PCRC-based HPF Compilation

Guansong Zhang, Bryan Carpenter, Geoffrey Fox
Xiaoming Li*, Xinying Li, Yuhong Wen

NPAC at Syracuse University
Syracuse, NY 13244
{*zgs,dbc,gcf,lxm,xli,wen*}*npac.syr.edu*

April 18, 1997

Abstract

This paper describes an ongoing effort supported by ARPA PCRC (Parallel Compiler Runtime Consortium) project. In particular, we introduce the design and implementation of an HPF compilation system based on PCRC runtime. The approaches to issues such as directive analysis, communication detection, and procedure interface processing are discussed in detail. Since this is already an operational compilation system, our discussion is centered around a complete node program generated by the compiler.

Key words: HPF, PCRC, compiler, runtime

Acknowledgement: Much of the work reported here is a result of collaboration among Syracuse University, Harbin Institute of Technology, China, and Peking University, China. The authors would like to acknowledge this collaboration and thank our friends in the other two groups.

1 Introduction

HPF has been around for a while [1]. While it seems not achieving its expectation, namely efficient and robust compilers arriving at the market within a few years after the specification's release, we do see PGI, IBM, and DEC have developed HPF compilers, and quite a few other companies, such as SUN, have HPF compiler groups.

On the other hand, HPF indeed provides an excellent context for parallel compilation system research and development, since it coherently embodies most of research issues and concepts developed in past years around parallel processing on distributed memory machines.

One of motivations behind HPF is that it is just too difficult to build a parallelizing compiler that produces efficient code from FORTRAN applications on parallel machines. Thus, HPF requests application programmer to help compiler.

We also believe that it is also quite difficult to build a *full-featured* and *high performance* compiler for such a complicated language as HPF, if the node program generated by the compiler directly calles `send()` and `receive()`. Thus, PCRC emphasizes the use of runtime [2], or in general library.

*Visiting scholar from HIT, China

As a result, node program will appear in a higher level, both easier to be generated by a compiler and understood by human. Of course, performance is always an issue, since this implies compiler gives up some opportunities in program global optimization. But it is unclear that how much performance gain can be actually obtained from those global optimizations.

As part of PCRC requirement, an HPF compiler is to be constructed based on this approach. Two natural aspects are to be investigated out of this research and development effort. One is to evaluate the effectiveness of this approach in parallel compiler construction; the other is to see the performance profile in comparison with more traditional approach.

At present, the system has been partially constructed. As it will be shown, this runtime-based approach allows us to attack full range of issues, which one must encounter in a real world compiler construction, from very beginning. For instance, the compilation of procedure calls is designed and implemented in the first stage, which is unusual in doing compiler in an academia environment.

This paper describes the design and implementation of our system. In particular, the approaches to issues such as directive analysis, communication detection, and procedure interface processing are discussed in detail. Section 2 provides an overview of the system architecture and some of global considerations. Section 3 describes some of the key technologies to some extent. Section 4 puts all things together, it illustrates how the compilation is done by a detailed presentation of a complete node program generated by our compiler. Section 5 is a survey of related work.

2 System overview

There are three major components in our system: a full featured HPF 1.0 front-end, `HPFfe` [3], a set of transformation modules, and the PCRC runtime [4].

2.1 `HPFfe`

`HPFfe` is a compiler front-end for High Performance Fortran Version 1.0. It's main thrust is its complete coverage of HPF 1.0 syntax and most of compile-time checkable semantics. As a result, Fortran 90 is fully covered. Besides syntax and semantics modules, a class library extended from Sage++ [6] is incorporated in the front-end, which allows us to write transformations effectively.

For a more detailed description of HPFfe, the reader is referred to [3] or Chapter 10 of [5].

2.2 Transformation modules

The compilation can be divided into two major phases: program analyzing phase and program transforming phase.

In the first part, compiler will try to use the available information to detect what kind of communication pattern is needed in the program.

The second part will carry out the actual transformation according to the record from the first phase to generate node program. Again it can subdivided as program format transformation and node program generation two parts.

We will introduce these modules more in this report later.

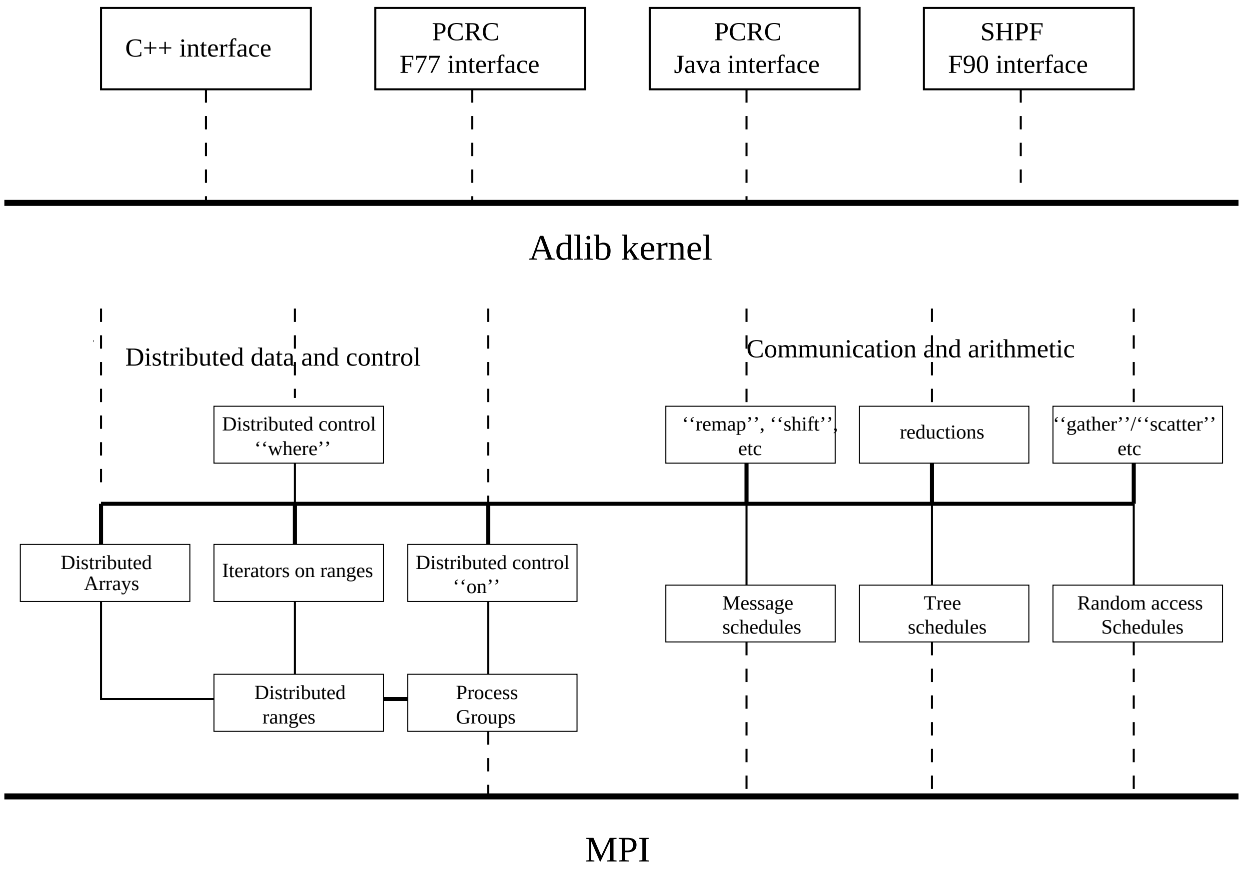

Figure 1: PCRC runtime architecture

2.3 PCRC runtime

The architecture of PCRC runtime is shown as in Figure 1. It basically consists of three groups of functions. One is distributed data management; the second is various data movement runtines; the third is computational functions corresponding to HPF intrinsic functions. It is implemented in C++, and also provides Fortran interface to the compiler. In section 4, the reader can see the flavor of the Fortran interface. For a complete description of PCRC runtime interface, the reader is referred to [4].

3 Key technologies

We describe two technologies employed in our compilation system, which are essential to both the compiler construction work and the performance of generated code. They are *distributed data descriptor* and *communication detection algorithm*. Other methods taken in handling various issues of the compilation will be illustrated in section 4 as we present a complete node program generated by the compiler.

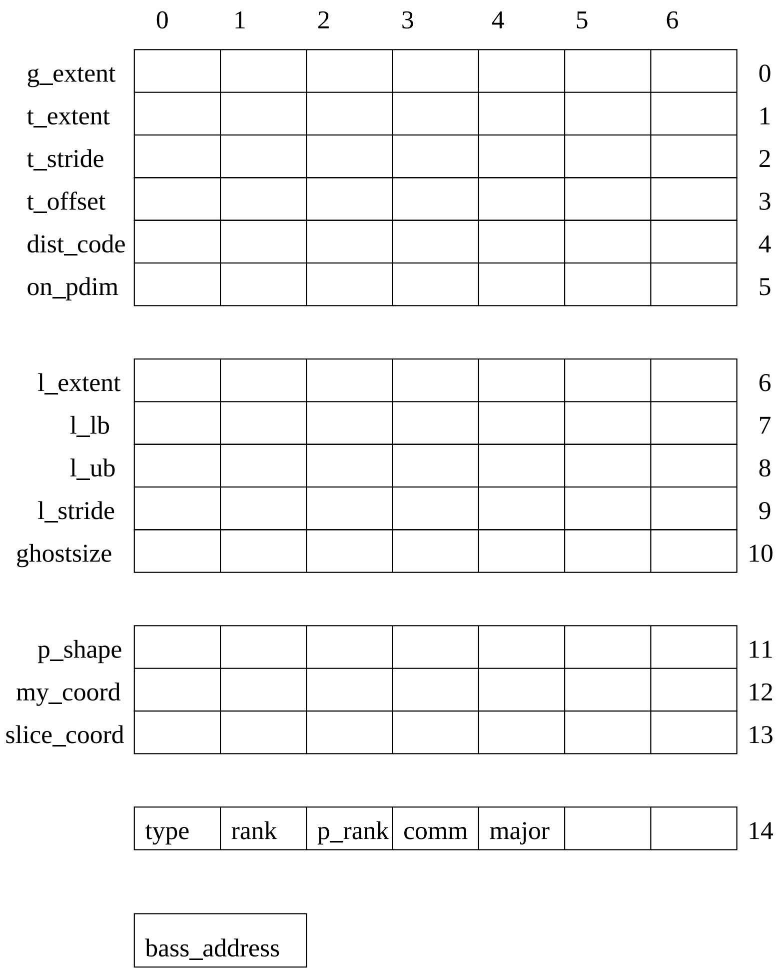

Figure 2: A practical distributed array descriptor

3.1 Distributed data descriptor

Explicit array data distribution is the core concept of HPF, which potentially frees compiler from the task of *data partitioning*. Data distribution directives, such as `ALIGN`, `DISTRIBUTE`, proposed in High Performance Fortran [1] provide a convenient way to describe how arrays in a global address (index) space are distributed among processors of a distributed memory machine. Whether there is an effective mechanism to tell the node program the data distribution is a key to effective compiler construction and runtime function implementation. We employ the notion of *distributed data descriptor* or `DAD` for this purpose. Similar mechanism is also used in other compilers (such as PGI compiler, shpf compiler, and previous NPAC F90D compiler), but actual designs may differ a lot. Our experience has shown that designing an effective DAD is non trivial, if you want it to support various data distributions (such as `block-cylic`, collapsed, replicated, etc.), and various dynamics of a distributed array during the course of a program execution (such as rank-reduced sectioning, passing to a subroutine, etc.), and if you also want runtime efficiency, etc.

Conceptually, PCRC runtime uses a DAD dipected as in Figure 2

Except `block-cyclic`, it efficiently supports all HPF 1.0 data distribution patterns, including the above mentioned dynamics, and efficient traversal of local array elements. In fact, the manipulation and management of DADs are major functions of PCRC runtime. For a thorough discussion of the DAD design, the reader is referred to [7] or Chapter 6 of [5].

3.2 Communication classification and detection

HPF directives release compiler from the task of data distribution, and owner computes rule (or other heuristics) more or less releases compiler from computation partitioning. Thus, essentially two pieces of work are left for compiler to do: communication detection and node program generation.

Taking the following array assignment as example,

```
      ...
      REAL X(16), Y(16)
      ...
      X(1:15) = Y(2:16)
```

Whether communication is needed depends on whether each pair of corresponding elements are in the same processor.

Because of the two level mappings (alignment and distribution) defined in HPF, the answer may not be readily obtainable. Our basic strategy is to classify communication requirement in an array assignment (the basis for every thing else) into three categories, namely, *no communication*, *shift communication*, and *remap communication*, and have developed a theory to detect them by the compiler. The first one is self evident. For instance, the following program requires no communication.

```
      REAL X(16), Y(16)
!HPF$ TEMPLATE T(48)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE T(BLOCK) ONTO P
!HPF$ ALIGN X(i) WITH T(i+1)
!HPF$ ALIGN Y(i) WITH T(i)
      ...
      X(1:15) = Y(2:16)
```

So is

```
      REAL X(16), Y(16)
!HPF$ TEMPLATE T(48)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE T(BLOCK) ONTO P
!HPF$ ALIGN X(i) WITH T(3*i-1)
!HPF$ ALIGN Y(i) WITH T(2*i+1)
      ...
      X(1:9:2) = Y(2:14:3)
```

Shift communication implies communication is needed, but a shift along array's template is adequate to make corresponding elements be in the same processor. For instance,

```
      REAL X(16), Y(16)
!HPF$ TEMPLATE T(48)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE T(BLOCK) ONTO P
!HPF$ ALIGN (i) WITH T(i) :: X,Y
       ...
      X(1:15) = Y(2:16)
```

and

```
       REAL X(16), Y(16)
!HPF$  TEMPLATE T(48)
!HPF$  PROCESSORS P(4)
!HPF$  DISTRIBUTE T(CYCLIC) ONTO P
!HPF$  ALIGN X(i) WITH T(3*i-1)
!HPF$  ALIGN Y(i) WITH T(2*i+1)
       ...
       X(1:9:2) = Y(2:14:3)
```

need only shift communication.

Of course, we also need to figure out the amount of shift (which is used to determine size of ghost area in the local array), once it is detected. And the amount also depends on local memory allocation model.

Remap communication is the 'final catch' kind of thing. It is the bag for all other situation. For instance,

```
       REAL X(16), Y(16)
!HPF$  TEMPLATE TX(36)
!HPF$  TEMPLATE TY(48)
!HPF$  PROCESSORS P(4)
!HPF$  DISTRIBUTE (BLOCK) ONTO P :: TX,TY
!HPF$  ALIGN X(i) WITH TX(i+1)
!HPF$  ALIGN Y(i) WITH TY(i)
       ...
       X(1:15) = Y(2:16)
```

Appropriate functions are designed in PCRC runtime to support the different situations. For instance, a `pcrc_edge2ghost()` function is designed to efficiently deal with *shift communications*, and a `pcrc_remap()` fucntion is designed to handle *remap communications*. For detailed derivations about our communication detection algorithm, the reader is referred to [8] or Chapter 8 of [5]. Section 4 will also show the specific application of the algorithm.

4 Putting pieces together

As mentioned above, transformation modules perform two basic functions, program analysis and transformation. In this section, we describe these modules through a concret node program generated by our compiler.

4.1 Program analysis

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 information, also including rank and size in each rank
- distribute information for each template
- align information for each distributed array
- variable reference in each forall statement
- array dummy in procedure argument

The first four items are reflected in PROCESSOR, TEMPLATE, DISTRIBUTE and ALIGN statement separately. And their corresponding translation in node program are straightforward, mainly for generating DAD for each array declaration, which can be seen later in this report as examples.

The last two items are the focus in this section. In this report, we only concerned with regular array reference, and here we use reference in forall statement as example.

Translating forall statement into Fortran DO construct to be executed on a sequential machine, the "owner computes" rule is used to assign the computation to each node processor, which means the processor who "owns" the LHS will execute the actual calculation for that part. We may also let the "owner" of the LHS to evaluate the optional *scalar-mask-expr*. Then a forall statement can be viewed as

FORALL *forall-triplet-sec-list* LHS=RHS

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

We can specify the communication patens used in the forall statement according to the forall index reference in the LHS and RHS.

A forall reference may be either a scalar or an 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.)

The LHS variable is an array element.

In this case, there are still two possibilities: the LHS is a partitioned array or a un-partitioned array.

If the array is not partitioned among the processors, the communication patterns needed are simple, it may be either no communication or sending a single copy to multiple processors.

For example:

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

If A is a non-partitioned array and B is a partitioned array, then broadcasting is needed.

If the array is a partitioned one. Then the communication needed is dependent on the reference pattern of the forall index. Possible communications include:

- No communication: The LHS and RHS array elements needed to do calculation are located on the same machine.
- Shift: The data movement is along one or more dimensions.
- Remap: The data movement is among different dimension of the array or not unified in the same dimension.
- Scatter: Data is duplicated to different positions.

During the analyzing phase, the compiler will determine what kind of communication is used in the forall statement.

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

`foreach` processor statement record processor rank and size `foreach` template statement record template dimension and size `foreach` align statement record array to be distributed. `foreach` forall and procedure call statement `do` recursively analysis each array reference, detecting communication pattern needed build up the transformation reference queue

The detecting communication method is based on the discussion made in the previous section.

4.2 Program transformation

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

4.2.1 Memory management

Before discussing the actual translation for different language components in details, it is neces sary to introduce the memory strategy we intend to use in the node program.

Temporary memory At present time, PCRC runtime library is based on standard Fortran 77 interface. Dynamic alloca tion of memory is not allowed. For memory operation during the execution of the node program, a huge memory chuck is set with common data block as memory stack. We only use the stack for temporary memory allocation when communication is involved, while statically allocate parti tioned array on each node processor.

The basic idea of using temporary memory can be illustrated through the following example:

if in a program, we have

```
A=B+C
```

and remap communication is needed in the assignment, then the temporary memory will be allo cated for array B and C, as tempB and tempC from the memory stack mentioned above.

The temporary memory was looked as if their corresponding array element is aligned and distributed the same way as the element in the array A, and runtime function will be called to generate DAD for them.

Then a runtime function will be called to map B in tempB and C in tempC respectively, after which the computation can be carried out without communication.

Finally the temporary memory are returned to the stack.

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

Memory access linearization For unified access of the memory, during the transformation multi-dimension array is changed to one dimension array in the finial node program. Explicit address calculation will be added.

This simplifies the translation of interface between procedure calls. All information needed for an array dummy when calling a procedure is the array address, which is referenced as an array name in Fortran language, and the DAD handle.

Ghost area Usually, when communication is needed, a temporary memory is allocated. But as we noticed in the previous, if the communication is of shift pattern, using ghost area will be a more effective way.

In this way, when shift communication is needed, only array elements in the edge will be moved among processors. The amount of data to be sent and received will be much less than the one in remap communication. In addition there is no additional memory operation overhead, since the memory used is already allocated.

Currently, the runtime functions do not support this feature yet, a tshift is used instead, which also need memory allocation, but have fast execution compared with remap. We are working to improve this part.

4.2.2 Translating different language components

In HPF language, the following parts need to be taken care of during the transformation,

- DAD generation for distributed data
- Expressions and assignment
- Program execution control
- Input and output
- Procedures

These parts are corresponding to HPF data mapping directives, chapter 7, 8, 9 and chapter 12 in Fortran 95 language specification respectively.

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

In format transformation, the components of the actual source program is changed, making them suitable for being further processed while keeping the semantics meaning. The language features encountered in the second phase will be narrowed down.

For example:

```
A=B
```

can be changed into

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

therefor the array assignment can be treated unanimously as forall statement in the next phase.

Other transformation similar to this kind may include:

- where construct (using forall mask expression)
- array section in assignment and input/output
- simplified data mapping directives
- array element used as scalar variable

Since the change keeps the semantics of the original program unchanged, it is also possi ble to further divide the whole process as different small parts, with each of them takes care of a particular issue in format transformation. This method helps us separate the transformation pro gram as different modules, implemented and tested independently.

The program generation phase carries out the actual transformation work and generating the node program. In the following, we will use simple examples to illustrate the translations done for dif ferent language components. For simplification, the examples used only have one-dimension array. The DAD support array section, the translation scheme introduced here can be easily generalized to deal with the multi-dimension arrays. It has already been implemented in our frame work of the HPF compiler.

DAD generation Given the follow HPF program

```
      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)
```

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 ((-227),3384,1,pcrc_grp&
     &_P,1)
```

```
 pcrc_rng_TX(1) = pcrc_new_range_distribute ((-238),3373,1,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,(-75),221,3,9,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,(-17),190,3,7,pcrc_rng_TX(&
&1))
```

We can summarize this part as following:

For each processor array, a *grp* value will be created with corresponding processor 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 stride and offset, this is the DAD handle for this array.

Note the generated name used in this report mainly have three parts. The pcrc is used as a prefix to indicate this is a PCRC run-time function. Followed by the name of variables or functions, such as `grp`, `rng`, `dad` and `set_array_align`. Finally, if needed a tag is used to indicate where the name come from.

Expressions and assignment As we said previously, some fundamental work has already been done in the format transformation phase, the major subject in this part 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 with examples separately.

We still use the same program header in previous example, a forall statement like:

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

does not need communication, so 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)
```

In this part of the node program segment, the local loop bounds lil, liu, lis is calculated out first according to the forall index range and the DAD of the LHS. Run time function coef is called to get the address coefficient for each array dimension, which later are used to collate the local address in the array reference. To make sure the calculation is needed on the current processor, an if statement is inserted. This is optional in main program but necessary in any procedures, since the LHS may be a dummy whose actual argument is an array section.

And finally, the destructor of the temporary value is called.

The computation is carried by

```
      X(pcrc_sdx0) = Y(pcrc_sdx1)+Y(pcrc_sdx2)
```

And we can see there is no need to exchange data among the processors.

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)
```

and a forall statement like:

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

it will cause both shift and remap communication. The corresponding translation of the forall statement 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)
```

The node program segment bears some similarity with the former one, except temporary memory need to be allocated, the *tbs* value are used as the pointer to the temporary memory in the stack, and *tdd* is their corresponding DAD handler. Run time function "`alloc`" is called to obtain the memory, and was dellocated with runtime "`free`" function in a reverse order after the computation is done.

The computation is carried out by,

```
X(pcrc_sdx0) = pcrc_real_stack(pcrc_tbs1+pcrc_sdx1)+pcrc_rea&
&l_stack(pcrc_tbs2+pcrc_sdx0)
```

The data accessed in the stack is prepared by the runtime function tshift and remap according to corresponding to DAD handler.

Since the run time function did not support the ghost area yet, the shift communication also need temporary array allocation.

For a scalar assignment, if we have

```
REAL sclr
```

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

Then for an assignment like,

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

will 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)
```

The remap function is used here to broadcast the array element to temporary memory located on every processors. Then the assignment is from the temporary memory to the scalar.

Another situation need to take care of is when forall index used as a reference in the forall assign ment, local-to-global index transformation is needed.

Program execution control In this part, the compiling strategy for independent do is beyond the scope of this report, so there is nothing special 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 cor responding translation part in the node program, for example, if the in previous example the assign statement is labeled as

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

then we may add

```
100   continue
```

before the translated node program segment for the original assign statement. Therefor, the branching in execution control won't need any special translation either.

Input and output The translation for the input and output statement is quite similar to the one for forall statement. The implied do loop index is treated the same way as the forall index. For example,

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

will be translated as:

```
      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)
```

Unlike the forall statement, the temporary array is allocated as the array section to be printed is replicated on each processor. After the remap communication, each processor will acquire a copy of the array section, though only root processor will execute the following print statement.

Here remap function is used as a multi-gather operation, though only one copy of the data moved to the rood processor will be enough. A little overhead is involved, we don't think it will have much effect on the performance.

Procedures The procedure transformation of the program also has similarity with forall statement. We can think the dummy array as LHS and actual array as RHS. Detecting communication introduced pre viously can also used here. The DAD needed to do communication can be generated according to the procedure interface. So the caller can map the data needed by the callee.

```
      CALL FOO(X(-75:194:1),Y(-49:821:3))
```

will change to,

```
        pcrc_sdd1 = pcrc_new_array_section (1,pcrc_dad_X)
        call pcrc_set_array_triplet (pcrc_sdd1,1,(-75),194,1,pcrc_dad_X,1)
        pcrc_sdd2 = pcrc_new_array_section (1,pcrc_dad_Y)
        call pcrc_set_array_triplet (pcrc_sdd2,1,(-49),821,3,pcrc_dad_Y,1)
        call FOO (X(0),pcrc_sdd1,Y(0),pcrc_sdd2)
        call pcrc_delete_array (pcrc_sdd2)
        call pcrc_delete_array (pcrc_sdd1)
```

if there is no communication needed for a data mapping provided in proceduer interface. We can see that the DAD handle are also put into the actrul argument list.

Now we can see the callee part of the procedure. If a procedure defined as,

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

      (execution part ... )

      END
```

Then it will be translated as,

```
      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)

      (translation of 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
```

Dummy DADs are also added in argument list. And it is also tried to save and restore the accrual DAD when come in and out of the procedure, since some operations inside the procedure transfor mation may change the DAD value.

Aside translation for each language component, there some preparing work also need to be done:

- add declaration for each processor (grp)
- add declaration for each template (rng)
- add declaration for each distributed array (local array)
- add declaration for DAD handle
- add declaration for loop bounds add declaration for temporary loop index and factor
- add call pcrc_init() and call pcrc_finalize()
- add processor definition statements
- add template definition statements
- add distributed array definition statements

5 Related works

Automatic generating message passing programs from data distribution specifications has been explored for some time in the context of various data parallel languages [9], [10], [11], [12] and [13].

In [12], the support of the run-time functions are relatively weak, compiler need to generate send and receive primitives to accomplish communication. Though this may have more efficient code generation after extensive program analyzing, the compiler may become too complicated to be operational.

The most resent paper on HPF compiling was [14], in which a local set enumeration method was used to generate local part of a loop iteration and derive communication set.

Comparatively speaking, we believe our run time support method to get these value is more straightforward and efficient, especially for regular access of the array data. It is true that HPF language allow the processors number to be set at run-time, but when solving the real program this may not be an important factor. We observed the following comments from DEC HPF compiler

> The optional `num` parameter specifies the number of processors on which the program is intended to execute. If not specified, the program will be compiled to execute on any number of processors. More efficient code is generated when `num` is specified.

In fact, if memory allocation is allowed in the node program generated by our compiler, or we use our own memory stack to allocate partitioned data, same conclusion will be true to our compiler.

So far, we compared our work with the commercial compilers from IBM, DEC and PGI, only PGI HPF compiler support inherit directives. Further comparing on performance of the node program will be done as the project going on.

6 Conclusion and future work

The PCRC-based HPF compilation system described above has been partially implemented. From this experience, we see runtime based approach to compiler construction is a viable methodology in compiler research and development, as well as education. It always emphasizes the 'bigger' picture, without getting lost in fine points.

Emphasizing runtime in compilation system construction is essentially taking a divide-conque philosophy. It allows the complicated system to be cleanly divided into two big pieces. Different people can independently work on different pieces. And once some function is well understood by runtime, it may be inlined in the compiler generated code, or used directly by compiler to improve performance. Rich runtime essentially become a valuable infrastructure supporting different compiler constructions. This is the idea of PCRC.

As our future work, the first priority is to implement the support for ghost area. Then some performance data can be collected and compared with other compilers. Then things like support for irregular problems should follow. Of course, a big issue is that the data description for irregularly distributed data is often totally different from that for regular problems. How to effectively incorporate them into our current runtime is some thing for elaboration.

References

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

[2] PCRC, Common Runtime Support for High Performance Data Parallel Languages, Project proposal, May, 1994.

[3] Xiaoming Li, et al, "HPFfe: a Front-end for HPF," NPAC Technical Report, SCCS-771, May 1996, and
http://www.npac.syr.edu/projects/pcrc/hpffe.html.

[4] Bryan Carpenter, Geoffrey Fox, Xiaoming Li, et al, "PCRC runtime interface," NPAC Technical Report, SCCS-XXX, (to appear).

[5] Xiaoming Li, Runtime Oriented HPF Compilation. Technical Report, SCCS-773, NPAC at Syracuse University, 1997.1.

[6] D. Gannon, et al, "A Class library for Building Fortran 90 and C++ Restructuring Tools," Nov. 1993,
http://www.extreme.indiana.edu/sage/index.html.

[7] Bryan Carpenter, James Cowie, Don Leskiw, and Xiaoming Li, "The Distributed Array Descriptor for a PCRC HPF Compiler," Version 2.0, NPAC Technical Report, SCCS-770d, Jan., 1997.

[8] Xiaoming Li, Yuhong Wen, "Efficient Compilation of Forall Statement with Runtime Support," NPAC Technical Report, SCCS-XXX, October, 1996.

[9] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," J. Supercomputing, Vol. 2, pp. 151-169, Oct. 1988

[10] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference", Proc. ACM SIGPLAN Intl Conf. Program language Design and Implementation, pp69-80, June 1989

[11] C. Koelbel and P. Mehrotra, "Compiling Global Name-Space Parallel Loops for Distributed Execution", IEEE Trans. Parallel and Distributed Systems, vol. 2, pp. 440-451, Oct. 1991

[12] C.-W. Tseng, "An Optimizing Fortran D Compiler for MIMD Distributed-Memory machines", PhD thesis, Rice University Jan. 1993

[13] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi, "Compilation Techniques for Block-Cyclic Distributions", Proc. Intl. Conf. Supercomputing, pp.392-401, July 1994

[14] 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 Trans. on parallel and distributed system, vol.7 Sep. 1996