Adlib

D.B. Carpenter

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

DRAFT

Contents

1 Introduction

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. It has subsequently been used as comunications platform for a port of a public domain *C** compiler. In this article we will emphasize the native C++ interface to Adlib, and illustrate how it can be used directly for parallel programming in C++.

1.1 Overview of Adlib

Adlib 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 *where* 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.

1.2 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 Adlib classes and concepts, section 2 takes the reader as quickly possible into Adlib programming through simple examples. The underlying concepts are then visited in more depth, in no particular logical order. By the end of the section the reader should be familiar with all the main ideas in Adlib.

Full logical development of the Adlib programming model, the detailed interface of the communication suite and major classes, and various data representation details, are deferred to the reference sections *[in preparation]*.

Section 3 discusses various implementation issues which have a direct impact on efficient programming, and interfacing of Adlib to external systems.

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

2 Introduction to Adlib Programming

2.1 Defining a process array

Adlib 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[1].

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 member function `on`, and *deactivated* again by using the member `no`. We will always use these functions in the context of an *if* construct, like this:

```
if(p.on()) {
  // ... some code
} p.no() ;
```

This idiom is called an *on construct*. The `on` member returns a truth value which is non-zero if the processor executing the code holds one or more processes from `p`. So the code inside the construct is only executed by processors which hold part of `p`. After the `p.on()` call, `p` is designated the *active process group*. It loses this status after the `p.no()` call.

[1]This was allowed in some earlier implementations but, besides complicating the implementation, it led to a more convoluted specification. In the name of simplicity, this facility has now been dropped.

2.2 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 Adlib parallel array are more complex, because the elements of the array are distributed over a process dimension. A new class `Range` is introduced to represent these ranges. Two instances, `x` and `y` can be declared as follows

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

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 pointer to a new class of object representing an individual dimension of a process array. The third argument of the constructor specifies the *distribution format* of the index—in this case simple blockwise decomposition. 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 analoguous to the individual dimensions of the template `T`.

Returning to Adlib, 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 clase `Array` which provides various inquiry functions, including

```
Range Array :: 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() ;
```

2.3 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 Adlib. The range objects introduced in the previous section can be used to construct *iterators* for distributed loops. The iterators are activated by the member function `where`. Suppose `p` is the active process group, the code

```
Index i(x), j(y) ;
for(i.where() ; i.test() ; i.next()) {
  for(j.where() ; j.test() ; j.next()) {
    a(i, j) = i + j ;
  } j.erewh() ;
} i.erewh() ;
```

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

```
for(i.where() ; i.test() ; i.next()) {
  // ... some code
} i.erewh() ;
```

is called a *where construct*. (This name is historical. Unfortunately it clashes with the name of a construct in Fortran 90. The Adlib construct does not involve masking.)

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

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

So the declarations

```
Subscript 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 `Subscript` 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()(Subscript&, Subscript&) ;
```

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 2.10). One way to test whether an element is held locally is to use the third and final control construct of Adlib, the *at construct*. The *at* construct looks rather like the *on* construct, but uses the `at` and `ta` members of the `Subscript` class as follows:

```
if(s.at()) {
  if(t.at()) {
    a(s, t) = 76 ;
  } t.ta() ;
} s.ta() ;
```

The `at` member returns a truth value which is non-zero if the processor executing the code holds a logical process having the coordinate defined by the subscript. In general if `s` is a local subscript the idiom for the *at* construct is

```
if(s.at()) {
  // ... some code
} s.ta() ;
```

We can now 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 `where`, `test` and `next` members are constructed 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 *where* 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 `Subscript` class. While the index is activated, its `Subscript` 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, Adlib program

```
Procs2 p(2, 2) ;

if(p.on()) {
  Range x(100, p.dim(0), BLK), y(10, p.dim(1), BLK) ;

  Array2<float> a(x, y) ;

  Index i(x), j(y) ;
  for(i.where() ; i.test() ; i.next()) {
    for(j.where() ; j.test() ; j.next()) {
      a(i, j) = i + j ;
    } j.erewh() ;
  } i.erewh() ;

} p.no() ;
```

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 Adlib 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. Adlib, on the contrary, *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 *where* construct, arbitrary sequential control constructs are allowed there. For example, the kernel of a Mandelbrot set computation is illustrated in figure 1.

```
Procs2 p(2, 2) ;

if(p.on()) {
  Range x(N, p.dim(0), BLK), y(N, p.dim(1), BLK) ;

  Array2<float> colour(x, y) ;

  Index i(x), j(y) ;
  for(i.where() ; i.test() ; i.next()) {
    for(j.where() ; j.test() ; j.next()) {
      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 ;
    } j.erewh() ;
  } i.erewh() ;

} p.no() ;
```

Figure 1: Mandelbrot set kernel.

2.4 Communication

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

```
void shift(Array dst, Array src,
           const int shift, const int dim, const int 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). 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 one or two more worthwhile programs. In figure 2 the *where* 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.

2.5 Derived ranges

The ranges introduce so far were *primitive ranges*. Adlib also has an idea of a *derived range*. A derived range is a sub-range of some other range (and, ultimately, of some primitive 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.

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

The range `u` is derived from `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 *where* construct in figure 2 could be replaced by

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

for(k.where() ; k.test() ; k.next()) {
  for(l.where() ; l.test() ; l.next()) {
    w(k, l) = 0.25 *
```

```
Procs2 p(2, 2) ;

if(p.on()) {
  Range x(N, p.dim(0), BLK), y(N, p.dim(1), BLK) ;

  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) ;
  for(i.where() ; i.test() ; i.next()) {
    for(j.where() ; j.test() ; j.next()) {
      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)) ;
    } j.erewh() ;
  } i.erewh() ;

} p.no() ;
```

Figure 2: PDE solver kernel.

```
Procs2 p(2, 2) ;

if(p.on()) {
  Range x(N, p.dim(0), BLK), y(N, p.dim(1), BLK) ;

  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) ;
  for(i.where() ; i.test() ; i.next()) {
    for(j.where() ; j.test() ; j.next()) {
      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 ;
      }
    } j.erewh() ;
  } i.erewh() ;

} p.no() ;
```

Figure 3: Conway's Life iteration.

```
        (wnx(k, l) + wpx(k, l) + wny(k, l) + wpy(k, l)) ;
  } l.erewh() ;
} k.erewh() ;
```

This works because a local subscript constructed from a derived range 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) ;
for(i.where() ; i.test() ; i.next()) {
  for(j.where() ; j.test() ; j.next()) {
    if(i != 0 && i != N - 1 && j != 0 && j != N - 1)
      w(i, j) = i + j ;
  } j.erewh() ;
} i.erewh() ;
```

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

```
for(k.where() ; k.test() ; k.next()) {
  for(l.where() ; l.test() ; l.next()) {
    w(k, l) = x(k) + y(l) ;
  } l.erewh() ;
} k.erewh() ;
```

This exploits the operator

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

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 derived range, 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 derived ranges. This mechanism allows affine alignment between arrays, as provided in HPF 1.0.

2.6 Process groups

A *process group* is a set of logical processes, such as a process array. In general a 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.

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 first 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) ;

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

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

if(p.on()) {
  // ... some code processing 'a'
} p.no() ;

if(q.on()) {
  // ... some code processing 'b'
} q.no() ;
```

Note that `p` and `q` can only activated in turn, *not* concurrently. We will see section 2.9 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.

```
Procs2 p(4, 4) ;

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

Subscript i(x(12)), j(y(7)) ;
```

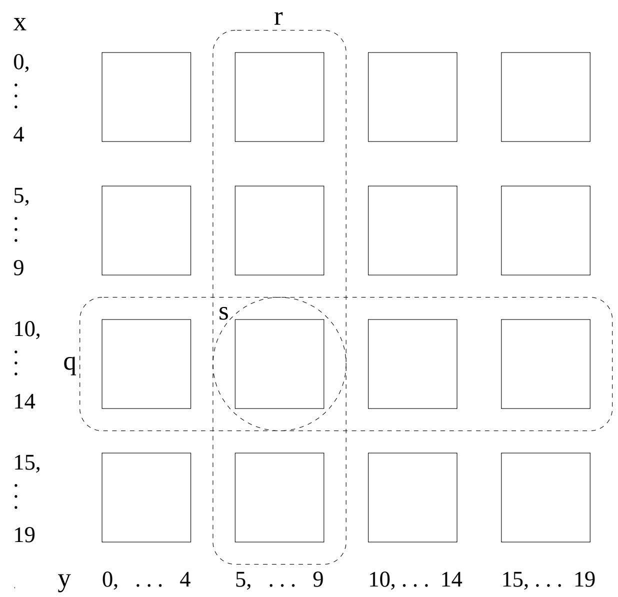

Figure 4: Examples of process groups. The square boxes represent the 16 logical processes in the process array p. The dashed lines embrace groups q, r and s.

```
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 Adlib control constructs (see section 2.7), and to describe the process groups over which general *array sections* are distributed (see section 2.8).

2.7 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) ;
if(p.on()) {
  Procs2 r(2, 2) ;

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

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

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 will also say that a group contains itself.

The three Adlib 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

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

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

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

  ```
  if(s.at()) {
    // ...
  } s.ta() ;
  ```

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

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

```
for(i.where() ; i.test() ; i.next()) {
  // ...
} i.erewh() ;
```

the `i.where()` operation changes the active process group to `p / i`, assuming the cooercion from index to subscript explained in section 2.3. The `i.erewh()` 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 *where* 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 *where* constructs if their subscripts or index ranges are distributed in the same dimensions. The groups activated by *at* and *where* 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.)

2.8 Array sections

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

Suppose we have the Adlib declarations

```
Procs2 p(2, 2) ;

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

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

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 Adlib a section of `a` can be introduced as follows

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

Array2<int> b = a.sect(u, v) ;
```

`u` and `v` are derived ranges similar to the ones introduced in section 2.5. 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 Adlib-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 Adlib, 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

```
Subscript i(y(5)) ;
Array1<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.
- If a *range* currently activated as a *where* iterator is to be used as a section subscript argument, it must first be cooerced to a local subscript.
- Like any other data array in Adlib, 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 Adlib code

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

is comparable to the HPF 1.0 code

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

The `Array` base class provides the inquiry function

```
Group Array :: grp() ;
```

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

2.9 *remap*

In section 2.4 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

```
remap(Array dst, Array 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[2].

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

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

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

could be coded (more clumsily) as

```
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)
```

[2]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.

2.10 Replicated data

Consider the array declarations in

```
Procs2 p(2, 2) ;
if(p.on()) {
  Range x(10, p.dim(0), BLK) ;
  Range y(10, p.dim(1), BLK) ;

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

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

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, then, 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 in 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, however that the `src` argument of `remap` had to be a rank-0 array section. We could not simply write `a(x(5), y(5))` here, because that expression evaluates to a simple `int`[3], 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) ;
for(i.where() ; i.test() ; i.next()) {
  for(j.where() ; j.test() ; j.next()) {
    a(i, j) = b(i) ;
```

[3]Moreover this expression is undefined except on the processor which holds the element.

```
    } j.erewh() ;
  } i.erewh() ;
```

ought to be allowed, whereas

```
  for(i.where() ; i.test() ; i.next()) {
    for(j.where() ; j.test() ; j.next()) {
      b(i) = a(i, j) ;
    } j.erewh() ;
  } i.erewh() ;
```

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 2.3. The new formulation is:

> 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 will insist that they satisfy the same access rules.

2.11 Other distribution formats

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

- 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) ;
Range y(100, p.dim(0), BLK) ;
Array1<float> b(p, y) ;

// ... 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[4]. 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

```
Range y(100, p.dim(d), CYC) ;
```

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 specified by, for example

```
Range z(100, p.dim(d), CYC, 10) ;
```

mapping element i to $[i/10] \bmod P_d$.

Step distribution is specified by

```
Range w(100, p.dim(d), STP) ;
```

This format is similar to simple block distribution except that the first 100 mod 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 primitive ranges support derived ranges as described in section 2.5.

[4] 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.

2.12 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, 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[5]. 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 perfectly legal to call a *shift* operation, say, inside a *where* construct, as in

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

Index j(y) ;
for(j.where() ; j.test() ; j.next()) {
  shift(b.sect(x, j), a.sect(x, j), j, 0, CYCL) ;
} j.erewh() ;
```

which implements a skewed shift of `a`. The `sect` functions create rank-1 arrays representing individual columns of `a` and `b`[6].

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

```
void gather(Array b, Array a, Array i_1) ;

void gather(Array b, Array a, Array i_1, Array i_2) ;

void gather(Array b, Array a, Array i_1, Array i_2,
                              Array 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

[5]It is generally required that all array arguments of the operation are accessible, as explained in section 2.9.

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

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

```
void vecGather(Array b, Array a, Array i_1) ;

void vecGather(Array b, Array a, Array i_1, Array i_2) ;

void vecGather(Array b, Array a, Array i_1, Array i_2,
                                 Array 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 array (which can constructed with the default constructor `Array :: Array()`) 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

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

The result, 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.

2.13 Miscellany

2.13.1 Sequential arrays

We saw in section 2.11 how a sequential array can be implemented in terms of a general Adlib array, by using collapsed ranges. It is convenient, however, to provide a separate family of lightweight, sequential array classes. They are parametrised simply by integer extents, and elements are accessed through integer subscripts

```
SArray2<float> a(10, 10) ;
SArray1<float> b(20) ;

a(5, 5) = b(10) ;
```

The only advantage over ordinary C++ arrays (which, of course, remain available to the Adlib programmer) is that Adlib sequential arrays are bundled with a hidden shape parameters, accessible through inquiry functions. These allow construction of shape-generic library functions.

2.13.2 Array Promotions

An array promotion is something like an array section in reverse. A promotion typically creates a wrapper which allows a given scalar or array value to be viewed as an array of higher rank, with values replicated in the extra dimensions.

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

Here `b` is an array, but every element of `b` has the value of `a`. This effect is achieved without any copying; every expression of the form `b(i, j)` just returns a reference to `a` (so updating `a` would apparently update every element of `b`).

```
Array2<int> a(x, y) ;
Array3<int> b(x, z, y, a, seq(0, 2)) ;
```

Again `b` has the same data as `a`, but the data is replicated in the extra range `z`. The final parameter of the `b` constructor defines the mapping from dimensions of `a` to `b`[7], so this constructor can also be used to permute the dimensions of an array.

Another conceptually similar operation goes under the same umbrella:

```
SArray2<int> a(10, 10) ;
Array2<int> b(x, y, a) ;
```

Here `a` is a sequential array. `b` provides a distributed view of this data. The ranges `x` and `y` should both have extent 10; they can have any distribution format.

These operations are provided so that data in "smaller" structures can be passed to functions expecting Adlib array parameters, without having to copy the data first. The elements of a promotion should only be accessed by value; the effect of updating an element is undefined

[7] *seq*(0, 2) is shortand for a two-element vector containing 0 and 2.

3 Some Implementation Issues

In the previous sections we have described a very high-level interface to Adlib. That interface essentially defines a data-parallel language extending C++. This "language" was implemented as a class library.

The class library approach has various advantages.

- It is much less work to implement a class library than to implement a whole compiler or a good preprocessor.
- If we decide that a particular semantic idea isn't quite right, it is much easier to change a library implementation.
- We avoid the maintenance problem associated with tracking developments in compilers for the base language. As the C++ language definition evolves, or C++ compilers improve, we simply use the new compilers that other people implement.

A compiler approach would have its own advantages

- A handful of language extensions would improve the readability of programs significantly. For example, it is hard on the eye to see

  ```
  for(i.where() ; i.test() ; i.next()) {
    // ... some code
  } i.erewh() ;
  ```

  when

  ```
  where(i) {
    // ... some code
  }
  ```

  could convey the same information
- By inlining the implementation of certain library calls, and performing a few mechanical transformations, major improvements in performance could be obtained.

This last item is an important issue. In the absence of a compiler or preprocessor, a programmer requiring real performance from Adlib programs will have to be prepared to go into the library at a lower level to do certain things. The *where* construct is a crucial example. The loop structure and the patterns of data access implied by this construct are actually very regular. But the interface we have defined uses library-defined iterator functions to control the loop, and calls a library-defined subscripting operator whenever an array element has to be accessed. The run-time overheads of these calls can have a severe impact on performance.

In this section we aim to provide enough detail about the implementation of the Adlib data structures to allow the programmer to avoid these overheads, by going directly into lower layers of the library.

Later in the section we expose the schedule mechanism underlying various communication routines. This will enable reuse of communication schedules, another useful optimisation. Then some aspects of memory management in Adlib are described. Finally we discuss interfacing of Adlib to external systems.

3.1 Representation of distributed arrays

The inquiry functions `rng` and `grp` were illustrated in 2.2 and 2.8. The full quota of inquiry functions on the `Array` base class is

```
int Array :: rnk() ;
int Array :: len() ;

char* Array :: dat() ;

Range*  Array :: shp() ;
Stride* Array :: lay() ;

Range  Array :: rng(const int axis) ;
Stride Array :: str(const int axis) ;

Group Array :: grp() ;

Array :: operator void*() ;
```

For each of these members there is a field in the data structure representing the array. We will preserve data abstraction by not naming the (private) fields themselves. The inquiry functions are all C++ inline functions, so accessing them through the public interface is perfectly efficient.

rnk() returns the rank of the array; 0 for the derived classes `Array0`<T>, 1 for the derived classes classes `Array1`<T>, etc.

len() returns the length in bytes of the array element data type. For any derived class `Array`R<T> this is `sizeof(`T`)`.

dat() returns the base address for (locally held) array data. This will be discussed further below.

shp() returns the vector of range structures for the array (the "shape").

lay() returns the vector of stride structures for the array, defining local memory offsets produced by subscripting dimensions of the array (the "memory layout"). These will be discussed below

rng(d) has been discussed in an earlier section. It is equivalent to `shp() [d]`.

str(d) is equivalent to `lay() [d]`.

grp() has been discussed in an earlier section.

*operator void**() is non-zero if the array is not null. The `Array` class is implemented as a handle class. The default constructor for the class creates a structure containing a null handle.

The range and process group classes have been discussed in some detail in earlier sections. The *stride* class is new. For now we only need to mention one operation on this class

```
int Stride :: offset(Subscript& i) ;
```

Given a local subscript, the `offset` member of the `Stride` class returns the corresponding offset in the local array segment.

The defining formula for the `dat` and `str` members of `Array` is just this:

$$\&\, a(s_1, \ldots, s_R) = \texttt{(T*)}\ a\texttt{.dat()} + a\texttt{.str(0).offset(}s_1\texttt{)} + \ldots + a\texttt{.str(}R-1\texttt{).offset(}s_R\texttt{)}$$

where a is of type `Array`R`<`T`>` and $s_1, \ldots, s_R$ are suitable local subscripts. To compute the address of the element given by this set of subscripts, add all the offsets, computed from the stride structures and matching subscript, to the local base address. The subscripting operators

```
T& ArrayR<T> :: operator()(Subscript&, ..., Subscript&) ;
```

are implemented as inline members of the derived classes; they directly use the above formula involving inquiry functions on the base class.

3.2 Efficient implementation of the distributed loop

Fast, concurrent traversal of distributed arrays is a performance-critical issue in data parallel programming. The implementation of the distributed loop given in section 2.3 is very easy to code, but, unfortunately, not particularly efficient.

In this section we will show how to use lower-level operations in Adlib to translate the *where* construct in a way which can be very efficient, but retains an important abstraction of the naive translation: the translation is independent of the distribution format of the construct's range.

The translation depends on the following property shared by all supported distribution formats: the portion of the range held on a particular processor consists of zero or more blocks of global index values. The index values in a block are contiguous or have a fixed stride. The Adlib `Stride`[8] has a corresponding property, namely that all array elements associated with global index values in a given block are scattered regularly in local memory. Individual elements are separated by an offset returned by

```
int Stride :: step(Block& x) ;
```

`Block` is a subclass of `Subscript`. It represents a regular block of local subscripts.

The `Index` class is derived from the `Block` class through an intermediate class `LocBlocksIndex`. The individual blocks of a range can be enumerated in terms of a set of low level iterator functions

```
void LocBlocksIndex :: begin() ;
int  LocBlocksIndex :: test() ;
void LocBlocksIndex :: next() ;
```

When an index is activated as an iterator, the parameters of the current block can be obtained from the members

```
int Block :: blockCount ;
int Block :: blockBase ;
int Block :: blockStep ;
int Block :: blockBaseSub ;
int Block :: blockStepSub ;
```

These are respectively the size of the block and the associated base and step of the global index and the `sub` compenent of the local subscript.

In general we have to take account of the effect of a *where* construct on the active process group. This can be done by invoking the `at` and `ta` members of the index class (exploiting the fact that index classes are derived from `Subscript`).

We now have all the ingredients we need for efficient implementation of the *where* construct. Consider this example

```
Array1<int> a(x) ;

Index i(x) ;
for(i.where() ; i.test() ; i.next()) {
  a(i) = i ;
} i.erewh() ;
```

An optimised version of the where construct has this structure

[8] Unfortunately the word “stride” is overloaded in Adlib. Please do not confuse a stride in a global index range with the Adlib `Stride` class, which is a generalisation of the idea of a stride in *local memory*.

```
LocBlocksIndex i(x) ;
i.at() ;
for(i.begin() ; i.test() ; i.next()) {
  ... process a block
}
i.ta() ;
```

The ellided code for processing a block could be written like this

```
i.sub = i.blockBaseSub ;
int* aData = (int*) a.dat() + a.str(0).offset(i) ;
int  aStep = a.str(0).step(i) ;
for(int l = 0 ; l < i.blockCount ; l++)
  aData [l * aStep] = i.blockBase + l * i.blockStep ;
```

Within the inner loop, which is the performance-critical section, all library calls have been eliminated[9].

In general the inner loop should be transformed one stage further as follows

```
int iGlob = i.blockBase ;
for(int l = 0 ; l < i.blockCount ; l++) {
  *aData = iGlob ;
  aData += aStep ;
  iGlob += i.blockStep ;
}
```

The message is that the inner loops generated by *where* constructs are typically very simple; a good compiler could generate excellent code from them.

Another minor optimisation in this example is to eliminate the calls to `i.at()` and `i.ta()`. The body of the construct contains no collective operations which depend on the active process group, so they are redundant.

The same kind of optimisations can be applied for quite general *where* constructs, including nested constructs accessing multi-dimensional arrays. The only analysis required to optimise a *where* is to scan its body and build a list of the arrays that are accessed inside it, so that address temporaries can be introduced and incremented.

We will finish this section with a multidimensional example. The *where* construct in

```
Array2<float> a(x, y) ;
Array1<float> b(x) ;

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

[9] It is necessary to explicitly set the `i.sub` field to point at the start of the block before calling `Stride :: offset`. The block enumeration functions will set the `Subscript :: blk` member of `i`, but not the `Subscript :: sub` field, which is multi-valued within a block.

```
for(i.where() ; i.test() ; i.next()) {
  for(j.where() ; j.test() ; j.next()) {
    a(i, j) = b(i) ;
  } j.erewh() ;
} i.erewh() ;
```

can be optimised to

```
LocBlockIndex i(x), j(y) ;
for(i.begin() ; i.test() ; i.next()) {
  i.sub = i.blockBaseSub ;
  float* aData0 = (float*) a.dat() + a.str(0).offset(i) ;
  int    aStep0 = a.str(0).step(i) ;
  float* bData0 = (float*) b.dat() + b.str(0).offset(i) ;
  int    bStep0 = b.str(0).step(i) ;
  for(int l0 = 0 ; l0 < i.blockCount ; l0++) {
    for(j.begin() ; j.test() ; j.next()) {
      j.sub = j.blockBaseSub ;
      float* aData1 = aData0 + a.str(1).offset(j) ;
      int    aStep1 = a.str(1).step(j) ;
      for(int l1 = 0 ; l1 < j.blockCount ; l1++) {
        *aData1 = *bData0 ;
        aData1 += aStep1 ;
      }
    }
    bData0 += bStep0 ;
    aData0 += aStep0 ;
  }
}
```

The point to observe is the extreme simplicity of the performance-critical inner loop.

Further simplifications are possible. If it is known at compile time that the ranges involved have simple distribution formats, with only a single block on every processor (this is the case for all currently supported distribution formats except block cyclic) the outer loops can be omitted. The `LocBlockIndex` constructor will initialize the `Block` fields to represent the unique local block.

3.3 Communication schedules

The implementation of the `remap` operation is

```
void remap(Array dst, Array src) {
  Remap schedule(dst, src) ;
```

```
    schedule.execute() ;
  }
```

The class `Remap` is derived from a base class `Schedule`. A schedule contains lists of block-memory-copy operations and low-level communication descriptors. The member

```
  void Schedule :: execute() ;
```

executes the local copy and communication operations in these lists. In general translation of a high-level data movement such as an array remap to a low-level communication schedule can be a costly procedure. This is even more true of irregular communcations such as `gather` and `scatter` where construction of the data movement schedule will itself involve communication. If the same data movement is to be repeated many times, which is the case in some important applications, it is best to construct the schedule once, and reuse it.

A schedule based version of the PDE solver from figure 2 is given in figure 5.

As the Adlib software matures the intention is to progressively document lower level interfaces of the implementation, so that a compiler can perform exactly the kind of inlining and simplification illustrated in this section and the previous one.

3.4 Memory management

Most of the major classes of Adlib are implemented as *handle classes*. A handle object contains only a pointer to some representation object. All public members of the class are inline functions which dispatch operations to the representation class. This means that copying a range, a process group, or even an array object only involves copying a pointer.

Using a standard idiom of C++, the representation objects are typically reference-counted, and only garbage-collected when there are no longer any active handles to the object.

Incrementing, decrementing and testing a reference count has a small cost. This cost is incurred whenever a handle is explicitly copied *or* passed as a procedure parameter. If it is necessary avoid this cost in a user-defined procedure interface, handle-valued procedure parameters can be passed by reference.

The Adlib classes which are implemented as handles are

```
  Dimension
  Group
  Range
  Array
```

Currently the classes `Procs`, `Subscript` and `Index` are not treated in this way. Although `Dimension` is a handle class its representation object is not reference-counted, because it always persists just as long as the process array to which

```
Procs2 p(2, 2) ;

if(p.on()) {
  Range x(N, p.dim(0), BLK), y(N, p.dim(1), BLK) ;

  Array2<float> w(x, y) ;

  // ... some code to initialise `w'

  // Create the communication schedules...

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

  Shift nx(wnx, w,  1, 0, EDGE) ;
  Shift px(wpx, w, -1, 0, EDGE) ;
  Shift ny(wny, w,  1, 1, EDGE) ;
  Shift py(wpy, w, -1, 1, EDGE) ;

  Index i(x.subrng(N - 2, 1)) ;
  Index j(y.subrng(N - 2, 1)) ;

  // Following code goes in a loop...

  nx.execute() ;
  px.execute() ;
  ny.execute() ;
  py.execute() ;

  for(i.where() ; i.test() ; i.next()) {
    for(j.where() ; j.test() ; j.next()) {
      w(i, j) = 0.25 *
          (wnx(i, j) + wpx(i, j) + wny(i, j) + wpy(i, j)) ;
    } j.erewh() ;
  } i.erewh() ;

} p.no() ;
```

Figure 5: PDE solver kernel using schedules.

it belongs. Likewise, `Index` is a handle class but its representation object is not reference-counted, in this case because there is no forseen requirement to support copying an `Index` object.

3.5 Low level array constructors: interfacing with external systems

In applications of Adlib it may be necessary to interface with libraries or programs that have their own parametrisations of distributed arrays. For example, the first application of the library was to implement the run-time library for an HPF translation system which outputs Fortran 90 code. The Fortran declares its own arrays to hold the local data segments of distributed arrays. These are passed as arguments to interface routines together with some parametrisation of the array distribution. The interface routines wrap the Fortran data in Adlib `Array` structures, which can then be processed like native Adlib arrays.

The approach is reminicent of the treatment of array sections in section 2.8, and of array promotions in section 2.13.2, but this time we need a constructor which takes the "raw data" as an argument.

The new constructor has the interface

```
Array :: Array(char* data, const Group procGrp,
               const int rank, Range* rngs, Stride* strs,
               const int typeLen, const int copy = 0) ;
```

It initialises the fields described in section 3.1 directly. The correspondence between the fields described there and the constructor parameters should be self-evident.

How are the actual arguments chosen?

Normally `data` will just be a pointer to the array segment in the external program.

The process group argument `procGrp` is more awkward. Presently the Adlib interface doesn't provide any means of specifying the mapping of its logical processes to the physical processor network. Instead the "external system" will have to abide by the mapping chosen by Adlib. This is not unreasonable, if we assume that Adlib will be responsible for communication level issues.

Assuming the external application knows how many physical processor are available (Adlib provides a global function

```
int AdlibNProcs() ;
```

which will return this number) it can, through suitable interface code, construct an Adlib process array of the desired shape, containing no more logical processes than there are physical processes. The interface code can determine the position of the local process in the process array through the `dim` inquiry function and the low-level inquiry functions

```
int Procs :: loc() ;
```

which returns a non-zero value if the local physical processor holds a logical process, and

```
int Dimension :: crd() ;
```

which returns the logical coordinate of the local processor in a partcular dimension.

In this scenario, the external application or the interface code will have a handle to an Adlib process array. This process array, or a subarray constructed from it appropriately by the interface code, is passed as the second parameter of the new `Array` constructor.

The next two arguments of the constructor are self-explanatory. We have seen at length how to construct range structures describing particular mappings.

The stride vector needs more discussion. The stride constructor has the interface

```
Stride :: Stride(Range x, int* dataLen) ;
```

The first argument is the range for the associated argument of the array. The easiest way to understand the second argument is probably to think of a `Stride` object as defining a one-dimensional array. In languages like C multi-dimensional arrays are implemented as one-dimensionsal arrays of arrays. At some level of abstraction the same is true of multi-dimensional arrays in most languages, and in Adlib. Now the value of `*datalen` before the call to the stride constructor is the length in bytes of the elements of this one-dimensional array (which are typically array segments themselves). The constructor side-effects to change `*datalen` to the length in bytes of the new array segment.

Now by example: suppose `rngs` is the vector of ranges of our array, and `typeLen` is the length of its element. The code fragment

```
int size = typeLen ;
for(int i = 0 ; i < rank ; i++)
  strs [i] = Stride(rngs [i], &size) ;
```

will build a `strs` vector describing a multi-dimensional array where the first range is "most-rapidly varying" in memory. The fragment

```
int size = typeLen ;
for(int i = rank - 1 ; i >= 0 ; i--)
  strs [i] = Stride(rngs [i], &size) ;
```

will build a `strs` vector describing an array where the last range is most-rapidly varying.

The exact amount of space allocated by the stride constructor, and the exact mapping of elements to offsets determined by the `Stride` structure, is an area of potential mismatch between Adlib and the external application. The Adlib layout is documented in the manual section. If the external application requires a different layout, the Adlib `Range` hierarchy must be extended.

The penultimate argument of the array constructor is the length of the array element type.

The final argument defines whether the range and stride vectors passed in to the constructor are to be copied, or whether references to these vectors are simply installed in the `Array` record. If the vectors will persist as long as the constructed `Array` it is cheaper not to copy (this is the default). If not, the constructor must copy them.

The low-level array constructor has potential uses in optimisating Adlib programs, independent of its use in defining external interfaces. For example, if a section of an array is to be constructed inside a *where* construct, subscripted by the range of the construct, one should be able to use the tricks of section 3.2 to compute the base address of the section's data, then install this address together with appropriate ranges and strides from the parent array in a new array constructed by means of the low-level constructor. As a simpler example, if several arrays of the same type and identical shape and mapping were declared consecutively, the low-level constructor could be used to allow them to share stride and range vectors.

3.6 I/O

The present implementation of Adlib provides only rudimentary support for I/O. There are two procedures called `gprintf` and `gscanf` which have identical specifications to the standard C routines `printf` and `scanf`. These routines can be used for output and input of *global* values and variables. A global value is one replicated over the process group active when the Adlib program starts (the *initial process group*).

For distributed data the only handle available is the following guarantee:

> If a rank-0 process array is declared when the initial process group is active, the unique process in the new process array is mapped to a processor supporting standard C I/O.

So a distributed array, or a section of an array, can be remapped to an array distributed over a scalar processor array p, then ordinary C I/O operations are allowed in an $on(p)$ construct.