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*

June 25, 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 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 functions [2], or in general,

*Visiting scholar from HIT, China

library. As a result, node program will appear in a higher level, both easier to be generated by a compiler and understood by human. Since this implies that compiler gives up some opportunities in program global optimization, performance is an issue. But it is unclear how much performance gain can be actually obtained in the traditional approach.

As part of PCRC requirement, an HPF compiler is to be constructed based on this idea. Naturally, two 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 the more traditional one.

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 implementing 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 and compares the performance of our system with the comercial compiler.

2 System overview

The over all work is show in Figure 1. 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 transformation modules effectively.

2.2 Transformation modules

The compilation can be divided into two major phases: program analyzing phase and program transforming phase. In the first part, the compiler will try to use the available information to detect what kind of communication pattern is needed in the program. In the second part, the actual transformation will be carried out on the intermediate representation to generate node program.

We will introduce these modules more in this report later.

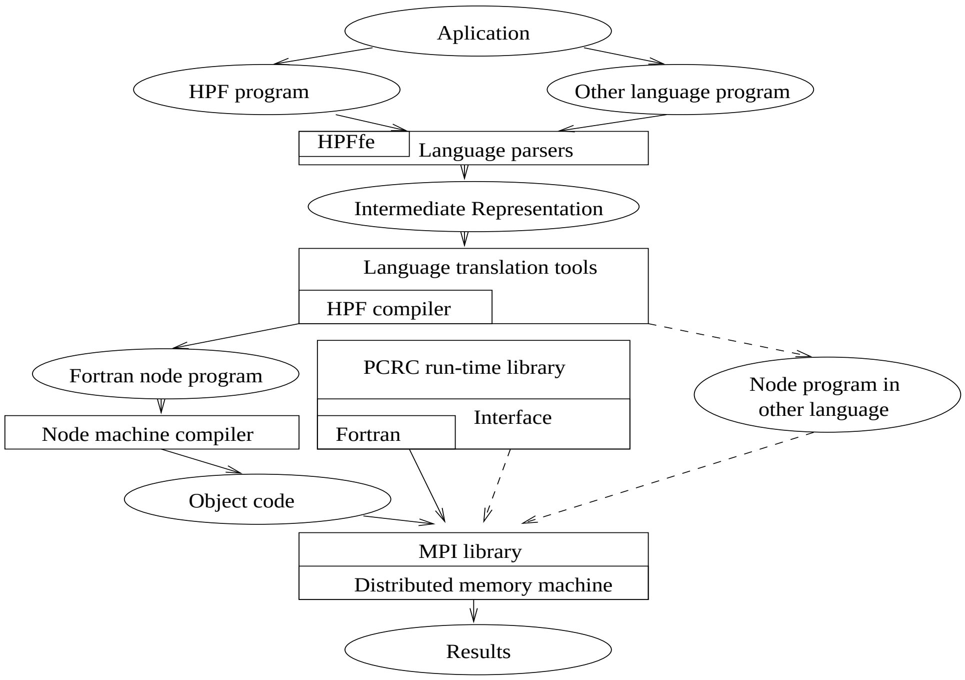

Figure 1: System overview

2.3 PCRC runtime

The PCRC runtime 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 Runtime-based compilation

Explicit array data distribution is the core concept of High Performance Fortran [1] , which frees compiler from the task of *data partitioning*, potentially *computation partitioning*.

Data mapping directives, such as `ALIGN`, `DISTRIBUTE`, provide a convenient way to describe how arrays in a global address (index) space are distributed among processors of a distributed memory machine. Given this mapping, each node processor may re-map data among one another through some high level collective communication routines. Compiling HPF program becomes generating seperate addressed node program based on the runtime library. This is so called runtime-based compilation.

Two questions imediately follow this approach: 1) What kind of run-time functions are to be supported and 2) how they are called from node programs.

The questions are answered by two key technologies employed in our compilation system: *distributed data descriptor* and *communication classification*, which will be discussed in this section. Other methods handling various issues of the compilation will be illustrated in section 4 as we

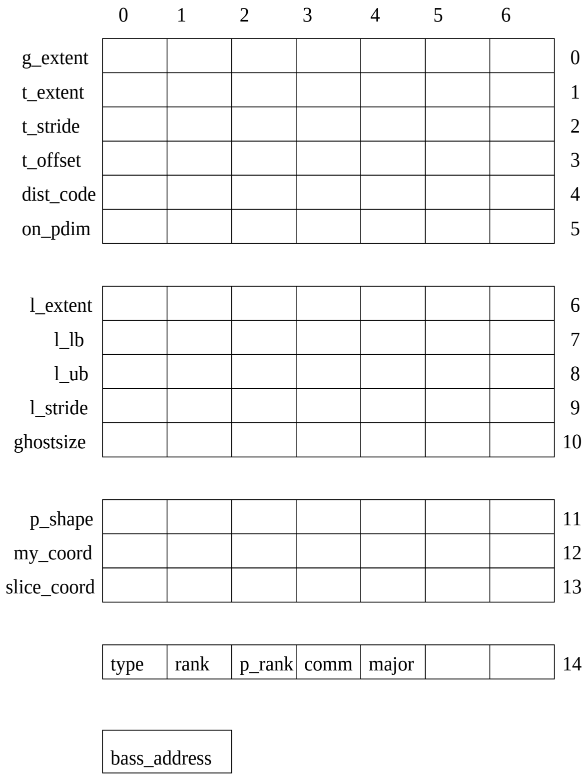

Figure 2: A practical distributed array descriptor

present a complete node program generated by the compiler.

3.1 Distributed data descriptor

An effective mechanism to tell the node processor the actual data mapping 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.), pluse 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. But the compiler must deside the execution control of the node program, in our runtime-based approach, what kind of communication is needed and when.

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.

We classified communication needed by an array assignment (the basis for every thing else) into three categories, namely, *no communication*, *shift communication*, and *remap communication*, and developed an effective algorithm to detect them.

3.2.1 Communication patterns

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_write_halo()` function is designed to efficiently deal with *shift communications*, and a `pcrc_remap()` fucntion is designed to handle *remap communications*.

3.2.2 Detecting communication

We will discuss the one dimenstion array.

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 concret node programs 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 HPF derictives: `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 as later examples in this report.

The last two items are the focus in this section. We only concerned with "*regular array reference*" (array index is a linear expression of FORALL index) here, and use FORALL statement as an example.

Using the "owner computes" rule to assign the computation to each node processor, we may also let the "owner" of the left-hand-side (*lhs*) to evaluate the optional *scalar-mask-expr*. Then a forall statement can be viewed as

`FORALL` *forall-triplet-sec-list lhs*`=` *rhs*

We can specify the communication paterns according to the FORALL index referenced in the *lhs* and *rhs*. In fact, the basic idea is the same as we discussed previously about the array assignment. The array assignment can be considered as a special FORALL statement, in which the FORALL index appeares in pair in *lhs* and *rhs*. In addition, FORALL index may have permutation (matrix transpose) and repeatation (matrix diagnose).

As the communication classification is based on each dimention, index permutation will not cause new problem. But the index repeatation need more support from runtime library.

So the possible communication routines needed in runtime include:

- No communication: The *lhs* and *rhs* array elements needed to do calculation are located on the same machine.
- Shift: The *rhs* data need to be moved in parallel a constant distance along one or more dimensions.
- Remap: The data movement is among different dimensions of the array, or the distance is not uniform along dimension.
- Scatter: For repeated FORALL index and more general cases .

During the analyzing phase, the compiler will determine what kind of communication is used in the FORALL statement, and if it is a shift communication, the accurate distance.

4.2 Program transformation

Program transformation phase is done through traveling along the parse tree again. In the end, unparse the modified intermediate representation to get the node program. Two aspects need to be considered: memory management and execution control of the node program.

4.2.1 Memory management

Memory management refers to memory handling needed in node program to complete communication, involving both static and dynamic memory control.

Temporary memory Temporary memory is dynamically allocated. At present time, PCRC runtime library is based on standard Fortran 77, dynamic allocation of memory is not allowed. For memory operation during the execution of the node program, a huge memory chuck is set with "data block" as memory stack, corresponding operations are provided. It is used for communication need new memory space.

The basic idea can be illustrated through the following example:

if in a program, we have the following assignment and need remap communication,

```
...
A=B+C
```

then the temporary memory will be allocated for array `B` and `C`, as `tempB` and `tempC` from the memory stack mentioned above. The actual size of the temporary memory allocated 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`.

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

Then a runtime function will move `B` to `tempB` and `C` to `tempC` respectively according to their DADs, after which the computation can be carried out without communication as:

```
...
A=tempB+tempC
```

Finally the temporary memory are returned to the stack.

For detailed operation, please refer to the examples later.

Ghost area Dynamically allocating memory can be used whenever temporary space is needed by the communication. But as we noticed in the previous, if the communication is of shift pattern, using ghost area will be a more efficient way.

In this case, when shift communication is needed, only array elements in the edge will be moved among processors. The rest of them remain "stable". The memory space (called "*ghost area*" or "*halo region*") for communication has already been allocated as part of the local array. Therefor the amount of data to be copied will be much less than using remap to complete the same communication. In addition, there is no additional overhead for memory operation.

Memory access In addition to memory allocation methods, the node program must deal with translation between global array subscripts and local subscripts. and linearize subscript computations for multi-dimentional arrays.

The translation functions are carefully designed, so they are never called from within the inner DO loops generated by translation of the FORALL statement. And the linearization of array segments, in conjunction with use of DAD inquiry functions provided in the runtime library, is important for implementing transcriptive features of HPF procedure, such as the `INHERIT` directive. Unnecessary copy-in and copy-out in caller or callee are generally avoided.

4.2.2 Translating different language components

We can summarize the language component in HPF as:

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

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

As we implement the compilation system, most of the transformation needed for each part can be subdivided as two phases: format transformation and node program generation.

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

For example, the array assigment will be changed to FORALL statement, so it can be treated unanimously as FORALL statement in the next phase.

Other transformation similar to this kind may include:

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

Since the change keeps the semantics of the original program unchanged, it is also possible 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 program as different modules, implemented and tested independently.

The program generation phase carries out the actual transformation work and generate node program. In the following, we will use simple examples to illustrate the translations done for different language components.

For simplification, the examples used here only have one-dimension array. 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.

DAD generation Given the follow HPF program

```
      REAL X(1:205), Y(-12:161)
!HPF$ PROCESSORS P(2)
!HPF$ TEMPLATE TX(-2:205),TY(-17:190)
!HPF$ DISTRIBUTE TX(BLOCK) ONTO P
!HPF$ DISTRIBUTE TY(BLOCK) ONTO P
!HPF$ ALIGN X(i) WITH TX(1*i+0)
!HPF$ ALIGN Y(i) WITH TY(1*i-14)
```

will be changed to:

```
      pcrc_shp_P(1) = 2
      pcrc_grp_P = pcrc_new_group_grid (1,pcrc_shp_P)
      pcrc_rng_TY(1) = pcrc_new_range_distribute ((-17),190,1,pcrc_grp_P&
     &,1)
      pcrc_rng_TX(1) = pcrc_new_range_distribute ((-2),205,1,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,(-12),161,1,(-14),0,0,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,1,1,205,1,0,0,0,pcrc_rng_TX(&
     &1))

      ...

      call pcrc_delete_array (pcrc_dad_X)
      call pcrc_delete_array (pcrc_dad_Y)
      call pcrc_delete_range (pcrc_rng_TX(1))
      call pcrc_delete_range (pcrc_rng_TY(1))
      call pcrc_delete_group (pcrc_grp_P)
```

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, it is the DAD handle for this array. And finally, at the end of the program, destructor will be called for these created objects.

Note the generated name used in the node program mainly have three parts. The `pcrc` is used as a prefix to indicate this is a PCRC run-time function or value. 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 comes 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=8:112:1) X(i) = Y(1*i-1)
```

does not need communication, so the corresponding node program will be:

```
  pcrc_irg0 = pcrc_new_range_loop (8,112,1,1,0,pcrc_range (pcrc_dad_&
 &X,1))
  call pcrc_loop_bounds (pcrc_irg0,pcrc_lil_i,pcrc_liu_i,pcrc_lis_i)

  pcrc_sdd0 = pcrc_new_array_section (1,pcrc_dad_X)
  call pcrc_set_array_triplet (pcrc_sdd0,1,8,112,1,pcrc_dad_X,1)
  call pcrc_coef (pcrc_dad_X,1,8,112,1,1,0,0,pcrc_u00,pcrc_v00)
  call pcrc_coef (pcrc_dad_Y,1,8,112,1,1,(-1),0,pcrc_u10,pcrc_v10)
  if (pcrc_on (pcrc_group (pcrc_sdd0))) then
     do  i=0,(pcrc_liu_i-pcrc_lil_i)/pcrc_lis_i,1
        pcrc_sdx1 = pcrc_v10+pcrc_u10*i
        pcrc_sdx0 = pcrc_v00+pcrc_u00*i
        X(pcrc_sdx0) = Y(pcrc_sdx1)
     enddo
  endif
  call pcrc_delete_array (pcrc_sdd0)
  call pcrc_delete_range (pcrc_irg0)
```

In this part, the local loop bounds and stride: `lil`, `liu`, `lis` is calculated out first according to the FORALL index range and the DAD of the *lhs*. Runtime function *coef* is called to get the address coefficient for each array dimension, which later are used to calculate the local address in the array reference. To make sure the calculation is needed on the current processor, an `if` statement is inserted.

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

The computation is carried by

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

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

If we change the alignment a little, as:

```
!HPF$ ALIGN Y(i) WITH TY(1*i-12)
```

a shift communication is needed. the local `Y` should be 2 emlement larger than the one in the previous case. The DAD generation for `Y` will also changed to:

```
  call pcrc_set_array_align (pcrc_dad_Y,1,(-12),161,1,(-12),0,2,pcrc&
 &_rng_TY(1))
```

The node program will do the following thing more before the computation:

```
pcrc_gtl_Y(1) = 0
pcrc_gtu_Y(1) = 2
call pcrc_write_halo (pcrc_dad_Y,pcrc_gtl_Y,pcrc_gtu_Y)
```

and caculate the address for the *RHS* array Y with:

```
call pcrc_coef (pcrc_dad_Y,1,8,112,1,1,(-1),2,pcrc_u10,pcrc_v10)
```

The pcrc_write_halo will send the edge data to the accurate position in the next processor. If we further change the distribution of the array,

```
!HPF$ DISTRIBUTE TY(CYCLIC) ONTO P
```

a remap is needed. The node program translation for the FORALL statement will be:

```
   pcrc_irg0 = pcrc_new_range_loop (8,112,1,1,0,pcrc_range (pcrc_dad_&
  &X,1))
   call pcrc_loop_bounds (pcrc_irg0,pcrc_lil_i,pcrc_liu_i,pcrc_lis_i)
   pcrc_sdd0 = pcrc_new_array_section (1,pcrc_dad_X)
   call pcrc_set_array_triplet (pcrc_sdd0,1,8,112,1,pcrc_dad_X,1)
   call pcrc_coef (pcrc_dad_X,1,8,112,1,1,0,0,pcrc_u00,pcrc_v00)
   pcrc_sdd1 = pcrc_new_array_section (1,pcrc_dad_Y)
   call pcrc_set_array_triplet (pcrc_sdd1,1,7,111,1,pcrc_dad_Y,1)
   pcrc_tdd1 = pcrc_new_array_section (1,pcrc_dad_X)
   call pcrc_set_array_triplet (pcrc_tdd1,1,8,112,1,pcrc_dad_X,1)
   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)
   if (pcrc_on (pcrc_group (pcrc_sdd0))) then
      do  i=0,(pcrc_liu_i-pcrc_lil_i)/pcrc_lis_i,1
         pcrc_sdx0 = pcrc_v00+pcrc_u00*i
         X(pcrc_sdx0) = pcrc_real_stack(pcrc_tbs1+pcrc_sdx0)
      enddo
   endif
   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)
```

This time, a temporary array is allocated from pcrc_stack, starting from pcrc_tbs1. pcrc_remap is used to copy original Y to the new position, so the computation can be carried on.

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)
```

it 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.

In addition, when FORALL index used as a *rhs* 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 almost nothing 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 corresponding translation part in the node program. Therefor, the branching in execution control won't need any special treatment 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 allocated as the array section to be printed is replicated on each processor. After the remap communication, each processor will have 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* reference and actual array as *rhs* reference. Detecting communication introduced previously 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 to the need of the callee.

If no communication is necessary, a `CALL` statement

```
      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)
```

We can see that the DAD handle are also put into the actrul argument list.

Now we 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 transformation may change the DAD value.

Aside translation for each language component, there some preparing work need to be done, including adding declarations for each processor (*grp*), template (*rng*), distributed array (local array), DAD handle, loop bounds, temporary loop index, address factors, and calling `pcrc_init()`, `pcrc_finalize()`.

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