Programming in ad++

D.B. Carpenter

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

DRAFT

Contents

1 Overview of ad++

Adlib is a C++ class library for data parallel programming. It was originally designed as part of the run-time support system for a public domain *High Performance Fortran* (HPF) compilation system. In this article we will emphasize the a user-level C++ interface to Adlib called *ad++*, and illustrate how it can be used directly for parallel programming in C++.

The ad++ library provides a set of abstract data types representing parallel arrays. As in HPF, an array can be distributed over a logical *process array*, which the library maps transparently to the available pool of physical processors.

The library also provides operations to implement a set of *distributed control* constructs. These constructs, called *on*, *at* and *overall* are data-parallel analogues of familiar sequential control constructs like *if* and *for*. They are used for traversing parallel arrays.

Finally it provides a set of high-level communication operations. These are collective operations on parallel arrays. They include

- simple but general *remap* operations, which copy the whole contents of one array to another of the same shape (but potentially different mapping to the process array),
- generalised *gather-scatter* operations which allow any pattern of array subscripting, and
- arithmetic reduction operations, which combine all elements of a parallel array by some operation, such as addition or logical conjunction.

The ad++ interface embodies a high-level model of programming with Adlib. It comes with a caveat. Although the array descriptor and regular communication schedules of the underlying library are supposed to be "production-quality", efficient implementations, the distributed control macros of the ad++ interface, in particular the distributed loop, are too inefficient to be used in real applications. The kernel library definition describes transformations that can be applied to make the *overall* construct acceptably efficient.

2 This article

Adlib has a rich functionality. Its distributed data model is a superset of the data model of HPF version 1.0. Its control constructs and communication library have been designed to fully support *nested parallelism*, a powerful but often unimplemented feature of HPF, which allows one to call parallel intrinisic functions or user-defined *pure procedures* inside other parallel constructs.

Rather than starting with complete, formal definitions of the ad++ classes and concepts, this article takes the reader as quickly possible into ad++ programming through simple examples. The underlying concepts are then visited in more depth, in no particular logical order. By the end of the article the reader should be familiar with all the main ideas in the library.

The article presumes some familiarity with data-parallel and SPMD programming. Familiarity with HPF would also be helpful. Some knowledge of C++ is required.

3 Defining a process array

ad++ provides a series of classes `Procs1`, `Procs2`, ...derived from a common base class `Procs`. These classes represent multi-dimensional *process arrays*. A process array represents a set of logical processes over which data and work can be divided.

If there are fewer logical processes than processors available, each logical process is mapped to a different processor, and some processors hold none of the logical processes in the array. In the present implementation of Adlib it is not allowed to declare a process array with more logical processes than the number of available processors.

A *rank*-2 (two-dimensional) process array `p` is declared by

```
Procs2 p(2, 2) ;
```

`p` represents 4 processes arranged in a two by two array. The *shape* of a process array should be chosen by the programmer to suit the the intended distribution format (blocking) of data arrays. There is no implied pattern of connectivity or nearness between individual processes.

The declaration of `p` is completely analogous to the HPF declaration

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

A process array is *activated* by using the distributed control macro `ON`, and *deactivated* again by using the macro `NO`. We will always use these functions in a context like this:

```
ON(p) {
  // ... some code
} NO(p) ;
```

This idiom is called an *on construct*. The code inside the construct is only executed by processors which hold part of `p`. Inside the construct, `p` is designated the *active process group*. It loses this status after on exit from the construct.

4 Distributed data

A conventional array has a set of index ranges. The ordinary C++ array

```
float a [100] [10] ;
```

has index ranges `0..99` and `0..9`. The index ranges of an ad++ parallel array are more complex, because the elements of the array are distributed over a process dimension. A base class `Range` is introduced to represent these ranges. Two instances, `x` and `y` can be declared as follows

```
BlockRange x(100, p.dim(0)), y(10, p.dim(1)) ;
```

`BlockRange` is a subclass of `Range` which has simple blockwise distribution format. The first argument of the constructor for `x` specifies that the *extent* of `x` is 100—its global indices run from `0..99`. The second argument specifies that `x` is distributed over the first dimension of `p`. The member function `dim` has the signature

```
Dimension Procs :: dim(const int d) ;
```

Its argument is in the range $0, \ldots, R-1$, where R is the rank of the process array. It returns a new class of object representing an individual dimension of a process array. In this example the extent of the process dimension is two, and array elements with *global index* `0..49` are stored on the first row of processes and elements with indices `50..99` on the second.

The range `y` is similar, but it has extent 10 and is distributed over the second dimension of `p`.

The `x`, `y` declarations are comparable with the HPF declarations

```
!HPF$ TEMPLATE T (100, 10)
!HPF$ DISTRIBUTE T (BLOCK, BLOCK) ONTO P
```

The ranges `x` and `y` are analogous to the individual dimensions of the template `T`.

Returning to ad++, if `p` is the currently active process group, a distributed array of floating point variables can be declared by

```
Array2<float> a(x, y) ;
```

This declaration is comparable to the HPF declarations

```
REAL A (100, 10)
!HPF$ ALIGN A WITH T
```

The `Array1`, `Array2`,... template classes are derived from a base class `DAD` which provides various inquiry functions, including

```
Range DAD :: rng(const int d) ;
```

The member `rng` is analogous to the `dim` member of `Procs`, but returns a range rather than a dimension. Its argument is in the range $0, \ldots, R-1$, where R is the rank of the array.

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

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

5 Accessing array elements

Only a processor which holds a copy of an array element is allowed to access that element directly. Elements of an array can either be accessed concurrently through a *distributed loop*, or they can be accessed sequentially. Concurrent access to distributed data is particularly simple in ad++. The range objects introduced in the previous section can be used to construct *iterators* for distributed loops. The iterators are activated by the macro `OVERALL`. Suppose `p` is the active process group, the code

```
Index i(x), j(y) ;
OVERALL(i) {
  OVERALL(j) {
    a(i, j) = i + j ;
  } ALLOVER(j) ;
} ALLOVER(i) ;
```

initialises every element of of `a` to the sum of its two *global* index values. A member of the `Index` class represents the iterator of a distributed loop. In general if `i` is an index the idiom

```
OVERALL(i) {
  // ... some code
} ALLOVER(i) ;
```

is called an *overall construct*.

To explain better how the code fragment above works we need to introduce a new class which represents *local subscripts*. A local subscript defines a particular element of a distributed range. The range object behaves as a map from global index values to local subscripts through the overloaded operator

```
Location& Range :: operator()(const int) ;
```

So the declarations

```
Location s(x(75)), t(y(1)) ;
```

create `s` as a local subscript representing element 75 of the range `x`, while `t` represents element 1 of `y`. A `Location` object defines a particular coordinate within a process array, and a particular offset within the index block (or array block) held on a process at that coordinate. Local subscripts can be used directly to subscript arrays through the overloaded operator

```
T& Array2<T> :: operator()(Location&, Location&) ;
```

But use of this operator is only legal on a processor which holds the selected element (a restriction which will be stated more formally in section 12). One way to test whether an element is held locally is to use the third and final control construct of ad++, the *at construct*. The *at* construct looks rather like the *on* construct, but uses the `AT` and `TA` macros, parametrized by members of `Location` class, as follows:

```
AT(s) {
  AT(t) {
    a(s, t) = 76 ;
  } TA(t) ;
} TA(s) ;
```

The body of the construct executes only if the processor executing the code holds a logical process with the coordinate defined by the subscript. In general if `s` is a local subscript the idiom for the *at* construct is

```
AT(s) {
  // ... some code
} TA(s) ;
```

Now we return to the first example in this section. The index object maintains some internal state defining the global index value associated with the current iteration. The `OVERALL` macro is defined so that the loop body executes just once for each value of the global index held by the processor executing the code. The body of the nested *overall* constructs:

```
a(i, j) = i + j ;
```

relies on a couple of clever tricks. The use of `i` and `j` on the left hand exploits the fact that the `Index` class is derived from the `Location` class. While the index is activated, its `Location` component represents the local subscript associated with the current iteration. The use of `i` and `j` on the right hand side of the assignment uses a defined conversion from `Index` to integer which returns the global index associated with the current iteration. The equivalent code with explicit casts is

```
a(i, j) = (int) i + (int) j ;
```

This is a good point at which to collect together some of the fragments introduced so far, to make our first complete, useless, ad++ program

```
Procs2 p(2, 2) ;

ON(p) {
  BlockRange x(100, p.dim(0)), y(10, p.dim(1)) ;

  Array2<float> a(x, y) ;

  Index i(x), j(y) ;
```

```
    OVERALL(i) {
      OVERALL(j) {
        a(i, j) = i + j ;
      } ALLOVER(j) ;
    } ALLOVER(i) ;

  } NO(p) ;
```

The program defines a process array, declares an array distributed over the process array, and initialises its elements. It is comparable to the HPF program

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

  !HPF$ TEMPLATE T (100, 10)
  !HPF$ DISTRIBUTE T (BLOCK, BLOCK) ONTO P

  REAL A (100, 10)
  !HPF$ ALIGN A WITH T

  FORALL (I = 1 : 100, J = 1 : 10)
    A (I, J) = I + J
```

There is one minor difference between the two programs and one major difference. The minor difference is that Fortran array indices start from 1 by default, whereas Adlib global indices always start from 0. The major difference is that the ad++ program specifies explicitly that the computation `i + j` occurs on the process holding element `a(i, j)`. In the HPF program the *owner computes* heuristic adopted by most compilers would probably lead to `I + J` being computed on the home process of `A(I, J)`, but this is not required by the language definition. HPF provides no means for the programmer to state explicitly where a computation is performed—it is at the compiler's discretion. ad++, on the other hand, *forces* the to programmer specify, through the distributed control constructs, which process is responsible for every computation.

Although there are restrictions on the patterns of data access in the body of the *overall* construct, arbitrary sequential control constructs are allowed there. For example, the kernel of a Mandelbrot set computation is illustrated in figure 1.

6 Communication

In Adlib, communication is normally achieved through collective operations on distributed arrays. One of the simplest is the `shift` operation, whose interface is

```
  template<class T>
```

```
Procs2 p(2, 2) ;

ON(p) {
  BlockRange x(N, p.dim(0)), y(N, p.dim(1)) ;

  Array2<float> colour(x, y) ;

  Index i(x), j(y) ;
  OVERALL(i) {
    OVERALL(j) {
      float cr = (4.0 * i - 2 * N) / (N - 1) ;
      float ci = (4.0 * j - 2 * N) / (N - 1) ;
      float zr = cr, zi = ci ;

      for (int i = 0 ; i < RESOLUTION &&
                       (zr * zr + zi * zi) < 4.0 ; i++) {
        zr = cr + zr * zr - zi * zi + cr ;
        zi = ci + 2 * zr * zi ;
      }

      colour (i, j) = i ;
    } ALLOVER(j) ;
  } ALLOVER(j) ;

} NO(p) ;
```

Figure 1: Mandelbrot set kernel.

```
  void shift(const Section1<T>& dst, const Section1<T>& src,
             const int shift, const int dim, const Mode mode) ;

  template<class T>
  void shift(const Section2<T>& dst, const Section2<T>& src,
             const int shift, const int dim, const Mode mode) ;

  ...
```

The parameters `dst` and `src` are arrays, which must have identical type, shape and mapping (for now, we take this to mean that they share the same range structures). `Section1<T>`, `Section2<T>`, ... are base classes of `Array1<T>`, `Array2<T>`, ... The operation shifts the values in `src` by `shift` places in the `dim` dimension, and puts the result in `dst`. `shift` is a signed integer. `dim` is in the range $0, 1, \ldots, R-1$, where R is the rank of the arrays. `mode` is one of `CYCL` or `EDGE`, selecting either cyclic or "edge-off" shift.

With `shift` we can write some more meaningful programs. In figure 2 the *overall* construct replaces each element of `w` (except those on the edges) with the average of the four neighbouring values. This is the kernel of simplified PDE solver. Figure 3 performs an iteration of Conway's Life.

Adlib provides many communication operations more powerful than `shift`. They will be introduced in later sections.

7 Subranges

The ranges introduce so far were *template ranges*. Adlib also has an idea of a *subrange*. A subrange is a subrange of some other range (and, ultimately, of some template range). It retains an *alignment* to its parent range. The location of a subrange element remains the same as the corresponding element of the parent range.

```
  BlockRange x(N, p.dim(0)) ;
  Range u = x.subrng(N - 2, 1) ;
```

The range `u` is a subrange of `x`. It has extent $N-2$ and offset 1. Element 0 of `u` is aligned with element 1 of `x`, element 1 with element 2, and so on. The last element of `u`, element $N-3$, is aligned with the penultimate element, $N-2$, of `x`.

As an immediate application, the *overall* construct in figure 2 could be replaced by

```
    Index k(x.subrng(N - 2, 1)) ;
    Index l(y.subrng(N - 2, 1)) ;

    OVERALL(k) {
```

```
Procs2 p(2, 2) ;

ON(p) {
  BlockRange x(N, p.dim(0)), y(N, p.dim(1)) ;

  Array2<float> w(x, y) ;

  // ... some code to initialise ‘w’

  Array2<float> wnx(x, y), wpx(x, y), wny(x, y), wpy(x, y) ;

  shift(wnx, w,  1, 0, EDGE) ;
  shift(wpx, w, -1, 0, EDGE) ;
  shift(wny, w,  1, 1, EDGE) ;
  shift(wpy, w, -1, 1, EDGE) ;

  Index i(x), j(y) ;
  OVERALL(i) {
    OVERALL(j) {
      if(i != 0 && i != N - 1 && j != 0 && j != N - 1)
        w(i, j) = 0.25 *
            (wnx(i, j) + wpx(i, j) + wny(i, j) + wpy(i, j)) ;
    } ALLOVER(j) ;
  } ALLOVER(i) ;

} ON(p) ;
```

Figure 2: PDE solver kernel.

```
Procs2 p(2, 2) ;

ON(p) {
  BlockRange x(N, p.dim(0)), y(N, p.dim(1)) ;

  Array2<int> c(x, y) ;

  // ... some code to initialise ‘c’

  Array2<int> cn_(x, y), cp_(x, y), c_n(x, y), c_p(x, y),
              cnn(x, y), cnp(x, y), cpn(x, y), cpp(x, y) ;

  shift(cn_, c,  1, 0, CYCL) ;
  shift(cp_, c, -1, 0, CYCL) ;
  shift(c_n, c,  1, 1, CYCL) ;
  shift(c_p, c, -1, 1, CYCL) ;
  shift(cnn, cn_,  1, 1, CYCL) ;
  shift(cnp, cn_, -1, 1, CYCL) ;
  shift(cpn, cp_,  1, 1, CYCL) ;
  shift(cpp, cp_, -1, 1, CYCL) ;

  Index i(x), j(y) ;
  OVERALL(i) {
    OVERALL(j) {
      switch (cn_(i, j) + cp_(i, j) + c_n(i, j) + c_p(i, j) +
              cnn(i, j) + cnp(i, j) + cpn(i, j) + cpp(i, j)) {
        case 2 :
          break ;
        case 3 :
          c(i, j) = 1 ;
          break ;
        default :
          c(i, j) = 0 ;
          break ;
      }
    } ALLOVER(j) ;
  } ALLOVER(i) ;

} NO(p) ;
```

Figure 3: Conway’s Life iteration.

```
    OVERALL(l) {
      w(k, l) = 0.25 *
          (wnx(k, l) + wpx(k, l) + wny(k, l) + wpy(k, l)) ;
    } ALLOVER(l) ;
  } ALLOVER(k) ;
```

This works because a local subscript constructed from a subrange is exactly equivalent to a subscript constructed from a parent range at an aligned point. The advantage of the new version is that we have eliminated an expensive test from the inner body of the loops.

If the original code had been something like

```
  Index i(x), j(y) ;
  OVERALL(i) {
    OVERALL(j) {
      if(i != 0 && i != N - 1 && j != 0 && j != N - 1)
        w(i, j) = i + j ;
    } ALLOVER(j) ;
  } ALLOVER(i) ;
```

with a dependence on global index value, we could still perform the above transformation by declaring `k`, `l` as above, then

```
  OVERALL(k) {
    OVERALL(l) {
      w(k, l) = x(k) + y(l) ;
    } ALLOVER(l) ;
  } ALLOVER(k) ;
```

This exploits the operator

```
 int Range :: operator()(Location&) ;
```

which effectively allows a range to subscripted as if it were a one-dimensional array holding its global index values.

It is possible to specify a stride in a subrange, as in

```
 Range t = x.subrng(10, 20, 2) ;
```

Element 0 of range `t` is aligned with element 20 of range `x`, element 1 with element 22, etc.

Arrays can be declared with subranges. This mechanism allows affine alignment between arrays, as provided in HPF 1.0.

8 Process groups

A *process group* is a set of logical processes, such as a process array. In general a process group has two roles—a particular action can be performed within a specified process group, or a particular data object can be distributed over a specified process group.

All the examples so far used process groups in a peripheral way. One process array was declared, immediately activated by an *on* construct, and deactivated at the end of the program. This static approach to process configuration is enough to write many useful programs, but Adlib provides more flexibility, if that flexibility is needed.

A program can declare several different process arrays, possibly of different rank. Data arrays declared in the earlier examples were always distributed over the unique *active process group*. In general the array constructor takes a process group as its last argument, specifying that the data is distributed over that group. In this way a program can contain coexisting arrays distributed over different groups. For example

```
Procs2 p(2, 2) ;
Procs1 q(3) ;

BlockRange x(10, p.dim(0)), y(10, p.dim(1)) ;
Array2<float> a(x, y, p) ;

BlockRange z(100, q.dim(0)) ;
Array1<int> b(z, q) ;

ON(p) {
  // ... some code processing ‘a’
} NO(p) ;

ON(q) {
  // ... some code processing ‘b’
} NO(q) ;
```

Note that `p` and `q` can only activated in turn, *not* concurrently. We will see section 11 how data can be transferred between data arrays distributed over different process groups.

Adlib also 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.

A process coordinate is defined implicitly by a local subscript. The operator `/` is overloaded to construct a new process group (class `Group`) from an existing process group and a subscript.

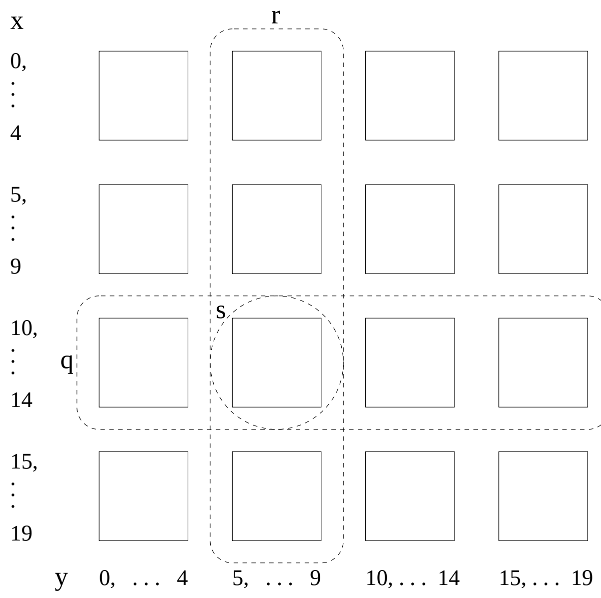

Figure 4: Examples of process groups. The square boxes represent the 16 logical processes in the process array p. The dashed lines embrace groups q, r and s.

```
Procs2 p(4, 4) ;

BlockRange x(20, p.dim(0)), y(20, p.dim(1)) ;

Location i(x(12)), j(y(7)) ;

Group q(p / i) ;
Group r(p / j) ;
Group s(q / j) ;
```

The groups `q`, `r` and `s` are illustrated in figure 4. The *parent process array* of all these groups is `p`. Sometimes we refer to groups like `q`, `r` and `s` as *restricted process groups*. A process group is either a process array or a restricted process group.

Any process group has a set of process dimensions taken from its parent array. The *dimension set* for `p` is `p.dim(0)` and `p.dim(1)`; the set for `q` is just `p.dim(1)`; the set for `s` is empty.

The generalised process group provides more refined way to describe the active process group inside the ad++ control constructs (see section 9), and to describe the process groups over which general *array sections* are distributed (see section 10).

9 The active process group

At any point of execution of a program, one group is singled out as the *active process group* (APG). From the viewpoint of a particular processor, the active process group is the set of processes sharing the current "thread of control".

There is a natural *contains* relationship between certain process groups. Group p contains group q if

1. p was the active process group at the point at which q was declared, or
2. q is a sub-array of p, constructed by the / operation.
3. p contains some process array r which in turn contains q.

For example

```
Procs1 p(6) ;
ON(p) {
  Procs2 r(2, 2) ;

  BlockRange x(100, r.dim(1)) ;
  Group q(r / x(50)) ;

  // ...
} NO(p) ;
```

Here, `p` contains `r` by rule 1 above, `r` contains `q` by rule 2, and `p` contains `q` by rule 3. As a matter of convention we say that a group contains itself.

The three ad++ control constructs affect the active process group in the following ways.

- If the active process group is currently `p`, and `q` is another process group, inside

  ```
  ON(q) {
    // ...
  } NO(p) ;
  ```

  the `ON(q)` operation changes the active process group to `q`. The `NO(p)` operation restores it to `p`.

- If the active process group is currently `p` and `s` is a subscript, then inside

  ```
  AT(s) {
    // ...
  } TA(s) ;
  ```

the `AT(s)` operation changes the active process group to `p / s`. The `TA(s)` operation restores it to `p`.

- If the active process group is `p` and `i` is an index, inside

  ```
  OVERALL(i) {
    // ...
  } ALLOVER(i) ;
  ```

  the `OVERALL(i)` operation changes the active process group to `p / i`, assuming the coercion from index to subscript explained in section 5. The `ALLOVER(i)` operation restores it to `p`.

Each construct has a precondition

- An *on* construct can appear only if its process group is contained in the active process group.
- An *at* construct can appear only if its subscript is derived from a range distributed over a dimension of the active process group.
- A *overall* construct can appear only if the range of its index is distributed over a dimension of the active process group.

One effect of these rules is to outlaw the appearance of nested *at* or *overall* constructs if their subscripts or index ranges are distributed in the same dimensions. The groups activated by *at* and *overall* are strictly smaller than the surrounding APG, reduced in rank by one. (The *on* construct, on the other hand, can leave the APG unchanged, if its argument happens to be identical to the current APG.)

10 Array sections

ad++ supports array sections analogous to Fortran array sections through the member function `sect`.

Suppose we have the ad++ declarations

```
Procs2 p(2, 2) ;

BlockRange x(10, p.dim(0)) ;
BlockRange y(10, p.dim(1)) ;

Array2<int> a(x, y, p) ;
```

which are comparable to the HPF declarations

```
!HPF$ PROCESSORS P (2, 2)
!HPF$ TEMPLATE T (10, 10)
!HPF$ DISTRIBUTE T(BLOCK, BLOCK) ONTO P

INTEGER A(10, 10)
!HPF$ ALIGN A (:, :) WITH T
```

In ad++ a section of `a` can be introduced as follows

```
Range u = x.subrng(8, 1) ;
Range v = y.subrng(8, 1) ;

Section2<int> b = a.sect(u, v) ;
```

`u` and `v` are subranges similar to the ones introduced in section 7. The array `b` represents a section of `a` containing only the non-edge points. It provides an alias for this section of `a`'s data. This alias can be passed to ad++-defined procedures (including other section constructors) and user-defined procedures in just the same way as a full data array.

The section `b` is comparable with the Fortran section

```
A (2 : 9, 2 : 9)
```

Fortran, unlike ad++, does not allow this section to be named, *except* by passing it to a procedure and referencing it as a dummy argument.

A local subscript can be passed to a section-constructor, in place of a range. With the earlier declarations we could also define

```
Location i(y(5)) ;
Section1<int> c = a.sect(x, i) ;
```

which makes `c` analogous to the Fortran section

```
A (:, 6)
```

Note:

- Any range or subscript passed to an array constructor must be derived from the range used to construct the parent array argument.
- Like any other data array in ad++, a section is distributed over a particular process group. The section `b` is distributed over `p`. The section `c` is distributed over `p / y(5)`. In general the section is distributed over the same group as the parent array, restricted by any local subscript arguments.

Incidentally, it is not only array sections that can be distributed over restricted process groups. With the above declarations, the ad++ code

```
Array1<int> d(x, p / i) ;
```

is comparable to the HPF 1.0 code

```
INTEGER D (10)
!HPF$ ALIGN D (:) WITH T (:, 6)
```

The `DAD` base class provides the inquiry function

```
Group DAD :: grp() ;
```

which returns the process group over which an array is distributed.

11 *remap*

In section 6 we introduced one of the simplest communication operations in Adlib—the shift. A much more powerful cousin of `shift` is `remap`.

The deceptively simple interface of `remap` is

```
template<class T>
void remap(const Section1<T>& dst, const Section1<T>& src) ;

template<class T>
void remap(const Section2<T>& dst, const Section2<T>& src) ;

...
```

The two arguments are arrays of the same type and shape. The operation just copies `src` to `dst`. The mapping of the two arrays is unrestricted. They can be distributed differently over the same process group, or they can be distributed over different process groups. The only constraint on the arguments is that they are both *accessible*. An array is accessible if it is distributed over a process group contained in the active process group. Nearly all collective operations in Adlib require their array arguments to be accessible[1].

Used in conjunction with array sections, `remap` can implement many patterns of communication. For example the shift in

```
BlockRange x(N, p.dim(0)) ;
Array1<float> a(x), b(x) ;

shift(a, b, 1, 0, EDGE) ;
```

could be coded (more clumsily) as

[1]A significant exception is the section constructor introduced in the last section, which only requires that the *constructed* array is accessible. The whole of the parent array need not by accessible.

```
Range u = x.subrng(N - 1, 1), v = x.subrng(N - 1, 0, x) ;
remap(a.sect(u), b.sect(v)) ;
```

Here we used the section constructor inline to create anonymous sections. The code is comparable to the Fortran

```
REAL A (N), B (N)

A (2 : N) = B (1 : N - 1)
```

12 Replicated data

Consider the array declarations in

```
Procs2 p(2, 2) ;
ON(p) {
  BlockRange x(10, p.dim(0)) ;
  BlockRange y(10, p.dim(1)) ;

  Array2<int> a(x, y) ;
  Array1<int> b(x) ;
  Array0<int> c ;

  // ...
} NO(p) ;
```

Arrays like `a`, with ranges distributed over all dimensions of their process group, have been illustrated many times in earlier examples. Array `b`, on the other hand, is distributed over the active process group `p`, but has no range distributed over `p.dim(1)`. Array `c` has no range distributed over *any* dimension of `p`.

In general an array need not have ranges distributed over *all* dimensions of its process group (but each distributed range of an array must be distributed over a distinct dimension of that group).

Array `b` is *replicated* over dimension `p.dim(1)`. Similarly `c` is replicated over both dimensions of the process array.

Any simple C++ variable is replicated over the process group active at the time at its declaration. The array `c` only differs from a such a variable by being wrapped up as an array. This mean that it can be passed to operations such as `remap`.

The fragment

```
remap(c, a.sect(x(5), y(5))) ;
```

is a broadcast, copying the `(5, 5)` element of `a` into the single replicated element of `c`. This value can then be referenced simply as `c()`. Notice that both arguments of `remap` must be rank-0 arrays here, so the second argument had to be a section. We could not simply write `a(x(5), y(5))`, because that expression evaluates to a simple `int`[2], whereas `remap` requires an array argument.

We have to follow some rule in updating replicated data, to ensure that copies remain consistent. An operation like

```
Index i(x), j(y) ;
OVERALL(i) {
  OVERALL(j) {
    a(i, j) = b(i) ;
  } ALLOVER(j) ;
} ALLOVER(i) ;
```

ought to be allowed, whereas

```
OVERALL(i) {
  OVERALL(j) {
    b(i) = a(i, j) ;
  } ALLOVER(j) ;
} ALLOVER(i) ;
```

should be forbidden, because it would (presumably) create inconsistent values for the elements of `b`, which are supposed to be replicated across different processors.

The rule is simple enough. Suppose a rank-R array a is distributed over process group p. If $s_1, \ldots, s_R$ are suitable local subscripts, then, as a matter of definition, the array element $a(s_1, \ldots, s_R)$, is replicated over $p/s_1/\ldots/s_R$. Now the required rule is

> An array element can only be updated when the active process group is identical to the group over which the element is replicated.

We can also generalize and make more precise a rule about general access to array elements which was stated informally in section 5. The new formulation is:

> The value of an array element can only be accessed if it is replicated over a group containing the active process group.

With the understanding that simple C++ variables are replicated over the group active at their point of declaration, we propose that they satisfy the same access rules.

13 Other distribution formats

So far we have only seen one distribution format for template ranges—simple block distribution. Adlib currently supports several other formats

[2]And this expression is undefined except on the processor which holds the element.

- collapsed
- simple cyclic
- block cyclic
- step

A collapsed range is declared with the syntax

```
Range x(100) ;
```

The range `x` is not distributed across any process array. It is mapped into the memory of a single logical process. For example

```
Range x(100) ;
Array1<float> a(x) ;

Procs1 p(4) ;
BlockRange y(100, p.dim(0)) ;
Array1<float> b(y, p) ;

// ... initialize 'b'

remap(a, b) ;
```

The array `a` retains the slightly awkward syntax `a(x(n))` for accessing element `n`, but is otherwise essentially a sequential array: a copy of all elements are held on all processes of the active process group[3]. The `remap` operation implements a "concatenate" operation which copies the distributed array `b` to the sequential array `a`. In general an array can have a free mixture of distributed ranges and collapsed ranges.

A range is declared with simple cyclic distribution by

```
CyclicRange y(100, p.dim(d)) ;
```

Element i of the range is mapped to a process with coordinate $i \bmod P_d$, where P_d is the extent of `p.dim(d)`.

[HPF-style block-cyclic distribution is also supported, but the ad++ syntax for the range constructor is under revision.]

Step distribution is specified by

```
StepRange w(100, p.dim(d)) ;
```

This format is similar to simple block distribution except that the first $100 \bmod P_d$ processes are assigned $b + 1$ range elements and the remaining processes are assigned b elements, for suitably chosen block size b.

All of the template ranges support subranges as described in section 7.

[3]A *where* construct parametrised by `x` would be perfectly legal, but an ordinary *for* loop is probably more conveninent for processing `a`. An *at* construct parametrised by a subscript in `x` is legal, but fairly redundant, as it has no effect on the active process group.

14 Other communication operations

There are three main families of communication operations in Adlib

- the *remap* family.
- the *gather-scatter* family, and
- the *reduction* family.

All the communication operations described here are *collective* operations. They must be invoked consistently by all members of the active process group. A powerful feature of Adlib, designed to support "nested parallelism", is that collective operations such as communication operations and array declarations (and even process array declarations), can appear at any point of the program. It is *not* required that all physical processors engage in the operations[4]. Any synchronisation implied by the collective operation affects just the processors cooperating in a particular active process group, leaving other processors undisturbed. So it is legal to call a *shift* operation, say, inside an *overall* construct, as in

```
Array2<int> a(x, y), b(x, y) ;

Index j(y) ;
OVERALL(j) {
  shift(b.sect(x, j), a.sect(x, j), j, 0, CYCL) ;
} ALLOVER(j) ;
```

which implements a skewed shift of `a`. The `sect` functions create rank-1 arrays representing individual columns of `a` and `b`[5].

The *remap* family includes `remap` itself, `shift` and a few similar operations.

The *gather-scatter* family includes various operations which allow more complex non-linear subscripting patterns in access to distribute arrays.

The general `gather` operations have interfaces

```
template<class T>
void gather(const Section1<T>& b, const Section1<T>& a,
            const Section1<int>& s_1) ;

template<class T>
void gather(const Section2<T>& b, const Section1<T>& a,
            const Section2<int>& s_1) ;
...
```

[4]It is generally required that all array arguments of the operation are accessible, as explained in section 11.

[5]the final use of *j* in the *shift* operation is as an integer global index.

```
template<class T>
void gather(const Section1<T>& b, const Section2<T>& a,
            const Section1<int>& s_1,
            const Section1<int>& s_2) ;

template<class T>
void gather(const Section2<T>& b, const Section2<T>& a,
            const Section2<int>& s_1,
            const Section2<int>& s_2) ;
...
```

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`. Adlib also provides a *combining scatter* which is parametrised by some combining function.

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

```
template<class T>
void vecGather(const Section1<T>& b, const Section1<T>& a,
               const Section1<int>* s_1) ;

template<class T>
void vecGather(const Section2<T>& b, const Section2<T>& a,
               const Section1<int>* s_1,
               const Section1<int>* s_2) ;

...
```

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 simplest reduction primitive has the inteface

```
template<class T>
T sum(const Section1<T>& src) ;

template<class T>
```

```
  T sum(const Section2<T>& src) ;

  ...
```

This adds all elements of `src`, broadcasting the result over the active process group. Variants include all reductions (including searches and scalar products) from the Fortran 90 transformational intrinsics.