Parallel Programming in HPJava

Bryan Carpenter, Guansong Zhang, ...

Northeast Parallel Architectures Center
University of Syracuse
Syracuse, NY 13244
{*dbc,zgs,...*}*@npac.syr.edu*

January 2, 2000

Contents

Chapter 1

Introduction

HPJava is a language for parallel programming. It extends the Java language with some syntax for manipulating a new kind of parallel data structure—the *distributed array*. The specific extensions evolved out of work on Fortran 90 and High Performance Fortran (HPF), but the programming model of HPJava is different to HPF. The HPJava model is one of explicitly cooperating processes. It is an implementation of of the *Single Program, Multiple Data* (SPMD) model. All processes execute the same program, but the components of data structures—in our case the elements of distributed arrays—are divided across processes. Individual processes operate on the locally owned segment of an entire array. At some points in the computation processes usually need access to elements owned by their peers. Explicit communications are needed to permit this access.

This general scheme has been very successful in realistic programs. It is probably no exaggeration to say that most successful applications of parallel computing to large scientific and numerical problems are programmed in this style. So HPJava is attempting to add some extra support at the language level for established practises of programmers.

Chapter 2

Processes and Arrays

2.1 Processes and Process Grids

An HPJava program is started concurrently in some set of processes[1]. In an HPJava program individual processes are distinguished by reference to special objects representing *groups* of processes. The processes in a group are typically organized as multidimensional arrays, or grids. For example, an HPJava program that is currently executing on 6 or more processes can define a 2 by 3 process grid as follows

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

`Procs2` is a class describing 2-dimensional grids of processes. The `Procs2` constructor is a *collective operation*. It should be invoked concurrently by all active processes. The process grid established as `p` is visualized in Figure 2.1. Here we imagine that the program started executing in 7 processes. The constructor singles out 6 of these processes and incorporates them in the grid. The labelling of the axes will be discussed below.

`Procs2` is a subclass of the special base class `Group`, which represents a general group of processes. The `Group` class has a special status in the language. An object which inherits this class can be used to parametrize the first special syntactic construct of HPJava—the *on construct*. After creating `p` we will typically want to perform some operations within the processes of the grid. An on construct restricts control to processes in its parameter group. For example in

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  ...
}
```

the code represented by the ellipsis is only executed by processes belonging to `p`. The lonely 7th processor of Figure 2.1 skips this block of code. The on construct

[1]Most of the time we will talk about *processes* rather than processors, but the assumption is that the different processes may be running on different processors to achieve a parallel speedup.

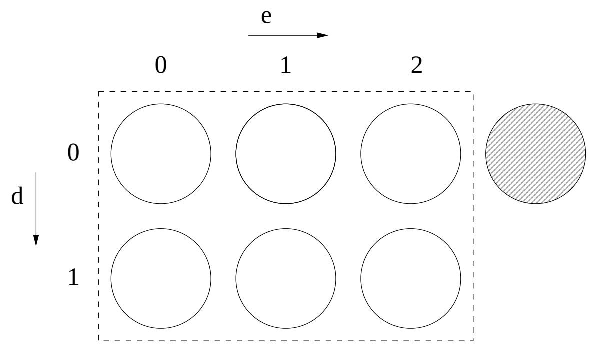

Figure 2.1: The process grid represented by `p`.

establishes `p` as the *active process group* for its body. We will see later that the active process group is singled out as a default argument for various operations that depend on groups.

An auxilliary class called `Dimension` is associated with process grids. Objects of this class describe a particular dimension or axis of a particular process grid. They will be referred to as *process dimensions*. The process dimensions for a grid are obtained through the inquiry member `dim`. The `Dimension` class in turn has a member `crd`, which returns the local *process coordinate* associated with the dimension, ie, the position of the local process within the dimension. If we executed the following HPJava program

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Dimension d = p.dim(0), e = p.dim(1) ;

  System.out.println("My coordinates are (" + d.crd() +
                                     ", " + e.crd() + ")") ;
}
```

we might see the output

```
My coordinates are (0, 2)
My coordinates are (1, 2)
My coordinates are (0, 0)
My coordinates are (1, 0)
My coordinates are (1, 1)
My coordinates are (0, 1)
```

Because the 6 processes are running concurrently there is no way to predict the order in which the messages appear (and if we were unlucky they might even be interleaved). If we had the bad judgement to apply `crd()` to `d` or `e` in a process outside `p` (such as our 7th processes) we could expect an exception.

Before ending this section we should reassure the reader that there is nothing special about 2-dimensional grids. The full `Group` hierachy of HPJava includes the classes of Figure 2.2. Not surprisingly, `Procs1` is a one-dimensional process

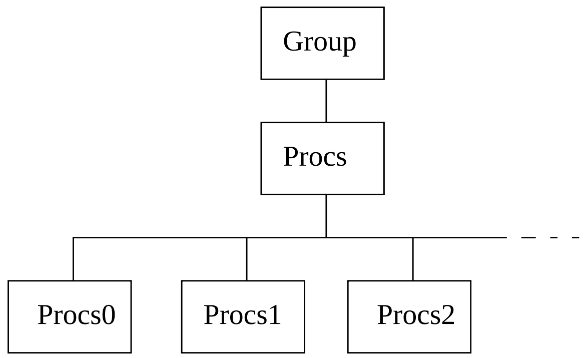

Figure 2.2: The `Group` hierachy of HPJava.

"grid". Less obviously, `Procs0` is a group containing exactly one process. Higher dimensional grids are also allowed.

2.2 Distributed Arrays

The biggest new feature HPJava adds to Java is the *distributed array.* Like the process grid objects introduced in the last section, a distributed array is logically an object shared by a number of processes. Like an ordinary array, a distributed array has some index space and stores a collection of elements of fixed type. Unlike an ordinary array, the index space and associated elements are scattered across the processes that share the array.

There are some superficial similarities between the way HPJava distributed arrays are used and the way ordinary Java arrays are used. There are also many differences. Apart from the fact that their elements are distributed in the manner just mentioned, the new HPJava arrays are true multi-dimensional arrays like those of Fortran (or, for that matter, C). As in Fortran, it is possible to form a *regular section* of an array. These characteristics of Fortran arrays have evolved to support scientific and parallel algorithms, and we consider them to be very desirable.

With these semantic distinctions in mind it is not particularly attractive to try and force the new HPJava arrays into a syntax closely reminiscent of ordinary Java arrays. Instead, HPJava distributed arrays look and feel quite different from standard arrays[2]. Of course HPJava includes Java as a subset,

[2]It is arguable that this is one of the places where High Performance Fortran went wrong. HPF tries to make distributed arrays semantically indistinguishable from the sequential arrays of the base language (Fortran). This means on the one hand that distributed arrays have to allow the same kind of unlimiteid random access as sequential arrays, which is difficult and inefficient to support. On the other hand, there are places where it is difficult to be sure at compile-time whether a subprogram will be dealing with a sequential array or a distributed array. The compiler may have to make a worst case assumption, leading to inefficiencies in treatment of sequential arrays.

and ordinary Java arrays can and should be used freely in an HPJava program. But they have no very close relationship to the new distributed arrays. The distributed array types can be regarded as a series of specialized container classes, dressed up in a special syntax.

The type signatures and constructors of distributed arrays use double brackets to distinguish them from ordinary Java arrays. The distribution of the index space is parametrized by objects belonging to the second special class of HPJava: the `Range` class. In the following example we create a two-dimensional, `N` by `N`, array of floating point numbers, with elements distributed over the grid `p`.

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  float [[,]] a = new float [[x, y]] ;

  ...
}
```

The decomposition of this array for the case `N` $= 8$ is illustrated in Figure 2.3. The choice of subclass `BlockRange` for the index ranges of the array mean that the index space of each array dimension is broken down into consecutive blocks. Other possible distribution formats will be discussed later. Notice how process dimensions are passed as arguments to the range constructors, specifying which dimensions the range is to be distributed over. Ranges of a single array must be distributed over different dimensions of the same process grid[3].

2.3 Parallel Programming

The previous section gave a simple example of how to create a distributed array. How do we use this kind of array? Figure 2.4 gives an example—parallel addition of two matrices.

The *overall construct* is the second special control construct of HPJava. It implements a parallel loop, sharing a heritage with the *forall* construct of HPF. The colon in the overall headers of the example is shorthand for an index *triplet*[4]. In general a triplet can include a lower bound, an upper bound, and a step, as follows:

```
overall(i = x for l : u : s)
```

In general l, u and s can be any integer expressions. The third member of the triplet (the step) is optional. The default step is 1, and most often we see something like

```
overall(i = x for l : u)
```

[3] Unless they are collapsed; see section 3.3.

[4] The syntax for triplets is lifted directly from Fortran 90.

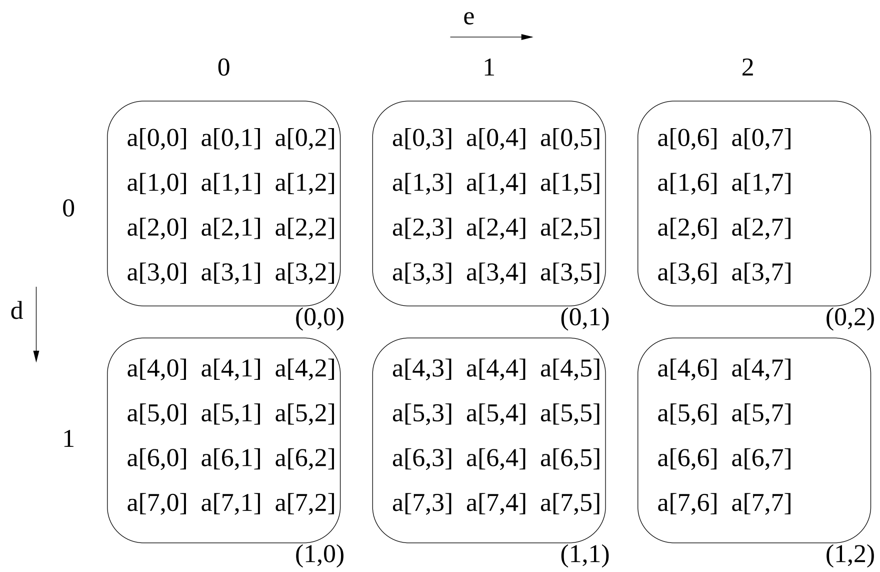

Figure 2.3: A two-dimensional array distributed over `p`.

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

  float [[,]] a = new float [[x, y]], b = new float [[x, y]],
              c = new float [[x, y]] ;

  ... initialize values in ‘a’, ‘b’

  overall(i = x for :)
    overall(j = y for :)
      c [i, j] = a [i, j] + b [i, j] ;
}
```

Figure 2.4: A parallel matrix addition.

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

  float [[,]] a = new float [[x, y]] ;

  float [[,]] n = new float [[x, y]], s = new float [[x, y]],
              e = new float [[x, y]], w = new float [[x, y]] ;

  ... initialize 'a'

  Adlib.shift(n, a,  1, 0) ;
  Adlib.shift(s, a, -1, 0) ;
  Adlib.shift(e, a,  1, 1) ;
  Adlib.shift(w, a, -1, 1) ;

  overall(i = x for 1 : N - 2)
    overall(j = y for 1 : N - 2)
      a [i, j] = 0.25 * (n [i, j] + s [i, j] + e [i, j] + w [i, j]) ;
}
```

Figure 2.5: A parallel stencil update program.

Finally, either or both of the expressions l and u can be omitted. The lower bound defaults to 0 and the upper bound defaults to $N-1$, where N is the extent of the range appearing before the `for` keyword. So in the original example the line

```
overall(i = x for :)
```

means "repeat for all elements, i, of the range x", and is equivalent to

```
overall(i = x for 0 : x.size() - 1 : 1)
```

The `size` member of `Range` returns the extent of the range.

For readers familiar with the forall construct of HPF (or Fortran 95) the only unexpected part of the overall syntax is the reference to a range object in front of the triplet. The significance of this will be discussed at length in the next section. Meanwhile we give another example of a parallel program. Figure 2.5 is a simple example of a "stencil update". Each interior element of array `a` is supposed to be replaced by the average of the values of its neighbours:

$$a[i,j] \;\leftarrow\; (a[i-1,j]+a[i+1,j]+a[i,j-1]+a[i,j+1])/4$$

The `shift` operation is not a new feature of the HPJava *language*, as such. Instead it is a member of a particular library called *Adlib*. This is a library of collective operations on distributed arrays[5]. The function `shift` is overloaded

[5]Many of them are modelled on the array transformational intrinsic functions of Fortran 90.

to apply to various kinds of array. In this example we are using the instance applicable to two dimensional arrays with `float` elements:

```
void shift(float [[,]] destination, float [[,]] source,
           int shiftAmount, int dimension) ;
```

The array arguments should have the same shape and distribution format. The values in the `source` array are copied to the `destination` array, shifted by `shiftAmount` in the `dimension`'th array dimension. Edge values from `source` that have no target in `destination` are discarded; edge elements of `destination` that are not overwritten by elements from `source` are unchanged from their input value.

In the example program, arrays of North, South, East and West neighbours are created using `shift`, then they are averaged in overall loops. An obvious question is: why go to the trouble of setting up these arrays? Surely it would be easier to write directly:

```
overall(i = x for 1 : N - 2)
  overall(j = y for 1 : N - 2)
    a [i, j] = 0.25 * (a [i - 1, j] + a [i + 1, j] +
                       a [i, j - 1] + a [i, j + 1]) ;
```

The answer relates to the nature of the symbols `i` and `j`. No declaration was given for these names, but it would be reasonable to assume that they stand for integer values.

They don't.

2.4 Locations

An HPJava range object can be thought of as a set of abstract *locations*. In our earlier example,

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  ...
}
```

the range `x`, for example, contains `N` locations. In an overall construct such as

```
overall(i = x for 1 : N - 2) {
  ...
}
```

the symbol `i` stands for a location, *not* simply an integer value. A location that is named by an overall construct is called a *bound location*.

With a few exceptions that will be discussed later, the subscript of a distributed array *must be a bound location*, and the location must be an element of

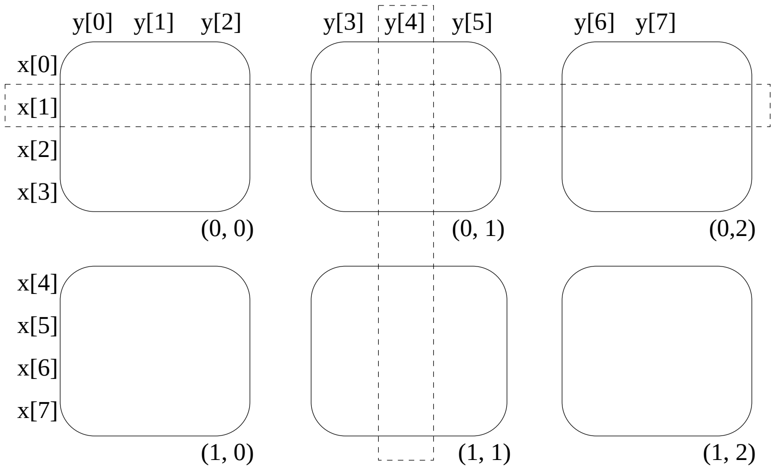

Figure 2.6: Mapping of `x` and `y` locations to the grid `p`.

the range associated with the array dimension. This is why we introduced the temporary arrays for neighbours in the stencil update example of the previous section. Arbitrary expressions are not usually legal subscripts for a distributed array. If we want to combine elements of arrays that are not precisely aligned, we first have to use a library function such as `shift` to align them[6].

Figure 2.6 is an attempt to visualize the mapping of locations from `x` and `y`. We will write the locations of `x` as `x[0]`, `x[1]`, ..., `x[N - 1]`. Each location is mapped to a particular group of processes. Location `x[1]` is mapped to the three processes with coordinates $(0,0)$, $(0,1)$ and $(0,2)$. Location `y[4]` is mapped to the two processes with coordinates $(0,1)$ and $(1,1)$.

Besides overall, there is another control construct in HPJava that defines a bound location—the simpler *at construct*. Suppose we want to update or access a single element of a distributed array (rather than accessing a whole set of of elements in parallel). We cannot simply write something like

```
float [[,]] a = new float [[x, y]] ;
...
a [1, 4] = 73 ;
```

because 1 and 4 are not bound locations, and therefore not legal subscripts. We *can* write:

```
float [[,]] a = new float [[x, y]] ;
...
at(i = x [1])
  at(j = y [4])
    a [i, j] = 73 ;
```

[6]Because stencil updates are so common, we will see in section 3.2 that, as a special case, the language *does* allow constantly shifted locations as subscripts. But exploiting this facility needs some special provisions.

Again the symbols i and j, scoped within the construct bodies, are bound locations.

The operational meaning of the at construct should be fairly clear. It is similar to the on construct. It restricts control to processes in the set that hold the specified location. Referring again to Figure 2.6, the outer

```
at(i = x [1])
```

construct limits execution of its body to processes with coordinates $(0, 0)$, $(0, 1)$ and $(0, 2)$. The inner

```
at(j = y [4])
```

restricts execution further to just process $(0, 1)$. This is exactly the process that owns element `a[1,4]`. The odd restriction that subscripts must be bound locations helps ensure that processes only manipulate array elements stored locally. If a process has to access non-local data, some explicit library call is needed to fetch it.

An operational definition of overall can be given in terms of the simpler at construct. The construct

```
overall(i = x for l : u : s) {
  ...
}
```

is equivalent in behaviour to

```
for(int n = l; n <= u ; n += s)
  at(i = x [n]) {
    ...
  }
```

The body of the at construct is skipped for values of `n` that don't correspond to locally held elements. In practise an HPJava compiler can translate the overall construct much more efficiently than this definition suggests.

The at construct completes the contingent of new control constructs in HPJava. We sometimes refer to the three constructs on, at and overall as *distributed control* constructs, and sometimes, more grandiosely, as *structured SPMD* control constructs.

The backquote symbol, \`, can be used as a postfix operator on a bound location, thus:

```
i`
```

This expression evaluates to the integer global index value. In the operational definition of the overall given above, this is the value called `n`.

We now know enough about HPJava to write some complete examples.

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

  float [[,]] a = new float [[x, y]] ;

  // Initialize ‘a’: set boundary values.

  overall(i = x for :)
    overall(j = y for :)
      if(i‘ == 0 || i‘ == N - 1 || j‘ == 0 || j‘ == N - 1)
        a [i, j] = i‘ * i‘ - j‘ * j‘ ;
      else
        a [i, j] = 0.0 ;

  // Main loop.

  float [[,]] n = new float [[x, y]], s = new float [[x, y]],
              e = new float [[x, y]], w = new float [[x, y]] ;

  float [[,]] r = new float [[x, y]] ;

  do {
    Adlib.shift(n, a,  1, 0) ;
    Adlib.shift(s, a, -1, 0) ;
    Adlib.shift(e, a,  1, 1) ;
    Adlib.shift(w, a, -1, 1) ;

    overall(i = x for 1 : N - 2)
      overall(j = y for 1 : N - 2) {
        float newA ;

        newA = 0.25 * (n [i, j] + s [i, j] + e [i, j] + w [i, j]) ;
        r [i, j] = Math.abs(newA - a [i, j]) ;
        a [i, j] = newA ;
      }

  } while(Adlib.maxval(r) > EPS) ;

  // Output results.

  Adlib.printArray(a) ;
}
```

Figure 2.7: Solution of Laplace equation by Jacobi relaxation.

2.5 A Complete Example

The example of Figure 2.7 only uses language features introduced in the preceding sections. It introduces two new library functions.

The problem is a very well-known one: solution of the two-dimensional Laplace equation with Dirichlet boundary conditions by the iterative Jacobi relaxation method[7]. The boundary conditions of the equation are fixed by setting edge elements of an array. These elements don't change throughout the computation. The solution for the interior region is obtained by iteration from some arbitrary starting values (zero, here). A single iteration involves replacing each interior element by the average of its neighbouring values. A similar update was already discussed in section 2.3. Here we put it in the context of a working program.

The initialization is done with a pair of nested overall constructs. Inside, a conditional tests if we are on an edge of the array. If so, the element values are set to some chosen expression—the boundary function. Otherwise we zero an interior element. As discussed at the end of the last section we, apply the operator ` to bound locations to get the global loop index.

Notice that one can freely use ordinary Java constructs like if inside an overall construct. HPJava distributed control construct are true, compositional control constructs. They differ in this respect from the HPF forall construct, which has restrictive rules about what kind of statement can appear in its body.

If preferred, the edges could have been initialized using at constructs:

```
at(i = x [0])
  overall(j = y for :)
    a [i, j] = i` * i` - j` * j` ;

at(i = x [N - 1])
  overall(j = y for :)
    a [i, j] = i` * i` - j` * j` ;

at(j = y [0])
  overall(i = x for :)
    a [i, j] = i` * i` - j` * j` ;

at(j = y [N - 1])
  overall(i = x for :)
    a [i, j] = i` * i` - j` * j` ;
```

with the interior initialized separately using nested overall constructs:

```
overall(i = x for 1 : N - 2)
  overall(j = y for 1 : N - 2)
    a [i, j] = 0.0
```

[7]Maybe we could have chosen a more creative first example. But the point is to explain language features—the more familiar the algorithm, the better.

This version is more long-winded but potentially more efficient, because it simplifies the inner loop bodies, improving the scope for compiler optimization.

The body of the main loop contains `shift` operations and nested overall loops. The body of the inner loop is slightly more complicated than the version in figure 2.5 because it saves changes to the main array in a separate array `r`.

Note the declaration of the `float` temporary `newA` inside the body of the parallel loop. This is perfectly good practise. The temporary is just an ordinary scalar Java variable—the HPJava translator doesn't treat it specially. Also note the call to a Java library function `abs` inside the loop. As we have emphasized, any normal Java operation is allowed inside an HPJava distributed control construct.

The main loop terminates when the largest change in any element is smaller than some predefined value `EPS`. The collective library function `maxval` finds the largest element of distributed array, and broadcasts its value to all processes that call the function. *[Need to initialize edges of* `r` *to zero.]*

Finally a collective library function `printArray` prints a formatted versions of the array on the standard output stream.

Chapter 3

More on Mapping Arrays

3.1 Other Distribution Formats

HPJava follows HPF in allowing several different distribution formats for the dimensions of its distributed arrays. The new formats are provided without further extension to the syntax of the language. Instead the `Range` class hierarchy is extended. The full hierarchy is shown in Figure 3.1.

The `BlockRange` subclass is very familiar by now. The `Dimension` class is also familiar, although previously it wasn't presented as a range class. Later examples will demonstrate how it can be convenient to use process dimension objects as array ranges. `CyclicRange` and `BlockCyclicRange` are directly analogous with the cyclic and block-cyclic distribution formats available in HPF.

Cyclic distributions are sometimes favoured because they can lead to better *load balancing* than the simple block distributions introduced so far. Some algorithms (for example dense matrix algorithms) don't have the kind of locality that favours block distribution for stencil updates, but they do involve phases where parallel computations are limited to subsections of the whole array. In block distributions these sections may map to only a fraction of the available processes, leaving the remaining processes idle. Here is a contrived example

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  float [[,]] a = new float [[x, y]] ;

  overall(i = x for 0 : N / 2 - 1)
    overall(j = y for 0 : N / 2 - 1)
      a [i, j] = complicatedFunction(i`, j`) ;
}
```

As shown in Figure 3.2, this leads to a very poor distribution of workload. The process with coordinates $(0, 0)$ does nearly all the work. The process at $(0, 1)$

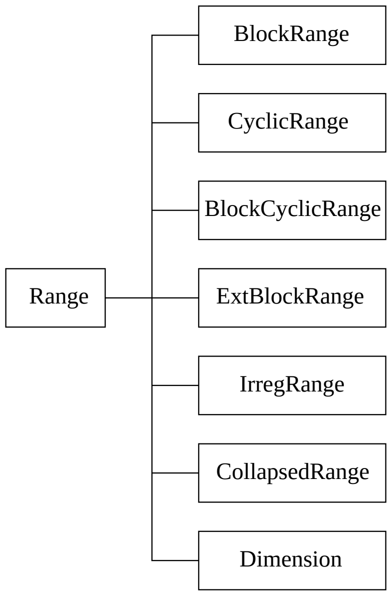

Figure 3.1: The `Range` hierachy of HPJava.

has a few elements to work on, and all the other processes are idle.

In *cyclic* distribution format, the index space is mapped to the process dimension in wraparound fashion. If we change our example to

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Range x = new CyclicRange(N, p.dim(0)) ;
  Range y = new CyclicRange(N, p.dim(1)) ;

  float [[,]] a = new float [[x, y]] ;

  overall(i = x for 0 : N / 2 - 1)
    overall(j = y for 0 : N / 2 - 1)
      a [i, j] = complicatedFunction(i`, j`) ;
}
```

Figure 3.3 shows that the imbalance is much less extreme. Notice that nothing changed in the program apart from the choice of range constructors. This is an attractive feature that HPJava shares with HPF. If HPJava programs are written in a sufficiently pure data parallel style (using overall loops and collective array operations) it is often possible to change the distribution format of arrays dramatically but leave much of the code that processes them invariant. HPJava does not guarantee this property in the same way that HPF does, and fully optimized HPJava programs are unlikely to be so easily redistributed. But it is still a useful feature.

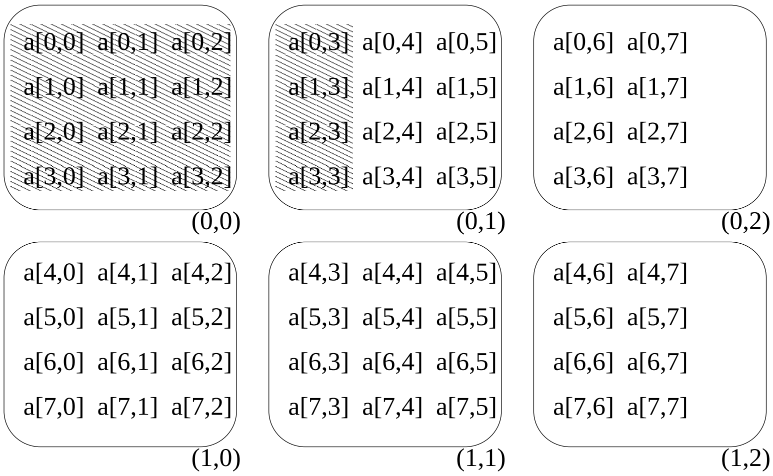

Figure 3.2: Work distribution for an example with block distribution.

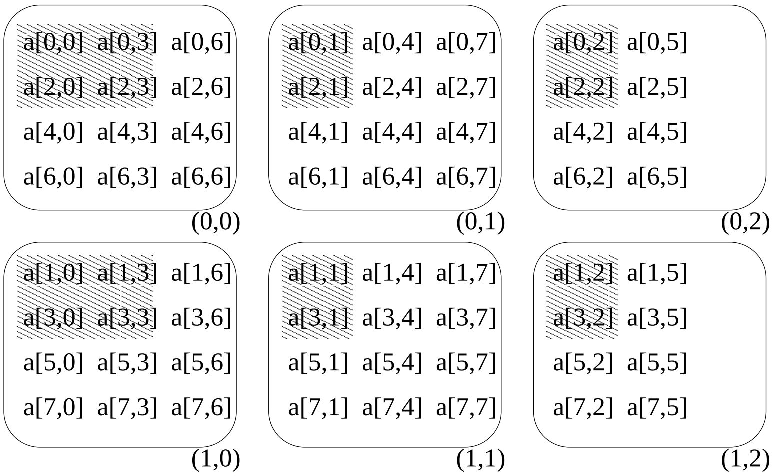

Figure 3.3: Work distribution for an example with cyclic distribution.

As a more graphic example of how cyclic distribution can improve load balancing, consider the Mandelbrot set code of Figure 3.4. Points away from the set are generally eliminated in a few iterations, whereas those close to the set or inside it take much more computation—points are assumed to be inside the set, and the corresponding element of the array is set to 1, when the number of iterations reaches `CUTOFF`. In the case `N` $= 64$ (with `CUTOFF` $= 100$), the block decomposition of the set is shown in Figure 3.5. The middle column of processors will do most of the work, because they hold most of the set. Changing the class of the ranges to `CyclicRange` gives the much more even distribution shown in Figure 3.6.

Block cyclic distribution format is a generalization of cyclic distribution that is used in some libraries for parallel linear algebra[1]. It will not be discussed further here.

The `ExtBlockRange` subclass represents block-distributed ranges extended with *ghost regions*.

3.2 Ghost Regions

In a distributed array with ghost regions, the memory for the locally held block of elements is allocated with extra space around the edges. These extra locations can be used to cache some of the element values properly belonging to adjacent processors. The inner loop of algorithms involving stencil updates can then written very simply. In accessing neighbouring elements, the edges of the block don't need special treatment. Rather than throwing an exception for an out of range subscript, shifted indices find the proper values cached in the ghost region. This is such an important technique in real codes that HPJava has a special extension to make it possible.

This is one place where the rule that the subscript of a distributed array must be a bound location is relaxed. In a special, slightly idiosyncratic, piece of syntax, the following expression is regarded as a bound location

$$\mathit{name} \pm \mathit{expression}$$

if *name* is the name of a bound location and *expression* is an integer expression—in practise is usually a small constant. This is called a shifted location. The significance of the shifted location is that an element displaced from the original location will be accessed. But if the shift goes outside the local block and surrounding ghost region, an exception will occur. So, after all, the example at the end of section 2.2:

```
overall(i = x for 1 : N - 2)
  overall(j = y for 1 : N - 2)
    a [i, j] = 0.25 * (a [i - 1, j] + a [i + 1, j] +
                       a [i, j - 1] + a [i, j + 1]) ;
```

[1]Notably ScaLAPACK.

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  int [[,]] set = new int [[x, y]] ;

  overall(i = x for :)
    overall(j = y for :) {
      float cr = (4.0 * i` - 2 * N) / N ;
      float ci = (4.0 * j` - 2 * N) / N ;

      float zr = cr, zi = ci ;

      set [i, j] = 0 ;

      int k = 0 ;
      while (zr * zr + zi * zi < 4.0) {
        if(k++ == CUTOFF) {
          set [i, j] = 1 ;
          break ;
        }

        // z = c + z * z

        float newr = cr + zr * zr - zi * zi ;
        float newi = ci + 2 * zr * zi ;

        zr = newr ;
        zi = newi ;
      }

    }

  Adlib.printArray(set) ;
}
```

Figure 3.4: Mandelbrot set computation.

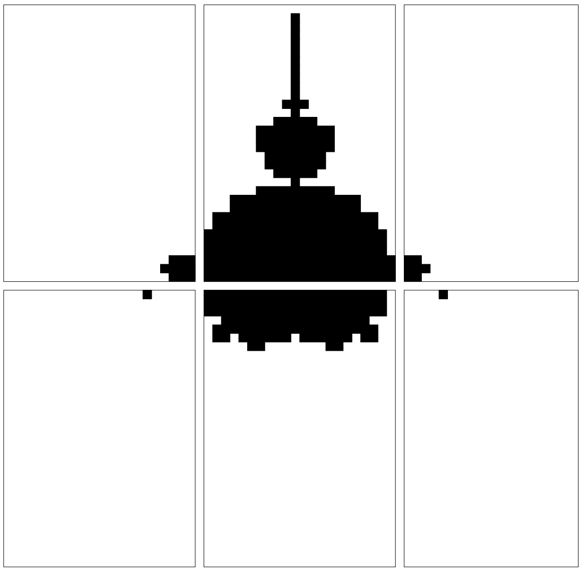

Figure 3.5: Blockwise decomposition of the Mandelbrot set (black region).

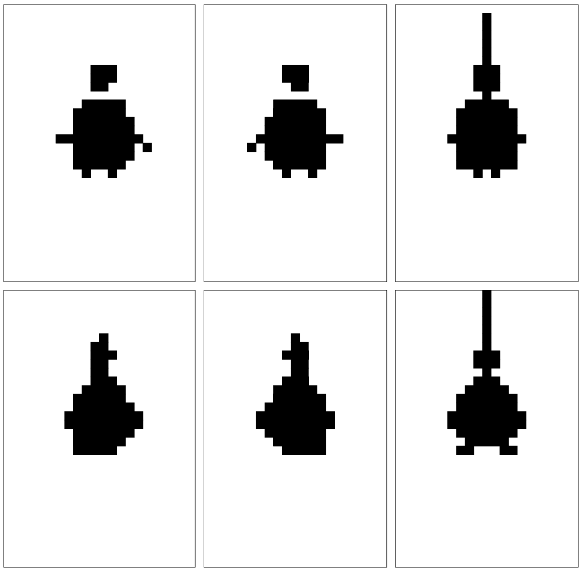

Figure 3.6: Cyclic decomposition of the Mandelbrot set.

is allowed if the array `a` has suitable ghost extensions[2].

Ghost regions are not magic. The extra locations around the edge of a block can only be made consistent with the values inside blocks on adjacent processes by a suitable communication. A library function called `writeHalo` updates the values cached in the ghost regions with proper element values from neigbouring processes.

Figure 3.7 is a version of the Laplace program that uses ghost regions. The ellipses stand for code that is unchanged from Figure 2.7. The last two arguments of the `ExtBlockRange` constructor define the widths of the ghost regions added to the bottom and top (respectively) of the local block. In this version we still introduced one temporary array, called `b`. The reason is is that in Jacobi relaxation one is supposed to express all the new values in terms of values from the previous iteration. The library function `copy` copies elements between two *aligned* arrays (`copy` does not implement communication; if it is passed non-aligned arrays, and exception occurs).

As a matter of fact, if we removed the temporary, `b`, and assigned `a` elements on-the-fly in terms of the current, partially updated `a` array, nothing very bad would happen. The algorithm may even converge faster because it is locally using the more efficient *Gauss-Siedel* relaxation scheme. Figure 3.8 is an implementation of the "red-black" scheme (a true Gauss-Siedel scheme), in which all even sites are updated, then all odd sites are updated in a separate phase. There is no need to introduce the temporary array `b`. This example illustrates the use of a stepped triplet in an overall construct.

The "footprint" of the stencil update can be more general. Figure 3.9 shows an implementation of Conway's Game of Life. The local update involves dependences on diagonal neighbours. This example also shows the most general form of `writeHalo` library function, which allows one to specify exactly how much of the available ghost areas are to be updated (this can be less than the total ghost area allocated for the array) and to specify a "mode" of updating the ghost cells at the extremes of the whole array. By specifying `CYCLIC` mode, cyclic boundary conditions are automatically implemented.

As a final example, Figure 3.10 is a Monte Carlo simulation of the well-known Ising model from condensed matter physics. It combines cyclic boundary conditions with red-black updating scheme. Random numbers are generated here using the `Random` class from the standard Java library. Random streams are created with different values in each process using some expression that depends on the local coordinate value. The method used here is certainly too naive for a reliable simulation, but it illustrates the principle.

[2] We need to be rather clear about the semantics of this extension, because it is an odd case. Suppose `x` is the range associated with the bound location `i`. Let `j` be the location `x [i‘ + e]`. If `j` is mapped to the same processes as `i`, and `i + e` is used as a subscript, it behaves just like the true location `j`. If `j` is mapped to a different set of processes from `i`, and `i + e` is used as a subscript, the resulting reference must be to an element in a ghost region on the process holding `i`.

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

  float [[,]] a = new float [[x, y]] ;

  ...

  // Main loop.

  float [[,]] b = new float [[x, y]], r = new float [[x, y]] ;

  do {
    Adlib.writeHalo(a) ;

    overall(i = x for 1 : N - 2)
      overall(j = y for 1 : N - 2) {
        float newA ;

        newA = 0.25 * (a [i - 1, j] + a [i + 1, j] +
                       a [i, j - 1] + a [i, j + 1]) ;

        r [i, j] = Math.abs(newA - a [i, j]) ;
        b [i, j] = newA ;
      }

    Adlib.copy(a, b) ;

  } while(Adlib.maxval(r) > EPS) ;

  ...
}
```

Figure 3.7: Solution of Laplace equation using ghost regions.

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

    Adlib.writeHalo(u) ;

    overall(i = x for 1 : N - 2)
      overall(j = y for 1 + (i` + parity) % 2 : N - 2 : 2) {
        float newA ;

        newA = 0.25 * (a [i - 1, j] + a [i + 1, j] +
                       a [i, j - 1] + a [i, j + 1]) ;

        r [i, j] = Math.abs(newA - a [i, j]) ;
        a [i, j] = newA ;
      }
  }

} while(Adlib.maxval(r) > EPS)) ;
```

Figure 3.8: Solution of Laplace equation using red-black relaxation (main loop only).

3.3 Collapsed and Replicated Distributions

The `CollapsedRange` subclass in Figure 3.1 stands for a range that is *not* distributed—all elements of the range are mapped to a single process. The code

```
Procs1 q = new Procs1(3) ;
on(p) {
  Range x = new CollapsedRange(N) ;
  Range y = new BlockRange(N, q.dim(0)) ;

  float [[,]] a = new float [[x, y]] ;

  ...
}
```

creates an array in which the second dimension is distributed over processes in `q`, with the first dimension *collapsed*. The situation is visualized for the case `N` = 8 in Figure 3.11. This is our first example of a one-dimensional process "grid".

Collapsed dimensions are often useful in algorithms where the pattern of access to elements is irregular in some but not all array dimensions. In the example above a process can access any element along the `x` dimension without communication, providing a fixed `y` location is mapped to the process concerned.

Unfortunately the language defined so far doesn't provide a good way to exploit this locality. If we want to assign the value in `a[6,1]` to `a[2,1]` we have to do something convoluted like

```
int wlo [] = {1, 1}, whi [] = {1, 1} ;  // ghost widths for ‘writeHalo’
int mode [] = {CYCL, CYCL} ;            // boundary conds for   "

Procs2 p = new Procs2(2, 2) ;
on(p) {
  Range x = new ExtBlockRange(N, p.dims(0), 1, 1) ;
  Range y = new ExtBlockRange(N, p.dims(1), 1, 1) ;

  int [[,]] state = new int [[x, y]] ;

  ... Define initial state of Life board

  // Main update loop.

  int sums [[,]] = new int [[x, y]] ;

  for(int iter = 0 ; iter < NITER ; iter++) {

    Adlib.writeHalo(state, wlo, whi, mode) ;

    // Calculate neighbour sums.

    overall(i = x for :)
      overall(j = y for :)
        sums [i, j] =
          state [i - 1, j - 1] + state [i - 1, j] + state [i - 1, j + 1] +
          state [i,     j - 1]                    + state [i,     j + 1] +
          state [i + 1, j - 1] + state [i + 1, j] + state [i + 1, j + 1] ;

    // Update state of board values.

    overall(i = x for :)
      overall(j = y for :)
        switch (sums [i, j]) {
          case 2 : break;
          case 3 : state [i, j] = 1; break;
          default: state [i, j] = 0; break;
        }
  }

  ... Output final state
}
```

Figure 3.9: Conway’s Game of Life.

```
int wlo [] = {1, 1}, whi [] = {1, 1} ;
int mode [] = {CYCL, CYCL} ;

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

  float [[,]] latt = new float [[x, y]] ;

  Random rand = new Random(97    * p.dim(0).crd() +
                           89793 * p.dim(1).crd()) ;

  ... Initialize 'latt' randomly with +1's and -1's.

  // Main loop.

  for (int sweep = 0 ; sweep < NSWEEPS ; sweep++) {

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

      Adlib.writeHalo(latt, wlo, whi, mode) ;

      overall(i = x for :)
        overall(j = y for (i` + parity) % 2 : : 2) {

          int oldVal = latt [i, j] ;
          int newVal = rand.nextFloat() < 0.5 ? -1 : 1 ;
                                  // Randomly choose +1 or -1

          int deltaE = (newVal - oldVal) *
                           (latt [i - 1, j] + latt [i + 1, j] +
                            latt [i, j - 1] + latt [i, j + 1]) ;

          if(rand.nextFloat() < Math.exp(- BETA * deltaE))
            latt [i, j] = newVal ;    // Accept, biased by energy
        }
    }
  }

  ... Analyse final configuration
}
```

Figure 3.10: Monte Carlo simulation of Ising model using the Metropolis algorithm.

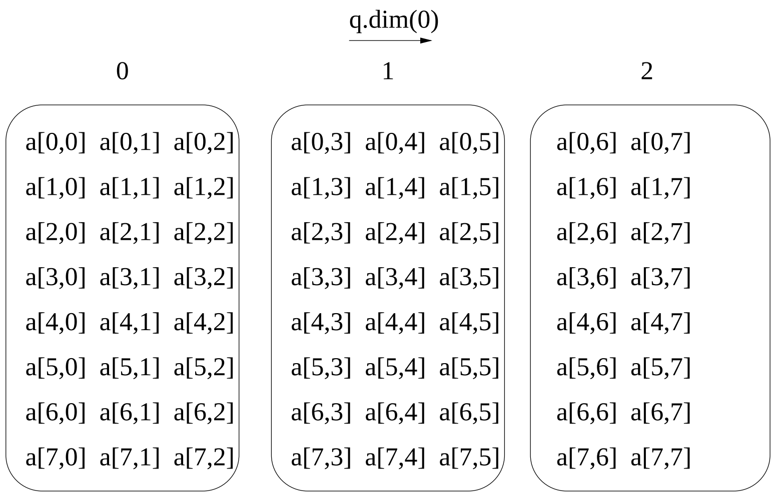

Figure 3.11: Two-dimensional array, `a`, distributed over the one-dimensional grid, `q`.

```
Procs1 q = new Procs1(3) ;
on(p) {
  Range x = new CollapsedRange(N) ;
  Range y = new BlockRange(N, q.dim(0)) ;

  float [[,]] a = new float [[x, y]] ;
  ...
  at(j = y [1]) {
    float local ;

    at(i = x [6])
      local = a [i, j] ;

    at(i = x [2])
      a [i, j] = local ;
  }
}
```

Because of the restriction that subscripts must be bound locations, the value of `a[6,1]` must first be read to a local variable in an at construct, then the value of the local variable must be copied to `a[2,1]` in another. This is very tedious, and probably quite inefficient.

To avoid this common problem, the HPJava model of distributed arrays is extended with the idea of *sequential dimensions*. If the type signature of a distributed array has an asterix in a particular dimension, that dimension will implicitly have a collapsed range, and the dimension can be subscripted with integer expressions just like a sequential array. The example becomes

```
Procs1 q = new Procs1(3) ;
```

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

  float [[,*]] a = new float [[x, N]], c = new float [[x, N]] ;
  float [[*,]] b = new float [[N, x]], tmp = new float [[N, x]] ;

  ... initialize 'a', 'b'

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

    overall(i = x for :) {

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

      c [i, (i` + s) % N] = sum ;
    }

    // cyclically shift 'b' (by amount 1 in x dim)...

    Adlib.cshift(tmp, b, 1, 1) ;
    HPspmd.copy(b, tmp) ;
  }
}
```

Figure 3.12: A pipelined matrix multiplication program.

```
on(p) {
  Range y = new BlockRange(N, q.dim(0)) ;

  float [[*,]] a = new float [[N, y]] ;
  ...
  at(j = y [1])
    a [6, j] = a [1, j] ;
}
```

The outer at construct is retained to deal with the distributed dimension, but there is no need for bound locations in the sequential dimension. The array constructor is passed integer extent expressions for sequential dimensions. A `CollapsedRange` object will be created for the array, but the programmer generally need not be aware of its existence.

Figure 3.12 is an example of a parallel matrix multiplication in which the first input array, `a`, and result array, `c`, are distributed by rows—each processor is allocated a consecutive set of complete rows. The first dimension of these array is distributed, breaking up the columns, while the second dimension is collapsed, leaving individual rows intact. The second input array, `b`, is distributed by columns.

Array types with sequential dimensions are technically subtypes of corresponding array types without sequential dimension. All operations generally applicable to distributed arrays are also applicable to arrays with sequential dimensions. The asterisk in the type signature adds the option of subscripting the associated with integer espressions. It does not remove any option allowed for distributed arrays in general.

Allowing collapsed array dimensions means that an array can be distributed over a process grid having smaller rank than the array itself. Conversely it is also allowed to distribute an array over a process grid with larger rank.

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

on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;

  float [[]] b = new float [[x]] ;

  ...
}
```

The array `b` has a dimension distributed over the first dimension of `p`, but none distributed over the second. The interpretation is that `b` is *replicated* over the second process dimension. Independent copies of the whole array are created at each coordinate where replication occurs. Usually programs maintain identical values for the elements in each copy (although there is nothing in the language definition itself to mandate this).

Replication and collapsing can both occur in a single array, for example

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

on(p) {
  float [[*]] c = new float [[N]] ;

  ...
}
```

The range of `c` is sequential, and the array is replicated over both dimensions of `p`. This makes it very similar to an ordinary Java array declared in all processes by

```
  float [] d = new float [N] ;
```

`c` and `d` are not identical, though. The array `c` can be passed to library functions that expect distributed arrays as arguments, whereas `d` cannot.

[Introduce the distribution group and the `on` *clause for the distributed array constructors here?]*

A simpler and potentially more efficient implementation of matrix multiplication can be given if the operand arrays have carefully chosen replicated/collapsed distributions. The program is given in Figure 3.13. As illustrated in Figure

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

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

  float [[,*]] a = new float [[x, N]] ;
  float [[*,]] b = new float [[N, y]] ;

  ... initialize 'a', 'b'

  overall(i = x for :)
    overall(j = y for :) {

      float sum = 0 ;
      for(int k = 0 ; k < N ; k++)
        sum += a [i, k] * b [k, j] ;

      c [i, j] = sum ;
    }
}
```

Figure 3.13: A direct matrix multiplication program.

3.14, the rows of a are replicated in the process dimension associated with y. Similarly the columns of b are replicated in the dimension associated with x. Hence all arguments for the inner scalar product are already in place for the computation—no communication is needed.

Clearly we would be very lucky to come across three arrays with such a special distribution format relative to one another (*alignment relation*). There is an important function in the Adlib library called remap, which takes a pair of arrays as arguments. These must have the same shape and type, but they can have unrelated distribution formats. The elements of the source array are copied to the destination array. In particular, if the destination array has a replicated mapping, the values in the source array are broadcast appropriately. Figure 3.15 shows how we can use remap to adapt the program in Figure 3.13 to create a general purpose matrix multiplication routine. Besides the remap function, this example introduces the two inquiry functions grp and rng which are defined for any distributed array.

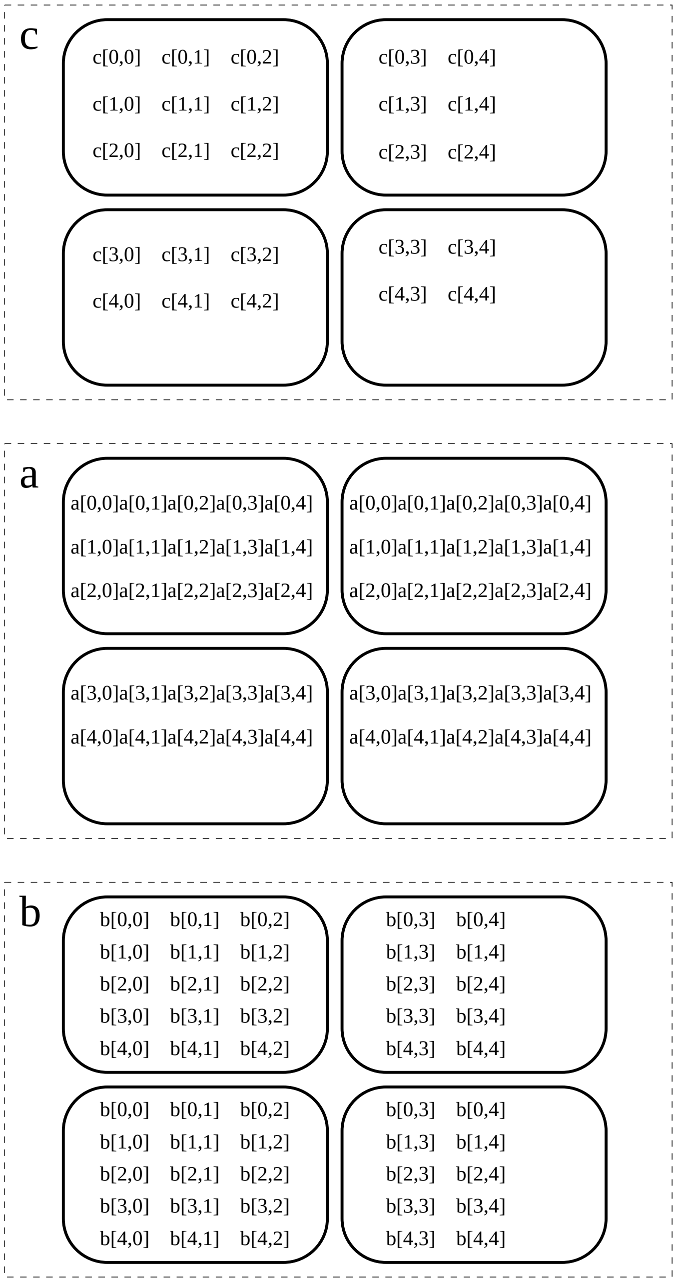

Figure 3.14: Distribution of array elements in example of Figure 3.13

```
void matmul(float [[,]] c, float [[,]] a, float [[,]] b) {

  Group p = c.grp() ;

  Range x = c.rng(0) ;
  Range y = c.rng(1) ;

  int N = a.rng(1).size() ;

  float [[,*]] ta = new float [[x, N]] on p ;
  float [[*,]] tb = new float [[N, y]] on p ;

  Adlib.remap(ta, a) ;
  Adlib.remap(tb, b) ;

  on(p)
    overall(i = x for :)
      overall(j = y for :) {

        float sum = 0 ;
        for(int k = 0 ; k < N ; k++)
          sum += ta [i, k] * tb [k, j] ;

        c [i, j] = sum ;
      }
}
```

Figure 3.15: A general matrix multiplication program.

Chapter 4

Array Sections

HPJava has a mechanism for representing subarrays. This mechanism is modelled on the *array sections* of Fortran 90. In HPJava an array section expression has a similar syntax to a distributed array element reference but uses double brackets. Whereas an element reference is a variable, an array section is an expression that represents a new distributed array object. The new array contains a subset of the elements of the parent array. Those elements can subsequently be referenced—read or updated—either through the parent array *or* through the array section[1].

We have seen that the subscripts in a distributed array element reference are either locations or (restrictedly) integer expressions. Options for subscripts in array section expressions are wider. Most importantly, as in Fortran 90, a section subscript is allowed be a triplet. Triplets were introduced in section 2.3 in the context of the overall construct. For each triplet subscript a section expression has an array dimension. In the commonest kinds of array section expression the rank of the result is equal to the number of triplet subscripts. The section may also have some scalar subscripts, similar to those appearing in element references. In this case the rank of the result will be lower than the rank of the parent array.

This fragment includes two examples of array section expressions:

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

  float [[,]] a = new float [[x, y]] ;

  float [[]] b = a [[0, :]]
```

[1]The HPJava idea of an array section expression has a close relationship to the Fortran 90 idea of an array pointer. In Fortran an array pointer can reference an arbitrary regular section of an array

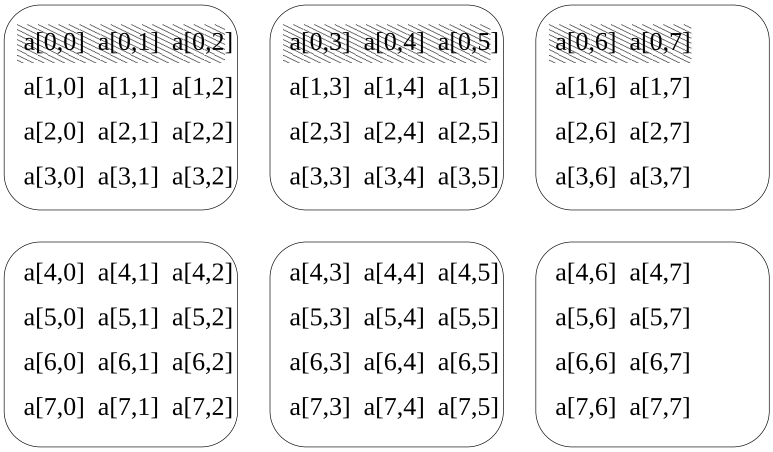

Figure 4.1: A one-dimensional section of a two-dimensional array (shaded area).

```
    foo(a [[0 : N / 2 - 1, : : 2]]) ;
  }
```

The first array section expression appears on the right hand side of the definition of `b`. It specifies `b` as an alias for the first row of `a` (Figure 4.1). Note that a scalar subscript in an array section expression is allowed to be an integer expression, even if the array dimension is distributed. This contrasts with element references, where they must be bound locations. The second array section expression appears as an argument to the method `foo`. It represents a two-dimensional, $N/2$ by $N/2$, subset of the elements of `a`, visualized in Figure 4.2. The lower and upper bounds are omitted in the second triplet subscript of this expression: as usual, they default 0 and $N - 1$.

Array sections allow us to implement a number of interesting applications. They are often passed as arguments to library functions like `remap`, implementing various interesting patterns of communication and arithmetic on subarrays.

4.1 Two-dimensional Fourier transform

In image processing applications Fast Fourier Transforms (FFTs) and related transformastions are sometimes applied to two-dimensional images. A two-dimensional FFT can be broken down into a series simpler one-dimensional FFTs—the one-dimensional transform is simply applied to every row of the image, then to every column. All rows can be transformed in parallel, then all columns can be transformed in parallel. An implementation is given in Figure 4.3 This implementation assumes the availability of a function for calculating FFTs on one-dimensional *collapsed* arrays (ie, a sequential one-dimensional FFT). Because Java doesn't have complex numbers, we store real and imaginary parts in pairs of arrays whose names are prefixed `re` and `im`. Alternatively a

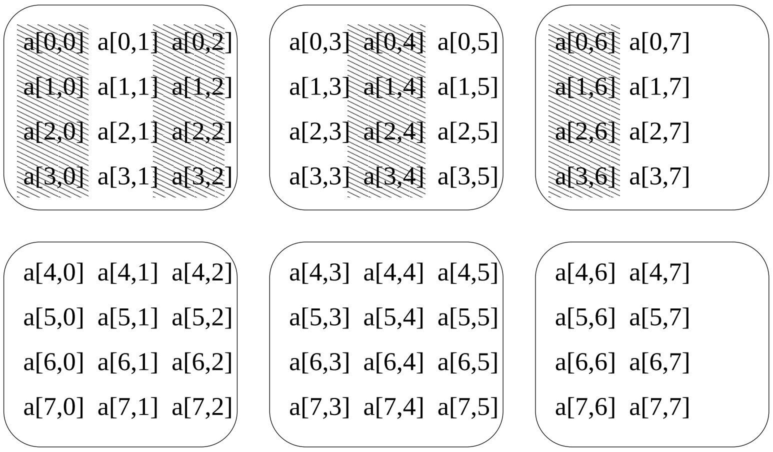

Figure 4.2: A two-dimensional section of a two-dimensional array (shaded area).

an extra dimension of extent 2 could be added to the arrays. After processing all columns, the data is remapped so that each row is a collapsed subarray. A section dimension naturally inherits the sequential property (the asterisk in the type signature) from the associated dimension of the parent array.

4.2 Cholesky decomposition

If A is a symmetric positive definite matrix, associated linear equations are often using Choleski decomposition:

$$A = LL^T$$

where L is a lower triangular matrix. In practise this is followed by forward and back substitutions:

$$L\mathbf{y} = \mathbf{b},\ L^T\mathbf{x} = \mathbf{y}$$

to complete the solution of $A\mathbf{x} = b$. A pseudocode algorithm for Cholesky decomposition is

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

A parallel version, assuming the main array is stored by columns with the rows cyclically distributed, is given in figure 4.4. The l array is accumulated in the lower part of the input array `a`. Note that the array `b` has a replicated

```
void fft1d(float [[*]] re, float [[*]] im) {

  // One-dimensional FFT on sequential (non-distributed) data.

  ...
}

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

  float [[,*]] reA = new float [[x, N]], imA = new float [[x, N]] ;
  float [[*,]] reB = new float [[N, x]], imB = new float [[N, x]] ;

  ... initial values in ‘reA’, ‘imA’

  overall(i = x for :)
    fft1d(reA [[i, :]], imA [[i, :]]) ;

  Adlib.remap(reB, reA) ;
  Adlib.remap(imB, imA) ;

  overall(i = x for :)
    fft1d(reB [[:, i]], imB [[:, i]]) ;

  ... result is in ‘reB’, ‘imB’
}
```

Figure 4.3: A two-dimensional Fourier Transform.

```
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]] ;  // a buffer

  ... some code to initialise 'a'

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

    at(j = x [k]) {
      float d =  Math.sqrt(a [k, j]) ;

      a [k, j] = d ;
      for(int s = k + 1 ; s < N ; s++)
        a [s, j] /= d ;
    }

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

    overall(j = x for k + 1 : )
      for(int i = j` ; i < N ; i++)
        a [i, j] -= b [i] * b [j`] ;
  }

  at(j = x [N - 1])
    a [N - 1, j] = Math.sqrt(a [N - 1, j]) ;
}
```

Figure 4.4: Choleksy decomposition.

distribution, so the `remap` operation is a broadcast of the relevant part of column k.

4.3 Matrix multiplication with reduced memory

One disadvantage of the program in Figure 3.15 is that it allocates two very large temporary arrays, `ta` and `tb`. Because these are both replicated in one dimension, they can easily consume more memory than the original data. This problem can be solved by only storing copies of elements in restricted bands of the original matrices at any one time. Figure 4.5 gives a modified algorithm where the maximum band width is `B`.

For simplicity we assumed here that `B` is a compile-time constant. Alternatively we can compute this value dynamically. The `volume()` method on `Range` is used internally by array constructors to control allocation of memory for array

```
void matmul(float [[,]] c, float [[,]] a, float [[,]] b) {

  Group p = c.grp() ;

  Range x = c.rng(0) ;
  Range y = c.rng(1) ;

  int N = a.rng(1).size() ;

  float [[,*]] ta = new float [[x, B]] on p ;
  float [[*,]] tb = new float [[B, y]] on p ;

  on(p)
    overall(i = x for :)
      overall(j = y for :)
        c [i, j] = 0 ;

  for(int base = 0 ; base < N ; base += B) {
    const int w = min(B, N - base) ;  // minimum value

    Adlib.remap(ta [[:, 0 : w - 1]], a [[:, base : base + w - 1]]) ;
    Adlib.remap(tb [[0 : w - 1, :]], b [[base : base + w - 1, :]]) ;

    on(p)
      overall(i = x for :)
        overall(j = y for :)
          for(int k = 0 ; k < w ; k++)
            c [i, j] += ta [i, k] * tb [k, j] ;
  }
}
```

Figure 4.5: Matrix multiplication with reduced memory requirement.

elements. It defines the largest block of locations for the current range held by any processor. Hence an upper bound on the number of elements held by any processor for `ta` and `tb` combined is

```
B * x.volume() + B * y.volume()
```

If `MAX_TEMPORARY_SIZE` is a constant defining a limit on the total volume of memory we ever wish to allocate for temporary arrays, a suitable formula for `B` might be

```
B = MAX_TEMPORARY_SIZE / (x.volume() + y.volume())
```

With a few refinements like this, the algorithm of Figure 4.5 is a credible basis for a library matrix multiplication routine, applicable to generic distributed arrays.

4.4 Subranges and restricted groups

Consider again the example of the array section in figure 4.2. We can capture this object in a named variable as follows

```
float [[,]] a = new float [[x, y]] ;

float [[,]] c = a [[0 : N / 2 - 1, : : 2]] ;
```

Now, what are the ranges of `c`—the objects returned by the `rng()` inquiry?

In fact they are a different sort of range from any considered so far—they are *subranges*. For completeness the HPJava language provides a special syntax for constructing subranges directly. Ranges equivalent to those of `c` can be created by

```
Range u = x [0 : N / 2 - 1] ;
Range v = y [: : 2] ;
```

This syntax should look quite natural. It is similar the subscripting syntax for locations, but the subscript is a triplet.

[Global subscripts in subranges.]

What about the distribution groups of sections? In this case triplet subscripts don't cause problems—the distribution group of `c` above can be defined to be the same as the distribution group of the parent array `a`. On the other hand the example of figure 4.1 is now problematic. This was constructed using a scalar subscript, effectively as follows:

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

float [[]] b = a [[0, :]]
```

The single range of `b` is clearly `y`, but identifying the distribution group of `b` with that of `a` doesn't seem to be right. If a one dimensional array is newly constructed with range `y` and distribution group `p`, as follows,

```
float [[]] bnew = new float [[y]] on p ;
```

it is replicated over the first dimension of p. The section b clearly isn't replicated in this way. Where does the information that b is localized to the top row of processes go?

Triplet section subscripts motivated us to define subranges as a new kind of range. Likewise, scalar section subscripts will drive us to define a new kind of group. A *restricted group* is defined as the subset of processes in some parent group to which a particular location is mapped. In the current example, the distribution group of b is defined to be the subset of processes in p to which the location x [0] is mapped. Instead of further extending subscripting notations to describe these subgroups, the division operator is overloaded. The distribution group of b is equivalent to q, defined by

```
Group q = p / x [0] ;
```

The expression in the initializer is called a *group restriction* operation[2].

In a sense the definition of a restricted group is tacit in the definition of an abstract location. Without formally defining the idea, we used it implicitly in section 2.4. In Figure 2.6 of that section the set of processes with coordinates $(0,0)$, $(0,1)$ and $(0,2)$, to which location x[1] is mapped, can now be written as

```
p / x [1]
```

and the set with coordinates $(0,1)$ and $(1,1)$, to which y[4] is mapped, can be written as

```
p / y [4]
```

The intersection of these two—the group containing the single process with coordinates $(0,1)$—can be written as

```
p / x [1] / y [4]
```

or as

```
p / y [4] / x [1]
```

At first sight the definition of HPJava restricted groups may appear slightly arbitrary. One good way to argue that a language construct is "natural" is to demonstrate that it has a simple and efficient implementation. The subgroups introduced here have an attractively simple concrete representation. A restricted group is uniquely specified by its set of effective process dimensions and the identity of the *lead* process in the group—the process with coordinate zero relative to the dimensions effective in the group. The dimension set can be specified as a subset of the dimensions of the parent grid using a simple bitmask. The identity of the lead process can be specified through a single integer ranking the processes of the parent grid. So a general HPJava group can be

[2]Choice of an arithmetic operator rather than a subscripting notation for group restriction is not ad hoc. A subscripting notation would require information about the position of a particular process dimension in the list of dimensions of the group. Experience shows that this positional information is often not fixed "statically"—the dimension may vary from invocation to invocation of a piece of code.

parametrized by a reference to the parent `Procs` object, with the addition of just two `int` fields. It turns out that this representation is not only compact, but also lends itself to efficient computation of the most commonly used operations on groups.

Now we can give a formal definition of the mapping (distribution group and ranges) of a general array section. As a matter of definition an integer subscript n in dimension r of array a is equivalent to a location-valued subscript a`.rng(`r`)[`n`]`. By definition, a triplet subscript l`:`u`:`s in the same dimension is equivalent to range-valued subscript[3] a`.rng(`r`)[`l`:`u`:`s`]`.

Suppose all integer and triplet subscripts are replaced by their equivalent location or range subscripts as just described. If the location subscripts are $i, j, \ldots$ the distribution group of the section is

```
p / i / j / ...
```

where p is the distribution group of the parent array. The sth range of the section is equal to the sth range-valued subscript.

Note that for a shifted location, as a matter of definition,

```
p / (i ± expression)  =  p / i
```

This makes sense—a shifted location is supposed to find an array element in the same process as the original location, albeit that the element could be in a ghost region.

It shouldn't come as a surprise that subranges and restricted groups can be used in array constructors, on the same footing as the ranges and groups described in earlier sections. This means, for example, that temporary arrays can be constructed with identical mapping to any given section. This facility is useful when writing generic library functions, such as the `matmul` of Figure 3.15, which must accept full arrays or array sections indiscriminately[4].

Unusually, this subsection introduced a couple of new pieces of syntax but did not give any full example programs that use them. The reason is that restricted groups and subranges largely exist "below the surface" in HPJava. The new notations are mainly needed to add a kind of semantic completeness to the language. It is not especially common to see subgroups or subranges constructed explicitly in HPJava programs.

4.5 Array restriction

Library functions operating on distributed arrays often specify certain *alignment relations* between their array arguments. Two arrays are *aligned* if they have the same distribution group and the same ranges[5]. The Adlib member `dotProduct`,

[3] Use of ranges as section subscripts was not mentioned before because it is not very useful in practise. It is allowed, for symmetry with integer and location subscripts.

[4] It also allows HPJava arrays to reproduce the full panoply of alignment options supported by the `ALIGN` directive of High Performance Fortran.

[5] Later we will give more detailed definitions.

for example, takes two distributed array arguments. These arguments must be aligned.

Occasionally it happens that two arrays we want to pass as arguments to a library function are *essentially* aligned, but one is replicated over a particular process dimension and the other isn't. It may be intuitively obvious that all the data needed by the function is in the right place, but still we cannot call the function—the ranges may match, but the replicated array has a larger distribution group. By the definition given above the arrays are not *identically* aligned.

One possibility is to relax the definition of argument alignment to take account of this situation. Anyone writing a library callable from HPJava is free to take this path—they can write their functions to accept arrays with some weaker alignment constraint. But experience suggests that the simple definition of alignment given above is easy to understand, and the specification and implementation of functions will be simpler if thay are based on this definition.

A minor extension to the HPJava language takes care of this situation. The restriction operation introduced for groups in the previous section can also be applied to an array. It returns a new array object—akin to an array section—which has the same ranges as the parent array, but has its group restricted by the specified location. Applied to a replicated array, it returns an array object referencing only the copies of the elements held in the restricted group.

Figure 4.6 is a generalization of the matrix multiplication program in Figure 3.13 to the case where the arrays are suitably distributed over a 3-dimensional process grid. Note that array `c` is replicated over the process dimension of `z`, `a` is replicated over the dimension of `y`, and `b` is replicated over the dimension of `x`. The sequential inner loop of Figure 3.13 is replaced by a call to `dotProduct` which directly forms the inner product of two sections with distributed range `z`.

If we didn't know about array restriction we would probably try to write the loop body as

```
c [i, j] = Adlib.dotProduct(a [[i, :]], b [[:, j]]) ;
```

The trouble is that according to the rules of the previous section the first argument of `dotProduct` has distribution group `p / i` whereas the second has distribution group `p / j`. So the arrays are not identically aligned. By forming restricted versions of both these sections we reduce both groups down to `p / i / j`. This, felicitously, is exactly the home group of the array element `c [i, j]`. The program will work correctly.

This is the first example we have given of a call to a collective library function *inside* the parallel overall construct. The library, Adlib, supports this kind of "nested parallelism" provided certain precautions are observed. These will be explained in section 6.

4.6 Scalars

We imposed no restriction that the list of subscripts in an array section expression *must* include some triplets (or ranges). It is legitimate for all the subscript

```
Procs3 p = new Procs3(P, P, P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;
  Range z = new BlockRange(N, p.dim(2)) ;

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

  float [[,]] a = new float [[x, z]] ;
  float [[,]] b = new float [[z, y]] ;

  ... initialize 'a', 'b'

  overall(i = x for :)
    overall(j = y for :)
      c [i, j] = Adlib.dotProduct(a [[i, :]] / j, b [[:, j]] / i) ;
}
```

Figure 4.6: A maximally parallel matrix multiplication program.

to be "scalar". In this case the resulting "array" has rank 0.

There is nothing pathological about rank-0 arrays. They logically maintain a single element, bundled with the distribution group over which this element is replicated. Because they are logically distributed arrays they can be passed as arguments to Adlib functions such as `remap`. If `a` and `b` are distributed arrays, we cannot usually write a statement like

```
a [10, 10] = b [30] ;
```

because the elements involved are generally held on different processors. As we have seen, HPJava imposes constraints that forbid this kind of direct assignment between array element references. However, we can usually achieve the same effect by writing

```
Adlib.remap(a [[10, 10]], b [[30]]);
```

The arguments are rank-0 sections holding just the destination and source elements.

There is a trivial syntactic problem with rank-0 arrays: the usual form of type signature for distributed arrays does not generalize down to the zero-dimensional case. The type signature for a one-dimensional distributed array typically has one empty slot enclosed in double brackets; there is no way to write zero empty slots! The symbol `#` is used as a degnerate form of the double brackets, and the type signature for a rank-0 array with element of type T is written T `#`.

The story of subranges and restricted groups repeats itself. The operation of array sectioning drives us to introduce a new kind of object into the language. Once that happens we should have a syntax for creating the new kind of object directly. Rank-0 distributed arrays, which we will also call simply "scalars", can be created as follows

```
float # c = new float # ;

float a [[,]] = new float [[x, y]] ;

Adlib.remap(c, a [[10, 10]]) ;

float d = c [] ;
```

This example illustrates one way to broadcast an element of a distributed array: remap it to a scalar replicated over the active process group. The element of the scalar is extracted through a distributed array element reference with an empty subscript list. As for other kinds of distributed array, scalar constructors can have `on` clauses, specifying a non-default distribution group.

Chapter 5

Some rules and definitions

In the preceding chapter, somewhat in passing, we completed the definition of the HPJava process group by adding the idea of a *restricted group* to the earlier idea of a process grid. This development has important applications to the basic distributed control constructs of the HPJava language.

5.1 Rules for distributed control constructs

In earlier sections we sometimes referred informally to the "active process group". A concrete role of this group was as the default distribution target in distributed array constructors. We used the fact that the on construct establishes its parameter group as the active process group inside the body of the construct.

The other distributed control constructs, at and overall, also affect the active process group. If the current active group is `p`, executing the construct construct

```
at(i = x [n]) {
  ...
}
```

or

```
overall(i = x for l : u : s) {
  ...
}
```

will change the active group to `p / i` inside the bodies of the constructs.

Now the expression `p / i` is only well defined if the location `i` belongs to a range distributed over a dimension of `p`[1]. So we infer that the control constructs given above can only appear at a point in the program where the dimension set of the active process group includes the process dimension over which `x` is distributed.

Note that the the dimension set of `p / x [n]` certainly *does not* include the process dimension associated with `x`—this follows directly from the nature of

[1]Alternatively `x` can be a collapsed range, in which case `p / i` is defined to be equal to `p`.

restriction operation. So one of the implications of the rule just given is that we should never expect to see exactly the two constructs above nested thus:

```
at(i = x [n])
  overall(i = x for l : u : s) {     // error!
    ...
  }
```

This is just as well, because the outer construct already restricts control to a single coordinate value, and it surely doesn't make sense to try distributing control across all coordinates of the same process dimension *inside* that construct[2].

Although it wasn't explicitly stated before, there is a similar restriction for the on construct. A precondition for appearance of the construct

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

is that the group `p` is contained in the group active in the immediately surrounding context.

5.2 Rules for distributed array constructors

The distributed array constructor expression

$$\texttt{new } T\texttt{[[}e_0\texttt{, ..., }e_r\texttt{, ...]] on } p$$

can only appear in a context where the distribution group, p, is contained in the currently active process group. If the "`on` p" clause is omitted, we identify the distribution group, p, with the active process group. If e_r is a (non-collapsed) range object, it must be distributed over a process dimension contained in the dimension set of p. No two range objects in $e_0, \ldots, e_r, \ldots$ can be distributed over the same process dimension.

5.3 Rules for accessing array elements

First we summarize rules for distributed array element references given in section 2.4. If a is a distributed array, then in the element reference

$$a\texttt{[}e_0\texttt{, ..., }e_r\texttt{, ...]}$$

the expression e_r may an integer expression (allowed only if the corresponding dimension of a has the sequential attribute), or a bound location. If it is an

[2]Technically this nesting is legal (but pointless) if `x` is collapsed.

integer, it must lie in the interval $0, \ldots, a$`.rng(`r`).size()`-1. If it is a location, it must be an element of the range a`.rng(`r`)`[3].

The rules on subscripts given in the previous paragraph go a long way towards ensuring a crucial requirement of HPJava, namely that a process may only access locally held array elements. Unfortunately there are still odd cases—typically involving array sections—where those rules are insufficient. Consider this pathological example

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

  float [[,]] a = new float [[x, y]] ;

  float [[]] b = a [[0, :]] ;

  at(i = x [N - 1])
    overall(j = y for :)
      b [j] = j` ;              // error!
}
```

The subscripts on the element reference `b [j]` are legal—`j` is certainly a location in `b.rng(0)` (which is equal to `y`). But, as illustrated in Figure 5.1, the section `b` is localized to `p / x [0]`—the top row of processes in the figure—whereas the at construct specifies that the element assignments are performed in the group `p / x [N - 1]`—the bottom row of processes.

This kind of error can be excluded by the following rule. Suppose the distribution group of a is p, and the list of subscripts $e_0, \ldots, e_r, \ldots$ in the element reference

$$a\texttt{[}e_0\texttt{, ..., }e_r\texttt{, ...]}$$

includes locations `{i, j, ...}`. The *home group* of the array element is defined to be

```
p / i / j / ...
```

(If the array has a replicated distribution this group may contain several processes; otherwise it contains a single process.) The general rule about accessing array elements can be stated formally as follows

[3]This last statement needs some interpretation, because locations in certain ranges may be identified with locations in others. For example, locations in a subrange will be identified with the matching locations of the parent range. In fact it is possible for two independently created ranges to be considered "aligned", in which case their locations will be identified. In general this will happen if the two ranges are are distributed over exactly the same process dimension and they have sufficiently similar distribution formats. "Sufficiently similar" usually means the distribution formats should be structurally identical, but their is even some leeway here. In particular locations in an `ExtBlockRange` can be identified with the corresponding locations of a `BlockRange` if the ranges have the same extent and process dimension. The `Range` class has includes methods such as `isAligned` that can be used to determine if two ranges are aligned, and thus logically share locations.

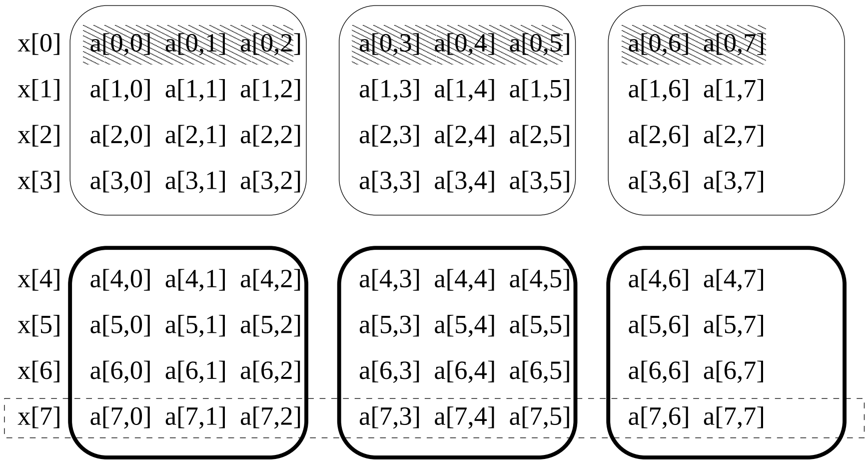

Figure 5.1: Access error discussed in text. Section `b` is shaded area at top.

> An array element can only be accessed if the active process group is contained in the home group of the element.

Informally this simply says that any process involved (active) at the point the array element is accessed must hold a copy of the array element.

The error above is now exposed, because the home group of `b [j]` is `p / x [0] / j`. This does not contain the active process group inside the overall construct, which is `p / x [N - 1] / j`. This kind of error can be trapped by runtime checks in the generated code. Very often this kind of check can be lifted out of innermost overall loops, leading to minimal overhead. In the current example a check that `p / x [0]` contains the active process group, placed immediately before entering the overall construct, would suffice.

5.4 Recommendations for updating variables

The definition of the *home group* of a distributed array element can be generalized to other kinds of variable: the home group of a variable (not a distributed array element) is the active process group at the point where the variable is declared.

A good rule of thumb for updating variables in general is

> A variable should only be updated when the active process group is identical to the home group of the variable.

If this rule is followed rigorously throughout a program (and if different processes only ever diverge in behaviour through dependencies on values of global indexes in overall constructs, or values of locally held program variables) it has the agreeable consequence that all variables remain *coherent*. A variable is coherent if, at corresponding stages of execution of an SPMD program, all processes in the home group of a variable always hold identical values in their local copies of the variable.

There are many places in HPJava programs where variables are *required* to be coherent. This is particularly true of arguments to collective operations. There are other places where it can be convenient to relax the coherence rule, which is why it is only advisory (also because it is relatively expensive to enforce this rule by runtime checks).

The style of programming in which all variables are held coherent is called the *canonical HPspmd style*. Out of the examples given so far in this article, the only one that doesn't follow canonical style is the Monte Carlo program of Figure 3.10. In that program the variable `rand` has home group `p`, but it is updated inside nested overall constructs, where the active process group is `p / i / j` (also, the initialization of `rand` involves a dependency on the `crd` method of `Dimension`, which is intrinsically incoherent). All other variables in all other examples are coherent[4]. The canonical HPspmd style has an affinity with the pure data parallel programming style of languages like HPF.

[4]Actually there is a short example in section 2.1 that uses `crd` inquiry, and therefore isn't canonical.

Chapter 6

A distributed array communication library

Many of the examples in this article use a communication library called Adlib. This library is not supposed to have a particularly unique status so far as the HPJava language definition is concerned. Adlib was developed independently of the HPJava project, to support HPF translation. Eventually HPJava bindings for other communication libraries will be needed. For example, the Adlib library does not provide the one-sided communication functionality of libraries like the Global Arrays toolkit; it doesn't provide optimized numerical operations on distributed arrays like those in ScaLAPACK; neither does it provide highly optimized collective operations for irregular patterns of array access, like those in CHAOS. All these libraries (and others) work with distributed arrays more or less similar to HPJava distributed arrays. We hope that bindings to these libraries, or functionally similar libraries, will be made available in HPJava. For now, this section summarizes essential features of the HPJava binding to Adlib.

6.1 Regular collective communications

There are three main families of collective operation in Adlib: regular communications, reduction operations, and irregular communications.

The regular communications are exemplified the operations `shift`, `cshift`, `writeHalo` and `remap`, introduced in earlier sections. The last of these, `remap`, is a very characteristic example. The `remap` function takes two distributed array arguments—a source array and a destination. These two arrays must have the same size and shape[1] but they can have any, unrelated, distribution formats. The effect of the operation is to copy the values of the elements in the source array to the corresponding elements in the destination array, performing any

[1]The *shape* of a distributed array is the list of its extents, `(a.rng(0).size(), a.rng(1).size(), ...)`.

communications required to do that. If the destination array has *replicated* mapping, the `remap` operation will broadcast source values to all copies of the destination array elements.

The `remap` function is a static member of the `Adlib` class. Like most of the functions in Adlib, the *remap* function is overloaded to apply to various ranks and types of array:

```
void remap(int    [[]] destination, int    [[]] source) ;
void remap(float  [[]] destination, float  [[]] source) ;
void remap(double [[]] destination, double [[]] source) ;
...
void remap(int    [[,]] destination, int    [[,]] source) ;
void remap(float  [[,]] destination, float  [[,]] source) ;
void remap(double [[,]] destination, double [[,]] source) ;
...
void remap(int    [[,,]] destination, int    [[,,]] source) ;
void remap(float  [[,,]] destination, float  [[,,]] source) ;
void remap(double [[,,]] destination, double [[,,]] source) ;
...
```

and scalars:

```
void remap(int    # destination, int    # source) ;
void remap(float  # destination, float  # source) ;
void remap(double # destination, double # source) ;
...
```

Currently the element-type overloading includes all Java *primitive* types. Later Adlib will be extended to support `Object` types.

There are four preconditions for a call to `remap`:

1. All processes in the active process group must make the call, and they must pass coherent arguments—in other words, for each argument, all processes pass local references to logically the same distributed array.

2. As mentioned above, the source and destination arrays should have the same shape and element types.

3. The arrays `source` and `destination` must not overlap—no element of `source` must be an alias for an element of `destination`. This is only an issue if both arguments are sections of the same array.

4. Both arguments must be *fully contained*.

By definition, an array is "fully contained" if its distribution group is contained in the active process group. So the requirement is that every copy of every element of the array is held on one of the processors engaging in the collective operation.

Most of the functions in Adlib have a similar set of preconditions—all operations are called collectively with coherent arguments, input and output arrays should never overlap, and array arguments must always be fully contained in the active group. The last requirement is probably the easiest to overlook. Consider the example of section 3.3, Figure 3.15. An easy mistake would be to put the calls to `remap` *inside* the following `on` construct. This is an error, because there is no guarantee that distribution groups of `a` and `b` are contained in the distribution group, `p`, of `c`. The function `matmul` is supposed to work for arguments with any, unrelated, distribution format. The Adlib library includes runtime checks for containment of arrays. If an argument is not fully contained, an exception occurs.

So long as the rule on containment is observed, Adlib calls can be made freely inside distributed control constructs, including the parallel loop, overall. If, for example, we want to "skew" an array—shift rows in the `y` direction by an amount that depends on the `x` index—we can do something like

```
on(p) {
  int [[,]] a = new int [[x, y]], b = new int [[x, y]] ;

  overall(i = x for :)
    Adlib.shift(b [[i, :]], a [[i, :]], i') ;
}
```

The section arguments of `shift` have distribution group `p / i`, which is identical to the active process group at this point, so the arguments are fully contained. A slightly more complicated example involving `dotProduct` was given earlier in section 4.5, Figure 4.6.

A prototype of the `shift` function was given in section 2.3. In general we have

```
void shift(T [[]] destination, T [[]] source,
           int shiftAmount) ;
void shift(T [[,]] destination, T [[,]] source,
           int shiftAmount, int dimension) ;
void shift(T [[,,]] destination, T [[,,]] source,
           int shiftAmount, int dimension) ;
...
```

where T stands for any primitive type of Java. The `dimension` argument is in the range $0, \ldots, R-1$ where R is the rank of the arrays. It selects the array dimension in which the shift occurs. The `shiftAmount` argument, which may be negative, specifies the amount and direction of the shift. Again the source and destination arrays must have the same shape, but now there is an extra precondition—they must also be identically aligned. That is, their distribution groups must be identical and all their ranges must be identical or satisfy the `isAligned` test. By design, `shift` implements a simpler pattern of communication than general `remap`. The alignment relation allows a more efficient implementation. The library includes runtime checks on alignment relations between arguments, where these are required.

The `shift` operation discards values from `source` that are moved past the edge of `destination`. At the other end of the range, elements of `destination` that are not targetted by elements from `source` are unchanged from their input value. The operation `cshift` is essentially identical to `shift` except that it implements a circular shift.

The function `writeHalo` is applied to distributed arrays that have ghost regions. It updates those regions. The simplest versions have prototypes

```
void writeHalo(T [[]] a) ;
void writeHalo(T [[,]] a) ;
void writeHalo(T [[,,]] a) ;
...
```

We can 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 `writeHalo` is to overwrite the ghost region with values from processes holding the corresponding elements in their physical segments.

A more general form of `writeHalo` allows to specify that only a subset of the available ghost area is to be updated, and to select circular wraparound for updating ghost cells at the extreme ends of the array, if desired.

```
void writeHalo(T [[]] a, int wlo [], int whi [], int [] mode) ;
void writeHalo(T [[,]] a, int wlo [], int whi [], int [] mode) ;
void writeHalo(T [[,,]] a, int wlo [], int whi [], int [] mode) ;
...
```

The three integer vectors are all of length R, the rank of `a`. The first two specify the widths at upper and lower ends of the bands to be updated (these values must be less than or equal to the widths of the actual ghost areas on the array). The elements of `mode` define for each dimension whether to update in the normal way, leaving ghost edges at extreme edges of the arrays unwritten (value should be `WriteHalo.EDGE`), whether to update using circular wraparound (`WriteHallo.CYCL`), or whether to not update any ghost regions in this dimension at all (`WriteHallo.NONE`, equivalent to setting the corresponding elements of `wlo`, `whi` to zero).

Operation of `writeHalo` is visualized in figure 6.1.

[Need to do something about `Adlib.copy`*. Probably this should be moved into a standard HPJava library.]*

6.2 Reductions

Reduction operations take one or more distributed arrays as input. They combine the elements to produce one or more scalar values, or arrays of lower rank. Adlib provides a large set of reduction operations, supporting the many kinds of reduction available in as "intrinsic functions" in Fortran. Here we mention only a few of the simplest reductions.

The `maxval` operation simply returns the maximum of all elements of an array. It has prototypes

```
T maxval(T [[]] a) ;
T maxval(T [[,]] a) ;
T maxval(T [[,,]] a) ;
...
```

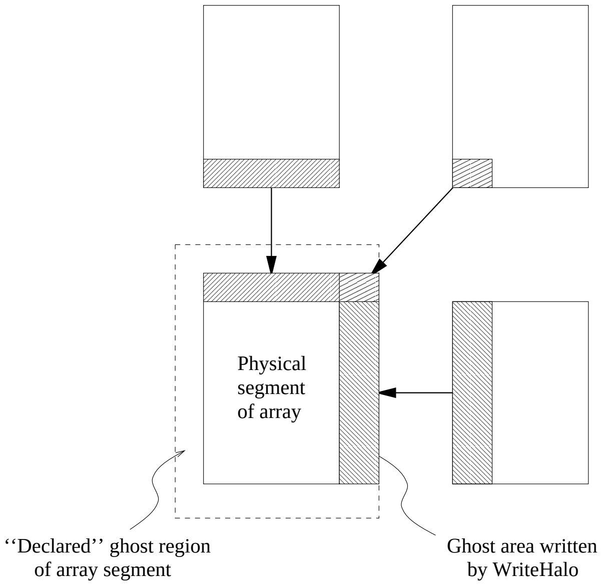

Figure 6.1: Illustration of the effect of executing the `writeHalo` function.

The result is broadcast to the active process group, and returned by the function. Other reduction operations with similar interfaces are `minval`, `sum` and `product`. Of these `minval` is minimum value, `sum` adds the elements of `a` in an unspecified order, and `product` multiplies them.

The function `dotProduct` used in some earlier examples is also logically a reduction, but it takes two one-dimensional arrays as arguments and returns their scalar product—the sum of pairwise products of elements. The situation with element types is complicated because the types of the two arguments needn't be identical. If they are different, standard Java binary numeric promotions are applied—for example if the dot product of an `int` array with a `float` array is a `float` value. Some of the prototypes are

```
int    dotProduct(int   [[]] a, int    [[]] b) ;
float  dotProduct(int   [[]] a, float  [[]] b) ;
double dotProduct(int   [[]] a, double [[]] b) ;
float  dotProduct(float [[]] a, int    [[]] b) ;
float  dotProduct(float [[]] a, float  [[]] b) ;
double dotProduct(float [[]] a, double [[]] b) ;
...
```

The arguments must have the same shape and must be aligned. As usual the result is broadcast to all members of the active process group.

The function `broacast` is not actually a reduction, but it has some features in common with other functions discussed in this section. The prototype is

```
T broadcast(T # s) ;
```

It takes a scalar (rank-0 distributed array) as argument and broadcasts the element value to all processes of the active process group. Typically it is used in conjunction with a scalar section to broadcast an element of a general array, as in this fragment:

```
int [[,]] a = new int [[x, y]] ;

int n = 3 + Adlib.broadcast(a [[10, 10]]) ;
```

6.3 Irregular collective communications

Adlib has some support for irregular communications in the form of collective `gather` and `scatter` operations. The simplest form of the gather operation for one-dimensional arrays has prototypes

```
void gather(T [[]] destination, T [[]] source, int [[]] subscripts) ;
```

The `subscripts` array should have the same shape as, and be aligned with, the `destination` array. In pseudocode, the `gather` operation is equivalent to

```
for all i in {0, ..., N − 1} in parallel do
  destination [i] = source [subscripts [i]] ;
```

where N is the size of the `destination` (and `subscripts`) array. If we are implementing a parallel algorithm that involves a stage like

```
for all i in {0, ..., N − 1} in parallel do
  a [i] = b [fun(i)] ;
```

where *fun* is an arbitrary function, it can be expressed in HPJava as

```
int [[]] tmp = new int [[x]] on p ;
on(p)
  overall(i = x for :)
    tmp [i] = fun(i) ;

Adlib.gather(a, b, tmp) ;
```

where `p` and `x` are the distribution group and range of `a`. The source array may have a completely unrelated mapping.

The one-dimensional case generalizes to give a rather complicated family of prototypes for multidimensional arrays:

```
void gather(T [[]] destination, T [[]] source,
            int [[]] subscripts) ;
void gather(T [[,]] destination, T [[]] source,
            int [[,]] subscripts) ;
void gather(T [[,,]] destination, T [[]] source,
            int [[,,]] subscripts) ;
...
void gather(T [[]] destination, T [[,]] source,
            int [[]] subscripts1, int [[]] subscripts2) ;
```

```
void gather(T [[,]] destination, T [[,]] source,
            int [[,]] subscripts1, int [[,]] subscripts2) ;
void gather(T [[,,]] destination, T [[,]] source,
            int [[,,]] subscripts1, int [[,,]] subscripts2) ;
...
...
```

The complexity arises because now that the source and destination arrays can have different ranks. The pattern is that the subscript arrays have the same and alignment shape as the destination arrays. The *number* of subscript arrays is equal to the rank of the source array. As an example, the last of the prototypes enumerated above behaves like

```
for all i in {0,...,L-1} in parallel do
  for all j in {0,...,M-1} in parallel do
     for all k in {0,...,N-1} in parallel do
       destination [i, j, k] = source [subscripts1 [i, j, k],
                                       subscripts2 [i, j, k]] ;
```

where (L, M, N) is the shape of `destination` array.

The basic `scatter` function has very similar prototypes, but the names `source` and `destination` are switched. The one-dimensional case is

```
void scatter(T [[]] source, T [[]] destination,
             int [[]] subscripts) ;
```

and it behaves like

```
for all i in {0,...,N-1} in parallel do
  destination [subscripts [i]] = source [i] ;
```

6.4 Schedules

In general the collective communication functions introduced in the last few sections involve two phases: an *inspector* phase in which the arguments are analysed to determine what communications and local copies will be needed to complete the operation, and an *executor* phase in which the schedule of these data transfers is actually performed. In iterative algorithms, it often happens that exactly the same communication pattern is repeated many times over. In this case it is wasteful to repeat the inspector phase in every iteration, because the data transfer schedule will be the same every time.

Adlib provides a class of *schedule objects* for each of its communication functions. The classes generally have the same names as the static functions, with the first letter capitalized (the name may also be extended with a result type). Each class has a series of constructors with arguments identical to the instances of the function. Every schedule class has one public method with no arguments called `execute`, which executes the schedule.

Using `WriteHalo` and `Maxval` schedules, the main loop of the red-black relaxation program from section 3.2, Figure 3.8 could be rewritten as in Figure 6.2.

```
WriteHalo writeHalo = new WriteHalo(u) ;
MaxvalFloat maxval = new MaxvalFloat(r) ;

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

    writeHalo.execute() ;

    overall(i = x for 1 : N - 2)
      overall(j = y for 1 + (i` + parity) % 2 : N - 2 : 2) {
        float newA ;

        newA = 0.25 * (a [i - 1, j] + a [i + 1, j] +
                       a [i, j - 1] + a [i, j + 1]) ;

        r [i, j] = Math.abs(newA - a [i, j]) ;
        a [i, j] = newA ;
      }
  }

} while(maxval.execute() > EPS)) ;
```

Figure 6.2: Red-black relaxation, re-using communication schedules.