Generating Global Name-Space Communication Sets for Array Assignment Statements1

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

1This 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.

<table>
<thead>
<tr>
<th>array assignment statements</th>
<th>forall statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A(l_1:u_1:s_1) = g(C(l_2:u_2:s_2)))</td>
<td>forall (i = {0, \frac{u_1-s_1}{b} } )</td>
</tr>
<tr>
<td>(A(l_1+i<em>s_1: l_1+i</em>s_1 + \left[ \frac{u_1-s_1}{b} \right] s<em>s_1 : s</em>s_1) = g(C(l_2: l_2+i<em>s_2 : l_2+i</em>s_2 + \left[ \frac{u_2-s_2}{b} \right] s<em>s_2 : s</em>s_2)))</td>
<td>forall (i = l, u, s)</td>
</tr>
<tr>
<td>(A(l_1+i<em>s_1) = g(C(l_2+l</em>i*s_2)))</td>
<td>(A(l_1+i<em>s_1) = g(C(l_2+l</em>i*s_2)))</td>
</tr>
</tbody>
</table>

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:

\[
\text{forall } i = 0, \left\lfloor \frac{u_1 - l_1}{s_1} \right\rfloor \quad A(l_1 + i * s_1) = g(C(l_2 + i * s_2)).
\]

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 $b = \frac{m}{N + 2}$ or $b = \frac{m}{N + 2}$ 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 $\frac{m}{N + 2}$ or $\frac{m}{N + 2}$ 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 \(j = 1, i\)) \(A(i, j) = \alpha \ast (C(i, j) + C(i - 1, j) + C(i + 1, j) + C(i, j - 1) + C(i, j + 1))\);
(\[do\ \(i = 1, m\)) (do \(j = 1, i\)) \(C(i, j) = \beta \ast (A(i, j) + A(i - 1, j) + A(i + 1, j) + A(i, j - 1) + A(i, j + 1))\);

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

diff = \(\frac{9m}{N}\), which means the difference in the work load between \(PE_0\) and \(PE_{N-1}\);

ratio = \(\frac{m}{m-N+1}\), which means the ratio of the work load between \(PE_0\) and \(PE_{N-1}\);

comm = \(2m(\frac{N}{mN} + 1 - \frac{1}{N} + \frac{1}{4m})\), which means the communication overhead in \(PE_0\).

<table>
<thead>
<tr>
<th>(b)</th>
<th>(2^1)</th>
<th>(2^2)</th>
<th>(2^3)</th>
<th>(2^4)</th>
<th>(2^5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff</td>
<td>1.88 (\cdot 2^{11})</td>
<td>1.88 (\cdot 2^{12})</td>
<td>1.88 (\cdot 2^{13})</td>
<td>1.88 (\cdot 2^{14})</td>
<td>1.88 (\cdot 2^{15})</td>
</tr>
<tr>
<td>ratio</td>
<td>1.01</td>
<td>1.02</td>
<td>1.06</td>
<td>1.06</td>
<td>1.12</td>
</tr>
<tr>
<td>comm</td>
<td>1.01 (\cdot 2^{19})</td>
<td>1.02 (\cdot 2^{18})</td>
<td>1.03 (\cdot 2^{17})</td>
<td>1.06 (\cdot 2^{16})</td>
<td>1.12 (\cdot 2^{15})</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>(b)</th>
<th>(2^1)</th>
<th>(2^2)</th>
<th>(2^3)</th>
<th>(2^4)</th>
<th>(2^5)</th>
<th>(2^6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff</td>
<td>1.97 (\cdot 2^{22})</td>
<td>1.97 (\cdot 2^{21})</td>
<td>1.97 (\cdot 2^{25})</td>
<td>1.97 (\cdot 2^{26})</td>
<td>1.97 (\cdot 2^{31})</td>
<td>1.97 (\cdot 2^{32})</td>
</tr>
<tr>
<td>ratio</td>
<td>1.00</td>
<td>1.00</td>
<td>1.06</td>
<td>1.06</td>
<td>1.13</td>
<td>1.28</td>
</tr>
<tr>
<td>comm</td>
<td>1.00 (\cdot 2^{35})</td>
<td>1.00 (\cdot 2^{34})</td>
<td>1.03 (\cdot 2^{26})</td>
<td>1.06 (\cdot 2^{25})</td>
<td>1.12 (\cdot 2^{24})</td>
<td>1.25 (\cdot 2^{23})</td>
</tr>
</tbody>
</table>

(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 may not 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, $[1 : 102] = \{1, 2, 3, \ldots, 102\}$.

- $[a : e_1 : s_1]$ 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 $\{[a : e_1], [a : e_1] + s_2, [a : e_1] + 2s_2, \ldots, \}$ until not greater
  than $e_2$. Thus, $[[1 : 30] : 102 : 40] = \{1, 2, 3, \ldots, 30, 41, 42, 43, \ldots, 70, 81, 82, 83, \ldots, 102\}$.

- $[[a : e_1 : s_1] : e_2 : s_2]$ means the set $\{[a : e_1 : s_1], [a : e_1 : s_1] + s_2, [a : e_1 : s_1] + 2s_2, \ldots,$
  until not greater than $e_2$. Thus, $[[1 : 30] : 102 : 40] = \{1, 11, 21, 41, 51, 61, 81, 91, 101\}$.


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 $A([a_1 + p \ast b : a_1 +
+ p \ast b + b - 1] : a_2 : N \ast b]$] 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 $b = [(a_2 - a_1 + 1)/N]$; and in a block-cyclic fashion if
$1 < b < [(a_2 - a_1 + 1)/N]$.

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, $nxt(x, y, z) = x + ((y - x) \mod z)$. 

5
3 Generation of Communication Sets for Array Assignments

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

3.1 Structure of Generated Code

<table>
<thead>
<tr>
<th>Code on processing element ( p ) (( PE_p ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Generate iteration sets and processor sets:</td>
</tr>
<tr>
<td>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 \cdot b_1 : a_1 + p \cdot b_1 + b_1 - 1] : a_2 : N \cdot b_1] );</td>
</tr>
<tr>
<td>1.2 ( send_{pc}(p) = {q</td>
</tr>
<tr>
<td>1.3 ( recv_{pc}(p) = {q</td>
</tr>
<tr>
<td>2. ( \forall q \in send_{pc}(p), \text{ do} )</td>
</tr>
<tr>
<td>2.1 ( send_C(p, q) = local_C(p) \cap f_2(\text{exec}(q)) ), which represents elements sent from ( PE_p ) to ( PE_q ), where ( local_C(p) = [[c_1 + p \cdot b_2 : c_1 + p \cdot b_2 + b_2 - 1] : c_2 : N \cdot b_2] );</td>
</tr>
<tr>
<td>2.2 ( send ) message containing ( send_C(p, q) ) to ( PE_q );</td>
</tr>
<tr>
<td>3. perform computations for iterations in ( \text{iter}(p, p) ), where ( \text{iter}(p, p) = f_2^{-1}(local_C(p) \cap [l_2 : u_2 : s_2]) \cap \text{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;</td>
</tr>
<tr>
<td>4. ( \forall q \in recv_{pc}(p), \text{ do} )</td>
</tr>
<tr>
<td>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 );</td>
</tr>
<tr>
<td>4.2 ( \text{iter}(p, q) = f_2^{-1}(local_C(q) \cap [l_2 : u_2 : s_2]) \cap \text{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 );</td>
</tr>
<tr>
<td>4.3 execute computations for iterations in ( \text{iter}(p, q) ).</td>
</tr>
</tbody>
</table>

Figure 1: Outline of implementing an array assignment statement.

Fig. 1 shows a detailed outline of the implementation of an array assignment statement (for all 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 \( exe(x, p) \) in Substep 1.1 is only formulated to derive other communication sets and processor sets. Since \( exe(x, p) = \text{iter}(p, p) \cup \bigcup_{q \in exe(x, p)} \text{iter}(p, q) \) and \( \text{iter}(p, q) = f_2^{-1}(recv_C(p, q)) \), 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 \( exe(x, p) \) and \( \text{iter}(p, q) \) 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, \( local_A(p) = \text{cyclic}(a_1 + pb_1 : a_1 + pb_1 + b_1 - 1) : a_2 : Nb_1 \) is the first block data of \( A \) in \( PE_p \). The subscript \( 'l' \) means local data; \( 'a' \) means accessed data; \( 'c' \) means iterations to be executed; and \( 'j' \) means the corresponding data between array \( A \) and array \( C \).

Let \( j_{pf} \) and \( j_{pl} \) be the first \( j \) and the last \( j \) such that \( bot(A, p, j) : top(A, p, j) \cap [l_1 : u_1 : s_1] \neq \phi \), respectively; and let \( k_{pf} \) and \( k_{pl} \) be the first \( k \) and the last \( k \) such that \( bot(C, p, k) : top(C, p, k) \cap [l_2 : u_2 : s_2] \). Fig. 1 shows an algorithm for computing \( j_{pf} \) and \( j_{pl} \). \( k_{pf} \) and \( k_{pl} \) also can be computed similarly. In Fig. 1, the value \( j_{\text{start}} = [(l_1 - a_1 - pb_1 - b_1 + 1) / Nb_1] \) is the first \( j \) such that \( top(A, p, j) \geq l_1 \). The value \( j_{\text{final}} = [(u_1 - a_1 - pb_1) / Nb_1] \) is the last \( j \) such that \( bot(A, p, j) \leq u_1 \). If \( s_1 \leq b_1 \), then \( j_{\text{start}} = j_{pf} \) and \( j_{\text{final}} = j_{pl} \). 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. 1 is \( O(s_1 / \text{gcd}(Nb_1, s_1)) \). 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 \( O(b_1 / \text{gcd}(Nb_1, s_1)) \) linear Diophantine equations. According to our experiments, in a majority of cases, the algorithm in Fig. 1 was more efficient than was solving \( O(b_1 / \text{gcd}(Nb_1, s_1)) \) linear Diophantine equations.
\[ \text{bot}(A, p, j) = a_1 + pb_1 + jNb_1 \]
\[ \text{top}(A, p, j) = a_1 + pb_1 + b_1 - 1 + jNb_1 \]
\[ \text{bot}_s(A, p, j) = \text{nxt}(\max\{\text{bot}(A, p, j), l_1\}, l_1, s_1) \]
\[ \text{top}_s(A, p, j) = \text{nxt}(\min\{\text{top}(A, p, j), u_1\} - s_1 + 1, l_1, s_1) \]
\[ \text{bot}_t(A, p, j) = (\text{bot}_s(A, p, j) - l_1) / s_1 \]
\[ \text{top}_t(A, p, j) = (\text{top}_s(A, p, j) - l_1) / s_1 \]
\[ \text{top}_f(A, p, j) = \text{top}_t(A, p, j)s_2 + l_2 \]
\[ \text{bot}_f(A, p, j) = \text{bot}_t(A, p, j)\text{top}_f(A, p, j) - l_1 \]
\[ \text{bot}(C, p, k) = c_1 + pb_2 + kNb_2 \]
\[ \text{top}(C, p, k) = c_1 + pb_2 + b_2 - 1 + kNb_2 \]
\[ \text{bot}_s(C, p, k) = \text{nxt}(\max\{\text{bot}(C, p, k), l_2\}, l_2, s_2) \]
\[ \text{top}_s(C, p, k) = \text{nxt}(\min\{\text{top}(C, p, k), u_2\} - s_2 + 1, l_2, s_2) \]
\[ \text{bot}_t(C, p, k) = (\text{bot}_s(C, p, k) - l_2) / s_2 \]
\[ \text{top}_t(C, p, k) = (\text{top}_s(C, p, k) - l_2) / s_2 \]
\[ \text{bot}_f(C, p, k) = \text{bot}_t(C, p, k)s_2 + l_2 \]
\[ \text{top}_f(C, p, k) = \text{top}_t(C, p, k)s_2 + l_2. \]

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:

\[ \text{local}_A(p) = \bigcup_{j=0}^{\frac{N - s_1 + 1}{s_1}} \{\text{bot}(A, p, j) : \text{top}(A, p, j)\} \]
\[ \text{exec}(p) = f_1^{-1}(\text{local}_A(p) \cap [l_1 : u_1]) \]
\[ = f_1^{-1}\left(\bigcup_{j=0}^{u_1/p} \{\text{bot}(A, p, j) : \text{top}(A, p, j) : s_1\}\right) \]
\[ = \bigcup_{j=0}^{u_1/p} \{\text{bot}(A, p, j) : \text{top}(A, p, j) : s_1\}. \]

Note that, in the expression \([\text{bot}(A, p, j) : \text{top}(A, p, j)]\), it may happen that \(\text{bot}_t(A, p, j) > \text{top}_t(A, p, j)\) when \(s_1 > b_1\). Throughout this paper, if \(a > \beta\), then \([a : \beta]\) is empty. Next, according to the order of appearance in Fig. 1, after deriving exec\((p)\), we should present the processor sets send\(_p\)\((p)\) and recv\(_p\)\((p)\). 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.
\[ j_{\text{start}} = \left\lfloor \left( l_1 - a_1 - p b_1 - b_1 + 1 \right)/\left( N b_1 \right) \right\rfloor; \]
\[ j_{\text{final}} = \left\lfloor \left( a_1 - a_1 - p b_1 \right)/\left( N b_1 \right) \right\rfloor; \]
if \((s_1 \leq b_1)\) then
\[ j_{pf} = j_{\text{start}}; \]
\[ j_{pl} = j_{\text{final}}; \]
else \((s_1 > b_1)\) \}
\[ j = j_{\text{start}}; \]
while \((j \leq j_{\text{final}})\) do
\[ j_{pf} = j; \]
break;
else
\[ j = j + 1; \]
endif
endwhile

if \((j > j_{\text{final}})\) then
\[ \text{exec}(p) = \phi; \]
else \(* j_{pf} \leq j_{\text{final }} *\)
\[ j = j_{\text{final}}; \]
while \((j \geq j_{pf})\) do
\[ \text{if } (\text{bot}_a(A, p, j) \leq \text{top}_a(A, p, j)) \]
\[ j_{pl} = j; \]
break;
else
\[ j = j - 1; \]
endif
endwhile
endif
endif
endwhile

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 \(f_2(\text{exec}(q))\), which will be used in deriving \(send_C(p, q) = local_C(p) \cap f_2(\text{exec}(q))\):

\[
f_2(\text{exec}(q)) = \bigcup_{j=j_{pf}}^{j_{pl}} f_2([\text{bot}_c(A, q, j) : \text{top}_c(A, q, j)])
\]
\[
= \bigcup_{j=j_{pf}}^{j_{pl}} [\text{bot}_c(A, q, j)s_2 + l_2 : \text{top}_c(A, q, j)s_2 + l_2 : s_2]
\]
\[
= \bigcup_{j=j_{pf}}^{j_{pl}} [\text{bot}_f(A, q, j) : \text{top}_f(A, q, j) : s_2].
\]

We now define the periodic coefficients of the communication set \(send_C(p, q)\). Let \(\text{period}^A_p\) be the period of the iteration pattern in \(\text{exec}(p)\) such that \(\text{period}^A_p * s_1\) is a multiple of \(N b_1\); let \(\text{period}^A_{eb}\) be the number of blocks of local elements of array \(A\) whose access pattern appears periodically; let \(\text{period}^C_{eb}\) be the number of blocks of local elements of array \(C\) whose access pattern appears periodically; let \(\text{period}_s\) be the period of the reference pattern of array \(C\) in \(send_C(p, q)\) whose value is a multiple of \(N b_2\); let \(\text{period}^C_{sb}\) be the number of blocks of local elements of array \(C\) whose reference pattern in \(send_C(p, q)\) appears periodically; and let \(\text{period}^A_{sb}\) 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 \(f_2(\text{exec}(q))\)) appears
periodically. Then, we have the following equations:

\[
\text{period}_a^A = (\text{lcm}(Nb_1, s_1))/s_1; \quad \text{period}_e = \text{lcm}(Nb_2, \text{period}_e^A + s_2);
\]
\[
\text{period}_b^A = (\text{lcm}(Nb_1, s_1))/(Nb_1);
\]
\[
\text{period}_e^B = \text{period}_e/(Nb_2);
\]
\[
\text{period}_b^B = (\text{period}_b + s_1)/(Nb_1s_2).
\]

We will now study the intersection of local\(_C(p) \cap f_2(exeC(q))\), which is equal to \(\bigcap_{k=bp(n)}^b [\text{bot}_f(C, p, k) : \text{top}_f(C, p, k)] \cap \left( \bigcup_{j=qf}^f \left[ \text{bot}_f(A, q, j) : \text{top}_f(A, q, j) : s_2 \right] \right)\). We found that if \(\frac{b_1}{s_1} \leq \frac{(N-1)b_2+1}{s_2}\), then each referenced block of array \(A\) in \(PE_q (\{ \text{bot}_f(A, q, j) : \text{top}_f(A, q, j) : s_2 \})\) will intersect with at most one local block of array \(C\) in \(PE_p (\{ \text{bot}_f(C, p, k) : \text{top}_f(C, p, k) \})\). Similarly, if \(\frac{b_2}{s_2} \leq \frac{(N-1)b_1+1}{s_1}\), 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 \(N \geq 2\), at least one of the following two conditions is true: (a) \(\frac{b_1}{s_1} \leq \frac{(N-1)b_2+1}{s_2}\) and (b) \(\frac{b_2}{s_2} \leq \frac{(N-1)b_1+1}{s_1}\).

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

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 \([[a : a + b - 1] : e : Nb] \cap [a : \beta : \gamma]\), respectively. Suppose that \(\frac{a+b+1}{\gamma} \leq \frac{(N-1)b+1}{\gamma}\). Then,

\([[a : a + b - 1] : e : Nb] \cap [a : \beta : \gamma] = \langle L : R : \gamma \rangle\),

where

\[
L = \begin{cases} 
\alpha, & \text{if } \alpha \in [[a : a + b - 1] : e : Nb] \\
\text{nxt}(\text{nxt}(\max\{a, \alpha\}, a, Nb), \alpha, \gamma), & \text{otherwise};
\end{cases}
\]

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

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

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

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

10
Since $\left\lfloor \frac{\beta - a + 1}{\gamma} \right\rfloor \leq \left\lfloor \frac{N - 1}{\gamma} \right\rfloor$, $[a : \beta : \gamma]$ will intersect with at most one local block of $[[a : a + b - 1] : e : Nb]$. Thus, $[[a : a + b - 1] : e : Nb] \cap [a : \beta : \gamma] = [nxt(\alpha, \gamma) : nxt(R' - \gamma + 1, \alpha, \gamma) : \gamma] = [L : R : \gamma]$.

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 $\left\lfloor \frac{b}{s_1} \right\rfloor \leq \left\lfloor \frac{N-1}{s_2} \right\rfloor$, $send_C(p, q)$ can be represented as follows:

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

$$= [[c_1 + pb_2 : c_1 + pb_2 + b_2 - 1] : c_2 : Nb2] \cap \left( \bigcup_{j=1}^{\min\{\beta, j+1\}} [bot_j(A, q, j) : top_j(A, q, j) : s_2] \right)$$

$$= \bigcup_{j=1}^{\min\{\beta, j+1\}} \left( [[c_1 + pb_2 : c_1 + pb_2 + b_2 - 1] : c_2 : Nb2] \cap [bot_j(A, q, j) : top_j(A, q, j) : s_2] \right)$$

$$= \left[ L(j) : R(j) : s_2 \right] \cup \left( \bigcup_{j=1}^{\min\{\beta, j+1\}} \left[ L(j) : R(j) : s_2 \right] : u_2 : period_s \right)$$

where

$$L(j) = \begin{cases} bot_j(A, q, j), & \text{if } bot_j(A, q, j) \in local_C(p) \\ nxt(nxt(\max\{c_1 + pb_2, bot_j(A, q, j)\}, c_1 + pb_2, N\beta2), l_2, s_2), & \text{otherwise}; \end{cases}$$

$$R(j) = \begin{cases} top_j(A, q, j), & \text{if } top_j(A, q, j) \in local_C(p) \\ nxt(nxt(\min\{c_1 + pb_2, top_j(A, q, j)\}, c_1 + pb_2, N\beta2), -N\beta2 + b_2 - s_2, l_2, s_2), & \text{otherwise}. \end{cases}$$

Second, if $\left\lfloor \frac{b}{s_2} \right\rfloor \leq \left\lfloor \frac{N-1}{s_1} \right\rfloor$, $send_C(p, q)$ can be represented as follows:

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

$$= f_2 f_1^{-1} \left( f_1 f_2^{-1} (f_2(exce(q)) \cap local_C(p)) \right)$$

$$= f_2 f_1^{-1} \left( \left[ a_1 + q b_1 : a_1 + q b_1 + b_1 - 1 \right] : a_2 : N\beta2 \right] \cap \left( \bigcup_{k=1}^{\min\{k, p, q\}} [bot_j(C, p, k) : top_j(C, p, k) : s_1] \right)$$

$$= \bigcup_{k=1}^{\min\{k, p, q\}} f_2 f_1^{-1} \left( \left[ a_1 + q b_1 : a_1 + q b_1 + b_1 - 1 \right] : a_2 : N\beta2 \right] \cap [bot_j(C, p, k) : top_j(C, p, k) : s_1]$$

$$= \left[ f_2 f_1^{-1} (L(k)) : f_2 f_1^{-1} (R(k)) : s_2 \right]$$

$$= \left[ f_2 f_1^{-1} (L(k)) : f_2 f_1^{-1} (R(k)) : s_2 \right] \cup \left( \bigcup_{k=1}^{\min\{k, p, q\}} \left[ f_2 f_1^{-1} (L(k)) : f_2 f_1^{-1} (R(k)) : s_2 \right] : u_2 : period_s \right)$$

$$= \left[ f_2 f_1^{-1} (L(k)) : f_2 f_1^{-1} (R(k)) : s_2 \right] \cup \left( \bigcup_{k=1}^{\min\{k, p, q\}} \left[ f_2 f_1^{-1} (L(k)) : f_2 f_1^{-1} (R(k)) : s_2 \right] : u_2 : period_s \right).$$
where

$$L(k) = \begin{cases} \text{bot}_f(C, p, k), & \text{if } \text{bot}_f(C, p, k) \in \text{local}_A(q) \\ \text{nx}_f(\text{nx}_f(\max\{a_1 + q_b, \text{bot}_f(C, p, k)\}, a_1 + q_b, N b_1), l_1, s_1), & \text{otherwise}; \end{cases}$$

$$R(k) = \begin{cases} \text{top}_f(C, p, k), & \text{if } \text{top}_f(C, p, k) \in \text{local}_A(q) \\ \text{nx}_f(\text{nx}_f(\min\{a_2, \text{top}_f(C, p, k)\}, a_1 + q_b, N b_1) - N b_1 + b_1 - s_1, l_1, s_1), & \text{otherwise}. \end{cases}$$

Next, we deal with $\text{recv}_C(p, q)$. Because $\text{recv}_C(p, q)$ is equal to $\text{send}_C(q, p)$, $\text{recv}_C(p, q)$ also can be represented by the union of a functional number of closed forms. Although $\text{recv}_C(p, q)$ specifies a set of indices of array $C$, in practice, we prefer that $\text{recv}_C(p, q)$ be represented based on indices of array $A$. For instance, the loop body of the forall statement $A(f_i(i)) = g(C(f_2(i))$ is equivalent to $A(f_i(i)) = g(C(f_2 f_1^{-1}(f_i(i))))$. 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 $f_2 f_1^{-1}(\text{recv}_C(p, q))$ because $\text{recv}_C(p, q) = f_2 f_1^{-1}(f_1 f_2^{-1}(\text{recv}_C(p, q)))$. Since the derivation of $\text{recv}_C(p, q)$ is similar to that of $\text{send}_C(p, q)$, we omit all of the middle steps and only present the final formulas.

First, if $[\frac{b_1 + 1}{s_1}] \leq [\frac{N - 1}{s_2}]$, $\text{recv}_C(p, q)$ can be represented as follows:

$$\text{recv}_C(p, q) = f_2 f_1^{-1}(f_1 f_2^{-1}(\text{recv}_C(p, q))) = f_2 f_1^{-1}(f_1 f_2^{-1}(\text{send}_C(q, p)))$$

$$= f_2 f_1^{-1}\left([f_1 f_2^{-1}(L(j)) : f_1 f_2^{-1}(R(j)) : s_1] \cup \left(\bigcup_{j=pj}^{\min(a_1 + q_b p + \text{period}^A_{jk})} \left[[f_1 f_2^{-1}(L(j)) : f_1 f_2^{-1}(R(j)) : s_1] : u_1 : \text{period}_s * s_1 / s_2\right]\right)\right)$$

$$= f_2 f_1^{-1}\left([f_1 f_2^{-1}(L(j)) : f_1 f_2^{-1}(R(j)) : s_1] \cup \left(\bigcup_{j=pj}^{\min(a_1 + q_b p + \text{period}^A_{jk})} [f_1 f_2^{-1}(L(j)) : f_1 f_2^{-1}(R(j)) : s_1] : u_1 : \text{period}_s * s_1 / s_2\right)\right),$$

where

$$L(j) = \begin{cases} \text{bot}_f(A, p, j), & \text{if } \text{bot}_f(A, p, j) \in \text{local}_C(q) \\ \text{nx}_f(\text{nx}_f(\max\{c_1 + q_b, \text{bot}_f(A, p, j)\}, c_1 + q_b, N b_2), l_2, s_2), & \text{otherwise}; \end{cases}$$

$$R(j) = \begin{cases} \text{top}_f(A, p, j), & \text{if } \text{top}_f(A, p, j) \in \text{local}_C(q) \\ \text{nx}_f(\text{nx}_f(\min\{c_2, \text{top}_f(A, p, j)\}, c_1 + q_b, N b_2) - N b_2 + b_2 - s_2, l_2, s_2), & \text{otherwise}. \end{cases}$$

Second, if $[\frac{b_2}{s_2}] \leq [\frac{N - 1}{s_1}]$, $\text{recv}_C(p, q)$ can be represented as follows:

$$\text{recv}_C(p, q) = f_2 f_1^{-1}(f_1 f_2^{-1}(\text{recv}_C(p, q))) = f_2 f_1^{-1}(f_1 f_2^{-1}(\text{send}_C(q, p)))$$

$$= f_2 f_1^{-1}\left([L(k_f) : R(k_f) : s_1]\right) \cup \left(\bigcup_{j=pj}^{\min(c_1 + q_b p + \text{period}^A_{jk})} [L(k_f) : R(k_f) : s_1] : u_1 : \text{period}_s * s_1 / s_2\right),$$

where

$$L(k_f) = \begin{cases} \text{bot}_f(A, p, k_f), & \text{if } \text{bot}_f(A, p, k_f) \in \text{local}_C(q) \\ \text{nx}_f(\text{nx}_f(\max\{c_1 + q_b, \text{bot}_f(A, p, k_f)\}, c_1 + q_b, N b_2), l_1, s_1), & \text{otherwise}; \end{cases}$$

$$R(k_f) = \begin{cases} \text{top}_f(A, p, k_f), & \text{if } \text{top}_f(A, p, k_f) \in \text{local}_C(q) \\ \text{nx}_f(\text{nx}_f(\min\{c_2, \text{top}_f(A, p, k_f)\}, c_1 + q_b, N b_2) - N b_2 + b_2 - s_2, l_1, s_1), & \text{otherwise}. \end{cases}$$
\[
\left( \bigcup_{k=k_p+f+1}^{\min\{k_p+f+1, + \text{period}_C}\}} [L(k) : R(k) : s_1] : u_1 : \text{period}_s \times s_1/s_2)] \right) \\
= f_2f_1^{-1} \left( [L(k) : R(k) : s_1] \cup \left( \left( \bigcup_{k=k_p+f+1}^{\min\{k_p+f+1, + \text{period}_C}\}} [L(k) : R(k) : s_1] : u_1 : \text{period}_s \times s_1/s_2)] \right) \right),
\]

where

\[
L(k) = \begin{cases} 
\text{bot}_f(C, q, k), & \text{if } \text{bot}_f(C, q, k) \in \text{local}_A(p) \\
\text{nxt}(\text{nxt}(\max\{a_1 + pb_1, \text{bot}_f(C, q, k)\}, a_1 + pb_1, Nb_1), l_1, s_1), & \text{otherwise};
\end{cases}
\]

\[
R(k) = \begin{cases} 
\text{top}_f(C, q, k), & \text{if } \text{top}_f(C, q, k) \in \text{local}_A(p) \\
\text{nxt}(\text{nxt}(\min\{a_2, \text{top}_f(C, q, k)\}, a_1 + pb_1, Nb_1) - Nb_1 + b_1 - s_1, l_1, s_1), & \text{otherwise}.
\end{cases}
\]

### 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 \( \text{send}_C(p, q) \neq \phi \) or whether \( \text{send}_C(q, p) \neq \phi \), 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(b₁); \( f_A(i) = \left( \frac{i-1}{b_1} \right) \mod N \), 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

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

Property 3 also holds for array \( C \) with its corresponding distribution by cyclic(b₂) and its data distribution function \( f_C \). We now process send\_pe(p), which is equal to \( f_A(f_1(f_2^{-1}(\text{local}_C(p) \cap [l_2 : u_2 : s_2])))) \):

\[
\text{send\_pe(p)} = f_A(f_1(f_2^{-1}(\text{local}_C(p) \cap [l_2 : u_2 : s_2])))
\]

\[
= \bigcup_{k=k_p+f}^{k_p+f+1} f_A(f_1([\text{bot}_f(C, p, k) : \text{top}_f(C, p, k)]))
\]

\[
= \bigcup_{k=k_p+f}^{\min\{k_p+f+1, + \text{period}_C\}} f_A([\text{bot}_f(C, p, k) : \text{top}_f(C, p, k) : s_1])
\]

\[
\subseteq \bigcup_{k=k_p+f}^{\min\{k_p+f+1, + \text{period}_C\}} f_A([\text{bot}_f(C, p, k) : \text{top}_f(C, p, k) : s_1]).
\]
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 $fc( f_2(\text{exec}(p)))$:

$$recv_{pe}(p) = fc( f_2(\text{exec}(p)))$$

$$= \bigcup_{j=pf}^{j=pl} fc( f_2([\text{bot}_s(A, p, j) : \text{top}_s(A, p, j)]))$$

$$= \bigcup_{j=pf}^{\min \{j \text{pl}, jpf + \text{period}_{A,b} \}} fc( [\text{bot}_f(A, p, j) : \text{top}_f(A, p, j) : s_2])$$

$$\subseteq \bigcup_{j=pf}^{\min \{j \text{pl}, jpf + \text{period}_{A,b} \}} fc( [\text{bot}_f(A, p, j) : \text{top}_f(A, p, j)])$$

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 \((period_{sb}^1 + 1)\) or \((period_{sb}^C + 1)\) closed forms. However, as one can see from a preliminary example in Figure 5, for many cases, \(L(j) > R(j)\) for some \(j \in [j_{bf} + 1 : \min\{j_{bf}, j_{bf} + period_{sb}^1\}]\); therefore, \([[L(j) : R(j) : s_2] : u_2 : \text{period}_{sb}\] is an empty set. Similarly, for many cases, \(I(k) > R(k)\) for some \(k \in [k_{pf} + 1 : \min\{k_{pf}, k_{pf} + period_{sb}^C\}]\); therefore, \([[f_{2f_{1}}^{-1}(I(k)) : f_{2f_{1}}^{-1}(R(k)) : s_2] : u_2 : \text{period}_{sb}\] 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, \(x = (i - a_1) / (Nb_1), y = ((i - a_1) \mod (Nb_1)), \) and \(p = (((i - a_1) / b_1) \mod N)\). 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 \( (i/5) \mod 4 \) corresponds to a two-dimensional index \( (i/20, i \mod 20) \in \mathbb{Z}^2 \). (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:

\[
\begin{align*}
R_v & \quad \text{if } y + b_v \leq pb_1 + b_1 - 1; \\
L_v & \quad \text{if } y + b_v > pb_1 + b_1 - 1 \text{ and } y + b_v \geq pb_1; \\
R_v + L_v & \quad \text{otherwise.}
\end{align*}
\]

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:

\[
R \quad \text{if } pb_1 \leq go_{\text{right}} \leq pb_1 + b_1 - 1; \\
L \quad \text{if } (\text{not} (pb_1 \leq go_{\text{right}} \leq pb_1 + b_1 - 1)) \text{ and } (pb_1 \leq go_{\text{left}} \leq pb_1 + b_1 - 1); \\
R + L \quad \text{otherwise},
\]

where \( go_{\text{right}} = ((i - a_1 + R) \mod (Nb_1)) \) and \( go_{\text{left}} = ((i - a_1 + L) \mod (Nb_1)) \).  

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:

\[
R \quad \text{if } (lb \leq go_{\text{right}} \leq rb); \\
L \quad \text{if } (\text{not} (lb \leq go_{\text{right}} \leq rb)) \text{ and } (lb \leq go_{\text{left}} \leq rb); \\
R + L \quad \text{otherwise},
\]

where \( go_{\text{right}} = ((i - a_1 + R) \mod (Nb_1)) \) and \( go_{\text{left}} = ((i - a_1 + L) \mod (Nb_1)) \).

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:

$$
R \quad \text{if} \quad (\text{not } (rb < go\text{right} < lb)); \\
L \quad \text{if} \quad (rb < go\text{right} < lb) \text{ and } (\text{not } (rb < go\text{left} < lb)); \\
R + L \quad \text{otherwise},
$$

where $go\text{right} = ((i - a_1 + R) \mod (Nb_1))$; and $go\text{left} = ((i - a_1 + L) \mod (Nb_1))$. \hfill \Box

### 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 $s_1 \leq b_1$, each block contains at least one accessed address. In addition, the memory access sequence can be represented by a union of $(\text{period}_r^A + 1)$ closed forms, where $\text{period}_r^A = (s_1 / \gcd(Nb_1, s_1)) \leq (b_1 / \gcd(Nb_1, s_1))$. Thus, in this case, it is better to use $(\text{period}_r^A + 1)$ 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 $s_1 \leq b_1$

$$
local_A(p) \cap [l_1 : u_1 : s_1] \\
= \bigcup_{j=p}^{j_1} [bot_a(A, p, j) : top_a(A, p, j) : s_1] \\
= [bot_a(A, p, j_1f) : top_a(A, p, j_1f) : s_1] \cup \\
\left( \bigcup_{j=j_1f+1}^{\min\{j_1f, j_1f + \text{period}_r^A\}} [bot_a(A, p, j) : top_a(A, p, j) : s_1] : u_1 : \text{period}_r^A + s_1 \right) \\
= [bot_a(A, p, j_1f) : top_a(A, p, j_1f) : s_1] \cup \\
\left( \bigcup_{j=j_1f+1}^{\min\{j_1f, j_1f + \text{period}_r^A\}} [bot_a(A, p, j) : top_a(A, p, j) : s_1] : u_1 : \text{period}_r^A + s_1 \right). 
$$
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 \( j_{b1} \) can be computed in constant time. Since the memory access sequence can be represented by a union of \((\text{period}_{cb}^A + 1)\) closed forms, the number of time units of the calculating boundary coefficients is \( O(\text{period}_{cb}^A) = O(s_1 / \gcd(Nb_1, s_1)) \).

### 4.2.2 Cases Where \((N - 1)b_1 < (s_1 \mod Nb_1) < Nb_1\)

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 \((N - 1)b_1 < (s_1 \mod Nb_1) < Nb_1\), index \( s_1 \) appears in column \((s_1 \mod Nb_1)\) in \( PE_{N-1}\); index \( 2s_1 \) appears in column \( 2(s_1 \mod Nb_1) - Nb_1\); index \( 3s_1 \) appears in column \( 3(s_1 \mod Nb_1) - 2Nb_1\); and so on. Suppose that \( y \) is the smallest integer such that \( ye(s_1 \mod Nb_1) - (y - 1)*Nb_1 < b_1 \); then, \( x = y + s_1 \). We have \( y = [(Nb_1 - b_1 + 1)/(Nb_1 - (s_1 \mod Nb_1))] \) and \( R = x = [(Nb_1 - b_1 + 1)/(Nb_1 - (s_1 \mod Nb_1))] * s_1 \).

Second, since \((N - 1)b_1 < (s_1 \mod Nb_1) < Nb_1\), index \((\text{period}_{cb}^A - 1) * s_1 \) appears in \( PE_0 \). Therefore, the largest index in the initial cycle accessed by \( PE_0 \) is \((\text{period}_{cb}^A - 1) * s_1 \). Thus, we have \( L = \text{period}_{cb}^A * s_1 - (\text{period}_{cb}^A - 1) * s_1 = s_1 \).

### 4.2.3 Cases Where \(0 < (s_1 \mod Nb_1) < b_1\)

These cases are dual cases where \((N - 1)b_1 < (s_1 \mod Nb_1) < Nb_1\), and they include interesting cases where \( s_1 = yNb_1 + 1 \) for every integer \( y \geq 1 \). First, since \( 0 < (s_1 \mod Nb_1) < b_1 \), 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 \( 0 < (s_1 \mod Nb_1) < b_1 \), index \((\text{period}_{cb}^A - 1) * s_1 \) appears in column \( Nb_1 - (s_1 \mod Nb_1) \) in \( PE_{N-1} \); index \((\text{period}_{cb}^A - 2) * s_1 \) appears in column \( Nb_1 - 2(s_1 \mod Nb_1) \); and so on. Suppose that \( y \) is the smallest integer such that \( Nb_1 - y * (s_1 \mod Nb_1) < b_1 \); then, \( x = (\text{period}_{cb}^A - y) * s_1 \). We have \( y = [(Nb_1 - b_1 + 1)/(s_1 \mod Nb_1)] \) and \( L = \text{period}_{cb}^A * s_1 - x = y * s_1 = [(Nb_1 - b_1 + 1)/(s_1 \mod Nb_1)] * s_1 \).
Input: $a_1, a_2$ (the range of an array $A(a_1 : a_2)$), $b_1, u_1, s_1$ (the parameters of the access pattern $A(l_1 : u_1 : s_1)$), $N$ (number of PE's), $b_2$ (block size), and $p$ (a processing element ID). Output: The $\Delta M$ table.

1. \#define nxt(x, y, z) = x + ((y - x) \mod z);
2. \(d, x, y\) \rightarrow \text{EXTENDED-EUCLID}(\text{Nb}_1, s_1);
   \{\text{* } d = \gcd(\text{Nb}_1, s_1) = x \* \text{Nb}_1 + y \* s_1. \text{* }\}
3. \text{period}_s^d = \text{Nb}_1 / d;
4. \text{lp} = a_1 + p \* b_2; \text{ rp} = a_1 + p \* b_2 + b_1 - 1;

Step 1: \{\text{* Handle the special cases where } s_1 \leq b_1. \text{* }\}
5. \text{if } (s_1 \leq b_1) \text{ return } \Delta M =
6. \{\text{bot}_a(A, p, j_0, j_f) : \text{top}_a(A, p, j_0, j_f) : s_1\}
7. \{\min\{j_{p, j_0, j_f} + \text{period}_s^d\} \text{bot}_a(A, p, j) : \text{top}_a(A, p, j) : s_1\}
8. \text{top}_a(A, p, j) : s_1\};

Step 2: \{\text{* Check whether }\Delta M \text{ is empty or not. }\text{* }\}
9. \text{if } (\text{nxt}(\text{lp} - l_1, d, d) > \text{rp} - l_1) \text{ return } \Delta M = \phi;

Step 3: \{\text{* Find the starting point accessed by } PE_p. \text{* }\}
10. \text{start} = \infty; \text{ length} = 0;
11. \text{for } i = \text{nxt}(\text{lp} - l_1, d, d), \text{rp} - l_1, d
12. \text{loc} = l_1 + (s_1 \* \text{d})(i \* \text{Nb}_1 \* \text{Nt}y/(\text{Nb}_1));
13. \text{start} = \text{min}(\text{start}, \text{loc});
14. \text{length} = \text{length} + 1;
15. \text{endfor}

Step 4: \{\text{* Derive distance vectors } R \text{ and } L. \text{* }\}
16. \text{if } ((N - 1) \* \text{b}_1 < (s_1 \text{ mod } \text{Nb}_1) < \text{Nt}b_1) \text{ then}
17. \text{R} = [(N \* \text{b}_1 - b_1 + 1)/(N \* \text{b}_1 - (s_1 \text{ mod } \text{Nb}_1))] \* s_1;
18. \text{L} = s_1;
19. \text{else if } (0 < (s_1 \text{ mod } \text{Nb}_1) < b_1) \text{ then}
20. \text{R} = s_1;
21. \text{L} = [(N \* \text{b}_1 - b_1 + 1)/(s_1 \text{ mod } \text{Nb}_1)] \* s_1;
22. \text{else}
23. \text{R} = \infty; \text{ L'} = 0;
24. \text{for } i = d, b_1 - 1, d
25. \text{loc} = (s_1 \text{ mod (Nt}b_1 - (s_1 \text{ mod } \text{Nb}_1));
26. \text{R} = \min(R, \text{loc});
27. \text{L'} = \max(L', \text{loc});
28. \text{endfor}
29. \text{L} = \text{Nt}b_1 * s_1 / d - L';
30. \text{endif}

Step 5: \{\text{* Calculate the first cycle of the memory access sequence } \delta M. \text{* }\}
31. \text{now} = \text{start};
32. \delta M[0] = \text{now}; \text{ i} = 1;
33. \text{while } (\text{i} < \text{length}) \text{ do}
34. \text{if } (p \text{b}_1 \leq ((\text{now} - a_1 + R) \text{ mod } (\text{Nt}b_1)) \text{ then }
35. \text{now} = \text{now} + R;
36. \text{else if } (p \text{b}_1 \leq ((\text{now} - a_1 + L) \text{ mod (Nt}b_1)) \text{ then }
37. \text{now} = \text{now} + L;
38. \text{endif}
39. \text{now} = \text{now} + R + L;
40. \text{endif}
41. \text{delta}[i] = \text{now}; \text{ i} = \text{i} + 1;
42. \text{endif}

Step 6: \{\text{* Formulate the memory access sequence } \Delta M. \text{* }\}
43. \text{return } \Delta M = \{\delta M : u_1 : \text{period}_s^d * s_1\}.

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 $\Delta M$ represent the memory access sequence of $A(l_1 : u_1 : s_1)$ in $PE_p$ and $\delta M$ 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 $\Delta M$. This algorithm contains six steps as follows.

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

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

if $s_1$ (stride) \leq $b_1$ (block size) then $\Delta M = [\text{bot}_a(A, p, j_0, j_f) : \text{top}_a(A, p, j_0, j_f) : s_1]$ \cup

$[\bigcup_{j = j_0 + 1}^{j_f + \text{period}_s^d} \text{bot}_a(A, p, j) : \text{top}_a(A, p, j) : s_1)] : u_1 : \text{period}_s^d * s_1$, and then STOP;
{ * In the following, } s_1 > b_1. { *}

**Step 2:** { * Line 9. *}

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

if $\Delta M = \phi$ 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 $\delta M$ according to Theorem 5;

**Step 6:** { * Line 42. *}

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

### 4.3 Relation Between Memory Access Sequence Generation and Communication Set Generation

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

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

$$= local_C(p) \cap \left( \left[ bot_j(A, q, j_{j_f}) : top_j(A, q, j_{j_f}) : s_2 \right] \cup \left( \bigcup_{j = j_{j_f} + 1}^{\min\{j_{j_f} + period_c^A\}} \left[ bot_j(A, q, j) : top_j(A, q, j) : s_2 \right] : u_2 : period_c^A * s_2 \right) \right).$$

Since for every $j \in [j_{j_f} + 1 : \min\{j_{j_f} + period_c^A\}]$, the set $[bot_j(A, q, j) : u_2 : period_c^A * s_2]$ forms a lattice, the problem of solving $local_C(p) \cap [bot_j(A, q, j) : u_2 : period_c^A * s_2]$ 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 $bot_j(A, q, j) + i * period_c^A * s_2$ is not in $local_C(p)$ for some $j$ and $i$, it is still possible that $local_C(p) \cap [bot_j(A, q, j) + i * period_c^A * s_2 : top_j(A, q, j) + i * period_c^A * s_2 : s_2]$. 
is not empty. In this case, we need to consider index $bot_f(A, q, j) + i \cdot period_c^A + s_2$ 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 $[bot_f(A, q, j) + i \cdot period_c^A + s_2 : top_f(A, q, j) + i \cdot period_c^A + s_2 : s_2]$ will intersect with at most one local block of $local_C(p)$, we will deal with the two cases, $\left[ \frac{b_1}{s_1} \right] \leq \left[ \frac{(N-1)b_2+1}{s_2} \right]$ and $\left[ \frac{b_1}{s_1} \right] \leq \left[ \frac{(N-1)b_2+1}{s_2} \right]$, separately as described in Property 1. In this presentation, we will only present the case where $\left[ \frac{b_1}{s_1} \right] \leq \left[ \frac{(N-1)b_2+1}{s_2} \right]$; the other case where $\left[ \frac{b_1}{s_1} \right] \geq \left[ \frac{(N-1)b_2+1}{s_2} \right]$ 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 $local'_c(p) \cap [bot_f(A, q, j) : u_2 : period_c^A + s_2]$ can be enumerated according to Corollary 6; in addition, for each element $bot_f(A, q, j) + i \cdot period_c^A + s_2$ in the mentioned set, $local_c(p) \cap [bot_f(A, q, j) + i \cdot period_c^A + s_2 : top_f(A, q, j) + i \cdot period_c^A + s_2 : s_2] \neq \phi$.

**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 $A(4 + i \cdot 2) = g(C(2 + i))$, 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 $A(4 + i \cdot 2)$ 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, $period_c^A_{b_1} = s_1 / \gcd(Nb_1, s_1) = 2$. First, the set $[bot_f(A, 0, 1) : u_2 : period_c^A + s_2] = [14 : 314 : 27]$ forms a lattice as shown in Figure 6—(a). Although index $bot_f(A, 0, 1) + 2 \cdot period_c^A + s_2 = 68$ is not in $local_c(2)$, $local_c(2) \cap [bot_f(A, 0, 1) + 2 \cdot period_c^A + s_2 : top_f(A, 0, 1) + 2 \cdot period_c^A + s_2 : s_2] = local_c(2) \cap [68 : 71 : 1] = [70 : 71 : 1]$. 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 \( lb = Nb_2 + pb_2 - top_j(A, 0, 1) + bot_j(A, 0, 1) = 12 \), as shown in Figure 7-(b).

<table>
<thead>
<tr>
<th>( PE_1 )</th>
<th>( PE_2 )</th>
<th>( PE_3 )</th>
<th>( PE_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Figure 7: When dealing with the lattice \([bot_j(A, 0, 1) : u_2 : period^A + s_2] \), (a) for processing elements \( p = 1 \) and 2, the virtual left boundary of \( PE_p \) is \( pb_2 - top_j(A, 0, 1) + bot_j(A, 0, 1) = 2 \) and 7, respectively; (b) for processing element \( p = 0 \), the virtual left boundary of \( PE_p \) is \( Nb + pb_2 - top_j(A, 0, 1) + bot_j(A, 0, 1) = 12 \).

Similarly, the set \([bot_j(A, 0, 2) : u_2 : period^A + s_2] = [27 : 314 : 27] \) forms a lattice as shown in Figure 6- (b). Although index \( bot_j(A, 0, 2) + 2 \times period^A + s_2 = 81 \) is not in \( local_c(2) \), \( local_c(2) \cap [bot_j(A, 0, 2) + 2 \times period^A + s_2 : top_j(A, 0, 2) + 2 \times period^A + s_2] = local_c(2) \cap [81 : 85 : 1] = [85 : 85 : 1] \). 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 \( lb = pb_2 - top_j(A, 0, 2) + bot_j(A, 0, 2) = 1 \) and 6, respectively, as shown in Figure 8-(a). For processing element \( p = 0 \), the virtual left boundary of \( PE_p \) is \( lb = Nb + pb_2 - top_j(A, 0, 2) + bot_j(A, 0, 2) = 11 \), as shown in Figure 8-(b).
Figure 8: When dealing with the lattice $[bot f(A, 0, 2) : u_2 : period^A_c * s_2]$, (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 $N_b - pb_2 - top f(A, 0, 2) + bot f(A, 0, 2) = 11$.

4.3.1 The Cases When $period^A_c * s_2 \leq b_2 + top f(A, q, j) - bot f(A, q, j)$

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

Define

\[
\begin{align*}
    l &= bot f(A, q, j); \quad u = u_2; \quad s = period^A_c * s_2; \\
    d &= \gcd(N_b, s); \quad period^C_s = s / d; \quad period_s = N_b / s / d; \\
    len &= top f(A, q, j) - bot f(A, q, j); \quad b = b_2 + len; \\
    lb &= \begin{cases} 
        pb_2 - len, & \text{if } pb_2 - len \geq 0; \\
        N_b + pb_2 - len, & \text{otherwise}; 
    \end{cases} \quad rb = pb_2 + b_2 - 1; \\
    lp &= c_1 + lb; \quad rp = c_1 + rb; \\
    k'_{pl} &= \frac{(l - rp) / (N_b)}{len}; \quad k'_{pl} = \frac{(u - lp) / (N_b)}{len}; \\
    bot'_{pl}(C, p, k) &= \max\{ip + kN_b, l, s\}; \\
    top'_{pl}(C, p, k) &= \min\{rp + kN_b, u - s + 1, l, s\}; \\
    \alpha_k &= \begin{cases} 
        k, & \text{if } lb < rb; \\
        k - 1, & \text{if } lb > rb; \\
        \beta_k = \begin{cases} 
            k, & \text{if } lb < rb; \\
            k + 1, & \text{if } lb > rb; 
        \end{cases} 
    \end{cases} \\
    local^L_C(p) \cap [bot'_{pl}(C, p, k) : bot'_{pl}(C, p, k) + len : s_2] &= [bot_{pl}(C, p, \beta_k) : \min\{top_{pl}(C, p, \beta_k), bot'_{pl}(C, p, k) + len : s_2\}.
\end{align*}
\]
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

$$local'_{C}(p) \cap [l: u:s] = [bot_{a}'(C, p, \alpha_{k_{pj}^{'}}): top_{a}'(C, p, k_{pj}^{'}): s] \cup$$

$$\left( \bigcup_{k=k_{pj}^{'}+1}^{\min(\beta_{pi}^{'}+k_{pj}^{'}+period_{C}^{\pi})} [bot_{a}'(C, p, \alpha_{k}): top_{a}'(C, p, k): s] \right) : u:period_{s}.$$  

Therefore,

$$local_{C}(p) \cap ([bot_{f}(A, q, j): top_{f}(A, q, j): s_{2}] : u_{2}: period_{C}^{A} + s_{2})$$

$$= local_{C}(p) \cap ([l: u: s_{2}] : u: s_{2})$$

$$= local_{C}(p) \cap ([local'_{C}(p) \cap [l: u: s])] \cap [l: u: s_{2}] : u: len:s_{2}$$

$$= [bot_{a}(C, p, k_{pj}^{'}): \min\{top_{a}(C, p, k_{pj}^{'}), bot_{a}'(C, p, \alpha_{k_{pj}^{'}}) + len\} : s_{2}] \cup$$

$$\left( [bot_{a}'(C, p, \alpha_{k_{pj}^{'}}) + s : \min\{top_{a}(C, p, k_{pj}^{'}), bot_{a}'(C, p, \alpha_{k_{pj}^{'}}) + s + len\} : s_{2}] : u: period_{s} \right).$$

### 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 $Nb_{2} - b < (s \mod Nb_{2}) < Nb_{2}$: Similar to the derivation in Section 4.2.2, we have $R = [(Nb_{2} - b + 1)/(Nb_{2} - (s \mod Nb_{2}))] * s$ and $L = s$.

2. When $0 < (s \mod Nb_{2}) < b$: Similar to the derivation in Section 4.2.3, we have $R = s$ and $L = [(Nb_{2} - b + 1)/(s \mod Nb_{2})] * s$.

### 4.4 An Algorithm for Deriving the Communication Sets

In this subsection, we will present an algorithm for calculating $local_{C}(p) \cap [[bot_{f}(A, q, j): top_{f}(A, q, j): s_{2}] : u_{2}: period_{C}^{A} + s_{2}]$. We will first use the algorithm described in Section 4.2.4, according to Corollary 6, to enumerate all indices $bot_{f}(A, q, j) + i * period_{C}^{A} + s_{2}$ such that $local_{C}(p) \cap [bot_{f}(A, q, j) + i * period_{C}^{A} + s_{2}] \neq \emptyset$. Then, the communication set $local_{C}(p) \cap$
[[bot_j(A,q,j) : top_j(A,q,j) : s_2] : u_2 : period^A \cdot s_2] can be derived incrementally according to these indices. We will now present the cases where \( lb = pb + bot_j(A,q,j) + bot_j(A,q,j) \geq 0 \), thus \( lb \leq pb + b - 1 = rb \). We will illustrate certain modified code in Figure 10 so that the modified algorithm can handle the cases where \( pb - top_j(A,q,j) + bot_j(A,q,j) < 0 \) and \( lb = Nb + pb - top_j(A,q,j) + bot_j(A,q,j) = pb + b - 1 = rb \) at the end of this section.

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

Step 1: {* Line 9. *}
Handle the special cases where \( \text{period}^A \cdot s_2 \leq b + top_j(A,q,j) - bot_j(A,q,j) \) as follows:

if \( \text{period}^A \cdot s_2 \leq b + top_j(A,q,j) - bot_j(A,q,j) \) then \( \Delta Z \) can be represented by the formula mentioned in Section 4.3.1, and then STOP;

{* In the following, \( \text{period}^A \cdot s_2 > b + top_j(A,q,j) - bot_j(A,q,j) \). *}

Step 2: {* Line 10. *}
check whether \( \Delta Z \) is empty or not in constant time;

if \( \Delta Z = \phi \) 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 \( \delta M \) according to Corollary 6;
compute the first cycle of the communication set \( \delta Z \) according to \( \delta M \);

Step 6: {* Line 45. *}
formulate the communication set \( \Delta Z = [\delta Z : u_2 : \text{period}^A] ; \) STOP. \( \square \)
Input: $a_1, a_2$ (the range of a generated array $A(a_1 : a_2)$), $b_1, b_2, s_1$ (parameters of the access pattern $A(b_1 : b_2 : s_1)$), $c_1, c_2$ (the range of a used array $C(c_1 : c_2)$), $b_2, w_2, s_2$ (parameters of the access pattern $C(b_2 : w_2 : s_2)$), $N$ (number of PEs), $b_2, b_2$ (block sizes of $A$ and $C$), $p$ and $q$ (processing element ID).

Output: The $\Delta Z$ table. $\{ \Delta Z = \text{localcalc}(p) \cap \text{bot}(A, q, j) : \text{top}(A, q, j) : s_2, w_2 : \text{period}_A \times s_2 \}$. 

1. #define nxt(x, y, z) = x + ((y - x) mod z); 
2. if (bot(A, q, j) > top(A, q, j)) return $\Delta Z = \phi$; 
3. for (d, x, y) $\Rightarrow$ EXTENDED-EUCLID($N_{b_2}$, s) 
   * $d = \text{gcd}(N_{b_2}, s) = x \times N_{b_2} + y \times s$. * 
4. period$_A = N_{b_2} / d; \text{ period}_s = \text{period}_C \times s; 
5. len = top(A, q, j) - bot(A, q, j); \text{ b = b} + \text{len}; 
6. lb = pb_2 - len; \text{ rb = pb}_2 + b_2 - 1; 
7. lp = c_1 + lb; \text{ rp = c}_1 + rb; 
8. Step 1: * Handle the special cases where $s \leq b$. * 
   if (s \leq b) return $\Delta Z = \text{the formula in Section 4.3.1}; 
9. Step 2: * Check whether $\Delta Z$ is empty or not. * 
   if (nxt(lp - l, d, d) > rp - l) return $\Delta Z = \phi$; 
10. Step 3: * Find the starting point accessed among the columns from lb to rb. * 
    start = $\infty$; length = 0; 
    for i = nrx(lp - l, d, d), rpx - l, d 
    loc = l + (s/d)(iy + N_{b_2}[-iy/(N_{b_2})]); 
    start = min(start, loc); 
    length = length + 1; 
    endfor 
11. Step 4: * Derive distance vectors $R$ and $L$. * 
    if (N_{b_2} - b < (s \mod N_{b_2}) < N_{b_2}) then 
    R = [(N_{b_2} - b + 1) / (N_{b_2} - (s \mod N_{b_2})]] \times s; 
    L = s; 
    12. else if (0 < (s \mod N_{b_2}) < b) then 
    R = s; 
    L = [(N_{b_2} - b + 1) / (s \mod N_{b_2})] \times s; 
    else 
    return $\Delta Z = \phi$; 
13. Step 5: * Calculate the first cycle of both the memory access sequence and the communication set $S_2$. * 
    35. while (i < length) do 
    36. if (lb \leq ((now - c_1 + R) \mod (N_{b_2}) \leq rb) then 
    37. now = now + R; 
    38. else if (lb \leq ((now - c_1 + L) \mod (N_{b_2}) \leq rb) then 
    39. now = now + L; 
    40. else now = now + R + L; 
    41. endif endif 
    42. k = (now - c_1) / (N_{b_2}); \text{ i = i + 1; 
    43. $S_2 = \text{top}(A, C, k, k, now + len) \times s_2]$; 
    44. endwhile 
14. Step 6: * Formulate the communication set $\Delta Z$. * 
    45. return $\Delta Z = [S_2 : w_2 : \text{period}_A]$.

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

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

7'. lb = N_{b_2} + pb_2 - len; \text{ rb = pb}_2 + b_2 - 1; 
10'. if (nxt(lp - l, d, d) > N_{b_2} + rp - l) 
    return $\Delta Z = \phi$; 
12'. for i = nrx(lp - l, d, d), N_{b_2} + rp - l, d 
    35'. if (not (rb < ((now - c_1 + R) \mod (N_{b_2}) \leq rb) then 
    36'. else if (not (rb < ((now - c_1 + L) \mod (N_{b_2}) \leq rb) then 
    38'. if (now mod N_{b_2}) \leq rb) then 
    39'. k = (now - c_1) / (N_{b_2}); 
    else k = (now - c_1) / (N_{b_2} + 1; 
    endif 
    i = i + 1; 
12'. for i = nrx(lp - l, d, d), N_{b_2} + rp - l, d 

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 $(\text{period}^A_{sb_1} + 1)$ or $(\text{period}^C_{sb_2} + 1)$ closed forms in Section 3; otherwise, we must apply an efficient algorithm $\text{period}^A_{sb}$ or $\text{period}^C_{sb}$ times in order to generate the communication set $\text{send}_C(p, q)$ 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, $\text{period}^A_{sb}$, $\text{period}^C_{sb}$; or $\text{period}^A_{eb}$, the computation overhead becomes serious if the corresponding $\text{period}^A_{sb}$, $\text{period}^C_{sb}$; or $\text{period}^A_{eb}$ 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 $p = (\lfloor \frac{i \cdot b_1}{s_1} \rfloor \mod N)$, and the entry $C(j')$ to $PE_{p'}$, where $p' = (\lfloor \frac{i' \cdot b_2}{s_2} \rfloor \mod N)$. Then, for $i \in \{0, 1, \ldots, h - 1\}$, $A(l_1 + i \cdot s_1)$ and $C(l_2 + i \cdot s_2)$ are in $PE_0$; for $i \in \{h, h + 1, \ldots, 2 \cdot h - 1\}$, $A(l_1 + i \cdot s_1)$ and $C(l_2 + i \cdot s_2)$ 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 $b_1 = s_1 \cdot h$ and $b_2 = s_2 \cdot h$.

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 $f_A(j) = (\lfloor \frac{j \cdot \text{offset}_A}{b_1} \rfloor \mod N)$ and $f_C(j') = (\lfloor \frac{j' \cdot \text{offset}_C}{b_2} \rfloor \mod N)$, 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.
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):

\[
\begin{align*}
A_1(l_1 + i * s_{11}) &= f_{11}(\cdots); \\
A_1(l_2 + i * s_{12}) &= f_{12}(\cdots); \\
&\vdots \hspace{8cm} \vdots \\
A_1(l_{x_1} + i * s_{x_1}) &= f_{1x_1}(\cdots); \\
A_2(l_{21} + i * s_{21}) &= f_{21}(\cdots); \\
&\vdots \hspace{8cm} \vdots \\
A_m(l_{m1} + i * s_{m1}) &= f_{m1}(\cdots); \\
&\vdots \hspace{8cm} \vdots \\
A_m(l_{mx_m} + i * s_{mx_m}) &= f_{mx_m}(\cdots).
\end{align*}
\]

In the above statements, \( f_{ij}(\cdot) \) is a function of \( (A_1(l_1 + i * s_{11}), A_1(l_2 + i * s_{12}), \ldots, A_1(l_{x_1} + i * s_{x_1})) \),

<table>
<thead>
<tr>
<th>cases</th>
<th>conditions</th>
<th>send_{peC}(p)</th>
<th>recv_{peC}(p)</th>
<th>send_{C}(p, q)</th>
<th>recv_{C}(p, q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>arbitrary ( b_1 ) and ( b_2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( b_1/s_1 ) is a factor of ( b_2/s_2 )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>3</td>
<td>( b_1/s_1 ) is a multiple of ( b_2/s_2 )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>4</td>
<td>all-closed-forms condition*</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
</tbody>
</table>

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 \( N \).
$A_2(l_{21}+i\ast s_{21}), \ldots, A_m(l_{m1}+i\ast s_{m1}), \ldots, A_m(l_{mxm}+i\ast s_{mxm}), \ldots, A_n(l_{n1}+i\ast s_{n1}), \ldots, A_n(l_{nxn}+i\ast s_{nxn})$.

It is reasonable to assume that each stride $s_{ij}$ is a small integer $\lfloor 3/4 \rfloor \leq i \leq 5$ because $b_i$ must be a multiple of all $s_{ij}$, for $1 \leq j \leq x_i$.

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 $\text{lcm}(s_{i1}, s_{i2}, \ldots, s_{ix_i})$. 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 $\pi$-block. The graph then includes an acyclic partial ordering on the $\pi$-blocks. We will now define a new level for each $\pi$-block below, where $\Pi$ is a $\pi$-block; $\Psi$ is a source $\pi$-block; and $\Omega$ is a sink $\pi$-block:

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

Suppose that the $nlevel$ of the acyclic $\pi$-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 $\pi$-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

$$b_j = \text{lcm}_{(k)}(b_j, (b_k/s_{ky})s_{jx}),$$ 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 \( i = \text{nl} \) down to 1 do

According to an ordering from parent to child by topological sorting in \( n\text{level} \ i \):

for each \( A_k \) in \( n\text{level} \ i \) do

for all edges \((A_j, A_k)\) from \( A_k \) to \( A_j \), where \( A_j \) are also in \( n\text{level} \ i \), and

all edges \((A_k, A_j)\) and \((A_j, A_k)\), where \( A_j \) are in \( n\text{level} \ i' \) and \( i' > i \) do

\[ b_k = \gcd_{\forall j}( (b_j/s_{jx})s_{ky} ), \]

where \( A_j(l_{jy} + s_{jy}) \) 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 \( b_i = b_i \times h \), where \( h \) is a granularity factor. \( \Box \)

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) \[ A_1(l_{11} + i \times s_{11}) = A_1(l_{11} + i \times s_{11}) + A_2(l_{21} + i \times s_{21}); \]

(2) \[ A_1(l_{12} + i \times s_{12}) = A_1(l_{12} + i \times s_{12}) - A_3(l_{31} + i \times s_{31}); \]

(3) \[ A_2(l_{22} + i \times s_{22}) = A_2(l_{22} + i \times s_{22}) \times A_4(l_{41} + i \times s_{41}); \]

(4) \[ A_2(l_{23} + i \times s_{23}) = A_2(l_{23} + i \times s_{23}) + A_5(l_{51} + i \times s_{51}); \]
(5) 
\[ A_3(l_{32} + i \times s_{32}) = A_3(l_{32} + i \times s_{32}) - A_5(l_{52} + i \times s_{52}) \]
(6) 
\[ A_4(l_{42} + i \times s_{42}) = A_4(l_{42} + i \times s_{42}) - A_2(l_{24} + i \times s_{24}) \]
(7) 
\[ A_5(l_{53} + i \times s_{53}) = A_5(l_{53} + i \times s_{53}) + A_6(l_{61} + i \times s_{61}) \]
(8) 
\[ A_6(l_{62} + i \times s_{62}) = A_6(l_{62} + i \times s_{62}) - A_7(l_{71} + i \times s_{71}) \]
(9) 
\[ A_6(l_{63} + i \times s_{63}) = A_6(l_{63} + i \times s_{63}) + A_8(l_{81} + i \times s_{81}) \]

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 \textit{nonevel} 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.

\[
\begin{align*}
\text{Step 1} & \\
\beta_1^1 & = 1 \text{lcm}(s_{11}, s_{12}) \\
\beta_2^1 & = 1 \text{lcm}(s_{21}, s_{22}, s_{23}, s_{24}) \\
\beta_3^1 & = 1 \text{lcm}(s_{31}, s_{32}) \\
\beta_4^1 & = 1 \text{lcm}(s_{41}, s_{42}) \\
\beta_5^1 & = 1 \text{lcm}(s_{51}, s_{52}, s_{53}) \\
\beta_6^1 & = 1 \text{lcm}(s_{61}, s_{62}, s_{63}) \\
\beta_7^1 & = s_{71} \\
\beta_8^1 & = s_{81} \\
\end{align*}
\]

**Table 4**

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b_1^1 )</td>
<td>( b_2^1 )</td>
<td>( b_4^1 )</td>
</tr>
<tr>
<td>( b_2^1 )</td>
<td>( b_3^1 )</td>
<td>( b_5^1 )</td>
</tr>
<tr>
<td>( b_3^1 )</td>
<td>( b_4^1 )</td>
<td>( b_6^1 )</td>
</tr>
<tr>
<td>( b_4^1 )</td>
<td>( b_5^1 )</td>
<td>( b_1^1 )</td>
</tr>
<tr>
<td>( b_5^1 )</td>
<td>( b_6^1 )</td>
<td>( b_2^1 )</td>
</tr>
<tr>
<td>( b_6^1 )</td>
<td>( b_7^1 )</td>
<td>( b_3^1 )</td>
</tr>
<tr>
<td>( b_7^1 )</td>
<td>( b_8^1 )</td>
<td>( b_4^1 )</td>
</tr>
<tr>
<td>( b_8^1 )</td>
<td>( b_9^1 )</td>
<td>( b_5^1 )</td>
</tr>
</tbody>
</table>

Figure 11: (a) The corresponding directed graph of the nine statements, where \((i)\) in each edge represents the \(i\)-th statement. (b) The corresponding \textit{nonevel} 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 \( b_1 = s_1 \times h_1 \) and \( b_2 = s_2 \times h_1 \times h_2 \)

In this case, \( b_1 / s_1 \) is a factor of \( b_2 / s_2 \). Therefore, \( send_{\text{pe}} C(p) \), \( send C(p, q) \), and \( recv C(p, q) \) have closed forms. First, we process \( send_{\text{pe}} C(p) \), which is equal to \( f_A( f_1( f_2^{-1}(\text{local}_C(p) \cap [l_2 : u_2 : s_2])) \). Since \( period_s = Nb_2 \) and \( period_s^C = period_s / (Nb_2) = 1 \), 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 \( f_A(\text{bot}_j(C, p, k_{p,j})) \) and \( f_A(\text{top}_j(C, p, k_{p,j})) \), respectively. Note that, if \( \text{next}(\text{bot}_i(C, p, k_{p,j}), l_2, s_2) < l_2 \), then \( f_A(\text{bot}_j(C, p, k_{p,j})) \) may not be equal to \( f_A(\text{bot}_j(C, p, k_{p,j})) \). Based on Property 3, we have the following closed form.

\[
send_{\text{pe}} C(p) = \begin{cases} 
[0 : N - 1], & \text{if } u_2 - l_2 + 1 \geq Nb_2 \text{ and } h_2 \geq N; \\
[f_A(\text{bot}_j(C, p, k_{p,j})): f_A(\text{top}_j(C, p, k_{p,j}))], & \text{if } u_2 - l_2 + 1 \geq Nb_2, h_2 < N, \text{ and } f_A(\text{bot}_j(C, p, k_{p,j})) \leq f_A(\text{top}_j(C, p, k_{p,j})); \\
[0 : f_A(\text{top}_j(C, p, k_{p,j})): f_A(\text{bot}_j(C, p, k_{p,j})): N - 1], & \text{if } u_2 - l_2 + 1 \geq Nb_2, h_2 < N, \text{ and } f_A(\text{bot}_j(C, p, k_{p,j})) > f_A(\text{top}_j(C, p, k_{p,j})); \\
f_A([\text{bot}_j(C, p, k_{p,j}): \text{top}_j(C, p, k_{p,j})]) \cup f_A([\text{bot}_j(C, p, k_{p,j}): \text{top}_j(C, p, k_{p,j})]), & \text{if } u_2 - l_2 + 1 < Nb_2.
\end{cases}
\]

Second, we formulate \( recv_{\text{pe}} C(p) \), which is equal to \( f_C( f_2(\text{exec}(p))) \). We start from \( \text{exec}(p) \) and check the elements of array \( C \) that these iterations will refer to. Recall that \( \text{exec}(p) = \bigcup_{j = 0}^{b_2} [\text{bot}_s(A, p, j) : \text{top}_s(A, p, j)] \). Then, \( f_2(\text{exec}(p)) = \bigcup_{j = 0}^{b_2} [\text{bot}_j(A, p, j) : \text{top}_j(A, p, j) : s_2] \), which represents the elements of array \( C \) that are referenced by iterations executed in \( PE_p \), and \( f_C( f_2(\text{exec}(p))) \) indicates the set of PEs that store these elements of array \( C \). Since \( period_s^A = period_s * s_1 / (Nb_1 s_2) = h_2 \), \( recv_{\text{pe}} C(p) \) can be represented by a union of at most \( h_2 + 1 \) closed forms:

\[
recv_{\text{pe}} C(p) = \begin{cases} 
[0 : N - 1], & \text{if } u_2 - l_2 + 1 \geq Nb_2 \text{ and } h_2 \geq N; \\
\bigcup_{j = 0}^{l_2 + h_2 - 1} f_C([\text{bot}_j(A, p, j) : \text{top}_j(A, p, j)]), & \text{if } u_2 - l_2 + 1 \geq Nb_2, h_2 < N, \text{ and } \text{next}(\text{bot}(A, p, j), l_1, s_1) \geq l_1; \\
\left( \bigcup_{j = 0}^{l_2 + h_2 - 1} f_C([\text{bot}_j(A, p, j) : \text{top}_j(A, p, j)]) \right) \cup \left( \bigcup_{j = 0}^{l_2 + h_2 - 1} f_C([\text{bot}_j(A, p, j) : \text{top}_j(A, p, j)]) \right) \cup f_C([\text{bot}_j(A, p, j) : \text{top}_j(A, p, j)]), & \text{if } u_2 - l_2 + 1 \geq Nb_2, h_2 < N, \text{ and } \text{next}(\text{bot}(A, p, j), l_1, s_1) \leq l_1; \\
\bigcup_{j = 0}^{l_2} f_C([\text{bot}_j(A, p, j) : \text{top}_j(A, p, j)]), & \text{if } u_2 - l_2 + 1 < Nb_2.
\end{cases}
\]

Note that, in the above formula, the set \( f_C([\text{bot}_j(A, p, j) : \text{top}_j(A, p, j)]) \) consists of only one or two PEs. In addition, all these PEs are distinct. However, in spite of these facts, \( recv_{\text{pe}} C(p) \) 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 \(local_C(p) \cap f_2(e_{exec}(q))\). This set will be represented by a union of three closed forms: \(shead_C(p, q)\), \(sbody(y)_C^1(p, q)\), and \(sbody(y)_C^2(p, q)\). 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, \(h_1 = 11\); \(s_1 = 2\); \(l_2 = 2\); \(s_2 = 1\); and \(u_2 = 369\). If we let \(h_1 = 2\) and \(h_2 = 11\), then \(h_1 = s_1 * h_1 = 4\) 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, \(send_C(0, 1) = shead_C(0, 1) \cup sbody(y)_C^1(0, 1) \cup sbody(y)_C^2(0, 1)\), where \(shead_C(0, 1) = [7 : 8 : 1] \cup [[15 : 16 : 1] : 21 : 8]\); \(sbody(y)_C^1(0, 1) = [[88 : 88 : 1] : 369 : 88]\); and \(sbody(y)_C^2(0, 1) = [[[95 : 96 : 1] : 109 : 8] : 369 : 88]\). \(send_C(0, 2) = shead_C(0, 2) \cup sbody(y)_C^2(0, 2)\), where \(shead_C(0, 2) = [2 : 2 : 1] \cup [[9 : 10 : 1] : 21 : 8]\) and \(sbody(y)_C^2(0, 2) = [[[89 : 90 : 1] : 109 : 8] : 369 : 88]\). 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, \(sbody(y)_C^1(0, 2) = \phi\).

![Diagram of array C in PE_0](image)

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

We notice that \(shead_C(p, q)\) is not empty if \(\text{nxt}(\text{bot}(i, C, p, k, j), l_2, s_2) < l_2\); \(sbody(y)_C^2(p, q)\) includes some elements if \(\text{bot}(i, C, p, k)\) is in between \(\text{bot}(j, A, q, j) + 1\) and \(\text{top}(j, A, q, j)\) for some \(j\) and \(k\); and
\[sbody_{C}^{2}(p, q)\] will be evaluated without any conditions. In addition, the period of \(f_{2}(exec(q))\) is 
\(((\text{lcm}(Nb_{1}, s_{1})/s_{1}) + s_{2}) = Ns_{2}h_{1}\) and \(period = Nb_{2}\). Let \(k'_{pj} = k_{pj} + 1\) if \(nxt(\text{bot}_{i}(C, p, k_{pj}), l, s_{2}) < l_{2}, \ k'_{pj} = k_{pj}\) otherwise. Then, we have

\[
\text{shead}_{C}(p, q) = \begin{cases} 
\{[\text{bot}_{f}(A, q, j_{f}) : \text{min}\{\text{top}_{f}(A, q, j_{f}), \text{top}_{a}(C, p, k_{pj})\} : s_{2}\} \cup \\
\{[\text{bot}_{f}(A, q, j_{f} + 1) : \text{top}_{f}(A, q, j_{f} + 1) : s_{2}] : \text{top}_{a}(C, p, k_{pj}) : Ns_{2}h_{1}\},
\end{cases}
\]

\[
\text{shead}_{C}(p, q) = \begin{cases} 
\{[\text{bot}_{a}(C, p, k'_{pj}) : ntx(\text{bot}_{a}(C, p, k'_{pj}), \text{top}_{f}(A, q, j_{f}), Ns_{2}h_{1}) : s_{2}] : u_{2} : Nb_{2}\},
\end{cases}
\]

\[
\text{sbody}_{C}(p, q) = \begin{cases} 
\{[\text{bot}_{a}(C, p, k'_{pj}) : \text{bot}_{f}(A, q, j_{f} + 1), Ns_{2}h_{1}) : \\
\{\text{bot}_{a}(C, p, k'_{pj}), \text{bot}_{f}(A, q, j_{f} + 1), Ns_{2}h_{1} + s_{2}(h_{1} - 1) : s_{2}] : \\
\text{top}_{a}(C, p, k'_{pj}) : Ns_{2}h_{1} : u_{2} : Nb_{2}\},
\end{cases}
\]

\[
\text{sbody}_{C}(p, q) = \text{shead}_{C}(p, q) \cup \text{sbody}_{C}^{1}(p, q) \cup \text{sbody}_{C}^{2}(p, q).
\]

In summary, it is only necessary to determine two conditions and at most eight boundary variables to calculate \(\text{send}_{C}(p, q)\) as shown in Table 5. Similarly, \(\text{recv}_{C}(p, q)\) also has this property, and we will not repeat another Table 5 again in the following.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>(\text{send}_{C}(p, q))</th>
</tr>
</thead>
</table>
| \(\lambda_{1}\)  | \([\mu_{1} : \mu_{2} : s_{2}] \cup [\mu_{3} : \mu_{3} + s_{2}(h_{1} - 1) : s_{2}] : \mu_{4} : Ns_{2}h_{1}\] \cup \\
| and \(\lambda_{2}\) | \([[[\mu_{5} : \mu_{6} : s_{2}] \cup [[\mu_{7} : \mu_{7} + s_{2}(h_{1} - 1) : s_{2}] : \mu_{8} : Ns_{2}h_{1}]]] : u_{2} : Nb_{2}\]              |
| \(\lambda_{1}\)  | \([\mu_{1} : \mu_{2} : s_{2}] \cup [\mu_{3} : \mu_{3} + s_{2}(h_{1} - 1) : s_{2}] : \mu_{4} : Ns_{2}h_{1}\] \cup \\
| and \(\neg \lambda_{2}\) | \([[[\mu_{7} : \mu_{7} + s_{2}(h_{1} - 1) : s_{2}] : \mu_{8} : Ns_{2}h_{1}] : u_{2} : Nb_{2}\]               |
| \(\neg \lambda_{1}\) | \([[[\mu_{5} : \mu_{6} : s_{2}] \cup [[\mu_{7} : \mu_{7} + s_{2}(h_{1} - 1) : s_{2}] : \mu_{8} : Ns_{2}h_{1}]]] : u_{2} : Nb_{2}\] |
| and \(\neg \lambda_{2}\) | \([[[\mu_{7} : \mu_{7} + s_{2}(h_{1} - 1) : s_{2}] : \mu_{8} : Ns_{2}h_{1}] : u_{2} : Nb_{2}\]   |

\(\lambda_{1} = (nxt(\text{bot}_{i}(C, p, k_{pj}), l_{2}, s_{2}) < l_{2}); \ \lambda_{2} = ((\mu_{1} < \mu_{5} \leq \text{top}_{f}(A, q, j_{f})) \text{ or } (\mu_{6} = s_{2}(h_{1} - 1) < \mu_{5} \leq \min\{\mu_{6}, \text{top}_{f}(A, q, j_{f})\})) ; \ \mu_{1} = \text{bot}_{f}(A, q, j_{f}); \ \mu_{2} = \min\{\text{top}_{f}(A, q, j_{f}), \mu_{4}\}; \ \mu_{3} = \text{bot}_{f}(A, q, j_{f} + 1); \ \mu_{4} = \text{top}_{a}(C, p, k_{pj}); \ \mu_{5} = \text{bot}_{a}(C, p, k'_{pj}); \ \mu_{6} = ntx(\mu_{5}, \text{top}_{f}(A, q, j_{f}), Ns_{2}h_{1}); \ \mu_{7} = ntx(\mu_{5}, \mu_{3}, Ns_{2}h_{1}); \ \text{and } \mu_{8} = \text{top}_{a}(C, p, k'_{pj}).

Table 5: It is only necessary to determine two conditions and at most eight boundary variables to calculate \(\text{send}_{C}(p, q)\).

Fourth, we are concerned with \(\text{recv}_{C}(p, q)\), which is equal to \(\text{send}_{C}(q, p)\). Hence, \(\text{recv}_{C}(p, q)\) also can be represented by a union of three closed forms. As indicated in Section 3, we prefer that \(\text{recv}_{C}(p, q)\)
be represented based on indices of array $A$. In addition, there is a one-to-one correspondence between $rhead_C(p, q) \cup rbody_1^1(p, q) \cup rbody_2^2(p, q)$ and $f_2 f_1^{-1}(f_1(f_2^{-1}(rhead_C(p, q))) \cup f_1(f_2^{-1}(rbody_1^1(p, q))) \cup f_1(f_2^{-1}(rbody_2^2(p, q))))$. Let $k'_{q, f} = k_{q, f} + 1$ if $\text{nxt}(bot_{q}(C, q, k_{q, f}), l_2, s_2) < l_2$, $k'_{q, f} = k_{q, f}$ otherwise. Then, $recv_C(p, q)$ can be represented as follows:

$$
rhead_C(p, q) = \begin{cases} 
  f_2 f_1^{-1}(\{\text{bot}_{q}(A, p, j_{p, f}) : \min(\text{top}_{q}(A, p, j_{p, f}), \text{top}_{q}(C, q, k_{q, f})) \} : s_1] \cup \nonumber \\
  \{\text{bot}_{q}(A, p, j_{p, f}) + 1 : \text{top}_{q}(A, p, j_{p, f} + 1) : s_1] : \text{top}_{q}(C, q, k_{q, f}) : N_{b_1} \}, \nonumber \\
  \phi, \text{ otherwise.} \nonumber 
\end{cases}
$$

$$
rbody_1^1(p, q) = \begin{cases} 
  f_2 f_1^{-1}(\{\text{nxt}(\text{bot}_{q}(C, q, k_{q, f}'), \text{bot}_{q}(A, p, j_{p, f}), N_{b_1}) : a_1 : N_{b_1 h_{2}} \}, \nonumber \\
  \phi, \text{ otherwise.} \nonumber 
\end{cases}
$$

$$
rbody_2^2(p, q) = f_2 f_1^{-1}(\{\text{nxt}(\text{bot}_{q}(C, q, k_{q, f}'), \text{bot}_{q}(A, p, j_{p, f} + 1), N_{b_1}) : \text{top}_{q}(C, q, k_{q, f}') : N_{b_1} \} : a_1 : N_{b_1 h_{2}} \}, \nonumber 
$$

$$
recv_C(p, q) = rhead_C(p, q) \cup rbody_1^1(p, q) \cup rbody_2^2(p, q). \nonumber $$

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, $f_1(f_2^{-1}(recv_C(1, 0))) = f_1(f_2^{-1}(rhead_C(1, 0))) \cup f_1(f_2^{-1}(rbody_1^1(1, 0))) \cup f_1(f_2^{-1}(rbody_2^2(1, 0)))$, where $f_1(f_2^{-1}(rhead_C(1, 0))) = \{21 : 23 \} \cup \{37 : 39 : 2 : 49 : 16\}$; $f_1(f_2^{-1}(rbody_1^1(1, 0))) = \{183 : 183 : 2 : 745 : 176\}$; and $f_1(f_2^{-1}(rbody_2^2(1, 0))) = \{197 : 199 : 2 : 225 : 16 : 745 : 176\}$. $f_1(f_2^{-1}(recv_C(2, 0))) = f_1(f_2^{-1}(rhead_C(2, 0))) \cup f_1(f_2^{-1}(rbody_1^1(2, 0))) \cup f_1(f_2^{-1}(rbody_2^2(2, 0)))$, where $f_1(f_2^{-1}(rhead_C(2, 0))) = \{11 : 11 : 2 \} \cup \{25 : 27 : 2 : 49 : 16\}$ and $f_1(f_2^{-1}(rbody_2^2(2, 0))) = \{185 : 187 : 2 : 225 : 16 : 745 : 176\}$. □

### 5.3 The Case Where $b_1 = s_1 \ast h_1 \ast h_2$ and $b_2 = s_2 \ast h_2$

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, $recv_{-pc}(p)$, $send_{-c}(p, q)$, and $recv_{-c}(p, q)$ have closed forms. First, we process $send_{-pc}(p)$,
In addition, $f_1(f_2^{-1}(\text{recv}_{C}(p, 0))) = f_1(f_2^{-1}(\text{rhead}_{C}(p, 0))) \cup f_1(f_2^{-1}(\text{rbody}_{C}(p, 0))) \cup f_1(f_2^{-1}(\text{rbody}_{C}^{2}(p, 0)))$, for $1 \leq p \leq 3$.

which is equal to $f_A(f_1(f_2^{-1}(\text{local}_{C}(p) \cap [l_2 : u_2 : s_2])))$. Since $\text{period}_{s_b} = \text{period}_{s}/(Nb_2) = h_1$, $send_{pe}(p)$ can be represented by a union of at most $h_1 + 1$ closed forms:

$$send_{pe}(p) = \left\{ \begin{align*}
[0 : N - 1], & \quad \text{if } u_1 - l_1 + 1 \geq Nb_1 \text{ and } h_1 \geq N; \\
\bigcup_{k = k_{p,f}}^{l_{p,f} + h_{1} - 1} f_A([\text{bot}_{f}(C, p, k) : \text{top}_{f}(C, p, k)]), & \quad \text{if } u_1 - l_1 + 1 \geq Nb_1, h_1 < N, \text{ and } \text{next}(\text{bot}_{f}(C, p, k_{p,f}), l_2, s_2) \geq l_2; \\
\bigcup_{k = k_{p,f}}^{l_{p,f} + h_{1} - 1} f_A([\text{bot}_{f}(C, p, k) : \text{top}_{f}(C, p, k)]) & \cup f_A([\text{bot}_{f}(C, p, k_{p,f} + h_1) : f_1((\text{next}(l_2 + \text{period}_{s} - s_2, l_2, s_2) - l_2)/s_2)]), & \quad \text{if } u_1 - l_1 + 1 \geq Nb_1, h_1 < N, \text{ and } \text{next}(\text{bot}_{f}(C, p, k_{p,f}), l_2, s_2) < l_2; \\
\bigcup_{k = k_{p,f}}^{l_{p,f}} f_A([\text{bot}_{f}(C, p, k) : \text{top}_{f}(C, p, k)]), & \quad \text{if } u_1 - l_1 + 1 < Nb_1.
\end{align*} \right.$$

Note that $send_{pe}(p)$ cannot be represented by a constant number of closed forms independent of $h_1$.  

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, $f_1(f_2^{-1}(\text{recv}_{C}(p, 0))) = f_1(f_2^{-1}(\text{rhead}_{C}(p, 0))) \cup f_1(f_2^{-1}(\text{rbody}_{C}(p, 0))) \cup f_1(f_2^{-1}(\text{rbody}_{C}^{2}(p, 0)))$, for $1 \leq p \leq 3$. 

<table>
<thead>
<tr>
<th>$p=1$</th>
<th>$p=2$</th>
<th>$p=3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 5 6 7</td>
<td>9 8 10</td>
<td>11 12 13 14 15</td>
</tr>
<tr>
<td>0 10 20</td>
<td>24 23 22</td>
<td>21 20 29 30 31</td>
</tr>
<tr>
<td>36 37 38</td>
<td>39 40 41</td>
<td>42 43 44 45 46 47</td>
</tr>
</tbody>
</table>

$a$ is the processor which stores $C(f_1^{l_{12}}(221))$  
$221$ is array index of array $A$
Second, we formulate \( \text{recv}_C(p) \), which is equal to \( f_2(\text{exec}_C(p)) \). Since \( \text{period}_s^A = (\text{period}_s * s_1)/(Nb_1s_2) = 1 \), it is sufficient to analyze the set of PEs which store elements of array \( A \) that will be accessed by elements of array \( C \) within a block of size \( b_1 \):

\[
\text{recv}_C(p) = \begin{cases} 
0 : N - 1, & \text{if } u_1 - h_1 + 1 \geq Nb_1 \text{ and } h_1 \geq N; \\
[ f_C(\text{bot}_f(A, p, j_{pf})) : \text{f}(\text{top}_f(A, p, j_{pf})) ] & \text{if } u_1 - h_1 + 1 \geq Nb_1, h_1 < N, \text{ and } f_C(\text{bot}_f(A, p, j_{pf})) \leq f_C(\text{top}_f(A, p, j_{pf})); \\
[0 : f_C(\text{bot}_f(A, p, j_{pf}))] \cup [ f_C(\text{bot}_f(A, p, j_{pf})) : N - 1], & \text{if } u_1 - h_1 + 1 \geq Nb_1, h_1 < N, \text{ and } f_C(\text{bot}_f(A, p, j_{pf})) > f_C(\text{top}_f(A, p, j_{pf})); \\
f_C([\text{bot}_f(A, p, j_{pf}) : \text{top}_f(A, p, j_{pf})]) \cup f_C([\text{bot}_f(A, p, j_{pf}) : \text{top}_f(A, p, j_{pf})]), & \text{if } u_1 - h_1 + 1 < Nb_1.
\end{cases}
\]

Third, we deal with \( \text{send}_C(p, q) \), which is equal to \( \text{local}_C(p) \cap f_2(\text{exec}_C(q)) \). This set can be represented by a union of three closed forms: \( \text{shead}_C(p, q) \), \( \text{sbody}_C^1(p, q) \), and \( \text{sbody}_C^2(p, q) \). Let \( j_{pf} = j_{pf} + 1 \) if \( \text{next}(\text{bot}_i(A, q, j_{pf}), l_1, s_1) < l_1, j_{pf} = j_{pf} \) otherwise.

\[
\text{shead}_C(p, q) = \begin{cases} 
[\text{bot}_a(C, p, k_{pf}), \text{min}\{\text{top}_a(C, p, k_{pf}), \text{top}_a(A, q, j_{pf})\}] : s_2 & \cup \\
[[\text{bot}_a(C, p, k_{pf} + 1), \text{top}_a(C, p, k_{pf} + 1)] : s_2] & : \text{top}_f(A, q, j_{pf}) : Nb_2, \\
\phi & \text{otherwise}.
\end{cases}
\]

\[
\text{sbody}_C^1(p, q) = \begin{cases} 
[[\text{bot}_f(A, q, j_{pf}), \text{next}(\text{bot}_f(A, q, j_{pf}), \text{top}_a(C, p, k_{pf}), Nb_2)] : s_2] & : u_2 : Nb_2h_1, \\
\phi & \text{otherwise}.
\end{cases}
\]

\[
\text{sbody}_C^2(p, q) = \begin{cases} 
[[\text{next}(\text{bot}_f(A, q, j_{pf}), \text{bot}_a(C, p, k_{pf} + 1), Nb_2)] : s_2] & : t_{topf}(A, q, j_{pf}, \text{bot}_a(C, p, k_{pf} + 1), Nb_2 + s_2(h_2 - 1) : s_2] : u_2 : Nb_2h_1, \\
\phi & \text{otherwise}.
\end{cases}
\]

\[
\text{send}_C(p, q) = \text{shead}_C(p, q) \cup \text{sbody}_C^1(p, q) \cup \text{sbody}_C^2(p, q).
\]

Fourth, we manage \( \text{recv}_C(p, q) \), which is equal to \( \text{send}_C(q, p) \). Hence, it also can be represented by a union of three closed forms. As stated before, we prefer that \( \text{recv}_C(p, q) \) be represented based on indices of array \( A \). Let \( j_{pf} = j_{pf} + 1 \) if \( \text{next}(\text{bot}_i(A, p, j_{pf}), l_1, s_1) < l_1, j_{pf} = j_{pf} \) otherwise. Then, we have

\[
\text{rhead}_C(p, q) = \begin{cases} 
f_2f_1^{-1}(\text{min}\{\text{top}_f(C, q, k_{pf}), \text{top}_a(A, p, j_{pf})\}) : s_1 & \cup \\
[[\text{top}_f(C, q, k_{pf}), \text{top}_a(A, p, j_{pf} + 1)] : s_1] : \text{top}_f(A, p, j_{pf} + 1) : Ns_1h_2), \\
\phi & \text{otherwise}.
\end{cases}
\]
Have closed forms.

Note that the above closed form has two exceptions. First, when $b_1 \neq 1$, let $b_1 = s_1 \cdot h_1$, $b_2 = s_2 \cdot h_2$, 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:

$$recv_{pe}(p) = \begin{cases} 
\{0 : N - 1\}, & \text{if } u_2 - l_2 + 1 \geq Nb_2 \text{ and } h_2 \geq N; \\
[f_C(bot_{f}(A, p, j_{p,f})) : \\
\quad f_C(bot_{f}(A, p, j_{p,f})) + \min\{N - 1, (j_{p,f} - j_{p,f})N/h_2) : N/h_2\} \text{ mod } N, \\
\quad \text{if } h_2 < N \text{ and } f_C(bot_{f}(A, p, j_{p,f})) = f_C(top_{f}(A, p, j_{p,f})), \text{ for all } j_{p,f} \leq j \leq j_{p,f} + 1; \\
[[f_C(top_{f}(A, p, j_{p,f})) - 1 : f_C(top_{f}(A, p, j_{p,f}))]] : \\
\quad f_C(top_{f}(A, p, j_{p,f})) + \min\{N - 2, (j_{p,f} - j_{p,f})N/h_2) : N/h_2\} \text{ mod } N, \\
\quad \text{if } h_2 < N \text{ and } f_C(bot_{f}(A, p, j_{p,f})) \neq f_C(top_{f}(A, p, j_{p,f})), \text{ for some } j_{p,f} \leq j \leq j_{p,f} + 1. 
\end{cases}$$

Note that the above closed form has two exceptions. First, when $f_C(bot_{f}(A, p, j_{p,f})) = f_C(top_{f}(A, p, j_{p,f}))$, $((f_C(top_{f}(A, p, j_{p,f})) - 1) \text{ mod } N)$ is not in $recv_{pe}(p)$. Second, when $u_2 - l_2 + 1 < Nb_2$ and $f_C(bot_{f}(A, p, j_{p,f})) = f_C(top_{f}(A, p, j_{p,f}))$, then $((f_C(top_{f}(A, p, j_{p,f})) + (j_{p,f} - j_{p,f})N/h_2) \text{ mod } N)$ is not in $recv_{pe}(p)$.

In the second case, let $b_1 = s_1 \cdot h_1 \cdot h_2$, $b_2 = s_2 \cdot h_2$, 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 $send_{\text{pe}}(p)$ also can be represented by closed forms:

$$send_{\text{pe}}(p) = \begin{cases} [0 : N - 1], & \text{if } u_1 - l_1 + 1 \geq Nb_1 \text{ and } h_1 \geq N; \\ [f_A(\text{bot}_f(C, p, k_{pf})): \\
\quad f_A(\text{bot}_f(C, p, k_{pf})) + \min\{N - 1, (k_{pi} - k_{pf})N/h_1\} : N/h_1 \text{ mod } N], & \text{if } h_1 < N \text{ and } f_A(\text{bot}_f(C, p, k)) = f_A(\text{top}_f(C, p, k)), \text{ for all } k_{pf} \leq k \leq k_{pf} + 1; \\ [([f_A(\text{top}_f(C, p, k_{pf})) - 1 : f_A(\text{top}_f(C, p, k_{pf}))]) \\
\quad f_A(\text{top}_f(C, p, k_{pf})) + \min\{N - 2, (k_{pi} - k_{pf})N/h_1 : N/h_1 \text{ mod } N], & \text{if } h_1 < N \text{ and } f_A(\text{bot}_f(C, p, k)) \neq f_A(\text{top}_f(C, p, k)), \text{ for some } k_{pf} \leq k \leq k_{pf} + 1. 
\end{cases}$$

Note that the above closed form also has two exceptions. First, when $f_A(\text{bot}_f(C, p, k_{pf})) = f_A(\text{top}_f(C, p, k_{pf})) \mod N$ is not in $send_{\text{pe}}(p)$. Second, when $u_1 - l_1 + 1 < N b_1$ and $f_A(\text{bot}_f(C, p, k_{pf})) = f_A(\text{top}_f(C, p, k_{pf})) \mod N$ is not in $send_{\text{pe}}(p)$.

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:

\[
\text{forall } i = 0, 80639 \\
A(1997 + i \cdot s_1) = C(5 + i \cdot s_2),
\]

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 \( \left\lfloor \frac{b_2}{s_2} \right\rfloor \leq \left\lfloor \frac{(N-1)b_2+1}{s_2} \right\rfloor \), and where \( N \) is the number of PEs; the other cases where \( \frac{b_2}{s_2} \leq \left\lfloor \frac{(N-1)b_2+1}{s_2} \right\rfloor \) can be presented in a similar way. The experimental results can be summarized as follows.

<table>
<thead>
<tr>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( I )</th>
<th>( II )</th>
<th>row-wise method</th>
<th>lattice method</th>
<th>closed-form method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>conventional</td>
<td>deposit</td>
<td>conventional</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>15</td>
<td>3</td>
<td>116 (110)</td>
<td>126 (116)</td>
<td>97 (90)</td>
</tr>
<tr>
<td>19</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>63 (48)</td>
<td>67 (45)</td>
<td>60 (49)</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>21</td>
<td>3</td>
<td>115 (109)</td>
<td>123 (112)</td>
<td>86 (80)</td>
</tr>
<tr>
<td>62</td>
<td>43</td>
<td>129</td>
<td>9</td>
<td>292 (287)</td>
<td>212 (200)</td>
<td>91 (85)</td>
</tr>
<tr>
<td>163</td>
<td>42</td>
<td>1</td>
<td>1</td>
<td>35 (34)</td>
<td>52 (49)</td>
<td>35 (34)</td>
</tr>
<tr>
<td>64</td>
<td>41</td>
<td>123</td>
<td>9</td>
<td>282 (276)</td>
<td>206 (195)</td>
<td>92 (86)</td>
</tr>
<tr>
<td>314</td>
<td>209</td>
<td>47</td>
<td>42</td>
<td>54 (51)</td>
<td>56 (49)</td>
<td>70 (67)</td>
</tr>
<tr>
<td>315</td>
<td>210</td>
<td>1</td>
<td>1</td>
<td>33 (29)</td>
<td>43 (35)</td>
<td>33 (29)</td>
</tr>
<tr>
<td>316</td>
<td>211</td>
<td>47</td>
<td>42</td>
<td>54 (50)</td>
<td>57 (49)</td>
<td>71 (68)</td>
</tr>
<tr>
<td>314</td>
<td>2519</td>
<td>47</td>
<td>47</td>
<td>124 (118)</td>
<td>127 (117)</td>
<td>1113 (1100)</td>
</tr>
<tr>
<td>315</td>
<td>2520</td>
<td>12</td>
<td>1</td>
<td>58 (52)</td>
<td>86 (76)</td>
<td>43 (38)</td>
</tr>
<tr>
<td>316</td>
<td>2521</td>
<td>47</td>
<td>47</td>
<td>127 (121)</td>
<td>129 (118)</td>
<td>1145 (1140)</td>
</tr>
<tr>
<td>3779</td>
<td>209</td>
<td>3</td>
<td>42</td>
<td>56 (50)</td>
<td>84 (73)</td>
<td>1338 (1333)</td>
</tr>
<tr>
<td>3780</td>
<td>210</td>
<td>1</td>
<td>1</td>
<td>49 (44)</td>
<td>79 (68)</td>
<td>50 (45)</td>
</tr>
<tr>
<td>3781</td>
<td>211</td>
<td>3</td>
<td>42</td>
<td>56 (50)</td>
<td>84 (73)</td>
<td>1286 (1280)</td>
</tr>
<tr>
<td>3779</td>
<td>2521</td>
<td>3</td>
<td>37</td>
<td>39 (37)</td>
<td>62 (58)</td>
<td>305 (303)</td>
</tr>
<tr>
<td>3780</td>
<td>2520</td>
<td>1</td>
<td>1</td>
<td>38 (36)</td>
<td>61 (57)</td>
<td>38 (36)</td>
</tr>
<tr>
<td>3781</td>
<td>2519</td>
<td>3</td>
<td>37</td>
<td>38 (36)</td>
<td>61 (57)</td>
<td>294 (292)</td>
</tr>
</tbody>
</table>

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. \textit{communication overhead} = \textit{execution time − pure computation time}. Factor (\( I \)) = \( \min\{j_{pl} - j_{p}, period_{sl}^{A}\} \) and factor (\( II \)) = \( \min\{j_{pl} - j_{p}, period_{el}^{A}\} \ast \frac{period_{sl}}{(period_{el} \ast N)}\). Cases indicated by \( \dagger \) 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_{eb}^{A}\}, 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_{eb}^{A}\} \times [period_{s}/(period_{eb}^{A} + N)], 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_{eb}^{A}\} times for computing send_{C}(p, q), and for each time roughly \[period_{s}/(period_{eb}^{A} + N)\] lattice points will be retrieved in the first period. We find that, if factor (I) > 1.5 \times factor (II), then the lattice method is more effective than the row-wise version algorithm; otherwise, if factor (I) < 1.5 \times 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

Indices of $A = g^2l_A \{f_1f_2^{-1}[l2gC(\text{indices of } C \text{ at } PE_p)]\},$

where $l2gC(i, p) = ([i/b_2] * N + p) * b_2 + (i \text{ mod } b_2) + c_1$ means the function of transforming an index of array $C$ at $PE_p$ from a local name space to a global name space; $g^2l_A(i) = [(i - a_1)/(N * b_1)] * b_1 + ((i - a_1) \text{ mod } b_1)$ 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 $> \text{ block sizes}$, 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 LHS array entries at the sending end.

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$b_1$</th>
<th>$s_2$</th>
<th>$b_2$</th>
<th>(I)</th>
<th>(II)</th>
<th>row-wise method</th>
<th>lattice method</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>105 (93)</td>
<td>98 (76)</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>9</td>
<td>7</td>
<td>49</td>
<td>7</td>
<td>104 (101)</td>
<td>85 (79)</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>21</td>
<td>7</td>
<td>88 (85)</td>
<td>82 (75)</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>54</td>
<td>9</td>
<td>104 (101)</td>
<td>83 (78)</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>36</td>
<td>9</td>
<td>88 (85)</td>
<td>73 (68)</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>55</td>
<td>11</td>
<td>127 (124)</td>
<td>116 (110)</td>
</tr>
<tr>
<td>23</td>
<td>19</td>
<td>11</td>
<td>7</td>
<td>161</td>
<td>23</td>
<td>330 (327)</td>
<td>261 (193)</td>
</tr>
</tbody>
</table>

Table 7: Experimental study 1 when strides $s_i > \text{ block sizes } b_i$.

6.2 Saxpy Operation

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

```plaintext
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.

<table>
<thead>
<tr>
<th>$b_2$</th>
<th>$b_1$</th>
<th>3</th>
<th>9</th>
<th>63</th>
<th>315</th>
<th>945</th>
<th>3780</th>
<th>7560</th>
<th>15120</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2 PE</td>
<td>876</td>
<td>1241</td>
<td>879</td>
<td>829</td>
<td>821</td>
<td>821</td>
<td>821</td>
<td>821</td>
</tr>
<tr>
<td>4</td>
<td>4 PE</td>
<td>438</td>
<td>412</td>
<td>476</td>
<td>428</td>
<td>420</td>
<td>417</td>
<td>417</td>
<td>417</td>
</tr>
<tr>
<td>8</td>
<td>8 PE</td>
<td>220</td>
<td>267</td>
<td>274</td>
<td>225</td>
<td>217</td>
<td>214</td>
<td>213</td>
<td>213</td>
</tr>
<tr>
<td>16</td>
<td>16 PE</td>
<td>110</td>
<td>108</td>
<td>173</td>
<td>125</td>
<td>117</td>
<td>113</td>
<td>113</td>
<td>295</td>
</tr>
<tr>
<td>2</td>
<td>2 PE</td>
<td>1238</td>
<td>541</td>
<td>564</td>
<td>516</td>
<td>508</td>
<td>508</td>
<td>506</td>
<td>506</td>
</tr>
<tr>
<td>4</td>
<td>4 PE</td>
<td>407</td>
<td>271</td>
<td>318</td>
<td>270</td>
<td>262</td>
<td>259</td>
<td>259</td>
<td>258</td>
</tr>
<tr>
<td>8</td>
<td>8 PE</td>
<td>205</td>
<td>136</td>
<td>125</td>
<td>145</td>
<td>137</td>
<td>134</td>
<td>134</td>
<td>133</td>
</tr>
<tr>
<td>16</td>
<td>16 PE</td>
<td>106</td>
<td>68</td>
<td>65</td>
<td>85</td>
<td>77</td>
<td>74</td>
<td>73</td>
<td>134</td>
</tr>
<tr>
<td>2</td>
<td>2 PE</td>
<td>875</td>
<td>582</td>
<td>349</td>
<td>380</td>
<td>371</td>
<td>369</td>
<td>368</td>
<td>367</td>
</tr>
<tr>
<td>4</td>
<td>4 PE</td>
<td>472</td>
<td>326</td>
<td>175</td>
<td>199</td>
<td>193</td>
<td>189</td>
<td>189</td>
<td>189</td>
</tr>
<tr>
<td>8</td>
<td>8 PE</td>
<td>270</td>
<td>125</td>
<td>88</td>
<td>89</td>
<td>102</td>
<td>99</td>
<td>99</td>
<td>98</td>
</tr>
<tr>
<td>16</td>
<td>16 PE</td>
<td>172</td>
<td>65</td>
<td>45</td>
<td>47</td>
<td>51</td>
<td>57</td>
<td>56</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>2 PE</td>
<td>826</td>
<td>534</td>
<td>395</td>
<td>369</td>
<td>347</td>
<td>349</td>
<td>349</td>
<td>348</td>
</tr>
<tr>
<td>4</td>
<td>4 PE</td>
<td>426</td>
<td>278</td>
<td>209</td>
<td>169</td>
<td>161</td>
<td>179</td>
<td>179</td>
<td>179</td>
</tr>
<tr>
<td>8</td>
<td>8 PE</td>
<td>223</td>
<td>149</td>
<td>89</td>
<td>85</td>
<td>84</td>
<td>94</td>
<td>94</td>
<td>94</td>
</tr>
<tr>
<td>16</td>
<td>16 PE</td>
<td>124</td>
<td>87</td>
<td>47</td>
<td>43</td>
<td>44</td>
<td>48</td>
<td>54</td>
<td>55</td>
</tr>
<tr>
<td>2</td>
<td>2 PE</td>
<td>819</td>
<td>527</td>
<td>387</td>
<td>357</td>
<td>297</td>
<td>344</td>
<td>345</td>
<td>345</td>
</tr>
<tr>
<td>4</td>
<td>4 PE</td>
<td>419</td>
<td>271</td>
<td>201</td>
<td>162</td>
<td>184</td>
<td>177</td>
<td>177</td>
<td>177</td>
</tr>
<tr>
<td>8</td>
<td>8 PE</td>
<td>216</td>
<td>141</td>
<td>107</td>
<td>84</td>
<td>93</td>
<td>93</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>16</td>
<td>16 PE</td>
<td>116</td>
<td>79</td>
<td>52</td>
<td>44</td>
<td>44</td>
<td>46</td>
<td>46</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>2 PE</td>
<td>819</td>
<td>525</td>
<td>384</td>
<td>363</td>
<td>359</td>
<td>367</td>
<td>336</td>
<td>337</td>
</tr>
<tr>
<td>4</td>
<td>4 PE</td>
<td>417</td>
<td>268</td>
<td>198</td>
<td>188</td>
<td>187</td>
<td>155</td>
<td>160</td>
<td>175</td>
</tr>
<tr>
<td>8</td>
<td>8 PE</td>
<td>213</td>
<td>139</td>
<td>163</td>
<td>99</td>
<td>84</td>
<td>78</td>
<td>82</td>
<td>83</td>
</tr>
<tr>
<td>16</td>
<td>16 PE</td>
<td>113</td>
<td>76</td>
<td>59</td>
<td>49</td>
<td>44</td>
<td>40</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>2</td>
<td>2 PE</td>
<td>818</td>
<td>524</td>
<td>383</td>
<td>362</td>
<td>358</td>
<td>361</td>
<td>299</td>
<td>335</td>
</tr>
<tr>
<td>4</td>
<td>4 PE</td>
<td>416</td>
<td>268</td>
<td>197</td>
<td>187</td>
<td>185</td>
<td>163</td>
<td>161</td>
<td>175</td>
</tr>
<tr>
<td>8</td>
<td>8 PE</td>
<td>213</td>
<td>138</td>
<td>163</td>
<td>98</td>
<td>99</td>
<td>83</td>
<td>76</td>
<td>85</td>
</tr>
<tr>
<td>16</td>
<td>16 PE</td>
<td>113</td>
<td>76</td>
<td>58</td>
<td>56</td>
<td>46</td>
<td>42</td>
<td>39</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>2 PE</td>
<td>817</td>
<td>522</td>
<td>380</td>
<td>361</td>
<td>357</td>
<td>358</td>
<td>358</td>
<td>295</td>
</tr>
<tr>
<td>4</td>
<td>4 PE</td>
<td>415</td>
<td>268</td>
<td>196</td>
<td>186</td>
<td>184</td>
<td>187</td>
<td>164</td>
<td>149</td>
</tr>
<tr>
<td>8</td>
<td>8 PE</td>
<td>212</td>
<td>138</td>
<td>102</td>
<td>97</td>
<td>97</td>
<td>84</td>
<td>86</td>
<td>75</td>
</tr>
<tr>
<td>16</td>
<td>16 PE</td>
<td>293</td>
<td>135</td>
<td>66</td>
<td>57</td>
<td>56</td>
<td>44</td>
<td>44</td>
<td>38</td>
</tr>
</tbody>
</table>

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 $b_1 = s_1 \times h$ and $b_2 = s_2 \times h \times h'$ was close to that of cases where $b_1 = s_1 \times h \times h'$ and $b_2 = s_2 \times h$.

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 $PE_p ([bot_j(C, p, k) : top_j(C, p, k)])$ intersected with at most one referenced block of array $A$ in $PE_q ([bot_j(A, q, j) : top_j(A, q, j) : s_2])$, where
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:

\[
\text{forall } i = 0, 241919 \quad A(i) = OLD_A(i),
\]

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.

\[
\begin{array}{cccccccccc}
\text{Table 9: Execution time (millisecond) of performing the data re-distribution operation using 2 PEs, 4 PEs, 8 PEs, and 16 PEs, respectively. Array } A \text{ was distributed by a } cyclic(b_1) \text{ distribution; array } OLD_A \text{ was distributed by a } cyclic(b_2) \text{ distribution.}
\end{array}
\]

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\((b_1)\)) distributions; however, none of them obtained closed-form representations. Stichnoth \textit{et al.} pointed out that a cyclic\((b_1)\) distribution can be regarded as a union of \( b_1 \) 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 \( b_1 \ast b_2 \) closed forms [36]. Gupta \textit{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 \textit{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 \textit{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 \textit{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 \( r_2 = r \cdot s_2 / (Nb) \) rows, and that the access pattern of the LHS array repeated after every \( r_1 = r \cdot s_1 / (Nb) \) rows, where \( r = \text{lcm}(Nb / \text{gcd}(s_1, Nb), Nb / \text{gcd}(s_2, Nb)) \). 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 \( \text{strides} > \text{blocksizes} \). 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 \( A(0 : n \cdot s_1 : s_1) = B(0 : n \cdot s_2 : s_2) \), 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 : 12), ..., A(n : n2)
!HPF$ PROCESSORS PES(N)
!HPF$ TEMPLATE T(:)
!HPF$ ALIGN A(1) WITH T(d1 * i + e1)
  :  
!HPF$ ALIGN A(n) WITH T(dn * i + en)
!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 $t = \min_{j \in J} \{d_j * a_{j1} + e_j\}$, 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.

<table>
<thead>
<tr>
<th>processor ID</th>
<th>two-level mapping</th>
<th>our method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$([(d_j \times i + e_j - t) \mod (bN)]/b)$</td>
<td>$[((i - a_{j1}) \mod (bN))/b_j]$</td>
</tr>
<tr>
<td>block</td>
<td>$[((d_j \times i + e_j - t)/bN)]$</td>
<td>$((i - a_{j1}) / (b_j N))$</td>
</tr>
<tr>
<td>offset</td>
<td>$[((d_j \times i + e_j - t) \mod b)/d_j]$</td>
<td>$(i - a_{j1}) \mod b_j$</td>
</tr>
<tr>
<td>max block size</td>
<td>$[b/d_j]$</td>
<td>$b_j$</td>
</tr>
</tbody>
</table>

Table 10: The corresponding local address for element $A_j(i)$ based on two models, where $t = \min_{ \{y, j\}} \{d_j \times a_{j1} + e_j\}$, 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 $b/(d_1 \times s_1)$ is a multiple of $b/(d_2 \times s_2)$ or when $b/(d_1 \times s_1)$ is a factor of $b/(d_2 \times s_2)$. That is, $b$ must be a multiple of both $d_1 \times s_1$ and $d_2 \times s_2$; in addition, either $d_1 \times s_1$ is a factor of $d_2 \times s_2$, or $d_1 \times s_1$ is a multiple of $d_2 \times s_2$. However, $d_1 \times s_1$ and $d_2 \times s_2$ 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 $t = \min_{ \{y, j\}} \{d_j \times a_{j1} + e_j\}$. 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 $a'_{j1} = (a_{j1} - \lfloor (d_j \times a_{j1} + e_j - t) / d_j \rfloor)$. That is, we can extend array $A_j(a_{j1} : a_{j2})$ to $A_j(a_{j1} - \lfloor (d_j \times a_{j1} + e_j - t) / d_j \rfloor : a_{j2})$, so that the first array element $A_j(a_{j1} - \lfloor (d_j \times a_{j1} + e_j - t) / d_j \rfloor)$ is
aligned to the template cell $T(d_j \ast a_j + e_j - d_j * [(d_j * a_j + e_j - l)/d_j])$, which is very close to the first template cell $T(t)$. Note that, the additional boundary array elements $A_j(d'_j : a_j - 1)$ 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


The memory access sequence of $A[0 \pm \frac{p}{2}]$, which represents elements of array $A$ and will be used to compute $A(y)$.

$PE_0$ forms a lattice.

Figure 5: (a) The memory access sequence of $A(4 + i \pm 2)$, 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) $bot(A, 0, 1) : u_2 : \text{period}_A + s_2 = [14 : 314 : 27]$ forms a lattice. (b) $bot(A, 0, 2) : u_2 : \text{period}_A + s_2 = [27 : 314 : 27]$ forms a lattice.