Abstract Interface to the PCRC Common Code Data Descriptor

DRAFT

Bryan Carpenter, James Cowie, Don Leskiew and Xiaoming Li

Northeast Parallel Architectures Centre,
Syracuse University,
111 College Place,
Syracuse, New York 13244-410

October 28, 1996

1 Introduction

This document defines an abstract model of the distributed array data structure implemented in the NPAC HPF compilation system. The model is proposed as a PCRC standard for representation of HPF-style, regular distributed data.

The exposition given here is intentionally language-independent. Several bindings of the interface to specific programming languages (Fortran 77, C, C++, Java, Ada) will be presented in accompanying documents.

Section 2 describes the HPF data model—the standard against which the PCRC model must be compared. Section 3 describes in words the contents of our data representation. Section 4 presents an interface to the model, using in a simplified subset of C++ (or Java) syntax. This subset includes only the most basic ideas of classes and member functions, and should be accessible an audience without detailed knowledge of C++. The appendices contain two "reference implementations": one in terms of a "flat" table of integer values, and one in terms of an existing C++ run-time library, Adlib.

2 HPF Requirements

[A concise description of the HPF distributed data model, stating such restrictions as exist in the current PCRC implementation (viz, that multiblock distri-

bution formats are not supported).]

3 Overview of the Data Representation—the 6-Component Model

The concrete representation on an HPF distributed array must be quite complex. This reflects the very general mapping strategy supported for an HPF array: general "affine" alignment to a template which is allowed several possible formats of distribution over an arbitrary virtual process grid. The model presented here attempts to be as economical as practical, within the constraints of these requirements. Although it has the same expressive power as the two-level alignment/distribution description in HPF, it is stated in terms of a different set of primitive concepts, oriented to the requirements of the compiler and the run-time library.

In this model the distributed array descriptor has the following logical components:

1. Element type information.
2. Rank or dimensionality of the array—an integer `R`.
3. Distribution target—the (process) *group* structure.
4. Information on the bounds and distribution of index ranges—a vector of `R` *range* structures.
5. A base address in local memory—a pointer.
6. Information on layout of elements in local memory—a vector of `R` integer-valued *memory strides*.

3.1 Element type information

An array element can have any type representable as a contiguous group of bytes. All elements of the array have the same type.

The type information in the descriptor consists of the element length in bytes and an integer tag field used to provide more detailed information about the type. The tag field may be used, for example, by arithmetic array operations that accept arrays with a specific set of primitive arithmetic types.

3.2 Rank

The rank is a simple integer—the number of dimensions of an array.

The rank may be zero. This allows the descriptor to describe a scalar variables which is localized to a particular process or group of processes.

3.3 Distribution target

An array is distributed onto a particular process grid. In the run-time representation it proves convenient to refine the idea of a distribution target to include some information that would be regarded as alignment information in HPF.

Consider this HPF fragment

```
!HPF$ PROCESSORS a(2, 2)

      INTEGER a(100, 100)
!HPF$ DISTRIBUTE ONTO p(BLOCK, BLOCK) :: p

      INTEGER b(100)
!HPF$ ALIGN b(:) WITH a(1, :)

      INTEGER c
!HPF$ ALIGN WITH a(1, 51) :: c
```

The alignments here are "embedded"; the align subscripts include scalar expressions.

Where an array such as `b` (or indeed a scalar such as `c`[1]) has an embedded alignment it is inconvenient and uneconomical to maintain information about the scalar-subscripted template dimension in the run-time representation of the array. The only vestige of the subscripted template dimension that affects handling of the array is the coordinate in the process grid dimension implied by the subscript.

In the representation presented here scalar align subscripts are avoided. Instead the array `a` and the variable `c` are regarded as being *distributed* over subgrids, as illustrated in figure 1. The first and second dimensions of `p` are labelled `p1` and `p2` respectively. Array `b` is distributed over the subgrid `q`. Variable `c`. is distributed over the subgrid `r`. Generically, a grid or a subgrid is referred to as a (process) *group*, and the run-time representation of an array contains a structure defining the group over which the array is distributed.

The subgrids introduced here are always simple slices of process grids. A particular dimension of the parent grid is adopted wholesale in the subgrid or restricted to a single coordinate value. There is no analogue of a "triplet" subscript for a process dimension.

Notice that even if the kind of embedded alignments illustrated here were not supported directly, exactly the same situation arises in creation of descriptors for array *sections*, where the section involves some scalar subscripts.

3.4 Index ranges

The global subscript associated with an array dimension takes values in the range `0, ..., n`, where `n` is the *extent* of the array in the dimension concerned.

[1] In practise the aligned HPF scalar `c` would be represented as a rank-0 array.

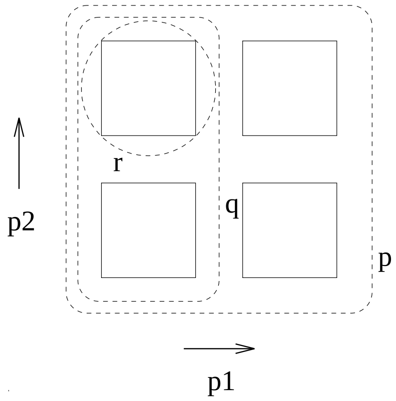

Figure 1: Examples of process groups. The square boxes represent the 4 processes in the process array p. The dashed lines embrace groups p, q and r.

Fortran allows arbitrary lower bounds for array subscripts. Where a subscript must be passed to a run-time library function, the compiler is responsible for reducing the subscript so that it "starts at" zero.

The extent of an array dimension, and the mapping of its global subscripts into a process dimension, is parametrized by a *range* structure. The subscript range of an HPF template dimension can also be described by a range structure.

A *primitive range* corresponds to the range of an HPF template, or an array whose mapping is specified directly through a `DISTRIBUTE` directive. It is effectively parametrized by its extent, its target processor dimension, and a distribution format—currently one of `BLOCK` or `CYCLIC`.

A range which has a non-trivial alignment is effectively parametrized by its extent, a parent primitive range, an alignment offset, and an alignment stride.

Various fundamental operations of the run-time library, such as translation between global and local (node-program) subscripts and calculation of local index bounds of parallel loops are parametrized solely by a range structure. In object-oriented, they are methods on the range class.

[Explanation of how replicated alignments are supported.]

3.5 Base address

The last two items in the descriptor are concerned with the layout of array elements in local (node-program) memory.

The base address is a pointer to base of the local array which holds these elements.

3.6 Memory strides

The range structures define a mapping from global to local subscripts (a mathematician might say that a range structure *is* a map from global to local subscripts). A local subscript actually defines a local element of a primitive range or template. It is the offset of the element from the base of the locally held block of that range or template. A local subscript does not *directly* encode information about the spread of array elements in physical memory.

The memory strides define a map from local subscripts to memory locations. If the local subscripts are

```
l_0, l_1, ...
```

and the memory strides are

```
str_0, str_1, ...
```

then the offset of the element from the base of the local array segment is

```
(str_0 * l_0) + (str_1 * l_1) + ...
```

As a concrete example the HPF declarations

```
!HPF$ PROCESSORS p(2, 2)

      INTEGER a(100, 20, 100)
!HPF$ DISTRIBUTE a(BLOCK, BLOCK, *) ONTO p
```

could yield the Fortran 77 node-program declaration

```
      INTEGER a(50, 10, 100)
```

The values of the strides in the array descriptor would be

```
str_0 = 1
str_1 = 50
str_2 = 10
```

Our model assumes that memory is always allocated according to the size of an array's template, rather than the size of the array itself. If we added the declaration

```
      INTEGER b(50, 20)
!HPF$ ALIGN b(i, j) WITH a(25 + 2 * i, j, *)
```

to the HPF, the extra node-program declaration would be

```
      INTEGER b(50, 10)
```

and the strides for `b` would be

```
str_0 = 1
str_1 = 50
```

in spite the fact that now around half the elements of the locally declared arrays are unused.

4 An Abstract Interface

As explained in the introduction, the language-independent interface will be presented in a simplified C++-like syntax. The benefits of an object-oriented specification are by now, presumably, uncontroversial. A true C++ interface would need considerably more detail. All the methods presented here are publicly accessible. Any private methods and data field are, naturally, omitted from the specification. We retain the convention that a method whose name is the same as the class is a constructor function.

4.1 Grid and group classes

```
class PCRC_Procs {

  // Constructors

  PCRC_Procs(const int rank, int shape []) ;


  // Inquiries

  PCRC_Dimension dim(const int d) ;
}
```

The constructor for the grid class is parametrized by the shape of the process grid. The total size of the grid should be less than or equal to the number of physical processes available. At most one logical process from a particular grid is mapped to each physical process. `PCRC_Dimension` is an auxilliary class in the abstract interface representing a single dimension of a grid. It's only role is to be passed to the range constructor: we omit further discussion except to say that the `d`th dimension is returned by the `dim(d)` member of the grid class. `d` is in the range `0, ..., rank - 1`.

```
class PCRC_Group {

  // Inquiries

  int PCRC_Group :: on() ;
}
```

The group class represents a process group. The member `on` returns 1 if the local physical process contains a logical process in the group, 0 if it does not.

[Add members for creating a group from a grid, and restricting dimensions of a group.]

4.2 The range class

```
class PCRC_Range {

  // Constructors

  PCRC_Range(const int extent,
             const int dist_code, PCRC_Dimension dim) ;

  PCRC_Range(const int extent) ;

  PCRC_Range subrng(const int extent,
                    const int stride, const int offset) ;


  // Subscripting

  PCRC_Range scalar(const int i) ;

  int loc(const int i) ;


  // Loop bounds

  void loop_bounds(int& count,
                   int& base, int& step,
                   int& baseSub, int& stepSub) ;

  void loop_bounds(int& count,
                   int& base, int& step,
                   int& baseSub, int& stepSub,
                   const int l, const int u, const int s,
                   const int a, const int b) ;

  void loop_coef(int& baseSub, int& stepSub,
                 const int l, const int u, const int s,
                 const int a, const int b,
                 const int shift_amount = 0) ;
}
```

The two proper constructor create distributed and collapsed primitive ranges. The member **subrng** returns a range representing a subrange of some parent range. The subrange is aligned to the parent with some stride and offset.

The member **scalar** returns a degenerate range structure representing a scalar subscript. A scalar range is a formal device whose only role is to be passed to the *section constructor* described later (this constructor requires range objects as its subscript arguments). Scalar ranges are *not* incorporated in array descriptors, and none of the normal member functions of the range class can be applied to a scalar range.

The member `loc` returns the local subscript associated with global subscript `i`.

The remaining operations are used to compute local bounds of parallel loops. `loop_bounds` returns the count of a local loop, the base and step of the *global* index in the local loop, and the base and step of the *local* scalar in the local loop. For example

```
      PROCESSORS p(4)

      INTEGER a(0 : 99)
!HPF$ DISTRIBUTE a(BLOCK)

      FORALL i = 0 : 99
        a(i) = i
```

could translate to

```
      PCRC_Procs p(4) ;

      int a [25] ;
      PCRC_Range i(100, PCRC_block, p.dim(0)) ;

      int count, base, step, baseSub, stepSub ;
      i.loop_bounds(count, base, step, baseSub, stepSub) ;

      for(int i = 0 ; i < count ; i++)
        a [baseSub + stepSub * i] = base + step * i
```

For illustration we have adopted C++ as the target language.

4.3 The array class

```
class PCRC_Array {

  // Constructors

  PCRC_Array(void* data, int element_type, int element_size,
             PCRC_Group grp,
             int rank, PCRC_Range rngs []) ;

  PCRC_Array(PCRC_Array parent, PCRC_Range rngs []) ;
```

```
  // Inquiries

  PCRC_Group grp() ;

  PCRC_Range rng(const int d) ;

  void* dat() ;

  int size() ;

  int str(int d) ;
}
```

The first constructor constructs an array from the components described above. The second constructor is the section constructor. The subscript vector contains range structures which represent either triplet subscripts (created using `PCRC_Range :: subrng`) or scalar subscripts (created using `PCRC_Range :: scalar`).

[Add examples of array and section construction.]

A A Tabular Reference Implementation

[Xiaoming's $7 \times N$ *table]*

B Adlib Reference Implementation

[Implementation on top of Adlib.]