Compiling regular FORALL statement based on data distribution directives

(Draft)

1.0 Introduction

Data distribution directives, such as ALIGN, DISTRIBUTE, proposed in High Performance Fortran [1] provided a convenient way to describe array location on a distributed memory machine. With the help of these directives, a *DAD* (Distributed Array Descriptor) [2] can be generated for each *regular reference* in FORALL statement and then used to translate this kind of FORALL statement in a global addressed program into Fortran DO construct with collective communication in a SPMD node program.

In this report, we introduce the major issues in translating the FORALL statement, using the actual program compiled in our framwork of a HPF compiler as examples.

1.1 Regular FORALL reference

Given a FORALL statement:

FORALL *forall-header forall-assignment-stmt*

The forall-header is of the following BNF form:

forall-header **is** *forall-triplet-sec-list* [, *scalar-mask-expr*]

while the *forall-assignment-stmt* is:

forall-assignment-stmt **is** *variable* = *expr*

The *variable* in above can also be called as *left-hand-side (LHS)* part of the assign statement, while the *expr* called *right-hand-side (RHS)* part. The expr will be further composed of possible multipule variables.

We call the varibles appeared in forall-triplet-sec-list as forall index. Usaually, these index will be used as the subscript of the array varibles accecssed in the forall assignment statement. We call the forall reference of the varibles as a regular reference if all of the forall index used in a linear expression of the varible address, and there is no array inderict addressing.

1.2 DAD

Distributed array descriptor

1.3 Runtime library

PCRC interface.

2.0 Program analyzing

General speaking, the compiler can be divided into two phases: program analyzing phase and program transforming phase. In program analyzing phase, the following items are examined to prepare basic information for the next phase:

- processor information, including rank and size in each rank;
- template inforamation, also including rank and size in each rank;
- distributing information for each template;
- aligning information for each distributed array;
- reference in each FORALL statement;

The first four items are reflected in PROCESSOR, TEMPLATE, DISTRIBUTE and ALIGN statement seperately. And their corrsponding translation in node program are strightforward, mainly for generating DAD for each array declaration in the source program, which can be found in [3], and later in this report as examples.

The fivth item is the focus in this section. And we only concerned regular forall reference in this report.

2.1 Communication patterns

Given a Forall statement as in

FORALL forall-triplet-sec-list [, *scalar-mask-expr*]   *variable* = *expr*

and translating it into Fortran DO construct to be executed on a sequenscal machine, the “owner computes” rule is used to assign the computation to each node processor, which means the processor who “own” the LHS will execute the actual caculation for that part. We may also let the “owner” of the LHS to evaluate the optional scalar-mask-expr. So the node program need to make the needed data, in both RHS part and in scalar-mask-expr, ready by communication if necessary before the computation is carried on.

If both LHS and RHS are on the same processor, there is no commnication needed, the computation can be carried on directly. Unfortranately, this is not always the case. When the corresponding data in the LHS and RHS indexed by the forall index is located in different machines, communication is necessary.

We can specify the communication pattens used in the FORALL construct according to the forall index reference in the LHS and RHS.

An regular forall reference may be a scalar or any array element, so we may have the following possible combinations:

The LHS variable is a scalar:

This does not comply the constrains of the forall statement, multiple assignment may happen. (If all the RHS are also scalar, it is just the same as one single assignment.)

If the LHS varibale is an array element:

In this case, there still two possibilities: it is a partitioned array or a un-partitioned array. If the array is not partitioned among the processors, the communication patterns needed are trival, it may either be no communication or sending a single copy to multipule processors.

For example:

```
FORALL (i=1:n)          A(i)=B(i)
```

While A is a non-partitioned array and B is a partitioned array.

If the array is a partitioned one. Then the communication needed is dicided by the reference pattern of forall index, including:

- No communication

  The LHS and RHS array elements needed to do caculation are located on the same machine.

- Shift

  The data movement is accurred along one or more dimention.

- Remap

  The data movement is among different dimention of the array.

- Scatter

  Data is duplicated to different position.

The simplest case of above movement can be show in figure. 1

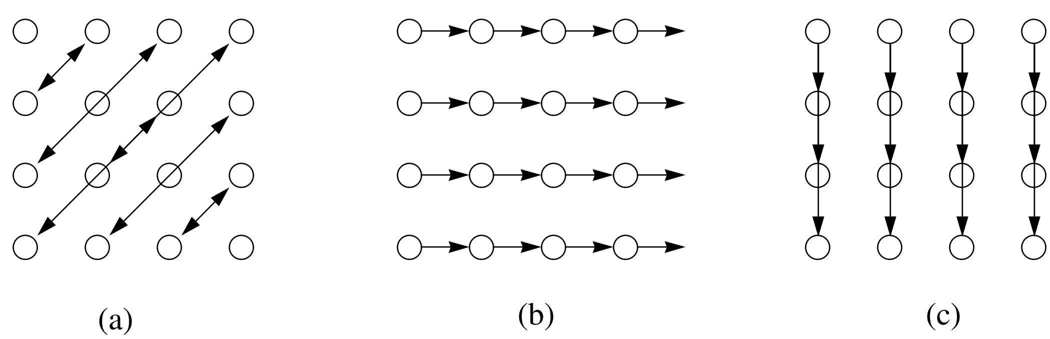

FIGURE 1. Data movement

During the analysing phase, the compiler will try to ditermine which kind of communication is used in the forall statement, using the detect communication method introduced in the next section.

2.2 Detecting shift or remap

decide the communication pattern

reduce the dimention, detecte communication.

So a simple algorithm for program analysis can be written as:

for each processor statement record processor rank and size

for each template statement record template dimention and size

for each align statement record arrary to be distributed.

for each forall statement do {

recursively analysis each reference,

detecting communication pattern needed

build up the transformation reference queue

}

3.0 Program transformation

Program transformation phase can be done through traveling along the parse tree again. During this course, different kind of operations are carried on the intermediate representation of the source program. In the end, unparse the new intermediate representation, the node program will be generated.

3.1 Memory managment

Before we discuss actrual translation in detail, we need to introduce the memroy strategy we intend to use in the node program.

3.1.1 Temporary memory

At present time, PCRC runtime library is based on standard Fortran 77 interface, no dynimac allocation of memory is allowed, instead a huge memory chuck is set up with common data block as stack for memory operations.

We only use the stack for tempory memory needed when communication is involved, while statically allocate partitioned array on each node processor. The basic idea can be illustrated through the following example:

```
A=B+C
```

If remap communication is necessary in this assignment, then tempory memory will be allocated for array B and C, as tempB and tempC from the memory stack. The corresponding array element will be aligned and distributed the same way as the element in the array A. Runtime function will be called to map B in tempB and C in tempC, then the computation can be carried out without communication. Then the temporary memory are returned to the stack.

The actual size of the temporary memory located is decided by the element size and template on which array A is aligned to. It can be caculated from runtime functions according to the DAD.

3.1.2 Memory access linerization

For unified access of the memory, multi-dimentional array is changed to one dimention. So explicit address caculation will be used in the node program. This simplify the translation of interface between procedure calls, all information needed is an array address, which is array name in Fortran language, and the DAD handle.

3.1.3 Ghost erea

There two ways can be used to prepare memory in node program for the required data movement. One as dynamic, another as static, which are supported by allocate-memory function in runtime libaray, and ghost-area in DAD respectively.

3.2 Translating different language components

There are following parts need to be taken care of during the transformation.

- DAD generation
- Expressions and assignment
- Program execution control
- Input and output
- Pocedures

These parts are corresponding to HPF languge data mapping derictives, Fortran 95 language sepcification chapter 7, 8, 9 and 12 respectively.

From the implementation point of view, most of the transformation neede to deal with each part is subdivided as two phases: format transformation and node program generation.

The first phase try to uniform the format of the actual program, making it suitable for further processing while keeping program's semantics meaning.

For example:

```
      A=B
```

can be changed into

```
      FORALL (tempI=1:n)       A(tempI)=B(tempI)
```

So the array assignment can be dealed uninamously in the second phase as forall statement.

Similiar transformation including where construct, (using forall mask expression), array object with simplified suscript in data mapping derictives and input/output statement.

Since this kind of change keeps the semantics of the original program, it is possible to further sudivided as different phases with each of them takes care of a particular issue. This method helps us seperate the transformation program as different modules, implemented and tested independently.

The second phase carries out the actual transformation work and generating the node program. We use examples to explain the actual translations for different parts.

3.2.1 DAD gemeration

Given the follow HPF program

```
      PROGRAM MAIN
      REAL X(1:221), Y(-2:701)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE TX(1:532),TY(-8:2116)
!HPF$ DISTRIBUTE TX(CYCLIC) ONTO P
!HPF$ DISTRIBUTE TY(BLOCK) ONTO P
!HPF$ ALIGN X(i) WITH TX(2*i+7)
!HPF$ ALIGN Y(i) WITH TY(3*i+5)
```

will be changed to:

```
  pcrc_shp_P(1) = 4
  pcrc_grp_P = pcrc_new_group_grid (1,pcrc_shp_P)
  pcrc_rng_TY(1) = pcrc_new_range_distribute ((-8),2116,1,pcrc_grp_P&
&,1)
  pcrc_rng_TX(1) = pcrc_new_range_distribute (1,532,2,pcrc_grp_P,1)
  pcrc_dad_Y = pcrc_new_array_data (Y,pcrc_real,pcrc_size_real,1,pcr&
&c_grp_P)
  call pcrc_set_array_align (pcrc_dad_Y,1,(-2),701,3,5,pcrc_rng_TY(1&
&))
  pcrc_dad_X = pcrc_new_array_data (X,pcrc_real,pcrc_size_real,1,pcr&
&c_grp_P)
  call pcrc_set_array_align (pcrc_dad_X,1,1,221,2,7,pcrc_rng_TX(1))
```

We can summurize this part as following:

For each processoer array, a *grp* value will be created with corresponding processoer shape; For each template, a *rng* value will be created to record its distribution code, distribution stride and offset. For each partitioned array, a *dad* value will be created to record its shape and its alignment stide and offset.

Note pcrc_ is used as a prefix to indicate this is a PCRC run-time function.

3.2.2 Expressions and assignment

Since some fundamental work has already been done in the format transformation phase, the major subject in node program generation is to deal with forall statement and scalar assignment.

For a forall statement, we explain the translation in case of no communication, shift and remap seperately.

Give the following program header:

```
PROGRAM MAIN
      REAL X(-17:190), Y(-75:221)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE TX(-238:3373),TY(-227:3384)
!HPF$ DISTRIBUTE TX(BLOCK) ONTO P
!HPF$ DISTRIBUTE TY(BLOCK) ONTO P
!HPF$ ALIGN X(i) WITH TX(3*i+7)
!HPF$ ALIGN Y(i) WITH TY(3*i+9)
```

A forall statament like:

```
  FORALL (i=-7:190:4) X(i) = Y(1*i+3) + Y(1*i+3)
```

will cause no communication, the corresponding node program will be:

```
  pcrc_irg0 = pcrc_new_range_loop ((-7),190,4,1,0,pcrc_rng (pcrc_dad&
&_X,1))
 call pcrc_loop_bounds (pcrc_irg0,pcrc_lil_i,pcrc_liu_i,pcrc_lis_i)
  call pcrc_coef (pcrc_dad_X,1,(-7),190,4,1,0,0,pcrc_u00,pcrc_v00)
  pcrc_sdd0 = pcrc_new_array_section (1,pcrc_dad_X)
  call pcrc_set_array_triplet (pcrc_sdd0,1,(-7),190,4,pcrc_dad_X,1)
  call pcrc_coef (pcrc_dad_Y,1,(-7),190,4,1,3,0,pcrc_u10,pcrc_v10)
```

```
call pcrc_coef (pcrc_dad_Y,1,(-7),190,4,1,3,0,pcrc_u20,pcrc_v20)
if (pcrc_on (pcrc_grp (pcrc_sdd0))) then
 do  i=0,(pcrc_liu_i-pcrc_lil_i)/pcrc_lis_i,1
      pcrc_sdx2 = (pcrc_v20+pcrc_u20*i)*pcrc_str (pcrc_dad_Y,1)
      pcrc_sdx1 = (pcrc_v10+pcrc_u10*i)*pcrc_str (pcrc_dad_Y,1)
      pcrc_sdx0 = (pcrc_v00+pcrc_u00*i)*pcrc_str (pcrc_dad_X,1)
      X(pcrc_sdx0) = Y(pcrc_sdx1)+Y(pcrc_sdx2)
 enddo
endif
call pcrc_delete_array (pcrc_sdd0)
call pcrc_delete_range (pcrc_irg0)
```

If we have the program header as:

```
PROGRAM MAIN
      REAL X(-98:197), Y(-47:390)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE TX(-199:473),TY(-95:786)
!HPF$ DISTRIBUTE TX(CYCLIC) ONTO P
!HPF$ DISTRIBUTE TY(CYCLIC) ONTO P
!HPF$ ALIGN X(i) WITH TX(2*i+5)
!HPF$ ALIGN Y(i) WITH TY(2*i+0)
```

A forall statement like:

```
FORALL (i=-8:190:3) X(i) = Y(1*i+4) + Y(2*i+1)
```

will cause both shift and remap communication, and its corresponding translation is:

```
  pcrc_irg0 = pcrc_new_range_loop ((-8),190,3,1,0,pcrc_rng (pcrc_dad&
&_X,1))
 call pcrc_loop_bounds (pcrc_irg0,pcrc_lil_i,pcrc_liu_i,pcrc_lis_i)

 call pcrc_coef (pcrc_dad_X,1,(-8),190,3,1,0,0,pcrc_u00,pcrc_v00)
 pcrc_sdd0 = pcrc_new_array_section (1,pcrc_dad_X)
 call pcrc_set_array_triplet (pcrc_sdd0,1,(-8),190,3,pcrc_dad_X,1)
 call pcrc_coef (pcrc_dad_Y,1,(-8),190,3,1,4,3,pcrc_u10,pcrc_v10)
 pcrc_sdd1 = pcrc_new_array_section (1,pcrc_dad_Y)
 call pcrc_set_array_triplet (pcrc_sdd1,1,(-4),194,3,pcrc_dad_Y,1)
 pcrc_tdd1 = pcrc_new_array_section (1,pcrc_dad_Y)
 call pcrc_set_array_triplet (pcrc_tdd1,1,(-4),194,3,pcrc_dad_Y,1)
 pcrc_tbs1 = pcrc_real_alloc (pcrc_size (pcrc_tdd1))
 call pcrc_reset_array_base (pcrc_tdd1,pcrc_real_stack(pcrc_tbs1))
 call pcrc_tshift (pcrc_real_stack(pcrc_tbs1),pcrc_sdd1,1,3)
 pcrc_sdd2 = pcrc_new_array_section (1,pcrc_dad_Y)
 call pcrc_set_array_triplet (pcrc_sdd2,1,(-15),381,6,pcrc_dad_Y,1)
 pcrc_tdd2 = pcrc_new_array_section (1,pcrc_dad_X)
 call pcrc_set_array_triplet (pcrc_tdd2,1,(-8),190,3,pcrc_dad_X,1)
 pcrc_tbs2 = pcrc_real_alloc (pcrc_size (pcrc_tdd2))
 call pcrc_reset_array_base (pcrc_tdd2,pcrc_real_stack(pcrc_tbs2))
 call pcrc_remap (pcrc_tdd2,pcrc_sdd2)
 if (pcrc_on (pcrc_grp (pcrc_sdd0))) then
```

```
    do  i=0,(pcrc_liu_i-pcrc_lil_i)/pcrc_lis_i,1
       pcrc_sdx1 = (pcrc_v10+pcrc_u10*i)*pcrc_str (pcrc_sdd1,1)
       pcrc_sdx0 = (pcrc_v00+pcrc_u00*i)*pcrc_str (pcrc_dad_X,1)
       X(pcrc_sdx0) = pcrc_real_stack(pcrc_tbs1+pcrc_sdx1)+pcrc_rea&
&l_stack(pcrc_tbs2+pcrc_sdx0)
    enddo
 endif
 call pcrc_real_free (pcrc_tbs2)
 call pcrc_delete_array (pcrc_tdd2)
 call pcrc_delete_array (pcrc_sdd2)
 call pcrc_real_free (pcrc_tbs1)
 call pcrc_delete_array (pcrc_tdd1)
 call pcrc_delete_array (pcrc_sdd1)
 call pcrc_delete_array (pcrc_sdd0)
 call pcrc_delete_range (pcrc_irg0)
```

For a scalor assigment, if we have

```
REAL sclr
```

in the program, a *dad* value will also be created for it, if necessary,

```
  pcrc_dad_sclr = pcrc_new_array_data (sclr,pcrc_real,pcrc_size_real&
&,0,pcrc_group_glob)
```

Then

```
sclr=X(3)
```

can be translated to:

```
pcrc_sdd1 = pcrc_new_array_section (0,pcrc_dad_X)
call pcrc_set_array_scalar (pcrc_sdd1,3,pcrc_dad_X,1)
pcrc_tdd1 = pcrc_new_array_copy (pcrc_real_stack,pcrc_dad_sclr)
pcrc_tbs1 = pcrc_real_alloc (pcrc_size (pcrc_tdd1))
call pcrc_reset_array_base (pcrc_tdd1,pcrc_real_stack(pcrc_tbs1))
call pcrc_remap (pcrc_tdd1,pcrc_sdd1)
sclr = pcrc_real_stack(pcrc_tbs1)
call pcrc_real_free (pcrc_tbs1)
call pcrc_delete_array (pcrc_tdd1)
call pcrc_delete_array (pcrc_sdd1)
```

3.2.3 Program execution control

In this part, the compling strategy for independent do is beyond the scope of this report, so there is nothing specil need to be done for the IF construct, CASE construct DO construct, and CONTINUE statement, STOP statement .

If a statement is labeled in source program, we can put a labeled continue statement before the corrsponding translation part in the node program, for example, if the in previous example the assignstatement is labeled as

```
100    sclr=X(3)
```

then we need add

```
100    continue
```

before the translated node program segment.

Therefor, the branching in execution control won't need any special translation either.

3.2.4 Input and output

The translation of input and output statement

```
       PRINT 111, (X(i),i=-98,197)
111    FORMAT(1x,f17.2)
```

The corresponding part will be:

```
       pcrc_sdd0 = pcrc_new_array_section (1,pcrc_dad_X)
       call pcrc_set_array_triplet (pcrc_sdd0,1,(-98),197,1,pcrc_dad_X,1)

       pcrc_tdd0 = pcrc_new_array_data (pcrc_real_stack,pcrc_real,pcrc_si&
      &ze_real,1,pcrc_group_root)
       call pcrc_set_array_collapse (pcrc_tdd0,1,0,295)
       pcrc_tbs0 = pcrc_real_alloc (pcrc_size (pcrc_tdd0))
       call pcrc_reset_array_base (pcrc_tdd0,pcrc_real_stack(pcrc_tbs0))
       call pcrc_remap (pcrc_tdd0,pcrc_sdd0)
       if (pcrc_on (pcrc_group_root)) then
         print 111, (pcrc_real_stack(pcrc_tbs0+i), i=0,295)
       endif
       call pcrc_real_free (pcrc_tbs0)
       call pcrc_delete_array (pcrc_tdd0)
       call pcrc_delete_array (pcrc_sdd0)
111     format (1X,F17.2)
```

3.2.5 Procduers

```
       SUBROUTINE FOO(X,Y)
       REAL X(:), Y(:)
!HPF$  INHERIT X,Y

       END

       subroutine FOO(X, pcrc_dad_X, Y, pcrc_dad_Y)
       integer :: pcrc_dad_X,pcrc_dad_Y
       real :: Y(0:*)
       real :: X(0:*)
       (other delaration ... )
       call pcrc_reset_push_bounds (pcrc_dad_Y,1,1,pcrc_alb_Y0,pcrc_aub_Y&
      &0,pcrc_ast_Y0)
      call pcrc_reset_push_bounds (pcrc_dad_X,1,1,pcrc_alb_X0,pcrc_aub_X&
```

```
&0,pcrc_ast_X0)
  (execution part ...)
  call pcrc_reset_pop_bounds (pcrc_dad_X,1,pcrc_alb_X0,pcrc_aub_X0,p&
&crc_ast_X0)
  call pcrc_reset_pop_bounds (pcrc_dad_Y,1,pcrc_alb_Y0,pcrc_aub_Y0,p&
&crc_ast_Y0)
  end subroutine
```

Add new components to the source program

The first step of the transformation is to add new components to the source program. These are mainly the declaration and definition of variables needed in the output node program.

After program header do if needed {

add declaration for each processor add declaration for each template (range)

add declaration for each distributed array (local array) (with accurate size)

add declaration for each temporary memory block (with accurate size and number)

add declarartion of handler name for each distributed array and temporary memory block

add declaration for temporary distributed array handler (for remap the distributed array)

add declaration for communication results (with accurate number)

add declaration for loop bounds add declaration for temporary loop index and factor

}

add call pcrc_init() before the first execution statement

add call pcrc_finalize() before stop (or end) statement

After pcrc_init() statement do {

add processor definition statements;

add template definition statements;

add distributed arrary definition statements;

}

function list

4.0 Conclusion

dad, runtime, memory.

5.0 Reference