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*

July 11, 1997

Abstract

This paper describes an ongoing effort supported by ARPA PCRC (Parallel Compiler Runtime Consortium) project. In particular, we discuess the design and implementation of an HPF compilation system based on PCRC runtime. The approaches to issues such as directive analysis and communication detection are discussed in detail. The discussion includes fragments of code 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.

[*] Visiting scholar from HIT, China

1 Introduction

HPF has been around for a while [1]. Some early expectations—efficient and robust compilers arriving at the market within a few years of the specification's release—have not been fully realized, but we do see HPF compilers from PGI, IBM, and DEC. A number of other companies, including SUN, have HPF compiler groups. In any case, HPF provides an excellent context for research and development on parallel compilation systems, because it coherently embodies many of the issues and concepts that have emerged over several years concerning parallel processing on distributed memory machines.

One of motivations for HPF was that it is just too difficult to build a parallelizing compiler that produces efficient code from raw FORTRAN applications. HPF requests the application programmer to help the compiler. Experience to date shows that, even with this help, it remains non-trivial to build a *full-featured* and *high performance* compiler for a language as complicated as HPF. This is especially true if the node program generated by the compiler must be directly concerned with low-level communication issues, at the level of calls `send()` and `receive()` operations. Thus, PCRC emphasizes the use of a *run-time library* [2]. The node program will implemented in terms of higher level operations, more easily generated by a compiler and more easily understood by human. Of course, performance remains an issue, since this implies the compiler must relinquish some opportunities for global optimization. It is unclear how much performance is actually gained from those global optimizations.

As part of the PCRC effort, an HPF compiler is being constructed based on this approach. Two aspects are being addressed through this effort. One is to evaluate the effectiveness of PCRC-based approach to parallel compiler construction; the other is to see the performance profile in comparison with "lower level" approaches.

At present, the system has been partially constructed. As we will illustrate, the runtime-based approach allows us to attack a full range of issues encountered in real world compiler construction. For instance, the compilation of procedure calls is implemented as a first priority, which is unusual in academic compiler work.

This paper describes the design and implementation of our system. In particular the approaches to issues such as directive analysis and communication detection 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 everything together, and illustrates how the compilation is done with some examples of generated code. Section 5 gives a few results from a benchmark comparision.

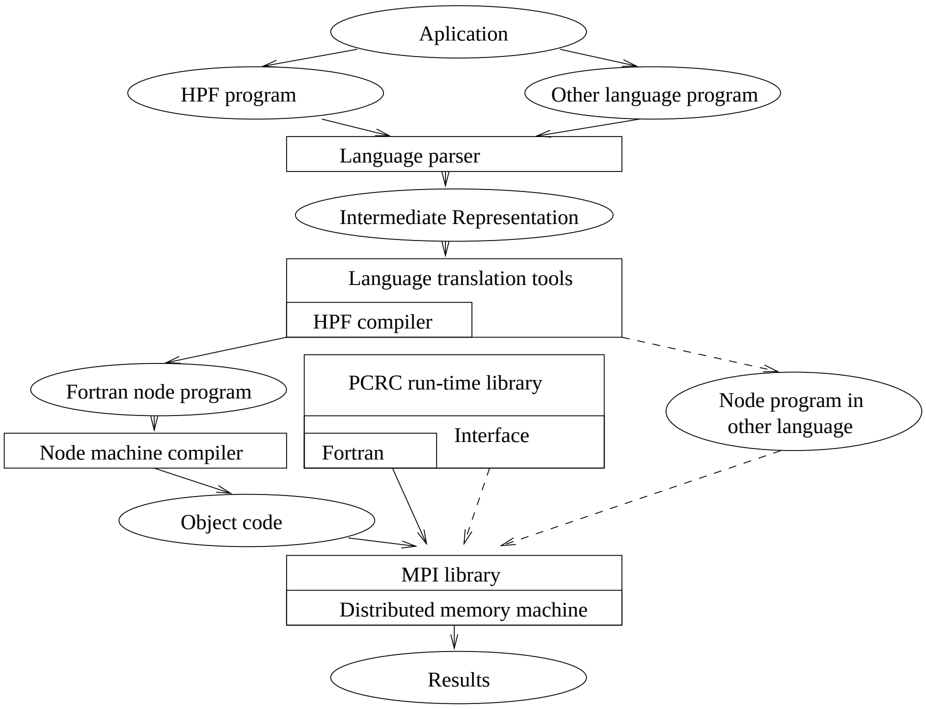

Figure 1: Compilation system overview

2 System Overview

There are three major components in our system (see figure 1): a full featured HPF 1.0 front-end, `HPFfe` [3], a set of transformation modules, and the PCRC runtime [4].

PCRC runtime

The architecture of NPAC PCRC runtime is discussed in section 3.2. 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. The library is implemented in C++, and provides a Fortran interface to the compiler. Section 4 gives a flavor of the Fortran interface.

`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++ [7] 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 [6].

Transformation modules

The compilation can be divided into two major phases: a program analysis phase and a program transforming phase. In the first part, the compiler will 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. It can subdivided as program format transformation and node program generation two parts. These modules will be discussed further in section 4.

3 Key technologies

We describe three technologies employed in our compilation system, which are essential both to the compiler construction work and the performance of generated code. They are *distributed data descriptor*, the NPAC *runtime kernel*, 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.

3.1 Distributed data descriptor

Explicit array data distribution is a core concept of HPF. It frees the compiler from the task of *data partitioning*. Data distribution directives, such as `ALIGN` and `DISTRIBUTE`, provide a convenient way to describe how arrays in a global address (index) space are distributed among processors of a distributed memory machine. 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 a *distributed data descriptor* or *DAD* for this purpose. Similar mechanisms are also used in other compilers (such as PGI compiler, shpf compiler, and previous NPAC F90D compiler), but actual designs differ considerably. Our experience has shown that designing an effective DAD is non-trivial, if it has 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) while still retaining runtime efficiency.

A notional tabular representation of the DAD is given in figure 2. This picture gives a feel for the information held in the actual array descriptor, although

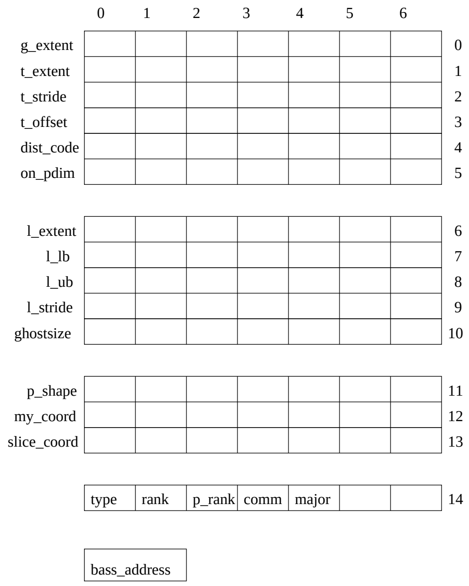

Figure 2: A representation of the distributed array descriptor

it is not a particularly accurate reflection of the runtime data structure—in the current implementation this is a C++ object with various subcomponents. It compactly supports all HPF 1.0 data distribution patterns, including the above mentioned dynamics, and supports efficient traversal of local array elements. 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 [8] or Chapter 6 of [6].

3.2 Runtime kernel

The kernel of NPAC library is a C++ class library. It is descended from the run-time library of an earlier research implementation of HPF[5] with influences from the Fortran 90D run-time and the CHAOS/PARTI libraries. The kernel is currently implemented on top of MPI. The library design is solidly object-oriented, but efficiency is maintained as a primary goal. Inlining is used extensively, and dynamic memory allocation, unnecessary copying, true procedure calls, virtual functions and other forms of indirection are generally avoided unless they have clear organizational or efficiency advantages.

The overall architecture of the library is illustrated in figure 3. At the top level there are several compiler-specific interfaces to a common run-time kernel. The four interfaces shown in the figure are illustrative. They include two different Fortran interfaces (used by different HPF compilers), a user-level

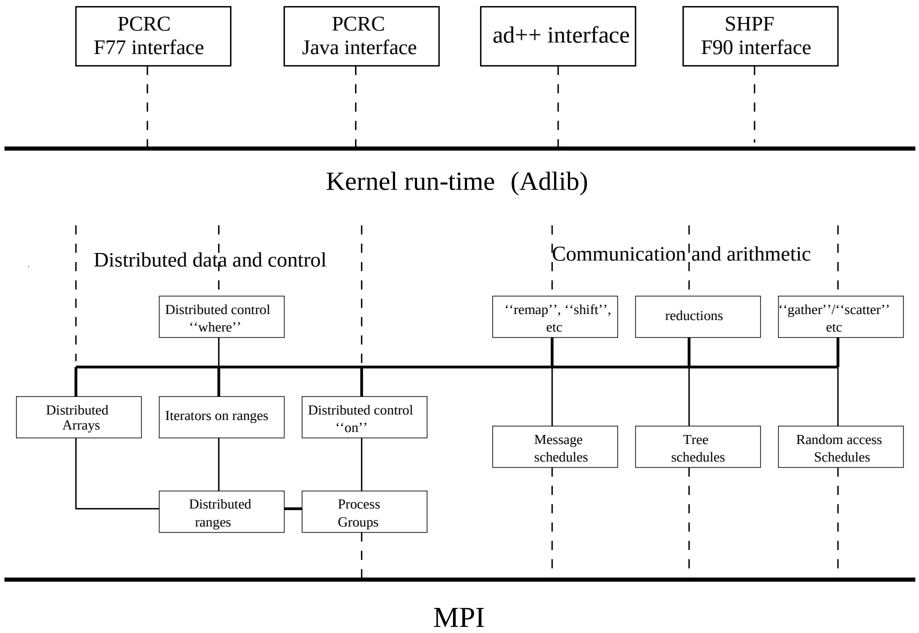

Figure 3: PCRC runtime architecture

C++ interface called ad++[1], and a proposed Java interface[2]. The development of several top-level interfaces has produced a robust kernel interface, on which we anticipate other language- and compiler- specific interfaces can be constructed relatively straightforwardly.

The largest part of the kernel is concerned with global communication and arithmetic operations on distributed arrays. These are represented on the right-hand side of figure 3. The communication operations supported include HPF/F90 array intrinsic operations such as `CSHIFT`, the function `pcrc_write_halo`, which updates ghost areas of a distributed array, the function `remap`, which is equivalent to a Fortran 90 array assignment between a conforming pair of sections of two arbitrarily distributed HPF arrays, and various gather- and scatter-type operations allowing irregular patterns of data access. Arithmetic operations supported include all F95 array reduction and matrix arithmetic operations, and HPF combining scatter. A complete set of HPF standard library functions is under development.

Nearly all these operations (including many of the arithmetic operations) are

[1]ad++ is currently implemented as a set of header files defining distributed arrays as type-secure container class templates, function templates for collective array operations, and macros for distributed control constructs.

[2]We also intend to produce a METACHAOS interface.

based on reusable schedules, in the PARTI/CHAOS mold. As well as supporting the inspector-executor compilation strategy, this organization is convenient in an object-oriented setting—a communication pattern becomes an object. As an illustration, consider the reduction operations. All reductions from a distributed array to a global result are described by an abstract base class using virtual functions for local block reductions. Specific instances such as `SUM` or `PRODUCT` are created by deriving concrete classes that instantiate the arithmetic virtual functions. This is a cleaner and more type-secure (hence, potentially, more efficiently compilable) alternative to passing function pointers to a generic reduction function.

For regular data movement operations a schedule consists of lists of source and destination blocks for local copies or send or receive operations. A block is defined as a multi-dimensional local array section parametrized by an offset and two short vectors of extents and strides. Where blocks are non-contiguous due to striding, or several blocks need to be communicated between the same pair of processors to execute a schedule, data is agglomerated by copying from user space to a buffer before sending, and copied back after receiving.

All the data movement schedules are dependent on the infra-structure on the left-hand side of the figure 3. This provides the distributed array descriptor, and basic support for traversing distributed data ("distributed control"). Important substructures in the array descriptor are the *range* object, which describes the distribution of an array global index over a process dimension, and the *group* object, which describes the embedding of an array in the active processor set.

At the time of writing the kernel is fully functional and quite mature, two of the four interfaces illustrated are complete, and others are in progress.

3.3 Communication classification and detection

HPF directives release the compiler from the task of choosing the 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*. We have developed a theory to detect them by the compiler.

The the meaning of *no communication* is self evident. Here is a reasonably straightforward example

```
      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)
```

In general the conditions for no communication may be non-trivial to compute. In our scheme *no communication* is assumed if the conditions defined below for *shift communication* obtain, but with a shift amount of zero (a sufficient but not exhaustive test).

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

```
      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)
```

needs only shift communication.

The condition for *shift communication* is based on the concept of *shift-homomorphism*. Consider the fragment of HPF in figure 4. Assume t_x and t_y are normalized to be multiples of p. The array sections in the assignment are shift-homomorphic if they have the same extent (number of elements) and

$$\frac{a_x.x_s}{a_y.y_s} = \frac{t_x}{t_y} \tag{1}$$

(a different definition applies if both templates are cyclically distributed).

If this condition holds the section assignment can be implemented by shifting the values of `X` along template `TY` then performing a local copy. We omit the proof of this claim and the formula for the shift amount.

Remap communication is the final catch-all—the bag in which all other section assignments are put.

Appropriate functions are provided in PCRC runtime to support the three situations. For instance, a `pcrc_write_halo()` function is designed to efficiently

```
!HPF$  PROCESSORS P(p)
!HPF$  TEMPLATE TX(t_x), TY(t_y)
!HPF$  DISTRIBUTE TX(BLOCK) ONTO P
!HPF$  DISTRIBUTE TY(BLOCK) ONTO P

!HPF$  DIMENSION X(M), Y(N)
!HPF$  ALIGN X(I) WITH TX(a_x* I + b_x)
!HPF$  ALIGN Y(I) WITH TY(a_y* I + b_y)
       ...
       X(x_l:x_u:x_s) = Y(y_l:y_u:y_s)
```

Figure 4: Generic array section assignment

deal with shift communications, and a `pcrc_remap()` function is designed to handle remap communications. For detailed derivation of our communication detection algorithm, the reader is referred to [9] or Chapter 8 of [6]. Section 4 will also give a specific application of the algorithm.

4 Putting the pieces together

The NPAC compiler is implemented as a translator from HPF to Fortran 77. It focusses on exploitation of explicit forall parallelism in the source HPF program. The transformation modules perform two basic functions, program analysis and transformation. In this section, we describe these modules and give concrete fragments of node code generated by our compiler.

4.1 Program analysis

In the program analysis 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
- distribution 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 obtained from PROCESSOR, TEMPLATE, DISTRIBUTE and ALIGN statements respectively. Their translation in node program are straightforward—generating a DAD for each array declaration, as illustrated later in this section.

In translating a forall statement into a FORTRAN DO construct to be executed on a sequential machine, the "owner computes" rule is used to assign the computation to each node processor. For example:

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

If A is a non-partitioned array and B is a partitioned array, then a broadcast is needed. If the array is a partitioned one the communication needed is dependent on the reference pattern of the forall index. Detection of the communication pattern was discussed in section 3.3.

4.2 Program transformation

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 fixed. For example simple array assignments can be trivially converted to forall statements, and treated as such in the next phase. The language features encountered in the second phase are thus narrowed down. Since the transformation keeps the semantics of the original program unchanged, it is 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 translation work and generates the node program. Below we will use simple examples to illustrate the translations done for dif ferent language components. For simplification, the examples only involve one-dimension arrays. The scheme introduced here can be generalized to deal with the multi-dimension arrays and array sections. This generalization is implemented in our HPF compiler framework.

Housekeeping: memory management and address translation

There are two memory allocation strategies used in our compiler: dynamically allocate a temporary for each RHS term, or allocate a "ghost area" for arrays that appear in RHS contexts where they need a small shift along the processor grid. The first method is used to handle "remap" communication. When a call to `pcrc_remap` is needed, a temporary array is allocated with the same alignment and distribution as the LHS target array and the RHS term is copied to the temporary array. The second method is used to efficently handle "shift" communication. If the compiler detects the need for a shift a "ghost area" is added to the RHS array. "Edge" elements are transferred using `pcrc_write_halo`. This saves the cost of copying a whole array.

As well as memory allocation, the node program must deal with translation between global array subscripts and local (node) subscripts. The run-time provides various functions to help with this translation[3]. The node program linearizes subscript computations for multi-dimensional arrays. 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.

DAD generation

The compiler must generate code and initialize the distributed array descriptors (DADs) passed to run-time functions and sub-programs. Using the PCRC-runtime Fortran interface, DAD initialization is straightforward.

The 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-12)

      ...
```

translates 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,(-12),0,2,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)
```

[3]But these functions are never called from within the inner DO loops generated by translation of a forall.

```
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)
```

For each processor array a *grp* value is created with the appropriate shape. For each template dimension, a *rng* value is created to record its distribution code, distribution stride and offset. For each partitioned array, a *dad* value is created to record its shape and its alignment stride and offset, it is the DAD handle for this array. These are all integer handles to runtime objects. At the end of the program, destructors will be called for the created objects.

Expressions and assignment

Some preliminary work has already been done in the format transformation phase, and the major task of this phase is to deal with forall statements and scalar assignments.

Assuming the program header in previous example, consider the forall statement

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

A shift communication is needed. The local `Y` segment should be extended by 2 element larger to include a ghost area on the left. The `pcrc_write_halo` will send the edge data to the appropriate position in the next processor. The translation is

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

The local loop bounds and stride: `lil`, `liu`, `lis` are calculated according to the forall index range and the DAD of the *lhs*. The 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 a portion of the *lhs* section is held on the current processor, an `if` statement is inserted. Finally, destructor for the temporary objects describing the lhs section and the forall range are called.

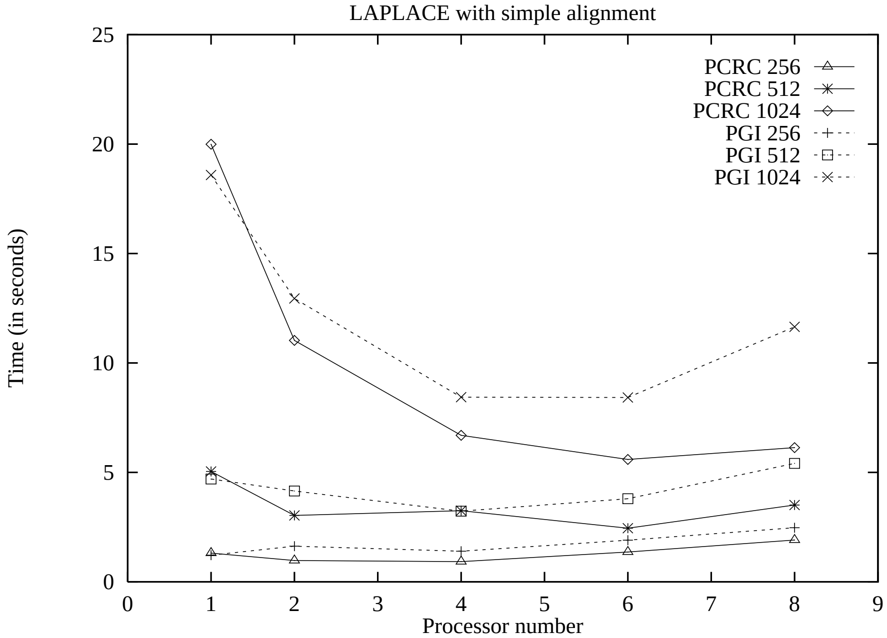

Figure 5: Laplace update.

5 Preliminary Benchmark Results

Figures 5 and 6 display select results of a benchmark comparision between the new NPAC compiler the PGI HPF compiler, version 2.0. The programs were run on the IBM SP2 at NPAC. The Laplace benchmark performs Jacobi relaxation on 256×256, 512×512 and 1024×1024 arrays, distributed blockwise over various numbers of processor. Both compilers achieve about the same performance on a single node, but generally our compiler exhibits better speedup on multiple processors, presumably due to more effective handling of communication. The synthetic benchmark involves no communication—it is a forall assignment involving large arrays. It suggests that (unlike the PGI compiler) we deal with address translation efficiently, even for cyclic distribution format. (Speedup is relative to an equivalent sequential program compiled with the IBM Fortran compiler.)

While these examples are necessarily select, in general we find that (on code that both compilers can successfully compile) the NPAC compiler compares very favourably with the commercial compiler.

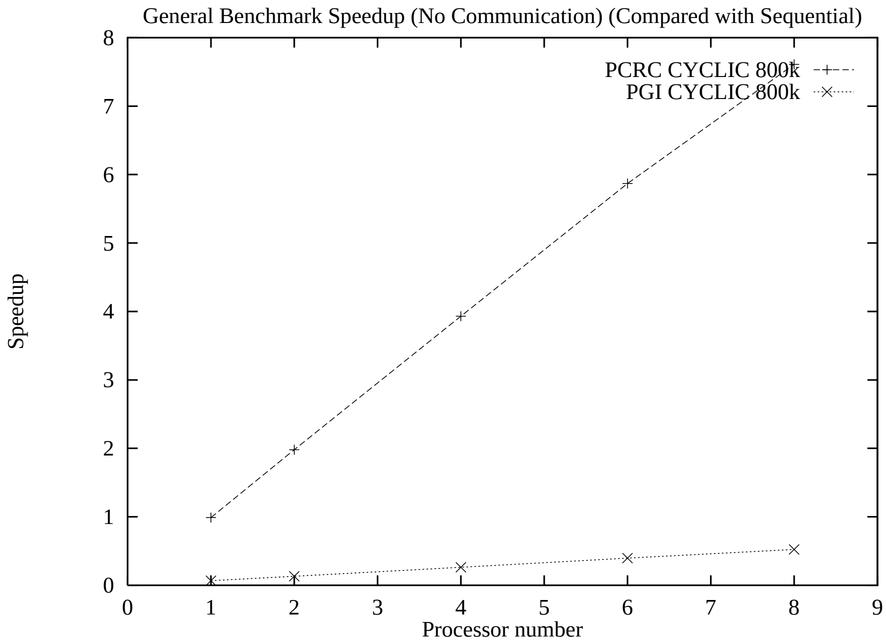

Figure 6: Synthetic benchmark involving cyclic distribution format.

6 Discussion

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

Automatic generation of message passing programs from data distribution specifications has been explored for some time in the context of various data parallel languages [10], [11], [12], [13] and [14].

In [13], the support of the run-time functions are relatively weak; the compiler needs to generate send and receive primitives to accomplish communication. Though this may have more efficient code generation after extensive program analysis, the compiler may become too complicated to be operational.

The most recent paper on HPF compiling was [15], in which a local set enumeration method was used to generate local part of a loop iteration and derive the communication set. Comparatively speaking, we believe our run time support method to get values is more straightforward and efficient, especially for regular access to the array data.

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

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, Don Leskiw, Xiaoming Li, "PCRC runtime interface (Ver 0.5)," NPAC Technical Report, SCCS-799, July 10, 1996.

[5] John Merlin, Bryan Carpenter and Tony Hey, "shpf: a Subset High Performance Fortran compilation system," Fortran Journal, pp 2-6, March 1996.

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

[7] 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.

[8] 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.

[9] Xiaoming Li, Yuhong Wen, "Efficient Compilation of Forall Statement with Runtime Support," NPAC Technical Report, SCCS-800, October, 1996.

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

[11] 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

[12] 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

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

[14] 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

[15] 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