Adlib Language Reference
[DRAFT]

D.B. Carpenter

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

November 1, 1996

Contents

Adlib is implemented as a C++ class library, and Adlib programs can be compiled with any standard C++ compiler. But a well-defined class library effectively extends the programming language in which it is implemented. This document introduces the basic principles of Adlib programming, emphasizing the high-level, language-semantic view of the library. It deliberately omits details of lower level classes, even where direct access to these classes may be highly desirable for efficiency reasons. As a rule, code inlining and optimization should be a mechanical procedure, once the program is expressed in terms of the higher level constructs introduced here.

1 Process groups

The Adlib model of distributed data is underpinned by a handful of closely related classes for describing sets of processes. The classes involved are

- the *process grid*,
- the *process dimension*,
- the *process coordinate*, and
- the *process group*—a subset of a process grid.

Important associated concepts are the *active process group*, which abstracts the set of processes cooperating in a single "loosely synchronous" thread of control, and the "distributed control" constructs *on* and *at* which manipulate this group.

1.1 Process grids and dimensions

A *process grid* is a multi-dimensional set of processes. Any process grid is a member of the class `Procs`. Instances are typically created through constructors for the subclasses `Procs0`, `Procs1`, `Procs2`, ..., which represent rank-0, rank-1, rank-2, ..., process grids. (A rank-0 process grid is a scalar "grid" containing a single virtual process.)

The standard constructor for a rank-`R` process grid takes `R` positive integer arguments—the extents of grid. The specified size of the grid—the product of the extents—must be less than or equal to the size of the process group active at the time of declaration of the grid[1]. At most one process from any particular grid is mapped to each available physical process.

Examples of process grid declarations:

```
Procs2 p(2, 2) ;
Procs0 s ;
```

[1] The term "active process group" will be precisely defined later.

For convenience default constructors are also provided for `Procs1`, `Procs2`, ... These will choose the extents of the constructed grid by some heuristic, taking account of the available number of processes.

A rank-`R` process grid has `R` associated *process dimensions*. A process dimension is a member of the class `Dimension`. The dimensions of a process grid are accessed through the inquiry

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

where `d` is in the range `0, ..., R - 1`. For example

```
Dimension p1 = p.dim(0), p2 = p.dim(1) ;
```

`p1` is initialized to the first dimension of `p`, and `p2` to the second.

The extent of a process dimension can be determined through the member function

```
int Dimension :: size() ;
```

For example

```
const int n1 = p1.size(), n2 = p2.size() ;
```

`n1` is initialized to the extent of `p1` and `n2` to the extent of `p2`. In the example `n1` or `n2` will both be 2. The position of the local process in a process dimension can be determined through the member function

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

This returns a value in the range `0, ..., n - 1`, where `n` is the extent of the dimension.

We will see later that there are ways to obtain a null process dimension. It is illegal to apply the `size` or `crd` inquiry functions to a null dimension.

1.2 Coordinates

A *coordinate* denotes a position in a particular process dimension. It is a member of the class `Coord`.

```
struct Coord {
  Dimension dim ;
  int crd ;

  Coord(int _crd, Dimension _dim) ;
} ;
```

A `Coord` is a simple record containing two publicly accessible data members: `dim` and `crd`. `dim` is the process dimension in which the coordinate is effective; `crd` is the coordinate itself. `crd` must be in the range `0, ..., n - 1`, where `n` is the extent of the dimension. For example, assuming the definitions of `p1`, `p2` given in the previous section,

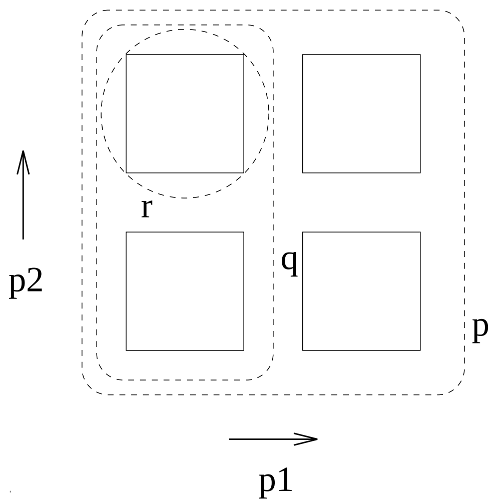

Figure 1: Examples of process groups. The square boxes represent the 4 processes in the process array p. The dashed lines embrace groups p, q and r.

```
Coord c(0, p1), d(1, p2) ;
```

declares `c` as a coordinate in `p1` and `d` as a coordinate in `p2`.

Although the data members of `Coord` are publicly accessible, it is not normally necessary to read or write them directly. It is even quite unusual to create a `Coord` object by direct use of its constuctor. More commonly these objects are created implicitly by creating objects of the derived class `Subscript`.

By definition, a null coordinate is one with a null `dim` field and an undefined `crd` field.

1.3 Groups

A (process) *group* may be a whole process grid or a *restricted group*, which is some "slice" of a process grid. In any case, a group is a member of the class `Group`.

A restricted group is constructed with the operator

```
Group Group :: operator/(Coord& c) const ;
```

For example, assuming the definitions of `p`, `c` and `d` given in the previous two sections,

```
Group q = p / c ;
Group r = q / d ;
```

These groups are illustrated in figure 1. The coordinate operand must be a coordinate in a dimension of the group operand. The dimension set of the restricted group is the dimension set of the operand group minus the dimension of the restricting coordinate. In the example, the dimension set of `q` is just `p2`, and dimension set of `r` is empty.

Applying a null coordinate to a group with the / operator returns the original group, unchanged.

1.4 The Active Process Group and the *on* construct

At any point of program execution a particular group is singled out as the *active process group*. This is the group of processes cooperating in the current "thread of control". Before explaining how the active process group is set, we need a couple of definitions.

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

1. `q` is a process grid and `p` was the active process group when `q` was declared (constructed), or
2. `q` is a sub-group of `p`, constructed by the / operation, or
3. `p` contains some group `r` which in turn contains `q`—"contains" is a transitive relation.

These cases are illustrated by example at the end of this section. As a matter of convention we will also say that a group contains itself.

A group is *accessible* at a particular point in program execution if it is contained in the active process group.

The active process group can be altered through the *on* construct. If `p` *is an accessible group* and `S` is some statement, the *on* construct takes the form

```
if(p.on())
  S
p.no() ;
```

The `on` member returns a truth value which is non-zero if the physical process executing the code holds a process from `p`. So the code inside the construct is only executed by processors that hold part of `p`. After the `p.on()` call, `p` is the designated active process group. It loses this status after the `p.no()` call—the earlier active process group is restored.

The idiom for the *on* construct is probably best regarded as an atomic language construct, and the macros

```
#define on(p) if(p.on())
#define no(p) p.no()
```

are sometimes defined so that the *on* construct can be written slightly more compactly as[2]

```
on(p)
  S
no(p) ;
```

Finally we will illustrate the definition of the "contains" relation. Take the example:

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

  Coord c(0, r.dim(0)) ;
  Group q = r / c ;
} no(p) ;
```

`p` contains `r` by rule 1 above (`p` is the active process group at the point of definition of `r`), `r` contains `q` by rule 2, and `p` contains `q` by rule 3.

1.5 The Initial Process Group

The group active when the Adlib program starts is the *initial process group*. Only limited information is available or needed about the structure of this group. Its total size will be the number of physical processes available. This value is returned by the global function

```
int AdlibNProcs() ;
```

1.6 The *at* construct

Two function members of the `Coord` record were ellided above:

```
int  Coord :: at() ;
void Coord :: ta() ;
```

Suppose the active process group is `p`, `c` is a coordinate in a dimension of `p`, and `S` is some statement. The *at* construct takes the form

```
if(c.at())
  S
c.ta() ;
```

[2]Syntactically this construct resembles the atomic `do S while(E) ;` statement of C or C++. This is slightly deceptive. The *on* construct is actually composed of two separate C++ statements: `on(P) S`, followed by `no(P) ;`. This is an inconvience imposed by the macro implementation.

The `c.at()` member returns a truth value which is non-zero if the physical process executing the code holds a process with the coordinate defined by `c`. After the `c.at()` call, the active process group is `p / c`. The earlier active process group is restored after the `c.ta()` call. Again, macros `at` and `ta` are sometimes defined so that the *at* construct can be written more compactly as

```
at(c)
  S
ta(c) ;
```

The *at* construct can be viewed as a special case of the general *on* construct. An `at(c)` statement is essentially equivalent to an `on(p / c)` statement. However, the `at` version can be implemented more efficiently where it is applicable, and is often more convenient in such circumstances.

2 Index ranges

The process groups introduced in the previous sections are one of two basic components from which Adlib distributed arrays are built. The other primitive structure is the index range. In general an index range denotes a mapping of some interval `0, ..., N - 1` into a process dimension. The index range is represented by a member of the `Range` class.

2.1 Primitive ranges: distribution formats

There are presently three `Range` constructors that create *primitive ranges*. A primitive range is analogous to a dimension of an HPF *template*. It is typically distributed over some process dimension. The constructors are

```
Range :: Range(const int extent,
               const Layout layout = STANDARD) ;

Range :: Range(const int extent, Dimension dim, const Format fmt,
               const Layout layout = STANDARD) ;

Range :: Range(const int extent, Dimension dim, const Format fmt,
               const int blockSize,
               const Layout layout = STANDARD) ;
```

The different constructor create ranges with different distribution formats. In all cases the first argument is the extent of the created range.

The first constructor creates a "collapsed" or sequential range. The whole of the index range is mapped to the local process. The other two constructors create *distributed* ranges.

The second constructor creates a range with "block", "simple cyclic" or "stepped" block distribution format. The `fmt` argument takes one of the values `BLK`, `CYC` or `STP`. Block and simple cyclic distribution are identical to the corresponding HPF distribution formats. In *stepped* block distribution the first `N mod P` processes are assigned `B + 1` range elements and the remaining processes are assigned `B` elements, where `B = [N / P]`. Here `N` is the extent of the range and `P` is the extent of the process dimension.

The third constructor creates a range with block or block cyclic distribution format, taking a user-specified block size. The `fmt` argument takes one of the values `BLK`, `CYC`. These distribution formats are identical to the corresponding HPF formats.

The optional `layout` parameter of these constructors affects the mapping of array elements into memory. It will not be discussed in this document.

There is an *orthogonality* relationship between certain primitive ranges. Primitive range `x` is orthogonal to primitive range `y` if

1. `x` and `y` are distributed over distinct dimensions of *the same* process grid.

2. `x` and `y` are distinct, and either or both are collapsed.

Two primitive ranges are distinct if they are created in different calls to one of the constructors above.

2.2 Subranges

The function

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

creates a subrange. Like any other range, a subrange has global indices in the range `0, ..., N - 1`, where `N` is the extent of the range. However, a subrange retains an alignment relation to its parent range (and ultimately to its parent *primitive* range, if the immediate parent is also a subrange).

Element `i` of the new range is aligned with (mapped to the same processor as) element

```
base + stride * i
```

of the parent range.

Subranges inherit the orthogonality properties of their parent primitive range. Subranges `x` and `y` are orthogonal if and only if their parent primitive ranges are orthogonal.

2.3 Inquiries

The members of `Range` include the inquiries

```
int size() ;

Dimension dim() ;

Range prm() ;
int bas() ;
int str() ;
```

Which return, respectively, the extent of the range, the dimension over which it is distributed (or a null dimension for a collapsed range), the parent primitive range of this range, and its alignment base and stride relative to the primitive range.

2.4 Subscripts

A *subscript* is an abstraction of the idea of a "local subscript". It denotes a particular position in a particular primitive range.

A `Subscript` is implemented as a simple record containing two publicly accessible data members in addition to the members of its base class `Coord`:

```
struct Subscript : public Coord {
  int blk ;
  int sub ;
} ;
```

`blk` and `sub` are components of a local subscript. Their precise interpretation depends on the implementation of the parent range, and is not strictly prescribed by the general Adlib framework. The *expected* use is as follows.

- `sub` is an offset within a local block—for simple distribution formats it is "the" local subscript.
- `blk` is intended for use in "multi-block" distribution formats such as *block cyclic* distribution. It identifies a particular local block, of which there may be several. For simple distribution formats with only a single block per process, the `blk` field is unused.

A subscript record will normally be initialized by one of the members of the `Range` class introduced in the next two sections. Although its fields are publicly accessible, it is not normally necessary to read or write them directly.

2.5 Subscripting a range

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

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

These are address translation operations. They convert between global index values and local subscripts. A global index is an `int` value from `0, ..., N - 1`, where `N` is the extent of the range. A local subscript is a member of the class `Subscript`.

The `operator()(const int i)` takes a global subscript into this range, and returns a reference to a correponding local subscript. For collapsed distribution formats the returned subscript contains a null process coordinate.

The operator `operator()(Subscript& s)` takes a local subscript and returns the corresponding global subscript into this range. The argument must be aligned to an element of the range, otherwise the operation is illegal. Typically, the argument was originally created from this range, or from one of its subranges.

For example, suppose `x` is some range of extent 75 or greater:

```
Range y = x.subrng(50, 25) ;  // y(i) aligned with x(25 + i).

Subscript s(y(25)) ;          // ‘s’ aligned with element 25 of ‘y’.

int i = y(s) ;                // initialize ‘i’ to 25

int j = x(s) ;                // initialize j’ to 50 (because
                              // ‘s’ aligned with element 50 of ‘x’).
```

2.6 The index class and the *where* construct

The class `Index` is used as a distributed loop index. The constructor takes a single range as its argument.

```
Index :: Index(const Range& x) ;
```

The Adlib distributed loop is called a *where* construct. The function members

```
void Index :: where() ;
int  Index :: test()  ;
void Index :: next()  ;
void Index :: erewh() ;

operator Index :: int() ;
```

are used to implement this construct. The `Index` class has `Subscript` as a base class.

Suppose the active process group is `p`, `x` is a range which is (collapsed or) distributed over a dimension of `p`, `i` is an index for this range, and `S` is some statement. The *where* construct takes the form

```
for(i.where() ; i.test() ; i.next())
  S
i.erewh() ;
```

Between the calls to `i.where()` and `i.erewh()` the index `i` is "activated as an iterator". As with the earlier control constructs, macros `where` and `erewh` are sometimes defined so that the *where* construct can be written more compactly as

```
where(i)
  S
erewh(i) ;
```

The *where* construct implements a distributed loop, as follows

- The statement `S` executes once for every element of the range held on the local processor.
- While an index is activated as an iterator, the `Subscript` component of the index is equal to the local subscript associated with the current iteration, and the type conversion `operator int()` allows the index to be coerced to the global subscript associated with the current iteration.
- During an iteration of the loop, the active process group is reduced to `p / i`, assuming the coercion from index to `Subscript` described above.

For example, suppose `x` is some range of extent 75 or greater, distributed over a dimension of the active process group

```
Range y = x.subrng(50, 25) ;  // y(i) aligned with x(25 + i).

Index j(y) ;
where(j) {
  int i = j ;                 // initialize 'i' to current global index.

  Subscript s = j ;           // 's' aligned with element 'i' of 'y'.

  int k = x(s) ;              // initialize 'k' to 25 + i.

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

The main use of the *where* construct is in traversing distributed arrays. More meaningful examples must be postponed until the next section.

Normally the conversion of an index to an integer can be inserted automatically by the compiler—it will be forced by context. Occasionally an explicit cast to `int` may be required.

3 Distributed arrays

A *distributed array*, or simply an *array*, is a multi-dimensional distributed data aggregate. An array is a member of the class `Array`. Instances are usually created through constructors for the subclasses `Array0`, `Array1`, `Array2`, ..., which represent rank-0, rank-1, rank-2, ..., arrays. These subclasses are template classes, parametrized by the type of the element.

3.1 Array constructors and inquiries

The standard data constructor for a rank-`R` array takes `R + 1` arguments—an accessible process group, `p`, followed by a list of `R` range objects. These ranges must all be orthogonal, and any distributed range must be distributed over a dimension of `p`. The array as a whole is said to be "distributed over `p`".

```
Procs1 p(4) ;
Range x(100, p.dim(0), BLK), y(100) ;

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

Array0<float> b(p) ;
```

A second form of the constructor omits the process group argument. This form is equivalent to passing the active process group as this argument.

```
Procs1 p(4) ;
on(p) {
  Range x(100, p.dim(0), BLK), y(100) ;

  Array2<int> a(x, y) ;

  Array0<float> b ;
} no(p) ;
```

The constructors allocate space to hold an array element for each locally held subscript combination.

We should emphasize that because arrays can be declared using arbitrary subranges of primitive ranges, and because they can be distributed over arbitrary process groups including restricted groups, Adlib fully supports the affine alignment model of HPF.

At a particular point in program execution an array is *accessible* if it is distributed over an accessible process group.

The members

```
Group Array :: grp() ;

Range Array :: rng(int d) ;
```

respectively return the group over which an array is distributed, and one of its ranges. `d` is in the range `0, ..., R - 1` where `R` is the rank of the array.

3.2 Subscripting and scalar access rules

The operators

```
T& Array0<T> :: operator()() ;

T& Array1<T> :: operator()(Subscript& s_1) ;

T& Array2<T> :: operator()(Subscript& s_1, Subscript& s_2) ;

...
```

are used to subscript an array. Here `T` is a type parameter. These operations look like function applications, taking local subscript records as arguments. They return a reference to the associated local array element.

Some fundamental definitions

- Any data object that is not a distributed array is a *scalar*. In particular distributed array *elements* are scalar.
- Any scalar data object is *replicated over* some group (which may consist of a single process).
- A simple variable (not a distributed array element) is replicated over the group active at the point at which it is declared or constructed.
- If `a` is an array distributed over `p`, the array element

  ```
  a(s_1, s_2, ...)
  ```

  is replicated over the group

  ```
  p / s_1 / s_2 / ...
  ```

  (recall that `Subscript` is derived from `Coord`, so a local subscript can stand for a process coordinate).

Now, the Adlib *access rules* for scalar objects:

> A scalar may be *read* only if it is replicated over a group containing the active process group.

and

> A scalar may be *written* only when the active process group is identical to to group over which the object is replicated

The first rule forbids reading of variables not stored locally. The second rule forbids writing to variables not stored locally, and ensures that replicated variables are always updated with globally consistent values.

For example

```
Procs1 p(4) ;
on(p) {
  Range x(100, p.dim(0), BLK) ;

  Array<int> a(x) ;

  int v ;

  Subscript s(x(25)) ;

  v = a(s) ;     // illegal!

  a(s) = 17 ;    // illegal!

  at(s) {
    v = 17 ;     // illegal!
  } ta(s) ;

} no(p) ;
```

The first assignment is illegal by the first rule, because the `a(s)` is only replicated over `p / s` while the active process group is the whole of `p`. The array access is non-local for most members of `p`. The second assignment is forbidden by the second rule, for much the same reason. This time most members of `p` would be *writing* (identical values) to a non-local location. The third assignment is also forbidden by the second rule. It occurs while the the active process group is `p / s`, but `v` is replicated over the whole of `p`. In this case a single process would have write to locations on all other processes in `p` or else the result would be inconsistent values for `v` in different processes.

Finally, a legal Adlib program, which illustrates much of the infrastructure established so far

```
Procs2 p(2, 2) ;
on(p) {
  Range x(100, p.dim(0), BLK), y(10, p.dim(1), BLK) ;

  Array2<int> a(x, y) ;

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

```
  } erewh(i) ;

} no(p) ;
```

The array `a` is distributed over `p`. The active process group inside the nested *where* constructs is `p / i / j`, assuming the coercion from `Index` to `Subscript` (and thus `Coord`) effective inside a *where*. The element `a(i, j)` is defined thtrough the same cooercion, and is replicated over the same process group, so the update is legal. On the right hand side of the assignment the coercion of `Index` to `int` is employed, yielding the global subscript value for the loop iteration. The net result is to assign the sum of the global subscript values to each element of the distributed array.

3.3 Array sections

Adlib supports array sections, analogous to Fortran array sections, through the members

```
Array0<T> Array0<T> :: sect() ;


Array1<T> Array1<T> :: sect(Range x_1) ;

Array0<T> Array1<T> :: sect(Subscript& s_1) ;


Array2<T> Array2<T> :: sect(Range x_1, Range x_2) ;

Array1<T> Array2<T> :: sect(Range x_1, Subscript& s_2) ;

Array1<T> Array2<T> :: sect(Subscript& s_1, Range x_2) ;

Array0<T> Array2<T> :: sect(Subscript& s_1, Subscript& s_2) ;


...
```

These return new array objects referencing a section of the data in the original. The arguments must be subranges of, or subscripts in, the corresponding ranges of the parent array[3].

Suppose `x` and `y` are ranges of extent 100 distributed over dimensions of the active process group, and `a` is declared as

[3]Because these functions accept an arbitrary mixture of range and subscript arguments there are 2 to the `R` members `sect` defined for the rank-`R` array class. This is slightly extravagant, but it gives better type security than the alternatives.

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

This is comparable with the Fortran 90 declaration

```
INTEGER A(0 : 99, 0 : 99)
```

Some examples of sections are

```
a.sect(x, y) ;                    // A ( : , : )

a.sect(x.subrng(50, 25), y) ;     // A (25 : 74, : )

a.sect(x, y.subrng(25, 25, 2)) ;  // A ( : , 25 : 73 : 2)

a.sect(x, y(50)) ;                // A ( : , 50)
```

The first example returns a section describing the whole of the parent array (the section is essentially indistinguishable from the parent). The second and third examples use subranges to emulate Fortran "triplet subscripts". The fourth example uses a scalar subscript to create an array of rank lower than the parent array.

If the parent array is distributed over `p` and the scalar subscripts passed to `sect` are `s_1, s_2, ...`, the constructed section is distributed over `p / s_1 / s_2 / ...`. A call to `sect` is legal provided the *constructed* section is distributed over an accessible process group. This does not necessarily require that the parent array is accessible—consider this example:

```
void foo(Array1<int>) ;

Array2<int> a(x, y) ;

Subscript s(x(25)) ;
at(s)
  foo(a.sect(s, y)) ;
ta(s) ;
```

The section constructor creates a rank-1 array representing an individual column of `a`. If the active process group is originally `p` the section is distributed over `p / s`. So the call to the section constructor is legal, even though the whole of `p` (and thus `a`) is not accessible inside the *at* construct. For a more useful application of this rule, the *at* could be replaced by a *where*:

```
Index i(x) ;
where(i)
  foo(a.sect(i, y)) ;
erewh(i) ;
```

Here `foo` is applied in parallel to every column of `a`.

4 Communication

4.1 `shift`

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 amount, const int dim, const int mode) ;
```

The parameters `dst` and `src` are arrays, which must have identical type, shape and mapping (they share the same process group and range list). The operation shifts the values in `src` by `amount` places in the `dim` dimension, and puts the result in `dst`. `amount` is a signed integer. `dim` is in the range `0, 1, ..., R - 1`, where `R` is the rank of the arrays. `mode` is one of `CYCL` or `EDGE`, selecting either cyclic or "edge-off" shift.

The array arguments of `shift` must be accessible. *All* Adlib collective communication routines impose this requirement on their array arguments.

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. As an application of subranges, the *where* construct in figure 2 could be replaced by

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

where(i) {
  where(j) {
    w(i, j) = 0.25 *
        (wnx(i, j) + wpx(i, j) + wny(i, j) + wpy(i, j)) ;
  } erewh(j) ;
} erewh(i) ;
```

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.

Figure 3 performs an iteration of Conway's Life.

It is important to realize that there are no restrictions on calling Adlib communication routines beyond those that have already been stated. It is perfectly legitimate to make several calls in parallel, for example. The following code invokes `shift` in parallel on the columns of a pair of arrays to implement a skewed shift:

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

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

  Array2<float> w(x, y) ;

  // ... some code to initialize '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) ;
  where(i) {
    where(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)) ;
    } erewh(j) ;
  } erewh(i) ;

} no(p) ;
```

Figure 2: PDE solver kernel.

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

  Array2<int> c(x, y) ;

  // ... some code to initialize '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) ;
  where(i) {
    where(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 ;
      }
    } erewh(j) ;
  } erewh(i) ;

} no(p) ;
```

Figure 3: Conway's Life iteration.

```
Index j(y) ;
where(j)
  shift(a.sect(x, j), b.sect(x, j), j, 0, CYCL) ;
erewh(j) ;
```

4.2 remap

A much more powerful cousin of `shift` is `remap`. The deceptively simple interface of `remap` is

```
void 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.

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

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

The code is comparable to the Fortran

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

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

Because the destination array may contain replicated data, the `remap` operation can also implement a wide variety of broadcast and multicast operations.

4.3 Gather-scatter operations

Besides the family of regular data remapping operations Adlib provides a *gather-scatter* family. This include various operations that allow arbitrary subscripting patterns in access to distributed 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 the gathered data, `b`. The number of subscripts should be equal to the rank of the scattered data, `a`, and their values should lie in the ranges expected for global 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.

4.4 Reductions

Finally Adlib provides a set of reduction operations based on the corresponding Fortran 90 array intrinsics. These include boolean operations

```
int all(Array_<int> src) ;
int any(Array_<int> src) ;
int count(Array_<int> src) ;
```

and arithmetic operations

```
T maxval(Array_<T> src) ;
T minval(Array_<T> src) ;
T sum(Array_<T> src) ;
T product(Array_<T> src) ;
```

Here `_` stands for an arbitrary rank and `T` stands for an arithmetic type: `int`, `float` or `double` (a defined `complex` type is also recognised by `sum` and `product`).

Versions which reduce in a single dimension, returning an array reduced in rank by one, are also provided:

```
void all(Array_<int> res, Array_<int> src, int dim) ;
void any(Array_<int> res, Array_<int> src, int dim) ;
void count(Array_<int> res, Array_<int> src, int dim) ;
```

and

```
void maxval(Array_<T> res, Array_<T> src, const int dim) ;
void minval(Array_<T> res, Array_<T> src, const int dim) ;
void sum(Array_<T> res, Array_<T> src, const int dim) ;
void product(Array_<T> res, Array_<T> src, const int dim) ;
```

as well as the search operations

```
void maxloc(int* res, Array_<T> src) ;
void minloc(int* res, Array_<T> src) ;
```

which return the subscripts of maximum and minimum elements.

5 I/O

Support for I/O comes in the form of variations on the standard C functions `printf` and `scanf`. Two variants of printf are

```
void gprintf(char* control, ...) ;
void aprintf(char* control, ...) ;
```

`gprintf` outputs global values (values replicated over the active process group). The control string is identical to the control string for the standard C function. Only the root process in the active process group will actually perform any output.

`aprintf` takes a list of arrays as arguments. The argument arrays should all have the same shape. The control string (and any format modifiers interspersed amongst the array arguments) should be global values. The print operation is performed "elementally", equivalent to calling `printf` in turn for every set of global subscripts of the arrays. In addition to all the standard format specifiers, `aprintf` recognises two others.

- `R`*D* format, where *D* is a digit in the range `0, ..., 9` outputs the current value of the *D*th subscript. The `R` may be preceded by the same format modifiers as for standard decimal integer format, `d`.
- `N` format outputs a newline. It can be preceded by one format modifier, either a decimal number or an asterisk, to specify the frequency with which the newline is printed. If the value of the modifier is `n`, the newline is only printed in every `n`th elemental output. The `N` specifier is ignored in the intervening outputs.

Note that neither `R` or `N` format specifiers correspond to any argument of the `aprintf` (unless they have asterisk format modifiers). They effectively alter the behaviour of the control string according to the current subscript set.

For input the functions

```
void gscanf(char* control, ...) ;
void ascanf(char* control, ...) ;
```

are provided. `gscanf` inputs global values. Only the root process in the active process group actually performs the inputs. The input values are then broadcast to all members of the active process group. `ascanf` has the same relation to `scanf` as `aprintf` has to `printf`. It inputs a list of arrays of the same shape. Both these functions take control strings identical to the control strings of the standard C function.

For file I/O the variants

```
void gfprintf(FILE* fd, char* control, ...) ;
void gfscanf(FILE* fd, char* control, ...) ;
```

```
void afprintf(FILE* fd, char* control, ...) ;
void afscanf(FILE* fd, char* control, ...) ;

FILE* gfopen(char* name, char* mode) ;
void gfclose(FILE* fd) ;
```

are provided.