An Expression-Rewriting Framework to Generate Communication Sets for HPF Programs with Block-Cyclic Distribution

Gwan-Hwan Hwang   Jenq Kuen Lee
Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan

Abstract

In this paper, we present a new framework based on expression rewritings and a calculus form called CSD calculus to generate the local enumeration set and communication set for HPF programs with Block-Cyclic distribution. Our framework is a practical software framework, and can handle the general cases so that the communication set of HPF programs of "Block-Cyclic" distributions with two-level alignments (or multiple-level alignments), multi-dimensional arrays, array intrinsic functions (such as Transpose operation), and affine indexes and axis exchange in the array subscript, can be calculated in a systematic way with a sound software foundation. Previously, existing work do not report a software framework to solve a problem with such general cases. In addition, our expression-rewriting framework is based on a new representative form, CSD (common-stride descriptor), to describe the regularity of the access patterns of HPF programs with "Block-Cyclic" distribution. We also demonstrate a calculus of CSD that CSD is closed under intersection and normalization, which helps the process of calculating local enumeration and communication sets of HPF programs with Block-Cyclic distributions. Experimental results show that our software scheme not only can be easily implemented in the practice, but also is with good efficiency.

Keywords: HPF Compiler, Distributed Memory Machines, Compiler Optimization, Parallelizing Compiler, Block-Cyclic Distributions.

1 Introduction

An increasing number of programming languages, such as Fortran D, HPF (High Performance Fortran)[8], and parallel C++[20, 2], are providing distributed arrays to support a global name space on distributed memory architectures. In those languages (such as HPF), programmers can specify the distribution of the distributed arrays among processors, and the compiler can then take distribution information (how data are distributed among processors), and generate the communication codes[3, 16, 19] for programs to emulate the shared memory space on distributed memory architectures. The distribution pattern specified by programmers in HPF includes "Block" pattern which has each processor get a consecutive block of elements, "Cyclic" distribution which is a round-robin rotation, "Collapse" distribution which mean no distribution, or"Block-Cyclic" also known as cyclic(k) distribution which is a round-robin rotation, but each time each processor takes a small block number of elements. The generation of the communication set for HPF programs with block, cyclic, or collapse distribution has been implemented with success in commercial HPF compilers, however few commercial HPF compilers have so far provided efficient support for block-cyclic

[1]This paper is submitted to *International Parallel Processing Symposium, 1998*. The correspondence author of this work is Jenq Kuen Lee. His e-mail address is jklee@cs.nthu.edu.tw, FAX number is 886-3-5723694, phone number is 886-3-5715131 EXT. 3519, and postal address is Prof. Jenq Kuen Lee, Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan.

distribution[11]. Nevertheless, block-cyclic distribution is very important for expressing "block-scatter" distribution in the design of scalable libraries for linear algebra computation[6]. The lack of commercial compiler support for HPF programs with block-cyclic distributions has recently prompted many research efforts[9, 10, 11, 13, 14, 28] to hope to lead to efficient schemes in supporting this issue.

While we think the recent research efforts will greatly help the implementation of block-cyclic distributions, we feel the lack of support in commercial compilers for block-cyclic distributions is in part due to the lack of a practical software framework which can lead to commercial strength implementation in a straightforward manner in the practice to support the communication set of the block-cyclic distribution with general cases. The general case of the block-cyclic distribution will be at least with multiple-level alignments, multi-dimensional arrays, array intrinsic functions (such as Transpose operation), and affine indexes and axis exchange in the array subscript. We show a typical example of such cases below in *Code Fragment 1*:

HPF Code Fragment 1

!HPF$ ALIGN A($i_1, i_2, \ldots, i_n$) WITH T1($\mu_1(i_1, \ldots, i_n), \ldots, \mu_n(i_1, \ldots, i_n)$)
!HPF$ ALIGN B($i_1, i_2, \ldots, i_n$) WITH T2($\nu_1(i_1, \ldots, i_n), \ldots, \nu_n(i_1, \ldots, i_n)$)
!HPF$ ALIGN T1($i_1, i_2, \ldots, i_n$) WITH T3($w_1(i_1, \ldots, i_n), \ldots, w_n(i_1, \ldots, i_n)$)
!HPF$ ALIGN T2($i_1, i_2, \ldots, i_n$) WITH T4($w'_1(i_1, \ldots, i_n), \ldots, w'_n(i_1, \ldots, i_n)$)
!HPF$ PROCESSOR PROCS($d_1, d_2, \ldots, d_n$)
!HPF$ DISTRIBUTE T3(CYCLIC(b_1),...,CYCLIC(b_n)) ONTO PROCS
!HPF$ DISTRIBUTE T4(CYCLIC(b'_1),...,CYCLIC(b'_n)) ONTO PROCS
⋮
FORALL ($i_1 = L_1 : U_1 : S_1, i_2 = L_2 : U_2 : S_2, \ldots, i_n = L_n : U_n : S_n$)
A($f_1(i_1, i_2, \ldots, i_n), \ldots, f_n(i_1, i_2, \ldots, i_n)$)=$\mathcal{F}$(B($g_1(i_1, i_2, \ldots, i_n), \ldots, g_n(i_1, i_2, \ldots, i_n)$))
END FORALL

In the code above, array A and B are first aligned to template array T1 and T2, respectively. T1 and T2 are then aligned to template array T3 and T4, respectively. Next, T3 and T4 are distributed among processors by block-cyclic distribution. Therefore, it's a program of multipl-level alignments. The functions, f_i and g_i are all affine functions to be used in the array subscript, and $\mathcal{F}$ is an array function or array intrinsic function (such as Transpose operation). In this paper, we present a software framework based on expression rewritings and a calculus form called CSD calculus to hope to lead to commercial strength implementation for HPF programs of block-cyclic distributions with multiple-level alignments, multi-dimensional arrays, array intrinsic functions (such as Transpose operation), and affine indexes and axis exchange in the array subscript.

Our expression-rewriting framework is based on a new representative form called CSD to represent the set of access elements in HPF programs with block-cyclic distribution. Previously, research community used RSD (regular section descriptor) to describe the set of access elements in HPF programs with "Block" or "Cyclic" distribution patterns. RSD is not sufficient to represent the access elements with "Block-Cyclic" distributions. Later, researchers[25] tried to use BSD to represent the access elements of "Block-Cyclic" distributions. Unfortunately, BSD is not closed under intersection or index normalization. Therefore, BSD is not appropriate for calculating communication set of HPF programs with "Block-Cyclic" distributions. On the other hand, CSD (common-stride descriptor) is a new representative form we propose to describe the regularity of the access patterns of HPF programs with "Block-Cyclic" distribution. We will demonstrate a calculus of CSD that CSD is closed under intersection and normalization. In addition, we present a five-step framework with mechanic arithmetic-rewritings to accompany the CSD calculus so that the communication set of HPF programs of "Block-Cyclic" distributions with two-level alignments (or multiple-level alignments) and axis alignments, multi-dimensional arrays, array intrinsic functions (such as Transpose operation), and affine indexes and axis exchange in the array subscripts, can be calculated in a straightforward way with a sound software foundation. Previously, when dealing with multi-dimensional cases, existing work tried to use a dimension-wise scheme to calculate the communication set at each dimension, separately. They, however, do not address the complicated software issues when multiple level alignments, axis alignments, and data movements between different dimensions are involved. Our five-step framework can perform mechanic arithmetic-rewritings to calculate communication sets to deal with such complicated cases. Our software framework is currently being incorporated into an experimental HPF II compiler which is under development at our university as a research prototype to explore compiler technologies[15, 21, 16, 23, 5, 22] for distributed environments. Preliminary experimental results with our CSD calculus and expression-rewriting framework show that our framework not only can be easily implemented in the practice to handle the very general cases, but also is with good efficiency.

The remainder of the paper is organized as follows. Section 2 describes the CSD calculus which lays the foundation of our framework. Section 3 then presents our five-step expression-rewriting framework to generate communication sets for HPF programs with block-cyclic distributions. Next, in Section 4, we use an example to show how our scheme works. Section 5 then presents a global to local mapping mechanism for HPF programs of block-cyclic distributions with multiple-level alignments. The work is important for code generations. Finally, Section 6 gives the experimental results and Section 7 concludes this paper.

2 CSD Calculus

In this section, we give definitions for RSD, BSD, and CSD representations. The representations will be used as a basis for our proposed five-step framework (which will be described in next section) with mechanic arithmetic-rewritings to generate communication set of HPF programs for "Block-Cyclic" distributions with multiple-level alignments, multi-dimensional arrays, and affine indexes and axis exchange in the array subscript.

RSD (regular section descriptor) describes the set of access elements in HPF programs with "Block" or "Cyclic" distribution patterns. RSD is not sufficient to represent the access elements with "Block-Cyclic" distributions. Researchers tried to use BSD to represent the access elements of "Block-Cyclic" distributions. Unfortunately, BSD is not closed under intersection or index normalization. Therefore, BSD is not appropriate for calculating communication set of HPF programs with "Block-Cyclic" distributions. On the other hand, CSD (common-stride descriptor) is a new representative form we propose to describe the regularity of the access patterns of HPF programs with "Block-Cyclic" distribution. We will demonstrate below a calculus of CSD that CSD is closed under intersection and normalization.

RSD is defined below.

Definition 1 *A* **regular section descriptor (RSD)** *represents a sequences of integer indices. It is with a triplet notation and defined as follows:*

$$(l : u : s) = \{\ k \mid l \leq k \leq u \ \textit{and}\ k = l + s * I,\ \textit{where}\ I\ \textit{is a non-negative integer.}\}$$

where l, u, and s are referred to as the lower bound, upper bound, and stride, respectively.

Researchers proposed a quadruplet notation (BSD) similar to RSD to represent the block-cyclic distribution of HPF [10, 25]. BSD is defined below.

Definition 2 *A* **block-cyclic section descriptor (BSD)** *represents a set of integer indices. It's with a quadruplet notation and defined as follows:*

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
P1	P1	P1	P2	P2	P2	P3	P3	P3	P1	P1	P1	P2	P2	P2	P3	P3	P3	P1	P1	P1	P2	P2	P2	P3	P3	P3	P1	P1	P1	P2	P2	P2	P3	P3	P3

Figure 1: Block-Cyclic Distribution of Array B

$$(l : u : b : c) = \bigcup_{k=0}^{b-1} (l + k : u : c),$$

where l, u, b, and c are the lower bound, upper bound, block width, and cyclic lenth, respectively.

We have an example of BSD below.

Example 1 *As shown in Figure 1, array B is with 36 elements and distributed among processors by cyclic(3) distribution. We have A(1:36:3:9), A(4:36:3:9), and A(7:36:3:9) distributed among P1,P2 and P3 respectively. As an example, the data section owned by P1 is (1:36:3:9)= (1:36:9) ∪ (2:36:9) ∪ (3:36:9)= {1,10,19,28} ∪ {2,11,20,29} ∪ {3,12,21,30} = {1,2,3,10,11, 12,19,20,21,28,29,30}.*

Unfortunately, BSD is not closed under intersection. Therefore, we propose a new representative form, CSD, defined below.

Definition 3 *A* **Common-Stride Section Descriptor(CSD)** *is a set of indices. It is a union of N (N is a positive integer) RSDs which are all with the same stride, and the difference between the lower bounds of each pair of RSDs is less than the stride. CSD is represented by a triplet notation as follows:*

$$((l_1, l_2, \ldots, l_N) : (u_1, u_2, \ldots, u_N) : S) = \bigcup_{k=1}^{N} (l_k : u_k : S),$$

where

(1) $\forall\ i \text{ and } j, 1 \le i \le N \text{ and } 1 \le j \le N, \text{ and } i \ne j \Rightarrow l_i \ne l_j$,

(2) $|l_i - l_j| < S, 1 \le i \le N \text{ and } 1 \le j \le N$.

The properties of the normalization and intersection operations of the three kinds of section descriptors are shown in *Table 1*. The CSD is closed under normalization and intersection operations. The normalization operation is to ask the set of elements representing i when given the set of elements representing $\alpha * i + \beta$, where α and β are integer constants.

	Normalization
RSD	RSD
BSD	CSD
CSD	CSD

∩	RSD	BSD	CSD
RSD	RSD	CSD	CSD
BSD	CSD	CSD	CSD
CSD	CSD	CSD	CSD

Table 1: Normalization and Intersection of RSD, BSD and CSD

We present the properties of the CSD below. Theorem 1 first describes the relationships among RSD, BSD, and CSD.

Theorem 1 *A set represented by a RSD can be represented by a BSD. Similarly, a set represented by a BSD can be represented by a CSD.*
Proof: See *Appendix A*.

Next, Theorem 2 and 3 describe the intersection properties with CSD.

Theorem 2 *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are represented by CSDs, then $\mathcal{C}_1 \cap \mathcal{C}_2$ can be represented by a CSD.*
Proof: See *Appendix A*.
Assume $\mathcal{C}_1 = ((l_1^1, l_2^1, \ldots, l_n^1) : (u_1^1, u_2^1, \ldots, u_n^1) : S)$ and $\mathcal{C}_2 = ((l_1^2, l_2^2, \ldots, l_m^2) : (u_1^2, u_2^2, \ldots, u_m^2) : S')$. We prove that $\mathcal{C}_1 \cap \mathcal{C}_2 = (\ (L_{1,1}, \ldots, L_{1,m}, L_{2,1}, \ldots, L_{2,m}, \ldots, L_{n,m}) : (U_{1,1}, \ldots, U_{1,m}, U_{2,1}, \ldots, U_{2,m}, \ldots, U_{n,m}) : LCM(S, S')\ )$, where $a \neq b$ *or* $c \neq d \Rightarrow L_{a,b} \neq L_{c,d}$ and $|L_{a,b} - L_{c,d}| < LCM(S, S')$. Also in *Appendix B*, we show that that $\mathcal{C}_1 \cap \mathcal{C}_2$ can be calculated in $O(min(log\ S, log\ S') + m * n)$ time.

Finally, Theorem 3 shows that normalization properties of CSD.

Theorem 3 *If $\alpha * i + \beta \in (l : u : b : c)$, then i can be represented by a CSD.*
Proof: See *Appendix A*.

In the proof of Theorem 3, the normalization of $\alpha * i + \beta \in (l : u : b : c)$ can be done constructively. We can derive that $i \in ((L_1, L_2, \ldots, L_n) : (U_1, U_2, \ldots, U_n) : |(-c)/GCD(\alpha, -c)|)$, where $x \neq y \Rightarrow L_x \neq L_y$ and $|L_x - L_y| < |(-c)/GCD(\alpha, -c)|$ (as shown in Appendix A). Also in *Appendix B*, we show the normalization can be performed in $O(min(log\ \alpha, log\ c) + b)$ time.

3 Expression-Rewritings to Generate Communication Sets

In this section, we present our proposed five-step framework with mechanic arithmetic-rewriting rules to accompany the CSD calculus so that the communication set of HPF programs of "Block-Cyclic" distributions with two-level alignments (or multiple-level alignments), multi-dimensional arrays, array intrinsic functions (such as Transpose operation), and affine indexes in the array subscripts, can be calculated in a systematic way. We will first give the auxiliary functions for data distributions in Section 3.1 and multiple alignments in Section 3.2, respectively. The major frameworks will then be presented Section 3.3 following auxiliary definitions.

3.1 Model Data Distribution with Functions

In this subsection, we define **distribution function** for the data loayout of an array in HPF programs. The communication set can then be calculated with a series of expression rewritings based on distribution functions. First, let's define the descriptor needed for a distribution function.

Definition 4 *A* **descriptor**, *ϕ, is a set of integer vectors, and defined as follows:*

$$\phi(/f_1(i_1, i_2, \ldots, i_n), \ldots, f_m(i_1, i_2, \ldots, i_n)/, /S_1, S_2, \ldots, S_m/) =$$
$$\{(i_1, i_2, \ldots, i_n) \mid \forall\ k,\ 1 \leq k \leq n,\ f_k(i_1, i_2, \ldots, i_n) \in S_k\ \}$$

S_k $(1 \leq k \leq n)$ represents a set of integers. In this paper, we are interested in having S_k be a RSD, BSD, or CSD. Next , we will use an example below to explain a distribution function.

Example 2

```
        REAL A(48,48)
!HPF$   PROCESSOR PROCS(2,3)
!HPF$   DISTRIBUTE A(CYCLIC(6),CYCLIC(8)) ONTO PROCS
```

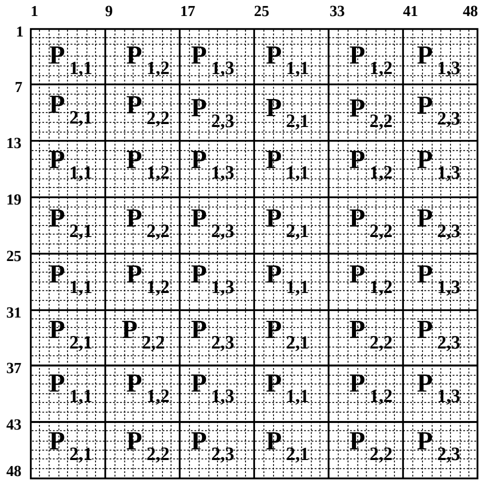

Figure 2: Data Layout of Array A

For the HPF programs above, array A is partitioned and shown in *Figure 2*. Array A is distributed among six processors. In the example, A(i,j) is located on processor $P_{1,1}$ if i $\in$ (1:48:6:12) and j $\in$ (1:48:8:24). Similarly, A(i,j) resides in processor $P_{2,1}$ if (i,j) $\in \phi$(/i,j/,/7:48:6:12,1:48:8:24/), A(i,j) resides in processor $P_{1,2}$ if (i,j) $\in$ ϕ(/i,j/,/1:48:6:12, 9:48:8:24/), A(i,j) resides in processor $P_{2,2}$ if (i,j) $\in \phi$(/i,j/,/7:48:6:12, 9:48:8:24/), A(i,j) resides in processor $P_{1,3}$ if (i,j) $\in \phi$(/i,j/,/1:48:6:12, 17:48:8:24/), and A(i,j) resides in processor $P_{2,3}$ if (i,j) $\in \phi$(/i,j/,/7:48:6:12, 17:48:8:24/).

With the above information, we construct the **distribution function** of array A as follows:

$$A_{dis}(i,j) = \begin{cases} P_{1,1} \mid \phi(\text{/i,j/,/1:48:6:12, 1:48:8:24/}) \\ P_{2,1} \mid \phi(\text{/i,j/,/7:48:6:12,1:48:8:24/}) \\ P_{1,2} \mid \phi(\text{/i,j/,/1:48:6:12, 9:48:8:24/}) \\ P_{2,2} \mid \phi(\text{/i,j/,/7:48:6:12, 9:48:8:24/}) \\ P_{1,3} \mid \phi(\text{/i,j/,/1:48:6:12, 17:48:8:24/}) \\ P_{2,3} \mid \phi(\text{/i,j/,/7:48:6:12, 17:48:8:24/}) \end{cases}$$

We list the general form of the distribution function of an n-dimensional array A for block-cyclic distribution as follows:

$$A_{dis}(i_1, i_2, \ldots, i_n) = \begin{cases} P_{v_1} \mid \phi(/i_1, i_2, \ldots, i_n/,\ /l_{1,1} : u_{1,1} : b_1 : c_1, \ldots, l_{n,1} : u_{1,n} : b_n : c_n/) \\ \ldots \\ P_{v_k} \mid \phi(/i_1, i_2, \ldots, i_n/,\ /l_{k,1} : u_{k,1} : b_1 : c_1, \ldots, l_{k,n} : u_{k,n} : b_n : c_n/) \\ \ldots \\ P_{v_m} \mid \phi(/i_1, i_2, \ldots, i_n/,\ /l_{m,1} : u_{m,1} : b_1 : c_1, \ldots, l_{m,n} : u_{m,n} : b_n : c_n/) \end{cases}$$

The equality relation, $A_{dis}(i_1, i_2, \ldots, i_n) = P_{v_k} \mid \phi(/i_1, i_2, \ldots, i_n/,\ /l_{k,1} : u_{k,1} : b_1 : c_1, \ldots, l_{k,n} : u_{k,n} : b_n : c_n/)$, means $A(i_1, i_2, \ldots, i_n)$ resides in processor P_{v_k} if $(i_1, i_2, \ldots, i_n) \in \phi(/i_1, i_2, \ldots, i_n/,\ /l_{k,1} : u_{k,1} : b_1 : c_1, \ldots, l_{k,n} : u_{k,n} : b_n : c_n/)$.

3.2 Solving Multiple-level Alignment Problems

Next, we will try to show how to map multi-level alignments into one-level alignments. The process is important for solving the general cases in calculating communications set of HPF programs with "Block-Cyclic" distributions and multi-level alignments. Consider the following code fragment,

Example 3

```
!HPF$   PROCESSOR PROCS(2,3)
!HPF$   ALIGN A(I,J) WITH T1(J-2,2*I)
!HPF$   ALIGN T1(I,J) WITH T2(2*I-5,3*J-6)
!HPF$   ALIGN T2(I,J) WITH T(I+9,J+20)
!HPF$   DISTRIBUTE T(CYCLIC(6),CYCLIC(8)) ONTO PROCS.
```

We need to calculate the multiple-level alignment relations to map the source array to the final target arrays. We suggest solving this problem by using array operation synthesis scheme[15, 16, 17], which we developed earlier in composing consecutive array operations together. It can be done as follows. First, we construct several functions according to the alignment relation defined in *Example 3.* They are

$$A(I,J) = T1(J-2, 2*I), T1(I,J) = T2(2*I-5, 3*J-6),\ and\, T2(I,J) = T(I+9, J+20).$$

Then, we do function compositions to synthesize the above three functions. We have $A(I,J) = T1(J-2, 2*I) = T2(2*(J-2)-5, 3*(2*i)-6) = T2(2*J-9, 6*I-6) = T((2*J-9)+9, (6*I-6)+20) = T(2*J, 6*I+14)$. Finally, we derive the ultimate alignment relation that $A(I,J)$ is aligned to $T(2*J, 6*I+14)$.

3.3 Generating Communication Set

The whole process to compute communication set can be done with a series of expression rewritings with five steps. It's described in details below, and we will use the code fragment right below as an example.

HPF Code Fragment 2

!HPF$ ALIGN A$(i_1, i_2, \ldots, i_n)$ WITH T1$(\mu_1(i_1,\ldots,i_n),\ldots,\mu_n(i_1,\ldots,i_n))$
!HPF$ ALIGN B$(i_1, i_2, \ldots, i_n)$ WITH T2$(\nu_1(i_1,\ldots,i_n),\ldots,\nu_n(i_1,\ldots,i_n))$
!HPF$ PROCESSOR PROCS$(d_1, d_2, \ldots, d_n)$
!HPF$ DISTRIBUTE T1(CYCLIC(b_1),...,CYCLIC(b_n)) ONTO PROCS
!HPF$ DISTRIBUTE T2(CYCLIC(b'_1),...,CYCLIC(b'_n)) ONTO PROCS
⋮
FORALL $(i_1 = L_1 : U_1 : S_1, i_2 = L_2 : U_2 : S_2, \ldots, i_n = L_n : U_n : S_n)$
A$(f_1(i_1,i_2,\ldots,i_n),\ldots,f_n(i_1,i_2,\ldots,i_n))$=$\mathcal{F}$(B$(g_1(i_1,i_2,\ldots,i_n),\ldots,g_n(i_1,i_2,\ldots,i_n))$)
END FORALL

In the code above, array A and B are aligned to template array T1 and T2, respectively. Next, T1 and T2 are distributed among processors by block-cyclic distribution. The functions, f_i and g_i are all affine functions to be used in the array subscript, and $\mathcal{F}$ is an array function or array intrinsic function (such as Transpose operation).

Step 1: To prepare the distribution function of template arrays T1 and T2

Since template array T1 and T2 are finally distributed among processors, we can derive their corresponding distribution functions. We assume the distribution functions of T1 and T2 are as follows:

$$T1_{dis}(i_1,i_2,\ldots,i_n) = \begin{cases} P_1 \mid & \phi(/i_1,i_2,\ldots,i_n/,\ /l_{1,1}:u_{1,1}:b_1:c_1,\ldots,l_{n,1}:u_{1,n}:b_n:c_n/) \\ P_2 \mid & \phi(/i_1,i_2,\ldots,i_n/,\ /l_{2,1}:u_{2,1}:b_1:c_1,\ldots,l_{2,n}:u_{2,n}:b_n:c_n/) \\ \ldots & \\ P_q \mid & \phi(/i_1,i_2,\ldots,i_n/,\ /l_{q,1}:u_{q,1}:b_1:c_1,\ldots,l_{q,n}:u_{q,n}:b_n:c_n/) \\ \ldots & \\ P_x \mid & \phi(/i_1,i_2,\ldots,i_n/,\ /l_{x,1}:u_{x,1}:b_1:c_1,\ldots,l_{x,n}:u_{x,n}:b_n:c_n/) \end{cases} \quad (1)$$

$$T2_{dis}(i_1,i_2,\ldots,i_n) = \begin{cases} P_1 \mid \ \phi(/i_1,i_2,\ldots,i_n/,\ /l'_{1,1} : u'_{1,1} : b'_1 : c'_1,\ \ldots, l'_{n,1} : u'_{1,n} : b'_n : c'_n/) \\ P_2 \mid \ \phi(/i_1,i_2,\ldots,i_n/,\ /l'_{2,1} : u'_{2,1} : b'_1 : c'_1,\ \ldots, l'_{2,n} : u'_{2,n} : b'_n : c'_n/) \\ \ldots \\ P_r \mid \ \phi(/i_1,i_2,\ldots,i_n/,\ /l'_{r,1} : u'_{r,1} : b'_1 : c'_1,\ \ldots, l'_{r,n} : u'_{r,n} : b'_n : c'_n/) \\ \ldots \\ P_y \mid \ \phi(/i_1,i_2,\ldots,i_n/,\ /l'_{y,1} : u'_{y,1} : b'_1 : c'_1,\ \ldots, l'_{y,n} : u'_{y,n} : b'_n : c'_n/) \end{cases} \quad (2)$$

Step 2: To solve the alignment relation between A, B and T

In our second step, we employ array operation synthesis scheme to derive the ultimate alignment relation between A, B and T, as mentioned in section 3.2. The alignment relation of arrays in *Code Fragment 2* can be directly derived from line 1 and line 2 as follows:

$$A(i_1,i_2,\ldots,i_n) \ \ is \ \ aligned \ \ to \ \ T1(\mu_1(i_1,\ldots,i_n),\ldots,\mu_n(i_1,\ldots,i_n)) \quad (3)$$

$$B(i_1,i_2,\ldots,i_n) \ \ is \ \ aligned \ \ to \ \ T2(\nu_1(i_1,\ldots,i_n),\ldots,\nu_n(i_1,\ldots,i_n)) \quad (4)$$

Note that $\mu_k(i_1,i_2,\ldots,i_n)$ and $\nu_k(i_1,i_2,\ldots,i_n)$,$1 \le k \le n$, are linear affine index functions of the form $C_0 * I + C_1$, where C_0 and C_1 are integer constants, I $\in \{i_1,i_2,\ldots,i_n\}$.

Step 3: To derive the distribution functions of array A and B

Combining (1) and (3), we have the distribution function of array A as follows:

$$A_{dis}(i_1,\ldots,i_n) = \begin{cases} P_1 \mid \ \phi(/\mu_1(i_1,\ldots,i_n),\ldots,\mu_n(i_1,\ldots,i_n)/,\ /l_{1,1} : u_{1,1} : b_1 : c_1,\ldots,l_{n,1} : u_{1,n} : b_n : c_n/) \\ P_2 \mid \ \phi(/\mu_1(i_1,\ldots,i_n),\ldots,\mu_n(i_1,\ldots,i_n)/,\ /l_{2,1} : u_{2,1} : b_1 : c_1,\ldots,l_{2,n} : u_{2,n} : b_n : c_n/) \\ \ldots \\ P_q \mid \ \phi(/\mu_1(i_1,\ldots,i_n),\ldots,\mu_n(i_1,\ldots,i_n)/,\ /l_{q,1} : u_{q,1} : b_1 : c_1,\ldots,l_{q,n} : u_{q,n} : b_n : c_n/) \\ \ldots \\ P_x \mid \ \phi(/\mu_1(i_1,\ldots,i_n),\ldots,\mu_n(i_1,\ldots,i_n)/,\ /l_{x,1} : u_{x,1} : b_1 : c_1,\ldots,l_{x,n} : u_{x,n} : b_n : c_n/) \end{cases} \quad (5)$$

Combining (2) and (4), we have the distribution function of array B as follows:

$$B_{dis}(i_1,i_2,\ldots,i_n) = \begin{cases} P_1 \mid \ \phi(/\nu_1(i_1,\ldots,i_n),\ldots,\nu_n(i_1,\ldots,i_n)/,\ /l'_{1,1} : u'_{1,1} : b'_1 : c'_1,\ \ldots, l'_{n,1} : u'_{1,n} : b'_n : c'_n/) \\ P_2 \mid \ \phi(/\nu_1(i_1,\ldots,i_n),\ldots,\nu_n(i_1,\ldots,i_n)/,\ /l'_{2,1} : u'_{2,1} : b'_1 : c'_1,\ \ldots, l'_{2,n} : u'_{2,n} : b'_n : c'_n/) \\ \ldots \\ P_r \mid \ \phi(/\nu_1(i_1,\ldots,i_n),\ldots,\nu_n(i_1,\ldots,i_n)/,\ /l'_{r,1} : u'_{r,1} : b'_1 : c'_1,\ \ldots, l'_{r,n} : u'_{r,n} : b'_n : c'_n/) \\ \ldots \\ P_y \mid \ \phi(/\nu_1(i_1,\ldots,i_n),\ldots,\nu_n(i_1,\ldots,i_n)/,\ /l'_{y,1} : u'_{y,1} : b'_1 : c'_1,\ \ldots, l'_{y,n} : u'_{y,n} : b'_n : c'_n/) \end{cases} \quad (6)$$

Step 4: To combine the definition and use of array A and B with the distribution functions

According to line 7 of *code fragment 2*,

A($f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)$) , where $(i_1,\ldots,i_n) \in \phi(/i_1,\ldots,i_n/,/L_1 : U_1 : S_1,\ \ldots,\ L_n : U_n : S_n/)$, are the array section of A which are defined in the loop. Similarly,

B($g_1(i_1,\ldots,i_n),\ \ldots,\ g_n(i_1,\ldots,i_n)$), where $(i_1,\ldots,i_n) \in \phi(/i_1,\ldots,i_n/,/L_1 : U_1 : S_1,\ \ldots,\ L_n : U_n : S_n/)$, are the array section of B which are used in the loop.

$$
\begin{aligned}
&(i_1, i_2, \ldots, i_n) \in \\
&\phi(/\mu_1(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)),\ldots,\mu_n(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n))/,\ /l_{q,1} : u_{q,1} : b_1 : c_1, \ldots, l_{q,n} : u_{q,n} : b_n : c_n/) \\
&\wedge\ \phi(/\nu_1(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n)),\ldots,\nu_n(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n))/,\ /l'_{r,1} : u'_{r,1} : b'_1 : c'_1,\ \ldots, l'_{r,n} : u'_{r,n} : b'_n : c'_n/) \\
&\wedge\ \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/).
\end{aligned}
$$

Figure 3: Communication set that P_r must send to P_q

The distribution function corresponding to the defined section of array A is as follows:

$$
A(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)) =
\begin{cases}
P_1 \mid \phi(/\mu_1(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)),\ldots,\mu_n(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n))/,\ /l_{1,1} : u_{1,1} : b_1 : c_1, \ldots, l_{n,1} : u_{1,n} : b_n : c_n/) \\
\qquad \wedge \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/ \\
P_2 \mid \phi(/\mu_1(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)),\ldots,\mu_n(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n))/,\ /l_{2,1} : u_{2,1} : b_1 : c_1, \ldots, l_{2,n} : u_{2,n} : b_n : c_n/) \\
\qquad \wedge\ \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/ \\
\ldots \\
P_q \mid \phi(/\mu_1(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)),\ldots,\mu_n(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n))/,\ /l_{q,1} : u_{q,1} : b_1 : c_1, \ldots, l_{q,n} : u_{q,n} : b_n : c_n/) \\
\qquad \wedge\ \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/ \\
\ldots \\
P_x \mid \phi/\mu_1(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)),\ldots,\mu_n(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n))\ /,\ /l_{x,1} : u_{x,1} : b_1 : c_1, \ldots, l_{x,n} : u_{x,n} : b_n : c_n/) \\
\qquad \wedge\ \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/
\end{cases}
\tag{7}
$$

The distribution function corresponding to the used section of array B is as follows:

$$
B(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n)) =
\begin{cases}
P_1 \mid \phi(/\nu_1(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n)),\ldots,\nu_n(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n))/,\ /l'_{1,1} : u'_{1,1} : b'_1 : c'_1, \ldots, l'_{n,1} : u'_{1,n} : b'_n : c'_n/) \\
\qquad \wedge \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/ \\
P_2 \mid \phi(/\nu_1(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n)),\ldots,\nu_n(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n))/,\ /l'_{2,1} : u'_{2,1} : b'_1 : c'_1, \ldots, l'_{2,n} : u'_{2,n} : b'_n : c'_n/) \\
\qquad \wedge\ \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/ \\
\ldots \\
P_r \mid \phi(/\nu_1(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n)),\ldots,\nu_n(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n))/,\ /l'_{r,1} : u'_{r,1} : b'_1 : c'_1, \ldots, l'_{r,n} : u'_{r,n} : b'_n : c'_n/) \\
\qquad \wedge\ \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/ \\
\ldots \\
P_y \mid \phi/\nu_1(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n)),\ldots,\nu_n(g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n))\ /,\ /l'_{y,1} : u'_{y,1} : b'_1 : c'_1, \ldots, l'_{y,n} : u'_{y,n} : b'_n : c'_n/) \\
\qquad \wedge\ \phi(/i_1, i_2, \ldots, i_n/, /L_1 : U_1 : S_1, \ldots, L_n : U_n : S_n/
\end{cases}
\tag{8}
$$

Step 5: To generate communication set according to the distribution functions (7) and (8)

HPF compiler uses the owner-computing rule to partition the computation among the processors. Each processor only computes values of data it owns [8, 19].

According to equation (7), A($f_1(i_1,\ldots,i_n)$, …, $f_n(i_1,\ldots,i_n)$) resides in processor P_q if

$(i_1,\ldots,i_n) \in \phi(/\mu_1(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)),\ldots,\mu_n(f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n))/,\ /l_{q,1} : u_{q,1} : b_{q,1} : c_{q,1},\ldots,l_{q,n} : u_{q,n} : b_{q,n} : c_{q,n}/) \wedge \phi(/i_1,\ldots,i_n/, /L_1 : U_1 : S_1,\ldots,L_n : U_n : S_n/$.

In addition, according to equation (8), B($g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n)$) resides in processor P_r if

$(i_1,\ldots,i_n) \in \phi(\ /\nu_1(g_1(i_1,\ldots,i_n),\ \ldots,\ g_n(i_1,\ldots,i_n)),\ \ldots,\ \nu_n(g_1(i_1,\ldots,i_n),\ \ldots,\ g_n(i_1,\ldots,i_n))/,\ /l'_{q,1} : u'_{q,1} : b'_{q,1} : c'_{q,1},\ \ldots,l'_{q,n} : u'_{q,n} : b'_{q,n} : c'_{q,n}/) \wedge \phi(/i_1,\ldots,i_n/, /L_1 : U_1 : S_1,\ldots,L_n : U_n : S_n/)$.

Finally, based on the owner-computes rule, process P_r must send B($g_1(i_1,\ldots,i_n),\ldots,g_n(i_1,\ldots,i_n)$) to process P_q, and store it in A($f_1(i_1,\ldots,i_n),\ldots,f_n(i_1,\ldots,i_n)$) if $(i_1,\ldots,i_n)$ satisfies the intersection equation in *Figure 3*. By solving the intersection of descriptors as shown in *Figure 3*, we can derive the communication set which processor P_r must send to processor P_q.

The intersection of segmentation descriptors is to do intersection of section descriptors in each dimension. In the follows, we explain how to apply the **CSD Calculus** to solve the equation in *Figure 3*. Note that $\mu_k(i_1,\cdots,i_n)$, $\nu_k(i_1,\cdots,i_n)$, $f_k(i_1,\cdots,i_n)$ and $g_k(i_1,\cdots,i_n)$, $1 \le k \le n$, are linear affine index functions with only one index variable. We can easily prove that $\mu_k(f_1(i_1,\cdots,i_n),\cdots,f_n(i_1,\cdots,i_n))$ and $\nu_k(g_1(i_1,\cdots,i_n),\ \cdots,g_n(i_1,\cdots,i_n))$, are also linear affine index functions with one index variable. See *Theorem 4*.

Theorem 4 *If $\mu_k(i_1,\cdots,i_n)$, $f_1(i_1,\cdots,i_n)$, $f_2(i_1,\cdots,i_n)$, $\cdots$, $f_n(i_1,\cdots,i_n)$ are linear affine index functions with one index variable, then $\mu_k(f_1(i_1,\cdots,i_n),\cdots,f_n(i_1,\cdots,i_n))$ is also linear affine index functions with one index variable.*

Proof: See *Appendix A*.

According to *Theorem 4*, $\mu_k(f_1(i_1,\cdots,i_n),\cdots,f_n(i_1,\cdots,i_n))$ and $\nu_k(g_1(i_1,\cdots,i_n),\ \cdots,g_n(i_1,\cdots,i_n))$, $\forall$ k, $1 \le k \le n$, are linear affine index functions with one index variable. Thus, we can continue to solve the equation in *Figure 3*. We can normalize $\mu_k(f_1(i_1,\cdots,i_n),\cdots,f_n(i_1,\cdots,i_n)) \in (l_{q,k} : u_{q,k} : b_k : c_k)$, $\forall$ k, $1 \le k \le n$, according to *Theorem 3*. Similarly, we can normalize $\nu_s(g_1(i_1,\cdots,i_n),\ \cdots,g_n(i_1,\cdots,i_n)) \in (l'_{r,s} : u'_{r,s} : b'_s : c'_s)$, $\forall$ s, $1 \le s \le n$. We get the results below for the equation in *Figure 3*.

$$\begin{aligned}(i_1,i_2,\cdots,i_n) \in \quad & \phi(/i_1,i_2,\cdots,i_n/,\ /\mathcal{C}_1,\mathcal{C}_2,\cdots,\mathcal{C}_n/)\ \wedge \\ & \phi(/i_1,i_2,\cdots,i_n/,\ /\mathcal{C}'_1,\mathcal{C}'_2,\cdots,\mathcal{C}'_n/)\ \wedge \\ & \phi(/i_1,i_2,\cdots,i_n/,/L_1:U_1:S_1,\cdots,L_n:U_n:S_n/)\,,\end{aligned}$$

where $\mathcal{C}_1,\mathcal{C}_2,...,\mathcal{C}_n,\mathcal{C}'_1,\mathcal{C}'_2,...,\mathcal{C}'_n$ are all CSDs.

We can conclude the results further as shown below.

$$(i_1,i_2,\cdots,i_n) \in \ \phi(/i_1,\cdots,i_n/,\ /\mathcal{C}_1\cap\mathcal{C}'_1\cap(L_1:U_1:S_1),\cdots,\mathcal{C}_n\cap\mathcal{C}'_n\cap(L_n:U_n:S_n)/)$$

Because the intersection of CSDs is a CSD (*Theorem 2*), and the intersection of a CSD and RSD (as a RSD is also a CSD according to Theorem 1) is again a CSD, the final result is a CSD. Thus, the communication set between data movement of multiple-level alignment and multi-dimensional arrays can be calculated with the five step scheme based on CSD calculus and expression rewritings.

3.4 An Example in Calculating Communication Sets

We use the following code as an example to demonstrate how the five step scheme can be used to generate communication sets for HPF programs with block-cyclic distributions.

Example 4

```
!HPF$    TEMPLATE T1(96,144)
!HPF$    TEMPLATE T2(240,288)
         REAL A(48,48),B(48,48)
!HPF$    ALIGN A(I,J) WITH T1(2*I,3*J)
!HPF$    ALIGN B(I,J) WITH T2(5*I,6*J)
!HPF$    PROCESSOR PROCS(2,3)
!HPF$    DISTRIBUTE T1(CYCLIC(12),CYCLIC(24)) ONTO PROCS
!HPF$    DISTRIBUTE T2(CYCLIC(30),CYCLIC(48)) ONTO PROCS
           .
           .
           .
         A(6:26:2,3:39:3)=SIN( TRANSPOSE(B(4:40:3,5:45:4)) )
```

Step 1: To prepare the distribution functions for template array T1 and T2 as follows.

$$T1_{dis}(i_1, i_2) = \begin{cases} P_{1,1} \mid \phi(/i_1, i_2/, /1:96:12:24, 1:144:24:72/) \\ P_{2,1} \mid \phi(/i_1, i_2/, /13:96:12:24, 1:144:24:72/) \\ P_{1,2} \mid \phi(/i_1, i_2/, /1:96:12:24, 25:144:24:72/) \\ P_{2,2} \mid \phi(/i_1, i_2/, /13:96:12:24, 25:144:24:72/) \\ P_{1,3} \mid \phi(/i_1, i_2/, /1:96:12:24, 49:144:24:72/) \\ P_{2,3} \mid \phi(/i_1, i_2/, /13:96:12:24, 49:144:24:72/) \end{cases}$$

$$T2_{dis}(i_1, i_2) = \begin{cases} P_{1,1} \mid \phi(/i_1, i_2/, /1:240:30:60, 1:288:48:144/) \\ P_{2,1} \mid \phi(/i_1, i_2/, /31:240:30:60, 1:288:48:144/) \\ P_{1,2} \mid \phi(/i_1, i_2/, /1:240:30:60, 49:288:48:144/) \\ P_{2,2} \mid \phi(/i_1, i_2/, /31:240:30:60, 49:288:48:144/) \\ P_{1,3} \mid \phi(/i_1, i_2/, /1:240:30:60, 97:288:48:144/) \\ P_{2,3} \mid \phi(/i_1, i_2/, /31:240:30:60, 97:288:48:144/) \end{cases}$$

Step 2: Solving the alignment relation between A and T1, and B and T2, respectively. We have $A(i_1, i_2)$ aligned to $T1(2*i_1, 3*i_2)$, and $B(i_1, i_2)$ aligned to $T2(5*i_1, 6*i_2)$.

Step 3: To derive the distribution functions of array A and B, respectively.

See follows:

$$A_{dis}(i_1, i_2) = \begin{cases} P_{1,1} \mid \phi(/2*i_1, 3*i_2/, /1:96:12:24, 1:144:24:72/) \\ P_{2,1} \mid \phi(/2*i_1, 3*i_2/, /13:96:12:24, 1:144:24:72/) \\ P_{1,2} \mid \phi(/2*i_1, 3*i_2/, /1:96:12:24, 25:144:24:72/) \\ P_{2,2} \mid \phi(/2*i_1, 3*i_2/, /13:96:12:24, 25:144:24:72/) \\ P_{1,3} \mid \phi(/2*i_1, 3*i_2/, /1:96:12:24, 49:144:24:72/) \\ P_{2,3} \mid \phi(/2*i_1, 3*i_2/, /13:96:12:24, 49:144:24:72/) \end{cases}$$

$$B_{dis}(i_1, i_2) = \begin{cases} P_{1,1} \mid \phi(/5*i_1, 6*i_2/, /1:240:30:60, 1:288:48:144/) \\ P_{2,1} \mid \phi(/5*i_1, 6*i_2/, /31:240:30:60, 1:288:48:144/) \\ P_{1,2} \mid \phi(/5*i_1, 6*i_2/, /1:240:30:60, 49:288:48:144/) \\ P_{2,2} \mid \phi(/5*i_1, 6*i_2/, /31:240:30:60, 49:288:48:144/) \\ P_{1,3} \mid \phi(/5*i_1, 6*i_2/, /1:240:30:60, 97:288:48:144/) \\ P_{2,3} \mid \phi(/5*i_1, 6*i_2/, /31:240:30:60, 97:288:48:144/) \end{cases}$$

Step 4:

We translate $A(6:26:2, 3:39:3) = SIN(TRANSPOSE(B(4:40:3, 5:45:4)))$ into the following code:

```
FORALL (i1 = 1 : 11 : 1, i2 = 1 : 13 : 1)
       A(2 * i1 + 4, 3 * i2) = SIN(B(3 * i2 + 1, 4 * i1 + 1))
END FORALL
```

The defined section of array A is $A(2*i_1+4, 3*i_2) \mid \phi(/i_1, i_2/, /1:11:1, 1:13:1/)$. The used section of array B is $B(3*i_2+1, 4*i_1+1) \mid \phi(/i_1, i_2/, /1:11:1, 1:13:1/)$.

The corresponding distribution function of the defined section of array A is as follows:

$$A_{dis}(2*i_1+4, 3*i_2) = \begin{cases} P_{1,1} \mid \phi(/2*(2*i_1+4), 3*(3*i_2)/, /1:96:12:24, 1:144:24:72/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{2,1} \mid \phi(/2*(2*i_1+4), 3*(3*i_2)/, /13:96:12:24, 1:144:24:72/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{1,2} \mid \phi(/2*(2*i_1+4), 3*(3*i_2)/, /1:96:12:24, 25:144:24:72/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{2,2} \mid \phi(/2*(2*i_1+4), 3*(3*i_2)/, /13:96:12:24, 25:144:24:72/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{1,3} \mid \phi(/2*(2*i_1+4), 3*(3*i_2)/, /1:96:12:24, 49:144:24:72/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{2,3} \mid \phi(/2*(2*i_1+4), 3*(3*i_2)/, /13:96:12:24, 49:144:24:72/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \end{cases}$$

The corresponding distribution function of the used section of array B is as follows:

$$B_{dis}(3*i_2+1, 4*i_1+1) = \begin{cases} P_{1,1} \mid \phi(/5*(3*i_2+1), 6*(4*i_1+1)/, /1:240:30:60, 1:288:48:144/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{2,1} \mid \phi(/5*(3*i_2+1), 6*(4*i_1+1)/, /31:240:30:60, 1:288:48:144/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{1,2} \mid \phi(/5*(3*i_2+1), 6*(4*i_1+1)/, /1:240:30:60, 49:288:48:144/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{2,2} \mid \phi(/5*(3*i_2+1), 6*(4*i_1+1)/, /31:240:30:60, 49:288:48:144/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{1,3} \mid \phi(/5*(3*i_2+1), 6*(4*i_1+1)/, /1:240:30:60, 97:288:48:144/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \\ P_{2,3} \mid \phi(/5*(3*i_2+1), 6*(4*i_1+1)/, /31:240:30:60, 97:288:48:144/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/) \end{cases}$$

Step 5:

Here, we will only demonstrate how to compute the communication set which processor $P_{1,2}$ should send to processor $P_{2,1}$.

(1) According to the result of *Step 4*, Processor $P_{1,2}$ should send $B(3*i_2+1, 4*i_1+1)$ to processor $P_{2,1}$ and store it in $A(2*i_1+4, 3*i_2)$, where $(i_1, i_2) \in \phi(/5*(3*i_2+1), 6*(4*i_1+1)/, /1:240:30:60, 49:288:48:144/) \wedge \phi(/2*(2*i_1+4), 3*(3*i_2)/, /13:96:12:24, 1:144:24:72/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/)$.

(2) Normalize all the BSD in the descriptors. We obtain CSD form as follows.

$(i_1, i_2) \in \phi(/i_1, i_2/, /((2,3,4):(20,21,22):6), ((1,2):(9:10):8)/) \wedge \phi(/i_1, i_2/, /((2,3):(8,9):6)/, ((0,1):(12,13):4)/) \wedge \phi(/i_1, i_2/, /1:11:1, 1:13:1/)$.

$\Rightarrow (i_1, i_2) \in \phi(/i_1, i_2/, /((2,3,4):(20,21,22):6) \cap ((2,3):(8,9):6) \cap (1:11:1), ((1,2):(9:10):8) \cap ((0,1):(12,13):4) \cap (1:13:1)/)$

$\Rightarrow (i_1, i_2) \in \phi(/i_1, i_2/, /((2,3):(8,9):6), (1:9:8)/)$

(3) Therefore, we derive that Processor $P_{1,2}$ should send $B(3*i_2+1, 4*i_1+1)$ to processor $P_{2,1}$ and store it in $A(2*i_1+4, 3*i_2)$, where $(i_1, i_2) \in \phi(/i_1, i_2/, /((2,3):(8,9):6), (1:9:8)/)$.

Similarly, the communication sets which processor $P_{1,1}$, $P_{2,1}$, $P_{1,2}$, $P_{2,2}$, $P_{1,3}$, and $P_{2,3}$ should send to processor $P_{2,1}$ can be calculated. We list the results in the following table.

	Send $B(3*i_2+1, 4*i_1+1)$ to $A(2*i_1+4, 3*i_2)$
$P_{1,1}$ to $P_{2,1}$	$\emptyset$
$P_{2,1}$ to $P_{2,1}$	$\emptyset$
$P_{1,2}$ to $P_{2,1}$	$(i_1, i_2) \in \phi(/i_1, i_2/, /((2,3):(8,9):6), (1:9:8)/)$
$P_{2,2}$ to $P_{2,1}$	$(i_1, i_2) \in \phi(/i_1, i_2/, /((2,3):(8,9):6), (2:10:8)/)$
$P_{1,3}$ to $P_{2,1}$	$(i_1, i_2) \in \phi(/i_1, i_2/, /(4:10:6), (1:9:8)/)$
$P_{2,3}$ to $P_{2,1}$	$(i_1, i_2) \in \phi(/i_1, i_2/, /(4:10:6), (2:10:8)/)$

4 Experimental Results

We have implemented our expression rewriting framework and CSD calculus to generate the communication codes for HPF programs. Our software framework includes (1) the definition of data structure of RSD, BSD and CSD, (2) intersection and normalization of RSD, BSD and CSD, and (3) the proposed expression rewriting framework. The software is currently being incorporated into an experimental HPF II compiler which is under development at Tsing-Hua university as a research prototype to explore compiler technologies[15, 21, 16, 23] In the following, we report performance results for the execution time of our framework in producing local enumeration set and communication set.

In the first part of our experiments, we compare our framework with the existing algorithm. Kennedy et al. [11] presented an algorithm based on integer lattice to find the local memory access sequence in computation involving

block size	stride=7		stride=99		stride=block size+1	
	Lattice's	Our Framework	Lattice's	Our Framework	Lattice's	Our Framework
8	14.2	7.5	17.5	10.8	16.7	6.7
16	13.3	6.7	20.0	10.8	21.7	7.5
32	17.5	9.2	28.3	11.7	24.2	8.5
64	23.3	11.7	34.2	14.2	38.3	11.7
128	31.7	15.0	70.0	20.8	50.0	15.8
256	54.2	30.0	92.5	32.5	69.2	25.0
512	98.3	50.0	130.8	55	112.5	45.0

Table 2: Execution times in microseconds for our algorithm compared with the integer lattice algorithm for local enumeration problem for a parallel loop.

regular sections of arrays with block-cyclic distributions. It's also known as the local enumeration problem. An example of the local enumeration problem is shown below.

HPF Code Fragment 3

```
!HPF$   ALIGN A(i) WITH T1(μ(i))
!HPF$   PROCESSOR PROCS(p)
!HPF$   DISTRIBUTE T1(CYCLIC(k)) ONTO PROCS
        ⋮
        FORALL (i = l : u : s)
            A(f(i))= i+3;
        END FORALL
```

The enumeration strategy in [11] to describe the local access sequence is to compute the first accessed address and derive a table of memory gaps(ΔM) for each processor based on integer lattice. Then, each processor uses the starting address and ΔM to enumerate local access sequence. The local enumeration problem in our framework is to solve the intersection of a CSD and a RSD in **CSD Calculus**. (As a RSD is also a CSD, so an intersection of a CSD and a RSD can be done as the intersection of CSDs.) Table 2 shows the performance comparison between integer lattice scheme[11] and our CSD scheme with the local enumeration problem for the parallel loop of the *Code Fragment 3*. The parameter is set to produce the local enumeration set for processor 3, and the rest of the parameters are set as follows, $p = 4$, $k = 8$, $s = 9$, $l = 0$, and 32 processors. The source code for calculating ΔM is directly from the work in [11]. The code for computing CSD $\cap$ RSD is from our **CSD Calculus**. All measurements are done on a digital DEC 3000 workstation using the *cc* compiler with **-O2** optimization level. Note that the theory complexity of our code is the same as the integer lattice in the local enumeration problem.

In the second part of our experiments, we measure the time for calculating the communication sets. We measure the performance of the following codes which is with multiple-level alignments, three dimensional arrays, and axis exchanges. The time needed to calculate the communication time is listed in Table 3. We list the time spent with calculation of the index in each dimension. We do not list the performance of other work, as none of existing work have reported an example with such general (complicated) cases. As the total time needed for the calculation for this problem is relative small, we feel our framework has the advantages of being a general and practical software framework, and can play an important role to lead to commercial strength implementation in the practice. Finally,

block size	Dimension One		Dimension Two		Dimension Three		Total	
	Max. time	Avg. time	Max. time	Avg. time	Max. time	Avg. time	Max. time	Avg. time
8	50	36	58	43	75	51	183	130
16	75	56	67	57	125	106	267	219
32	158	132	108	98	317	303	583	533
64	408	380	250	235	1033	1022	1691	1637
128	1401	1289	799	735	3999	3835	6199	5859

Table 3: Execution times in microseconds for our algorithm to generate communication sets

our software based on CSD calculus and arithmetic rewritings is available in `http:// falcon.cs.nthu.edu.tw/ ~ghhwang/papers.html.` for references.

HPF Code Fragment 4

```
!HPF$   ALIGN A(i,j,k) WITH T1(k-2,j-2,i-2)
!HPF$   ALIGN B(i,j,k) WITH T1(j+2,i+2,k+2)
!HPF$   PROCESSOR PROCS(d)
!HPF$   DISTRIBUTE T1(CYCLIC(b1,b2,b3)) ONTO PROCS
        ⋮
        FORALL (i=1:10000:3 :,j=1:10000:4,k=1:10000:5)
            A(7*i+5,8*j+6,13*k+10)=F(B(4*i+13,15*k+6,17*j+20) )
        END FORALL
```

5 Related Work

The issue of generating SPMD codes for HPF program with data distribution pattern was first addressed by Koelbel in [19]. It shows ways to calculate the communication set for distributed arrays with block or cyclic distribution. However, arrays with block-cyclic distribution was not considered. Later, the problem to deal with array statements distributed with block-cyclic distribution patterns was addressed by Chatterjee et al. [9]. It described a method for the enumeration of local indices in increasing order based on a finite-state machine. In [11, 29], linear algorithms for the general case are given based on by an integer lattice method[11, 29]. Their work mainly focused on the local set enumeration and did not discuss generating communication sets in any detail. Samuel et al. [27] solved the same problem. They represented the local iteration set as an affine function in two variables. The technique extended the ability of previous techniques by allowing intersections of local iteration sets. Note that we represent the local iteration set by CSD. The intersections of CSD is again a CSD. Samuel's work also did not address the generation of communication sets. In [30], it presents a comprehensive study of the run-time performance of the communication codes which are generated in [9, 11, 29]. In contrast to their work, our work here has mainly focused on another important aspect of the code generations in describing a framework to generate the communication set at static time.

The approach proposed by Gupta et al.[14] views a block-cyclic distribution as a block (or cyclic) distribution on a set of virtual processor, which are cyclically (or block-wise) mapped to physical processors. However, they did not present the cases as general as we have to deal with multi-dimensional arrays, multiple-level alignments (possibly axis alignments), array intrinsic functions (such as Transpose operation), and axis exchanges in the array subscript. Reeuwijk [26] et al. presents methods for rowwise and columnwise scanning of arrays in forall loops. The two orders of traversal are derived from different decomposition of the position equation, which specifies the relation between array indices and alignment and distribution parameters. They also handle general linear and affine subscripts in their compiler. In [25], Stinchnoth et al. described that the communication sets for block-cyclic distributions can be represented by union of regular section descriptors. Their framework focused one one-level alignment arrays. In this paper, we consider the most general cases that multi-dimensional arrays are distributed by two-level mapping in each dimension.

In summary, our framework is a practical software framework, and can handle the general cases so that the communication set of HPF programs of "Block-Cyclic" distributions with two-level alignments (multiple-level alignments or axis alignments), multi-dimensional arrays, array intrinsic functions (such as Transpose operation), and affine indexes and axis exchange in the array subscript, can be calculated in a systematic way with a sound software foundation. Previously, existing work do not report a software framework to solve a problem with such general cases. In addition, our expression-rewriting framework is based on a new representative form, CSD (common-stride descriptor), to describe the regularity of the access patterns of HPF programs with "Block-Cyclic" distribution. We also demonstrate a calculus of CSD that CSD is closed under intersection and normalization, which helps the process of calculating local enumeration and communication sets with efficiency and easiness. Our algorithm based on CSD to calculate the local enumeration set (as described in the experimental section) is the same as the intersection of CSD and RSD (see the complexity analysis 1 in Appendix B), which is as efficient as any existing algorithm in theoretical complexity[11]. The fastest algorithm in theoretical complexity in calculating the communication set for HPF programs with block-cyclic distributions belongs to the work in [12]. Although, our work is not as fast as the work in [12] in theoretic complexity, we demonstrate a more general software framework which can handle "Block-Cyclic" distributions with two-level alignments (multiple-level alignments or axis alignments), multi-dimensional arrays, array intrinsic functions (such as Transpose operation), and affine indexes and axis exchange in the array subscript. In addition, the experiments done in Section 4 also showed that our scheme is with good efficiency in generating communication set for block-cyclic distributions with two-level alignments, axis alignments, and axis exchange in the array subscript.

6 Conclusion

The generation of communication sets for HPF programs with block, cyclic, or collapse distribution has been implemented with success in commercial HPF compilers, however no commercial HPF compilers have so far provided efficient support for block-cyclic distribution[11]. We think the lack of support in commercial compilers for block-cyclic distributions is in part due to the lack of a software framework which can lead to commercial strength implementation in the practice to support the communication set of the block-cyclic distribution with the general cases. Therefore, we presented a new framework based on expression rewritings and a calculus form called CSD calculus to generate the local enumeration set and communication set for HPF programs with block-cyclic distribution. Our framework is a practical framework, and can handle the general cases so that the communication set of HPF programs of "Block-Cyclic" distributions with multiple-level alignments, multi-dimensional arrays, array intrinsic functions (such as Transpose operation), and affine indexes and axis exchange in the array subscript, can be calculated in a systematic way with a sound software foundation. Previously, existing work did not report a software framework to solve a problem with such general cases. Experimental results showed that our scheme not only can be easily implemented in the practice to support the general cases, but also was with good efficiency.

Acknowledgements

The software with integer lattice used as a base code for comparison in the experiments comes from Rice University[11]. We gratefully acknowledge their efforts.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. Fortran 90 Handbook Complete Ansi/Iso Reference. *Intertext Publications McGraw-Hill Book Company*, 1992.

[2] Bodin, F., Beckman, P., Gannon, D., Narayana, S., and Yang, S. Distributed pC++: Basic Ideas for an Object Parallel Language, *Scientific Programming*, 2(3), Fall 1993.

[3] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., Ranka, S., and Wu, M. Y. Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers, *J. Parallel Distrib. Comput.*, 21, 1 (April 1994), 15-26.

[4] Corman, T. L. , Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*, The MIT Press.

[5] Tyng-Ruey Chuang, Rong-Guey Chang, and Jenq Kuen Lee. Sampling and Analytical Techniques for Data Distribution of Parallel Sparse Computation, in SIAM Conference on Parallel Processing for Scientific Computing, March 1997.

[6] J. Dongarra, R. van de Geijn, and D. Walker. A Look at Scalable Dense Linear Algebra Libraries. In Proceedings of Scalable High-Performance Computing Conference, pages 372-379, Williamsburg, VA, April 1992.

[7] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-memory Machines. *Communications of the ACM*, (9):66–80, August 1992.

[8] C. Koelbel, D. Loveman, G.Stelle Schreiber, and M.Zosel. *The High Performance Fortran Handbook*. MIT-press, Cambridge, 1994.

[9] S.Chatterjee, J.Gilbert, F.Long, R. Scheriber, and S.Teng. Generating Local Addresses and Communication Sets for Data Parallel Programs. *Proceedings of Fourth ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pages 149–158, May 1993.

[10] Seema Hiranandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Compilation Techniques for Block-cyclic Distributions. *Proceeding of International Conference on Supercomputing*, pages 392–403, 1994.

[11] Ken Kennedy, Nenad Nedeljkovic, and Ajay Sethi. A Linear-time Algorithm for Computing the Memory Access Sequence in Data-parallel Programs. *Proceedings of Fifth ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pages 102–111, July 1995.

[12] Ken Kennedy, Nenad Nedeljkovic, and Ajay Sethi. *Communication Generation for Cyclic(k) Distributions*, the 3rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR '95), Troy, NY, May 1995.

[13] Henk J. Sips, Kees van Reeuwijk, and Will Denissen. Analysis of Local Enumeration and Storage Schemes in HPF. *Proceeding of International Conference on Supercomputing*, pages 10–17, 1996.

[14] S.K.S. Gupta, S.D. Kaushik, S Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On Compiling Array Expressions for Efficient Execution on Distributed-memory Machines. *In Proceedings of ICPP*, pages 301–305, 1993.

[15] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz ching Ju. An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs. *Proceedings of Fifth ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pages 112–122, July 1995.

[16] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. *Array Operation Synthesis to Optimize HPF Programs*, Proceedings of International Conference on Parallel Processing, Aug. 1996.

[17] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-Ching Ju. *Integrating Automatic Data Alignment and Array Operation Synthesis to Optimize Data Parallel Programs*, Workshop on Languages and Compilers for Parallel Computing, 3 pages, Minneapolis, AUg. 1997.

[18] S.K.S. Gupta, S.D.Kaushik, C.-H.Huang, and P.Sadayappan. Compiling Array Expressions for Efficient Execution on Distributed-memory Machines. *Proceedings of International Conference on Parallel Processing*, 1993.

[19] C. Koelbel. Compile-time generation of regular communications patterns. *Proceedings of Supercomputing'91*, pages 101–110, 1991.

[20] Jenq Kuen Lee and D. Gannon. *Object-Oriented Parallel Programming: Experiments and Results*, *Proceedings of Supercomputing '91*, New Mexico, November, 1991.

[21] Jenq Kuen Lee, Ing-Kuen Tsaur, and San-Yih Hwang. *Parallel Array Object I/O Support on Distributed Environments*, Journal of Parallel and Distributed Computing, 40, 227-241, 1997.

[22] Jenq Kuen Lee, Dan Ho, and Yue-Chee Chuang. *Data Distribution Analysis and Optimization for Pointer-Based Distributed Programs*, International Conference on Parallel Processing, August 1997.

[23] Jenq Kuen Lee and C. T. King. *Design HPF Compiler on Workstation Farms with High Performance Switch*, A Research Proposal to National Science Counsel of Taiwan, 1995-1997.

[24] J. Li and M. Chen. *The data alignment phase in compiling programs for distributed-memory machines*, Journal of parallel and Distributed Computing, Vol. 13, pp. 213-221, Oct. 1991.

[25] J. Stichnoth, D.O'Hallaron, and T. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, pages 150–159, 1994.

[26] Kees van Reeuwijk, Will Denissen, Henk J. Sips and Edwin M.R.M. Paalvast. *An implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems*, IEEE Tran. Parallel and Distributed Systems, Vol. 7, No. 9, Sep. 1996.

[27] Simuel P. Midkiff, *Local Iteration Set Computation For Block-cyclic Distributions*, 1995 International Conference on Parallel Processing.

[28] Kuei-Ping Shih, C. H. Huang, J. Sheu. Table-Lookup Approach for Compiling Two-Level Data-Processor Mappings in HPF. *Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, Aug. 1997.

[29] A. Thirumalai and J. Ramanujam, *Fast Address Sequence Generation for Data-Parallel Programs Using Integer Lattices*, Proc. Eighth Int'l Workshop Languages and Compilers for Parallel Computing, 1995.

[30] L. Wang, J. M. Stichnoth, and S. Chatterjee. *Runtime Performance of Parallel Array Assignment: An Empirical Study*, Proceedings of the 1996 ACM/IEEE Supercomputing Conference, Pittsburgh, PA, November 1996.

[31] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley Publishing Company, 1990.

A Theorem Proof

Theorem 1 *A set represented by a RSD can be represented by a BSD. A set represented by a BSD can be represented by a CSD.*

Proof:

We will show that RSD is a special case of BSD and BSD is a special case of CSD.

Since (l:u:s) equals (l:u:1:s), we have RSD $\subset$ BSD.

$(l:u:b:c) = \bigcup_{k=0}^{b-1}(l+k:u:c) = ((l, l+1, \ldots, l+b-1):(u, u, \ldots, u):c)$.

Also, $c > b > |(l+b-1)-l|$, we obtain BSD $\subset$ CSD.

Thus, we conclude that RSD $\subset$ BSD $\subset$ CSD.

Theorem 2 *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are represented by CSDs, then $\mathcal{C}_1 \cap \mathcal{C}_2$ can be represented by a CSD.*

Proof:

(1) Assume $\mathcal{C}_1 = ((l_1^1, l_2^1, \ldots, l_n^1):(u_1^1, u_2^1, \ldots, u_n^1):S)$ and $\mathcal{C}_2 = ((l_1^2, l_2^2, \ldots, l_m^2):(u_1^2, u_2^2, \ldots, u_m^2):S')$.

$\mathcal{C}_1 \cap \mathcal{C}_2$

$= ((l_1^1, l_2^1, \ldots, l_n^1):(u_1^1, u_2^1, \ldots, u_n^1):S) \cap ((l_1^2, l_2^2, \ldots, l_m^2):(u_1^2, u_2^2, \ldots, u_m^2):S')$

$= (\bigcup_{i=1}^{n}(l_i^1:u_i^1:S)) \cap (\bigcup_{j=1}^{m}(l_j^2:u_j^2:S'))$

$= \bigcup_{\substack{i=1,n \\ j=1,m}} ((l_i^1:u_i^1:S) \cap (l_j^2:u_j^2:S'))$

Assume $(l_i^1:u_i^1:S) \cap (l_j^2:u_j^2:S') = (L_{i,j}:U_{i,j}:LCM(S,S'))$.

Then, $\mathcal{C}_1 \cap \mathcal{C}_2 = \bigcup_{\substack{i=1,n \\ j=1,m}} ((l_i^1:u_i^1:S) \cap (l_j^2:u_j^2:S'))$

$= (\ (L_{1,1}, \ldots, L_{1,m}, L_{2,1}, \ldots, L_{2,m}, \ldots, L_{n,m}):(U_{1,1}, \ldots, U_{1,m}, U_{2,1}, \ldots, U_{2,m}, \ldots, U_{n,m}):LCM(S,S')\ )$

(2) As shown in Definition 3, we need to show two more properties to see if a descriptor is indeed a CSD. Here, we try to show $L_{a,b} \neq L_{c,d}$ if $(a,b) \neq (c,d)$[1]. We first assume that $L_{a,b}$ and $L_{c,d}$ may be equal. Then, we that it leads to contradictions.

- Assume $\exists\ L_{a,b}{=}L_{c,d}$ where $(a,b) \neq (c,d)$.
- $L_{a,b} \in ((l_a^1:u_a^1:S) \cap (l_b^2:u_b^2:S')) \Rightarrow L_{a,b} = l_a^1 + S*i$ (i is some integer).
- $L_{c,d} \in ((l_c^1:u_c^1:S) \cap (l_d^2:u_d^2:S')) \Rightarrow L_{c,d} = l_c^1 + S*j$ (j is some integer).

$\Rightarrow l_a^1 + S*i = l_c^1 + S*j \Rightarrow l_a^1 - l_c^1 = S*(j-i)$. If $j-i=0$ then $l_a^1 - l_c^1$=0. Also, If $j-i \neq 0$ then $|l_a^1 - l_c^1|$ is multiple of S.

- However, according to the definition of CSD, we have $l_a^1 \neq l_c^1$ and $l_a^1 - l_c^1 < S$. It contradicts our original assumption that $L_{a,b}{=}L_{c,d}$.

(3) Finally, we will try to show that that $|L_{a,b} - L_{c,d}| < LCM(S,S')$. We prove this by assuming that $L_{a,b} - L_{c,d} \geq LCM(S,S')$ and then it will lead to contradictions.

- Because $(L_{a,b}:U_{a,b}:LCM(S,S')) = ((l_a^1:u_a^1:S) \cap (l_b^2:u_b^2:S'))$ and $(L_{c,d}:U_{c,d}:LCM(S,S')) = ((l_c^1:u_c^1:S) \cap (l_d^2:u_d^2:S'))$, we have $L_{a,b} \geq MAX(l_a^1, l_b^2)$ and $L_{c,d} \geq MAX(l_c^1, l_d^2)$.
- Second, by the definition of CSD, we have $|l_c^1 - l_a^1| < S \Rightarrow -S < l_c^1 - l_a^1 < S \Rightarrow l_a^1 - S < l_c^1 < l_a^1 + S$ $\Rightarrow l_a^1 - S + 1 \leq l_c^1 \leq l_a^1 + S - 1$.
- Assume $\exists\ L_{a,b} - L_{c,d} \geq LCM(S,S')$. $\Rightarrow L_{a,b} - LCM(S,S') \geq L_{c,d}$.
- Let $\omega{=}L_{a,b} - LCM(S,S')$. Because the $L_{a,b}$ is the lower bound of $(L_{a,b}:U_{a,b}:LCM(S,S'))$, it is trivial that $\omega \notin (L_{a,b}:U_{a,b}:LCM(S,S'))$.

[1] $(a,b) \neq (c,d)$ means $a \neq c$ or $b \neq d$.

- Since ω=$L_{a,b} - LCM(S,S')$, we have $\underline{L_{a,b} > \omega}$. Also, by $\omega = L_{a,b} - LCM(S,S') \geq L_{c,d}$, we obtain $\underline{\omega \geq L_{c,d} \geq MAX(l_c^1, l_d^2) \geq l_c^1}$.
- Because $L_{a,b} \in (l_a^1 : u_a^1 : S)$, we have $L_{a,b} = l_a^1 + S * I$ where I is some non-negative integer. Then, ω $= L_{a,b} - LCM(S,S') = l_a^1 + S * I - LCM(S,S')$. Since both $S * I$ and $LCM(S,S')$ are multiples of S, $S * I - LCM(S,S')$ is also a multiple of S. Let $\omega = l_a^1 + S * I'$, where I' is some integer.
- Combining $l_a^1 - S + 1 \leq l_c^1 \leq l_a^1 + S - 1$ and $\omega \geq l_c^1$, we obtain $l_a^1 - S + 1 \leq \omega$. Then, as $l_a^1 - S + 1 \leq \omega$ and $\omega = l_a^1 + S * I'$ we have $\omega \geq l_a^1$.
- As $\omega \geq l_a^1$, $L_{a,b} \in (l_a^1 : u_a^1 : S)$ and ω =$L_{a,b} - LCM(S,S')$, we have $\omega \in (l_a^1 : u_a^1 : S)$. For the same reason, we can prove $\omega \in (l_b^2 : u_b^2 : S')$. By $\omega \in (l_a^1 : u_a^1 : S)$ and $\omega \in (l_b^2 : u_b^2 : S')$, we have $\omega \in (L_{a,b} : U_{a,b} : LCM(S,S'))$. It contradicts our original assumption that $\omega \notin (L_{a,b} : U_{a,b} : LCM(S,S'))$. Thus, $L_{a,b} - L_{c,d} \ngeq LCM(S,S')$.

Q.E.D.

Theorem 3 *If $\alpha * i + \beta \in (l : u : b : c)$, then i can be represented by a CSD.*

Proof:

Without loss of generality, we assume $\alpha \geq 0$. We have the followings.

- Since $\alpha * i + \beta \in (l : u : b : c)$, we have $l \leq \alpha * i + \beta \leq u$. We can easily find the least and largest integer of i. Let l' and u' be the least and largest integers of the above inequality. Then, $\alpha * i + \beta \in (l' : u' : \alpha)$. According to *Theorem 2*, let $(l' : u' : \alpha) \cap (l : u : b : c) = (\ (l'_0, l'_1, \ldots, l'_{b_1-1}) : (u'_0, u'_1, \ldots, u'_{b_1-1}) : LCM(\alpha, c)\ )$, where $l'_x \neq l'_y$ and $|l'_x - l'_y| < LCM(\alpha, c)$, $\forall\ x \neq y$.
- It is trivial that $\alpha * i + \beta \in (\ (l'_0, l'_1, \ldots, l'_{b_1-1}) : (u'_0, u'_1, \ldots, u'_{b_1-1}) : LCM(\alpha, c)\ ) \Rightarrow i \in (\frac{(l'_0 - \beta}{\alpha}, \ldots, \frac{l'_{b_1-1} - \beta}{\alpha}) : (\frac{(u'_0 - \beta}{\alpha}, \ldots, \frac{u'_{b_1-1} - \beta}{\alpha}) : \frac{LCM(\alpha, c)}{\alpha})$.

Q.E.D.

Theorem 4 *If $\mu_k(i_1, \cdots, i_n)$, $f_1(i_1, \cdots, i_n)$, $f_2(i_1, \cdots, i_n)$, $\cdots$, $f_n(i_1, \cdots, i_n)$ are linear affine index functions with one index variable, then $\mu_k(f_1(i_1, \cdots, i_n), \cdots, f_n(i_1, \cdots, i_n))$ is also linear affine index functions with one index variable.*

Proof:

Since $\mu_k(i_1, \cdots, i_n)$ is a linear affine index function. We assume $\mu_k(i_1, \cdots, i_n)$=$\alpha * i_p + \beta$, α and β are integer, $1 \leq p \leq n$. Then,

$\mu_k(f_1(i_1, \cdots, i_n), \cdots, f_n(i_1, \cdots, i_n)) = \alpha * f_p(i_1, \cdots, i_n) + \beta$.

Since $f_p(i_1, \cdots, i_n)$ is also a linear affine index function. We assume $f_p(i_1, \cdots, i_n)$=$\alpha' * i_q + \beta'$, α' and β' are integer, $1 \leq q \leq n$. Then,

$\mu_k(f_1(i_1, \cdots, i_n), \cdots, f_n(i_1, \cdots, i_n)) = \alpha * f_p(i_1, \cdots, i_n) + \beta = \alpha * (\alpha' * i_q + \beta') + \beta =$
$\alpha * \alpha' * i_q + (\alpha * \beta' + \beta)$

It is trivial that both $\alpha * \alpha'$ and $\alpha * \beta' + \beta$ are integers.

Q.E.D.

B Complexity Analysis

Diophantine Equation And Its Solution

Let $p * x + q * y = c$ denote a linear diophantine equation. Note that p,q are integers and x,y are integer variables. The first step for solving it is to run the extended Euclid's algorithm. The extended Euclid's algorithm takes p and q as input arguments and outputs three integers u,v and g where g=GCD(p,q), $g = p * u + q * v$. Then the set of all solution of (x, y) is $(u * c/g + T * q/g, v * c/g - T * p/g)$ where T $\in$ integer numbers. See [31] for more details.

Complexity Analysis 1 Assume $\mathcal{C}_1 = ((l_1^1, l_2^1, \ldots, l_n^1) : (u_1^1, u_2^1, \ldots, u_n^1) : S)$ and $\mathcal{C}_2 = ((l_1^2, l_2^2, \ldots, l_m^2) : (u_1^2, u_2^2, \ldots, u_m^2) : S')$. Then,$\mathcal{C}_1 \cap \mathcal{C}_2$ can be performed in time $O(min(log\ S, log\ S') + m * n)$.

Proof:

See *Theorem 2*. $\mathcal{C}_1 \cap \mathcal{C}_2 = \bigcup_{\substack{i=1,n \\ j=1,m}} ((l_i^1 : u_i^1 : S) \cap (l_j^2 : u_j^2 : S'))$

The above equation shows that the solution of $\mathcal{C}_1 \cap \mathcal{C}_2$ needs $m * n$ intersections of two RSDs. $m * n$ intersections of two RSDs in this case only need to solve the extended Euclid's algorithm once as all of them are with the same input S and S'. Therefore, the total running time is $O(min(log\ S, log\ S') + m * n)$.

Q.E.D.

Complexity Analysis 2 The normalization of $\alpha * i + \beta \in (l : u : b : c)$ can be performed in time $O(min(log\ \alpha, log\ c) + b)$.

Proof:

The normalization of the BSD requires to do b normalizations of RSD. However, it only requires to run the extended Euclid algorithm once (CSD properties with the common stride). Thus, the total running time is $O(min(log\ \alpha, log\ c) + b)$.
Q.E.D.