Efficient Compilation of Forall Statement with Runtime Support

Xiaoming Li*, Yuhong Wen

Parallel Software System Group in NPAC at
Syracuse University,
111 College Place,
Syracuse, New York, 13244-4100

October 8, 1996

Abstract

This is an implementation document, intended as a definitive guidence for compiling FORALL statements in an HPF program, the core activity in our effort of constructing an HPF compiler on message passing platforms.

After introducing some background about the document, a model for one dimensional array and one index FORALL statements is presented. The algorithm for determining local DO loop bounds is introduced in section 3, followed by a section on communication detection and generation. Section 5 is a discussion on generalization of previous result to multi dimensional arrays and multi indices FORALL stetements. In the appendix, a study on so-called strided-subintervals is given, which serves as theoretical foundation for the techniques presented in section 4. We also provide the algorithms (as currently implemented and used by our compiler) on communication detection and coefficient calculation in appendix C and D.

*Visiting from Harbin Institute of Technology of China

1 Introduction

There are four primary language features in HPF that enable programmer to describe data parallelism.

- Array (array section) assingment statement (conditional or not);
- FORALL statement (construct);
- Intrinsic functions with array as argument; and
- DO independent directive.

FORALL statement is of core importance, since

- It's semantics is a 'true super set' of array assignment statement;
- The most prominent addition from Fortran 90 to Fortran 95 is the FORALL statement/construct; and
- FORALL construct can be equivalently seen as a sequence of FORALL statements.

Thus, effective compilation of FORALL statements is essential for an effective HPF compiler.

As we know, the same problem has been discussed by others, such as in [4] and [3]. But we found that they are not general enough and can not be directly used for our current activity, though ideas are borrowed. We need a document for directly guiding our compiler implementation.

In general, a compiler for data parallel language has four kinds of job to do.

- Data partitioning;
- Computation partitioning;
- Communication detection and insertion; and
- node program generation.

If we see a source program as a system, compilation process as an optimization process. This is indeed a highly complicated, convoluted optimization problem. HPF language tries to free the compiler from the first item by encouraging programmer to specify data partition; if we follow owner computes rule, the second item is also essentially removed, though sometimes it's non trivial to determine who is the owner. Thus, our compiler has been left with two items to work on. It's great ! This report mainly addresses the first of the two, while some flavor of node program generation will be observed on the way.

2 Model

We consider the following canonical form

```
FORALL (i=l:u:s) X(a0*i+b0) = Y(a1*i+b1)
```

where X and Y are distributed onto the same processor grid (assume p processors, numbered as 0, 1, ..., p-1) by HPF directives. More specifically, we observe the following attributes pertaining to X and Y.

dimension:	$X(l_x : u_x)$	$Y(l_y : u_y)$
template:	$T_x(l_{tx} : u_{tx})$	$T_y(l_{ty} : u_{ty})$
alignment:	$(a_x * i + b_x)$	$(a_y * i + b_y)$
distribution:	d_x	d_y

This is a 22 variable system, where $l, u, s, a_0, b_0, a_1, b_1, p, l_x, u_x, l_y, u_y, l_{tx}, u_{tx}, l_{ty}, u_{ty}, a_x, b_x, a_y, b_y$ are integer variables and d_x, d_y are either `BLOCK` or `CYCLIC`.

Let $t_x = u_{tx} - l_{tx} + 1$, the declared size of template T_x, and we call $\overline{t_x} = \lceil \frac{t_x}{p} \rceil \cdot p$ the effective size of it, i.e. each of the p processors actually allocates $\overline{t_x}/p$ storage for local array X. A similar statement goes for T_y.

When we say "variables" in the above, we mean they can actually be variables defined in source program so that their values may not be determined at compile time, for instance, input when program is running, or dummies in subprograms.

For convenience of illustration, we shall use a table similar to the above table to represent a piece of HPF program that involves one FORALL statement. For example, the following program

```
      REAL X(80), Y(100)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE TX(200), TY(300)
!HPF$ ALIGN X(i) WITH TX(2*i+5)
!HPF$ ALIGN Y(i) WITH TY(3*i-1)
!HPF$ DISTRIBUTE TX(BLOCK) ONTO P
!HPF$ DISTRIBUTE TY(CYCLIC) ONTO P
      FORALL (i=-1:30:2) X(2*i+5) = Y(3*i+10)
      END
```

will be identified as

4; -1:30:2	X(2*i+5)	Y(3*i+10)
dimension:	X(1:80)	Y(1:100)
template:	Tx(1:200)	Ty(1:300)
alignment:	(2*i+5)	(3*i-1)
distribution:	BLOCK	CYCLIC

The task is to design a scheme that turns this FORALL statement to an SPMD node program segment, such that a collective execution of multi copies of the node program achieves the same semantic effect of the FORALL statement.

What we are interested is communication patterns that may present in such a FORALL statement. For instance,

8; 0:28:2	X(i)	Y(2*i+13)
dimension:	X(-5:77)	Y(4:127)
template:	Tx(1:1328)	Ty(1:664)
alignment:	(4*i+21)	(i+8)
distribution:	BLOCK	BLOCK

has no communication need, while

8; 0:10:3	X(i)	Y(i+15)
dimension:	X(-9:83)	Y(6:91)
template:	Tx(1:186)	Ty(1:190)
alignment:	(2*i+20)	(2*i+8)
distribution:	CYCLIC	CYCLIC

needs a shift type communication, and

8; 6:271:4	X(i)	Y(2*i-2)
dimension:	X(0:341)	Y(3:859)
template:	Tx(-2:362)	Ty(7:1788)
alignment:	(i+7)	(2*i+3)
distribution:	BLOCK	CYCLIC

needs some 'chaotic' communication, which we shall refer as remap communication. The core of this paper is a development of an algorithm that distinguish the above three types of communication patterns. The idea behind this distinction is

- For no communication, our node program should result in no data movement in runtime.
- For shift communication, a ghostarea support will reduce runtime data movement to minimum.
- For remap communication, a general data movement runtime routine, remap(), is supplied to handle this case. Optimization is achived in the design and implementation of this runtime routine.

One possible question is: why just three ? can we further break down the case remap ? Our understanding is: possible, but may not be worthy the effort.

Using owner computes rule, we see the node program segment corresponding to the FORALL statement might be reasonablly generated by the following procedure: (we assume some runtiem support, but no ghostarea support.)

1. Insert a call to runtime function as

```
CALL loop_bounds(dad_x,1,l,u,s,ll,lu,ls)
```

   where dad_x is a descriptor for distributed array X.

2. Check if communication is detectable. Communication is detectable if all the above 22 parameters are constants. If not detectable, go to 6.

3. Detect communication pattern between arrays X and Y under this FORALL statement. If shift communication is detected, the shift amount is also determined.

4. If no communication, insert

```
CALL coef(dad_y,1,l,u,s,a1,b1,0,u,v)
DO i = ll,lu,ls
   X(i) = Y(u*i+v)
END DO
```

   where dad_y is a descriptor for distributed array Y.

5. Else if shift communication, insert

```
tshift(dad_tmpy, dad_y, 1, amount)
CALL coef(dad_tmpy,1,l,u,s,a1,b1,amount,u,v)
DO i = ll,lu,ls
   X(i) = tmpy(u*i+v)
END DO
```

   where dad_tmpy is a descriptor for some temporary storage distributed the same as Y.

6. Else remap communication is needed, insert

```
remap(dad_tmpx, dad_y)
DO i = ll,lu,ls
   X(i) = tmpx(i)
END DO
```

   where dad_tmpx is a descriptor for some temporary storage distributed the same as X.

The above procedure has some small problem, since coefficient u and v so calculated may not be integers, though u*i+v for effective value of i is always integer. Thus, an alternative is the following:

1. Insert a call to runtime function as

```
CALL loop_upper_bound(dad_x,1,l,u,s,ub)
```

where dad_x is a descriptor for distributed array X.

2. Check if communication is detectable. Communication is detectable if all the above 22 parameters are constants. If not detectable, go to 6.

3. Detect communication pattern between arrays X and Y under this FORALL statement. If shift communication is detected, the shift amount is also determined.

4. If no communication, insert

```
CALL coef(dad_x,1,l,u,s,a0,b0,0,u0,v0)
CALL coef(dad_y,1,l,u,s,a1,b1,0,u1,v1)
DO i = 0, ub
   X(u0*i+v0) = Y(u1*i+v1)
END DO
```

where dad_y is a descriptor for distributed array Y.

5. Else if shift communication, insert

```
tshift(dad_tmpy, dad_y, 1, amount)
CALL coef(dad_x,1,l,u,s,a0,b0,0,u0,v0)
CALL coef(dad_tmpy,1,l,u,s,a1,b1,amount,u,v)
DO i = 0, ub
   X(u0*i+v0) = tmpy(u1*i+v1)
END DO
```

where dad_tmpy is a descriptor for some temporary storage distributed the same as Y.

6. Else remap communication is needed, insert

```
remap(dad_tmpx, dad_y)
CALL coef(dad_x,1,l,u,s,a0,b0,0,u0,v0)
DO i = 0, ub
   X(u0*i+v0) = tmpx(u0*i+v0)
END DO
```

where dad_tmpx is a descriptor for some temporary storage distributed the same as X.

This time, every u and v are integers. Nevertheless, one might prefer the following:

1. Insert a call to runtime function as

```
CALL loop_bounds(dad_x,1,l,u,s,ll,lu,ls)
```

where dad_x is a descriptor for distributed array X.

2. Check if communication is detectable. Communication is detectable if all the above 22 parameters are constants. If not detectable, go to 6

3. Detect communication pattern between arrays X and Y under this FORALL statement. If shift communication is detected, the shift amount is also determined.

4. If no communication, insert

```
CALL loop_bounds(dad_y,1,l,u,s,lly,luy,lsy)
iy = lly
DO i = ll,lu,ls
   X(i) = Y(iy)
   iy = iy + lsy
END DO
```

   where dad_y is a descriptor for distributed array Y.

5. Else if shift communication, insert

```
tshift(dad_tmpy, dad_y, 1, amount)
CALL loop_bounds(dad_tmpy,1,l,u,s,lltmpy,lutmpy,lstmpy)
itmpy = lltmpy
DO i = ll,lu,ls
   X(i) = tmpy(itmpy)
   itmpy = itmpy + lstmpy
END DO
```

   where `dad_tmpy` is a descriptor for some temporary storage distributed the same as Y.

6. Else remap communication is needed, insert

```
remap(dad_tmpx, dad_y)
DO i = ll,lu,ls
   X(i) = tmpx(i)
END DO
```

   where dad_tmpx is a descriptor for some temporary storage distributed the same as X.

There are three issues here

- Determine local loop bounds `ll,lu`, and `ls` or determine the local upper bound `lu`.
- Detect communication at compiler time, and generate appropriate communication calls in node program, if needed.

- Figure out coefficients u and v.

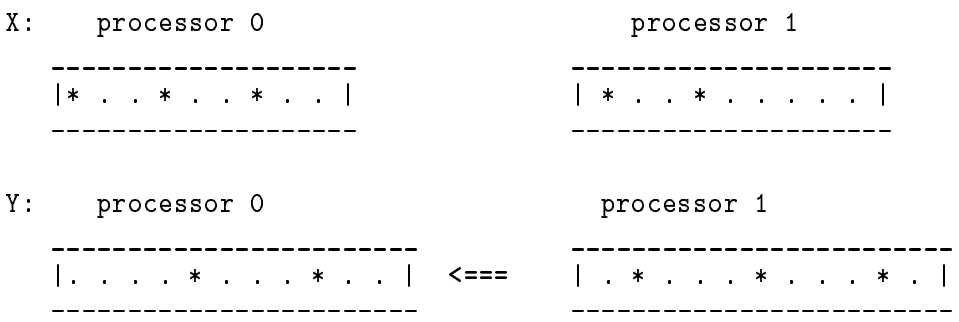
To facilitate communication, we have assumed the following two routines in runtime system.

- `tshift:` shift the distributed array Y to some temporary TMPY by certain displacement, which has the same shape and distribution as Y.
- `remap:` remap elements of Y that are involved in the FORALL to a temporary TMPX, which has the same distribution as X.

As it will become clear later, the `tshift` function is to shift the Y array in terms of TEMPLATE positions, instead of global array element positions as for CSHIFT or EOSHIFT in Fortran 90.

Example. Consider the following case.

If we have the corresponding X array elements and Y array elements distributed in the following pattern.



When implementing the forall assignment in our model, we will keep the left hand side clause unmoved. That means they will not be moved into a temporary buffer and the operation will be directly on the local array. Apply this assumption to the above case, we just move the global template of Y array one position from right to left, then Y array elements will have the same shape and distribution as the corresponding X array elements. So in this case, we only need to shift the template of Y array to complete the operation and keep the X array unmoved. This is different from the shift communication on Y array elements, and actually that is not able to implement this.

The `remap` function is to change the shape and distribution of Y array to the shape and distribution of X array. It first forms a section of Y array elements, then redistributed these data in the shape and distribution the same as X array to ensure that every X array element in the forall statement will get its corresponding Y array element in the same processor. (some careful reader may laught here: since you have such a wonderful runtime function, why can't you move Y directly to X, instead of moving it first to some TMPX, then TMPX to X via a DO loop ?)

3 Determination of local loop bounds

As mentioned in previous section, a runtime function loop_bounds() is called in node program to compute local loop bounds on each processor for corresponding global FORALL index range, with respect to array (left hand side in the assignment statement). We discuss algorithms used by loop_bound() in this section.

It is equivalent to ask:

> Given array section X(l:u:s) and DAD of X, what is the corresponding local index-triplet (ll(i),lu(i),ls(i)) for each processor $i \in (1:p)$.

Local bounds is also a function of index base of local array declared in node program. In what follows, we consider 0-based index for all local arrays, though global HPF arrays may be declared arbitrarily. For ease of reference, we recall our model again.

p; l:u:s	$X(a_0 i + b_0)$	$Y(a_1 i + b_1)$
dimension:	$X(l_x : u_x)$	$Y(l_y : u_y)$
template:	$T_x(l_{tx} : u_{tx})$	$T_y(l_{ty} : u_{ty})$
alignment:	$(a_x * i + b_x)$	$(a_y * i + b_y)$
distribution:	d_x	d_y

The algorithm is different for different distribution mode.

- BLOCK distribution

```
if s does not divede (u-l), set u=floor((u-l)/s)*s;
w = ceil(tx/p);
tl = l_tx + (i-1)*w;    tu = l_tx + i*w - 1;
if (tu < ax*l+bx) or (ax*u+bx < tl), then
   set ll = 0; lu = -1; ls = 1;    // no effective element
else
   find the least j in (l:u:s)
      such that ax*j+bx >= tl;
   set ll = ax*j+bx - tl;
   find the greatest j in (l:u:s)
      such that ax*j+bx <= tu;
   set lu = ax*j+bx - tl;
   set ls = ax*s;
end if
```

- CYCLIC distribution

  We see two arithmetic progressions here

  processor i's relative positions in template: i, i+p, i+2*p, ..., i+n*p, ...

(l:u:s) element relative positions in template: o, o+q, o+2*q, ..., o+m*q, ...

where o = ax*l+bx - l_tx + 1, q = ax*s. Then we would have to solve the following Diophantine equation for each processor i:

$$i + n * p = o + m * q \tag{1}$$

with the boundary conditions,

$$n \geq 0, i + n * p \leq u_{tx} - l_{tx} + 1 \tag{2}$$

$$0 \leq m \leq \frac{u - l}{s} \tag{3}$$

We know the equation (1) has solution iff gcd(q,p) divides $\mid i - o \mid$. And if so, we may employ Extended Euclidean algorithm to find all pairs of (n,m) that satisfy the equation, and then impose the boundary conditions.

Now, the algorithm outline for computing ll and lu on processor i,

```
if gcd(q,p) divides |i-o| then
   using Euclidean algo. to compute the solution (n,m) of eq. (1);
   find the subset from the solution set that satisfy the constraints
   for n and m;
   if the subset is not empty, then
      ll = minimum n; lu = maximum n;
   else
      ll = 0; lu = -1; ls = 1; no effective element
   end if
else
   ll = 0; lu = -1; ls = 1; no effective element
end if
```

Banerjee [6] had a very practical treatment on how to solve the Diophantine equation. We have included it in Appendix B for the sake of completeness of this document.

To figure out `ls`, we need the least common multiple of ax*s and p, and denote it by lcm(ax*s,p). Here is the idea: processor i see its 'name' in template every p positions, while X(l:u:s) sees its element in template every ax*s positions. Thus, if processor i and X(l:u:s) meet at position t of the template then the next time they meet again will be at position $t + lcm(ax * s, p)$. Since template is cyclically distributed (p positions in template is equal to 1 position in local array), processor i will see the next element of X(l:u:s) locally at position $\frac{lcm(ax*s,p)}{p}$ away, i.e., `ls` $= \frac{lcm(ax*s,p)}{p}$, processor independent.

Example. Consider the following program.

```
      PROGRAM MAIN
      DIMENSION X(1:20)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE T(1:60)
!HPF$ DISTRIBUTE T(CYCLIC) ONTO P
!HPF$ ALIGN X(i) WITH T(2*i+1)
      ...
      FORALL (i=4:19:3) X(i) = ...
      ...
      END
```

To compute `ll and lu` for processor i= 1,2,3,4, we see
o=2*4+1-1+1=9
q = 2*3 = 6
o, o+6, o+12, o+18, ..., o+6*m, ...
i, i+4, i+8, i+12, ..., i+4*n, ...
The equation to be solved is

$$i + 4n = 9 + 6m \tag{4}$$

i=2,4: No solution.

i=1: $1 + 4n = 9 + 6m, n = 2 + \frac{3}{2} \cdot m$

constraint: $1 + 4n \leq 60 - 1 + 1, n \leq 14$

$0 \leq m \leq (19 - 4)/3 = 5$

m = 0, 2, 4

n = 2, 5, 8

That is, ll(1) = 2, lu(1) = 8.

i=3: $3 + 4n = 9 + 6m, n = \frac{3+3m}{2}$

constraint: $1 + 4n \leq 60 - 1 + 1, n \leq 14$

$0 \leq m \leq (19 - 4)/3 = 5$

m = 1, 3, 5

n = 3, 6, 9

That is, ll(3) = 3, lu(3) = 9.

Finally, lcm = lcm(2*3,4) = 12. ls = lcm/p = 3
In summary, global X(4:19:3) maps to local (2:8:3) for processor 1, (3:9:3) for processor 3, and processor 2 and 4 get no elements of X(4:19:3).

4 Communication detection and generation

We first develop a concept of shift-isomorphism beteween two array sections.

Recalling arrays X and Y are aligned with templates T_x and T_y, as illustrated in previous section, we can see a mapping from array sections to template sections, i.e., from $X(l_x : u_x : s_x)$, we have $T_x(l'_{tx} : u'_{tx} : s'_{tx})$. And from $Y(l_y : u_y : s_y)$, we have $T_y(l'_{ty} : u'_{ty} : s'_{ty})$.

The mapping functions are

$$l'_{tx} = a_x \cdot l_x + b_x, \quad u'_{tx} = a_x \cdot u_x + b_x, \quad s'_{tx} = a_x \cdot s_x$$

A similar situation exists for Y .

We say $X(l_x{:}u_x{:}s_x)$ and $Y(l_y{:}u_y{:}s_y)$ are shift-isomorphic iff

$$\lfloor \frac{u_x - l_x}{s_x} \rfloor = \lfloor \frac{u_y - l_y}{s_y} \rfloor, \quad s'_{tx} = s'_{ty}, \quad and \quad \overline{t_x} = \overline{t_y} \tag{5}$$

for block distribution, and

$$\lfloor \frac{u_x - l_x}{s_x} \rfloor = \lfloor \frac{u_y - l_y}{s_y} \rfloor, \quad s'_{tx} = s'_{ty} \quad mod\ (p) \tag{6}$$

for cyclic distribution.

And the shift-amount from $Y(l_y : u_y : s_y)$ to $X(l_x : u_x : s_x)$ is equal to $(l'_{ty} - l_{ty}) - (l'_{tx} - l_{tx})$ (parameters from templates), $(l'_{ty} - l_{ty})$ is the first element position in Y array and $(l'_{tx} - l_{tx})$ is the position in Tx that X will be shifted to. Here we imply that positive amount indicates shift to the left, while negative indicates shift to the right.

The intention is clear: we want a condition for `tshift` being adequate.

Instantiating it with our model,

```
FORALL (i=l:u:s) X(a0*i+b0) = Y(a1*i+b1)
```

we have,

$$X(l_x : u_x : s_x) = X(a0 \cdot l + b0 : a0 \cdot u + b0 : a0 \cdot s),$$
$$Y(l_y : u_y : s_y) = Y(a1 \cdot l + b1 : a1 \cdot u + b1 : a1 \cdot s)$$

and

$l'_{tx} = a_x \cdot (a0 \cdot l + b0) + b_x$	$l'_{ty} = a_y \cdot (a1 \cdot l + b1) + b_y$
$u'_{tx} = a_x \cdot (a0 \cdot u + b0) + b_x$	$u'_{ty} = a_y \cdot (a1 \cdot u + b1) + b_y$
$s'_{tx} = a_x \cdot a0 \cdot s$	$s'_{ty} = a_y \cdot a1 \cdot s$

Thus, $X(a0 \cdot l + b0 : a0 \cdot u + b0 : a0 \cdot s)$ is shift-isomorphic to $Y(a1 \cdot l + b1 : a1 \cdot u + b1 : a1 \cdot s)$ iff $a_x \cdot a0 \cdot s = a_y \cdot a1 \cdot s$, or simply

$$a_x \cdot a0 = a_y \cdot a1 \tag{7}$$

and

$$amount_{shift} = (a_y \cdot (a1 \cdot l + b1) + b_y - l_{ty}) - (a_x \cdot (a0 \cdot l + b0) + b_x - l_{tx}) \tag{8}$$

for `BLOCK` distribution, or

$$s'_x = s'_y \ mod \ p \tag{9}$$

for `CYCLIC` distribution. Special care has to be taken to calculate shift amount for `CYCLIC` distribution. We need to figure out which processors contain the first X and Y elements in question, and their local positions, respectively.

Let p_x and p_y be the processors, and $first_x, first_y$ be local positions. We have,

```
if (p_x > p_y and first_y > first_x)
   amount = p_y + (p - p_x);
else
   amount = p_y - p_x;
```

- If Tx and Ty have the same size, and are distributed same way, a shift of Y with that amount does the job. Then the node program may look something like:

```
CALL loopBounds(DAD_X,l,u,s,ll,lu,ls)
CALL tshift(Y,amount,T)
DO i = ll,lu,ls
   X(i) = T(i)
END DO
```

Example 1: Consider the following program.

```
      REAL X(1:20),Y(1:20),Z(1:20)
!HPF$ PROCESSORS P(4)
!HPF$ ALIGN Y(i) WITH X(i)
!HPF$ ALIGN Z(i) WITH X(i)
!HPF$ DISTRIBUTE X(block)
      ...
      FORALL (i=3:18:3) X(i) = Y(i+2)
      ...
      FORALL (i=3:18:5) X(i) = Z(i+2)
      ...
      END
```

Here is the situition in templates for the first FORALL

```
Tx:    --x--x--x--x--x--x--
       11111222223333344444

Ty:    ----y--y--y--y--y--y
       11111222223333344444
```

We see a shift of Y with amount +2 is necessary. For the situition of the second FORALL,

```
Tx:    --x----x----x----x--
       11111222223333344444

Tz:    ----z----z----z----z
       11111222223333344444
```

A shift of Z with amount +2 will make the elements in Tx and Tz aligned, but is not necessary.

As we may have noticed in this example, a shift may not be needed if we allow `X(i) = T(i+v)` in the DO loop, under certain condition. We defer its discussion after getting a good picture on the main stream.

- If T_x and T_y have the same size, but they are distributed in different fashions, we generally can not meet the communication requirement by a shift.

  Example 2: Consider the same program, but Y being cyclically distributed.

  ```
  Tx:    --x--x--x--x--x--x--
         11111222223333344444

  Ty:    ----y--y--y--y--y--y
         12341234123412341234
  ```

  We see it is not possible to accomplish the communication required by a shift.

- If T_x and T_y does not have the same size, no matter if they are distributed in the same fashion or not, we generally can not meet the communication requirement by a shift.

Example 3: Consider the same X, but Y with 32 elements.

```
Tx:    --x--x--x--x--x--x--
       11111222223333344444

Ty:    --------y--y--y--y--y--y--------
       11111111222222223333333344444444
```

We see it is not possible to accomplish the communication required by a shift, since a shift can not make each processor contain the same number of x's and y's, respectively.

The above discussion actually motivates a generalization of the concept of shift-isomorphic to shift-homomorphic. We first state the following useful lemma.

Lemma 1 (`BLOCK` distribution) Consider two intervals $I_1 = [1 : t_1]$ and $I_2 = [1 : t_2]$. Assume p (some positive integer) is a common divisor of t_1 and t_2, and write $w_1 = \frac{t_1}{p}$ and $w_2 = \frac{t_2}{p}$. Let s_1 and s_2 be two positive integers satisfying

$$\frac{s_1}{s_2} = \frac{t_1}{t_2} \tag{10}$$

Then the mapping Φ from I_1 to I_2: $i \longmapsto \lceil i \cdot \frac{t_2}{t_1} \rceil$ has the following property:
For any strided subinterval of I_1,

$$i, i + s_1, i + 2 \cdot s_1, ..., i + k \cdot s_1$$

if

$$(n-1) \cdot w_1 + 1 \leq i, \text{ and } \; i + k \cdot s_1 \leq n \cdot w_1$$

for some positive integer n, then

$$(n-1) \cdot w_2 + 1 \leq \Phi(i), \text{ and } \; \Phi(i) + k \cdot s_2 \leq n \cdot w_2.$$

Proof: From $(n-1) \cdot w_1 + 1 \leq i$, we can have

$$\begin{gathered}((n-1) \cdot w_1 + 1) \cdot \frac{t2}{t1} \leq i \cdot \frac{t2}{t1}, \\ \lceil ((n-1) \cdot w_1 + 1) \cdot \frac{t2}{t1} \rceil \leq \lceil i \cdot \frac{t2}{t1} \rceil = \Phi(i), \\ \lceil (n-1) \cdot w_2 + \frac{t2}{t1} \rceil \leq \Phi(i) \\ (n-1) \cdot w_2 + 1 \leq (n-1) \cdot w_2 + \lceil \frac{t2}{t1} \rceil \leq \Phi(i)\end{gathered}$$

Furthermore, from $i + k \cdot s_1 \leq n \cdot w_1$, we have

$$\begin{gathered}(i + k \cdot s_1) \cdot \frac{t2}{t1} \leq n \cdot w_1 \cdot \frac{t2}{t1} \\ \lceil i \cdot \frac{t_2}{t_1} \rceil + k \cdot s_2 = \lceil (i + k \cdot s_1) \cdot \frac{t2}{t1} \rceil \leq \lceil n \cdot w_1 \cdot \frac{t2}{t1} \rceil \\ \Phi(i) + k \cdot s_2 \leq \lceil n \cdot w_1 \cdot \frac{t2}{t1} \rceil = n \cdot w_2\end{gathered}$$

□

Example 4: Consider

$$t_1 = 60,\ t_2 = 84,\ p = 4,\ s_1 = 5,\ s_2 = 7$$

```
1--...15--...30--...45--...60

1----....21----....42----....63----....84
```

We have $w_1 = 60/4 = 15,\ w_2 = 84/4 = 21$. The subinterval of I_1, 18, 23, 28 is bounded by $w_1 + 1$ and $2 \cdot w_1$. The correspond subinterval in I_1, according to the lemma, is 26, 33, 40, which is bounded by $w_2 + 1$ and $2 \cdot w_2$.

Let t be the size of a template T. We use $\overline{t(p)}$ to denote $p \cdot \lceil \frac{t}{p} \rceil$. When no ambiguity results, we may simply use $\overline{t}$ for $\overline{t(p)}$. Thus, $\overline{t_x} = \overline{u_{tx} - l_{tx} + 1}$ and $\overline{t_y} = \overline{u_{ty} - l_{ty} + 1}$.

We say $X(l_x : u_x : s_x)$ and $Y(l_y : u_y : s_y)$ are shift-homomorphic if

$$\lfloor \frac{u_x - l_x}{s_x} \rfloor = \lfloor \frac{u_y - l_y}{s_y} \rfloor \tag{11}$$

and

$$s'_{tx} = s'_{ty} \cdot \frac{\overline{t_x}}{\overline{t_y}} \tag{12}$$

for BLOCK-distributed X and Y, or

$$s'_{tx} \equiv s'_{ty} \pmod{p} \tag{13}$$

for CYCLIC-distributed X and Y.

Shift-homomorphism allows us to efficiently handle some cases where X and Y have templates of different sizes. Clearly, shift-isomorphism is a special case of shift-homomorphism.

Instantiating it with our model,

```
FORALL (i=l:u:s) X(a0*i+b0) = Y(a1*i+b1)
```

we have,

$X(l_x : u_x : s_x) = X(a0 \cdot l + b0 : a0 \cdot u + b0 : a0 \cdot s)$,
$Y(l_y : u_y : s_y) = Y(a1 \cdot l + b1 : a1 \cdot u + b1 : a1 \cdot s)$
and

$l'_{tx} = a_x \cdot (a0 \cdot l + b0) + b_x$	$l'_{ty} = a_y \cdot (a1 \cdot l + b1) + b_y$
$u'_{tx} = a_x \cdot (a0 \cdot u + b0) + b_x$	$u'_{ty} = a_y \cdot (a1 \cdot u + b1) + b_y$
$s'_{tx} = a_x \cdot a0 \cdot s$	$s'_{tx} = a_y \cdot a1 \cdot s$

Thus, $X(a0 \cdot l + b0 : a0 \cdot u + b0 : a0 \cdot s)$ is shift-homomorphic to $Y(a1 \cdot l + b1 : a1 \cdot u + b1 : a1 \cdot s)$ if $a_x \cdot a0 \cdot s \cdot \overline{t_y} = a_y \cdot a1 \cdot s \cdot \overline{t_x}$, or simply

$$\frac{a_x \cdot a0}{a_y \cdot a1} = \frac{\overline{t_x}}{\overline{t_y}} \tag{14}$$

for BLOCK-distribution, or

$$a_x \cdot a0 \cdot s \equiv a_y \cdot a1 \cdot s \pmod{p} \tag{15}$$

for CYCLIC-distribution.

Theorem 1 If $X(l_x : u_x : s_x)$ and $Y(l_y : u_y : s_y)$ are shift-homomorphic, then a shift of Y suffices to make X(i) and Y(j) in the same processor for $i = l_x : u_x : s_x$, $j = l_y : u_y : s_y$, correspondingly.

Proof: Let k_x be the relative position of $X(l_x)$ in T_x. We locate $k_y = \lceil k_x \cdot \frac{\overline{t_y}}{\overline{t_x}} \rceil$ in T_y. Shift Y such that $Y(l_y)$ takes the position k_y in T_y. We claim the job is done, since we can easily see an "isomorphism" between lemma one and the theorem, namely we can have a 1-1 onto mapping between the parameters of the two systems. Thus, an "isomorphic" conclusion is obtained.

□

Example: Consider $X(l_x : u_x : s_x)$ takes position 33,40,47,54,61,68 in its template. And $Y(l_y : u_y : s_y)$ takes 1,6,11,16,21,26 in its template.

```
Tx:   ----....21----....42----....63----....84

Ty:   --...15--...30--...45--...60
```

$k_y = \lceil 33 \cdot \frac{5}{7} \rceil = 24$. That is, we move Y such that $Y(l_y : u_y : s_y)$ takes the positions 24,29,34,39,44,49. It is easy to verify that each processor contains the same number of x's and y's, correspondingly. That is, the shift amount for Y is 1-24 = -23.

Parameters in the node program: Once it is detected that $X(l : u : s)$ and $Y(a \cdot l + b : a \cdot u + b : a \cdot s)$ are shift-homomorphic, the node program may take the form (the first of three mentioned in model section):

```
shift Y to some temporary T
DO i=ll,lu,ls
  X(i)=T(u*i+v)
END DO
```

We want to determine u0, v0, u1, and v1. First of all, we claim u0 ans u1 are processor independent, $u0 = s_x,\ u1 = s_y$. To compute v0, v1, we first determine the index k_x to the global template T_x of the first effective element for each processor. Then using $k_y = \lceil k_x \cdot \frac{t_y}{t_x} \rceil$ to calculate corresponding positions in T_y.

The algorithm is:

$k_x = a_x * l + b_x$;
if $(k_x > l_tlb_x)$ we get the k_x;
else $k_x = k_x + s_x * \lceil \frac{l_tlb_x - k_x}{s_x} \rceil$.

We then observe that $X(ll)$ is in fact $X(k_x - l_tlb_x + 1)$ for X being defined 1-based in node program, and $X(k_x - l_tlb_x)$ for X 0-based. Simalarly, $T(u * ll + v)$ is in fact $T(k_y - l_tlb_y + 1)$ for Y (or T) being defined 1-based in node program, and $T(k_y - l_tlb_y)$ for Y 0-based, respectively. Thus, we would either solve

$$(k_x - l_tlb_x + 1) \cdot u + v = (k_y - l_tlb_y + 1) \tag{16}$$

if 1-base is used as convention for array definitions in node program. Or

$$(k_x - l_tlb_x) \cdot u + v = (k_y - l_tlb_y) \tag{17}$$

if 0-base is used as convention for array definitions in node program. to get v for each processor. For the above example, $k_x = 33, 47, 68$ and $k_y = 24, 34, 49$ for processors 2, 3, 4, respectively. If 1-base is taken as convention, $u = \frac{5}{7}$, and
$v(2) = (24 - 16 + 1) - (33 - 22 + 1) \cdot u = \frac{3}{7}$
$v(3) = (34 - 31 + 1) - (47 - 43 + 1) \cdot u = \frac{3}{7}$
$v(4) = (49 - 46 + 1) - (68 - 64 + 1) \cdot u = \frac{3}{7}$
The v's happend to be the same. Then for every processor,

```
shift Y to some temporary T
DO i=ll,lu,ls
  X(i)=T((5*i+3)/7)
END DO
```

where (ll,lu,ls) = (12,19,7), (5,19,7), and (5,5,7) for processors 2, 3, and 4, respectively, which have corresponding (9,14,5), (4,14,5), and (4,4,5) triples for T, respectively.

If we consider $X(l_x : u_x : s_x)$ takes positions 6,12,18,24, 30,36,42,48 in its template. And $Y(l_y : u_y : s_y)$ takes 1,6,11,16,21,26,31,36 in its template. Then we would have $k_x = 6, 24, 48$ and $k_y = 5, 20, 40$ for processors 1, 2, 3, respectively. $u = \frac{5}{6}$, and
$v(1) = (5 - 1 + 1) - (6 - 1 + 1) \cdot u = 0$
$v(2) = (20 - 16 + 1) - (24 - 22 + 1) \cdot u = \frac{5}{2}$
$v(3) = (40 - 31 + 1) - (48 - 43 + 1) \cdot u = 5$
If 0-base is taken, we would end up with

$v(1) = (5-1) - (6-1) \cdot u = -\frac{1}{6}$
$v(2) = (20-16) - (24-22) \cdot u = \frac{7}{3}$
$v(3) = (40-31) - (48-43) \cdot u = \frac{29}{6}$

Improve efficiency: We also want to know the condition for no communication. Clearly, no communication needed iff every processor has the corresponding elements from both $X(l_x : u_x : s_x)$ and $Y(l_y : u_y : s_y)$. That is, a shift may not be necessary, even if the two arrays are not aligned as the mapping Φ in lemma. Appendix A gives us the algorithm for determining this situation, namely, we can effeciently determine whether each processor contains the same number of elements from both $X(l_x : u_x : s_x)$ and $Y(l_y : u_y : s_y)$. And appendix C gives the detail algorithm to implement the detect communication function in forall statement.

5 Multi dimension cases

We could discuss the following issues.

non-coupled indices, such as that occurs in Jacobi iterations.

coupled indices, such as that occurs in FFT.

Let's first consider the following canonical form:

```
FORALL (i=l1:u1:s1, j=l2:u2:s2)
X(a00*i+b00, a01*j+b01) = Y(a10*i+b10, a11*j+b11)
```

where X and Y are two-dimensional arrays distributed onto the same processor grid by HPF direc tives. Then the task is to determine the specific segment of an SPMD program on each processor , such that a collective execution of multi copies of the node program achieves the same semantic effect of the forall statement.

In our implementation, we use the following general strategies:

- all arrays are "linearized", namely , no array in the SPMD node program has more than one dimension.
- temporary arrays are allocated dynamically , each node program maintains a large one dimensional array, and temporary arrays are allocated from it.
- we will normalize the loop bounds to be

```
DO i = 0, i_ub
   DO j = 0, j_ub
    ..........
```

in node program. Thus, we need to calculate the index of both left side clause and right-hand side clauses.

First, we will determine if the corresponding X array elements and Y array elements are on the same processor (needs no communication) or we can use only shift communication to move the Y array elements to the processor where the corresponding X array elements are located, or we have to use remap function to move the Y array elements to the corresponding processor according to the owner computes rule. In this case, we will use the same concept of shift-homomorphism discussed in the previous section and then expand it to use in the multi-dimensional cases.

The detect-communication function to determinate the shift-homomorphism will be done in the compiler. We use the previous linear algorithm in each dimension of the multi-dimensional array to determine if it needs no-communication, or shift-communication, or remap on this dimension. If all the dimensions are shift-homomophism, the compiler will just insert a multi-tshift function in the translated codes to implement multi-dimensional

shift communication. If any one dimension is not shift-homomophism (needs remap on this dimension), then we say it needs remap function to implement this assignment. Because the implementation of all arrays of forall statement in node program are "linearized", we need to calculate the corresponding index in this one dimensional array . This will be done in the runtime by calling the coef function to each dimension of X array and Y array to calculate the value of coefficients u and v. (see appendix D the implementation algorithm of coef function).

So after that, the codes of the node program to implement the forall statement will be something like that: (attention: the size0 and size1 are the range of the first dimension in global template TX and TY. This is because we will implement the two dimensional operation in one dimension in the local processor. In FORTRAN, its storage is column first, after placing a whole column, then the second. So size is equal to the range of the first dimension of global template.)

- In case of no communication:

```
CALL coef(dad_x, 1, l1, u1, s2, a00, b00, 0, u00, v00)
CALL coef(dad_x, 2, l2, u2, s2, a01, b01, 0, u01, v01)
CALL coef(dad_y, 1, l1, u1, s2, a00, b10, 0, u10, v10)
CALL coef(dad_y, 2, l2, u2, s2, a11, b11, 0, u11, v11)

DO i = 0, i_ub    // i_ub is the local loop bound in the first dimension
  DO j = 0, j_ub   // j_ub is the local loop bound in the second dimension
     X((u00*i+v00) + (u01*j+v01)*size0)
        = Y((u10*i+v10) + (u11*j+v11)*size1) // in case of no-communication
  END DO
END DO
```

- In case of multi-tshift communication:

```
CALL multi_tshift(dad_x, dad_y, 2, amount)
// amount is a array containing the amount of sfhit communication in every dimension

CALL coef(dad_x, 1, l1, u1, s2, a00, b00, 0, u00, v00)
CALL coef(dad_x, 2, l2, u2, s2, a01, b01, 0, u01, v01)
CALL coef(dad_y, 1, l1, u1, s2, a00, b10, amount0, u10, v10)
CALL coef(dad_y, 2, l2, u2, s2, a11, b11, amount1, u11, v11)

DO i = 0, i_ub // i_ub is the local loop bound in the first dimension
  DO j = 0, j_ub // j_ub is the local loop bound in the second dimension
     X((u00*i+v00) + (u01*j+v01)*size0)
        = tmpy((u10*i+v10) + (u11*j+v11)*size1) // multi-shift communication
```

```
  END DO
END DO
```

- In case of remap:

```
CALL remap(dad_x, dad_y)
CALL coef(dad_x, 1, l1, u1, s2, a00, b00, 0, u00, v00)
CALL coef(dad_x, 2, l2, u2, s2, a01, b01, 0, u01, v01)

DO i = 0, i_ub // i_ub is the local loop bound in the first dimension
  DO j = 0, j_ub // j_ub is the local loop bound in the second dimension
      X((u00*i+v00) + (u01*j+v01)*size0)
        = tmpx((u00*i+v00) + (u01*j+v01)*size0) // remap
  END DO
END DO
```

In the above, we have discussed the most common case of forall statement with two-dimensional arrays. But there still exist some special cases with multi-dimensional array forall statements. We will discuss their implementation in the following respectively.

1. FORALL (i=l1:u1:s1, j=l2:u2:s2)
   X(a00*i+b00, a01*j+b01) = Y(a10*j+b10, a11*i+b11)

   In this case, the fist dimension index of Y array is corresponding to the second dimension of X array, and the second dimension is corresponding to the first dimension of X array. The specific example of this case is the following permutation calculation. It's often used in the matrix computing.

   ```
   FORALL (i=l1:u1:s1, j=l2:u2:s2) X(i,j) = Y(j,i)
   ```

   When implementing this kind of permutation computing, we first use detect communication to determine if X array and Y array are homo-morphism (the concept discussed in section 4) in every dimension. From that we can get the knowledge if the elements of two dimensional array X(i,j) and Y(j,i) are distributed in the same processors, or it only needs a multi-tshift in the global template of Y array, or it needs a remap to move the Y array elements to the corresponding processor. In the first two cases, we can use the same strategy as the canonical form discussed to implement the forall assignment.

   If multi-tshift communication is not enough to move the Y array elements to the corresponding processor, it means the X(i,j) and Y(j,i) are not homo-morphism, we need a remap function to implement this. When using remap function, in order to keep X(i,j) unmoved and make the temporary Y array the same shape and distribution

as X array, we construct a new DAD structure dad_tmp which points to the element section of Y array and the first dimension is mapped to the second dimension of Y array, the second dimension map to the first dimension of Y array. Then dad_tmp has the same shape of X array. Then we use remap function to move the dad_tmp elements to the corresponding X element position according to owner computes rule.

```
rng_i = pcrc_new_range_align(a00*l1+b00, a00*u1+b00, a00*s1,
0, pcrc_rng(dad_x, 1))
rng_j = pcrc_new_range_align(a01*l2+b01, a01*u2+b01, a01*s2,
0, pcrc_rng(dad_x, 2))

dad_tmp1 = pcrc_new_array_data(tmp1, pcrc_real, pcrc_size_real, &
  & 2, grp_p)
CALL pcrc_set_array_copy(dad_tmp1, 1, rng_i)
CALL pcrc_set_array_copy(dad_tmp1, 2, rng_j)

dad_ys = pcrc_new_array_section(2, dad_y)
CALL pcrc_set_array_triplet(dad_ys, 1, a10*l2+b10, a10*u2+b10,
a10*s2, dad_y, 2)
CALL pcrc_set_array_triplet(dad_ys, 2, a11*l2+b11, a11*u2+b11,
a11*s2, dad_y, 1)

CALL pcrc_remap(dad_tmp1, dad_ys)
```

Thus we construct a new section of Y array and it has the same shape with the left hand side clause. We then can use the method as the common multi-dimensional forall statement to implement this assignment.

2. FORALL (i=l1:u1:s1, j=l2:u2:s2)
   X(a00*i+b00, a01*j+b01) = Y(a10*i+b10)

   In this case, all the X array elements with the same first dimension index will get the same value from a one dimension array Y. Generally, the left hand side clause has different dimension from the right hand side clauses in the forall statement.

   To deal with this assignment, we need to use a scalar function to scale the Y array in that dimension to a temporary array T which has the same dimension as X array. In the actuall implementation, we first use detect communication to determine if X array in the first dimension is homo-morphism as Y array to determine if it needs no communication or tshift communication or remap. This implementation is the same as the one dimensional case we discussed before. After that, Y array elements have been moved to the corresponding processor as the first dimension as X array. Then we use a scalar function to Y array to form a two dimensional temporary array which has the same shape and distribution as X array to complete the assignment operation. The generated codes will be something like:

```
rng_i = pcrc_new_range_align(a00*l1+b00, a00*u1+b00, a00*s1,
0, pcrc_rng(dad_x, 1))
rng_j = pcrc_new_range_align(a01*l2+b01, a01*u2+b01, a01*s2,
0, pcrc_rng(dad_x, 2))

dad_tmp1 = pcrc_new_array_data(tmp1, pcrc_real, pcrc_size_real, &
& 2, grp_p)
CALL pcrc_set_array_copy(dad_tmp1, 1, rng_i)
CALL pcrc_set_array_copy(dad_tmp1, 2, rng_j)

dad_ys = pcrc_new_array_section(1, dad_y)
CALL pcrc_set_array_triplet(dad_ys, 1, a10*l1+b10, a10*u1+b10,
a10*s1, dad_y, 1)
CALL pcrc_set_array_scalar(dad_ys, 2, dad_x, 2)

CALL pcrc_remap(dad_tmp1, dad_ys)
```

3. FORALL (i=l1:u1:s1, j=l2:u2:s2)
   X(a00*i+b00, a01*j+b01) = Y(a10*i+b10,1)

   In this forall statement, although the right hand clause has the same dimensions as the left hand clause, its second dimension index is a constant, not changed with the forall index. So it is actually a one dimension array assigned to a two-dimensional array statement. When dealing with this kind of forall, we will first get a section of Y array in the first column, then then use detect communication to determine it is homo-mophism with X array in the first dimension. Then we use the same strategy to scale it to a two dimension array as discussed in the previous case.

```
rng_i = pcrc_new_range_align(a00*l1+b00, a00*u1+b00, a00*s1,
0, pcrc_rng(dad_x, 1))
rng_j = pcrc_new_range_align(a01*l2+b01, a01*u2+b01, a01*s2,
0, pcrc_rng(dad_x, 2))

dad_tmp1 = pcrc_new_array_data(tmp1, pcrc_real, pcrc_size_real, &
& 2, grp_p)
CALL pcrc_set_array_copy(dad_tmp1, 1, rng_i)
CALL pcrc_set_array_copy(dad_tmp1, 2, rng_j)

dad_ys = pcrc_new_array_section(1, dad_y)
CALL pcrc_set_array_triplet(dad_ys, 1, a10*l1+b10, a10*u1+b10,
a10*s1, dad_y, 1)
CALL pcrc_set_array_scalar(dad_ys, 2, dad_y, 2)
```

```
CALL pcrc_remap(dad_tmp1, dad_ys)
```

4. FORALL (i=l1:u1:s1)
   X(a00*i+b00, a01*i+b01) = Y(a10*i+b10)

   In this case, although the Y array is assigned to a two dimension array, the index in both dimen sions of X array have only one variable from the forall index. It actually implements a one dimension array assigned to a one dimension array.

   There are two ways to deal with this kind of forall statement. One is that, though the left hand clause expresses only a linear space in the X array , we still treat it as a two dimension clause. We will construct a whole two dimension section of X array to include all the linear space in the left hand side. Then the problem becomes a one dimension array assigned to a two dimension array, just as the second case we discussed. It only differs that we should add a conditional judge in the DO loop to implement the necessary assignment only.

   In this method, we can implement the forall assignment, but we can see, that we allocate a much larger two dimensional buffer ((u1-l1) x (u2-l2)) for the X array . Actually the left hand side clause will only use a small portion of it. This is because that the template TX which X array is aligned with is distributed in a two dimensional mode, and the left hand side clause is only a one dimensional space. And in our DAD structure right now, we can not describe the following one dimension array aligned with a two dimension template.

   ```
   ALIGN A(i) WITH TX(a0*i+b0, a1*i+b1)
   ```

   This is not a alignment in HPF , but if we implement this kind of alignment, we can easily complete the previous forall assignment. Actually , the assignment needs a alignment like this, and we will construct a temporary one dimension array aligned with template TX in this mode. So, if we add a vector descriptor in the DAD structure to describe this kind of alignment, it's easy to implement this kind of X(i,i) = Y(i) assignment. Meanwhile, with this vector descriptor, we also can describe the matrix permutation directly in the DAD, to implement the assignment of X(i,j) = Y(j,i).

5. Ghost area problem

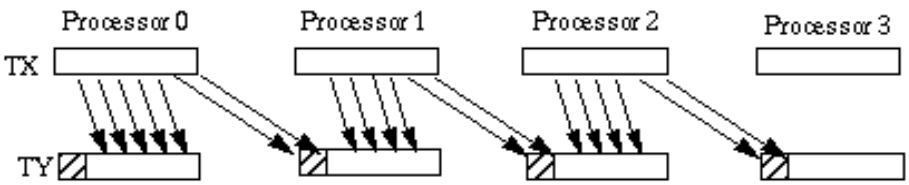
   ```
   FORALL (i=l1:u1:s1, j=l2:u2:s2) X(i, j) = Y(i+1, j)
   ```

   In this case of forall statement assignment, we assume that X array and Y array have the same distribution, then X(i,j) and Y(i,j) will be on the same processor. When implementing X(i,j) = Y(i+1, j), we need shift Y array in the first dimension one position to the left side. Actually , there's a large amount of applications which contains this kind of computing, such as in Jacobi iterations. If we use shift communication to

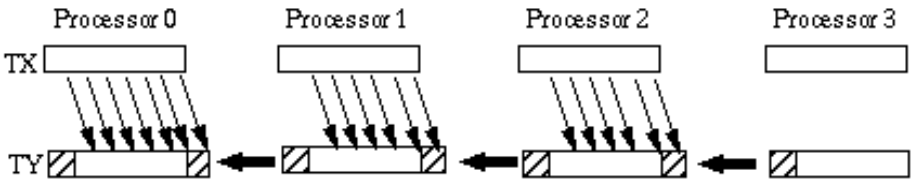
deal with it, we have to shift the whole Y array for one or two or only a few positions. But actually, we only need to move these small amount of data from one processor to another processor. So if we add a ghost area in the DAD structure to describe this shift area, we don't need to shift the whole Y array.

When implementing this kind of shift communication, most of the corresponding Y array elements are on the same processor as X array elements. So we only need to shift the small amount of corresponding elements in Y array from one processor to another processor in Y array's ghost area, instead of shifting other elements in Y array. In this way, we don't need to allocate a new temporary array buffer to contain Y array elements after shift communication (memory saved), and this will also save the operation of memory copy. This can greatly improve the performance of shift communication, especially when Y array is rather large and the shift amount is small. For example:

If the corresponding elements in the FORALL statement above is like:



Then we just need to shift two elements in template Y from right side processor to the left side processor to the ghost area of Y array, instead of shifting the whole Y array elements. It will be like this:



In this way, we greatly reduce this memory copy operations in shift communication. Meanwhile, because the shift elements will be stored in the ghost area of Y array, not in a new temporary buffer, it's easy for the compiler to locate the elements in Y array and to generate codes to implement this assignment.

The ghost area in the DAD structure can be used to improve the efficiency of shift communication. But there still exists a problem, how to determine the amount of ghost area? In some cases, if the shift amount is rather small, but if their positions are rather apart from the elements in this processor, then it's not good to use ghost area to implement this kind of shift communication. Otherwise, we should allocate a

large amount a buffer in the Y array ghost area to store a small number of elements. Fortunately, we can detect this in the compiling time. The compiler will determine if it is worthy to use ghost area and how large buffer to allocate for that. If the shift amount is too large, then we will just use the standard shift communi cation function to implement this, instead of using ghost area.

We have discussed the two dimensional forall statement in vary kinds of cases in the above. The strategies discussed can also be expanded to deal with the multi dimensioal case.

A A study of strided-subintervals in divided integral intervals

The terms:

Integral interval is a sequence of consecutive integer points on the axis, represented as $I = [i : e]$, or $I(i : e)$, where i is the initial point (lower bound) and e end point (upper bound). $h_I = e - i + 1$ is called the length of the integral interval.

When no confussion results, we use 'interval' for short.

Divided integral interval is an integral interval I with a positive integer p, denoted as (I, p).

We call $w = \lceil \frac{h_I}{p} \rceil$ the block size of the divided interval (I, p), and $t = p \cdot \lceil \frac{h_I}{p} \rceil$ the length of the divided integral interval. Thus, we can view an interval is divided into p blocks.

Strided-subinterval is a sequence of equally spanned points in an interval $I(i : e)$, denoted as $I(l : u : s)$, where l is the initial point (lower bound), u end point (upper bound), and s stride (span).

Here l, u, s are integers. $i \leq l \leq u \leq e,\ 1 \leq s$.

$\lfloor \frac{u-l}{s} \rfloor + 1$ is called the size of the strided-subinterval.

The maximum number of points of a strided-subinterval in a block Write $w = q \cdot s + r$, where $0 \leq r < s$. The number of points from $I(l : u : s)$ that a block can contain is $q + \lceil \frac{r}{s} \rceil$.

The relative position i of the first point from $I(l : u : s)$ in the block determines the number of points from $I(l : u : s)$ that are contained in the block. In particular, the block contains the maximum number of points if and only if $1 \leq i \leq r$ for $r \neq 0$ or $1 \leq i \leq s$ for $r = 0$.

The relative position of the first point from $I(l : u : s)$ in the next block. It is useful to know this in order to determine how many points are contained in each block.

We start from $I(l)$ being at position i. Consider the 'next' first point. We want to find the k, such that

$$i + (k - 1) \cdot s \leq w < i + k \cdot s \tag{18}$$

namely, $k = \lfloor \frac{w-i}{s} \rfloor + 1$. Or the current block contains k points from the strided-subinterval.

Thus, the relative position of the next first point is at $i + k \cdot s - w = i + s - w + \lfloor \frac{w-i}{s} \rfloor \cdot s$.

Some specially interesting cases.

- $r = 0, i \leq s$.

  $i + s - w + \lfloor \frac{w-i}{s} \rfloor \cdot s = i + s - w + (\frac{w}{s} - 1) \cdot s = i$

  This is what we expect. In general, when $r = 0$, let $j = mod(i, s)$. If $j = 0$, the next first position is s, otherwise j.

- $r \neq 0, i \leq r$.

  $i + s - w + \lfloor \frac{w-i}{s} \rfloor \cdot s = i + s - w + q \cdot s = i + s - r$

- $r \neq 0, i > r$.

  Write $j = mod(i - r, s)$, or $(i - r) = t \cdot s + j$, we will have the next first position being s for $j = 0$, j for $j \neq 0$. This is because

$$\begin{array}{rl} & i + s - w + \lfloor \frac{w-i}{s} \rfloor \cdot s \\ = & i + s - w + \lfloor \frac{q \cdot s + r - i}{s} \rfloor \cdot s \\ = & i + s - w + \lfloor q - \frac{i-r}{s} \rfloor \cdot s \\ = & i + s - w + \lfloor q - t - \frac{j}{s} \rfloor \cdot s \\ = & i + s - w + (q - t) \cdot s - \lceil \frac{j}{s} \rceil \cdot s \\ = & s + j - \lceil \frac{j}{s} \rceil \cdot s \end{array} \tag{19}$$

  (note: $w = q \cdot s + r$, and $(i - r) = t \cdot s + j$.)

If we interpret $x = mod(a, s)$ as the minimum non-negative solution to $x \equiv a \pmod{s}$, thus allow a being negative, we can unify the above 3 points as: for given position i in a block, the position of the first point in the next block

$$j = \begin{cases} mod(i - r, s), & (\text{mod } i - r, s) \neq 0 \\ s, & (\text{mod } i - r, s) = 0 \end{cases} \tag{20}$$

(We note, $(\text{mod } i - r, s) = s + i - r$ if $i - r < 0$ and $\mid i - r \mid < s$.)

A useful equality. For integers $a \geq 0,\ b > 0$, we have

$$\lceil \frac{a+1}{b} \rceil = \lfloor \frac{a}{b} \rfloor + 1 \tag{21}$$

since writing $a = q \cdot b + r, 0 \leq r < b$, we have $\lceil \frac{a+1}{b} \rceil = \lceil q \frac{r+1}{b} \rceil = q + 1$, $\lfloor \frac{a}{b} \rfloor + 1 = \lfloor q + \frac{r}{b} \rfloor + 1 = q + 1$.

The number of points of a strided-subinterval in a block. Given starting position i in a block, there will be $\lfloor \frac{w-i}{s} \rfloor + 1$ points contained in the block.

From the above, we see that we have reached an effective algorithm to determine the number of points of $I(l : u : s)$ in each block of $(I(i : e), p)$.

Two strided-subintervals. We consider $(I_1(i_1 : e_1), p_1)$,$I_1(l_1 : u_1 : s_1)$ and $(I_2(i_2 : e_2), p_2)$,$I_2(l_2 : u_2 : s_2)$. We are particularly interested the condition that the corresponding blocks of $(I_1(i_1 : e_1), p_1)$ and $(I_2(i_2 : e_2), p_2)$ contains the same number of points of $I_1(l_1 : u_1 : s_1)$ and $I_2(l_2 : u_2 : s_2)$, respectively.

First, some examples.

$s_1 = 3, w_1 = 5, r_1 = 2$ and $s_1 = 12, w_1 = 20, r_1 = 8$.

If $i_1 = 1$, we have

the first positions: 1, 2, 3, 1, ...

the number of points: 2, 2, 1, 2, ...

If $i_2 = 4$, we have

the first positions: 4, 8, 12, 4, ...

the number of points: 2, 2, 1, 2, ...

We see the two strided-subintervals are 'isomorphically' distributed in their underline intervals, respectively. If $i_2 = 8$, we have

the first positions: 8, 12, 4, 8, ...

the number of points: 2, 1, 2, 2, ...

We see it is not 'isomorphically' distributed with respect to I_1, even though the two first blocks contain the same number of points.

Shift isomorphism between two strided-subintervals. We say $I_1(l_1 : u_1 : s_1)$ and $I_2(l_2 : u_2 : s_2)$ are shift isomorphic if

$$\lfloor \frac{u_1 - l_1}{s_1} \rfloor = \lfloor \frac{u_2 - l_2}{s_2} \rfloor, \quad t_1 = t_2, \quad and \quad s_1 = s2 \tag{22}$$

Shift homomorphism between two strided-subintervals. Consider $p = p_1 = p_2$ only. We say $I_1(l_1 : u_1 : s_1)$ and $I_2(l_2 : u_2 : s_2)$ are shift homomorphic if

$$\lfloor \frac{u_1 - l_1}{s_1} \rfloor = \lfloor \frac{u_2 - l_2}{s_2} \rfloor, \quad and \quad \frac{t_1}{s_1} = \frac{t_2}{s_2} \tag{23}$$

Some simple properties. Write,

$$\begin{array}{rcl} t_1 & = & q_1 \cdot s_1 + r_1, \quad 0 \leq r_1 < s_1 \\ t_2 & = & q_2 \cdot s_2 + r_2, \quad 0 \leq r_2 < s_2 \end{array} \tag{24}$$

We have,

$$q_1 = q_2, \quad and \frac{t_1}{t_2} = \frac{s_1}{s_2} = \frac{r_1}{r_2} \tag{25}$$

To see this, we divide the equations in 24 by s_1 and s_2, respectively, and equate the right hand sides (left hand sides are equal by definition.) – integer parts and fraction parts must be equal, respectively.

Clearly, isomorphism defined here is a special case of homomorphism.

B Algorithm of PCRC Detect-Communication

HPF FORALL statement:

$$\text{FORALL (i=l:u:s)} \quad X(a_0 \times i + b_0) = Y(a_1 \times i + b_1)$$

. In case of BLOCK distribution mode

0. $s_x = a_0 * a_x * s \qquad s_y = a_1 * a_y * s$

1. determine if the assignment is shift-homomorphic:

   if $\dfrac{a_x \times a_0}{a_y \times a_1} = \dfrac{\overline{tx}}{\overline{ty}}$ then it's shift-homomorphic

   else it needs remap;

2. Compute

   $(l_tlb_x(i), l_tub_x(i)) = (l_{tx} + (i-1) * w_x, l_{tx} + i * w_x - 1)$

   i = 1, 2, ... , p, (Processor number)

   $w_x = \overline{tx} / p, \qquad \overline{tx} = \left\lceil \dfrac{u_{tx} + l_{tx} - 1}{p} \right\rceil \times p$

   $(l_tlb_y(i), l_tub_y(i)) = (l_{ty} + (i-1) * w_y, l_{ty} + i * w_y - 1)$

   i = 1, 2, ... , p, (Processor number)

   $w_y = \overline{ty} / p, \qquad \overline{ty} = \left\lceil \dfrac{u_{ty} + l_{ty} - 1}{p} \right\rceil \times p$

3. . compute the first effective element position in X array:

   $k_x = a_x * (a_0 * 1 + b_0) + b_x$

   for each processor, initialize $lk_x(i) = l_{tx} - 1$ i = 1, 2, ... , p

   use:

   $$lk_x(i) = k_x + \left\lceil \frac{l_tlb_x(i) - k_x}{s_x} \right\rceil \times s_x$$

   to update proper elements of $lk_x(i)$.

   . compute the last effective element position in X array:

   if $(l_tub_x(i) < a_x * (a_0 * u + b_0) + b_x)$

$$uk_x(i) = k_x + \left\lfloor \frac{l_tub_x(i) - k_x}{s_x} \right\rfloor \times s_x$$

else $uk_x(i) = a_x * (a_0 * u + b_0) + b_x)$

. compute the number of related elements in each processor

initialize $e_x(i) = 0, \quad i = 1, 2, \ldots, p$

use $e_x(i) = \left\lfloor \frac{uk_x(i) - lk_x(i)}{s_x} \right\rfloor$ to update related elements of e_x

4. do similarly as step 2 to get the value of

k_y, $lk_y(i)$, $uk_y(i)$, $e_y(i)$

5. if for each $e_x(i) == e_y(i)$, then no_communication needed

then calculate the value of u and v:

$$u = \frac{s_y}{s_x}, \qquad v = lk_y(i) - l_tlb_y(i) - u * (lk_x(i) - l_tlb_x(i))$$

else it needs shift-communication

6. compute the amount of shift-communication:

$$k_y' = \left\lceil k_x \times \frac{\overline{ty}}{tx} \right\rceil$$

amount = $k_y - k_y'$

7. compute the value of u and v of each processor:

if ($l_tlbx(i) <= kx <= l_tubx(i)$ kx_each = kx;

else $kx_each = kx + \left\lceil \frac{l_tlb_x(i) - k_x}{s_x} \right\rceil \times s_x$

$$ky_each = \left\lceil kx_each \times \frac{\overline{ty}}{tx} \right\rceil$$

$$u = \frac{s_y}{s_x}, \qquad v = (ky_each - l_tlb_y(i)) - (kx_each - l_tlb_x(i)) * u$$

. In case of CYCLIC distribution mode:

0. if (s_x mod p == s_y mod p) then it's shift-homomorphic

else it needs remap;

1. compute the position of first effective element in X array and Y array:
   $k_x = a_x * (a_0 * l + b_0) + b_x$
   $k_y = a_y * (a_1 * l + b_1) + b_y$

2. compute which processor the first element is in:
   $p_x = (k_x - l_{tx}) \bmod p$
   $p_y = (k_y - l_{ty}) \bmod p$

3. if ($p_x == p_y$), no communication needed

4. compute the amount of shift-communication:
   if ($p_x > p_y$ && (k_y / p) > (k_x / p))
   amount = $p_y + (p - p_x)$;
   else amount = $p_y - p_x$;

5. calculate the value of u and v:

$$u = \frac{s_y}{s_x}$$

$$v = \frac{a_y \times \left(\frac{a \times (localCoord - b_x + l_{tx})}{a_x} + b \right) + b_y - l_{ty} - localCoord - amount}{p}$$

C PCRC Coef Function

HPF FORALL statement:

FORALL (i=l:u:s) $X(a_0 \times i + b_0) = Y(a_1 \times i + b_1)$

```
void pcrc_coef_(int * dad, int *dim, int *l, int *u, int *s, int *a, int *b,
                                        int *m, int *c, int *amount)
```

Description: This function is aim to calculate the value of *u* and *v.* The FORALL statement will be translated into the following F77 codes.

```
CALL pcrc_upperBound(pcrc_rng(dad_x, 1), lu)
CALL pcrc_coef(dad_x, 1, l, u, s, a0, b0, u0, v0, 0)
CALL pcrc_coef(dad_y, 1, l, u, s, a1, b1, u1, v1, amount)
DO i = 0, lu
  X(u0 * i + v0) = Y(u1 * i + v1)        !  if in case of no-communication
  X(u0 * i + v0) = tmp(u1 * i + v1)      ! if in case of shift communication
END DO
```

. Algorithm

(1). Collapse Distribution

```
u = ax * (a0 * l + b) + bx
v = kx - ltx
```

(2). BLOCK Distribution

```
u = ax * (a0 * l + b) + bx
v = lkx - l_tlb - amount
// lkx is the position of first effective element in the array in this proces
   sor. (Refer to detect-communication algorithm)
```

(3). CYCLIC Distribution

```
sx = ax * a * s
u = sx / gcd(sx, p)      //   p is the processor number in this dimension
lkx = (kx + i * sx - ltx - amount) % p     34
//    calculate the relative position of the first effective element in the
      processor lkx;
v = lkx / p
```

References

[1] HPFF, High Performance Fortran Language Specification (version 1.0). May 3, 1993.

[2] C. Koelbel, D. Loveman, et al., The High Performance Fortran Handbook. The MIT Press, Cambridge, MA, 1994.

[3] Zeki Bozkus, Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers. Ph.D. Thesis, Syracuse University, August 1995.

[4] Z. Bozkus, A. Choudhary, G. Fox, et al, "Compiling Forall statement for distribution memory machines. SCCS report 389, NPAC, Syracuse University, December 10, 1992.

[5] J. Cowie, D. Leskiwd, X. Li, "The Distributed Array Descriptor for a PCRC HPF Compiler," Version 1.0, SCCS-7xx, NPAC at Syracuse University, February 5, 1996.

[6] Utpal Banerjee, Dependence Analysis for Supercomputing. Kluwer Academic Publishers, 1988, pp. 67-92.