Introduction to Java-Ad

Bryan Carpenter, Guansong Zhang, Geoffrey Fox
Xinying Li, Yuhong Wen

NPAC at Syracuse University
Syracuse, NY 13244
{*dbc,zgs,gcf,xli,wen*}*npac.syr.edu*

November 14, 1997

Abstract

We outline an extension to Java for programming with distributed arrays. The basic programming style is Single Program Multiple Data (SPMD), but parallel arrays are provided as new language primitives. Further extensions include three *distributed control* constructs, the most important being a data-parallel loop construct. Communications involving distributed arrays are handled through a standard library of collective operations. Because the underlying programming model is SPMD programming, direct calls to MPI or other communication packages are also allowed in a Java-Ad program.

1 Introduction

The Java-Ad programming model is a flexible hybrid of the SPMD[1] and data parallel approaches to parallel programming. It provides HPF-like distributed arrays as language primitives, and new *distributed control* constructs to facilitate access to the local elements of these arrays. In the SPMD mold, the model allows processors the freedom to independently execute complex procedures on local elements: it is not restricted by SIMD-style array syntax.

In Java-Ad, all access to *non-local* array elements must go through library functions—typically collective communication operations. This puts an extra onus on the programmer; but making communication explicit encourages the programmer to write algorithms that exploit locality, and simplifies the task of the compiler writer. On the other hand, by providing distributed arrays as language primitives we are able to simplify error-prone tasks such as converting between local and global array subscripts and determining which processor holds a particular element. In the Java-Ad model the programmer never has to deal explicitly with local array subscripts. As in HPF, it is possible to write programs at a natural level of abstraction where the meaning is insensitive to the detailed mapping of elements. Lower-level styles of programming are also possible.

Our Java-Ad compiler will be implemented as a translator to ordinary Java with calls to a suitable run-time library. At the time of writing the underlying library is already available, and the interface needed by the Java-Ad translator is under development. The translator itself is being implemented in a compiler construction framework developed in the PCRC project.

2 Multidimensional arrays

First we describe a modest extension of Java that would add a class of true multidimensional arrays to the standard Java language. The new arrays allow regular section subscripting, similar to Fortran 90 arrays. The syntax described in this section is a subset of the Java-Ad syntax for parallel arrays and algorithms: the motivation for introducing the sequential subset first is just to simplify the overall presentation.

No attempt is made to integrate the new multidimensional arrays with the standard Java arrays: they are a new kind of entity that coexists in the language with ordinary Java arrays. There are good technical reasons for keeping the two kinds of array separate[2].

The type-signatures and constructors of the multidimensional array use double brackets to distinguish them from ordinary arrays:

```
int [[,]] a = new int [[5, 5]] ;
```

[1] Single Program, Multiple Data

[2] The run-time representation of our multi-dimensional arrays includes extra descriptor information that would simply encumber the large class "non-scientific" Java applications.

```
float [[,,]] b = new float [[10, n, 20]] ;

int [[]] c = new int [[100]] ;
```

a, b and c are respectively 2-, 3- and one- dimensional arrays. Of course c is very similar in structure to the standard array d, created by

```
int [] d = new int [100] ;
```

c and d are not identical, though[3].

Accessing individual elements of a multidimensional array goes through a subscripting operation involving single brackets

```
for(int i = 0 ; i < 4 ; i++)
  a [i, i + 1] = i + c [i] ;
```

For reasons that will become clearer in later sections, this style of subscripting is called *local subscripting*. In the current sequential context, apart from the fact that a single pair of brackest may include several comma-separated subscripts, this kind of subscripting behaves just like ordinary Java array subscripting. Subscripts always start at zero, in the ordinary Java or C style (there is no Fortran-like lower bound).

In general Java-Ad has no idea of Fortran-like array assignments. In

```
int [[,]] e = new int [[n, m]] ;
...
a = e ;
```

the assignment simply copies a handle to object referenced by e into a. There is no element-by-element copy involved. Similarly Java-Ad has no idea of elemental arithmetic or elemental function application. If e and a are arrays, the expressions

```
e + a
Math.cos(e)
```

are type errors.

Java-Ad *does* import a Fortran-90-like idea of array *regular sections*. The syntax for *section subscripting* is different to the syntax for local subscripting. Double brackets are used. These brackets can include scalar subscripts or *subscript triplets*.

A section is an object in its own right—its type is that of a suitable multidimensional array. It describes some subset of the elements of the parent array. This is slightly different to the situation in Fortran, where sections cannot usually be captured as named entities[4].

[3] For example, c allows section subscripting, whereas d does not.

[4] Unless a section appears as an actual argument to a procedure, in which case the dummy argument names that section, or it is the target of a pointer assignment.

```
int [[]] e = a [[2,  2 :]] ;

foo(b [[ : , 0, 1 : 10 : 2]]) ;
```

`e` becomes an alias for the 3rd row of elements of `a`. The procedure `foo` accepts a two-dimensional array as argument. It can read or write to the set of elements of `b` selected by the section. As in Fortran, upper or lower bounds can be omitted in triplets, defaulting to the actual bound of the parent array, and the stride entry of the triplet is optional. Note that the subscripts of `e`, like any other array, start at 0, although the first element is identified with `a [2, 2]`.

In Java-Ad, unlike Fortran, it is not allowed to use vectors of integers as subscripts. The only sections recognized are regular sections defined through scalar and triplet subscripts.

Java-Ad provides a library of functions for manipulating its arrays, closely analogous to the array transformational intrinsic functions of Fortran 90:

```
int [[,]] f = new int [[5, 5]] ;
Adlib.shift(f, a, -1, 0, CYCL) ;

float g = Adlib.sum(b) ;

int [[]] h = new int [[100]] ;
Adlib.copy(h, c) ;
```

The `shift` operation with shift-mode `CYCL` executes a cyclic shift on the data in its second argument, copying the result to its first argument—an array of the same shape. In the example the shift amount is -1, and the shift is performed in dimension 0 of the array—ie, the first of its two dimensions. The `sum` operation simply adds all elements of its argument array. The `copy` operation copies the elements of its second argument to its first—it is something like an array assignment. These functions may have to be overloaded to apply to some finite set of array types, eg they may be defined for arrays with elements of any suitable Java primitive type, up to some maximum rank of array. Alternatively the type-hierarchy for arrays may be defined in a way that allows these functions to be more polymorphic.

3 Process arrays

Java-Ad adds class libraries and some additional syntax for dealing with *distributed arrays*. These arrays are viewed coherent global entities, but their elements are divided across a set of cooperating processes. As a pre-requisite to introducing distributed arrays we discuss the *process arrays* over which their elements are scattered.

An abstract base class `Procs` has subclasses `Procs1`, `Procs2`, ..., representing one-dimensional process arrays, two-dimensional process arrays, and so on.

```
Procs2 p = new Procs2(2, 2) ;
Procs1 q = new Procs1(4) ;
```

These declarations set `p` to represent a 2 by 2 process array and `q` to represent a 4-element, one-dimensional process array. In either case the object created describes a group of 4 processes. At the time the `Procs` constructors are executed the program should be executing on four or more processes. Either constructor selects four processes from this set and identifies them as members of the constructed group[5].

`Procs` has a member function called `member`, returning a boolean value. This is `true` if the local process is a member of the group, `false` otherwise.

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

The code inside the `if` is executed only if the local process is a member `p`. We will say that inside this construct the *active process group* is restricted to `p`.

The multi-dimensional structure of a process array is reflected in its set of *process dimensions*. An object is associated with each dimension. These objects are accessed through the inquiry member `dim`:

```
Dimension x = p.dim(0) ;
Dimension y = p.dim(1) ;

Dimension z = q.dim(0) ;
```

The object returned by the `dim` inquiry has class `Dimension`. The members of this class include the inquiry `crd`. This returns the coordinate of the local process with respect to the process dimension. The result is only well-defined if the local process is a member of the parent process array. The inner body code in

```
if(p.member())
  if(x.crd() == 0)
    if(y.crd() == 0) {
      ...
    }
```

will only execute on the first process from `p`, with coordinates $(0, 0)$.

4 Distributed arrays

Some or all of the dimensions of a multi-dimensional array can be declared to be *distributed ranges*. In general a distributed range is represented by an

[5]There is no cooperation between the two constructor calls for `p` and `q`, so an individual physical process might occur in both groups or in neither. As an option not illustrated here, vectors of ids can be passed to the `Procs` constructors to specify exactly which processes are included in a particular group.

object of class `Range`. A `Range` object defines a range of integer subscripts, and defines how they are mapped into a process array dimension. In fact the `Dimension` class introduced in the previous section is a subclass of `Range`. In this case the integer range is just the range of coordinate values associated with the dimension. Each value in the range is mapped, of course, to the process (or slice of processes) with that coordinate. This kind of range is also called a *primitive range*. More complex subclasses of `Range` implement more elaborate maps from integer ranges to process dimensions. Some of these will be introduced in later sections. For now we concentrate on arrays constructed with `Dimension` objects as their distributed ranges.

The syntax of section 2 is extended in the following way to support distributed arrays

- A distributed range object may appear in place of an integer extent in the "constructor" of the array (the expression following the `new` keyword).
- If a particular dimension of the array has a distributed range, the corresponding slot in the type signature of the array should include a `#` symbol.
- In general the constructor of the distributed array must be followed by an `on` clause, specifying the process group over which the array is distributed. Distributed ranges of the array must be distributed over distinct dimensions of this group[6].

Assume `p`, `x` and `y` are declared as in the previous section, then

```
float [[#,#,]] a = new float [[x, y, 100]] on p ;
```

defines `a` as a 2 by 2 by 100 array of floating point numbers. Because the first two dimensions of the array are distributed ranges—dimensions of `p`—`a` is actually realized as four segments of 100 elements, one in each of the processes of `p`. The process in `p` with coordinates `i`, `j` holds the section `a [[i, j, :]]`.

The distribute array `a` is equivalent in terms of storage to four local arrays defined by

```
float [] b = new float [100] ;
```

But because `a` is declared as a collective object we can apply collective operations to it. The `Adlib` functions introduced in section 2 apply equally well to distributed arrays, but now they imply inter-processor communication.

```
float [[#,#,]] a = new float [[x, y, 100]] on p,
               b = new float [[x, y, 100]] on p ;

Adlib.shift(a, b, -1, 0, CYCL) ;
```

[6] The `on` clause can be omitted in some circumstances—see section 5.

The `shift` operation causes the local values of `a` is overwritten with the values of `b` from a processor adjacent in the `x` dimension.

There is a catch in this. When subscripting the distributed dimensions of an array it is *simply disallowed* to use subscripts that refer to off-processor elements. While this:

```
int i = x.crd(), j = y.crd() ;

a [i, j, 20] = a [i, j, 21] ;
```

is allowed, this:

```
int i = x.crd(), j = y.crd() ;

a [i, j, 20] = b [(i + 1) % 2, j, 20] ;
```

is forbidden. The second example could apparently be implemented using a nearest neighbour communication, quite similar to the `shift` example above. But Java-Ad imposes an strict policy distinguishing it from many data parallel languages: while library functions may introduce communications, language primitives such as array subscripting *never* imply communication.

If subscripting distributed dimensions is so restricted, why are the `i`, `j` subscripts on the arrays needed at all? In the examples of this section these subscripts are only allowed one value on each processor. Well, the inconvience of specifying the subscripts will be reduced by language constructs introduced later, and the fact that only one subscript value is local is a special feature of the *primitive ranges* used here. The higher level distributed ranges introduced later map multiple elements to individual processes. Subscripting will no longer look so redundant.

We finish this section with a fairly complex example using the notation established so far. The algorithm of figure 1 implements multiplication of two $\mathtt{N} \times \mathtt{N}$ matrices. One dimension of each of the two matrices is block-distributed over the `P` processors of `p`, so `N` is equal to $\mathtt{P} \times \mathtt{B}$ where `B` is the (constant) local block size.

The matrices are represented as three dimensional arrays, with their distributed dimensions explicitly split into a distributed range of extent `P` and a local sequential range of extent `B`. In later sections we will see how to represent this distribution format with a single block-distributed `Range` object. Even with that facility available, the representation used here may still be more natural for algorithms like the current one, where the block structure is an integral to the algorithm. The undistributed dimensions of the matrices are just sequential ranges of extent `N`. The operation of the algorithm for $\mathtt{P} = 2$ is visualized in figure 2. There are two phases. Between the phases the data in `b` is exchanged by the `shift` operation[7].

[7] In fact it is neccessary to use a `shift` and `copy` operation because the source and destina-

```
Procs1 p = new Procs1(P) ;
if(p.member()) {
  Range x = p.dim(0) ;

  float a [[#,,]] = new float [[x, B, N]] on p ;
  float b [[#,,]] = new float [[x, N, B]] on p ;

  ... initialize 'a', 'b'

  float c [[#,,]] = new float [[x, B, N]] on p ;

  for(int s = 0 ; s < P ; s++) {

    const int ip = x.crd() ;
    const int base = B * ((ip + s) % P) ;

    // c [[ip, :, base : ...]] =
    //         a [[ip, :, :]] * b' [[(ip + s) % P, :, :]] ...

    for(int ib = 0 ; ib < B ; ib++)
      for(int kb = 0 ; kb < B ; kb++) {

        float sum = 0 ;
        for(int j = 0 ; j < N ; j++)
          sum += a [ip, ib, j] * b [ip, j, kb] ;

        c [ip, ib, base + kb] = sum ;
      }

    float tmp [[#,,]] = new float [[x, N, B]] on p ;

    Adlib.shift(tmp, b, 1, 0, CYCL) ;
    Adlib.copy(b, tmp) ;
  }
}
```

Figure 1: A parallel matrix multiplication program.

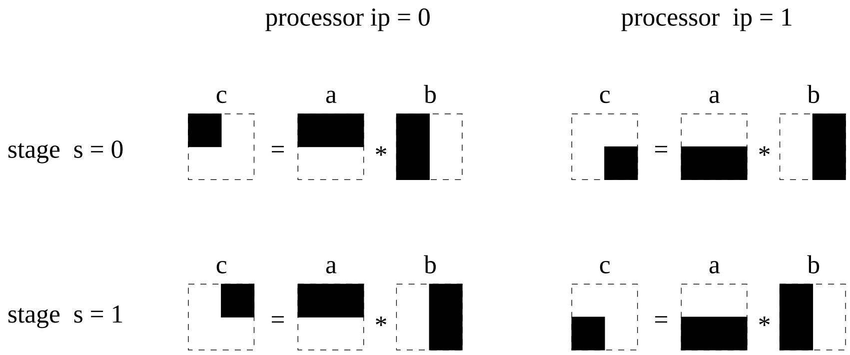

Figure 2: Operation of the program in figure 1 for P = 2.

5 The *on* construct and the active process group

In the last two section the idiom

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

has appeared. Our language provides a short way of writing this construct

```
on(p) {
  ...
}
```

In fact the *on* construct provides some extra value. Informally we said in section 3 that the *active process group* is restricted to `p` inside the body of the `p.member()` conditional construct. As part of the language, Java-Ad includes a more formal idea of an active process group (APG). At any point of execution some process group is singled out as the APG. An `on(p)` construct specifically changes the value of the APG to `p`. On exit from the construct, the APG is restored to its value on entry.

Elevating the active process group to a part of the language allows some simplifications. For example, it provides a natural default for the `on` clause in array constructors. In the matrix multiplication program of the previous section the code

```
if(p.member()) {
  ...
  float a [[#,,]] = new float [[x, B, N]] on p ;
```

tion arguments of `shift` must be distinct arrays. In the comment explaining the inner block matrix multiplication, by the way, the symbol `b'` means the original unshifted value of the array `b`.

```
    float b [[#,,]] = new float [[x, N, B]] on p ;
    ...
  }
```

can be simplified to

```
  on(p) {
    ...
    float a [[#,,]] = new float [[x, B, N]] ;
    float b [[#,,]] = new float [[x, N, B]] ;
    ...
  }
```

More importantly, formally defining the active process group will simplify the statement of various rules about what operations are legal *inside* distributed control constructs like *on*.

6 Higher-level ranges and locations

The class `BlockRange` is a subclass of `Range` which describes a simple block-distributed range of subscripts. Like `BLOCK` distribution format in HPF, it maps blocks of contiguous subscripts to each element of its target process dimension[8]. The constructor of `BlockRange` usually takes two arguments: the extent of the range and a `Dimension` object defining the process dimension over which the new range is distributed.

```
  Procs2 p = new Procs2(3, 2) ;

  Range x = new BlockRange(100, p.dim(0)) ;
  Range y = new BlockRange(200, p.dim(1)) ;

  float [[#,#]] a = new float [[x, y]] on p ;
```

`a` is created as a 100×200 array, block-distributed in over the 6 processes in `p`. The fragment is essentially equivalent to the HPF declarations

```
  !HPF$ PROCESSORS p(3, 2)

  REAL a(100, 200)

  !HPF$ DISTRIBUTE a(BLOCK, BLOCK) ONTO p
```

Subscripting distributed arrays with non-primitive ranges introduces some new problems. An array access such as

```
  a [17, 23] = 13 ;
```

[8]Other higher-level ranges include `CyclicRange`, which produces the equivalent of `CYCLIC` distribution formation in HPF.

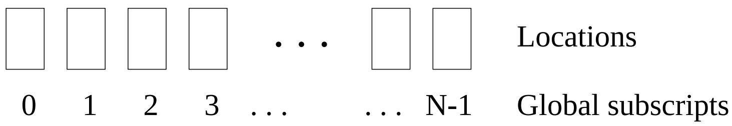

Figure 3: A range regarded as a set of locations, or slots.

is perfectly legal *if* the local process holds the element in question. But deterimining whether an element is local is not so easy any more. When arrays had only *primitive* distributed ranges, it was straightforward to check that accesses were local—the subscript simply had to be equal to the local coordinate. With higher-level ranges, that simple condition no longer holds.

In practise it is unusual to use integer values directly as local subscripts. Instead the idea of a *location* is introduced. A location can be viewed as an abstract element, or "slot", of a distributed range. Conversely, a range can be thought of as a set of locations. This model of a range is visualized in figure 3. An individual location is described by an object of the class `Location`. Each `Location` element is mapped to a particular slice of a process grid. In general two locations are identical only if they come from the same position in the same range. A subscripting syntax is used to represent location `n` in range `x`:

```
Location i = x [n]
```

This is an important idea in HPJava. By working in terms of abstract locations—elements of distributed ranges—on can usually respect locality of reference without resorting explicitly to low-level local subscripts and process ids. In fact the location can be viewed as an abstract data type incorporating these lower-level offsets.

Publically accessible fields of `Location` include `dim` and `crd`. The first is the process dimension of the parent range. The second is coordinate in that dimension to which the element is mapped. So the access to element `a [17, 23]` could now be guarded by conditionals as follows:

```
Location i = x [17], j = y [23] ;

if(i.crd == i.dim.crd())
  if(j.crd == j.dim.crd())
    a [17, 23] = 13 ;
```

This is still quite clumsy and error-prone. The language provides a second *distributed control construct* (analogous to *on*) to deal with this common situation. The new construct is called *at*, and takes a location as its argument. The fragment above can be replaced with

```
Location i = x [17], j = y [23] ;

at(i)
  at(j)
```

```
    a [17, 23] = 13 ;
```

This is more concise, but still involves some redundancy because the subscripts 17 and 23 appear twice. A natural extension is to allow locations to be used directly as array subscripts:

```
Location i = x [17], j = y [23] ;

at(i)
  at(j)
    a [i, j] = 13 ;
```

Locations used as array subscripts must be elements of the corresponding ranges of the array.

The range class has a member function

```
int Range.idx(Location i)
```

which can be used to recover the integer subscript, given a location in the range.

There is a restriction that an `at(i)` construct should only appear at a point of execution where `i.dim` is a dimension of the active process group. In the examples of this section this means that an `at(i)` construct, say, should normally be nested directly or indirectly inside an `on(p)` construct.

7 Distributed loops

As a matter of observation, good parallel algorithms don't usually expend many lines of code assigning to isolated elements of distributed arrays. Sequential access to elements of parallel arrays is best avoided. The *at* mechanism of the previous section is sometimes useful, but a more pressing need is an idiom for *parallel* access to distributed array elements.

The last and most important distributed control construct in Java-Ad is called *over*. It implements a distributed parallel loop. Conceptually it is quite similar to the `FORALL` construct of Fortran, except that the *over* construct specifies exactly where its parallel iterations are to be performed. The argument of *over* is a member of the special class `Index`. An index is associated with a particular range, which appears in the constructor of the object. The class `Index` is a subclass of `Location`, so it is syntactically correct to use an index as an array subscript[9]. Here is an example of a pair of nested *over* loops:

```
float [[#,#]] a = new float [[x, y]],
              b = new float [[x, y]] ;
...
Index i = new Index(x), j = new Index(y) ;
over(i)
```

[9] But the effect of such subscripting is only well-defined inside an *over* construct parametrised by the index in question.

```
  over(j)
    a [i, j] = 2 * b [i, j] ;
```

The body of an *over* construct executes, conceptually in parallel, for every location in the range of its index. An individual "iteration" executes on just those processors holding the location associated with the iteration. In a particular iteration, the location component of the index (the base class object) is equal to that location. The net effect of the example above should be reasonably clear. It assigns twice the value of each element of `b` to the corresponding element of `a`. Because of the rules about *where* an individual iteration iterates, the body of an *over* can usually only usually combine elements of arrays that have some simple alignment relation relative to one another.

The `idx` member of range can be used in parallel updates to give expressions that depend on global index values, as in

```
Index i = new Index(x), j = new Index(y) ;
over(i)
  over(j)
    a [i, j] = x.idx(i) + y.idx(j) ;
```

With the *over* construct we can give some more useful examples of parallel programs. Figure 4 is the famous Jacobi iteration for a two dimensionsional Laplace equation. We have used cyclic `shift` to implement nearest neighbour communications[10].

Copying whole arrays into temporaries is not an efficient way of accessing nearest neighbours in an array. Because this is such a common pattern of communication, Java-Ad supports *ghost regions*. Distributed arrays can be created in such a way that the segment stored locally is extended with some halo. This halo caches values stored in the segments of adjacent processes. The cached values are explicitly bought up to date by the library operation `writeHalo`.

An optimized version of the Jacobi program is give in figure 5. This version only involves a singe array temporary. A new constructor for `BlockRange` is provided. This allows the width of the ghost extensions to be specified. The arguments of `writeHalo` itself are an array with suitable extensions and two vectors. The first defines in each dimension the width of the halo that must actually be updated, and the second defines the treatment at the ends of the range—in this case the ghost edges are updated with cyclic wraparound. The new constructor and new `writeHalo` function are simply standard library extensions. One new piece of syntax is needed: the addition and subtraction operators are overloaded so that integer offsets can be added or subtracted to `Location` objects, yielding new, shifted, locations. The usual access rules apply—this kind

[10]Laplace's equation with cyclic boundary conditions is not particularly useful, but it illustrates the language features. More interesting boundary conditions can easily be incorporated later. Incidentally, this is a suitable place to mention that the array arguments of `shift` must be *aligned* arrays—they must have identical distributed ranges.

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

on(p) {
  Range x = new BlockRange(100, p.dim(0)) ;
  Range y = new BlockRange(200, p.dim(1)) ;

  int [[#,#]] u = new int [[x, y]] ;

  // ... some code to initialise 'u'

  int [[#,#]] unx = new int [[x, y]], upx = new int [[x, y]],
              uny = new int [[x, y]], upy = new int [[x, y]] ;

  Adlib.shift(unx, u,  1, 0, CYCL) ;
  Adlib.shift(upx, u, -1, 0, CYCL) ;
  Adlib.shift(uny, u,  1, 1, CYCL) ;
  Adlib.shift(upy, u, -1, 1, CYCL) ;

  Index i = new Index(x), j = new Index(y) ;
  over(i)
    over(j)
      u [i, j] = 0.25 * (unx [i, j] + upx [i, j] +
                         uny [i, j] + upy [i, j]) ;
}
```

Figure 4: Jacobi iteration using `shift`.

```
Procs2 p(2, 2) ;

on(p) {
  Range x = new BlockRange(100, p.dim(0), 1) ;  // ghost width 1
  Range y = new BlockRange(200, p.dim(1), 1) ;  // ghost width 1

  int [[#,#]] u = new int [[x, y]] ;

  // ... some code to initialise 'u'

  int [] widths = {1, 1} ;        // Widths actually updated
  Mode [] modes = {CYCL, CYCL} ;  // Wraparound at ends.

  Adlib.writeHalo(u, widths, modes) ;

  int [[#,#]] v = new int [[x, y]] ;

  Index i = new Index(x), j = new Index(y) ;
  over(i)
    over(j)
      v [i, j] = 0.25 * (u [i + 1, j] + u [i - 1, j] +
                         u [i, j + 1] + u [i, j + 1]) ;

  Adlib.copy(u, v) ;
}
```

Figure 5: Jacobi iteration using `writeHalo`.

of shifted access is illegal if it implies access to off-processor data. It only works if the subscripted array has suitable ghost extensions.

8 Subranges

A *subrange* is a section of a range, parametrized by a subscript triplet. Logically a subrange can be viewed as a subset of the locations of the original range. Subranges are members of the class `Range`. Because locations in a subrange are locations of the parent range, subranges retain an *alignment relation* to their parent range. Note that the *integer* subscripts for a subrange are in the range $0, \ldots, N-1$ where N is the extent of the *subrange*. See figure 6.

A triplet-subscripting syntax is used for creation of subranges: if `x` is a range, then `x [0 : 49]` is a contiguous subrange and `x [1 : 98 : 2]` is a strided subrange.

As a first application of subranges, we can uses strided subranges to trans-

Figure 6: Locations of a subrange (shaded slots).

form the Jacobi update of the previous section to a more efficient red-black form. The result is shown in figure 7. The iteration is split into two phases, the first with `parity` $= 0$ and the second with `parity` $= 1$. The range of the inner *over* construct is either `y [0 : : 2]` or `y [1 : : 2]`, according to whether the global `x` index of the outer loop has the same or different parity (odd/even) to the current phase. This version eliminates *all* temporary arrays[11]

As a second application involving subranges, figure 8 is a parallel version of Cholesky decomposition. In pseudocode the algorithm is

$$
\begin{array}{l}
l_{11} = a_{11}^{1/2} \\
For\ k = 1\ to\ n-1 \\
\quad For\ s = k+1\ to\ n \\
\qquad l_{sk} = a_{sk}/l_{kk} \\
\quad For\ j = k+1\ to\ n \\
\qquad For\ i = j\ to\ n \\
\qquad\quad a_{ij} = a_{ij} - l_{ik}l_{jk} \\
\quad l_{k+1,k+1} = a_{k+1,k+1}^{1/2}
\end{array}
$$

The array is distributed by columns, using cyclic distribution to improve load balancing. The collective communication function `remap` is used to broadcast updated columns. The `remap` function is one of the more powerful functions in the communication library. Like `copy`, its effect is to copy data from one distributed array to another of the same shape and type. But `copy` (like `shift`) has a restriction that its array arguments must be *aligned*—`copy` never introduces communication. With `remap` there is no such restriction—the mapping of the two arrays can be unrelated. One common application of `remap` is to *broadcast* data. If the target array has no ranges distributed over a dimension of the *process group* on which it lives, `remap` assumes that the result is to be stored in a replicated fashion. It therefore implements a broadcast. In the current example `remap` actually implements something more sophisticated than a simple broadcast. In MPI terms it executes a *gather-to-all* operation.

[11] Incidentally, subranges, and particularly strided subranges, introduce an ambiguity in the definition of the shift operators `+` and `-` on locations. Is the numeric shift measured terms of subscript relative to the subrange or of the parent range? As a matter of definition, the shift is always in terms of subscript in the *template* range—the ultimate parent from which a subrange is derived (by zero or more stages of triplet subscripting).

```
Procs2 p(2, 2) ;

on(p) {
  Range x = new BlockRange(100, p.dim(0), 1) ;  // ghost width 1
  Range y = new BlockRange(200, p.dim(1), 1) ;  // ghost width 1

  int [[#,#]] u = new int [[x, y]] ;

  // ... some code to initialise 'u'

  int [] widths = {1, 1} ;         // Widths actually updated
  Mode [] modes = {CYCL, CYCL} ;  // Wraparound at ends.

  for(int parity = 0 ; parity < 2 ; parity++) {

    Adlib.writeHalo(u, widths, modes) ;

    Index i = new Index(x) ;
    over(i) {
      Index j = new Index(y [(x.idx(i) + parity) % 2 : : 2]) ;
      over(j)
        u [i, j] = 0.25 * (u [i + 1, j] + u [i - 1, j] +
                           u [i, j + 1] + u [i, j + 1]) ;
    }
  }
}
```

Figure 7: Red-black iteration.

```
Procs1 p = new Procs1(P) ;
on(p) {
  Range x = new CyclicRange(N, p.dim(0));

  float [[,#]] a = new float [[N, x]] ;

  float [[]]   b = new float [[N]] ;  // buffer

  Location l = x [0] ;
  at(l)
    a [0, l] = sqrt(a [0, l]) ;

  for(int k = 0 ; k < N - 1 ; k++) {

    l = x [k] ;
    at(l)
      for(int s = k + 1 ; s < N ; s++)
        a [s, l] /= a [k, l] ;

    Adlib.remap(b [[k + 1 : ]], a [[k, k + 1 : ]]);

    Index m = new Index(x [k + 1 : ]) ;
    over(m)
      for(int i = x.idx(m) ; i < N ; i++)
        a [i, m] -= b [i] * b [m] ;

    l = x [k + 1] ;
    at(l)
      a [k + 1, l] = sqrt(a [k + 1, l]) ;
  }
}
```

Figure 8: Choleksy decomposition.

Some final comments on subranges. Creating a triplet subscripted section of a distributed array implicitly creates subranges of the ranges in the parent array. Also, arrays can be created directly with subranges, as in

```
Range xs = x [0 : 50] ;
Range ys = y [1 : 198 : 2] ;

int [[#,#]] e = new int [[xs, ys]] ;
```

In HPF terms, `e` has a non-trivial linear alignment to the template spanned by `x` and `y`. By allowing subranges (and subgroups, see section 9) to appear in array constructors we reproduce the two-level alignment model of HPF in full generality, at little cost in terms of syntax extensions.

9 Subgroups

A *subgroup* is some slice of a process array, formed by restricting the process coordinates in one or more dimensions to single values. Process arrays (class `Procs`) and subgroups have a common base class, `Group`. In general the argument of an *on* construct and the `on` clause in an array constructor is a member of `Group`. This implies that the active process group or the group over which an array is distributed may be just some a slice of a complete process array.

By definition, any group has a parent process array and a dimension set. In general the dimension set is some subset of the dimensions of the parent array. The *restriction* operation on a group takes a slice in a particular dimension. It is quite natural and convenient to express this restriction procedure is in terms of a *location*. If `i` is a location in a range distributed over a dimension of `p`, then

```
p / i
```

represents a subgroup of p—the slice of `p` to which location `i` is mapped.

Using the `/` operator on groups explicitly is fairly unusual practise. But subgroups are occur naturally in two ways:

- If an array `a` is distributed over `p`, a section of `a` will generally be distributed over some subgroup of `p`. For example, if the only *scalar* subscript in the section subscript list was location `i`, the section would be distributed over the subgroup `p / i`. Triplet subscripts don't change the group—only scalar subscripts.

- The distributed control constructs *over* and *at* change the active process group in a way that has not been described so far. Assume the current active process group is `p`, and `i` is a location. Then inside the construct

  ```
  at(i) {
    ...
  }
  ```

the active process group is equal to `p / i`. If the current active process group is `p`, and `i` is an *index*, then inside the construct

```
over(i) {
  ...
}
```

the active process group is equal to `p / i`. This case is slightly more subtle, because in different parallel "iterations" of the loop the location component of `i` has different values. In other words, the *over* construct *partitions* the original active process group into several subgroups (slices) operating independently.

To illustrate how subgroups can be used—in particular how the effect of *over* on the active process group can be exploited—we return to the matrix multiplication example of figure 1. As a preliminary step, figure 9 is transcribes that program using the distributed control constructs developed over the last few sections. The changes are very minor. Because the active process group is formally changed to `p`, the `on p` clauses can be omitted from the array constructors. Use of the `crd` inquiry to obtain the integer subscript in the `x` range is replaced by use of an *over* construct. Now we want to change from distribution over a one-dimensional process array to a two-dimensional grid.

The result is given in figure 10. Inside the `over(ip)` construct there are `P` independent active process groups corresponding to the rows of the original grid. The temporary array `tmp` is replicated over these rows. The `remap` operation, working independently in the `P` separate groups, implements a broadcast of the array section representing the block of the `a` stored on process `(ip + s) % P` within each group.

The freedom to embed calls to collective communication functions inside distributed control constructs is a distinctive feature of Java-Ad. There are a few restrictions on what operations are allowed. In general the requirement is that all array arguments of a collective operation should be *accessible* at the point of call. An array is accesible if it is distributed over a group contained in the active process group.

10 Class libraries or syntax extensions?

We have presented the Java-Ad language using a fairly liberal number of syntax extensions. The main extensions are:

- the syntax for distributed array type signatures,
- the syntax for the distributed array constructors,
- the syntax for local subscripting of distributed arrays,

```
Procs1 p = new Procs1(P) ;
on(p) {
  Range x = p.dim(0) ;

  float a [[#,,]] = new float [[x, B, N]] ;
  float b [[#,,]] = new float [[x, N, B]] ;

  ... initialize 'a', 'b'

  float c [[#,,]] = new float [[x, B, N]] ;

  for(int s = 0 ; s < P ; s++) {

    Index ip = new Index(x) ;
    over(ip) {
      const int base = B * ((x.idx(ip) + s) % P) ;

      // c [[ip, :, base : ...]] =
      //         a [[ip, :, :]] * b' [[(ip + s) % P, :, :]] ...

      for(int ib = 0 ; ib < B ; ib++)
        for(int kb = 0 ; kb < B ; kb++) {
          float sum = 0 ;
          for(int j = 0 ; j < N ; j++)
            sum += a [ip, ib, j] * b [ip, j, kb] ;

          c [ip, ib, base + kb] = sum ;
        }
    }

    float tmp [[#,,]] = new float [[x, N, B]] ;

    Adlib.shift(tmp, b, 1, 0, CYCL) ;
    Adlib.copy(b, tmp) ;
  }
}
```

Figure 9: Matrix multiplication program using distributed control constructs.

```
Procs2 p = new Procs2(P, P) ;
on(p) {
  Range x = p.dim(0), y = p.dim(1) ;

  float a [[#,#,,]] = new float [[x, y, B, B]] ;
  float b [[#,#,,]] = new float [[x, y, B, B]] ;

  ... initialize 'a', 'b'

  float c [[#,#,,]] = new float [[x, y, B, B]] ;

  for(int s = 0 ; s < P ; s++) {

    Index ip = new Index(x), jp = new Index(y) ;
    over(ip) {

      // Broadcast a [[ip, (ip + s) % P, :, :]]...

      float [[,]] tmp = new float [[,]] ;
      Adlib.remap(tmp, a [[ip, (x.idx(ip) + s) % P, :, :]]) ;

      over(jp) {

        // c [[ip, jp, :, :]] +=
        //             a [[ip, (ip + s) % P, :, :]] *
        //                 b' [[(ip + s) % P, jp, :, :]] ...

        for(int ib = 0 ; ib < B ; ib++)
          for(int kb = 0 ; kb < B ; kb++) {
            float sum = 0 ;
            for(int jb = 0 ; jb < B ; jb++)
              sum += tmp [ib, jb] * b [ip, jp, jb, kb] ;

            c [ip, jb, ib, kb] += sum ;
          }
       }
    }

    float tmp [[#,#,,]] = new float [[x, y, B, B]] ;

    Adlib.shift(tmp, b, 1, 0, CYCL) ;
    Adlib.copy(b, tmp) ;
  }
}
```

Figure 10: Matrix multiplication on a grid of processors.

- the syntax for section subscripting of distributed arrays,
- the syntax for the three distributed control constructs,
- the range subscripting syntax for creating locations and subranges,
- the overloaded division operator for creating subgroups, and the overloaded + and - operators for shifting locations.

Besides distributed arrays, we introduced four base classes that have a special significance in the context of the above syntax extensions:

- `Group`
- `Range`
- `Location`
- `Index`

Some of the syntax extensions are less important than others. The special syntax for creating locations, subranges and subgroups could be relaced by members of `Range` and `Group`:

```
Location Range.loc(int i)

Range Range.subrng(int lb, int ub, int stride)

Group Group.subgrp(Location i)
```

without much inconvenience to the programmer.

It would be possible to replace the special type signatures and constructors for distributed arrays with a series of ordinary class types,

```
Array1int
Array2int
...
Array1float
Array2float
...
...
```

and associated constructors. `Array1int` represents a the class of one-dimensional arrays of `int`, `Array2int` represents the class of two-dimensional arrays of `int`, and so on. The set of classes has to be quite large, and we must fix a finite limit on the rank of an array, and the set of array elements supported. Unless we allow the number of distinct array classes to grow *exponentially*, it becomes impractical to distinguish between distributed and sequential dimensions in the static type signature of an object. If an array is passed to a function (probably most arrays are) it is difficult for the compiler to deduce that a particular dimension is a sequential dimension. The compiler may end up generating code for global to local address conversion, even in sequential dimensions. This is probably a serious problem for efficiency, because sequential dimensions tend to be subscripted in an irregular way, not in uniform *over* loops. This problem could probably be overcome by requiring the subscripting operations for sequential and distributed dimensions explicitly differ (somehow) in the user's program. But that seems to complicate the language.

The syntax for the distributed control constructs and local subscripting seems more difficult to eliminate. In particular, the special syntax for *over* is critical. A Java-Ad translator will do the simple but tedious code transformations needed to replace distributed loops involving distributed arrays by local sequential loops involving local arrays, subscripted by expressions linear in the local loop indices. Unless the translator does this work, data-parallel programming becomes much less appealing. From our point of view this functionality is central to the SPMD data parallel model.