NPAC PCRC Runtime Kernel (Adlib) Definition [DRAFT]

Bryan Carpenter, Guansong Zhang, and Yuhong Wen

CRPC-TR97726
Revised January 1998

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted in November 1997

NPAC PCRC Runtime Kernel (*Adlib*) Definition *[DRAFT]*

Bryan Carpenter, Guansong Zhang and Yuhong Wen

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

January 16, 1998

Contents

Chapter 1

Introduction

The library defined in this document (informally called *Adlib*) is designed as a common runtime kernel for manipulating HPF-style regular distributed arrays. It is not expected that the kernel interface described here will be used directly by a programmer, or even appear directly in code generated by a compiler. In either case, additonal interface code, built on top of the kernel, will probably be needed.

One example of a user-level interface to the kernel (effectively the *native* user-level interface to library) is the *ad++* interface. This is a set of C++ header files which define type-secure template classes—*container classes*—representing distributed arrays, various macros implementing *distributed control* constructs, and template functions for performing transformations on distributed arrays. The latter functions fulfil a role something like the array transformational intrinsics of Fortran 90. In particular, they abstract communication.

Compiler-level interfaces provide any additional run-time structure assumed by the distributed array model of the source language, and make the kernel callable from the target language of the compiler. In principle this target language could be machine code, but historically the library has been used by source-to-source translators targetting Fortran 77 or Fortran 90. So existing "compiler-level" interfaces to the kernel are actually Fortran bindings. A Java interface is under development.

The kernel itself provides:

- A distributed array descriptor (DAD), implemented as a series of C++ classes.

- A large set of *communication schedules* implementing Fortran-90 like array transformational functions and other basic communication operations.

- As part of the DAD—support for “distributed control”. In practise what this means is support for “distributed loops”: support for translation of parallel loops whose index range is distributed over some processor set.

This document is not supposed to be read in isolation. The underlying model of data distribution should be understood, for example by learning one of user-level interfaces to the library, before attempting direct use of the kernel.

The next three chapters define the components of the Adlib array descriptor—groups, ranges, then the array descriptor record itself. Chapter 5 describes the communication library. This will followed by a chapter on the `ad++` interface, then a chapter about using the Adlib descriptor technology in translation of distributed loops. A final chapter will discuss the implementation of the communication schedules.

Chapter 2

Process Groups

The Adlib idea of a process group is somewhat analogous to the MPI idea of *group*. The Adlib version is more specialized. Adlib groups are frequently created and modified in the runtime. For efficiency it is important to provide a specialized lightweight representation—one cannot afford to import the general group technology of MPI. More specifically, Adlib needs the structure of multi-dimensional grids without the overheads involved in building an MPI *Cartesian communicator*.

The most direct analogue of the MPI Cartesian communicator is the Adlib *process array*, represented by a `Procs` object. Even in Adlib, constructing a process array is a relatively expensive procedure. It involves initializing several sub-structures. The *general* group object of Adlib—class `Group`—has a much more streamlined implementation. `Group` is essentially a handle class. Its objects contain a pointer to a parent `Procs` object plus a few extra words defining a subset of that Cartesian grid. Group objects are created, copied, changed, and discarded freely within the library code. They can be handled with similarly free rein in user code.

2.1 General Definitions

Initial process group: The set processes on which an Adlib program is initiated.

Active process group: The set of processes notionally sharing the local thread of control. This may be the initial process group or it may be an arbitrary subset of the initial process group. When an Adlib program starts the active process group is the initial process group.

The logical active process group may be changed by operations that cause some processors to skip a block of code (leaving the remaining processors active for that block), or cause the current active process group to be partitioned into a set of smaller groups, with each group acting independently for some time.

Every process executing an Adlib program must maintain a static data structure, `apg`, describing the current active process group. This variable is read by collective functions in the library to determine which processes are cooperating in the operation. If the logical active process group changes, the variable `apg` must be updated (see section 2.7) before invoking a collective operation. During the library initialization procedure, `apg` is set to a value describing the initial process group.

Collective object: A collection of similar objects, one in every process from a particular group, which can also be viewed as a single logical entity.

Collective operation: An operation executed cooperatively by all members of the active process group. Typically this will take the form of a call to the same function with the same arguments. In this context two arguments are "the same" if they are simple expressions with the same value, or are references to local components of the same collective object.

A collective operation may or may not involve synchronization between members of the active process group. In general the programmer should work on the (worst case) assumption that synchronization *is* implied. If any member of the active process group fails to engage in a collective operation which others are executing, or executes collective operations in a different order to other members of the group, the program is liable to fail.

Examples of collective operations are constructors for collective objects and collective communication operations.

Process array: Some subset of the initial process group, organized in a Cartesian grid.

Coordinate: Dimensions of a process array are labelled by *coordinates*.

Process group: In general, either a process array, or some slice of a process array defined by uniquely fixing the coordinate in one or more of its dimensions.

Process id: An identifier relative to a particular process group[1]. If the size of the group is P, the id is in the range $0 \ldots P-1$. Relative to a one-dimensional process array, the value of the process id coincides with the value of the unique coordinate. The id relative to the initial process group is also called the *absolute process id*.

[1] Called a *rank* in MPI. Following Fortran, we will reserve "rank" for the dimensionality of arrays or process arrays.

2.2 class Procs

A *process array* is a set of processes organized in a multi-dimensional grid. A process array is described by a collective object with local components of class Procs. The public interface of the Procs class is

```
class Dimension ;
class DimensionSet ;
class Coord ;
class Group ;

class Procs {
public :
  Procs(const int _rank, int* _n, int* subIds = 0) ;
  ~Procs() ;

  int rnk() const ;
  Dimension dim(const int r) const ;

  DimensionSet dims() const ;

  int member() const ;

  int size() const ;

  int id() const ;

  int id_abs(const int id) const ;

  int lead_abs() const ;

  Group operator/(const Coord& i) const ;

private :
  Procs() ;
  Procs(const Procs& p) ;

  ...
} ;
```

2.2.1 Constructors and destructors

`Procs(const int _rank, int* _shape, int* ids = 0)`

The normal constructor. Initializes a Procs object describing a process grid of rank _rank and shape _shape. The rank is a non-negative integer. If it is zero the new group is a scalar “grid” containing a single process. The shape is a vector of `_rank` positive integer extents. The size, P, of the new grid is the product of the elements of `_shape`. This must be less than or equal to the size of the active process group.

If the optional argument `ids` is specified, this should be a vector of P distinct values, each in the range $0, \ldots, A-1$. A is the size of the group defined by the current value of `apg`. It defines a mapping of the new process group into the active process group. If the `ids` argument is omitted, the library selects an arbitrary mapping. In either case if P is less than A, some members of the active process group are not identified with any member of the new grid. At most one member of the new grid may be mapped to each process of the active group.

This constructor is a collective operation, and calls to it should obey the normal rules for collective operations.

`Procs()`

The default constructor is *private*. Creating an uninitialized `Procs` object is not allowed in normal code.

`Procs(const Procs& p)`

The copy constructor is *private*. Copying a `Procs` object is not allowed in normal code.

`~Procs()`

Destructor.

2.2.2 Methods

`rnk()`

An inquiry function returning the rank (dimensionality) of the grid.

`dim(const int r)`

An inquiry function returning an object that describes the `r`th dimension of the grid. The argument `r` is in the range $0, \ldots, R-1$ where R is the rank of the grid.

`dims()`

An inquiry function returning an object that describes the complete set of dimensions of the grid.

`member()`

An inquiry function which returns 1 if a process from the grid is mapped to the local process, and 0 if not. In other words, a boolean function that returns true iff this process is a member of the grid.

`size()`

An inquiry function returning the total number of processes, P, in the grid.

`id()`

An inquiry function that returns the id of the local process relative to this grid. Its value is defined if and only if the local process is a member of the grid, in which case the result is in the range $0, \ldots P-1$.

`id_abs(const int id)`

A conversion function from id relative to this grid to id relative to the initial process group—ie, to absolute process id.

`lead_abs()`

The absolute id of the lead process of the grid—equivalent to `id_abs(0)`.

`operator/(const Coord& i)`

See section 2.6.

2.2.3 Examples

```
int shp [1] = {4} ;
Procs p(1, shp) ;
```

Creates a one-dimensional process array representing a linear set of 4 processes. The active process group must contain at least 4 processes when the declaration of `p` is encountered.

```
int shp [4] = {2, 2, 2, 2} ;
Procs h(4, shp) ;
```

Creates a process array representing a 4-dimensional binary hypercube. The active process group must contain at least 16 processes when the declaration of `h` is encountered.

```
  int shp [2] = {2, 3}, ids [6] = {0, 1, 3, 6, 10, 15} ;
  Procs q(2, shp, ids) ;
```

Creates a process array representing a 2 by 3 grid with a user-defined mapping to members of the active process group. The active process group must contain at least 16 processes when the declaration of `q` is encountered (because `ids` includes values up to 15).

```
  Procs s(0, 0) ;
```

Creates scalar process grid containing a single process.

2.3 class Dimension

A *process dimension* is a dimension of a process array. A process dimension is described by a collective object with local components of class `Dimension`. The public interface of the `Dimension` class is

```
class Coord ;

class Dimension {
public :
  Dimension() ;

  int size() const ;
  int crd() const ;

  operator int() ;

  int str_prc() const ;

  operator int() const ;

private :
  ...
} ;
```

2.3.1 Constructors and destructors

`Dimension()`

The default constructor. Initializes a `Dimension` object describing a *collapsed dimension*. This is a degenerate, "internal" dimension, not associated with a process array.

2.3.2 Methods

`size()`

An inquiry function returning the extent, N, of the dimension. For a dimension of a process array, this is the extent specified in the shape vector for that array. For a collapsed dimension, N defined to be 1.

`crd()`

An inquiry function returning an integer coordinate for the local process relative to this process dimension. For a process array dimension the result value is only defined if the local process is a member of the parent array. It is then in the range $0, \ldots, N-1$, where N is the extent of the dimension. For a collapsed dimension, the result is 0.

`str_prc()`

An inquiry function defining the stride in process id (relative to the parent array) associated with this dimension. For a collapsed dimension the function is ill-defined.

2.3.3 Operators

`operator int()`

Returns a non-zero value if the dimension belongs to a process array. Returns 0 if the dimension is collapsed. *[Replace this with a* `dims` *member? In any case, more consistent with* `Range` *to provide conversion to* `DimensionRep*`*]*

2.3.4 Examples

```
int shp [2] = {2, 3} ;
Procs p(2, shp) ;

Dimension d = p.dim(0), e = p.dim(1) ;
Dimension f ;
```

Create a process array `p` representing a 2 by 3 grid. The active process group must contain at least 6 processes when the declaration of `p` is encountered. Set `d` to represent the first dimension of `p` and `e` to represent the second. Set `f` to a collapsed dimension.

The inquiries `d.size()`, `e.size()` and `f.size()` return 2, 3 and 1 respectively. The inquiries `d.crd()` and `e.crd()` will return values in the ranges $\{0, 1\}$ and $\{0, 1, 2\}$ respectively *if* the inquiry `p.member()` returns value 1. They return unspecified values if `p.member()` returns 0. The inquiry `f.crd()` will return 0.

2.3.5 Implementation notes

`Dimension` is currently implemented as a simple, non-reference-counting handle class. If it is a process array dimension it contains a pointer to an object of type `DimensionRep` associated with the parent `Procs` structure. If it is collapsed dimension it contains a null pointer.

2.4 class `DimensionSet`

A *dimension set* is some subset of the dimensions of a particular process array. A dimension set is described by a collective object *[?]* with local components of class `DimensionSet`. The public interface of the `DimensionSet` class is

```
class Dimension ;

class DimensionSet {
public :
  DimensionSet() ;

  int member(Dimension d) const ;

  int contains(DimensionSet ds) const ;

  operator int() const ;

  DimensionSet operator+(DimensionSet ds) const ;
  DimensionSet operator-(DimensionSet ds) const ;

  DimensionSet& operator+=(DimensionSet ds) ;
  DimensionSet& operator-=(DimensionSet ds) ;

  DimensionSet operator+(Dimension d) const ;
  DimensionSet operator-(Dimension d) const ;

  DimensionSet& operator+=(Dimension d) ;
  DimensionSet& operator-=(Dimension d) ;

private :
  ...
} ;
```

2.4.1 Constructors and destructors

`DimensionSet()`

The default constructor. Initializes a `DimensionSet` object to an empty set.

2.4.2 Methods

`member(Dimension d)`

Inquiry function. Returns a non-zero result if `d` is a member of this set, and 0 if it is not.

`contains(DimensionSet ds)`

Returns a non-zero result if this set contains `ds` and 0 if it does not.

2.4.3 Operators

`operator int()`

Returns a non-zero value if the set is non-empty, or 0 if it is empty.

`operator+(DimensionSet ds)`

Create a new set containing the union of this set with `ds`. If both input sets are non-empty, they must contain dimensions from the same process array.

`operator-(DimensionSet ds)`

Create a new set by subtracting `ds` from this set. If both input sets are non-empty, they must contain dimensions from the same process array.

`operator+=(DimensionSet ds)`

Replace this set by its union with `ds`. If both input sets are non-empty, they must contain dimensions from the same process array.

`operator-=(DimensionSet ds)`

Subtract `ds` from this set. If both input sets are non-empty, they must contain dimensions from the same process array.

`operator+(Dimension d)`

Create a new set by adding a dimension to this set. If the set is non-empty before this operation `d` must be collapsed or a dimension from the same process array as the current elements. Adding a collapsed dimension, or one already in the set, does not change the set.

`operator-(Dimension d)`

Create a new set by removing `d` from this set. If the set is non-empty before this operation `d` must be collapsed or a dimension from the same process array as the current elements. Removing a collapsed dimension, or one not in the set, does not change the set.

```
operator+=(Dimension d)
```

Add a dimension to this set. If the set is non-empty before this operation `d` must be collapsed or a dimension from the same process array as the current elements. Adding a collapsed dimension, or one already in the set, does not change the set.

```
operator-=(Dimension d)
```

Remove a dimension from this set. If the set is non-empty before this operation `d` must be collapsed or a dimension from the same process array as the current elements. Removing a collapsed dimension, or one not in the set, does not change the set.

2.4.4 Examples

```
int shp [3] = {2, 2, 2} ;
Procs p(3, shp) ;

DimensionSet ds ;
ds += p.dim(0) ;
ds += p.dim(2) ;
```

Create a process array `p` representing a 3 dimensional binary hypercube. Create `ds` as an empty dimension set, then add the first and third dimension of `p` to it.

2.4.5 Implementation notes

`DimensionSet` is currently realized as a simple, one-word record containing a `long` integer used as a bitmask. All the members can be implemented as short inline functions using bitwise operations—this class is very lightweight.

This implementation imposes the Adlib kernel's only limitation on dimensionality—the number of process dimensions in a process array should normally be less than the number of bits in a `long` integer. Unless, for unknown reasons, an application involves a process array with many dimensions of unit extent, this constraint is not really stronger than the tacit assumption that all processes can be distinctly labelled by a single integer. On a 32-bit computer, for example, this implementation theoretically supports binary hypercubes of up to about 4 billion processors. . .

2.5 struct `Coord`

A record that bundles together a process dimension and a coordinate value in that dimension. The definition of the `Coord` struct is

```
class Dimension ;

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

2.6 class `Group`

A process group is described by a collective object *[?]* with local components of class `Group`. The public interface of the `Group` class is

```
class Procs ;
class DimensionSet ;
class Coord ;

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

  const Procs* prc() const ;

  DimensionSet dims() const ;

  int member() const ;

  int size() const ;

  int id() const ;

  int id_prc(const int id) const ;
  int id_abs(const int id) const ;

  int lead_prc() const ;
  int lead_abs() const ;

  void restrict(Dimension d, const int coord) ;
  void restrict(Dimension d) ;
  void restrict(DimensionSet ds) ;

  inline Group operator/(const Coord& i) const ;
  inline Group& operator/=(const Coord& i) ;

private :
  ...
} ;
```

2.6.1 Constructors and destructors

`Group()`

The default constructor. Creates an uninitialized `Group` object.

`Group(const Proc& p)`

Conversion constructor. Initializes a `Group` object representing all processes in process array `p`. By definition `p` is the *parent process array* of the constructed group. The new group object can subsequently be reduced to a subset of `p` by using the `restrict` members below.

2.6.2 Methods

`prc()`

An inquiry function returning a pointer to the local `Procs` object associated with the parent process array.

`dims()`

An inquiry function returning the effective dimension set of this group. This is a subset of the dimensions of the parent process array,

`member()`

An inquiry function that returns a non-zero value if the local process is a member of the group, and 0 if not.

`size()`

An inquiry function returning the total number of processes, P, in the group.

`id()`

An inquiry function that returns the id of the local process relative to this group. Its value is defined if and only if the local process is a member of the group, in which case it is in the range $0, \ldots P-1$.

`id_prc(const int id)`

A conversion function from id relative to this group to id relative to the parent process array.

`id_abs(const int id)`

A conversion function from id relative to this group to id relative to the initial process group—ie, to absolute process id. Equivalent to `prc()->id_abs(id_prc(id))` `lead_prc()`

The id relative to the parent process array of the lead process of this group—equivalent to `id_prc(0)`.

`lead_abs()`

The absolute id of the lead process of the group—equivalent to `id_abs(0)`.

`restrict(Dimension d, const int coord)`

Reduce the process group by restricting the coordinate in `d` to the value `coord`. The operation removes `d` from `dims()`. It is ill-defined if `d` is not a member of `dims()` beforehand. It is a null operation if `d` is collapsed.

`restrict(Dimension d)`

Reduce the process group by restricting the coordinate in `d` to the value associate with the local process, equivalent to `restrict(d, d.crd())`. The operation is ill-defined if the local process is not a member of the parent process array, or if `d` is not a member of `dims()` beforehand. It is a null operation if `d` is collapsed.

`restrict(DimensionSet ds)`

Reduce the process group by restricting the coordinate in all dimensions from `ds` to the values associate with the local process. Equivalent to applying `restrict(d)` for every member, `d`, of `ds`. The operation is ill-defined if the local process is not a member of the parent process array, or if `ds` is not a subset of `dims()` beforehand.

2.6.3 Operators

`operator/(const Coord& i)`

Create a new group formed by restricting this group by `i`. The inline definition is

```
Group operator/(const Coord& i) const {
  Group result(*this) ;
  result.restrict(i.dim, i.crd) ;
  return result ;
}
```

`operator/=(const Coord& i)`

Restrict this group by `i`. The inline definition is

```
    inline Group& operator/=(const Coord& i) {
      restrict(i.dim, i.crd) ;
      return *this ;
    }
```

2.6.4 Related functions

`Procs :: operator/(const Coord& i)`

Create a new group formed by restricting this process array by `i`. The inline definition is

```
    Group Procs :: operator/(const Coord& i) const {
      Group result(*this) ;
      result.restrict(i.dim, i.crd) ;
      return result ;
    }
```

2.6.5 Examples

```
  int shp [2] = {2, 2} ;
  Procs p(2, shp) ;

  Coord c(0, p.dim(0)), d(1, p.dim(1)) ;

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

These groups are illustrated in figure 2.1. The dimension set of `q` contains just `p.dim(1)`, and dimension set of `r` is empty.

2.6.6 Implementation notes

`Group` can be realized as a simple three-word record containing a pointer to the parent `Procs` object, a `DimensionSet` (one word) representing the set of effective dimensions, and an integer specifying the id relative to the parent process array of the lead process of the group. These are the values returned by `prc`, `dims` and `lead_prc` respectively.

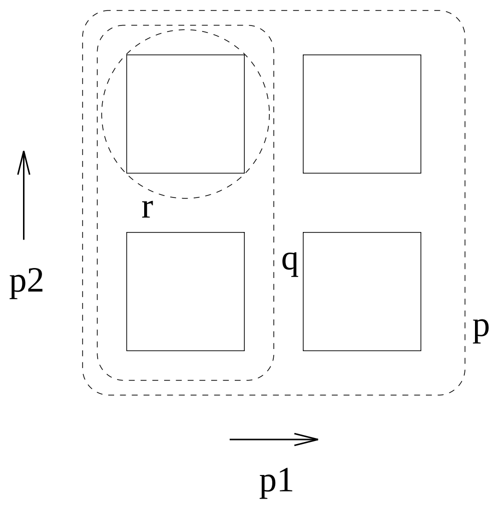

Figure 2.1: Examples of process groups. The square boxes represent the 4 processes in the process array `p`. The dashed lines embrace groups `p`, `q` and `r`.

2.7 Group apg

The static variable `apg` is accessed by most of the collective operations in the library. Its value should reflect the currently effective *active progress group*. Its declaration is

```
  extern Group apg ;
```

When an Adlib program starts the value of `apg` represents the initial process group. If collective operations are to function properly there is an onus on the programmer to maintain the value of `apg` consistently. In this section three common idiomatic ways of modifying the active process group are discussed.

The first idiom involves a conditional construct which restricts the set of processes performing a block of code to some group, `p`. Membership of the local process in this group is determined by calling `p.member()`, as follows:

```
if(p.member()) {
  ...
}
```

If it is necessary to invoke collective library operations *inside* the construct, `apg` must be reset appropriately. Assuming the construct appears at the point where `p` is a subset of the active process group, the logical active process group inside the construct will be the whole of `p`. Appropriate manipulations of `apg` are

```
Group apgSave = apg ;
if(p.member()) {
  apg = p ;
  ...
}
apg = apgSave ;
```

The only subtlety is the need to save the old value of `apg` so that it can be restored on completion of the construct. With these manipulations of the `apg` variable we can safely invoke collective operations before, during and after the conditional construct. This idiom will sometimes be called an *on* construct. In the ad++ interface to Adlib macros `ON`/`NO` are defined to allow the whole of the fragment above to be written as

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

A second common idiom for modifiying `apg` occurs when the logical active process group is partitioned by breaking up one of its process dimensions. Suppose the current active process group is a multi-dimensional array of processes. One of its process dimensions is d. Suppose we need to perform operations *collectively* across dimensions *orthogonal* to d, but *independently* for each value of the d coordinate. In this case the value of apg should be temporarily changed as follows

```
Group apgSave = apg ;
apg.restrict(d) ;
...
apg = apgSave ;
```

In the ad++ interface, for example, this kind of manipulation of apg occurs in the AT/TA construct and OVERALL/ALLOVER distributed loop.

The preceding idiom allows for regular partitioning of the active process group across one of its dimensions. A different technique can be used to achieve arbitrary partitioning. Suppose each of $A_1, \ldots, A_n$ is a set of processes, and together they partition the current active process group, A. For each i in $1, \ldots, n$, a_i is a vector of ids (relative to A) for the processes in A_i, and r_i and s_i define a rank and a shape for a grid of size $|A_i|$. Now, if the local process is a member of A_l, the following constructor call

```
Procs p(r_l, s_l, a_l)
```

creates n logically *distinct* collective objects representing non-overlapping process arrays. An *on* construct using p will now partition the active process group as required. For example, suppose the current active process group has 5 processes. We can partition these temporarily into groups of 2 and 3 as follows

```
int shp [1], *ids ;
if(apg.id() < 2) {
  shp [0] = 2 ;
  ids = {0, 1} ;
}
else {
  shp [0] = 3 ;
  ids = {2, 3, 4} ;
}

Procs p(1, shp, ids) ;

Group apgSave = apg ;
if(p.member()) {
  apg = p ;
  ...
}
apg = apgSave ;
```

This use of the `Procs` constructor departs slightly from usual rules for collective operations, because when the constructor is called the value of `apg` is the original active process group, but its arguments take different values in sectors of that group which will later be in separate partitions.

Notice that the idioms we have introduced in this section can be nested freely. As usual, the only subtlety is in ensuring that old values of the active process group are restored properly when the constructs complete. This can be achieved either by using a suitable series of automatic variables (like `apgSave`), or by using an explicit stack of group objects.

Chapter 3

Ranges

An Adlib *range* is a map from an integer interval $0, \ldots, N-1$ into a process dimension. Each value, or *global subscript*, in the interval is mapped to a particular process coordinate. While the library does *not* support completely general, irregular mappings from global subscript to process coordinate, it does provide many of the most popular options (*block*, *cyclic*, etc). In particular the allowed mappings support all distribution formats and intra-dimensional alignments of HPF 1.0. Adlib mappings generalize the single-level level distribution strategy of HPF to a *multi-level* strategy. In HPF a template can only be distributed over a processor arrangement. In Adlib, within certain limits, higher level templates can be distributed over sections of lower-level templates.

3.1 General Definitions

Level: A non-negative integer, characteristic of any range.

Primitive range: The range of a process dimension. The global subscripts are simply the coordinates of the process dimension.

A primitive range has level 0.

Subrange: A range defined as a (strided) subinterval of some parent range.

A subrange retains an *alignment* relation to the parent range—the mapping of a subrange element to the process dimension is the same as that of the corresponding element of the parent range. Subranges implement the "affine alignment" of HPF *[check this is proper use of the terminology]*.

The level of a subrange is the same as the level of the parent range.

Template range: A range that is not a subrange of any range except itself.

Template ranges can be used to represent the dimensions of an HPF template. In Adlib a primitive range (representing a process dimension) is also regarded as a template range.

Template ranges include primitive ranges, block-distributed ranges with a uniform block size and (in HPF terms) identity alignment, and cyclic-distributed ranges with identity alignment.

Parent template range: Any range is a subrange of its unique *parent template.*

Kernel range: Any range of level higher than 0 is distributed over some *kernel range.*

The level of a range's kernel one less than range's own level.

Simple range: A range whose kernel is a primitive range (ie, a process dimension). A simple range has level 1.

In HPF, block distributed array dimensions and simple-cyclic distributed array dimensions correspond to simple ranges. *Block-cyclic* distribution, in contrast, cannot be represented by a simple Adlib range. There is no primitive block-cyclic distribution format in Adlib, but block-cyclic distributions can be handled efficiently by using a level 2 range distributed blockwise over a cyclically distributed simple range[1]

Note that not all level 1 Adlib ranges are simple. A level 1 range could be distributed over a *subrange* of a process dimension. This does not correspond to any HPF 1.0 distribution format, and we will not regard it as a "simple".

Also note that simple ranges *do not* have any special status in Adlib. They are defined here only because they are a common case and will recur in examples.

[1]As another example of a non-simple range, an HPF compiler might allow virtual processor arrangements larger than the set of physical processes. This situation can be handled by distributing some level 1 ranges representing *virtual process dimensions* over the physical process dimensions of Adlib, then implementing HPF template dimensions as level 2 ranges distributed over these virtual process dimensions. (In the current implementation of Adlib the distribution format of the level 2 templates would have to be restricted. Currently ranges can only be distributed *cyclically* over level 0 ranges, not over level 1 or higher ranges. Also, level 2 is the highest level supported, so *block-cyclic* distribution over the virtual process dimensions would be disallowed on *two* counts. Future versions of Adlib may relax these restrictions.)

Global subscript: The primary subscript associated with a range. A value from the interval $0, \ldots,$`size()`-1. Sometimes just called the *subscript*.

Template subscript: For any element of a range, the associated global subscript in the parent template range.

Kernel subscript: For any element of a range, the associated global subscript in the kernel range.

Block: The section of a range associated with a particular fixed value of the kernel subscript. Any element of the range lies in a particular block. Note that elements of a block need not have contiguous global subscripts—in cyclic distribution format, adjacent elements of the block have template subscript differing by the extent of the kernel.

Block subscript: For any element of a range, a subscript identifying the position of the element within its own block. In general there is a one-two one mapping between between legal kernel-subscript/block-subscript pairs and legal template subscripts.

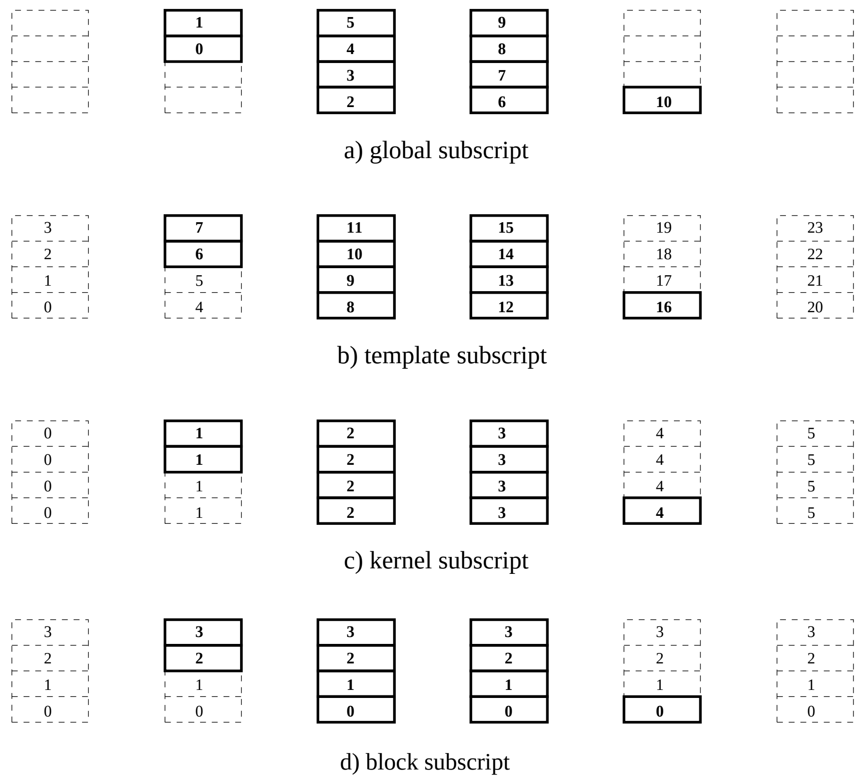

Figure 3.1: Illustration of various subscripts for a block-distributed subrange.

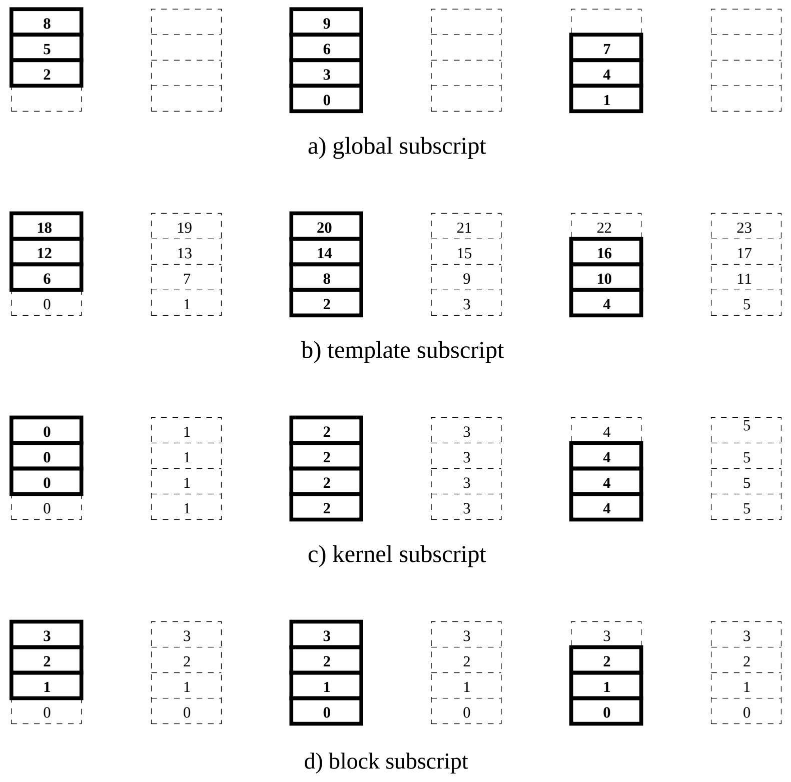

Figure 3.2: Illustration of various subscripts for a cyclic-distributed subrange with stride 2.

3.2 class Range

A *range* is a mapping from an integer interval to a process dimension. A range is described by a collective object with local components of class Range. The public interface of the Range class is

```
class Stride ;
class Location ;
class Block ;
class RangeRep ;
enum Format {DIST_PRIMITIVE, DIST_COLLAPSED,
             DIST_BLOCK, DIST_CYCLIC, ...} ;

class Range {
public :
  Range() ;
  Range(const Range& x) ;
  Range(const int extent) ;
  Range(Dimension dim) ;
  ~Range() ;

  Range subrng(const int extent,
               const int base = 0, const int stride = 1) ;

  int size() const ;

  Dimension dim() const ;

  int lev() const ;
  Range ker() const ;

  Range tem() const ;
  int bas() const ;
  int str() const ;

  Format format() const ;

  int ker_glb(const int glb, int* sub) const ;

  int block(const int ker_glb, int* sub, int* count) const ;
  int block_step(int* sub_stp) const ;

  int crd(const int glb) const ;
  int local(int* glb) const ;

  Range subker(int* ker_bas, int* ker_str) const ;
```

```
    int width() const ;

    int volume() const ;
    Stride stride(int &size) const ;

    Location operator()(const int k) ;

    Range& operator=(const Range& x) ;

    operator RangeRep*() const ;
  private :
    ...
  } ;
```

The constructors of the `Range` class itself only provide primitive and collapsed distribution formats. An (in principle extensible) series of derived classes provides more general options.

3.2.1 Constructors and destructors

`Range()`

The default constructor. Creates a null (uninitialized) range object. The only members that can legally be applied to such an object are the assignment `operator=(const Range& x)` and the conversion `operator RangeRep*()` and the copy constructor.

`Range(const Range& x)`

Copy constructor. `Range` is implemented as a reference-counted handle class to an object of type `RangeRep`. The copy constructor copies the reference and increments the reference count in the representation object.

`Range(Dimension dim)`

Conversion constructor. Constructor for a level 0 template range (a primitive range) describing `dim`. The size of this range is `dim.size()`. The `dim()` inquiry applied to the constructed range returns the constructor argument. The `ker()` inquiry returns a null range.

`Range(const int extent)`

Conversion constructor. Constructor for a *collapsed* template range. This is a range of size `extent` mapped entirely to the local process. The `dim()`

inquiry applied to a collapsed range returns a collapsed dimension. The constructed range is not strictly primitive—its level is 1. The `ker()` inquiry will return a primitive range representing a collapsed dimension[2].

`~Range()`

Destructor. If the range is non-null, the representation object's reference count is decremented. If this reduces the count to 0, the representation object is deleted.

3.2.2 Methods

`subrng(const int extent, const int base, const int stride)`

Returns a `Range` object representing a subrange of this range. The size of the subrange is `extent` and its elements are labelled 0, . . . ,`extent`−1. The subrange includes elements

```
  base, base + stride, base + 2 * stride, ...,
                       base + (extent - 1) * stride
```

from the parent. The elements of the subrange are mapped to the underlying process dimension in the same way as the corresponding elements of the parent range (they are *aligned* with the corresponding elements of the parent). The value of `stride` must be positive or negative (not zero). The values of `base` and `base + (extent - 1) * stride` must be in the range 0, . . . ,`size()`−1. If the `stride` actual argument is omitted, it defaults to 1. If the `base` actual argument is also omitted, it defaults to 0.

If this range (the parent) is itself a subrange, the alignment parameters `base` and `stride` for the new subrange are composed with those of the parent, to give a simple map to the common template range.

`size()`

An inquiry function returning the extent of the range.

`dim()`

An inquiry function returning the underlying process dimension. For a collapsed range this is a collapsed dimension.

`lev()`

[2]This is different to a null range.

An inquiry function returning the level of this range. Zero for (a range describing) a process dimension or subrange of a process dimension, positive for any other range.

`ker()`

An inquiry function returning the kernel range. In general this is a range of level one lower than this range. For a level 0 range it is a null range.

`tem()`

An inquiry function returning the parent template range.

`bas()`

An inquiry function returning the alignment base of this range in the parent template range. Element 0 of this range corresponds to element `bas()` in the template range.

`str()`

An inquiry function returning the alignment stride of this range relative to the parent template range. Adjacent elements of this range correspond to elements separated by `str()` in the template range. The result may be positive or negative (not zero).

`format()`

An inquiry function returning a code defining the distribution format of this range. *[Give enumeration type of result values.]*

`ker_glb(const int glb, int* sub)`

Compute the kernel subscript and the block subscript associated with a specified global subscript. The kernel subscript is returned as the result of the function. The block subscript overwrites `*sub`.

For a *simple range* the return value is the process coordinate to which element `glb` of the range is mapped.

`block(const int ker_glb, int* sub, int* count)`

This function is defined only for ranges *with level higher than 0.* Given a kernel subscript value, `ker_glb`, return base and size of the associated block of this range. Used in translation of parallel loops—see section 7 for examples. This function can be regarded as an inverse of the `ker_glb()` member described above.

The return value is the smallest value of the global subscript contained in the block. The corresponding value of the block subscript overwrites `*sub`. The number of active elements in the block overwrites `*count`.

`block_step(int* sub_stp)`

This function is defined only for ranges *with level higher than 0*. Returns the difference in the values of global and block subscripts between adjacent active elements in a single block. Used in translation of parallel loops—see section 7 for examples.

The return value is the global subscript step. The corresponding step in block subscript overwrites `*sub_stp`.

`crd(const int glb)`

This function is defined *only for level 0 ranges*. Given a global subscript value, it returns the coordinate value. If the range is primitive, this is the identity function. For a subrange of a primitive range it returns the value `bas() + str() * glb`.

`local(int* glb)`

This function is defined *only for level 0 ranges*, and then only if the local process is a member of the process array to which `dim()` belongs. It returns the value 1 if the local process holds an element of the range and zero otherwise. If the range is primitive—a complete process dimension, the result is always non-zero. For a subrange it may be zero. Used in translation of parallel loops—see section 7 for examples.

If the result is non-zero, the value of the global subscript of the local process with respect to the range overwrites `*glb`. If the range is primitive, the global subscript is value of the local coordinate. For a subrange of a primitive range it returns the global subscript `(crd - bas()) / str()` where `crd` is the local coordinate.

`subker(int* ker_bas, int* ker_str)`

A subrange of the kernel, including all elements of the kernel for which the function `block` defines a non-empty block. Used to optimize enumeration of the blocks of a range—see section 7.

The alignment parameters of the subkernel *relative to the kernel* (which is not necessarily a template range) overwrite `*ker_bas` and `*ker_str`.

`width()`

A bound on the number of elements for any block of the range. Ideally this is the largest `count` value returned by `block` for any allowed value of `ker_glb`. Some forms of range, especially subranges, will be less economical in the definition of `width`, returning an overestimate. This member is used directly or indirectly to control the allocation of memory for elements of distributed arrays.

`volume()`

A bound on the number of elements of the range associated with any single process. Equivalent to

```
  inline int Range :: volume() {
    int size = 1 ;
    for(Range x = *this ; x.lev() ; x = x.ker())
      size *= x.width() ;
    return size ;
  }
```

ie, it is the product of the `width()` values from this and any kernel levels.

`stride(int &size)`

Returns a `Stride` object associated with this range—see section 4.2.

3.2.3 Operators

`operator()(const int glb)`

Returns a `Location` object associated with global subscript value `glb`—see section 4.1.

`operator=(const Range& x)`

Assignment operator. Copies the reference in handle `x` and increments the reference count in the representation object. Decrements the reference count of (and deletes, if necessary) any representation object referenced by the assignment variable prior to the assignment.

`operator RangeRep*()`

Returns a pointer to the representation object. Returns a null pointer if the range is null.

This low-level operator is useful (for example for testing for a null range), but note that it returns a handle which is not recorded in the reference count of the representation object. The specific value of the result is therefore "ephemeral"—a reference that can easily be rendered invalid by unexpected deletion of the representation object.

3.3 class `BlockRange`

The class `BlockRange` is a subclass of `Range` describing *uniform block-distributed* ranges. The `format` inquiry returns `DIST_BLOCK` for a `BlockRange` or any subrange.

The public interface of the `BlockRange` class is

```
class BlockRange : public Range {
public :
  BlockRange(Range ker, const int blockSize,
             const Layout layout = LAY_STANDARD) ;

  BlockRange(const int extent, Range ker,
             const Layout layout = LAY_STANDARD) ;

  BlockRange(Range ker, const int blockSize,
             const Layout layout, const int wlo, const int whi) ;

  BlockRange(const int extent, Range ker,
             const Layout layout, const int wlo, const int whi) ;
}
```

3.3.1 Constructors

`BlockRange(Range ker, const int blockSize, const Layout layout)`

Creates a block-distributed template range with kernel `ker` and uniform block size `blockSize`. The extent of the constructed range is

```
blockSize * ker.size()
```

The `Layout` argument controls the memory layout of arrays constructed with this range or its subranges. It does this by controlling the values returned by the `width` and `stride` members. The options for `layout` are currently `LAY_STANDARD` or `LAY_PACKED`. The only difference is for arrays created with strided subranges (ie, with strided alignments). With *standard* layout, array elements are allocated for every element of the template; with *packed* layout the `disp` member of the associated `Stride` object divides the block subscript by the alignment stride, reducing memory requirements by this factor.

`BlockRange(const int extent, Range ker, const Layout layout)`

Creates a uniform block-distributed range of extent `extent` and kernel `ker`. The block size is

```
  (extent + ker.size() - 1) / ker.size()
```

If this exactly divides `extent`, the constructed range is a template range. Otherwise it is a subrange of a template range with this block size (and `bas()`= 0 and `str()`= 1). The options for `layout` are the same as the previous constructor. *[Having two constructors with such similar argument lists may be error prone.]*

```
BlockRange(Range ker, const int blockSize,
           const Layout layout, const int wlo, const int whi)
```

Similar to the first constructor above, but the value of `Layout` must be `LAY_EXTENDED`. Arrays constructed with this range or its subranges have ghost regions of width `wlo`, `whi` at the upper and lower edges of each block of the range.

```
BlockRange(const int extent, Range ker,
           const Layout layout, const int wlo, const int whi) ;
```

Similar to the second constructor above, but the value of `Layout` must be `LAY_EXTENDED`. Arrays constructed with this range or its subranges have ghost regions of width `wlo`, `whi` at the upper and lower edges of each block of the range.

3.4 class CyclicRange

The class `CyclicRange` is a subclass of `Range` describing *cyclically-distributed* ranges. The `format` inquiry returns `DIST_CYCLIC` for a `CyclicRange` or any subrange.

The public interface of the `CyclicRange` class is

```
class CyclicRange : public Range {
public :
  CyclicRange(Range ker, const int numCycles,
              const Layout layout = LAY_STANDARD) ;

  CyclicRange(const int extent, Range ker,
              const Layout layout = LAY_STANDARD) ;
}
```

3.4.1 Constructors

`CyclicRange(Range ker, const int numCycles, const Layout layout)`

Creates a cyclically-distributed template range with kernel `ker`, wrapping round the kernel `numCycles` times. The kernel should be a level-0 range[3]. The extent of the constructed range is

```
numCycles * ker.size()
```

The `Layout` argument controls the memory layout of arrays constructed with this range or its subranges. It does this by controlling the values returned by the `width` and `stride` members. The options for `layout` are currently `LAY_STANDARD` or `LAY_ADAPT`. The only difference is for arrays created with subranges (ie, with non-trivial alignments). With *standard* layout, array elements are allocated for every element of the template; with `ADAPT` layout the `width` member of the associated `Stride` cuts down the number of allocated elements, not allotting space for elements with block subscripts *higher* than any actually inside the subrange.

`CyclicRange(const int extent, Range ker, const Layout layout)`

Creates a cyclically-distributed range of extent `extent` and kernel `ker`. The number of cycles is

```
(extent + ker.size() - 1) / ker.size()
```

[3]If this restriction is ignored, certain communication functions, notably `remap`, will not work properly.

If this exactly divides `extent`, the constructed range is a template range. Otherwise it is a subrange of a template range with this number of cycles (and `bas()`= 0 and `str()`= 1). The options for `layout` are the same as the previous constructor.

Chapter 4

Arrays

An Adlib array is a rectangular distributed array of any rank, whose elements are partioned or replicated across some set of processes. The layout of the array is described by an object of type `DAD` which contains

- The *rank*, `R` of the array (ie, its number of dimensions). `R` is greater than or equal to zero.
- The *process group* over which the elements of the array are distributed.
- A vector of `R` *range objects*, describing the shape of the array, and the mapping of its index space into the dimensions of of the process group.
- A vector of `R` *stride objects*, defining how elements with locally-resident subscript tuples are arranged in the locally-held segment of the array.

The definition of the array is completed by a local array, or a pointer to a base address in the memory of each process, where the elements are actually stored. The type of the array elements is determined by the type of the local array segment—the `DAD` is blind to element type.

Note that the Adlib kernel makes no intrinsic assumptions about the ordering of array elements in local memory. For example, when a new distributed array is created, the stride vectors can set up to describe the situation where the first or the last dimension is "most rapidly" varying in memory. Similarly, no assumption is made that array elements fill a contiguous region of memory. In Fortran 90 terms, a `DAD` record can describe an arbitrary *regular section* of some parent array.

4.1 struct `Location`

A record that bundles together a process dimension, a coordinate, and any block subscripts implied by a particular global subscript value in a particular range.

This class is used to hold intermediate results in address computations. It is used frequently in Adlib code that has to deal with ranges whose level is not known in advance (its use can sometimes be avoided if the level of ranges is known at compile time). The constant `MAX_LEVELS` is a library-wide limit on the highest level of range supported.

The interface of the `Location` struct is

```
struct Location : public Coord {
  Location() ;
  Location(const Range x, const int glb) ;

  int subs [MAX_LEVELS] ;
} ;
```

4.1.1 Public data fields

`subs`

Stack of block subscripts for the range and any kernels of level higher than 0. Assuming the range has level greater than 0, the subscript for the top-level range is held in `subs [0]`. If its kernel has level higher than 0, the kernel's subscript is held in `subs [1]` (and so on).

Dropping through kernels brings us to a level 0 range. The template subscript (ie, the process coordinate) in this inner kernel is stored in the `crd` field of the base class. The dimension of the range is stored in the `dim` field.

4.1.2 Constructors

`Location()`

The default constructor. Creates an uninitialized `Location` object.

`Location(const Range x, const int glb)`

Create a `Location` object containing the block subscripts and process coordinate associated with global subscript `glb` in range `x`. Equivalent to

```
  Location(Range x, int glb) {
    int* sub = subs ;
    for(Range y = x ; y.lev() ; y = y.ker())
      glb = y.ker_glb(glb, sub++) ;

    dim = y.dim() ;
    crd = y.crd(glb) ;
  }
```

In this implementation `glb` is reduced by successive applications of `Range :: ker_glb`, storing the template subscripts for the higher levels in the `subs` vector, then initializing fields of the `Coord` base class (see section 2.5) using the `dim` and `crd` members of the innermost kernel range.

4.1.3 Related functions

`Range :: operator()(const int glb)`

Returns a `Location` object associated with global subscript value `glb`. Inline definition is

```
  inline Location Range :: operator()(const int glb) {
    return Location(*this, glb) ;
  }
```

4.2 class `Stride`

A *stride* is part of a distributed array descriptor. It is a map from a `Location` (or equivalent stack of block subscripts) to an offset in a local data segment. Strides are associated with particular parent ranges, because in general local memory layout depends on the distribution format of the range to which the subscript relates. In contexts where the unqualified term "stride" may lead to confusion with the *alignment stride* of a range, the phrase "array stride" will be used instead.

A stride is described by a collective object with local components of class `Stride`. The public interface of the `Stride` class is

```
class Location ;
class Block ;
class StrideRep ;

class Stride {
  Stride() ;
  Stride(const Stride& x) ;

  ~Stride() ;

  Stride ker() const ;

  int len() const ;

  int disp(const int sub) const ;
  int disp_step(const int sub_stp) const ;

  int offset(Location i) const ;
  int step(Block b) const ;

  Stride& operator=(const Stride& u) ;

  operator StrideRep*() ;
private :
  ...
} ;
```

4.2.1 Constructors and destructors

`Stride()`

The default constructor. Creates a null (uninitialized) stride object. The members `offset` and `step` return 0 for null strides. Null strides are occasionally useful. They can be used to create the illusion of replicating values in array dimensions, without actually copying data.

`Stride(const Stride& u)`

Copy constructor. `Stride` is implemented as a reference-counted handle class to an object of type `StrideRep`. The copy constructor copies the reference and increments the reference count in the representation object.

`~Stride()`

Destructor. If the stride is non-null, the representation object's reference count is decremented. If this reduces the count to 0, the representation object is deleted.

4.2.2 Methods

`ker()`

An inquiry function returning the kernel stride. This is a stride associated with the kernel of the parent range. If the corresponding kernel range is level 0, the stride returned is null.

`len()`

An inquiry function returning the the original item length used in creation of this stride.

`disp(const int sub)`

Translation from block subscript, `sub`, to an offset (displacement) in a local array segment. The form of this function depends on the detailed structure of the parent range.

Note that for higher-level parent ranges this member does *not* take account of any contribution to the local offset arising from kernel ranges—to compute that contribution the `disp` members of the kernel ranges must be called explicitly, adding together the contributions from different levels. See `offset`, below.

If `x` is the parent range, for any legal value of `sub` the inequality

```
  disp(sub) < len() * x.width()
```

applies.

If this is a null stride, the result of `disp` is always zero.

`disp_step(const int sub_stp)`

The difference in the value of `disp`—the local segment offset—between two points in the same block of the range separated in block subscript by `sub_stp`.

If this is a null range, the result is zero.

`offset(Location i)`

Returns the total offset for `Location` record `i`. The result is defined by adding together the results of the `disp` members for this range and all kernels. If this is a null stride, the result is zero. Equivalent to

```
int offset(const Location& i) const {
  int off = 0 ;
  const int* sub = i.subs ;
  for(Stride u = *this ; u ; u = u.ker())
    off += u.disp(*(sub++)) ;
  return off ;
}
```

`step(Block b)`

The difference in the value of `offset`—the total segment offset—between two points in the block defined by `b`. Inline definition is

```
inline int Stride :: step(const Block& b) const {
  return disp_step(b.sub_stp) ;
}
```

4.2.3 Operators

`operator=(const Stride& u)`

Assignment operator. Copies the reference in handle `u` and increments the reference count in the representation object. Decrements the reference count of (and deletes, if necessary) any representation object referenced by the assignment variable prior to the assignment.

`operator StrideRep*()`

Returns a pointer to the representation object. Returns a null pointer if the stride is null.

This low-level operator is useful (for example for testing for a null stride), but note that it returns a handle which is not recorded in the reference count of the representation object. The specific value of the result is therefore "ephemeral"—a reference that can easily be rendered invalid by unexpected deletion of the representation object.

4.2.4 Related functions

`Range :: stride(int& size)`

Returns a `Stride` object associated with this range. On entry `size` is the size of an individual item of the array[1]. On exit `size` is multiplied by the value returned by the `volume()` inquiry.

[1]For a multi-dimensional array, an "individual item" might be a whole array slice containing the nested, more-rapidly-varying dimensions.

4.3 struct DAD

A *distributed array descriptor* describes the layout of the elements of a distributed array. An array descriptor is represented by a collective object with local components of class DAD. The public interface of the DAD class is

```
class Group ;
class Range ;
class Stride ;
class DimensionSet ;

struct DAD {
  DAD(const int _rank, const Group& _group,
      Range* _ranges, Stride* _strides) ;

  DAD(const int _rank, const Group& _group) ;

  int rnk() const ;

  const Group &grp() const ;

  const Range  rng(const int r) const ;
  const Stride str(const int r) const ;

  DimensionSet sig() const ;

  const int rank ;

  Group group ;

  Range*  ranges  ;
  Stride* strides ;
} ;
```

4.3.1 Public data fields

`rank`

The rank, R, of the array. Greater than or equal to zero.

`group`

The group over which the array is distributed.

`ranges`

The vector of ranges of the array. Size of the vector should be `R`. Distributed ranges in this vector should be distributed over distinct (orthogonal) dimensions of `group`.

`strides`

The vector of strides of the array. Size of the vector should be `R`. There is a one-to-one correspondence between elements of this vector and the `ranges` vector. *[Need to define the exact nature of that "correspondence".]*

4.3.2 Constructors

```
DAD(const int _rank, const Group& _group, Range* _ranges,
    Stride* _strides)
```

The normal constructor. Initializes all fields of the `DAD` object, simply copying the values passed as arguments. The constructor *does not* allocate any new vectors internally. The `_ranges` and `_strides` pointers are simply copied. It is the caller's responsibility to ensure that associated vectors persist as long as the constructed `DAD` record.

`DAD(const int _rank, const Group& _group)`

Simplified constructor. The `ranges` and `strides` fields are left undefined. Since they are publically accessibly, they can be initialized later. *[Ought to be a default constructor, too].*

4.3.3 Methods

`rnk()`

An inquiry function returning the rank of the array (the value of `rank`).

`grp()`

An inquiry function returning the group over which the array is distributed (the value of `group`).

`rng(const int r)`

An inquiry function returning the `r`th range of the array—the value of `ranges [r]`.

`str(const int r)`

An inquiry function returning the `r`th stride of the array—the value of `strides [r]`.

`DimensionSet sig()`

Return the *signature* of the array. The signature is the set of process dimensions over which the ranges of the array are distributed. Equivalent to

```
DimensionSet sig() const {
  DimensionSet signature ;
  for(int r = 0 ; r < rank ; r++)
    signature += ranges [r].dim() ;
  return signature ;
}
```

One common use for the `sig` member is to compute the set of process dimensions over which the array is replicated. This is given by

```
grp().dims() - sig()
```

4.3.4 Examples

Suppose `p` is a group and `x` and `y` are ranges distributed over dimensions of this group. We want to create a new distributed array of floating point numbers described by these parameters. A typical procedure would be

```
Range  [2] a_ranges ;
Stride [2] a_strides ;

int size = 1 ;

a_ranges [0] = x ;
a_ranges [1] = y ;

a_strides [0] = x.stride(size) ;
a_strides [1] = y.stride(size) ;

DAD a_frm(2, p, a_ranges, a_strides) ;
float* a_dat = new float [size] ;
```

`x` and `y` are copied into the vector `ranges`. New strides are created in the vector `strides`. This code implements an array with first dimension "most-rapidly-varying" in memory[2]. After the two calls to `Range :: stride` the

[2]To implement an array with last dimension most-rapidly varying one would simply reverse the order of the calls to `stride`

value accumulated in `size` is the total volume of the local array segment (see section 4.2). The group and vectors are installed in `a_frm` and the local data segment is allocated in `a_dat`.

Now suppose `i` and `j` are `Location` objects with parent ranges `x` and `y` respectively. We want to access the array element associated with these subscripts. A typical producedure would be

```
Location i(x, l1), j(y, l2) ;
...
if(i.dim.crd() == i.crd)
  if(j.dim.crd() == j.crd) {
     ...
     a_dat [a_frm.str(0).offset(i) + a_frm.str(1).offset(j)]
     ...
  }
```

First we test if the element is held on the local process by looking at the coordinate field in the subscript objects[3], then the `offset` member is applied to the stride objects to find the offsets in the local data segment.

Next, suppose we want to set up a description of a two-dimensional section of `a`, with subranges of `x` and `y` (something like `a(l1:u1, l2:u2)` in Fortran).

```
Range  [2] b_ranges ;
Stride [2] b_strides ;

b_ranges [0] = x.subrng(n1, l1) ;
b_ranges [1] = y.subrng(n2, l2) ;

b_strides [0] = a.str(0) ;
b_strides [1] = a.str(1) ;

DAD b_frm(2, p, b_ranges, b_strides) ;
float* b_dat = a_dat ;
```

In this case no new stride objects should be created, because we are simply building a new description of a subset of the existing array elements. Instead the strides vector is copied from the original array descriptor. Similarly, the local data segment is shared with the original array. Element-by-element copying of the `strides` vector is actually unnecessary here: one could have written

```
DAD b_frm(2, p, b_ranges, a.strides) ;
```

[3]An outer test may be needed, to see if the local process is a member of `p` (ie, of `a_frm.grp()`)

If, instead, we wanted to set up a description of a one dimensional section of a, with subscript i in the first dimension, retaining the whole of the second dimension (something like a(i, :) in Fortran) we could write

```
Range  [1] c_ranges ;
Stride [1] c_strides ;

c_ranges [0] = a.rng(1) ;

c_strides [0] = a.str(1) ;

DAD c_frm(1, p / i, c_ranges, c_strides) ;
float* c_dat = a_dat + a_frm.str(0).offset(i) ;
```

Again no new stride objects should be created: the stride of the unsubscripted dimension is copied from the original array descriptor. The process group over which the section is distributed is p / i. The offset for the first dimension is added to the base address of the original array[4]. Subsequently we may access element j of the section b by

```
if(b_frm.grp().member())
  if(j.dim.crd() == j.crd) {
     ...
     b_dat [b_frm.str(0).offset(j)]
     ...
  }
```

Finally we illustrate the kind of optimizations that are possible in access to array elements if the ranges x and y are known to be *simple*. The earlier example can then be changed to

```
int sub1, sub2 ;
...
if(x.dim().crd() == x.ker_glb(l1, &sub1))
  if(y.dim().crd() == y.ker_glb(l2, &sub2))
     ...
     a_dat [a_frm.str(0).disp(sub1) + a_frm.str(1).disp(sub2)]
     ...
  }
```

We can avoid creating the Location records by calling ker_glb directly, and use the simpler and more efficient disp member of Stride to to compute the offset.

[4]The offset pointer value is only valid on processes that hold part of the section, but it does no harm to do the addition on other processes. The value b_dat should never be *used* on processes where it is not valid—ie on processes outside the group p / i.

4.4 Array shape

As in Fortran, the *shape* of an array, a, is defined as the vector of extents of its ranges, ie (`a.rng(0).size()`, ..., `a.rng(R-1).size()`), where R is the rank of the array.

4.5 Alignment and value-alignment

An array, `a` is *aligned with* an array `b` if they are distributed over the equivalent process groups and their ranges are all equivalent:

$$\begin{aligned} \texttt{a.grp()} &\approx \texttt{b.grp()} \\ \texttt{a.rng(0)} &\approx \texttt{b.rng(0)} \\ &\vdots \\ \texttt{a.rng(R-1)} &\approx \texttt{b.rng(R-1)} \end{aligned}$$

We omit full formal definition of alignment equivalence between groups and ranges. Informally, two groups or two ranges are equivalent if they are *structurally equivalent*[5]. The informal meaning of alignment is that corresponding elements of the two arrays are stored on the same process, or replicated over the same group of processes.

An array, `a` is *value-aligned* with an array `b` if their ranges are all equivalent and `a.grp()` *contains* `b.grp()`. The informal meaning of value-alignment is that every process that holds a copy of an element of `b` holds a copy of the corresponding element of `a` (although the converse may not be true, because `a` can be replicated over a larger process group).

Note that for value-alignment it *is* required that ranges are equivalent. A common mistake is to assume if a particular range of `b` is collapsed, but `a` has replicated alignment with respect to the corresponding range of `b`, the arrays satisfy the criteria for value-alignment. Informally this situtation seems to meet the requirements for value-alignment. Unfortunately it does not satisfy the strict definition, and in general functions that assume value-alignment will not work unless the strict definition is adhered to.

The array, `a` is aligned with `b` *with replicated-alignment* in some dimensions if the groups are equivalent, and the ranges of `a` can be paired with equivalent ranges of `b` by omitting the ranges of `b` associated with the specified dimensions. *Value-alignment* with replication in specified dimensions is defined similarly.

[5] A special feature is that alignment equivalence of ranges does not imply that the ranges have the same *layout* options. So, for example, two arrays may be aligned even if one has ghost extensions and the other doesn't.

4.6 Accessibility and value-accessibility

An array, `a`, is *accessible* at a particular point in program execution if it is distributed over a group contained in the active process group:

$$\texttt{a.grp()} \subseteq \texttt{apg}$$

Informally this means that all copies of all elements of the array are available within the set of processes sharing the current thread of control.

The definition of *value-accessibility* is slightly more complicated. An array, `a`, is *value-accessible* at a particular point if it is accessible *or* if the dimension set of the intersection of `a.grp()` with the active process group[6] contains all process dimensions in `a.sig()`. This is a clumsy way of saying that `a.grp()` may have dimensions outside the active process group, but the array must be replicated over those dimensions. A sufficient condition for value-accessibility is that `a` be value-aligned to an accessible array.

Informally the meaning of value-accessibility is that at least one copy of every element of the array is available within the set of process sharing the current thread of control.

[6]This intersection is only well-defined if `a.grp()` and `apg` have the same parent process array.

Chapter 5

Communication schedules

This chapter defines the Adlib communication library. Currently all communication functions in Adlib take the form of collective transformations on distributed arrays. These transformations are implemented in terms *communication schedules*. Each type of transformation has an associated class of schedules. Specific instances of these schedules, involving particular data arrays and particular parameters, are created as objects from the classes. *Executing* a schedule initiates the communications required to implement the transformation. A single schedule may be executed many times, repeating the same communication pattern.

Future versions of Adlib may support other communication paradigms. A convenient extension would allow primitives for one-sided communication. These would permit direct read or write access to remote patches of distributed arrays.

5.1 General features of schedules

Typically the communication schedules described in this chapter have only three public members: a constructor, a destructor, and an `execute` member.

Usually the constructor is passed all the detailed information describing how the input and output data is organized and how it is to be transformed. This includes the array descriptors and any parameters of the transformation. The constructor may perform extensive processing on these arguments to convert them into a simplified list of data movements. In some cases this processing may involve exchange of information—ie, communication—between the active processes. A schedule constructor should always be treated as a collective operation. As usual with arguments of collective operations, every member of the active process group must pass consistent arguments to the schedule constructor call. In particular this means that any simple value passed to a schedule constructor (any argument that is not a local component of a collective object such as a DAD, or a vector of local array elements associated with a DAD) must have *identical* values in every process.

Local segment addresses of input and output arrays—the vectors where individual array elements are stored—are usually *not* passed to the schedule constructor. Instead these addresses are passed to the `execute` member. One practical reason for not storing the data pointers in schedule itself is that these vectors are typically allocated by the user, outside control of the library. The user's program may be written in a programming language other than C++. Cacheing pointers to memory allocated and managed outside the library itself can cause problems in some programming environments—especially in garbage-collected languages such as Java. Another advantage of specifying the data pointers at execution time rather than schedule construction time is that in principle it allows the same schedule to be used with different sets of arrays, providing corresponding arrays have identical DADs.

The `execute` member nearly always involves communication. It should of course be treated as a collective operation, executed by all members of the active process group.

Most schedule constructors will access the `apg` variable. The state of this variable should accurately describe the set of processes involved in creating the schedule. The active process group at the point of execution of a schedule should be the same as the active process group at its point of creation.

The following sections describe the interfaces of the schedule classes. The choice of transformations in the current schedule library has been strongly influenced by HPF. Apart from a handful of generic operations like `Remap`, `Gather`, `Scatter` and `WriteHalo`, most of the schedules are designed to support the specific array syntax and array transformational intrinsics of Fortran 90[1]. Focussing on Fortran provides a concrete standard with respect to which some kind of completeness can be achieved.

In the interfaces given in the following sections, only the constructor and the `execute` members will be exhibited. Base classes, public or otherwise, members of base classes, destructors, etc, will be suppressed. Subsections describing the constructors detail any restrictions the arguments must sat-

[1]An exception is the `MultiShift` schedule, which was actually introduced to support the array syntax of C*.

isfy. Subsections describing the `execute` members specify the effects of the schedules. Below we briefly discuss various terms and notations used, following the subheadings used in the schedule definitions.

5.1.1 Value restrictions

These are simply restrictions on the input values of data, such as constraints ensuring values used as subscripts are in the required bounds.

5.1.2 Type restrictions

These are restrictions on the types of array elements. Typically if an array's elements are to be communicated, they must have *POD* type. The idea of a POD type is defined in the ANSI C++ standard. Informally it is any type that can be copied to a byte array by standard operations such as `memcpy`.

Schedules that perform arithmetic operations or comparisions will impose further restrictions on the types of the array elements. In the current library all such schedules are template classes, parametrized by the element type. The template argument can only be instantiated to a type that supports a suitable set of arithmetic operations.

5.1.3 Shape restrictions

Restrictions on the shape of the array arguments, such as requirements that particular pairs of arrays passed to the constructor should be the same shape.

5.1.4 Alignment restrictions

Many of the schedules in the library assume some alignment relations (see section 4.5) between their array arguments. Although these are usually natural restrictions from the point of view of the parallel implementation, they can sometimes take programmers by surprise, or appear unnecessarily complex.

For example, it is required that the source array for a `Shift` is aligned with the destination array. This sometimes surprises people, although it shouldn't. Historically, an essential feature of the shift operation is that it can be implemented very efficiently by simple nearest neighbour communications. The library could easily have been defined to implement shift without the alignment constraint, but then implementation would be essentially the same as the more complex `Remap` operation. In some sense the whole point of `Shift` is that it is a simpler, lighter-weight operation. If versions of the library functions without alignment restrictions are needed, they can always be constructed by combining the constrained operation with `Remap` operations.

Because the alignment constraints implied by particular implementations of the schedules can be quite complex, the restrictions are sometimes simplified slightly in the following sections. The conditions given are always sufficient, but sometimes weaker conditions would have been adequate. For simplicity, we restrict ourselves to the vocabularly defined in section 4.5.

5.1.5 Accessibility restrictions

Accessibility restrictions are needed to ensure that necessary copies of array elements are available inside the group of processes that execute a schedule. Access to elements stored outside the active process group is unnatural in the context of the collective communication paradigm currently implemented by Adlib.

5.1.6 Argument persistence

This usually refers to the situation where a pointer to a DAD is passed to a schedule constructor. To avoid the overhead of copying the DAD, the schedule often saves a reference to the existing DAD object. The programmer must then ensure that the DAD is not deleted during the lifetime of the schedule.

Deleting a DAD before completing all communications involving the array would be strange practise, so these argument persistence restrictions are unlikely to be troublesome.

5.1.7 Effect

In describing the effect of schedules, array subscripting notation will often be used informally. In this context, the subscripting should always be understood in terms of global subscripts to abstract global arrays, without reference to the distributed nature of the actual arrays.

5.1.8 Replicated data

An array is replicated over a particular process dimension if the dimension appears in its process group but not its signature (ie, it the array no range distributed over the dimension).

As a rule it is good practise for programmers to maintain the same values in all copies of an element of a replicated array. If all arrays input to the communication schedules meet this requirement, it is guaranteed that those output do. This is not an absolute requirement on input arrays, and the sections on individual schedules discuss the effect of defaulting on this rule.

5.1.9 Overlap restrictions

In general the library does not allow in-place updates. No array written by a communication schedule should overlap with an array read by the schedule. The sections on individual schedules give the specific restrictions.

5.2 class `Remap`

A *remap schedule* is a communication schedule for copying the elements of one distributed array to another. The source and destination must have the same shape and same element-type, but no relation between the mapping of the two arrays is required. If the target array has a replicated mapping, the remap operation implements a broadcast.

A remap schedule is described by a collective object with local components of class `Remap`. The public interface of the `Remap` class is

```
class DAD ;

class Remap {
public :
  Remap(const DAD* dst, const DAD* src, const int len) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.2.1 Constructor

`Remap(const DAD* dst, const DAD* src, const int len)`

The source array is described by the DAD `*src` and the destination array is described by the DAD `*dst`. `len` specifies the size of each array element, in bytes.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

Shape restrictions: The source and destination array must have the same shape (see section 4.4).

Accessibility restrictions: The source array must be value-accessible and the destination array must be accessible (see section 4.6).

Argument persistence: The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.2.2 Method

`execute(void* dstDat, void* srcDat)`

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: Copy the elements of the source array to the corresponding elements of the destination array.

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from *one* of its copies. If the destination array has replicated mapping, identical values are broadcast to *every* copy of the element.

Overlap restrictions: In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.3 class Shift

A *shift schedule* is a communication schedule for shifting the elements of a distributed array along one of its dimensions, placing the result in another array. The source and destination have the same shape and same element-type, and they must have a certain alignment relation.

A shift schedule is described by a collective object with local components of class `Shift`. The public interface of the `Shift` class is

```
class DAD ;
enum Mode {CYCL, EDGE} ;

class Shift {
public :
  Shift(const DAD* dst, const DAD* src, const int len,
        const int shift, const int dim, const Mode mode) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.3.1 Constructor

```
Shift(const DAD* dst, const DAD* src, const int len,
      const int shift, const int dim, const Mode mode)
```

The source array is described by the DAD `*src` and the destination array is described by the DAD `*dst`. `len` specifies the size of each array element, in bytes. The shift amount, which may be negative, is given by `shift`. The `dim` argument selects the array dimension in which the shift occurs. The flag `mode` specifies the type of shift. It takes one of the values `CYCL`, `EDGE` or `NONE`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

Shape restrictions: The source and destination array must have the same shape (see section 4.4).

Alignment restrictions: The source array *must be value-aligned with* the destination array (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the destination array must be accessible (see section 4.6).

Argument persistence: The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.3.2 Method

`execute(void* dstDat, void* srcDat)`

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, if `mode` is `CYCL`, the value of

$\texttt{dst}\ [x_0, \ldots, x_{\texttt{dim}}, \ldots, x_{R-1}]$

is

$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim}} + \texttt{shift} \mod N, \ldots, x_{R-1}]$

where N is the extent of dimension `dim`. If `mode` is `EDGE`, the exit value of the `dst` element is

$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim}} + \texttt{shift}, \ldots, x_{R-1}]$

if $x_{\texttt{dim}} + \texttt{shift}$ is in the range $0, \ldots, N-1$, or *unchanged from the entry value*, if not. If `mode` is `NONE` executing the schedule has no effect.

Replicated data: If the arrays have replicated mapping, values for individual copies of the destination are generally taken from the nearest copy of the corresponding source array element. The definition of "nearest" is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array relies on consistency of copies of the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.4 class Skew

A *skew schedule* is a communication schedule for performing a skewed shift—a shift where the shift amount is itself an array—in a particular dimension of a distributed array placing the result in another array. The source and destination must have the same shape and same element-type, and they must have a certain alignment relation.

A skew schedule is described by a collective object with local components of class Skew. The public interface of the Skew class is

```
class DAD ;
enum Mode {CYCL, EDGE} ;

class Skew {
public :
  Skew(const DAD* dst, const DAD* src, const int len,
       const DAD* shf, int* shfDat, const int dim,
       const Mode mode) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.4.1 Constructor

```
Skew(const DAD* dst, const DAD* src, const int len,
     const DAD* shf, int* shfDat,
     const int dim, const Mode mode)
```

The source array is described by the DAD *src and the destination array is described by the DAD *dst. len specifies the size of each array element, in bytes. The array of shift amounts (any of which may be negative), is described by the DAD shf. The base address for the local segment of this array is shfDat. The shift-amount array should have rank one less than the source array. The dim argument selects the array dimension in which the shift occurs. The flag mode specifies the type of shift. It takes one of the values CYCL, EDGE or NONE.

Value restrictions: The value of dim must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

Shape restrictions: The source and destination array must have the same shape (see section 4.4). The shape of the shift array must be obtained from the shape of the source array by deleting dimension `dim`.

Alignment restrictions: The source array *must be value-aligned with* the destination array The shift-amount array should be value-aligned with the destination array, with replicated alignment over dimension `dim` (see section 4.5).

Accessibility restrictions: The source array and the shift-amount array must be value-accessible. The destination array must be accessible (see section 4.6).

Argument persistence: The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the shift-amount array).

5.4.2 Method

`execute(void* dstDat, void* srcDat)`

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: The description of the exit value of `dst` is identical to the description after execution of a `Shift` schedule (see section 5.3), except that the constant `shift` is replaced by

`shf` $[x_0, \ldots, x_{\texttt{dim}-1}, x_{\texttt{dim}+1}, \ldots, x_{R-1}]$

Replicated data: If the arrays have replicated mapping, values for individual copies of the destination are generally taken from the nearest copy of the corresponding source array element. The definition of "nearest" is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array relies on consistency of copies of the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.5 class `MultiShift`

A *multishift schedule* is a communication schedule for shifting the elements of a distributed array along some or all of its dimensions concurrently, placing the result in another array. The source and destination must have the same shape and same element-type, and they must have a certain alignment relation.

A multishift schedule is described by a collective object with local components of class `MultiShift`. The public interface of the `MultiShift` class is

```
class DAD ;
enum Mode {CYCL, EDGE} ;

class MultiShift {
public :
  MultiShift(const DAD* dst, const DAD* src, const int len,
             int shift [], Mode mode []) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.5.1 Constructor

```
MultiShift(const DAD* dst, const DAD* src, const int len,
           int shift [], Mode mode [])
```

The source array is described by the DAD `*src` and the destination array is described by the DAD `*dst`. `len` specifies the size of each array element, in bytes. Vectors `shift` and `mode` have extent R—the rank of the source array. The shift amount, which may be negative, in dimension r is given by `shift [`r`]`. The flag `mode [`r`]` specifies the type of shift: It takes one of the values `CYCL`, `EDGE` or `NONE`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

Shape restrictions: The source and destination array must have the same shape (see section 4.4).

Alignment restrictions: The source array *must be value-aligned with* the destination array (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the destination array must be accessible (see section 4.6).

Argument persistence: The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.5.2 Method

`execute(void* dstDat, void* srcDat)`

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: Equivalent in effect (not in implementation) to successive execution of R schedules of the form

`Shift(dst, src, len,` r`, shift [`r`], mode [`r`])`

for r in the range $0, \ldots, R-1$ (see section 5.3).

Replicated data: If the arrays have replicated mapping, values for individual copies of the destination are generally taken from the nearest copy of the corresponding source array element. The definition of "nearest" is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array relies on consistency of copies of the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.6 class `WriteHalo`

A *write-halo schedule* is a communication schedule for filling *overlap regions* or *ghost cells* surrounding the local segment of a distributed array. A write-halo schedule is described by a collective object with local components of class `WriteHalo`. The public interface of the `WriteHalo` class is

```
class DAD ;

class WriteHalo {
public :
  WriteHalo(const DAD* src, const int len,
            const int wlo [], const int whi [],
            const Mode [] mode) ;

  void execute(void* srcDat) ;
private :
  ...
}
```

5.6.1 Constructor

```
WriteHalo(const DAD* src, const int len,
          const int wlo [], const int whi [],
          const Mode [] mode)
```

The array is described by the DAD `*src`. `len` specifies the size of each array element, in bytes. The vectors `wlo`, `whi` and `mode` have extent R—the rank of the array.

Vectors `wlo` and `wlo` define the halo of ghost cells updated by the schedule. The upper and lower widths in dimension r are given by `wlo` [r] and `whi` [r]. These values are non-negative, and can only be non-zero if array `src` actually has suitable ghost extensions in the dimension concerned. More specifically, if the array `src` was created using a range with ghost extensions $\texttt{wlo}_{\text{act}}$, $\texttt{whi}_{\text{act}}$—eg, a range created by by a constructor call such as `BlockRange(N, d, LAY_EXTENDED,`$\texttt{wlo}_{\text{act}}$`,`$\texttt{whi}_{\text{act}}$`)`—as its rth dimension, it is required that

$$\texttt{whi [}r\texttt{]} \le \texttt{whi}_{\text{act}}$$
$$\texttt{wlo [}r\texttt{]} \le \texttt{wlo}_{\text{act}}$$

The situation is complicated if the array is a section, or some other array with non-trivial alignment stride. In practise it is unusual to construct `WriteHalo` schedules for such arrays, but for completeness we describe the constraints that apply in that case. First, by definition, the ghost extensions of a subrange are those of its template range. Now, suppose the *array of which* `src` *is a section* has ghost extensions $\texttt{wlo}_{\text{act}}$, $\texttt{whi}_{\text{act}}$. If the alignment stride of `src.rng(`r`)` is s, the required constraints are

$$s \times \texttt{whi [}r\texttt{]} \le \texttt{whi}_{\text{act}}$$
$$s \times \texttt{wlo [}r\texttt{]} \le \texttt{wlo}_{\text{act}}$$

The point to note is that the widths defined in the `WriteHalo` construtor are expressed in terms of the global subscript of the range of the array, whereas the ghost extensions of the array are measured in units of *template range* subscripts.

The vector `mode` defines how ghost cells are updated in each dimension—including how the cells at the extremes of the array are updated. Its elements take value `CYCL`, `EDGE` or `NONE`.

Note that (as usual for ordinary data arguments of collective constructors) the vectors `wlo`, `whi` and `mode` must have identical values in all members of the active process group.

Type restrictions: The elements of array must be a POD type (see section 5.1.2) of size `len` bytes.

Accessibility restrictions: The array must be accessible (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.6.2 Method

`execute(void* srcDat)`

Argument is the base addresses for the local segment of the array. It should point to a vector of the locally held elements.

Effect: We distinguish between the locally held *physical segment* of an array and the surrounding *ghost region*, which is used to cache local copies of remote elements. The effect of this operation is to overwrite a portion of the ghost region—a halo of extent defined by the `wlo`, `whi` vectors of the constructor—with values from processes holding the corresponding elements in their physical segments. The operation is visualized in figure 5.1.

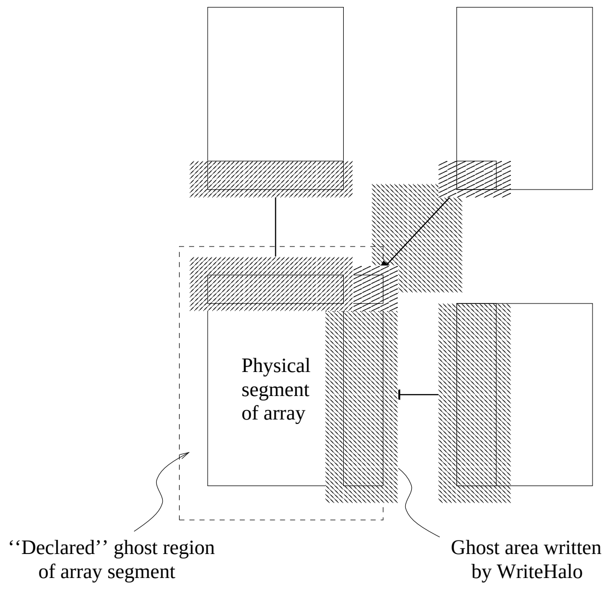

Figure 5.1: Illustration of the effect of executing a write-halo schedule.

Note that, so long as it fits in the ghost area *allocated* for the array, there is no restriction on the width of the halo region. In particular, the width of the halo region may be larger than the width of the neighbouring physical segment, in which case values will be fetched from next-nearest neighbours, and so on.

If the value of the `mode` element for a dimension is `EDGE`, ghost cells past the extreme ends of the array range are not updated by the the write-halo operation. If the value is `CYCL`, those cells are updated assuming cyclic wraparound[2]. If the value is `NONE`, there is no updating at all of the ghost cells associated with this dimension.

Replicated data: If the array has replicated mapping, values for individual copies of the ghost cell are generally taken from the nearest copy of the corresponding physical array element. The definition of "nearest" is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the final state of the array array relies on consistency of copies in the initial state of the array.

[2]This option may lead to odd effects if the source array is actually a section with a triplet subscript. The updated cells, past the ends of the section, may actually reside in the physical segment of the parent array.

5.7 class Gather

A *gather schedule* is a communication schedule for collecting an arbitrary set of values from one distributed array (the source array) into the elements of another (the destination array). The selected set of elements is defined by a vector of *subscript arrays*, with an optional *mask array*. A gather schedule is described by a collective object with local components of class `Gather`. The public interface of the `Gather` class is

```
class DAD ;

class Gather {
public :
  Gather(const DAD* dst, const DAD* src, const int len,
         const DAD* subs [], int* subsData [],
         const DAD* msk, int* mskData) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.7.1 Constructor

```
Gather(const DAD* dst, const DAD* src, const int len,
       const DAD* subs [], int* subsData [],
       const DAD* msk, int* mskData)
```

The source array is described by the DAD `*src` and the destination array is described by the DAD `*dst`. `len` specifies the size of each array element, in bytes. The vectors `subs` and `subsData` have extent R—the rank of the *source* array. The subscript arrays are defined by the pairs `*subs [`r`]`, `subsData [`r`]`. If `msk` is non-null, the pair `*msk`, `mskData` defines a mask array.

Value restrictions: All elements of the rth subscript array must be in the range $0, \ldots, N-1$ where N is the extent of the source array in its rth dimension.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

Shape restrictions: The destination array, all subscript arrays, and the mask array, if defined, must have the same shape (see section 4.4).

Alignment restrictions: All subscript arrays and the mask array, if defined, *must be value-aligned with* the destination array (see section 4.5).

Accessibility restrictions: The source and subscript arrays, and the mask array, if defined, must be value-accessible. The destination array must be accessible (see section 4.6).

Argument persistence: The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript and mask arrays).

5.7.2 Methods

`execute(void* dstDat, void* srcDat)`

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

`dst` $[x_0, \ldots, x_{S-1}]$

is

`src [subs`$_0$ $[x_0, \ldots, x_{S-1}]$`, ..., subs`$_{R-1}$ $[x_0, \ldots, x_{S-1}]$`]`

unless the mask array is defined and

`msk` $[x_0, \ldots, x_{S-1}]$

was zero, in which case the exit value of the `dst` element is unchanged from the entry value. Here `subs`$_r$ denotest the rth subscript array and S is the rank of the destination array.

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from *one* of its copies. If the destination array has replicated mapping, identical values are broadcast to *every* copy of each element.

Overlap restrictions: In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.8 class Scatter

A *scatter schedule* is a communication schedule for scattering values from one distributed array (the source array) into elements of another another (the destination array) in an arbitrary way. The target set of elements is defined by a vector of *subscript arrays*, with an optional *mask array*. A scatter schedule is described by a collective object with local components of class `Scatter`. The public interface of the `Scatter` class is

```
class DAD ;

class Scatter {
public :
  Scatter(const DAD* src, const DAD* dst, const int len,
          const DAD* subs [], int* subsData [],
          const DAD* msk, int* mskData) ;

  void execute(void* srcDat, void* dstDat) ;
private :
  ...
} ;
```

5.8.1 Constructor

```
Scatter(const DAD* src, const DAD* dst, const int len,
        const DAD* subs [], int* subsData [],
        const DAD* msk, int* mskData)
```

The source array is described by the DAD `*src` and the destination array is described by the DAD `*dst`. `len` specifies the size of each array element, in bytes. The vectors `subs` and `subsData` have extent R—the rank of the *destination* array. The subscript arrays are defined by the pairs `*subs [`r`]`, `subsData [`r`]`. If `msk` is non-null, the pair `*msk`, `mskData` defines a mask array.

Note well that the source and destination arguments of `Scatter` members are reversed relative to the conventions for other communication schedules. This is to emphasize the symmetry with `Gather`.

Value restrictions: All elements of the rth subscript array must be in the range $0, \ldots, N-1$ where N is the extent of the destination array in its rth dimension.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

Shape restrictions: The source array, all subscript arrays, and the mask array, if defined, must have the same shape (see section 4.4).

Alignment restrictions: All subscript arrays and the mask array, if defined, *must be value-aligned with* the source array (see section 4.5).

Accessibility restrictions: The source and subscript arrays, and the mask array, if defined, must be value-accessible. The destination array must be accessible (see section 4.6).

Argument persistence: The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript and mask arrays).

5.8.2 Methods

`execute(void* srcDat, void* dstDat)`

Arguments are the base addresses for local segments of the source and destination arrays. They should point to vectors of the locally held elements.

Effect: If `msk` was non-null, the effect is like executing the conditional assignment

```
if(msk [$x_0, \ldots, x_{S-1}$])
  dst [subs$_0$ [$x_0, \ldots, x_{S-1}$], ..., subs$_{R-1}$ [$x_0, \ldots, x_{S-1}$]] =
                      src [$x_0, \ldots, x_{S-1}$]
```

for every tuple $(x_0, \ldots, x_{S-1})$ of global subscripts of the source array. These assignments are executed in an undefined order. If `msk` is a null pointer, the assignment is unconditional. Here `subs`$_r$ denotest the rth subscript array.

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from *one* of its copies. If the destination array has replicated mapping, identical values are broadcast to *every* copy of each element.

Overlap restrictions: In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.9 class `VecGather`

A *vector-subscript gather schedule* is a communication schedule for collecting a set of values from one distributed array (the source array) into the elements of another array of the same rank (the destination array). The selected set of elements is defined by a vector of rank-1 arrays—the *subscript arrays*. This schedule can be regarded as an optimized special case of the general gather schedule of section 5.7. It is functionally equivalent to certain Fortran-90 array assignments. A vector-subscript gather schedule is described by a collective object with local components of class `VecGather`. The public interface of the `VecGather` class is

```
class DAD ;

class VecGather {
public :
  VecGather(const DAD* dst, const DAD* src, const int len,
            const DAD* subs [], int* subsData []) ;

  void execute(void* dstDat, void* srcDat) ;
private :
  ...
} ;
```

5.9.1 Constructor

```
VecGather(const DAD* dst, const DAD* src, const int len,
          const DAD* subs [], int* subsData [])
```

The source array is described by the DAD `*src` and the destination array is described by the DAD `*dst`. `len` specifies the size of each array element, in bytes. The source and destination arrays have the same rank, R. The vectors `subs` and `subsData` have extent R. If `subs [`r`]` is non-null, the pair `*subs [`r`]`, `subsData [`r`]` defines the rth subscript array.

Value restrictions: If the rth subscript array is defined, all its elements must be in the range $0, \ldots, N-1$ where N is the extent of the source array in its rth dimension.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

Shape restrictions: If the rth subscript array is defined it must be rank-1 and its extent must be M, where M is the extent of the destination array in its rth dimension.

If the rth subscript array is *undefined* (because `subs [`r`]` is null) the destination and source arrays must have the same extent in their rth dimensions.

Alignment restrictions: If defined, the rth subscript array must be value-aligned to the destination array, *with replicated alignment in all dimensions except* r (see section 4.5).

Accessibility restrictions: The source and subscript arrays must be value-accessible. The destination array must be accessible (see section 4.6).

Argument persistence: The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript arrays).

5.9.2 Methods

`execute(void* dstDat, void* srcDat)`

Arguments are the base addresses for local segments of the destination and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

`dst [`$x_0, \ldots, x_{S-1}$`]`

is

`src [subs`$_0$ `[`x_0`], ..., subs`$_{R-1}$ `[`x_{R-1}`]]`

Here `subs`$_r$ denotest the rth subscript array. If `subs [`r`]` is null, the value of the expression `subs`$_r$ `[`x_r`]` in this formula is defined to be x_r (hence if all subscripts are undefined, this operation is equivalent to a `Remap` operation).

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from *one* of its copies. If the destination array has replicated mapping, identical values are broadcast to *every* copy of each element.

Overlap restrictions: In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.10 class `VecScatter`

A *vector-subscript scatter schedule* is a communication schedule for scattering values from one distributed array (the source array) into elements of another (the destination array). The target set of elements are defined by a vector of rank-1 arrays—the *subscript arrays*. This schedule can be regarded as an optimized special case of the general scatter schedule of section 5.8. It is functionally similar to certain Fortran-90 array assignments—minus the constraints that enforce determinism in Fortran. A vector-subscript scatter schedule is described by a collective object with local components of class `VecScatter`. The public interface of the `VecScatter` class is

```
class DAD ;

class VecScatter {
public :
  VecScatter(const DAD* src, const DAD* dst, const int len,
             const DAD* subs [], int* subsData []) ;

  void execute(void* srcDat, void* dstDat) ;
private :
  ...
} ;
```

5.10.1 Constructor

```
VecScatter(const DAD* src, const DAD* dst, const int len,
           const DAD* subs [], int* subsData [])
```

The source array is described by the DAD `*src` and the destination array is described by the DAD `*dst`. `len` specifies the size of each array element, in bytes. The source and destination arrays have the same rank, R. The vectors `subs` and `subsData` have extent R. If `subs [`r`]` is non-null, the pair `*subs [`r`]`, `subsData [`r`]` defines the rth subscript array.

Note well that the source and destination arguments of `VecScatter` members are reversed relative to the conventions for other communication schedules. This is to emphasize the symmetry with `VecGather`.

Value restrictions: If the rth subscript array is defined, all its elements must be in the range $0, \ldots, N-1$ where N is the extent of the destination array in its rth dimension.

Type restrictions: The elements of the source and destination arrays must have the same type. This must be a POD type (see section 5.1.2) of size `len` bytes.

Shape restrictions: If the rth subscript array is defined it must be rank-1 and its extent must be M, where M is the extent of the source array in its rth dimension.

If the rth subscript array is *undefined* (because `subs` [r] is null) the destination and source arrays must have the same extent in their rth dimensions.

Alignment restrictions: If defined, the rth subscript array must be value-aligned to the destination array, *with replicated alignment in all dimensions except r* (see section 4.5).

Accessibility restrictions: The source and subscript arrays must be value-accessible. The destination array must be accessible (see section 4.6).

Argument persistence: The `dst` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule. (There are no such requirements for the subscript arrays).

5.10.2 Method

```
execute(void* srcDat, void* dstDat)
```

Arguments are the base addresses for local segments of the source and destination arrays. They should point to vectors of the locally held elements.

Effect: The effect is like executing the assignment

$$\texttt{dst [subs}_0\texttt{ [}x_0\texttt{], ..., subs}_{R-1}\texttt{ [}x_{R-1}\texttt{]] = src [}x_0, \ldots, x_{R-1}\texttt{]}$$

for every tuple, $[x_0, \ldots, x_{R-1}]$, of global subscripts of the source array. These assignments are executed in an undefined order. Here $\texttt{subs}_r$ denotest the rth subscript array. If `subs` [r] is null, the value of the expression $\texttt{subs}_r$ [x_r] in the left-hand-side of the assignment is defined to be x_r (hence if all subscripts are undefined, this operation is equivalent to a `Remap` operation).

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from *one* of its copies. If the destination array has replicated mapping, identical values are broadcast to *every* copy of each element.

Overlap restrictions: In-place updates are not allowed. The pairs `*dst`, `dstDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.11 class `ScatterComb`

[There are are existing schedules for combining scatter, but their interface is under revision.]

5.12 class `Reshape`

[There are are existing schedules for this F90 intrinsic, but their interface is under revision.]

5.13 class `Sum`

A *sum schedule* is a communication schedule for adding together all elements of a distributed array (the source array). A sum schedule is described by a collective object with local components of class `Sum`. The public interface of the `Sum` class is

```
class DAD ;

template<class T>
class Sum {
public :
  Sum(const DAD* src) ;

  void execute(T* res, T* srcDat) ;
private :
  ...
} ;
```

5.13.1 Constructor

`Sum(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `T`.

Type restrictions: `T` should be a POD type (see section 5.1.2). A conventional binary addition operator, `+`, must be defined for objects of type `T`, the assignment operator `+=` must operate in a compatible way, and the symbol 0 must be convertible to type `T`, yielding a zero of the addition operation.

All standard arithmetic types of C++ satisfy these requirements for `T`.

Accessibility restrictions: The source array must be value-accessible (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.13.2 Method

`execute(T* res, T* srcDat)`

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

Effect: Executing the schedule adds together all elements of the array in an unspecified order. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.14 class SumMsk

A *masked sum schedule* is a communication schedule for adding together all elements of a distributed array (the source array) under the control of a mask array. A masked sum schedule is described by a collective object with local components of class `SumMsk`. The public interface of the `SumMsk` class is

```
class DAD ;

template<class T>
class SumMsk {
public :
  SumMsk(const DAD* src, const DAD* msk) ;

  void execute(T* res, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.14.1 Constructor

`SumMsk(const DAD* src, const DAD* msk)`

The source array is described by the DAD `*src`. It will have elements of type `T`. The mask array is described by the DAD `*msk`.

Type restrictions: `T` should be a POD type (see section 5.1.2). A conventional binary addition operator, `+`, must be defined for objects of type `T`, the assignment operator `+=` must operate in a compatible way, and the symbol 0 must be convertible to type `T`, yielding a zero of the addition operation.

All standard arithmetic types of C++ satisfy these requirements for `T`.

Shape restrictions: The mask array must have the same shape as the source array (see section 4.4).

Alignment restrictions: The mask array *must be value-aligned with* the source array (see section 4.5).

Accessibility restrictions: The source array and the mask array must be value-accessible. (see section 4.6).

Argument persistence: The `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.14.2 Method

```
execute(T* res, T* srcDat, int* mskDat)
```

The arguments are the address to which the result should be written and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

Effect: Executing the schedule adds together all elements of the array for which the corresponding element of the mask array is non-zero. The addition is performed in an unspecified order. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.15 class Product

A *product schedule* is a communication schedule for multiplying together all elements of a distributed array (the source array). A product schedule is described by a collective object with local components of class `Product`. The public interface of the `Product` class is

```
class DAD ;

template<class T>
class Product {
public :
  Product(const DAD* src) ;

  void execute(T* res, T* srcDat) ;
private :
  ...
} ;
```

5.15.1 Constructor

`Product(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `T`.

Type restrictions: `T` should be a POD type (see section 5.1.2). A conventional binary multiplication operator, `*`, must be defined for objects of type `T`, the assignment operator `*=` must operate in a compatible way, and the symbol 1 must be convertible to type `T`, yielding a unit of the multiplication operation.

All standard arithmetic types of C++ satisfy these requirements for `T`.

Accessibility restrictions: The source array must be value-accessible (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.15.2 Method

`execute(T* res, T* srcDat)`

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

Effect: Executing the schedule multiplies together all elements of the array in an unspecified order. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.16 class ProductMsk

A *masked product schedule* is a communication schedule for multiplying together all elements of a distributed array (the source array) under the control of a mask array. A masked product schedule is described by a collective object with local components of class `ProductMsk`. The public interface of the `ProductMsk` class is

```
class DAD ;

template<class T>
class ProductMsk {
public :
  ProductMsk(const DAD* src, const DAD* msk) ;

  void execute(T* res, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.16.1 Constructor

`ProductMsk(const DAD* src, const DAD* msk)`

The source array is described by the DAD `*src`. It will have elements of type `T`. The mask array is described by the DAD `*msk`.

Type restrictions: `T` should be a POD type (see section 5.1.2). A conventional binary multiplication operator, `*`, must be defined for objects of type `T`, the assignment operator `*=` must operate in a compatible way, and the symbol 1 must be convertible to type `T`, yielding a unit of the multiplication operation.

All standard arithmetic types of C++ satisfy these requirements for `T`.

Shape restrictions: The mask array must have the same shape as the source array (see section 4.4).

Alignment restrictions: The mask array *must be value-aligned with* the source array (see section 4.5).

Accessibility restrictions: The source array and the mask array must be value-accessible. (see section 4.6).

Argument persistence: The `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.16.2 Method

```
execute(T* res, T* srcDat, int* mskDat)
```

The arguments are the address to which the result should be written and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

Effect: Executing the schedule multiplies together all elements of the array for which the corresponding element of the mask array is non-zero. The mulitplication is performed in an unspecified order. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.17 class `Maxval`

A *maximum value schedule* is a communication schedule for finding the largest element of a distributed array (the source array). A maximum value schedule is described by a collective object with local components of class `Maxval`. The public interface of the `Maxval` class is

```
class DAD ;

template<class T>
class Maxval {
public :
  Maxval(const DAD* src) ;

  void execute(T* res, T* srcDat) ;
private :
  ...
} ;
```

5.17.1 Constructor

`Maxval(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `T`.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Accessibility restrictions: The source array must be value-accessible (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.17.2 Method

`execute(T* res, T* srcDat)`

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

Effect: Executing the schedule finds the largest element of the array. The maximum value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.18 class `MaxvalMsk`

A *masked maximum value schedule* is a communication schedule for finding the largest element of a distributed array (the source array) under the control of a mask array. A masked maximum value schedule is described by a collective object with local components of class `MaxvalMsk`. The public interface of the `MaxvalMsk` class is

```
class DAD ;

template<class T>
class MaxvalMsk {
public :
  MaxvalMsk(const DAD* src, const DAD* msk) ;

  void execute(T* res, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.18.1 Constructor

`MaxvalMsk(const DAD* src, const DAD* msk)`

The source array is described by the DAD `*src`. It will have elements of type `T`. The mask array is described by the DAD `*msk`.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The mask array must have the same shape as the source array (see section 4.4).

Alignment restrictions: The mask array *must be value-aligned with* the source array (see section 4.5).

Accessibility restrictions: The source array and the mask array must be value-accessible. (see section 4.6).

Argument persistence: The `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.18.2 Method

```
execute(T* res, T* srcDat, int* mskDat)
```

The arguments are the address to which the result should be written and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

Effect: Executing the schedule finds the largest element of the array for which the corresponding element of the mask array is non-zero. The maximum value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.19 class `Minval`

A *minimum value schedule* is a communication schedule for finding the smallest element of a distributed array (the source array). A minimum value schedule is described by a collective object with local components of class `Minval`. The public interface of the `Minval` class is

```
class DAD ;

template<class T>
class Minval {
public :
  Minval(const DAD* src) ;

  void execute(T* res, T* srcDat) ;
private :
  ...
} ;
```

5.19.1 Constructor

`Minval(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `T`.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostpos` must be overloaded with the entry point

```
void mostpos(T* res) ;
```

which sets `*res` to the largest allowed value of type `T`. The Adlib header files define `mostpos` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Accessibility restrictions: The source array must be value-accessible (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.19.2 Method

`execute(T* res, T* srcDat)`

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

Effect: Executing the schedule finds the smallest element of the array. The minimum value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.20 class `MinvalMsk`

A *masked minimum schedule* is a communication schedule for finding the smallest element of a distributed array (the source array) under the control of a mask array. A masked minimum schedule is described by a collective object with local components of class `MinvalMsk`. The public interface of the `MinvalMsk` class is

```
class DAD ;

template<class T>
class MinvalMsk {
public :
  MinvalMsk(const DAD* src, const DAD* msk) ;

  void execute(T* res, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.20.1 Constructor

`MinvalMsk(const DAD* src, const DAD* msk)`

The source array is described by the DAD `*src`. It will have elements of type `T`. The mask array is described by the DAD `*msk`.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostpos` must be overloaded with the entry point

```
void mostpos(T* res) ;
```

which sets `*res` to the largest allowed value of type `T`. The Adlib header files define `mostpos` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The mask array must have the same shape as the source array (see section 4.4).

Alignment restrictions: The mask array *must be value-aligned with* the source array (see section 4.5).

Accessibility restrictions: The source array and the mask array must be value-accessible. (see section 4.6).

Argument persistence: The `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.20.2 Method

```
execute(T* res, T* srcDat, int* mskDat)
```

The arguments are the address to which the result should be written and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

Effect: Executing the schedule finds the smallest element of the array for which the corresponding element of the mask array is non-zero. The minimum value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.21 class `All`

A *all schedule* is a communication schedule for computing the logical conjunction of the elements of a distributed array of boolean values (the source array). An all schedule is described by a collective object with local components of class `All`. The public interface of the `All` class is

```
class DAD ;

class All {
public :
  All(const DAD* src) ;

  void execute(int* res, int* srcDat) ;
private :
  ...
} ;
```

5.21.1 Constructor

`All(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `int`.

Accessibility restrictions: The source array must be value-accessible (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.21.2 Method

`execute(int* res, int* srcDat)`

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

Effect: Executing the schedule forms the logical conjunction (boolean *and*) of the elements of the array. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.22 class `Any`

A *any schedule* is a communication schedule for computing the logical disjunction of the elements of a distributed array of boolean values (the source array). An any schedule is described by a collective object with local components of class `Any`. The public interface of the `Any` class is

```
class DAD ;

class Any {
public :
  Any(const DAD* src) ;

  void execute(int* res, int* srcDat) ;
private :
  ...
} ;
```

5.22.1 Constructor

`Any(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `int`.

Accessibility restrictions: The source array must be value-accessible (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.22.2 Method

`execute(int* res, int* srcDat)`

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

Effect: Executing the schedule forms the logical disjunction (boolean *or*) of the elements of the array. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.23 class `Count`

A *count schedule* is a communication schedule for counting the number of true elements in a distributed array of boolean values (the source array). A count schedule is described by a collective object with local components of class `Count`. The public interface of the `Count` class is

```
class DAD ;

class Count {
public :
  Count(const DAD* src) ;

  void execute(int* res, int* srcDat) ;
private :
  ...
} ;
```

5.23.1 Constructor

`Count(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `int`.

Accessibility restrictions: The source array must be value-accessible (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.23.2 Method

`execute(int* res, int* srcDat)`

The arguments are the address to which the result should be written and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

Effect: Executing the schedule returns the number of true (non-zero) elements of the array. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.24 class `DotProduct`

A *dot product schedule* is a communication schedule for computing the dot product of two distributed arrays (the source arrays). A dot product schedule is described by a collective object with local components of class `DotProduct`. The public interface of the `DotProduct` class is

```
class DAD ;

template<class S, class T, class U>
class DotProduct {
public :
  DotProduct(const DAD* src1, const DAD* src2) ;

  void execute(S* res, T* src1Dat, U* src2Dat) ;
private :
  ...
} ;
```

5.24.1 Constructor

`DotProduct(const DAD* src1, const DAD* src2)`

The source arrays is described by the DADs `*src1` and `*src2`. They will have elements of type `T` and `U` respectively. The result will have type `S`.

Type restrictions: `S` should be a POD type (see section 5.1.2). A conventional binary multiplication operator, `*`, must be defined to multiply objects of type `T` with objects of type `U`. The assignment operator `+=` must be defined on objects of type `S`. This operator must be able to accept an RHS operand of type `S`, and must also be able to accept an operand produced by multiplying together `T` and `U` (if this is has type different to `S`). The symbol 0 must be convertible to type `S`, yielding a unit of the addition operation.

All standard arithmetic types of C++ satisfy these requirements, provided the product of a `T` with a `U` is convertible to an `S`.

Shape restrictions: The two source arrays must have the same shape (see section 4.4).

Alignment restrictions: The source arrays must be aligned with one another (see section 4.5).

Accessibility restrictions: The source arrays must be value-accessible (see section 4.6).

Argument persistence: The `src1` and `src2` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.24.2 Method

```
void execute(S* res, T* src1Dat, U* src2Dat) ;
```

The arguments are the address to which the result should be written and the base addresses of the local segments of the source arrays. The latter should point to vectors of the locally held elements.

Effect: Executing the schedule multiplies together corresponding elements of the source arrays, in pairs, then adds together all pairwise products. The addition occurs in an unspecified order. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source arrays have replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.25 class `BoolDotProduct`

A *boolean dot product schedule* is a communication schedule for computing the boolean dot product of two distributed arrays of boolean values (the source arrays). A boolean dot product schedule is described by a collective object with local components of class `BoolDotProduct`. The public interface of the `BoolDotProduct` class is

```
class DAD ;

class BoolDotProduct {
public :
  BoolDotProduct(const DAD* src1, const DAD* src2) ;

  void execute(int* res, int* src1Dat, int* src2Dat) ;
private :
  ...
} ;
```

5.25.1 Constructor

`BoolDotProduct(const DAD* src1, const DAD* src2)`

The source arrays is described by the DADs `*src1` and `*src2`.

Shape restrictions: The two source arrays must have the same shape (see section 4.4).

Alignment restrictions: The source arrays must be aligned with one another (see section 4.5).

Accessibility restrictions: The source arrays must be value-accessible (see section 4.6).

Argument persistence: The `src1` and `src2` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.25.2 Method

`void execute(int* res, int* src1Dat, int* src2Dat) ;`

The arguments are the address to which the result should be written and the base addresses of the local segments of the source arrays. The latter should point to vectors of the locally held elements.

Effect: Executing the schedule computes logical *and* of corresponding elements of the source arrays, in pairs, then computes logical *or* of all the pairwise conjunctions. The result value, written to `*res`, is broadcast to all members of the active process group.

Replicated data: If the source arrays have replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.26 class `Maxloc`

A *maximum location schedule* is a communication schedule for finding the location of the largest element of a distributed array. A maximum location schedule is described by a collective object with local components of class `Maxloc`. The public interface of the `Maxloc` class is

```
class DAD ;

template<class T>
class Maxloc {
public :
  Maxloc(const DAD* src) ;

  void execute(T* val, int pos [], T* srcDat) ;
private :
  ...
} ;
```

5.26.1 Constructor

`Maxloc(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `T`.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Accessibility restrictions: The source array must be value-accessible. (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.26.2 Method

```
execute(T* val, int pos [], T* srcDat)
```

The arguments are the addresses to which the result value and location should be written, and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

The size of the `pos` vector must be equal to the rank of the source array.

Effect: The value of largest element in the array is written to `*val`. The global subscripts of the first occurrence of this element are written to the vector `pos`. If the maximum value occurs more than once in the array, "first occurence" is defined by ordering the set of global subscripts with first subscript *least* significant (Fortran-like array-element ordering).

The result values written to `*val` and `pos` are broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for a particular element are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.27 class `MaxlocMsk`

A *masked maximum location schedule* is a communication schedule for finding the location of the largest element of a distributed array, under the control of a mask array. A masked maximum location schedule is described by a collective object with local components of class `MaxlocMsk`. The public interface of the `MaxlocMsk` class is

```
class DAD ;

template<class T>
class MaxlocMsk {
public :
  Maxloc(const DAD* src, const DAD* msk) ;

  void execute(T* val, int pos [], T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.27.1 Constructor

`MaxlocMsk(const DAD* src, const DAD* msk)`

The source array is described by the DAD `*src`. It will have elements of type `T`. The mask array is described by the DAD `*msk`.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The mask array must have the same shape as the source array (see section 4.4).

Alignment restrictions: The mask array *must be value-aligned with* the source array (see section 4.5).

Accessibility restrictions: The source array and the mask array must be value-accessible. (see section 4.6).

Argument persistence: The `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.27.2 Method

```
execute(T* val, int pos [], T* srcDat, int* mskDat)
```

The arguments are the addresses to which the result value and location should be written, and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

The size of the `pos` vector must be equal to the rank of the source array.

Effect: The value of largest element in the array for which the corresponding element of the mask array is non-zero is written to `*val`. The global subscripts of the first occurrence of this element are written to the vector `pos`. If the maximum value occurs more than once in the unmasked part of the array, "first occurence" is defined by ordering the set of global subscripts with first subscript *least* significant (Fortran-like array-element ordering).

The result values written to `*val` and `pos` are broadcast to all members of the active process group.

Replicated data: If the source or mask array has replicated mapping, values for a particular element are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.28 class `Minloc`

A *minimum location schedule* is a communication schedule for finding the location of the smallest element of a distributed array. A minimum location schedule is described by a collective object with local components of class `Minloc`. The public interface of the `Minloc` class is

```
  class DAD ;

  template<class T>
  class Minloc {
  public :
    Minloc(const DAD* src) ;

    void execute(T* val, int pos [], T* srcDat) ;
  private :
    ...
  } ;
```

5.28.1 Constructor

`Minloc(const DAD* src)`

The source array is described by the DAD `*src`. It will have elements of type `T`.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostpos` must be overloaded with the entry point

```
  void mostpos(T* res) ;
```

which sets `*res` to the largest allowed value of type `T`. The Adlib header files define `mostpos` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Accessibility restrictions: The source array must be value-accessible. (see section 4.6).

Argument persistence: The `src` argument is stored in the schedule as a reference. The associated object must persist for the lifetime of the constructed schedule.

5.28.2 Method

`execute(T* val, int pos [], T* srcDat)`

The arguments are the addresses to which the result value and location should be written, and the base address of the local segment of the source array. The latter should point to a vector of the locally held elements.

The size of the `pos` vector must be equal to the rank of the source array.

Effect: The value of smallest element in the array is written to `*val`. The global subscripts of the first occurrence of this element are written to the vector `pos`. If the minimum value occurs more than once in the array, "first occurence" is defined by ordering the set of global subscripts with first subscript *least* significant (Fortran-like array-element ordering).

The result values written to `*val` and `pos` are broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for a particular element are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.29 class `MinlocMsk`

A *masked minimum location schedule* is a communication schedule for finding the location of the smallest element of a distributed array, under the control of a mask array. A masked minimum location schedule is described by a collective object with local components of class `MinlocMsk`. The public interface of the `MinlocMsk` class is

```
class DAD ;

template<class T>
class MinlocMsk {
public :
  Minloc(const DAD* src, const DAD* msk) ;

  void execute(T* val, int pos [], T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.29.1 Constructor

`MinlocMsk(const DAD* src, const DAD* msk)`

The source array is described by the DAD `*src`. It will have elements of type `T`. The mask array is described by the DAD `*msk`.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostpos` must be overloaded with the entry point

```
void mostpos(T* res) ;
```

which sets `*res` to the largest allowed value of type `T`. The Adlib header files define `mostpos` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The mask array must have the same shape as the source array (see section 4.4).

Alignment restrictions: The mask array *must be value-aligned with* the source array (see section 4.5).

Accessibility restrictions: The source array and the mask array must be value-accessible. (see section 4.6).

Argument persistence: The `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.29.2 Method

```
execute(T* val, int pos [], T* srcDat, int* mskDat)
```

The arguments are the addresses to which the result value and location should be written, and the base addresses of the local segments of the source and mask arrays. The latter should point to vectors of the locally held elements.

The size of the `pos` vector must be equal to the rank of the source array.

Effect: The value of smallest element in the array for which the corresponding element of the mask array is non-zero is written to `*val`. The global subscripts of the first occurrence of this element are written to the vector `pos`. If the minimum value occurs more than once in the unmasked part of the array, "first occurence" is defined by ordering the set of global subscripts with first subscript *least* significant (Fortran-like array-element ordering).

The result values written to `*val` and `pos` are broadcast to all members of the active process group.

Replicated data: If the source or mask array has replicated mapping, values for a particular element are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

5.30 class SumDim

A *dimension sum schedule* is a communication schedule for summing the elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension sum schedule is described by a collective object with local components of class `SumDim`. The public interface of the `SumDim` class is

```
class DAD ;

template<class T>
class SumDim {
public :
  SumDim(const DAD* res, const DAD* src, const int dim) ;

  void execute(T* resDat, T* srcDat) ;
private :
  ...
} ;
```

5.30.1 Constructor

`SumDim(const DAD* res, const DAD* src, const int dim)`

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `T`. The reduction occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type (see section 5.1.2). A conventional binary addition operator, `+`, must be defined for objects of type `T`, the assignment operator `+=` must operate in a compatible way, and the symbol 0 must be convertible to type `T`, yielding a zero of the addition operation.

All standard arithmetic types of C++ satisfy these requirements for `T`.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.30.2 Method

`execute(T* resDat, T* srcDat)`

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is

$$\sum_{x_{\texttt{dim}}=0}^{N-1} \texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

where N is the extent of the source array in dimension `dim`. The sums are performed in an unspecified order.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the sums are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*res`, `resDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.31 class SumDimMsk

A *masked dimension sum schedule* is a communication schedule for summing, under the control of mask, the elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A masked dimension sum schedule is described by a collective object with local components of class SumDimMsk. The public interface of the SumDimMsk class is

```
class DAD ;

template<class T>
class SumDimMsk {
public :
  SumDimMsk(const DAD* res, const DAD* src, const int dim,
            const DAD* msk) ;

  void execute(T* resDat, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.31.1 Constructor

```
SumDimMsk(const DAD* res, const DAD* src, const int dim,
          const DAD* msk)
```

The source array is described by the DAD *src and the result array is described by the DAD *res. They will both have elements of type T. The reduction occurs in dimension dim. The mask array is described by the DAD *msk.

Value restrictions: The value of dim must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: T should be a POD type (see section 5.1.2). A conventional binary addition operator, +, must be defined for objects of type T, the assignment operator += must operate in a compatible way, and the symbol 0 must be convertible to type T, yielding a zero of the addition operation.

All standard arithmetic types of C++ satisfy these requirements for T.

Shape restrictions: The mask array must be the same shape as the source array. The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The mask array must be value-aligned with the source array. The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source and mask arrays must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res`, `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.31.2 Method

`execute(T* resDat, T* srcDat, int* mskDat)`

Arguments are the base addresses for local segments of the result, source and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is

$$\sum_{\substack{x_{\texttt{dim}} = 0 \\ \texttt{msk}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, \\ x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq 0}}^{N-1} \texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

where N is the extent of the source array in dimension `dim`. The sum is performed in an unspecified order.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the sums are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pair `*res`, `resDat` must define an array that has no overlap with the arrays defined by either of the pairs `*src`, `srcDat` and `*msk`, `mskDat`

5.32 class `ProductDim`

A *dimension product schedule* is a communication schedule for multiplying together the elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension product schedule is described by a collective object with local components of class `ProductDim`. The public interface of the `ProductDim` class is

```
class DAD ;

template<class T>
class ProductDim {
public :
  ProductDim(const DAD* res, const DAD* src, const int dim) ;

  void execute(T* resDat, T* srcDat) ;
private :
  ...
} ;
```

5.32.1 Constructor

`ProductDim(const DAD* res, const DAD* src, const int dim)`

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `T`. The reduction occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type (see section 5.1.2). A conventional binary multiplication operator, `*`, must be defined for objects of type `T`, the assignment operator `*=` must operate in a compatible way, and the symbol 1 must be convertible to type `T`, yielding a unit of the multiplication operation.

All standard arithmetic types of C++ satisfy these requirements for `T`.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.32.2 Method

```
execute(T* resDat, T* srcDat)
```

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is

$$\prod_{x_{\texttt{dim}}=0}^{N-1} \texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

where N is the extent of the source array in dimension `dim`. The product is computed in an unspecified order.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the product are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*res`, `resDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.33 class ProductDimMsk

A *masked dimension product schedule* is a communication schedule for multiplying together, under the control of mask, the elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A masked dimension product schedule is described by a collective object with local components of class `ProductDimMsk`. The public interface of the `ProductDimMsk` class is

```
class DAD ;

template<class T>
class ProductDimMsk {
public :
  ProductDimMsk(const DAD* res, const DAD* src, const int dim,
                const DAD* msk) ;

  void execute(T* resDat, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.33.1 Constructor

```
ProductDimMsk(const DAD* res, const DAD* src, const int dim,
          const DAD* msk)
```

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `T`. The reduction occurs in dimension `dim`. The mask array is described by the DAD `*msk`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type (see section 5.1.2). A conventional binary multiplication operator, `*`, must be defined for objects of type `T`, the assignment operator `*=` must operate in a compatible way, and the symbol 1 must be convertible to type `T`, yielding a unit of the multiplication operation.

All standard arithmetic types of C++ satisfy these requirements for `T`.

Shape restrictions: The mask array must be the same shape as the source array. The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The mask array must be value-aligned with the source array. The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source and mask arrays must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res`, `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.33.2 Method

`execute(T* resDat, T* srcDat, int* mskDat)`

Arguments are the base addresses for local segments of the result, source and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is

$$\prod_{\substack{x_{\texttt{dim}} = 0 \\ \texttt{msk}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, \\ x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq 0}}^{N-1} \texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

where N is the extent of the source array in dimension `dim`. The product is computed in an unspecified order.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the product are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pair `*res`, `resDat` must define an array that has no overlap with the arrays defined by either of the pairs `*src`, `srcDat` and `*msk`, `mskDat`

5.34 class `MaxvalDim`

A *dimension maximum value schedule* is a communication schedule for finding the largest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension maximum value schedule is described by a collective object with local components of class `MaxvalDim`. The public interface of the `MaxvalDim` class is

```
class DAD ;

template<class T>
class MaxvalDim {
public :
  MaxvalDim(const DAD* res, const DAD* src, const int dim) ;

  void execute(T* resDat, T* srcDat) ;
private :
  ...
} ;
```

5.34.1 Constructor

`MaxvalDim(const DAD* res, const DAD* src, const int dim)`

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `T`. The reduction occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.34.2 Method

```
execute(T* resDat, T* srcDat)
```

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

`res` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the maximum value of

`src` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

across allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values of particular elements for comparision are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*res`, `resDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.35 class MaxvalDimMsk

A *masked dimension maximum value schedule* is a communication schedule for finding, under the control of mask, the largest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A masked dimension maximum value schedule is described by a collective object with local components of class `MaxvalDimMsk`. The public interface of the `MaxvalDimMsk` class is

```
class DAD ;

template<class T>
class MaxvalDimMsk {
public :
  MaxvalDimMsk(const DAD* res, const DAD* src, const int dim,
               const DAD* msk) ;

  void execute(T* resDat, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.35.1 Constructor

```
MaxvalDimMsk(const DAD* res, const DAD* src, const int dim,
          const DAD* msk)
```

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `T`. The reduction occurs in dimension `dim`. The mask array is described by the DAD `*msk`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The mask array must be the same shape as the source array. The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The mask array must be value-aligned with the source array. The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source and mask arrays must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res`, `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.35.2 Method

```
execute(T* resDat, T* srcDat, int* mskDat)
```

Arguments are the base addresses for local segments of the result, source and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the maximum value of

$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

for which

$$\texttt{msk}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq 0$$

over the allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular elements for comparision are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pair `*res`, `resDat` must define an array that has no overlap with the arrays defined by either of the pairs `*src`, `srcDat` and `*msk`, `mskDat`

5.36 class `MinvalDim`

A *dimension minimum value schedule* is a communication schedule for finding the smallest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension minimum value schedule is described by a collective object with local components of class `MinvalDim`. The public interface of the `MinvalDim` class is

```
class DAD ;

template<class T>
class MinvalDim {
public :
  MinvalDim(const DAD* res, const DAD* src, const int dim) ;

  void execute(T* resDat, T* srcDat) ;
private :
  ...
} ;
```

5.36.1 Constructor

`MinvalDim(const DAD* res, const DAD* src, const int dim)`

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `T`. The reduction occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostpos` must be overloaded with the entry point

```
void mostpos(T* res) ;
```

which sets `*res` to the largest allowed value of type `T`. The Adlib header files define `mostpos` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.36.2 Method

```
execute(T* resDat, T* srcDat)
```

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the minimum value of

$$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

across allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values of particular elements for comparision are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*res`, `resDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.37 class `MinvalDimMsk`

A *masked dimension minimum value schedule* is a communication schedule for finding, under the control of mask, the smallest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A masked dimension minimum value schedule is described by a collective object with local components of class `MinvalDimMsk`. The public interface of the `MinvalDimMsk` class is

```
class DAD ;

template<class T>
class MinvalDimMsk {
public :
  MinvalDimMsk(const DAD* res, const DAD* src, const int dim,
               const DAD* msk) ;

  void execute(T* resDat, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.37.1 Constructor

```
MinvalDimMsk(const DAD* res, const DAD* src, const int dim,
             const DAD* msk)
```

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `T`. The reduction occurs in dimension `dim`. The mask array is described by the DAD `*msk`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostpos` must be overloaded with the entry point

```
void mostpos(T* res) ;
```

which sets `*res` to the largest allowed value of type `T`. The Adlib header files define `mostpos` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The mask array must be the same shape as the source array. The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The mask array must be value-aligned with the source array. The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source and mask arrays must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res`, `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.37.2 Method

```
execute(T* resDat, T* srcDat, int* mskDat)
```

Arguments are the base addresses for local segments of the result, source and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the minimum value of

$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

for which

$$\texttt{msk}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq 0$$

over the allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular elements for comparision are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pair `*res`, `resDat` must define an array that has no overlap with the arrays defined by either of the pairs `*src`, `srcDat` and `*msk`, `mskDat`

5.38 class `AllDim`

A *dimension all schedule* is a communication schedule for computing the logical conjunction of the elements of a distributed array of boolean along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension all schedule is described by a collective object with local components of class `AllDim`. The public interface of the `AllDim` class is

```
class DAD ;

class AllDim {
public :
  AllDim(const DAD* res, const DAD* src, const int dim) ;

  void execute(int* resDat, int* srcDat) ;
private :
  ...
} ;
```

5.38.1 Constructor

`AllDim(const DAD* res, const DAD* src, const int dim)`

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `int`. The reduction occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.38.2 Method

`execute(int* resDat, int* srcDat)`

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is true if

$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is true (non-zero) for *all* allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the conjunctions are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*res`, `resDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.39 class `AnyDim`

A *dimension any schedule* is a communication schedule for computing the logical disjunction of the elements of a distributed array of boolean values along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension any schedule is described by a collective object with local components of class `AnyDim`. The public interface of the `AnyDim` class is

```
class DAD ;

class AnyDim {
public :
  AnyDim(const DAD* res, const DAD* src, const int dim) ;

  void execute(int* resDat, int* srcDat) ;
private :
  ...
} ;
```

5.39.1 Constructor

`AnyDim(const DAD* res, const DAD* src, const int dim)`

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `int`. The reduction occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.39.2 Method

`execute(int* resDat, int* srcDat)`

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is true if

$$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is true (non-zero) for *any* allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the disjunctions are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*res`, `resDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.40 class `CountDim`

A *dimension count schedule* is a communication schedule for counting the number of true elements of a distributed array of boolean values along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension count schedule is described by a collective object with local components of class `CountDim`. The public interface of the `CountDim` class is

```
class DAD ;

class CountDim {
public :
  CountDim(const DAD* res, const DAD* src, const int dim) ;

  void execute(int* resDat, int* srcDat) ;
private :
  ...
} ;
```

5.40.1 Constructor

`CountDim(const DAD* res, const DAD* src, const int dim)`

The source array is described by the DAD `*src` and the result array is described by the DAD `*res`. They will both have elements of type `int`. The reduction occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The result array must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result array must be accessible (see section 4.6).

Argument persistence: The `res` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.40.2 Method

`execute(int* resDat, int* srcDat)`

Arguments are the base addresses for local segments of the result and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the number of true (non-zero) elements

$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

over allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular contributions to the count are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. The pairs `*res`, `resDat` and `*src`, `srcDat` must define non-overlapping arrays.

5.41 class `MaxlocDim`

A *dimension maximum location schedule* is a communication schedule for searching for the largest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension maximum location schedule is described by a collective object with local components of class `MaxlocDim`. The public interface of the `MaxlocDim` class is

```
class DAD ;

template<class T>
class MaxlocDim {
public :
  MaxlocDim(const DAD* val, const DAD* pos,
            const DAD* src, const int dim) ;

  void execute(T* valDat, int* posDat, T* srcDat) ;
private :
  ...
} ;
```

5.41.1 Constructor

```
MaxlocDim(const DAD* val, const DAD* pos,
          const DAD* src, const int dim)
```

The source array is described by the DAD `*src` and the array of maximum values is described by the DAD `*val`. They will both have elements of type `T`. The array of maximum locations is described by the DAD `*pos`. The search occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The shape of the two result arrays (`*val` and `*pos`) must be must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The two result arrays (`*val` and `*pos`) must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result arrays must be accessible (see section 4.6).

Argument persistence: The `val`, `loc` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.41.2 Method

```
execute(T* valDat, int* posDat, T* srcDat)
```

Arguments are the base addresses for local segments of the result value, result position, and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

`val` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the maximum value of

`src` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

across allowed values of $x_{\texttt{dim}}$. The value of

`pos` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the smallest $x_{\texttt{dim}}$ value at which this maximum occurs.

As implied by the replicated alignment of the result arrays, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values of particular elements for comparision are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. Neither of the arrays defined by the pairs `*val`, `valDat` or `*pos`, `posDat` may have any overlap with the array defined by the pair `*src`, `srcDat`.

5.42 class `MaxlocDimMsk`

A *masked dimension maximum location schedule* is a communication schedule for searching, under the control of a mask, for the largest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A masked dimension maximum location schedule is described by a collective object with local components of class `MaxlocDimMsk`. The public interface of the `MaxlocDimMsk` class is

```
class DAD ;

template<class T>
class MaxlocDimMsk {
public :
  MaxlocDimMsk(const DAD* val, const DAD* pos,
               const DAD* src, const int dim, const DAD* msk) ;

  void execute(T* valDat, int* posDat, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.42.1 Constructor

```
MaxlocDimMsk(const DAD* val, const DAD* pos,
             const DAD* src, const int dim,
             const DAD* msk)
```

The source array is described by the DAD `*src` and the array of maximum values is described by the DAD `*val`. They will both have elements of type `T`. The array of maximum locations is described by the DAD `*pos`. The search occurs in dimension `dim`. The mask array is described by the DAD `*msk`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostneg` must be overloaded with the entry point

```
void mostneg(T* res) ;
```

which sets `*res` to the smallest allowed value of type `T`. The Adlib header files define `mostneg` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The shape of the two result arrays (`*val` and `*pos`) must be must be obtained from the shape of the source array by deleting dimension `dim`. The mask array must be the same shape as the source array (see section 4.4).

Alignment restrictions: The two result arrays (`*val` and `*pos`) must be aligned to the source array, *with replicated alignment in dimension* `dim`. The mask array must be value-aligned with the source array (see section 4.5).

Accessibility restrictions: The source and mask arrays must be value-accessible and the result arrays must be accessible (see section 4.6).

Argument persistence: The `val`, `loc`, `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.42.2 Method

`execute(T* valDat, int* posDat, T* srcDat, int* mskDat)`

Arguments are the base addresses for local segments of the result value, result position, source, and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\texttt{val}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the maximum value of

$$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

for which

$$\texttt{msk}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq 0$$

over the allowed values of $x_{\texttt{dim}}$. The value of

$$\texttt{pos}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the smallest $x_{\mathtt{dim}}$ value at which this maximum occurs.

As implied by the replicated alignment of the result arrays, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular elements for comparision are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. Neither of the arrays defined by the pairs `*val`, `valDat` or `*pos`, `posDat` may have any overlap with either of the arrays defined by the pairs `*src`, `srcDat` and `*msk`, `mskDat`

5.43 class `MinlocDim`

A *dimension minimum location schedule* is a communication schedule for searching for the smallest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A dimension minimum location schedule is described by a collective object with local components of class `MinlocDim`. The public interface of the `MinlocDim` class is

```
class DAD ;

template<class T>
class MinlocDim {
public :
  MinlocDim(const DAD* val, const DAD* pos,
            const DAD* src, const int dim) ;

  void execute(T* valDat, int* posDat, T* srcDat) ;
private :
  ...
} ;
```

5.43.1 Constructor

```
MinlocDim(const DAD* val, const DAD* pos,
          const DAD* src, const int dim)
```

The source array is described by the DAD `*src` and the array of minimum values is described by the DAD `*val`. They will both have elements of type `T`. The array of minimum locations is described by the DAD `*pos`. The search occurs in dimension `dim`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostpos` must be overloaded with the entry point

```
void mostpos(T* res) ;
```

which sets `*res` to the largest allowed value of type `T`. The Adlib header files define `mostpos` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The shape of the two result arrays (`*val` and `*pos`) must be must be obtained from the shape of the source array by deleting dimension `dim` (see section 4.4).

Alignment restrictions: The two result arrays (`*val` and `*pos`) must be aligned to the source array, *with replicated alignment in dimension* `dim` (see section 4.5).

Accessibility restrictions: The source array must be value-accessible and the result arrays must be accessible (see section 4.6).

Argument persistence: The `val`, `loc` and `src` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.43.2 Method

`execute(T* valDat, int* posDat, T* srcDat)`

Arguments are the base addresses for local segments of the result value, result position, and source arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

`val` $[x_0, \ldots, x_{\mathtt{dim-1}}, x_{\mathtt{dim+1}}, \ldots, x_{R-1}]$

is the minimum value of

`src` $[x_0, \ldots, x_{\mathtt{dim-1}}, x_{\mathtt{dim}}, x_{\mathtt{dim+1}}, \ldots, x_{R-1}]$

across allowed values of $x_{\mathtt{dim}}$. The value of

`pos` $[x_0, \ldots, x_{\mathtt{dim-1}}, x_{\mathtt{dim+1}}, \ldots, x_{R-1}]$

is the smallest $x_{\mathtt{dim}}$ value at which this minimum occurs.

As implied by the replicated alignment of the result arrays, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values of particular elements for comparision are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. Neither of the arrays defined by the pairs `*val`, `valDat` or `*pos`, `posDat` may have any overlap with the array defined by the pair `*src`, `srcDat`.

5.44 class MinlocDimMsk

A *masked dimension minimum location schedule* is a communication schedule for searching, under the control of a mask, for the smallest elements of a distributed array along one of its dimensions, yielding a reduced array with rank one less than the source.

A masked dimension minimum location schedule is described by a collective object with local components of class `MinlocDimMsk`. The public interface of the `MinlocDimMsk` class is

```
class DAD ;

template<class T>
class MinlocDimMsk {
public :
  MinlocDimMsk(const DAD* val, const DAD* pos,
               const DAD* src, const int dim, const DAD* msk) ;

  void execute(T* valDat, int* posDat, T* srcDat, int* mskDat) ;
private :
  ...
} ;
```

5.44.1 Constructor

```
MinlocDimMsk(const DAD* val, const DAD* pos,
             const DAD* src, const int dim,
             const DAD* msk)
```

The source array is described by the DAD `*src` and the array of minimum values is described by the DAD `*val`. They will both have elements of type `T`. The array of minimum locations is described by the DAD `*pos`. The search occurs in dimension `dim`. The mask array is described by the DAD `*msk`.

Value restrictions: The value of `dim` must be in the range $0, \ldots, R-1$ where R is the rank of the source array.

Type restrictions: `T` should be a POD type. Conventional binary comparision operators, `>` and `<`, must be defined for objects of type `T`, and a function `mostpos` must be overloaded with the entry point

```
void mostpos(T* res) ;
```

which sets `*res` to the largest allowed value of type `T`. The Adlib header files define `mostpos` for all the standard arithmetic types of C++, so these types satisfy all the requirements for `T`.

Shape restrictions: The shape of the two result arrays (`*val` and `*pos`) must be must be obtained from the shape of the source array by deleting dimension `dim`. The mask array must be the same shape as the source array (see section 4.4).

Alignment restrictions: The two result arrays (`*val` and `*pos`) must be aligned to the source array, *with replicated alignment in dimension* `dim`. The mask array must be value-aligned with the source array (see section 4.5).

Accessibility restrictions: The source and mask arrays must be value-accessible and the result arrays must be accessible (see section 4.6).

Argument persistence: The `val`, `loc`, `src` and `msk` arguments are stored in the schedule as references. The associated objects must persist for the lifetime of the constructed schedule.

5.44.2 Method

```
execute(T* valDat, int* posDat, T* srcDat, int* mskDat)
```

Arguments are the base addresses for local segments of the result value, result position, source, and mask arrays. They should point to vectors of the locally held elements.

Effect: On exit, the value of

$$\texttt{val}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the minimum value of

$$\texttt{src}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

for which

$$\texttt{msk}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq 0$$

over the allowed values of $x_{\texttt{dim}}$. The value of

$$\texttt{pos}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the smallest $x_{\texttt{dim}}$ value at which this minimum occurs.

As implied by the replicated alignment of the result arrays, the results are broadcast in the process dimension associated with dimension `dim` of the source array.

Replicated data: If the source array has replicated mapping, values for particular elements for comparision are generally taken from the nearest copy of the source element. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

Overlap restrictions: In-place updates are not allowed. Neither of the arrays defined by the pairs `*val`, `valDat` or `*pos`, `posDat` may have any overlap with either of the arrays defined by the pairs `*src`, `srcDat` and `*msk`, `mskDat`

Chapter 6

The ad++ interface

Chapter 7

Distributed loops

In this chapter we discuss various ways to use the Adlib run-time technology in translation of distributed loops—loops whose ranges is partitioned across the active process group. Typically such loops are used to access and modify the data in distributed arrays. For definiteness, we work in the context of the *ad++* interface. The techniques can be adapted to other interfaces to the kernel library.

In ad++, the general *overall* construct is a distributed, parallel loop. It is parametrized by an `Index` object which maintains local *loop state*. If `x` is a range the *overall* construct has the syntax.

```
Index i(x) ;
OVERALL(i) {
  ...
} ALLOVER(i) ;
```

If `x` has extent `N`, this construct can be compared to the sequential loop

```
int i ;
for(i = 0 ; i < N ; i++) {
  ...
}
```

The difference is that in the *overall* construct the `N` instances of the body of the loop will be partitioned across the set of active processes, following the mapping of `x`.

The `Index` class is a subclass of `Location`. Within an *overall* construct parametrized by an `Index i`, the `Subcript` component of `i` is set to the local subscript for the current iteration. So `i` can be used as an array subscript, as in

```
Array1<float> c(x) ;

Index i(x) ;
OVERALL(i) {
  c(i) = ...
} ALLOVER(i) ;
```

The general *overal* construct has an effect on the active process group as described in section 2.7. If a construct parametrized by `i` appears in the context of an active process group `p`, the body of the construct executes in the context of an active process group `p / i` (recall that `Index` is a subclass of `Location` which is in turn is a subclass of `Coord`, so this expression is well-formed.) The parent range of `i` must be distributed over a dimension of `p`.

The conversion from an `Index i` to an integer has a well-defined value inside an *overall* construct parametrized by `i`. It returns the global subscript for the current iteration.

Combining these features, we can give a more complete example

```
Array2<float> a(x, y) ;
Array1<float> b(y) ;
...
Index i(x), j(y) ;
OVERALL(i) {
  OVERALL(j) {
    a(i, j) = 2 * b(j) + i ;
  } ALLOVER(j) ;
} ALLOVER(i) ;
```

To each element of `a`, this assigns an expression computed from the aligned value of `b` and the global subscript of `x` (obtained through the conversion `(int) i`)[1]. All data accesses through legal subscripting operations are *local*. If a non-local array element was required, it would take a specific call to a member of the communication library to access it.

The remainder of this chapter discusses several schemes for translating the distributed loop. The first scheme uses the Adlib `Index` class directly. The "translation" is the trivial one, using only the C macro preprocessor to replace the `OVERALL` and `ALLOVER` "keywords". The second scheme uses another auxilliary (iterator) class from the library—`LocBlocksIndex`. The

[1] Sometimes it is necessary to know the global subscript of the iteration relative to some parent range of `x` (some range of which `x` is a subrange). This value be expressed in terms of `(int) i` using the alignment parameters of `x` relative to the parent. Earlier specifications of the library provided special syntax and computational support for obtaining parent global subscripts, but this now seems like an unnecessary complication.

translation is still relatively straightforward and has the advantage of being independent of the level of the parametric range. The `LocBlocksIndex` mechanism is quite efficient, and is used extensively in the implementation of the Adlib communication library. Finally we describe a scheme which works directly in terms of the members of `Range` class, without introducing any auxilliary iterator class.

7.1 class Index

An iterator class, maintaining loop state for simple enumerations of the elements of a range.

The `Index` class is derived from the `LocBlocksIndex` class (see section 7.2) which is in turn derived from `Location`. These classes are used extensively in Adlib code that has to deal with ranges whose level is not known in advance. Their use can be avoided if the level of the ranges is known at compile time, using code transformations described in section 7.3.

The public interface of the `Index` class is

```
class LocBlocksIndex ;

class Index : public LocBlocksIndex {
public :
  Index(Range x) ;

  operator int() const ;

  void begin() ;
  void next() ;

  Location operator<<(const int shift) ;
  Location operator>>(const int shift) ;
} ;
```

7.1.1 Constructor

`Index(Range x)`

Create an index object for enumerating the elements of range `x`.

7.1.2 Methods

`begin()`

Used to implement the general *overall construct*—a distributed loop. Begins an enumeration of the elements of the range mapped to the local process. On exit from this member, the `Location` component of the index represents the first element of the local segment of the range[2].

If `i` is the loop index, the idiom for the loop is

[2]Here "first element" simply means first in the enumeration. It *does not* necessarily mean the local element with the smallest global subscript.

```
for(i.begin() ; i.test() ; i.next())
  S
```

The member `test()` is inherited from `LocBlocksIndex` (see section 7.2).

`next()`

Move to next element in enumeration.

7.1.3 Translation of *overall* construct by macro expansion

The most naive translation of the *overall* construct is by using the standard C preprocessor to expand the macros

```
#define OVERALL(i) for(apgStack.push(&apg), apg.restrict(i.dim), \
                       i.begin() ; i.test() ; i.next())

#define ALLOVER(i) apgStack.pop(&apg)
```

Effectively, the loop

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

becomes

```
apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.begin() ; i.test() ; i.next()) {
  ...
}
apgStack.pop(&apg) ;
```

We moved the `apg` manipulations (see section 2.7) outside the *for* construct to improve readability. The `begin()`, `test()` and `next()` members of `Index` are used in the *for* loop to enumerate the local elements of the range.

This translation is trivial to implement, but it is very inefficient. First we have the overhead of the calls to the iterator members of `Index` in every iteration. Secondly (probably even more seriously) every array or range subscripting operation in the body of the loop involves calls to member functions. The overhead of all these library calls can easily downgrade performance by an order of magnitude or more relative to a comparable sequential *for* loop.

7.2 class `LocBlocksIndex`

An iterator class, maintaining loop state for enumerations of the locally held blocks of a range.

The public interface of the `LocBlocksIndex` class is

```
class Block ;

class LocBlocksIndex : public Location, public Block {
public :
  LocBlocksIndex(Range _x) ;

  void beginLocBlk() ;
  int test() ;
  void nextLocBlk() ;
} ;
```

7.2.1 Constructor

`LocBlocksIndex(Range x)`

Create an index object for enumerating the local blocks of range `x`.

7.2.2 Methods

`beginLocBlk()`

Begins an enumeration of the blocks of the range mapped to the local process. On exit from this member, the `Block` componenent of the `Index` contains the parameters of the first block of the locally held segment of the range, and the `Location` component represents the first element of that block. "First block" simply means first in the enumeration.

If `i` is the loop index, the idiom for the loop is

```
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk())
  S
```

`nextLocBlk()`

Move to next block in enumeration.

7.2.3 Translation of *overall* construct using `LocBlocksIndex`

The class `LocBlocksIndex` is a superclass of `Index` that provides members to enumerate local *blocks* of a range, rather than individual local elements. Library functions return base addresses of the array sections associated with these blocks. Elements within the block are then enumerated with a simple, efficient *for* loop, computing offsets from base addresses using linear expressions. The performance-critical inner loops can be compiled with high efficiency. If the block size is large enough, most of the cost of the library calls is amortized.

`LocBlocksIndex` is a subclass of `Index` *and* of `Block`. The definition of `Block` is

```
struct Block {
  int count ;

  int bas ;
  int stp ;

  int sub_bas ;
  int sub_stp ;
} ;
```

It is a simple record parametrizing a block in terms of the size of the block, and global and block subscript bases and steps. The outer level in the translation of a *overall* construct now looks something like

```
LocBlocksIndex i(x) ;

apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
  ... deal with block 'i'
}
apgStack.pop(&apg) ;
```

In this translation scheme we have an outer loop enumerating the locally held blocks of the range. In general this loop is needed because Adlib supports higher-level distribution formats like *block-cyclic*. These allow multiple blocks of a single range to reside on the same processor. In a context where it is known in advance that the local process holds a single block (for example, if the range involved is *simple*), the translation scheme given in the next section allows further optimizations.

In general code to "deal with block `i`", takes the form

```
  ... precompute some bases and increments for block

  for(int l = 0 ; l < i.count ; l++) {
    ...
  }
```

Before filling the details of this code, we need to put the source code into a normalized form. The body of a *overall* construct parametrized by an index `i` may use `i` in several contexts:

1. As a subscript in a local subscripting operation.
2. In a conversion to `int` used to obtain the global subscript of the current iteration relative to the index range.
3. As a scalar subscript in a section subscripting operation.
4. In a group restriction operation of the form `p / i`, where `p` is some group.

As described in chapter 4, subscripting operations on arrays can be replaced with lower-level operations on the `Stride` and `Group` classes. For example, if `a` is an array of `float`, the reference

```
... a(i, j) ...
```

can be replaced with

```
float* a_dat  = a.dat() ;

... a_dat [a.str(0).offset(i) + a.str(1).offset(j)] ...
```

Of course the inquiries `dat()` and `str()` can be lifted outside any loop. Similarly, the section construction

```
... a.sect(i, y) ...
```

can be replaced with

```
float* a_dat  = a.dat() ;

... Section1<float>(y, a.grp() / i,
                    a.str(1), a_dat + a.str(0).offset(i)) ...
```

By applying these transformations, and replacing any conversions to `int` with explicit casts, the set of uses of `i` inside the loop can be reduced to three cases

1. As an argument of `Stride :: offset`.

2. In cast to `int`, yielding the global subscript.

3. In a group restriction operation of the form `p / i`.

We will deal with these cases in turn. *[Replace follwing enumeration and associated figures with straight descriptive text.]*

Offset computation. Within a particular block the expression `u.offset(i)` can be rewritten as a linear function of the inner induction variable. The base for this function is given by the value of `offset` for the first `Location` of the range, which is the `Location` component of `i`. Its increment is returned by the member `Stride :: step`. Figure 7.1 illustrates the translation. The call to `offset` has been removed from the inner loop. A good compiler will generate very efficient code for the linear expression that replaces it.

Global subscript computation is slightly simpler. The base and increment for this expression are already contained in the `Block` component of `i`, computed by its iterator members. Figure 7.2 illustrates the translation.

Group restriction is translated trivially, as illustrated in figure 7.3.

Now, consider this example from the preamble to this chapter:

```
Array2<float> a(x, y) ;
Array1<float> b(y) ;
...
Index i(x), j(y) ;
OVERALL(i) {
  OVERALL(j) {
    a(i, j) = 2 * b(j) + i ;
  } ALLOVER(j) ;
} ALLOVER(i) ;
```

It can be normalized to the form

```
Array2<float> a(x, y) ;
Array1<float> b(y) ;
...
float* a_dat = a.dat() ;

float* b_dat = b.dat() ;

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

SOURCE:

```
Range x ;
Stride u ;

Index i(x) ;
OVERALL(i) {
  ... u.offset(i) ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;
Stride u ;

LocBlocksIndex i(x) ;

apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
  int u_bas = u.offset(i) ;
  int u_stp = u.step(i) ;

  for(int l = 0 ; l < i.count ; l++) {
    ... u_bas + u_stp * l ...
  }
}
apgStack.pop(&apg) ;
```

Figure 7.1: Translation of offset computation.

SOURCE:

```
Range x ;

Index i(x) ;
OVERALL(i) {
  ... (int) i ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;

LocBlocksIndex i(x) ;

apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
  for(int l = 0 ; l < i.count ; l++) {
    ... i.bas + i.stp * l ...
  }
}
apgStack.pop(&apg) ;
```

Figure 7.2: Translation of global subscript computation.

SOURCE:

```
Range x ;

Index i(x) ;
OVERALL(i) {
  ... p / i ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;

LocBlocksIndex i(x) ;

apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
  for(int l = 0 ; l < i.count ; l++) {
    ... p / i ...
  }
}
apgStack.pop(&apg) ;
```

Figure 7.3: Translation for computation of group restriction.

```
    OVERALL(j) {
      a_dat [a.str(0).offset(i) + a.str(1).offset(j)] =
          b_dat [b.str(0).offset(j)] + (int) i ;
    } ALLOVER(j) ;
  } ALLOVER(i) ;
```

A translation of the loop nest is given in figure 7.4.

Note that in this example the manipulations of **apg** could have been omitted, because there are no collective operations inside the loop that depend on the state of **apg**.

As a final straightforward optimization, when we have perfectly nested *overall* constructs, the loop nesting can be changed to put all intra-block loops innermost. In that case the inner loops become

```
for(int l = 0 ; l < i.count ; l++) {
  for(int m = 0 ; m < j.count ; m++) {
    a_dat [a_off0_bas + a_off0_stp * l +
           a_off1_bas + a_off1_stp * m] =
              b_dat [b_off0_bas + b_off0_stp * m] +
              i.bas + i.stp * l ;
  }
}
```

All subscript expressions are linear in the loop induction variables, and we expect very good code generation from these loops.

The translation scheme described in this section is summarized in figure 7.5.

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

apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
  int a_off0_bas = a.str(0).offset(i) ;
  int a_off0_stp = a.str(0).step(i) ;

  for(int l = 0 ; l < i.count ; l++) {
    apgStack.push(&apg) ;
    apg.restrict(j.dim) ;
    for(j.beginLocBlk() ; j.test() ; j.nextLocBlk()) {
      int a_off1_bas = a.str(1).offset(j) ;
      int a_off1_stp = a.str(1).step(j) ;

      int b_off0_bas = b.str(0).offset(j) ;
      int b_off0_stp = b.str(0).step(j) ;

      for(int m = 0 ; m < j.count ; m++) {
        a_dat [a_off0_bas + a_off0_stp * l +
               a_off1_bas + a_off1_stp * m] =
                  b_dat [b_off0_bas + b_off0_stp * m] +
                  i.bas + i.stp * l ;
      }
    }
    apgStack.pop(&apg) ;
  }
}
apgStack.pop(&apg) ;
```

Figure 7.4: Translation of example.

SOURCE:

```
Range x ;
Stride u ;
Group p ;

Index i(x) ;
OVERALL(i) {
  ... u.offset(i) ...
      (int) i ...
      p / i ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;
Stride u ;
Group p ;

LocBlocksIndex i(x) ;

apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
  int u_bas = u.offset(i) ;
  int u_stp = u.step(i) ;

  for(int l = 0 ; l < i.count ; l++) {
    ... u_bas + u_stp * l ...
        i.bas + i.stp * l ...
        p / i ...
  }
}
apgStack.pop(&apg) ;
```

Figure 7.5: Summary of `LocBlocksIndex`-based translation scheme for *overall* construct.

7.3 Translation using the kernel range

Internally, all the iterator classes depend on the `block` member of the kernel range. In this section we show how to use this member directly for translation of *overall* constructs. The scheme given here effectively inlines the iterator members used in the previous translations. It also inlines the `offset` member of the `Stride` class, in terms of lower level `disp` and `disp_step` members.

To apply the translation scheme described in this section we need some compile-time knowledge about the level of the range involved. The scheme works recursively by expanding a *overall* construct in the source program in terms of a *overall* construct for a range one level lower. The procedure can (if desired) be continued recursively. Sooner or later we get down to a *overall* construct for a level-0 range. This base case has a different, simpler translation. For this scheme to be effective we must at least know in advance if the original parametric range has level 0 (ie, it already represents the base case). Ideally we should know the exact level, as a compile-time constant.

Before applying the translation proper, the input program should be normalized as discussed in the previous section. Rather than discuss translation of the three uses of `Index` individually, we will go straight to the combined summary form, in the style of figure 7.5. For a parametric range of level greater than zero, the translation summary is given in figure 7.6.

The outer loop is a *overall* construct parametrized by `x.ker()`. The `block` and `block_step` members `x` of initializes variables describing the block selected by the current value of the kernel subscript. These variables correspond exactly with the fields of the `Block` component of `LocBlocksIndex` (and, of course, this is exactly how they are computed in the implementation of that class).

The `offset` operation is expanded in terms of `disp` and `disp_step` operations, and an `offset` for a kernel range. Typically the transformation may be applied recursively as necessary to eliminate the `offset` function altogether.

The rest of the translation closely follows that of the previous section.

If the parametric range has level 0 (it is a process dimension, or a subrange of a process dimension) the summary is given in figure 7.7.

One stage of recursion applied to the example of the previous section gives the translation in figure 7.8. If ranges `x` and `y` both have level 1 application of the rule for translating level 0 constructs then gives the translation in figure 7.9. Finally, figure 7.10 gives an optimized form on the assumption that the original ranges were *simple*. Recall that a simple range is a level 1 range whose kernel is primitive—ie the kernel is the whole of a process dimension. We can remove the `local` conditional because every active

SOURCE:

```
Range x ;
Stride u ;
Group p ;

Index i(x) ;
OVERALL(i) {
  ... u.offset(i) ...
      (int) i ...
      p / i ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;
Stride u ;
Group p ;

Index i1(x.ker()) ;
OVERALL(i1) {
  int i_count, i_sub_bas, i_sub_stp ;
  int i_bas = x.block((int) i1, &i_sub_bas, &i_count) ;
  int i_stp = x.block_step(&i_sub_stp) ;

  int u_bas = u.disp(i_sub_bas) + u.ker().offset(i1) ;
  int u_stp = u.disp_step(i_sub_stp) ;

  for(int l = 0 ; l < i_count ; l++) {
    ... u_bas + u_stp * l ...
        i_bas + i_stp * l ...
        p / i1 ...
  }
} ALLOVER(i1) ;
```

Figure 7.6: Summary of recursive translation scheme for *overall* construct with level greater than 0.

SOURCE:

```
Range x ;
Stride u ;
Group p ;

Index i(x) ;
OVERALL(i) {
  ... u.offset(i) ...
      (int) i ...
      p / i ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;
Stride u ;
Group p ;

Dimension d = x.dim() ;

apgStack.push(&apg) ;
apg.restrict(d) ;

int glb ;
if(x.local(&glb)) {
  ... 0 ...
      glb ...
      p.restrict(d) ...
}

apgStack.pop(&apg) ;
```

Figure 7.7: Summary of translation for *overal* construct of level 0.

process must contain an element of the kernel, and replace the value `i1` initialized by `local` with `d.crd()`. Three other optimizations do not depend on the assumption of simplicity: additions of 0 are constant-folded away, `x.ker().dim()` is replaced with `x.dim()`, and the `apg` manipulations are removed, because the loop body contains no calls to collective operations.

Because simple ranges are an important case, figure 7.11 summarizes the translation of the overal construct for simple parametric ranges.

```
Index i1(x.ker()), j1(y.ker()) ;
OVERALL(i1) {
  int i_count, i_sub_bas, i_sub_stp ;
  int i_bas = x.block((int) i1, &i_sub_bas, &i_count) ;
  int i_stp = x.block_step(&i_sub_stp) ;

  int a_off0_bas = a.str(0).disp(i_sub_bas) + a.str(0).ker().offset(i1) ;
  int a_off0_stp = a.str(0).disp_step(i_sub_stp) ;

  for(int l = 0 ; l < i_count ; l++) {
    OVERALL(j1) {
      int j_count, j_sub_bas, j_sub_stp ;
      y.block((int) j1, &j_sub_bas, &j_count) ;
      y.block_step(&j_sub_stp) ;

      int a_off1_bas = a.str(1).disp(j_sub_bas) + a.str(0).ker().offset(j1) ;
      int a_off1_stp = a.str(1).disp_step(j_sub_stp) ;

      int b_off0_bas = b.str(0).disp(i_sub_bas) + b.str(0).ker().offset(i1) ;
      int b_off0_stp = b.str(0).disp_step(i_sub_stp) ;

      for(int m = 0 ; m < j_count ; m++) {
        a_dat [a_off0_bas + a_off0_stp * l +
               a_off1_bas + a_off1_stp * m] =
                  b_dat [b_off0_bas + b_off0_stp * m] +
                  i_bas + i_stp * l ;
      }
    } ALLOVER(j1) ;
  }
} ALLOVER(i1) ;
```

Figure 7.8: Translation of example. Pass 1, assuming ranges **x** and **y** have level greater than zero.

```
Dimension d = x.ker().dim() ;
apgStack.push(&apg) ;
apg.restrict(d) ;

int i1 ;
if(x.ker().local(&i1) {
  int i_count, i_sub_bas, i_sub_stp ;
  int i_bas = x.block(i1, &i_sub_bas, &i_count) ;
  int i_stp = x.block_step(&i_sub_stp) ;

  int a_off0_bas = a.str(0).disp(i_sub_bas) + 0 ;
  int a_off0_stp = a.str(0).disp_step(i_sub_stp) ;

  for(int l = 0 ; l < i_count ; l++) {
    Dimension e = y.ker().dim() ;
    apgStack.push(&apg) ;
    apg.restrict(e) ;

    int j1 ;
    if(y.ker().local(&j1) {
      int j_count, j_sub_bas, j_sub_stp ;
      y.block(j1, &j_sub_bas, &j_count) ;
      y.block_step(&j_sub_stp) ;

      int a_off1_bas = a.str(1).disp(j_sub_bas) + 0 ;
      int a_off1_stp = a.str(1).disp_step(j_sub_stp) ;

      int b_off0_bas = b.str(0).disp(i_sub_bas) + 0 ;
      int b_off0_stp = b.str(0).disp_step(i_sub_stp) ;

      for(int m = 0 ; m < j.count ; m++) {
        a_dat [a_off0_bas + a_off0_stp * l +
               a_off1_bas + a_off1_stp * m] =
                  b_dat [b_off0_bas + b_off0_stp * m] +
                  i_bas + i_stp * l ;
      }
    }
    apgStack.pop(&apg) ;
  }
}
apgStack.pop(&apg) ;
```

Figure 7.9: Translation of example. Pass 2, assuming ranges **x** and **y** have level 1, so their kernels are level 0.

```
Dimension d = x.dim() ;

int i_count, i_sub_bas, i_sub_stp ;
int i_bas = x.block(d.crd(), &i_sub_bas, &i_count) ;
int i_stp = x.block_step(&i_sub_stp) ;

int a_off0_bas = a.str(0).disp(i_sub_bas) ;
int a_off0_stp = a.str(0).disp_step(i_sub_stp) ;

Dimension e = y.dim() ;

int j_count, j_sub_bas, j_sub_stp ;
y.block(e.crd(), &j_sub_bas, &j_count) ;
y.block_step(&j_sub_stp) ;

int a_off1_bas = a.str(1).disp(j_sub_bas) ;
int a_off1_stp = a.str(1).disp_step(j_sub_stp) ;

int b_off0_bas = b.str(0).disp(i_sub_bas) ;
int b_off0_stp = b.str(0).disp_step(i_sub_stp) ;

for(int l = 0 ; l < i_count ; l++) {
  for(int m = 0 ; m < j_count ; m++) {
    a_dat [a_off0_bas + a_off0_stp * l + a_off1_bas + a_off1_stp * m] =
              b_dat [b_off0_bas + b_off0_stp * m] +
              i_bas + i_stp * l ;
  }
}
```

Figure 7.10: Translation of example. Optimizations assuming ranges `x` and `y` are simple.

SOURCE:

```
Range x ;
Stride u ;
Group p ;

Index i(x) ;
OVERALL(i) {
  ... u.offset(i) ...
      (int) i ...
      p / i ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;
Stride u ;
Group p ;

Dimension d = x.dim() ;
apgStack.push(&apg) ;
apg.restrict(d) ;

int i_count, i_sub_bas, i_sub_stp ;
int i_bas = x.block(d.crd(), &i_sub_bas, &i_count) ;
int i_stp = x.block_step(&i_sub_stp) ;

int u_bas = u.disp(i_sub_bas) ;
int u_stp = u.disp_step(i_sub_stp) ;

for(int l = 0 ; l < i_count ; l++) {
  ... u_bas + u_stp * l ...
      i_bas + i_stp * l ...
      p.restrict(d) ...
}

apgStack.pop(&apg) ;
```

Figure 7.11: Summary of translation for *overal* construct with simple `x`.

7.4 Translation using the subkernel range

[Make this a subsection of the previous section.]

Occasionally an effective optimization to the translation scheme of the last section is to use the *subkernel* instead of the kernel for outer loops. If the parametric range is a small subrange of its parent template, covering only a small part of its kernel, using the subkernel may avoid enumerating of many empty blocks. See figures 3.1 and 3.2.

For a parametric range of level greater than zero, the translation summary is given in figure 7.14. The kernel index `j` is now parametrized by `x.subker()` rather than `x.ker()`, and a linear computation is needed to obtain the correct `ker_glb` argument of `block` in terms of the subkernel global subscript.

In practise this optimization is more important in the global block enumerations of section 7.6 than in distributed loops. As well as providing a useful optimization to this enumeration, the subkernel imposes an ordering on block enumeration which is important in certain communication operations (specifically, `remap`). If the alignment stride of the parent range is negative, the result for `ker_str` is also negative.

For completeness, we give a translation of the overall construct using the subkernel, although this is probably not a useful translation scheme.

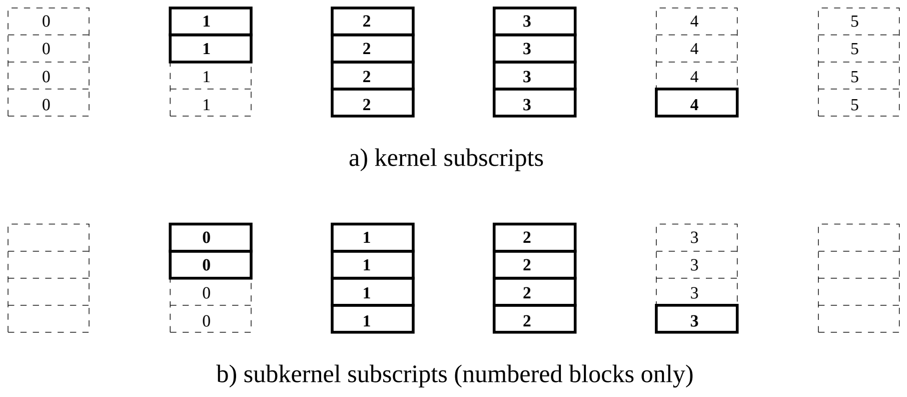

Figure 7.12: Possible definition of the subkernel for the range of figure 3.1.

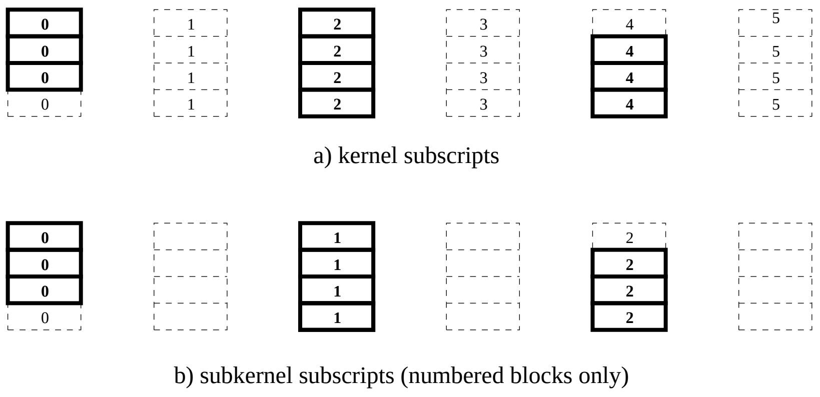

Figure 7.13: Possible definition of the subkernel for the range of figure 3.2.

SOURCE:

```
Range x ;
Stride u ;
Group p ;

Index i(x) ;
OVERALL(i) {
  ... u.offset(i) ...
      (int) i ...
      p / i ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;
Stride u ;
Group p ;

int ker_bas, ker_str ;
Index i1(x.subker(&ker_bas, &ker_str)) ;
OVERALL(i1) {
  int i_count, i_sub_bas, i_sub_stp ;
  int i_bas = x.block(ker_bas + ker_str * i1, &i_sub_bas, &i_count) ;
  int i_stp = x.block_step(&i_sub_stp) ;

  int u_bas = u.disp(i_sub_bas) + u.ker().offset(i1) ;
  int u_stp = u.disp_step(i_sub_stp) ;

  for(int l = 0 ; l < i_count ; l++) {
    ... u_bas + u_stp * l ...
        i_bas + i_stp * l ...
        p / i1 ...
  }
} ALLOVER(i1) ;
```

Figure 7.14: Summary of subkernel-based translation scheme for *overall* construct with level greater than 0.

7.5 Access to ghost regions

[Make this a subsection of the `LocBlocksIndex` *section, and add related material to* `Index` *section.]*

If an array appearing in a *overall* construct has ghost regions, the subscripts of the array may be shifted by some amount without violating the restriction that all array element accesses are local. In ad++ this can be expressed by an idiom like

```
Array1<float> b(x), c(x) ;

Index i(x) ;
OVERALL(i) {
  b(i) = c(x(i + 1)) + c(x(i - 1)) ;
} ALLOVER(i) ;
```

This assumes that the array `c` has ghost regions of extent one or more on both sides of its "physical" segments. We emphasise that using the `operator()(const int)` member of `Range` to compute local subscripts within a *overall* construct is very likely to lead to errors in general. However the particular usage illustrated here is very useful in "stencil" computations. ad++ provides a special syntax using the shift operators:

```
Index i(x) ;
OVERALL(i) {
  b(i) = c(i >> 1) + c(i << 1) ;
} ALLOVER(i) ;
```

For definiteness, the translation scheme in figure 7.15 uses the `blocksIndex`-based style of translation, but the same techniques can be carried over directly to the recursive style.

SOURCE:

```
Range x ;
Stride u ;

Index i(x) ;
OVERALL(i) {
  ... u.offset(x(i + s)) ...
} ALLOVER(i) ;
```

OUTPUT:

```
Range x ;
Stride u ;

LocBlocksIndex i(x) ;

apgStack.push(&apg) ;
apg.restrict(i.dim) ;
for(i.beginLocBlk() ; i.test() ; i.nextLocBlk()) {
  int u_stp = u.step(i) ;
  int u_bas = u.offset(i) + s * u_stp ;

  for(int l = 0 ; l < i.count ; l++) {
    ... u_bas + u_stp * l ...
  }
}
apgStack.pop(&apg) ;
```

Figure 7.15: Translation of shifted offset computation.

7.6 class `AllBlocksIndex`

An iterator class, maintaining loop state for enumerations of all the blocks of a range.

The public interface of the `AllBlocksIndex` class is

```
class AllBlocksIndex : public LocBlocksIndex {
public :
  AllBlocksIndex(Range _x) ;

  void beginAllBlk() ;
  void nextAllBlk() ;
} ;
```

7.6.1 Constructor

`AllBlocksIndex(Range x)`

Create an index object for enumerating all blocks of range `x`.

7.6.2 Methods

`beginAllBlk()`

Begins an enumeration of all blocks of the range. On exit from this member, the `Block` componenent of the `Index` contains the parameters of the first block of the locally held segment of the range, and the `Location` component represents the first element of that block. "First block" simply means first in the enumeration.

If `i` is the loop index, the idiom for the loop is

```
for(i.beginAllBlk() ; i.test() ; i.nextAllBlk())
  S
```

`nextAllBlk()`

Move to next block in enumeration.

7.6.3 Use of `AllBlocksIndex` in the communication library

The run-time technology introduced in this chapter for translation of distributed loops is used extensively in the implementation of the collective communication library. In that context, blocks of ranges are usually more important than individual elements, so use of the `LocBlocksIndex` iterator is common.

Besides enumeration of locally held blocks, it is sometimes necessary to enumerate *all* blocks of a range—local and non-local. For example, a data remapping operation defines a map between the elements of two ranges with different distribution formats. A local block in one range maps to a specific subrange of the other. But in general this subrange does not correspond to a single block of the target range—it may be distributed over several processes, divided into a number of blocks. If the original block of data is to be moved to the target array, it must be split into a corresponding number of data blocks, each sent to the appropriate destination.

All blocks enumeration resembles the local blocks enumeration given in section 7.2, but the loop is started using the `beginAllBlk()` member of `AllBlocksIndex`.

```
AllBlocksIndex i(x) ;
...
for(i.beginAllBlk() ; i.test() ; i.nextAllBlk()) {
  ...
}
```

In each iteration, the `crd` field inherited from the `Location` component of `i` gives the process coordinate of the remote block, while the fields of the `Block` component define the subscript range. If it is necessary to compute memory offsets in the remote process, this can be done using `offset` and related members of `Stride` in exactly the same way as for a local block.

There is no guarantee that enumeration of the blocks occurs in an obvious sequential order. It *is* guaranteed that the subsequence of blocks associated with a single process is ordered in the same way as for *local blocks* enumeration—this guarantee is necessary to ensure message blocks are sent and received in the same order *[need clarification on this point of ordering].*

Chapter 8

Implementation of the communication schedules