Adlib: A Distributed Array Library to Support HPF Translation

D.B. Carpenter

High Performance Computing Centre,
University of Southampton,
Southampton, S09 5NH

DRAFT

Abstract

Adlib is a class library implementing the HPF distributed data model. Its native interface is in C++. A Fortran interface provides a subset of the Adlib functionality. This paper mainly discusses the C++ interface and implementation.

1 Introduction

The *Adlib* communication library is a component of the Southampton HPF translation system. It combines ideas from the ADAPT system for translation of Distributed Fortran 90, and from earlier collective communication libraries developed at Southampton. Its features include

- Extensive use of abstract data types used to describe process arrays, distributed data, distributed control, and collective communication in an integrated way.
- A communication functionality adequate to fully support translation of distributed Fortran.
- Data distribution formats to support full HPF.

Adlib is implemented as a C++ class library. There is a reduced interface to Fortran.

2 Process Arrays

The HPF declaration

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

introduces a 4 by 4 arrangment of virtual processors, over which program data can be distributed. In Adlib the role of the HPF processor arrangement is taken by the *process array*. A rank 2 process array, equivalent to `P`, is declared by

```
Procs2 p(4, 4) ;
```

An Adlib process array has a set of *process dimensions* of class `Dimension`. The individual dimensions of a process array are accessed by the member function `dim`, illustrated in the next section.

A `Procs` array describes *virtual processes*, which are mapped (somehow) into the available set of physical processes. In the remainder of this paper we will occasionally be cavalier about the distinction between virtual and physical processes; if the library in used as intended the distinction should not be significant.

3 Ranges

A central idea in Adlib is the *range*. There is no completely analogous object in HPF. A range is a one-dimensional index space. A range of extent n has the values $0 \ldots n-1$ associated with it. The range defines a mapping from this interval into a process dimension.

A *primitive range* is distributed over a process dimension, according to one of the HPF distribution formats. The statements

```
Range x(100, p.dim(1), BLK) ;
Range y(200, p.dim(2), CYC, 10) ;
Range z(50) ;
```

declare that `x` has extent 100, and is distributed block-wise over the first dimension of `p`; `y` has extent 200 and is distributed block-cyclically over the second dimension of `p`. `z` is a "collapsed" range of extent 50. An Adlib primitive range is directly analogous to a single dimension of an HPF template.

A *derived range* is a range linearly aligned to some primitive range. The statements

```
Range i(20, 0, 5, x) ;
Range j(100, 10, 1, y) ;
Range k(50, 74, -1, j) ;
```

declare that `i` has extent 20; it is aligned to `x` starting at position 0 in `x`, with stride 5—that is, element 0 of `i` has the some process mapping as element 0 of x, element 1 of i has the same mapping as element 5 of x, etc. j has extent 100; it is aligned to y starting at postion 10 with stride 1. k has extent 50; it is aligned to y starting at position 84 with stride -1 (the immediate parent of k is j but the two alignments are composed to yield the mapping to the primitive range—two derived ranges are equivalent if they have to same extent and the same alignment to equivalent primitive ranges, however they are derived).

The extent of a range can be determined by the member function size. Its interface is

```
int Range :: size() ;
```

4 Embeddings and process groups

An *embedding* is similar to a derived range, but is "scalar". It is aligned with a single element of some range.

```
Embed e(60, x) ;
Embed f(20, k) ;
```

declare that e is aligned to element 60 of the range x. f is aligned to element 20 of the range k. k has primitive range y, so f is an embedding in y, at element 64.

A *process group* is a set of (virtual) processes. The process group has two röles—a particular action can be performed within a specified process group, or a particular data object can be distributed over a specified process group. The simplest example of a process group is a process array. Adlib has a more general idea of a process group, which is a "slice" of a process array formed by restricting one or more of the process dimensions to a single coordinate. The Adlib interface doesn't provide a direct means of specifying a process coordinate—a coordinate is defined implicitly by an embedding.

The operator / is overloaded to construct a new process group (class ProcessGroup) from an existing one and an embedding.

```
ProcessGroup q(p / e) ;
ProcessGroup r(q / f) ;
ProcessGroup s(p / e / f) ;
```

The process groups q, r and s are illustrated in figure 1. The *parent process array* of all these groups is p. A process group has a set of *effective process dimensions* taken from the parent process array. The effective dimension set for p contains p.dim(1) and p.dim(2). The effective dimension set for q contains just p.dim(2), while that for r contains just p.dim(1). The set for s is empty.

At any stage of execution of a program a particular process group is singled out as the *active process group*. From the viewpoint of a particular process, the active process group is the set of processes sharing the current "thread of

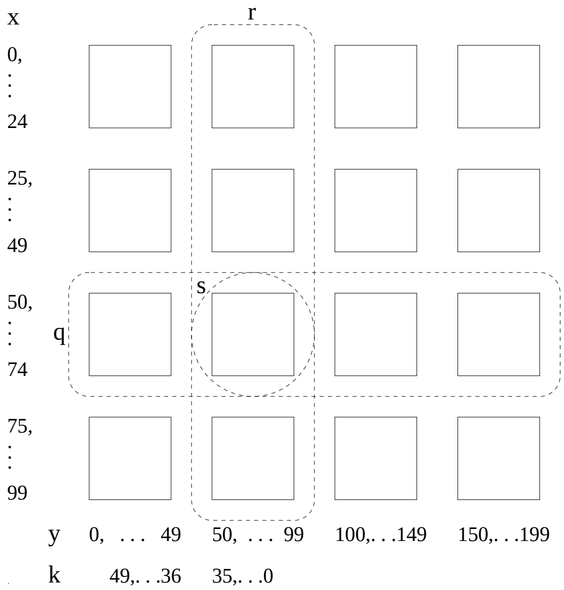

Figure 1: Examples of process groups. The square boxes represent the 16 virtual processes in the process array p. The dashed lines embrace groups q, r and s.

control". Adlib supports a form of MIMD execution (see section 7) so there may be several distinct processes groups active at the same time, with the physical processes partitioned between these groups.

A process group `q` with parent array `p` is *accessible* at a particular stage of program execution if the active process group is the same as the one in effect when `p` was created, or if it is a subarray of `p` which contains `q`.

5 Arrays

An *array* is distributed over a particular process group. The shape and mapping of an array of rank r are defined by list of r ranges. Each of these ranges must be distributed over a dimension in the effective dimension set of the array's process group, or they must be collapsed; no two of them may be distributed over the same dimension.

```
Array2<float> a(p, x, y) ;
Array1<int> b(q, k) ;
Array1<float> c(z) ;
```

The (optional) first argument of the constructor is the process group over which to distribute the array. If it is omitted, as in the declaration of `c`, the array is distributed over the active process group.

According to these declarations `a` is a rank 2 array of floats, shape `(100, 200)`, distributed over the whole of the process array `p`. `b` and `c` are rank 1 arrays of integers and floats respectively, shape `(50)`. `b` is embedded in the process array `p` at `e` (assuming the earlier definition of `q`). `c` has a collapsed range and is replicated over the active process group. (If no range of an array is distributed over an effective dimension of its process group, the array is replicated over that dimension. `c` has no distributed ranges so it must be replicated over all dimensions of the active process group.)

Creation of an array is a collective operation. The process group over which the array is distributed must be accessible, and all members of the active process group must cooperate in the construction.

Adlib arrays can be subscripted by objects from the `Subscript` class, which is a common base class of the `Range` and `Embed` classes.

There are two ways to subscript an array. *Simple subscripting* is achieved with the overloaded `()` operator. This operation returns a reference to an array element. For now we will only illustrate the case where the arguments are of type `Embed`—section 7 describes situations in which a `Range` can be used as a simple subscript.

```
b(f) = 17 ;
a(e, f) = 2 * b(f) ;
```

The embeddings used as subscripts must be in the same primitive ranges as those of the corresponding ranges of the array, and the selected element is the one aligned to the primitive ranges in the same way as the subscripts. If a range of the array is a derived range with no element at the specified point, the operation is ill-defined. Also, the simple subscript operation is ill-defined if executed on a process which isn't home to all embeddings used as arguments—the the fragment above could only be executed on the single process in group `s` (see figure 1). To put it another way, the active process group must be `s`.

The user has to exercise some care when updating array elements. It is important to ensure that elements which are meant replicated over a process group (according the mapping implied by the array declaration) are updated with a consistent values. Provided the active process group is only manipulated using the constructs introduced in sections 6 and 7, it is easy enough to devise a set of variable usage conventions which will ensure this requirement is met—these conventions will be omitted here.

The other way to subscript an array is by means of the section constructor

```
Section1<float> d(a, sect(e, k)) ;
```

declares `d` to be a section of `a`. The subscripts are the embedding `e`, and the range `k`. The rules for embedding-valued subscripts are similar to those for simple subscripting, except that in this case there is no restriction about only performing the operation at the home of the embeddings—construction of a section is a collective operation. A range-valued subscript must be representable as a range derived from the corresponding array range. In the example, `d` represents a slice of the array `a`—it provides an alias for the data in that slice. The shape and mapping of `d` is identical to that of `b`.

The process group over which a section is distributed is defined to be the process group over which the subscripted array was distributed, restricted by any embedding-valued subscripts. This process group must be accessible when the section is created.

Neither mechanism for subscripting requires any transfer of data between processes. If Adlib is used as the target of HPF translation, Fortran subscripting operations which *do* require data transfer must make explicit calls to the `remap` or gather-scatter primitives described in section 8.

The array classes and the section classes are derived from a common base class, `Frame`, and have identical interfaces. The `Frame` class also provides various inquiry functions: for example `prc` returns the process group over which the array is distributed, and `rng` returns a specified range of the array.

```
ProcessGroup* Frame :: prc() ;
Range* Frame :: rng(const int axis) ;
```

6 Embedding control: *on* and *at*

The library provides three operations which can be used to change the active process group. This section describes the first two—*on* and *at*.

As explained above, at a particular point of program execution, the active process group is the process group sharing the current "thread of control". Various operations in the Adlib communication library rely on knowing the active process group—for example some operations must synchronise all processes in this group, and others must broadcast information to all processes in the group. In general it is important that the active process group be changed in a disciplined way, using the Adlib primitives provided. These maintain a private data structure describing this group.

When an Adlib program starts executing the active process group is *world*, a rank-1 process array containing one virtual process for each physical process on which the program has been invoked.

The *on* construct is used to change the active process group. Suppose `q` is an accessible process group, then

```
if(q.on()) {
  ...
} q.no() ;
```

After the call to `q.on()` the active process array is `q`. `q.on()` returns a truth value which specifies whether any virtual process in `q` is held on the local physical process. Immediately before the call the `q.no()` the process group must be `q` (although it may have changed in the interim); after the call it is restored to whatever it was before the earlier matching call to `q.on()`.

The *at* construct is used to embed control within a dimension of the active process array. For example, if the active process group is currently `p`, the fragment of code in the previous section could be executed within a pair of nested *at* constructs:

```
if(e.at()) {
  if(f.at()) {
    b(f) = 17 ;
    a(e, f) = 2 * b(f) ;
  } f.ta() ;
} e.ta() ;
```

Between the calls to `e.at()` and `e.ta()` the active process array is `p / e`. `e.at()` returns a truth value which specifies whether any virtual process in the new active process group is held by the local physical process.

There is a restriction on where the *at* construct can appear: `e.at()` can only be executed when the dimension in which `e` is an embedding in an effective dimension of the active process group.

If the active process group is initially `q`, the effect of $at(e)$ is identical to $on(q/e)$, but using `at` is simpler.

7 Distributing control: *where*

If `at` is like a conditional, `where` is like a loop. Instead of an embedding it is parametrised by a range. Suppose the active process group is currently `q`, then:

```
for(k.where() ; k.test() ; k.next()) {
  b(k) = k() ;
} k.erewh() ;
```

`k.where()` can only be executed when the dimension over which `k` is distributed is an effective dimension of the active process group. Between the calls to `k.where()` and `k.erewh()` the active process group is reduced by restricting the process dimension associated with `k` to the locally held coordinate. The loop body executes once for every index value of `k` held on the local process.

The fragment above introduces two other features. In section 5 it was explained how an embedding could be used as a simple subscript. A range can also be used as a simple subscript when (and only when) it has been activated as a `where` iterator. In this context a range used as a subscript behaves exactly like an embedding in the range, at the point corresponding to the current iteration of the loop.

The other new feature is that, also when a range has been activated as an iterator, the `()` operator can be applied to it to find the current index value, as on the right hand side of the assignment.

8 Communication

There are three families of communication operations in Adlib

8.1 The *remap* family

The `remap` operation copies an array to another array of the same shape. Its interface is

```
void remap(Frame& dst, Frame& src) ;
```

The mappings of the source and destination arrays are arbitrary, but in common with all the communication routines described in this paper, `remap` is a collective operation. All array-valued arguments must be distributed over process groups which are accessible at the point of the call.

We will now give a small example of an Adlib program. The HPF program in figure 2 is a toy example of the nearest neighbour update paradigm. The Adlib program in 3 is an equivalent Adlib program. Construction of the array sections and declaration of temporaries has been lifted out of the loop over `n`

The *remap* family potentially contains other operations which can be expressed in terms of `remap`, but may be more efficiently implemented directly. These include broadcasts, shifts and matrix transpose.

8.2 The *gather-scatter* family

The general `gather` operations have interfaces

```
void gather(Frame& b, Frame& a, Frame* i_1) ;

void gather(Frame& b, Frame& a, Frame* i_1, Frame* i_2) ;

void gather(Frame& b, Frame& a, Frame* i_1, Frame* i_2,
                                            Frame* i_3) ;
...
```

Here `b` is the *gathered data* and `a` is the *scattered data*. These are followed by subscript arrays, which are integer arrays with the same shape and mapping as the gathered data. The number of subscripts should be equal to the rank of the scattered data, and their values should lie in the ranges expected by subscripts of that array. Data is copied from `a` to `b` with the specified subscripting pattern. A completely analogous primitive `scatter` copies data from `b` to `a`.

```
real u(10)

!hpf$ distribute u (block) onto p
!hpf$ processors p (4)

integer i, n

do i = 1, 10
  if(i .eq. 1 .or. i .eq. 10) then
    u(i) = 1.0
  else
    u(i) = 0.0
  endif
enddo

do n = 1, 5
  u(2 : 9) = 0.5 * (u(1 : 8) + u(3 : 10))
enddo
```

Figure 2: Simple nearest neighbour update in HPF.

These operations are unnecesarily general if it is only required to implement the patterns of access in Fortran 90 array syntax (but they are needed for parallelising `FORALL`, for example). Vector subscripts can be handled by the simpler primitives

```
void vecGather(Frame& b, Frame& a, Frame* i_1) ;

void vecGather(Frame& b, Frame& a, Frame* i_1, Frame* i_2) ;

void vecGather(Frame& b, Frame& a, Frame* i_1, Frame* i_2,
                                               Frame* i_3) ;
...
```

where now `a` and `b` have the same rank, and the subscripts are rank-1 arrays, whose individual ranges are the same as the corresponding to range of `b`. If any subscript argument is `NULL` it is assumed that the gathered and scattered data have the same extent in the associated dimension, and the operation behaves like an array assignment with respect to that dimension. `vecScatter` is similar.

The *gather-scatter* family is related superficially to the *remap* family, but differs from it in a fairly deep way at the implementation level. Simple remapping operations can be implemented efficiently in terms of direct message-passing;

```
Procs1 p(4) ;

if(p.on()) {
  Range x(10, p.dim(1), BLK) ;

  Array1<float> u(x) ;

  for(x.where() ; x.test() ; x.next()) {
    if(x() == 0 || x() == 9)
      u(x) = 1.0 ;
    else
      u(x) = 0.0 ;
  } x.erewh() ;

  Range l(8, 0, 1, x), r(8, 2, 1, x) ;

  Section1<float> u_l(u, sect(l)), u_r(u, sect(r)) ;

  Range c(8, 1, 1, x) ;

  Array1<float> tmp1(c), tmp2(c) ;

  for(int n = 1 ; n <= 5 ; n++) {
    remap(tmp1, u_l) ;
    remap(tmp2, u_r) ;

    for(c.where() ; c.test() ; c.next()) {
      u(c) = 0.5 * (tmp1(c) + tmp2(c)) ;
    } c.erewh() ;
  }
} p.no() ;
```

Figure 3: Simple nearest neighbour update in Adlib.

gather-scatter operations typically need data request protocols and bulk synchronisation.

8.3 The reduction family

The simplest reduction primitive has the inteface

```
void reduce(const int typeLen, char* res, Frame* src,
            void (*opZero)(void*),
            void (*opAccum)(void*, void*)) ;
```

The cast to a character vector of length `typeLen`, is replicated over the active process group. The array `src` holds the input data; the accumulation operation is defined by the function `opAccum`, and the function `opZero` sets data of the return type to the zero of the accumulation operation.

Variants do reductions on pairs of arrays (scalar product operations and masked reductions) and reductions in single array dimensions, where the result is an array with rank reduced by one.

9 Miscellany

9.1 Lower level interfaces

The interface described so far is probably at too high a level to be suitable as a target of compilation. The expectation is that an intelligent compiler would in-line the implementations of the less efficient operations, in terms lower level interfaces.

As an example, the implementation given for the *where* construct is fairly inefficient. The iterator functions are quite complex, and calling the subscript member function on the arrays in every iteration is wasteful, when, in practise, data pointers are simply being incremented in most iterations.

Lower level iterator functions are provided which just enumerate the set of individual contiguous index blocks held on the local physical process, and lower level inquiry functions on arrays provide direct access to the data layout—base addresses and strides. For example, the where construct given in section 7

```
for(k.where() ; k.test() ; k.next()) {
  b(k) = k() ;
} k.erewh() ;
```

can be translated more efficiently by

```
char*  b_base = b.dat() ;
Stride* b_str = b.str(1) ;
```

```
for(k.locBlocksEnum() ; k.locBlocksTest() ; k.locBlocksNext()) {
  int count = k.blockCount() ;
  int k_val = k() ;

  char* b_data = b_base + b_str->offset(&k) ;
  int   b_step = b_str->step(&k) ;

  for(int i = 0 ; i < count ; i++) {
    *((int*) b_data) = k_val ;

    b_data += b_step ;
    k_val++ ;
  }
}
```

(Other lower level-functions directly manipulate the active process group data structure; these calls were "optimised out" in the example above, because there are no communication operations inside the *where* construct which access this data structure.)

Within the implementation of higher level-operations like `remap`, *where* constructs appear frequently. They are implemented in the more efficient style illustrated here.

9.2 Non-Cartesian arrays

The original specification of Adlib called for support of *non-Cartesian arrays*. These are non-rectangular arrays—essentially arrays of arrays. The intention was that these would be understood by communication primitives like `remap`, which could thus be used to move about array intermediates generated in translation of non-rectangular `FORALL` constructs. Non-Cartesian arrays are not yet implemented.

10 Fortran interface

The current Fortran interface to Adlib is quite restrictive (and inefficient). Essentially it only provides access to the communication functions; Fortran arrays representing local segments of distributed arrays are passed in with dope vectors describing the data mapping. The interface code constructs Adlib class objects from this parametrisation, and dispatches the relevant C++ communication procedures.

The Fortran interface provides functions which directly implement all the Fortran 90 intrinsic functions on arrays.

11 Implementation status

Adlib is part of the distribution of the Southamption High Performance Fortran, version 0.1, which is currently in Beta testing. It is written in C++, layered on top of MPI.

The released version of Adlib is a prototype, offering full functionality, but poor performance. Major optimisations are in progress (for example, a new version of `remap` is up to an order of magnitude faster on some platforms).