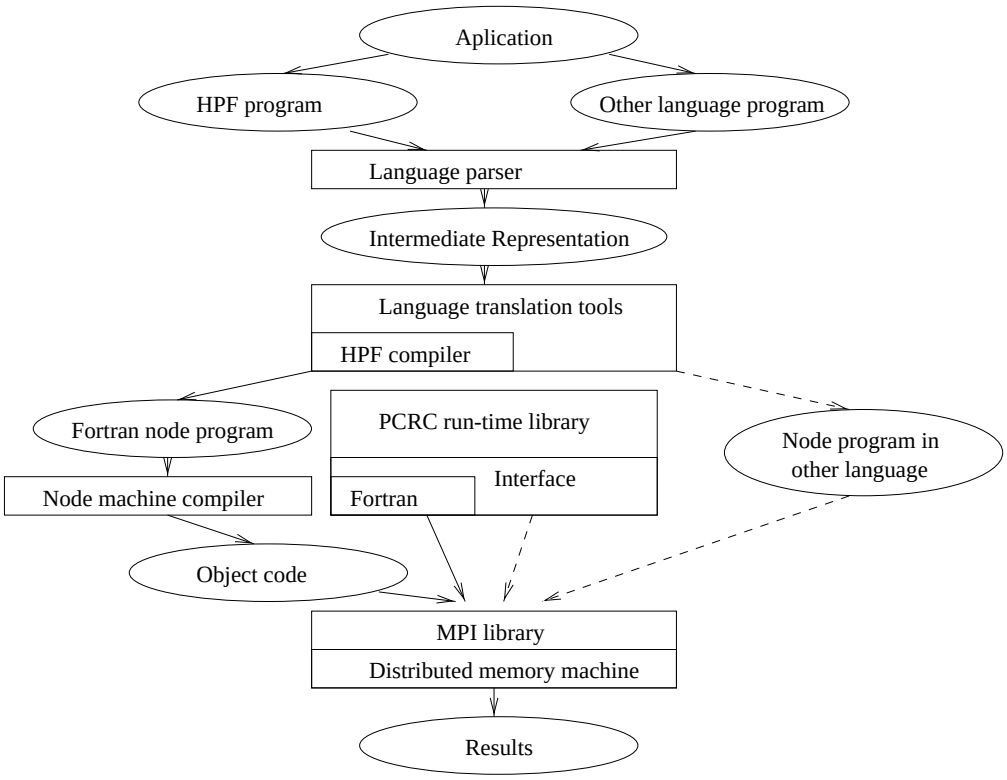

Figure 1: System overview

1 Run-time based HPF compilation

The HPF compiler is embeded in a compiler construction framework showed in the figure system overview. Currently it has already handled different HPF language features. In the following, we give an introduction of the compiler construction framework, then discuss the major issues considered in the compiler implementation.

1.1 Compiler construction framework

The compiler construction framework, called *frontEnd* system, is an extendible software package for constructing programming language processing tools. The system analyzes the input source language, and converts it to a uniformed intermediate representation (IR), which later can be manipulated by a set of functions provided in a C++ class library.

We can see the structure of the package from the following components:

- **Language parsers** The language parsers are used to convert program source code to intermediate representation (IR). The IR is designed to be suitable for multipule programming language. We already have an HPF language parser (including a semantics checking module to check all constrains in language definition) and a Java language

parser. A C++ parser is being developed. So the usage of the package will certainly beyond the HPF compiler itself as more compilation work is involved.

- **Package tools** These tools are used for displaying the itermediate representation. Two of them are available now. One is to unparse the IR back to different language source code for which the parse tree is generated. The other one is to dump IR in plain text or HTML format for debugging purpose.

- **C++ class library** A C++ class library was extended from Sage++ class library. In the library, the supported language component is mapped to different C++ class, with corresponding member functions provided to perform standard transformation, such as insert or delete a statement on the syntax tree.

- **Testsuite** An extensive testsuite is designed for different language parsers. For example, the package include 1) a "first order" exhaustive syntax test, in which each syntax rule are used at least once. 2) all example programs from "Migrating to Fortran 90" and "HPF handbook". as test program for the HPF parser.

The package has been autoconfigured and tested on different platforms, including: IBM AIX, SUN OS/Solaris, DEC OSF, HP HP-UX, SGI IRIX and PC Linux.

1.2 Compiler implementation

The implementation of the compiler is based on our classification for the communication pattern needed in each computation.

1.2.1 Communication detection

We use FORALL statement as example and use *LHS* to denote array reference in the left-hand-side of the FORALL assignment, *RHS* for the right-hand-side. And we use "owner computes" rule to partition the computation, then the communication pattern will be decided by the FORALL index range, index referrence in *LHS*, *RHS*, and the data mapping directives for the *LHS*, *RHS*.

A comprehensive detect-communication algrithom is used in our compiler, possible communication patterns include:

- No communication: The corresponding *LHS* and *RHS* array elements needed to do calculation are located on the same machine.

- Shift communication: The *RHS* data need to move in parallel with same distance along one or more dimensions. The communication detection algorithom is also used to caculate the actual shift amount.

- Remap communication: The *RHS* data movement is among different dimension of the array or the distance is not equal in the same dimension.

The communication pattern will determine two importand aspects in the generated node program: 1) memory allocation and 2) execution control for computation/communication.

1.2.2 Memory management

There are two memory allocation strategy used in our compiler: 1) dynimicaly allocate temproray array. 2) allocate "ghost area" along the *RHS* array which need a small amount of shift on the processor grid.

The first method is useful to handle remap communication. Usually, a data used in the computation is allocated according to HPF `TEMPLATE` size distributed on each processor. When remap is needed, a temprary array will be allocated with the same alignment and distribution as the *LHS* array. The *RHS* need communication will be copied to the temporary array from its old distribution, so the compuation can be carried on.

The second method is used to efficently handle shift communication. As compiler can detect the shift amount for each *RHS* array, a "ghost area" will be set along each array need shift. Then whenever required by the computation, only "edge" elements of the array need to be sent to the neighbor processor. This will save memory copy time compared with the remap case.

In addition to allocation method, the memory management need to take care of two levels of address translation: 1) The global address and local address. 2) The node program use linearized array index to facilitate procedure call with array varibale as argument.

1.2.3 Execution control

The execution control in node program have to deal with DAD initialization, expressions and assignment, execution control in source program, input/output and procedure call.

Given PCRC-runtime Fortran interface, DAD initialization is strightforward. The execution control need to be remained in the node program. The rest three items can be considered as "computation" in the source program. Different kinds of communication may happen as we described previously. The difference among them are: in procedure call the dummy argument will be treated as the *LHS* and actual argument as *RHS*; in I/O statement, remap is always used to send data to or from the root processor.

Particularly, linearization of array index and the DAD handler provided in runtime library will help to implement the transcriptive features of HPF procedure, such as `INHERIT` derictive, effectively. Copy-in and copy-out in neither caller nor callee are necessary.

1.2.4 Examples

We use an example to illustrate the actual transform needed in node program. Suppose we have the following program header:

```
      PROGRAM MAIN
      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)
```

The communication detection will find a FORALL statement like:

```
FORALL (i=8:112:1) X(i) = Y(1*i-1)
```

does not need communication, so the corresponding node program 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)
 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 the program segment, `pcrc_new_range_loop` is used for caculating local loop bounds, `pcrc_coef` for address translation. The actual compuation is guarded by an IF statement to make sure it owns the *LHS* array.

If we change the alignment a little, as:

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

A shift communication is needed. 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 Y should be 2 emlement larger than the one in the previous case. 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 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.