162	184	177	177	177
	8 PE	216	141	107	84	93	83	93	93
	16 PE	116	79	52	44	47	44	46	53
2520	2 PE	819	525	384	363	359	307	336	337
	4 PE	417	268	198	188	187	155	160	175
	8 PE	213	139	103	99	84	78	82	83
	16 PE	113	76	59	49	44	40	43	44
5040	2 PE	818	524	383	362	358	361	299	335
	4 PE	416	268	197	187	185	163	151	160
	8 PE	213	138	103	98	99	83	76	85
	16 PE	113	76	58	56	46	42	39	45
10080	2 PE	817	522	380	361	357	358	358	295
	4 PE	415	268	196	186	184	187	164	149
	8 PE	212	138	102	97	97	84	86	75
	16 PE	293	135	66	57	56	44	44	38

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

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

2. When h' was less than the number of PEs, the execution time became better when h' was close to 1. This is because, in these cases, each block of array C in PE_p ($[bot_l(C,p,k) : top_l(C,p,k)]$) intersected with 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, some optimization could be obtained by using two-nested closed forms to represent $send_C(p,q)$ and $recv_C(p,q)$ instead of using the proposed 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 needed to send data messages to at most $(h'+1)$ PEs. Therefore, the communication time was reduced when h' became smaller.

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

4. All the cases except three showed scalable improvement when the number of PEs grew. Three exception cases were when the number of PEs was 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 was significant; in addition, the communication overhead also became worse when the number of PEs grew because of certain *all-to-all* communications.

5. Because the iteration space was linear and each PE executed roughly the same number of iterations, there was no load unbalance problem. Therefore, according to the communication oracle, it was preferable to choose large block sizes b_1 and b_2. From Table 8, we can summarize that it is preferable to choose block sizes $b_1 \geq 63$ and $b_2 \geq 42$ for this saxpy operation.

6. The cases where $b_1/3 = b_2/2$ ran faster than did other cases where $b_1 \geq 63$ and $b_2 \geq 42$. This result is consistent with the suggestion concerning the algorithm in Section 5.1.

6.3 Data Re-distribution

Consider the following data re-distribution operation:

forall $i = 0,\ 241919$

$$A(i) = OLD\_A(i),$$

where array A is distributed by a $cyclic(b_1)$ distribution; array OLD_A is distributed by a $cyclic(b_2)$ distribution. Table 9 lists the experimental results of implementing this data re-distribution operation with various block sizes, b_1 and b_2. The experimental results show that the behavior of the execution time of this data re-distribution operation was similar to that of the saxpy operation. From Table 9, we can summarize that it is preferable to choose block sizes $b_1 \geq 63$ and $b_2 \geq 63$ for this data re-distribution operation.

b_2 \ b_1		3	9	63	315	945	3780	7560	15120
3	2 PE	0	1652	1292	1244	1236	1233	1233	1233
	4 PE	0	627	688	639	631	630	628	629
	8 PE	0	322	378	331	323	320	319	319
	16 PE	0	170	225	177	169	166	165	165
9	2 PE	1660	0	930	892	885	883	882	882
	4 PE	642	0	505	462	455	452	452	452
	8 PE	331	0	223	241	234	232	231	231
	16 PE	173	0	115	132	124	121	121	121
63	2 PE	1299	938	0	712	714	716	716	716
	4 PE	692	509	0	366	369	369	368	368
	8 PE	382	228	0	181	190	189	189	189
	16 PE	227	118	0	90	99	100	100	100
315	2 PE	1251	898	718	0	673	692	693	694
	4 PE	644	464	371	0	343	355	356	357
	8 PE	333	243	186	0	182	182	182	183
	16 PE	178	132	96	0	107	93	96	97
945	2 PE	1243	891	721	678	0	680	686	688
	4 PE	636	458	372	353	0	345	352	353
	8 PE	325	236	192	185	0	178	178	180
	16 PE	170	124	99	99	0	112	92	94
3780	2 PE	1240	888	723	699	688	0	668	679
	4 PE	633	455	372	359	352	0	367	346
	8 PE	322	233	191	184	183	0	240	178
	16 PE	167	122	101	96	96	0	93	111
7560	2 PE	1240	888	723	700	693	677	0	667
	4 PE	633	455	372	360	355	358	0	366
	8 PE	322	232	191	184	181	203	0	239
	16 PE	167	122	101	97	94	103	0	119
15120	2 PE	1239	887	722	701	695	686	675	0
	4 PE	632	454	371	361	357	350	356	0
	8 PE	321	232	190	185	183	182	205	0
	16 PE	167	122	101	98	96	94	103	0

Table 9: Execution time (millisecond) of performing the data re-distribution operation using 2 PEs, 4 PEs, 8 PEs, and 16 PEs, respectively. Array A was distributed by a $cyclic(b_1)$ distribution; array OLD_A was distributed by a $cyclic(b_2)$ distribution.

In the above three 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 the 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 than expected.

7 Related Work

Koelbel and Mehrotra first provided closed-form representations for special cases where $l_1 = 0$ and $s_1 = 1$, and where arrays are distributed in *block* or *cyclic* distributions [23, 25]. The following researchers were concerned with *block-cyclic* ($cyclic(b_i)$) distributions; however, none of them obtained 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 exist closed forms to represent communication sets for *cyclic* distributions, communication sets for *block-cyclic* distributions can be represented by a union of $b_1 * b_2$ closed forms [36]. 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. The virtual-block (or virtual-cyclic) approach views a *block-cyclic* distribution as a *block* (or *cyclic*) distribution on a set of virtual processors, which are then cyclically (or block-wise) mapped to the physical processors [12, 13]. The virtual-block approach is suitable for cases where block sizes are large; the virtual-cyclic approach is suitable for cases where block sizes are small. Kaushik *et al.* extended the virtual processor approach to array statements involving arrays aligned with distributed template arrays and mapped using a two-level mapping [19]. The above two approaches did not uncover periodic patterns in communication sets. Benkner *et al.*, instead, utilizing periodic properties, also proposed a technique similar to [19] which was implemented in their Vienna Fortran compiler [3] and Prepare HPF compiler [4].

The following researchers derived communication sets based on their proposed algorithms for computing the memory access sequence of $A(l_1 : u_1 : s_1)$ in each PE, where array A is distributed by $cyclic(b_1)$. Chatterjee *et al.* enumerated the local memory access sequence based on a finite-state machine (FSM). Their run-time algorithm involves a solution of b_1 linear Diophantine equations to determine the pattern of accessed addresses, followed by sorting of these addresses to derive the accesses in a linear order. The time complexity of determining the first period of accessed addresses is $O(b_1 \log b_1 + \log(\min\{Nb_1, s_1\}))$, which is dominated by the sorting phase, and $O(\log(\min\{Nb_1, s_1\}))$ time is needed to perform an extended-Euclid algorithm, which is used to solve b_1 linear Diophantine equations. To generate communication sets, each PE makes a single pass over the RHS data in its local memory using the FSM technique, determines the destination of each data element, and packs elements using an address-value pair [6]. Their approach, however, requires an explicit local-to-global and global-to-local index translation for each referenced address-value pair. In addition, the computation phase and the communication phase of their method cannot be overlapped because communication between PEs can take place only after all the data to be sent has been packed into the send buffers, and the receive-execute step can be performed only after each PE has received all the communication sets.

Hiranandani *et al.* also presented algorithms which were based on an FSM for computing the local memory access sequence. They did not sort accessed addresses; instead, they constructed a memory-access-gap table and solved additional b_1 linear Diophantine equations to determine a starting point. After that, the memory access sequence can be enumerated in linear time. The time complexity of constructing their memory-access-gap table and of determining a starting point is $O(b_1 + \log(\min\{Nb_1, s_1\}))$. To calculate communication sets, they used a scanning technique similar to the merge sort to compute the intersection of two reference patterns corresponding to the LHS and the RHS array subscripts [14]. Their methods, however, incur certain run-time overheads due to indirect addressing of data.

Kennedy *et al.* adopted an integer lattice method to generate the memory access sequence. They solved $O(b_1/\gcd(Nb_1, s_1))$ linear Diophantine equations to determine the distance vectors R_v and L_v; after that, the memory access sequence could be enumerated in a linear time [21, 22]. The time complexity of deriving the distance vectors is $O(b_1/\gcd(Nb_1, s_1) + \log(\min\{Nb_1, s_1\}))$. We notice that the pair of distance vectors found by Kennedy *et al.* is the best. However, they did not provide closed-form expressions of the distance vectors for certain interesting cases. To compute communication sets, they went through one pass over the locally owned RHS (LHS) data using their integer lattice method; then, they packed elements into send (receive) buffers according to a processor table [20]. According to the results of their experimental studies, the table construction overhead of their technique is significantly smaller than that incurred by the virtual processor approach [13], which incurs substantial overhead in mapping communication sets from virtual processors to physical processors. Their method, however, like that in [6], cannot overlap the computation phase and the communication phase.

Thirumalai *et al.* presented closed-form expressions for distance vectors for certain cases while deriving the memory access sequence [39]. According to their experimental study, they were able to improve the execution time when $s_1 \leq b_1$; however, when stride s_1 is larger than block size b_1, their method may be worse than the methods in [21, 22]. This is because their method cannot always find the best pair of distance vectors for certain cases. To deal with communication sets, they only handled a special case where $b_1 = b_2 = b$. They found that the send pattern of the RHS array repeated after every $r_2 = r * s_2/(Nb)$ rows, and that the access pattern of the LHS array repeated after every $r_1 = r * s_1/(Nb)$ rows, where $r = \mathrm{lcm}(Nb/\gcd(s_1, Nb), Nb/\gcd(s_2, Nb))$. Each PE scans the memory access sequence of the RHS (and LHS) array of the first r_2 (and r_1) rows to accumulate send (and receive) sets [40, 43]. They also proposed course (row) padding and column padding techniques to enhance the data locality of references [44]. Their method, however, requires additional memory to store processor indices, addresses and corresponding data. In addition, as in [6], the computation phase and the communication phase cannot be overlapped.

Furthermore, Ancourt *et al.* [1], van Dongen [41] and Le Fur *et al.* [10] expressed the communication sets and the iteration sets as sets of integer linear constraints, which correspond to polyhedrons. Then, the execution of generated code consists in scanning these polyhedrons. Midkiff formulated the local iteration set by means of linear Diophantine equations, which, then, are converted as a nested loop, the bounds of which have closed-form expressions [32]. His method, however, requires computation of all the loop bounds, even when some bounds may be not necessary when $strides > blocksizes$. van Reeuwijk *et al.* also presented a technique, based on resolution of the associated linear Diophantine equations, to illustrate row-wise and column-wise data allocation and addressing schemes [42]. Coelho *et al.* [7] discussed the pros and cons of using closed forms, FSM or the integer lattice method, and polyhedron theory.

For experimental studies, Wang *et al.* [46] presented a comprehensive study of the run-time performance of the code generated from three classes of published algorithms: linear algebraic methods [32], table-driven methods [6, 20, 21, 22, 39], and set-theoretic methods [12, 36]. Their conclusion is that for the array assignment statement $A(0 : n*s_1 : s_1) = B(0 : n*s_2 : s_2)$, the best rule of thumb is to use the LSU [39] algorithm for small block sizes, and the OSU [12] algorithm for large block sizes. In addition, Li and Chen proposed methods to generate aggregate communication operations based on pattern matching techniques [30]. Wu presented an algebraic transformation framework which allows a compiler to optimize data movement for a sequence of Do loops at an abstract level without going into machine-dependent details [48]. Wolfe gave a detailed tutorial for message-passing machines [47].

Next, for the special case where the parameters $a_1 = c_1$, $a_2 = c_2$, $l_1 = l_2 = 0$, and $s_1 = s_2 = 1$, the target problem is reduced to a data re-distribution problem. Research on this data re-distribution problem also has been reported [16] [17] [18] [33] [37] [38] [45].

7.1 Comparison with Two-level Mapping Model

It is instructive to illustrate that under the two-level mapping model, there are no closed-form expressions for communication sets for arbitrary accessed strides; and under our model, we can represent communication sets by closed forms.

Two-level mapping model:

For instance, HPF provides directives which allow programmers to specify the data distribution. Consider the following directives:

	REAL $A_1(a_{11} : a_{12}), \ldots, A_n(a_{n1} : a_{n2})$
!HPF$	PROCESSORS $PES(N)$
!HPF$	TEMPLATE $T(:)$
!HPF$	ALIGN $A_1(i)$ WITH $T(d_1 * i + e_1)$
$\vdots$	$\vdots$
!HPF$	ALIGN $A_n(i)$ WITH $T(d_n * i + e_n)$
!HPF$	DISTRIBUTE $T(cyclic(b))$ ONTO PES.

In the first level, array element $A_j(i)$ is aligned to a template cell $d_j * i + e_j$, where d_j is called an alignment factor and e_j is called an alignment offset. In the second level, the template is distributed across N PEs using a $cyclic(b)$ distribution. Then, elements of array A_j are mapped onto processors according to this two-level mapping.

The corresponding local address for element $A_j(i)$ is summarized in Table 10. Readers can check that one requires six parameters (i, d_j, e_j, N, b, and t) for referencing one element $A_j(i)$ under the two-level mapping model, where $t = \min_{(\forall j)}\{d_j * a_{j1} + e_j\}$, which is the smallest index of the template [47]. However, we only require four parameters (i, a_{j1}, N, and b_j) for referencing one element $A_j(i)$ under our model. Thus, the two-level mapping model is more complicated than our method. Furthermore, under the two-level mapping model, the numbers of elements between two local blocks may be different, which prevents use of closed-form representations for communication sets.

local address	two-level mapping	our method
processor ID	$\lfloor((d_j * i + e_j - t) \bmod (bN))/b\rfloor$	$\lfloor((i - a_{j1}) \bmod (b_j N))/b_j\rfloor$
block	$\lfloor(d_j * i + e_j - t)/(bN)\rfloor$	$\lfloor(i - a_{j1})/(b_j N)\rfloor$
offset	$\lfloor((d_j * i + e_j - t) \bmod b)/d_j\rfloor$	$(i - a_{j1}) \bmod b_j$
max block size	$\lceil b/d_j\rceil$	b_j

Table 10: The corresponding local address for element $A_j(i)$ based on two models, where $t = \min_{(\forall j)}\{d_j * a_{j1} + e_j\}$, which is the smallest index of the template.

For an array assignment statement $A_1(l_1 : u_1 : s_1) = A_2(l_2 : u_2 : s_2)$, closed-form expressions for communication sets exist only when $b/(d_1 * s_1)$ is a multiple of $b/(d_2 * s_2)$ or when $b/(d_1 * s_1)$ is a factor of $b/(d_2 * s_2)$. That is, b must be a multiple of both $d_1 * s_1$ and $d_2 * s_2$; in addition, either $d_1 * s_1$ is a factor of $d_2 * s_2$, or $d_1 * s_1$ is a multiple of $d_2 * s_2$. However, $d_1 * s_1$ and $d_2 * s_2$ generally do not have any factor or multiple relationship between each other. That is, for arbitrary accessed strides s_1 and s_2, closed-form representations for communication sets are not guaranteed. As shown in experimental studies described in Section 6, if communication sets cannot be represented by closed forms, the software overhead due to packing and unpacking of communication sets is high.

Our model:

Unlike the two-level mapping model, this paper only considers cases which can be interpreted as each array A_j is aligned to a separate template. For example, the first array element $A_j(a_{j1})$ is aligned to the first element of the corresponding template. When dealing with HPF, if the alignment factor d_j is equal to 1, there has a one-to-one correspondence between elements of the physical array A_j and cells of the virtual template array. However, if the alignment factor d_j is not equal to 1, there are *holes* for a factor of $d_j - 1$ when the physical array A_j is aligned to the virtual template array. The offset alignment can be improved in a preprocessing phase as follows. Let $t = \min_{(\forall j)}\{d_j * a_{j1} + e_j\}$. If the alignment factor d_j is equal to 1, we can extend the left boundary of array A_j from a_{j1} to $a'_{j1} = t - e_j$. That is, we can extend array $A_j(a_{j1} : a_{j2})$ to $A_j(t - e_j : a_{j2})$, so that the first array element $A_j(t - e_j)$ is aligned to the first template cell $T(t)$.

On the other hand, if the alignment factor d_j is not equal to 1, we can extend the left boundary of array A_j from a_{j1} to $a'_{j1} = (a_{j1} - \lfloor(d_j * a_{j1} + e_j - t)/d_j\rfloor)$. That is, we can extend array $A_j(a_{j1} : a_{j2})$ to $A_j(a_{j1} - \lfloor(d_j * a_{j1} + e_j - t)/d_j\rfloor : a_{j2})$, so that the first array element $A_j(a_{j1} - \lfloor(d_j * a_{j1} + e_j - t)/d_j\rfloor)$ is aligned to the template cell $T(d_j * a_{j1} + e_j - d_j * \lfloor (d_j * a_{j1} + e_j - t)/d_j \rfloor)$, which is very close to the first template cell $T(t)$. Note that, the additional boundary array elements $A_j(a'_{j1} : a_{j1} - 1)$ need not be allocated physical memory space. Since most alignment constraints are satisfied, especially, for those arrays whose alignment factors d_j are equal to 1, thus, communication overhead may be reduced.

After the preprocessing phase, depending on accessed strides s_j, block sizes b_j can be determined by compilers as indicated in Section 5.1, where the closed-form conditions in Table 4 can be changed to a set of more restricted ones that $b_1/(d_1 * s_1)$ *is a factor or a multiple of* $b_2/(d_2 * s_2)$. (For instance, let $s'_1 = d_1 * s_1$ and $s'_2 = d_2 * s_2$ in Table 4.) Note that, the closed-form conditions shown in Table 4 have already guaranteed that communication sets can be represented by closed-form expressions. The new sufficient conditions, which emphasize that $b_1/(d_1 * s_1)$ is a factor or a multiple of $b_2/(d_2 * s_2)$, are dealing with alignments having arbitrary alignment factors d_1 and d_2. Note that our model is powerful enough to deal with the ScaLAPACK library [9], in which all alignment factors d_j are equal to 1.

8 Conclusions

We have presented three methods for deriving communication sets, all three of which utilize periodical properties of communication sets. The first and the second methods deal with cases where data arrays are distributed in the most general regular data distribution. The first method adopts row-wise block-to-block intersections; the second method adopts an integer lattice method. But none of them can derive communication sets using a constant number of closed forms. The third method emphasizes that compilers can assign suitable block sizes for data distribution, so that communication sets can be represented using a constant number of closed forms. For example, $send_C(p, q)$ can be represented by the union of at most **three** closed-form expressions with at most **eight** boundary unknowns.

We have carried out experimental studies on a 16-node nCUBE/2E parallel computer. The results of these experimental studies support the idea that block sizes should be determined by compilers; then, software overhead for generating communication sets will not be significant. As for cases where block sizes are arbitrary, each of the proposed row-wise version method and the lattice method has its special niche, as has been summarized in Section 6.1. However, these two methods require from 35% up to 240% more software overhead than does the (third) closed-form version method for similar block sizes.

In order to give an easy-to-understand presentation, although in this paper we derived communication sets using the global name space, it was straightforward to map these sets to corresponding sets using the local addresses when we implemented experimental studies on a 16-node nCUBE/2E parallel computer. Our experimental studies also showed that the indexing overhead of the proposed methods scaled well as the number of PEs increased. Our first and third methods, basically, are row-wise approaches, which are especially efficient when block sizes are (not too small) medium size or large. However, if block sizes are very small, then the method of Stichnoth *et al.* [36] is recommended.

We have studied array assignment statements in this paper. If the alignments and distributions of each dimension in a multi-dimensional array are independent of one another, extension of our approach to multi-dimensional arrays is straightforward. For instance, suppose that the two-dimensional data arrays A and C are distributed on an $N \times N$ processor mesh by $(cyclic(b_1), cyclic(b_2))$ and $(cyclic(b_3), cyclic(b_4))$, respectively. Then, for a two dimensional array assignment statement $A(l_1 : u_1 : s_1,\ l_2 : u_2 : s_2) = C(l_3 : u_3 : s_3,\ l_4 : u_4 : s_4)$, $send_C((p_1, p_2), (q_1, q_2)) = send_{C_1}(p_1, q_1) \times send_{C_2}(p_2, q_2)$, where $send_{C_k}(p_k, q_k)$ means send sets in order to perform the array assignment statement $A_k(l_k : u_k : s_k) = C_k(l_{k+2} : u_{k+2} : s_{k+2})$, for $k = 1$ or 2. A_k and C_k are the k-th dimensions of A and C, respectively; and $'\times'$ is a Cartesian product operator.

References

[1] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static high performance Fortran code distribution. *Scientific Programming*, 6:3–27, 1997.

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

[3] S. Benkner. Handling block-cyclic distributed arrays in Vienna Fortran 90. In *Proc. International Conference on Parallel Architecures and Compilation Techniques (PACT-95)*, Limassol, Cyprus, June 1995.

[4] 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.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21:15–26, 1994.

[6] 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.

[7] F. Coelho, C. Germain, and J.-L. Pazat. State of the art in compiling HPF. In U. Banerjee et al., editor, *Proc. Sixth International Workshop on Languages and Compilers for P arallel Computing*, 1997.

[8] P. Crooks and R. H. Perrott. Language constructs for data partitioning and distribution. *Scientific Programming*, 4:59–85, 1995.

[9] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, June 1995.

[10] M. Le Fur, J.-L. Pazat, and F. André. An array partitioning analysis for parallel loop distribution. In *Lecture Notes in Computer Science 966, International Conference EURO-PAR'95 Parallel Processing*, pages 351–364, Stockholm, Sweden, Aug. 1995.

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

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

[13] S. 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.

[14] 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.

[15] 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.

[16] E. T. Kalns and L. M. Ni. Processor mapping techniques toward efficient data redistribution. *IEEE Trans. Parallel Distributed Syst.*, 6(12):1234–1247, Dec. 1995.

[17] 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.

[18] 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.

[19] S. D. Kaushik, C. H. Huang, and P. Sadayappan. Efficient index set generation for compiling HPF array statements on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 38(2):237–247, 1996.

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

[21] K. Kennedy, N. Nedeljković, 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.

[22] K. Kennedy, N. Nedeljković, 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*, pages 149–158, Santa Barbara, CA, July 1995.

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

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

[25] 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.

[26] H. T. Kung and J. Subhlok. A new approach for automatic parallelization of blocked linear algebra computations. In *Proc. of Supercomputing '91*, pages 122–129, Albuquerque, NM, Nov. 1991.

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

[28] P.-Z. Lee. Efficient algorithms for data distribution on distributed memory parallel computers. *IEEE Trans. Parallel Distributed Syst.*, 8(8):825–839, Aug. 1997.

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

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

[31] 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.

[32] S. Midkiff. Local iteration set computation for block-cyclic distributions. In *Proc. International Conf. on Parallel Processing*, pages II–77–84, Aug. 1995.

[33] S. Ramaswamy, B. Simons, and P. Banerjee. Optimizations for efficient array redistribution on distributed memory multicomputers. *Journal of Parallel and Distributed Computing*, 38(2):217–228, 1996.

[34] Z. Shen, Z. Li, and P. C. Yew. An empirical study on array subscripts and data dependences. In *Proc. International Conf. on Parallel Processing*, pages II–145–152, St. Charles, IL, Aug. 1989.

[35] Z. Shen, Z. Li, and P. C. Yew. An empirical study of fortran programs for parallelizing compilers. *IEEE Trans. Parallel Distributed Syst.*, 1(3):356–364, July 1990.

[36] 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.

[37] 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.

[38] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Trans. Parallel Distributed Syst.*, 7(6):587–594, June 1996.

[39] A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing*, 38(2):188–203, 1996.

[40] A. Thirumalai, J. Ramanujam, and A. Venkatachar. Communication generation and optimization for HPF. In B. Szymanski and B. Sinharoy, editors, *Prof. Third Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, pages 311–316, Troy, NY, May 1995. Kluwer Academic Publishers.

[41] V. van Dongen. Compiling distributed loops onto SPMD code. *Parallel Processing Letters*, 4(3):301–312, 1994.

[42] K. van Reeuwijk, W. Denissen, H. J. Sips, and E. M. R. M. Paalvas t. An implementation framework for HPF distributed arrays on message-pas sing parallel computer systems. *IEEE Trans. Parallel Distributed Syst.*, 7(9):897–914, Sep. 1996.

[43] A. Venkatachar, J. Ramanujam, and A. Thirumalai. Communication generation for block-cyclic distribution. *Parallel Processing Letters*, 7(2):195–202, 1997.

[44] A. Venkatachar, J. Ramanujam, and A. Thirumalai. Generalized overlap regions for communication optimization in data-parallel programs. In U. Banerjee et al., editor, *Proc. Tenth International Workshop on Languages and Compilers for Parallel Computing*, 1997.

[45] A. Wakatani and M. Wolfe. Optimization of array redistribution for distributed memory multicomputers. *Parallel Computing*, 21:1485–1490, 1995.

[46] L. Wang, J. M. Stichnoth, and S. Chatterjee. Runtime performance of parallel array assignment: An empirical study. In *Proc. of Supercomputing '96*, Pittsburgh, PA, Nov. 1996, on available via WWW http://www.supercomp.org/sc96/proceedings/SC96PROC/CHATTER/INDEX.HTM.

[47] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.

[48] J.-J. Wu. *Optimization and Transformation Techniques for High Performance Fortran.* PhD thesis, Yale University, 1995.

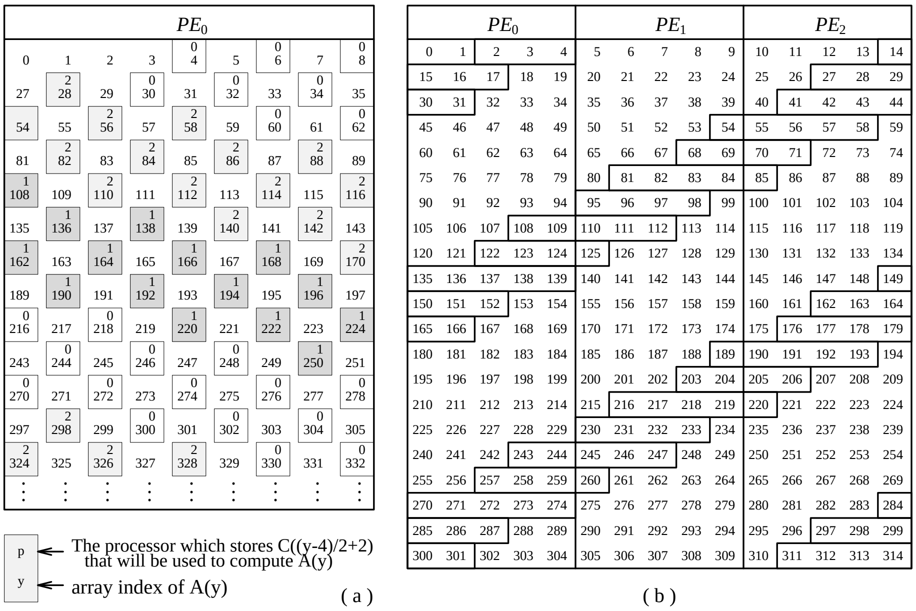


Figure 5: (a) The memory access sequence of $A(4 + i * 2)$, for $i \geq 0$, by PE_0. (b) $send_C(p, 0)$, for $0 \leq p \leq 2$, which represent elements of array C and will be sent to PE_0.

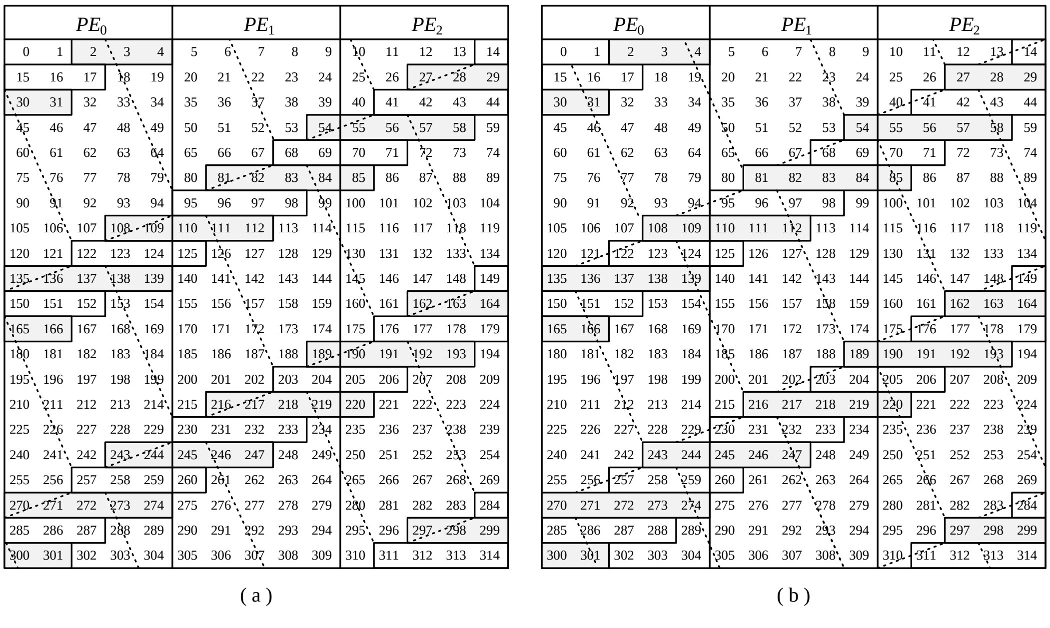


Figure 6: (a) $[bot_f(A, 0, 1) : u_2 : period_e^A * s_2] = [14 : 314 : 27]$ forms a lattice. (b) $[bot_f(A, 0, 2) : u_2 : period_e^A * s_2] = [27 : 314 : 27]$ forms a lattice.