Issues in Translation of High Performance Fortran

Bryan Carpenter
NPAC at Syracuse University
Syracuse, NY 13244
dbc@npac.syr.edu

January 2, 2000

Abstract

This lecture reviews some issues in the translation of an HPF program to a SPMD program. The need for a run-time array descriptor is discussed. An abstract model of a suitable Distributed Array Descriptor is developed. Illustrative translations of HPF fragments to Fortran and C++ are presented.

Contents

1 Introduction

Here is an exceedingly simple HPF program:

```
!HPF$ PROCESSORS P(4)

      REAL A(50)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P

      FORALL (I = 1:50)  A(I) = 1.0 * I
```

We want to convert this data-parallel program into a program that can be executed directly on a typical MIMD computer or cluster of workstations. We assume that the target machine provides a Fortran compiler and an MPI library. A reasonable translation is given in Figure 1. This looks a little complicated. We will break it down into its parts.

First we have a series of statements that simply set up the environment:

```
      INTEGER W_RANK, W_SIZE, ERRCODE
      ...

      MPI_INIT()

      MPI_COMM_RANK(MPI_COMM_WORLD, W_RANK, ERRCODE)
      MPI_COMM_SIZE(MPI_COMM_WORLD, W_SIZE, ERRCODE)
      ...

      MPI_FINALIZE()
```

So far as the present example is concerned, their only important role is to define the variables `W_RANK`, `W_SIZE`.

Secondly we have statements associated with the declaration of the distributed array:

```
      INTEGER BLK_SIZE
      PARAMETER (BLK_SIZE = (50 + 3) / 4)

      REAL A(BLK_SIZE)
```

These compute a bound on the number of elements of the distributed array held on any processor, and allocate a local array large enough the hold those elements. In the present case the size of the distributed array (50) and the number of processors over which it is distributed are both known at compile-time. So the block size, $\lceil 50/4 \rceil$, can be computed at compile-time, and the array segment can be declared statically.

The remainder of the code is associated with transation of the FORALL statement assigning `A`. There is a test to see if the local processor is a member of the processor arrangement `P` over which the elements of the assigment variable `A` are distributed:

```
INTEGER W_RANK, W_SIZE, ERRCODE

INTEGER BLK_SIZE
PARAMETER (BLK_SIZE = (50 + 3) / 4)

REAL A(BLK_SIZE)

INTEGER BLK_START, BLK_COUNT
INTEGER L, I

MPI_INIT()

MPI_COMM_RANK(MPI_COMM_WORLD, W_RANK, ERRCODE)
MPI_COMM_SIZE(MPI_COMM_WORLD, W_SIZE, ERRCODE)

IF (W_RANK < 4) THEN
    BLK_START = W_RANK * BLK_SIZE
    IF(50 - BLK_START >= BLK_SIZE) THEN
        BLK_COUNT = BLK_SIZE
    ELSEIF(50 - BLK_START > 0) THEN
        BLK_COUNT = 50 - BLK_START
    ELSE
        BLK_COUNT = 0
    ENDIF

    DO L = 1, BLK_COUNT
        I = BLK_START + L
        A(L) = 1.0 * I
    ENDDO
ENDIF

MPI_FINALIZE()
```

Figure 1: MPI translation of a simple HPF program.

```
IF (W_RANK < 4) THEN
    ...
ENDIF
```

We will always adopt the policy that an HPF processor arrangement may have any number of abstract processors up to the number of physical processors available at run-time (or conversely number of physical processors provided at run-time must be at least the size size of the largest processor arrangement declared in the program). This is not the only possible policy, but it is a reasonable one, and it is consistent with what the HPF standard says[1].

If the local processor *is* in `P`, we compute some parameters of the locally held subset of distributed array elements—the values `BLK_START` and `BLK_COUNT`. These characterize the position of the local segment in the global index space, and the number of elements in that segment:

```
INTEGER BLK_START, BLK_COUNT
...

    BLK_START = W_RANK * BLK_SIZE
    IF(50 - BLK_START >= BLK_SIZE) THEN
        BLK_COUNT = BLK_SIZE
    ELSEIF(50 - BLK_START > 0) THEN
        BLK_COUNT = 50 - BLK_START
    ELSE
        BLK_COUNT = 0
    ENDIF
```

Finally we have the actual loop over the local elements:

```
INTEGER L, I
...
    DO L = 1, BLK_COUNT
        I = BLK_START + L
        A(L) = 1.0 * I
    ENDDO
```

The global index is computed as a linear function of the local loop index `L`, and the assignment to the array element is performed.

Of course there was quite a lot of book-keeping in the MPI translation, but nothing conceptually difficult.

Now consider this superficially similar fragment of HPF:

```
      SUBROUTINE INIT(A)

      REAL A(50)
!HPF$ INHERIT A

      FORALL (I = 1:50)  A(I) = 1.0 * I
      END
```

[1] To be thorough we ought to associate a run-time check that `W_SIZE >= 4` with the declaration of `P`.

The INHERIT directive specifies that the mapping of the dummy argument `A` should be the same as the actual argument, whatever that is. At compile-time we have no information about that mapping, other than it is a legal HPF decomposition. If the main program was

```
!HPF$ PROCESSORS P(4)

      REAL B(50)
!HPF$ DISTRIBUTE B(BLOCK) ONTO P

      CALL INIT(B)
```

then the translated subroutine ought to behave exactly as in the previous example. But at the time `INIT` is translated this fact is generally not known. The subroutine may be part of a library compiled long before the calling program is written.

A few of the possibilities for mapping of the actual argument are illustrated in Figure 2. The first case has already been discussed. The array `A` will be divided into 4 contiguous blocks of sizes (13,13,13,11). In the second case the blocking is different—(13,13,12,12)—and the formula for computing the index value `I` inside the local loop will be quite different. The third case illustrates that `A` might actually be some section of an array. In this example the blocking is (13,12,13,12) and the translation must somehow take account that the subscripting into the local segment of the array is strided, and there is an offset which is sometimes zero and sometimes one. In the final case `A` is again a section, but now the parent array is two-dimensional, and we have to deal with the fact that `A` as a whole is mapped only to a portion of the processor arrangement. The physical processors associated with the bottom row of `Q` should do nothing.

Somehow the procedure `INIT` needs to be translated in such a way that it can deal with all these cases, and many others.

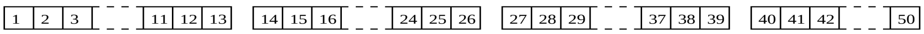
We have deliberately started with a very simple case. The parallel part of the program only deals with a single one-dimensional array, and it is clear that the translation requires absolutely no inter-processor communication. Nevertheless we have already stumbled into a whole wad of complexity associated with the translation of procedures.

This is not an artificial example. General purpose libraries for dealing with distributed arrays are likely to encounter just this kind of problem. As a specific example, the array communication libraries discussed in the next lecture have exactly this requirement: they must deal with distributed arrays that have unrestricted HPF mappings, unknown at the time the library code is compiled[2].

This lecture mainly discusses a run-time parameterization of HPF arrays that is designed to deal with this kind of problem.

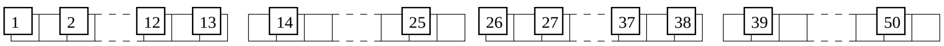
[2]The functions in such a library need not actually be *written* in HPF. Regardless, the author of the library must deal with the same problems as an HPF compiler translating procedures with INHERIT mappings.

```
!HPF$ PROCESSORS P(4)

      REAL B(50)
!HPF$ DISTRIBUTE B(BLOCK) ONTO P

      CALL INIT(B)
```




```
!HPF$ PROCESSORS P(4)

      REAL B(50)
!HPF$ DISTRIBUTE B(CYCLIC) ONTO P

      CALL INIT(B)
```


```
!HPF$ PROCESSORS P(4)

      REAL B(100)
!HPF$ DISTRIBUTE B(BLOCK) ONTO P

      CALL INIT(B(1:100:2))
```




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

      REAL B(6, 50)
!HPF$ DISTRIBUTE B(BLOCK, BLOCK) ONTO Q

      CALL INIT(B(2, :))
```

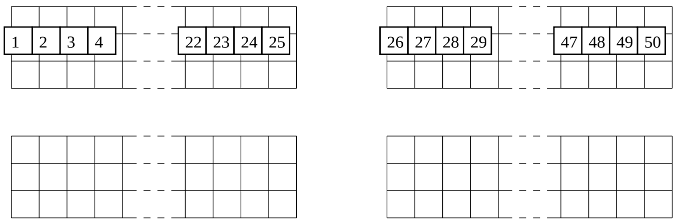


Figure 2: Mapping of `A` for some possible calls to `INIT`.

2 Requirements for an array descriptor

It seems that, in the translation of procedure calls like the one discussed in the last section, some non-trivial data structure must be passed to the translated procedure to describe the layout of the actual argument. This data structure is called a Distributed Array Descriptor or DAD. We wish to understand the requirements on the DAD, and how best to organize it. In view of the times, that organization should probably be informed by object-oriented principles.

One of the most obvious structural features of an HPF array is that it is a multi-dimensional entity. Each dimension can be mapped essentially independently by many different strategies including:

- collapsed (serial),
- simple block distribution,
- simple cyclic distribution,
- block cyclic distribution,
- general block distribution[3],
- through a linear alignment relation to some template dimension with *any* of the above distribution formats.

In the translation of Figure 1 there were various formulae and code fragments that were special to the choice of simple block distribution format. These included the formula:

```
BLK_SIZE = (N + P - 1) / P
```

for the size of locally allocated memory in terms of the array extent, N, and the extent, P, of the processor arrangement; the formula

```
BLK_START = R * BLK_SIZE
```

for the least global index associated with the local block in terms of the process coordinate R; the recipe

```
IF(N - BLK_START >= BLK_SIZE) THEN
    BLK_COUNT = BLK_SIZE
ELSEIF(N - BLK_START > 0) THEN
    BLK_COUNT = N - BLK_START
ELSE
    BLK_COUNT = 0
ENDIF
```

for the actual number of elements held locally[4], and the formula

[3] In HPF 2.0 this is a block distribution with an arbitrary pattern of local block-sizes, specified by some vector.

[4] The value assigned to BLK_COUNT could be expressed as:

$$\max(\min(\texttt{BLK\_SIZE}, N - \texttt{BLK\_START}), 0)$$

```
I = BLK_START + L
```

which computes the global index in terms of the local loop index L.

All of these prescriptions can differ for other mapping strategies. For example, if the array had been distributed in CYCLIC fashion the formula for BLK_START would be simply

```
BLK_START = R
```

the recipe for BLK_COUNT would be just

```
BLK_COUNT = (N - R + P - 1) / P
```

(ie, $\lceil (N - R)/P \rceil$) and the global index would be given by

```
I = BLK_START + (P * L)
```

These formulae may be checked against the second picture in Figure 2. If we are dealing with, say, block-cyclic distribution, or a dimension with strided alignment, the corresponding prescriptions get considerably more complicated.

The important thing is that we see evidence of a common set of operations that must be performed on different kinds of distributed array dimensions. The actual computations differ, depending on the distribution format and alignment of the individual dimensions. To the object-oriented programmer it looks as if a class hierarchy is emerging, with the common operations implemented as virtual functions. In the organization of the DAD described in this lecture this will be the `Range` hierarchy—a series of classes describing distributed index ranges. The details will be left to Section 4.

Apart from ranges, there is another important kind of subcomponent in our DAD. We can explain the need for this component by considering the last example in Figure 2, wich involved a rank-1 section of a rank-2 array.

In most respects this section should behave exactly like an ordinary rank-1 array—this is a requirement of the Fortran language, and a very convenient one. It means, for example, that sections can be freely passed to all the transformational intrinsic functions of Fortran. A reasonable expectation is that the DAD for the rank-1 section should contain just one `Range` object describing its single dimension. In the current example it would be most natural for that `Range` object to simply be a copy of the corresponding object in the parent DAD.

But the implementation of the section is not *exactly* like an ordinary one-dimensional array. It retains some vestige of the orthogonal dimension of its parent array. In particular, in the example, the section is only mapped to the top row of the processor arrangement `Q`. The translation of `INIT` must somehow take account of the possible existence of hidden "higher dimensions" of some parent array. If they exist, the translated code should only execute the assignments on processors in the appropriate row, or column—or more general slice—of the processor arrangement.

In fact a related problem was addressed in our first example, when we insisted on a test

```
IF (W_RANK < 4) THEN
    ...
ENDIF
```

to ensure that the local processor was a member of the processor arrangement over which elements of the assignment variable were distributed. The solution of the current problem is to generalize the idea of distributing an array over a processor arrangement, and explicitly allow that an array can be distributed over some *restricted slice* of a processor arrangement. A class will be introduced to represent this more general idea of a process group, and an object from this class will be added to the DAD. Then the translated code just needs to check that the local process is a member of the group, by a suitable method. As we will see in the next section, the required class has a compact and efficient implementation.

3 Groups

Before giving a detailed specification of the interesting `Range` hierarchy we need a more concrete language for discussing the processor arrangements over which index ranges will be distributed. We will also take the opportunity to discuss representation of DAD process groups in general.

We are familiar with the MPI idea of a process group, which is represented by a data structure of type `MPI_GROUP`. The groups referenced by a DAD are conceptually similar, but likely to be implemented in a different way. If we are going to create and manipulate DADs with the same freedom that typical data parallel programs create references to array sections, for example, the group object in the DAD better be very lightweight—highly streamlined.

3.1 The process grid

Any DAD group will have a parent processor arrangement. The preceding discussion about the DAD was supposed to be at a logical, design level. But from now on we will need to become more concrete. So we give specific interfaces in C++[5]. A C++ run-time representation of an HPF processor arrangement would naturally be implemented as an object (just as a group or communicator is logically an object in MPI). We can easily see that the HPF directive

```
!HPF$ PROCESSORS P(4)
```

might map to a C++ declaration something like

```
Procs1 p(4);
```

and

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

[5]Of course one can do equivalent things in Fortran, but they look relatively clumsy and difficult to read.

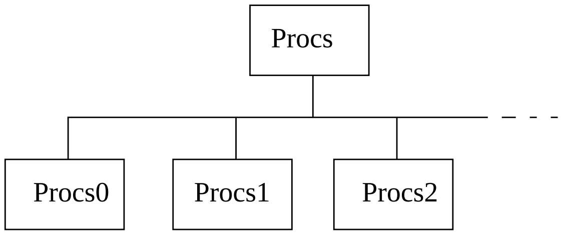


Figure 3: A hierarchy of process grids.

```
class Procs {
public:
    int member() const ;

    Dimension dim(const int d) const ;
    ...
} ;

class Dimension {
public:
    int size() const ;

    int crd() const ;
    ...
} ;
```

Figure 4: Interfaces of `Procs` and `Dimension`.

might map to a C++ declaration something like

```
Procs2 q(2, 2);
```

This supposes that `Procs1` is a class of one-dimensional process grids and `Procs2` is a class of two-dimensional grids[6]. We will generalize to a family (class hierarchy) of process grids illustrated in Figure 3.

What methods are needed on the `Procs` class? Looking back at Figure 1 we see that we need the ability to check if the local process is a member of the process grid in question, and we need to find the coordinates of the local process relative to the grid. Later we will find that it is useful to have a separate concept of *process dimension*—one dimension of a process grid. We introduce another class—`Dimension`—to stand for this concept. Finding the coordinate becomes a method on the dimension class. Concrete interfaces for `Procs` and `Dimension` are given in Figure 4.

[6]In MPI groups themselves do not have any grid structure. On the other hand a *Cartesian communicator* does have a grid structure. So one possible implementation of a `Procs` object might be in terms of this kind of communicator.

If we decided we wanted to use these primitives in a C++ version of our original translation example, the lines

```
INTEGER W_RANK, ...
...

MPI_COMM_RANK(MPI_COMM_WORLD, W_RANK, ERRCODE)
...

IF (W_RANK < 4) THEN
    BLK_START = W_RANK * BLK_SIZE
    ...
ENDIF
```

could be replaced with something like

```
Procs1 p(4);
...

if (p.member()) {
    blk_start = p.dim(0).crd() * blk_size ;
    ...
}
```

A `Procs` object is probably not particularly "lightweight". It is likely to contain a vector of `Dimension` objects and probably some vector defining how logical processes are mapped to physical processes, and so on. Most likely a DAD should not contain a *copy* of this information, but some *reference* to an externally created `Procs` object.

3.2 Restricted process groups

At the end of Section 2 we saw that in fact the DAD should be able to reference a *restricted slice* of a process grid. Specifically what is needed is the ability to reference a portion of a process grid selected by fixing the coordinates in any subset of the dimensions to single values. Figure 5 gives some examples.

Luckily there is a very compact way to represent this kind of subgroup. If we call the horizontal dimension in the figure `dim(0)` and the vertical dimension `dim(1)`, then **a)** represents the whole of the grid and has dimension set {`dim(0)`,`dim(1)`}, **b)** has dimension set {`dim(0)`}, **c)** has dimension set {`dim(1)`}, and **d)** has the empty dimension set {}. With the dimension set given, a group can be uniquely specified by its lead process—the process with coordinates 0 relative to the effective dimensions. In our examples **a)** has lead process 0, **b)** has lead process 8, **c)** has lead process 1, and **d)** has lead process 6.

The dimension set can be represented as a subset of the parent grid dimensions using a bitmask. Since any reasonable grid will have less than 32 dimensions, this bitmask will always fit in a single word. So we see that our 4 examples can be parametrized by the tuples $(11_2, 0)$, $(10_2, 8)$, $(01_2, 1)$ and

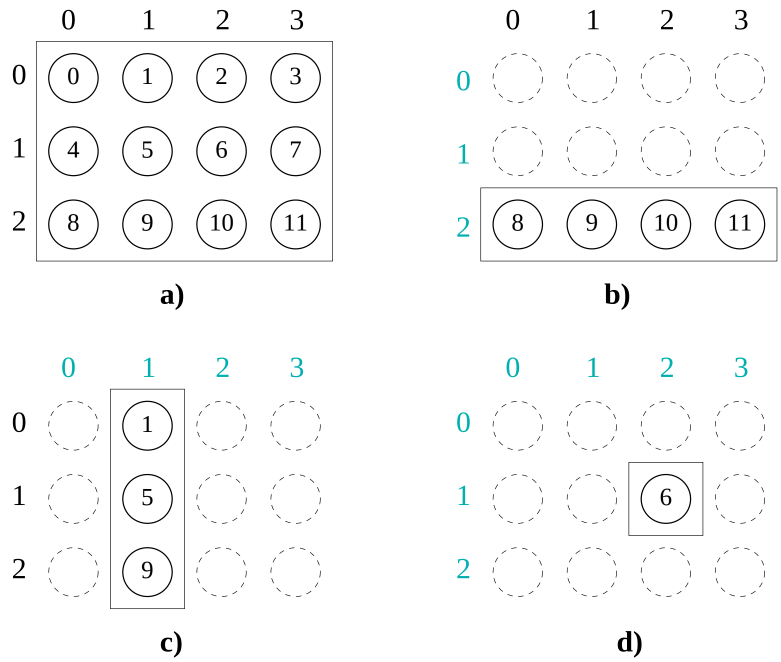


Figure 5: Example restricted groups in a 2-dimensional process grid

$(00_2, 6)$, respectively, together with a pointer to the `Procs` object representing the parent grid.

The `Group` class will have an interface like

```
class Group {
public:
    Group(const Procs& p) ;

    void restrict(Dimension d, const int coord) ;

    int member() const ;
    ...
} ;
```

and it can be implemented in about 3 words of memory. So `Group` objects can be freely copied and discarded: they fit naturally in the DAD. The conversion constructor provides a user-defined type conversion from a process grid object to a group object representing the whole of the grid. The `restrict()` method restricts the group in one of its dimensions.

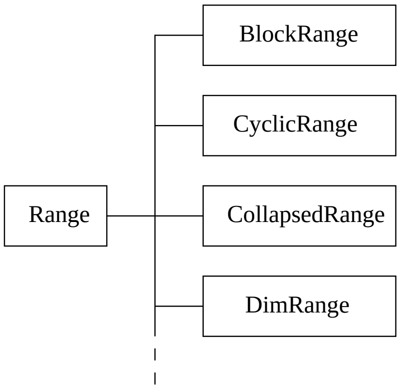


Figure 6: A hierarchy of ranges.

4 Ranges

Inside our DAD a range object will describe the extent and distribution format of one dimension of a distributed array. We can expect a class hierarchy of different kinds of range object (Figure 6). Each subclass represents a different kind of distribution format for an array dimension. The simplest distribution format is *collapsed* (sequential) format. Other distribution formats include *regular block* and *simple cyclic* decomposition. We will see later that some range classes allow *ghost extensions* to support stencil-based computations.

Figure 7 gives a possible interface of a `Range` class in C++. To understand this API, let's give an example of its use. Our original HPF example and a possible translation to C++ as shown in Figure 8. We declare a process grid `p` as described in the previous section, then create a `Range` object, `x`, of subclass `BlockRange`. The arguments of the constructor are the process dimension over which the range is to be distributed, and the extent of the range. The value previously called `BLK_SIZE` is returned by the method call `x.volume()`, and a local array segment is allocated with this size. In any process that is a member of `p` we call the method `block`, passing it the local process coordinate. This initializes a `Block` structure which has fields:

```
struct Block {
    int count ;

    int glb_bas ;
    int glb_stp ;

    int sub_bas ;
    int sub_stp ;
} ;
```

The fields define the count of the local loop and the base and step for the local subscript and global index[7].

[7]In the translation of Figure 8, note that the expressions `b.glb_bas + b.glb_stp * l` and

```
class Range {
public:
    int size() const ;

    Dimension dim() const ;

    int volume() const ;

    Range subrng(const int extent, const int base,
                 const int stride = 1) const ;

    void block(Block* blk, const int crd) const ;

    void location(Location* loc, const int glb) const ;
    ...
} ;
```

Figure 7: Interface of the `Range` class.

Figure 8 is shorter and (arguably) more readable than the original translation in Figure 1. It also has the interesting property that if we change the distribution directive in the HPF code to

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

the only change needed in the C++ translation is that the declaration of `x` becomes

```
CyclicRange x(p.dim(0), 50);
```

This suggests we are making progress towards the goal of being able to produce generic code that works for data with any distribution format.

`b.sub_bas + b.sub_stp * l` are "induction variables"—linear functions of the loop index. The compiler can compute these efficiently by constantly incrementing temporary variables.

SOURCE:

```
!HPF$ PROCESSORS P(4)

      REAL A(50)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P

      FORALL (I = 1:50)  A(I) = 1.0 * I
```

TRANSLATION:

```
... do some global initialization

Procs1 p(4);

BlockRange x(p.dim(0), 50);

float* a = new float [x.volume()];

if (p.member()) {
    Block b;
    x.block(&b, p.dim(0).crd());

    for(int l = 0 ; l < b.count ; l++) {
        const int i = b.glb_bas + b.glb_stp * l + 1 ;
        a [b.sub_bas + b.sub_stp * l] = 1.0 * i ;
    }
}

... do some global finalization
```

Figure 8: C++ translation of simple HPF program.

5 A DAD

Before ranges and groups can be put together to make a DAD we need to deal with one other detail. This is the issue of *memory strides.*

An array dummy argument may be matched with an array *section* in the calling program. So for a dummy the layout of elements in memory may be different to what would be expected for a general Fortran array. This problem is not specific to HPF—it is a feature of Fortran 90. In:

```
REAL D(100, 100)
...
CALL FOO(D(1, :))


SUBROUTINE FOO(C)
REAL C(:)
...
END
```

the first dimension of `D` is most-rapidly-varying, so the stride in that dimension—the memory displacement between adjacent elements—is 1 word. The second dimension has a stride of 100 words. So the stride for the dummy argument `C` will also be 100. Fortran compilers typically take this into account by passing a *dope vector* for an array argument. For a rank-r argument the dope vector contains the r extents and the r strides that describe the shape of the array and its memory layout [2]. The procedure code uses the stride information from the dope vector when resolving array references.

In our DAD the extent information fits naturally in the `Range` objects. We find it convenient to keep the memory stride information separate. It seems useful to allow range objects to be shared by arrays that have the same mapping in particular dimensions, although they may have different overall shape and memory layout.

The upshot is that the DAD for a rank-r array will contain:

- A distribution group,
- r range objects, and
- r integer strides

Figure 9 gives an interface for the `DAD` class. Here `Map` is a small structure that bundles a range with a stride:

```
struct Map {
public :
    Map(Range _range, const int _stride) ;

    Range range ;
    int stride ;
} ;
```

```
struct DAD {
    DAD(const int _rank, const Group& _group, Map _maps []) ;

    int rnk() const {return rank ; }

    const Group &grp() const ;

    Range rng(const int r) const ;

    int str(const int r) const ;
    ...
} ;
```

Figure 9: Interface of the DAD class.

With this DAD, we can finally give a translation of the second example in section 1. Source and translation is given in Figure 10. To fully appreciate how this translation works, we need to see how the calling program sets up the DAD. For this we revist the four examples in Figure 2. Sources and translations are given in Figures 11 through 14.

The translations in Figures 11 and 12 are fairly self-explanatory. A DAD is set up describing array `b`. This contains a group object describing `p` (the conversion from `Procs` to `Group` is implicitly used) and and a map object containing the range `x` and stride 1. A pointer to the DAD is passed to the procedure, together with the pointer to the elements.

Figure 13 is a little more complicated. We again create a range parameterizing the 100-element array `b`. In mechanically translated code we might go on to set up a DAD describing the whole of `B`, but for simplicity we skip this stage here. We only set up a DAD describing the strided section of `b`. This involves using the `subrng()` method on `Range` to create a new range object representing a subrange. The subrange has stride 2. When the the procedure `init()` calls the `block()` method on this object, it will return appropriately adjusted values for subscript and global index bases and steps.

Figure 14 is more complicated again. To deal with the scalar subscript it uses the `location()` method. This takes the global subscript as argument and returns a `Location` structure, which has fields:

```
struct Location {
  int crd ;
  int sub ;
  ...
} ;
```

These define the corresponding process coordinate and local subscript. In some sense `location()` is the dual method to `block()`—`block()` takes a process coordinate and returns a range of indices (and local subscripts)—`location` takes an index and returns a coordinate (and a local subscript)[8]. Note the memory stride for the second dimension, `x.volume()`, goes into the `Map` object, and the base address passed to the procedure is offset in memory by an amount determined by the scalar subscript. Of course the group in the DAD was suitably restricted.

References

[1] Bryan Carpenter, Guansong Zhang, and Yuhong Wen. NPAC PCRC runtime kernel definition. Technical report, Northeast Parallel Architectures Center, 1997. http://www.npac.syr.edu/projects/pcrc/kernel.html.

[2] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[8] We assumed the argument of `location()` is a C-style index, starting from 0, rather than a Fortran-style index starting from 1.

SOURCE:

```
      SUBROUTINE INIT(A)

      REAL A(50)
!HPF$ INHERIT A

      FORALL (I = 1:50)  A(I) = 1.0 * I
      END
```

TRANSLATION:

```
void init(float* a, DAD* a_dad) {

    Group p = a_dad->grp();
    if (p.member()) {
        Range x = a_dad->rng(0);
        int   s = a_dad->str(0);

        Block b;
        x.block(&b, x.dim().crd());

        for(int l = 0 ; l < b.count ; l++) {
            const int i = b.glb_bas + b.glb_stp * l + 1 ;
            a [s * (b.sub_bas + b.sub_stp * l)] = 1.0 * i ;
        }
    }
}
```

Figure 10: C++ translation of HPF program with inheritted mapping.

SOURCE:

```
!HPF$ PROCESSORS P(4)

      REAL B(50)
!HPF$ DISTRIBUTE B(BLOCK) ONTO P

      CALL INIT(B)
```

TRANSLATION:

```
... do some global initialization

Procs1 p(4);

BlockRange x(p.dim(0), 50);

float* b = new float [x.volume()];

Map maps [1];
maps [0] = Map(x, 1);

DAD dad(1, p, maps);

init(b, &dad);

... do some global finalization
```

Figure 11: Translation of first example from Figure 2.

SOURCE:

```
!HPF$ PROCESSORS P(4)

      REAL B(50)
!HPF$ DISTRIBUTE B(CYCLIC) ONTO P

      CALL INIT(B)
```

TRANSLATION:

```
... do some global initialization

Procs1 p(4);

CyclicRange x(p.dim(0), 50);

float* b = new float [x.volume()];

Map maps [1];
maps [0] = Map(x, 1);

DAD dad(1, p, maps);

init(b, &dad);

... do some global finalization
```

Figure 12: Translation of second example from Figure 2.

SOURCE:

```
!HPF$ PROCESSORS P(4)

      REAL B(100)
!HPF$ DISTRIBUTE B(BLOCK) ONTO P

      CALL INIT(B(1:100:2))
```

TRANSLATION:

```
... do some global initialization

Procs1 p(4);

BlockRange x(p.dim(0), 100);

float* b = new float [x.volume()];

// Create DAD for section b(::2)

Range x2 = x.subrng(50, 0, 2);

Map maps [1];
maps [0] = Map(x2, 1);

DAD dad(1, p, maps);

init(b, &dad);

... do some global finalization
```

Figure 13: Translation of third example from Figure 2.

SOURCE:

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

      REAL B(6, 50)
!HPF$ DISTRIBUTE B(BLOCK, BLOCK) ONTO Q

      CALL INIT(B(2, :))
```

TRANSLATION:

```
... do some global initialization

Procs2 q(2, 2);

BlockRange x(q.dim(0), 6),
           y(q.dim(1), 50);

float* b = new float [x.volume() * y.volume()];

// Create DAD for section b(1, :)

Location i;
x.location(&i, 1);

Group p = q;
p.restrict(q.dim(0), i.crd)

Map maps [1];
maps [0] = Map(y, x.volume());

DAD dad(1, p, maps);

init(b + i.sub, &dad);

... do some global finalization
```

Figure 14: Translation of fourth example from Figure 2.

A Other features of the Adlib DAD

The DAD described in this lecture was a simplified version of the Adlib descriptor developed at Syracuse during the PCRC project [1]. In fact we strictly followed the current version of the Adlib spec, and the C++ examples are all executable using Adlib. But in the interests of simplifying the presentation many fields and methods were omitted. In this appendix we will briefly summarize some of the more important things that were missed out.

A.1 Support for block cyclic distributions

The approach to translating parallel loops given in this lecture is too simple to deal with block-cyclic distribution. In that case the local loop needs an outer loop enumerating the set of locally-held blocks. Adlib provides some "iterator" classes (`LocBlocksIndex`, etc) for enumerating these blocks. It also has an extra field in the `Location` class to specify the block. An `offset()` method can be used to compute the total memory offet implied by a `Location` object, taking account of this generalized case.

A.2 Support for ghost extensions and other memory layouts

Adlib supports ghost extensions on suitable ranges. However the local subscript field computed by `block()` or `location()` does not incorporate the associated shifts in base address. Also Adlib supports a packed layout for arrays with strided alignment, but again the local subscript field is not divided by the packing factor. Instead local subscripts or `Location` objects must be converted by methods called `disp()`, `offset()` to allow for these layout options. Steps in local subscripts must be converted by a `step()` method. The need for the extra calls is slightly unfortunate, but the advantages of having a universal definition of the local subscript that is independent of these details of the memory layout seem to warrant them.

A.3 Support for loops over subranges

The `block()` method has variants that take triplet parameters to efficiently enumerate local blocks of a *sub*range. This is an important optimization—more efficient than creating a subrange *object* and using the simple `block()` method on that.

Certain subranges only have blocks defined for a subset of coordinate values, and there are methods to compute these ranges of coordinate values. As a matter of fact it is illegal to call `block()` for coordinate values outside these ranges—a trap for the unwary, but a restriction that improves the efficiency of the implementation.

A.4 Features to support communication libraries

The components of the Adlib DAD have accumulated various other methods to support the communication library. Associated infrastructure includes iterator classes (`AllBlocksIndex`) for iterating over *all* blocks of a range (not just the ones in a single process).

A.5 Miscellaneous inquiries and predicates

There are inquiry functions to discover many properties of ranges, groups and DADs, and various predicate methods desighed for run-time checking for correctness of programs.