Language Bindings for a Data-Parallel Runtime

Bryan Carpenter Geoffrey Fox Donald Leskiw Xinying Li Yuhong Wen
Guansong Zhang
NPAC at Syracuse University
Syracuse, NY 13244
{dbc,gcf,leskiwd,xli,wen,zgs}@npac.syr.edu

Abstract

The NPAC kernel runtime, developed in the PCRC (Parallel Compiler Runtime Consortium) project, is a runtime library with special support for the High Performance Fortran data model. It provides array descriptors for a generalized class of HPF-like distributed arrays, support for parallel access to their elements, and a rich library of collective communication and arithmetic operations for manipulating these arrays. The library has been successfully used as a component in experimental HPF translation systems. With prospects for early appearance of fully-featured, efficient *HPF compilers looking questionable, we discuss a class of more easily implementable data-parallel language extensions that preserve many of the attractive features of HPF, while providing the programmer with direct access to runtime libraries such as the NPAC PCRC kernel.*

1 Introduction

As part of the PCRC [10] project we completed development of a high-level runtime library for data-parallel languages [4]. The motivating goal was to simplify translation of High Performance Fortran (HPF) [6] by providing a coherent interface to the distributed array descriptors and collective operations needed for straightforward, efficient translation of parallel constructs like FORALL and array assignments. This goal was achieved quite successfully, and two experimental subset HPF translators have used the library to manage their communications [13, 7].

Unfortunately it is evident that implementing compilers for a language as complex as *full* HPF is a formidable task. On the other hand the runtimes that have evolved to support HPF and its kin are powerful and have an underlying elegance, using them without a compiler—through direct calls from an SPMD program written in a standard language—is clumsy and error prone. This is due in part to the large number of parameters needed to describe distributed arrays. We face the possibility that—in the near term future at least—neither HPF or conventional languages will permit full exploitation of libraries such as the one developed in PCRC.

In this paper we discuss possibilities for enhancing programming languages like Fortran with relatively simple extensions to support declaration and manipulation of HPF-like distributed arrays. In contrast to HPF, our extensions assume the programmer specifies in full logical detail exactly where computations and communications are performed. This makes compilers a much more straightforward proposition. In spite of this simplification, we argue that—supplemented by a binding to a suitable collective communication library such as ours—the resulting hybrid data-parallel/SPMD programming models can achieve a level of elegance and expressivity comparable to full HPF.

2 Background: runtime kernel

The kernel of NPAC library is a C++ class library. It is most directly descended from the run-time library of an earlier research implementation of HPF [7] with influences from the Fortran 90D run-time and the CHAOS/PARTI libraries [1, 11, 5]. The kernel is currently implemented on top of MPI. The library design is solidly object-oriented, but efficiency is maintained as a primary goal.

The overall architecture of the library is illustrated in figure 1. 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 C++ interface

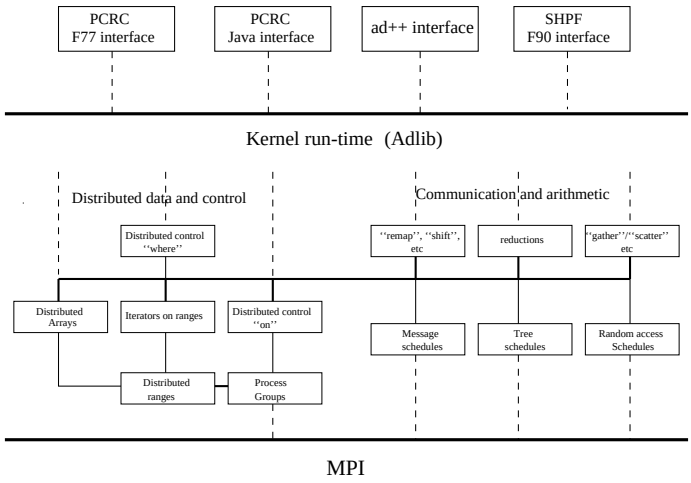

Figure 1. NPAC runtime architecture

called ad++[1], and a Java interface under development. 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 1. The communication operations supported include HPF/F90 array intrinsic operations such as `CSHIFT`, the function `write_halo`, which updates ghost areas of a distributed array, the function `remap`, which is equivalent to a Fortran 90 array assignment between two conforming 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.

All the data movement schedules are dependent on the infra-structure on the left-hand side of the figure 1. 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. Results of a preliminary benchmarks reported in [13] suggest that an HPF compiler based on the high-level NPAC runtime can be competetive with commercial compilers.

[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.

3 The language model

The next section gives an outline of a Fortran dialect for explicit SPMD programming with distributed arrays. Before attempting a concrete syntax, we will discuss some of the general goals and features of such a language.

We aim to provide a flexible hybrid of the data parallel and SPMD approaches. To this end HPF-like distributed arrays should appear as language primitives. New *distributed control* constructs will be added to facilitate access to the local elements of these arrays. In the SPMD mold, the model should allow processors the freedom to independently execute complex procedures on local elements: programs should not be constrained by SIMD-style array syntax.

A design decision is made that all access to *non-local* array elements should go through library functions—typically collective communication operations. This puts an extra onus on the programmer; but making communication explicit encourages the programmer to write algorithms that exploit locality, and simplifies the task of the compiler writer.

For the newcomer to HPF, a great strength of the language lies in the fact that the semantic effect of a particular operation is generally identical to the effect in the corresponding sequential program. This means that, so long as the programmer understands conventional Fortran, it is very easy for him or her to understand the behaviour of a program at the level of what values are held in program variables, and the final results of procedures and programs. Of course the ease of understanding this "value semantics" of a program is counterbalanced by the difficulty in knowing exactly how the compiler translates the program. Understanding the *performance* of an HPF program may require the programmer to have quite detailed knowledge of how arrays are distributed over processor memories, and what strategy the compiler adopts for distributing computations across processors.

The language model we discuss has some superficial (and some deeper) similarities to the HPF model, but the HPF-style semantic equivalence between the data-parallel program and a sequential program is abandoned in favour of a more direct equivalence between the data-parallel program and an SPMD program. Because understanding an SPMD program is presumably more difficult than understanding a sequential pro-

gram, learning naive use of our language will certainly be harder than for HPF. Our claim is that once a set of related concepts about distributed data and *distributed control* are mastered, the kind of language we discuss gives the programmer more intricate control over the behaviour of a program, and this may ultimately lead to better performance. On the other hand, by retaining many of the array-level features of HPF as language primitives, we still enable a higher-level of programming than is possible in the direct message-passing style.

We will adopt a distributed data model semantically equivalent to to the HPF data model. However we describe the distributed arrays in terms of a slightly different set of basic concepts. In general HPF describes the decomposition of an array through alignment to some *template*, which is in turn distributed over a *processor arrangement*. A processor arrangement is a multidimensional grid of abstract processors. A template is a an abstraction of the index space of a distributed array—it is like a multi-dimensional array mapped to the process grid, but has no data associated with its elements.

The analogous concepts in our parametrization of the distributed array are the *distributed range* (or simply *range*) and the *process group* (or simply *group*). A distributed range is like a single dimension of an HPF template (or of some triplet-selected subset of that dimension). It defines a map from an integer subscript interval into a single dimension of an HPF-like processor arrangement. A *process group* is equivalent to an HPF processor arrangement, or to a certain subset of such an arrangement. We emphasize that switching from templates to ranges and groups is a change of parametrization only. In itself it does not change the set of allowed ways to decompose an array. The reason for the change is that groups and ranges seem to be better suited to specifying the distribution of program *control*.

Our language model differs materially from HPF in its dependence on *distributed control* constructs. HPF 2.0 provides some similar mechanisms, but only as optional directives. In our language, by contrast, they are *required* to specify which process performs each action. The simplest example of a distributed control construct is the `ON` construct. This is a control construct parametrized by a process group. It specifies that the enclosed code block is only executed on processes in the group. The second distributed control construct is called `AT`. This is very similar to `ON`, except that it is parametrized by an element of a distributed range. The enclosed code block is only executed on processes that hold the range element concerned (because a distributed range corresponds to a single dimension of an HPF template, this is equivalent to restricting the operation to some slice of a processor arrangement). The last and most important distributed control construct is the `OVERALL` distributed loop. This construct is parametrized by a range or a subrange. It is a direct abstraction of the loops in low-level SPMD programs that iterate over elements of the local portion of some distributed data structure.

Each of these control constructs restricts the *active* group of processors to some subset. In the case of `ON` and `AT` this restriction is obvious. In the case of `OVERALL` the group of processors is effectively partitioned along the process dimension associated with the range. With certain restrictions which are easily stated in terms of the current *active process group*, the distributed control constructs can be nested, and collective operations can be called *inside* distributed control constructs. In other words, collective library operations need not imply global synchronization—only synchronization among members of the current active process group is assumed.

The fundamental constraint that forces the use of the distributed control constructs is the requirement that all access to array elements must be local. Although arrays are subscripted with global subscripts as in HPF, a subscripting operation must not imply access to an element not held on this processor. Meeting this constraint sounds onerous. We will try to illustrate with concrete examples that the distributed control constructs match the requirements of typical parallel algorithms well, and once the basic ideas are grasped, programming around this constraint becomes quite natural.

A further advantage of this style is that, because the underlying programming model is SPMD, the switch between data parallel programming and low-level message-passing is merely a change in point of view. Unlike HPF, there is no need to pass through an awkward "extrinsic" interface if a particular subcomputation cannot easily be handled by collective data-parallel operations. We simply need to provide inquiry functions that provide access to the local sequential array component of a distributed array, and to the local physical process id.

4 Outline of an extended Fortran dialect

4.1 Example syntax

The examples in the following sections use certain syntax extensions to Fortran. The syntax is illustra-

tive, and by no means finalized.

A distinguishing property of the proposed system, compared to HPF, is that it includes ordinary Fortran as a strict subset, *and ordinary Fortran constructs are unchanged by the translator*. Our system will *not* attempt to exploit parallelism even in "explicitly parallel" constructs such as the array syntax of Fortran 90 or the FORALL statement of Fortran 95. This policy drastically simplifies the translator, and gives the programmer much finer control over the generated code.

Processor arrangements will be declared as in HPF. The explicit `TEMPLATE` concept of HPF is abandoned in favour a *distributed range* concept. A distributed range represents a range of subscript values, and incorporates a mapping of that range into a dimension of a processor arrangement. The mapping options will be similar to HPF: block, cyclic, block-cyclic, etc. A distributed array declaration is distinguished from a sequential array declaration by using a different kind of brackets:

```
PROCESSORS P(4, 4)

RANGE X(N), Y(M)
DISTRIBUTE X ONTO P(BLOCK, *)
DISTRIBUTE Y ONTO P(*, CYCLIC)

REAL A [X, Y]
INTEGER B [Y]
INTEGER C [X, Y, 10]
REAL D [X(1 : N / 2), Y(::2)]
```

Ranges `X` and `Y` are each distributed over a single dimensions of the processor arrangement. The array `A` is an `N` by `M` distributed array, distributed blockwise in its first dimension and cyclically in its second. In general the brackets in a distributed array declaration contain a list of *orthogonal* ranges[2]. `B` is a one dimensional array distributed cyclically in the second dimension of `P` and implicitly replicated over the first. `C` is a three dimensional array with two distributed and one *collapsed* range. The declaration of D illustrates how HPF-like non-trivial alignment relations can be introduced by using subranges in array declarations. `D` is an `N / 2` by `N / 2` distributed array[3]

Provision of distributed arrays as language primitives provides a clean, systematic interface to libraries of collective operations. Examples of standard collective operations are

```
CALL DA_CSHIFT(DST, SRC, DIM, SHIFT)
RES = DA_SUM(SRC)
CALL DA_REMAP(DST, SRC)
```

Here `DST` and `SRC` are distributed arrays. The subroutine `DA_CSHIFT` is closely analogous to the Fortran 90 `CSHIFT` intrinsic. The function `DA_SUM` sums all elements of the argument and broadcasts the result value. The subroutine `DA_REMAP` takes two distributed arrays of the same shape *which may have any, unrelated mapping* and copies the elements of one to the other[4].

The syntax for declaration of *distributed data* is complemented by syntax extensions for *distributed control*. The simplest example is the `AT` construct. A distributed array is subscripted with a global subscript. Subscripting expressions are only meaningful on the processors that hold the elements selected. To make sure a statement using a distributed array element is only executed on a processor that holds copies of the element, a stylized form of conditional called the `AT` construct is provided. The code fragment below illustrates a pair of nested *at* constructs.

```
AT(X(n))
  AT(Y(17))
    A [n, 17] = B [17] + 23 ;
  ENDAT
ENDAT
```

This construct is an abstraction of the *if* statement in a low-level SPMD program that tests whether the local processor contains a desired array element. The body of the construct only executes on processors that hold the specified range element.

A more important and powerful distributed control construct is the `OVERALL` distributed loop. This construct superficially resembles the Fortran 95 `FORALL` construct. It differs from `FORALL` in several respects

- The index ranges are distributed ranges, not triplets.
- There are no restriction on what kind of statements can appear in the body of the construct. Any executable statements are allowed. This reflects the explicit SPMD emphasis of the language, in constrast to the SIMD heritage of HPF.
- Each "iteration" of the construct is localized to a well-defined processor (or group of processors) through the use of distributed ranges for the loop indices. An iteration should only access array elements held locally on a processor executing the iteration. This restriction sounds inconvenient:

[2] Two ranges are considered orthogonal if they are distributed over different dimensions of the same processor arrangement. As a special case a collapsed, on-processor array range is orthogonal to any other.

[3] Triplet subscripting of ranges works in the same was as triplet subscripting of arrays in Fortran. If lower or upper bounds are omitted they default to the bounds of the subscripted object, so `Y(::2)` is equivalent to `Y(1:N:2)`—a range including every 2nd element of `Y`.

[4] As a rule operations like `DA_CSHIFT` and `DA_REMAP` cannot perform in-place updates. Their arguments should be distinct and non-overlapping.

in practise it can usually be accomodated quite painlessly by limiting access to arrays aligned with the loop ranges.

In this simple example of an `OVERALL` construct is used to initialize the array `A` with some expression involving the global subscripts

```
OVERALL(I = X, J = Y)
  A [I, J] = I + J
ENDOVERALL
```

In a slightly more complex example we add together elements from two arrays

```
OVERALL(I = X, J = Y)
  A [I, J] = B [J] + C [I, J, 17]
ENDOVERALL
```

This operation is legal due to the alignment relation between the `A`, `B` and `C` arrays.

4.2 Example 1: Cholesky decomposition

Figure 2 gives a parallel implementation of Cholesky decomposition in the extended language. In the declaration of `A` the first dimension is specified with an integer range rather than a distributed range. This means that this dimension is on-processor—*collapsed* in HPF terminology. In general the collective operation `DA_REMAP` copies the elements of one distributed array or section to another of the same shape. In the current example, because `B` has replicated mapping, it implements a broadcast. In spite of the fact that we have demanded that the programmer explicitly specify which processor performs every operation, and exactly how communications are to be inserted, we claim that this implementation is at least as simple as any that could be achieved in HPF (and clearer than anything that could be written in MPI).

Other features illustrated by this example are construction of sections of distributed arrays (directly analogous to construction of sections of Fortran 90 arrays) and use of a *subrange* to parametrize an `OVERALL` loop.

4.3 Example 2: Red-Black relaxation

Figure 3 gives a parallel implementation of red-black relaxation in the new language. Following the proposals of HPF 2.0 standard, *ghost regions* are allowed for arrays. In the example the width of these regions is defined by specifying the `SHADOW` attribute (in our language this attribute is specified for ranges rather than arrays). Ghost regions are extensions of the locally

```
INTEGER, PARAMETER :: N = 100

PROCESSORS P(NP)
RANGE, DISTRIBUTE ONTO P(CYCLIC) :: X(N)

REAL A [N, X]

REAL B [N]          ! used as a buffer

! ... initialize the array

ON(P)
  DO K = 1, N - 1
    AT(X (K))
      A [K, K] = SQRT(A [K, K])

      DO L = K + 1, N
        A [L, K] = A [L, K] / A [K, K]
      ENDDO
    ENDAT

    CALL DA_REMAP(B [K + 1 :], A [K + 1 : , K])

    OVERALL(I = X (K + 1 :))
      DO J = I, N
        A [J, I] = A [J, I] - B [J] * B [I]
      ENDDO
    ENDOVERALL
  ENDDO

  AT(X (N))
    A [N, N] = SQRT(A [N, N])
  ENDAT
ENDON
```

Figure 2. Implementation of Cholesky decomposition in mooted syntax.

```
PROCESSORS P(NP, NP)

RANGE X(N), Y(N)
DISTRIBUTE X ONTO P(BLOCK, *)
DISTRIBUTE Y ONTO P(*, BLOCK)
SHADOW (1) X, Y

REAL A [X, Y]

ON(P)
  ! Initialize the array...

  OVERALL(I = X, J = Y)
    A [I, J] = ...
  ENDOVERALL

  DO ITER = 1, NITER
    DO ICOLOUR = 0, 1
      CALL DA_WRITE_HALO(A, (/CYCL, CYCL/), (/1, 1/))

      OVERALL(I = X)
        OVERALL(J = Y(MOD(ICOLOUR + I, 2) : : 2))
          A [I, J] = (A [I, J - 1] + A [I, J + 1] +     &
     &                A [I - 1, J] + A [I + 1, J]) / 4
        ENDOVERALL
      ENDOVERALL
    ENDDO
  ENDDO
ENDON
```

Figure 3. Implementation red-black relaxation in mooted syntax.

held block of a distributed array, used to cache values of elements held on adjacent processors. The ghost regions are explicitly brought up to date using the subroutine `DA_WRITE_HALO` from the standard library. The arguments following the array itself define the mode of updating ghost regions at the extremes of the array (cyclic wraparound in this slightly unrealistic example) and the actual width of the halo to be written. Subsequently the `OVERALL` construct can access elements of the array displaced by up to one place from the alignment of the range parametrizing the construct.

Note the ease with which sites of a particular colour are selected by using nested `OVERALL` constructs, parametrizing the inner one with a subrange. In HPF this could be expressed using array syntax and a `WHERE` construct, but that would be relatively inefficient. Alternatively it could be expressed using nested `INDEPENDENT DO` loops or `FORALL`s, but nesting these constructs makes it difficult for a compiler to analyse them and emit efficient parallel code. In our scheme no special analysis is necessary. A simple efficient scheme is applied universally to translate subscripts in `OVERALL` constructs.

5 Discussion and related work

The language model described in this paper has previously been investigated using C++ class libraries [2]. Currently we are moving similar syntactic ideas to to an extended Java dialect. This is seen as an intermediate stage, preliminary to implementing a Fortran translator. For a more complete exposition of the underlying parallel programming model proposed here, the reader is refered to [3], which gives examples in an extended Java syntax. The C++ and Java versions are essentially prototype systems, not offering the immediate expectation of very high performance. Our hope is that a Fortran translator for a language such as the one described here will be an order of magnitude easier to implement than a compiler for full HPF. We also suspect it may offer higher performance than many existing HPF systems, relying on reasonably sophisticated programmers, rather than on very advanced compilers. Moreover, we hope that for certain applications the explicitly SPMD language model espoused here will be more flexible and convenient than HPF.

An initial implementation of our language will take the form of a translator to ordinary Fortran. The distributed arrays of the extended language will appear in the emitted code as a pair—an ordinary Fortran array of local elements and a handle to a Distributed Array Descriptor (DAD). Details of the distribution format, including non-trivial details of global-to-local transla-

tion of the subscripts, are managed in the run-time library. Acceptable performance should nevertheless be achievable, because we expect that in useful parallel algorithms most work on distributed arrays will occur inside `OVERALL` constructs. In normal usage, the formulae for address translation can then be linearized. The non-trivial aspects of address translation (including array bounds checking) can be absorbed into the startup overheads of the loop. Since distributed arrays are usually large, the loop ranges are typically large, and the startup overheads (including all the run-time calls associated with address translation) can be amortized. This approach to translation of parallel loops is discussed in detail in [4].

Note that if array accesses are genuinely irregular, the necessary subscripting cannot usually be *directly* expressed in our language, because subscripts cannot be computed randomly in parallel loops without violating the fundamental SPMD restriction that all accesses be local. This is not regarded as a shortcoming: on the contrary it forces explicit use of an appropriate library package for handling irregular accesses (such as CHAOS [5]). Of course a suitable binding of such a package is needed in our language.

A complementary approach to communication in a distributed array environment is the one-sided-communication model of Global Arrays (GA) [8]. For task-parallel problems this approach is often more convenient than the schedule-oriented communication of CHAOS (say). Again, the language model we advocate here appears quite compatible with GA approach—there is no obvious reason why a binding to a version of GA could not be straightforwardly integrated with the the distributed array extensions of the language described here.

We mention two projects that have some similarity to the work described here.

ZPL [12] is a new programming language for scientific computations. Like Fortran 90, it is an array language. It has an idea of performing computations over a *region*, or set of indices. Within a compound statement prefixed by a *region specifier*, aligned elements of arrays distributed over the same region can be accessed. This idea has certain similarities to our `OVERALL` construct. In ZPL, parallelism and communication are more implicit than in our proposed languge. The connection between ZPL programming and SPMD programming is not explicit. While there are certainly attractions to the more abstract point of view, the language we are proposing deliberately provides lower-level access to the parallel machine.

F-- [9] is an extended Fortran dialect for SPMD programming. The approach is quite different to the one proposed here. In F--, array subscripting is local by default, or involves a combination of local subscripts and explicit process ids. There is no analogue of global subscripts, or HPF-like distribution formats. In F-- the logical model of communication is built into the language—remote memory access with intrinsics for synchronization. In our proposal there are no communication primitives in the language itself. We follow the MPI philosophy of providing communication through separate libraries. While F-- and our approach share an underlying programming model, we believe that our framework offers greater opportunities for exploiting established software technologies, such as the PCRC libraries.

References

[1] A. Agrawal, A. Sussman, and J. Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6, 1995.

[2] B. Carpenter. *Programming in ad++*, 1998. http://www.npac.syr.edu/projects/pcrc/doc.

[3] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. *Introduction to Java-Ad*, 1997. http://www.npac.syr.edu/projects/pcrc/doc.

[4] B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. Up-to-date version maintained at http://www.npac.syr.edu/projects/pcrc/doc.

[5] R. Das, M. Uysal, J. Salz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.

[6] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.

[7] J. Merlin, B. Carpenter, and T. Hey. shpf: a subset High Performance Fortran compilation system. *Fortran Journal*, pages 2–6, Mar. 1996.

[8] J. Nieplocha, R. Harrison, and R. Littlefield. The Global Array: Non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[9] R. Numrich and J. Steidel. F–: A simple parallel extension to Fortran 90. *SIAM News*, page 30, 1997.

[10] Parallel Compiler Runtime Consortium. Common runtime support for high-performance parallel languages. In *Supercomputing '93*. IEEE Computer Society Press, 1993.

[11] R. Ponnusamy, Y.-S. Hwang, R. Das, J. H. Saltz, A. Choudhary, and G. Fox. Supporting irregular distributions using data-parallel languages. *IEEE Parallel and Distributed Technology*, Spring, 1995.

[12] L. Snyder. A ZPL programming guide. Technical report, University of Washington, May 1997. http://www.cs.washington.edu/research/projects/zpl/.

[13] G. Zhang, B. Carpenter, G. Fox, X. Li, X. Li, and Y. Wen. PCRC-based HPF compilation. In *10th International Workshop on Languages and Compilers for Parallel Computing*, 1997. To appear in Lecture Notes in Computer Science.