Parallel Programming in HPJava

Bryan Carpenter, Guansong Zhang, Han-Ku Lee, Sang Boem Lim, ...

School of Computational Science and Information Technology
Florida State Universtiy
Tallahassee, FL 32306-4120

June 11, 2001

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.

[HPJava is not *a parallelizing compiler. Before writing an HPJava program you must understand parallel algorithms. HPJava is a notation for expressing and implementing parallel algorithms.]*

Chapter 2

Processes and Arrays

2.1 Processes and Process Grids

An HPJava program is started concurrently in all members of some process set[1]. From the point of view of the HPJava program, the processes are organized and identified through special objects representing *process groups*. In general the processes in an HPJava group are arranged in multidimensional grids.

Suppose a program is running concurrently on 6 or more processes. It may then 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 grid `p` is visualized in Figure 2.1. This figure assumes that the program was executing in 11 processes. The call to the `Procs2` constructor selected 6 of these available processes and incorporated them into a grid.

The `Procs2` constructor is an example of a *collective operation*. It should be invoked concurrently by all members of the active process group.

`Procs2` is a subclass of the special class `Group`. The `Group` class has a privileged status in the HPJava language. An object that inherits this class can be used in various special places. For example, it can be used to parametrize an *on construct*.

After creating `p` we will probably want to perform some operations within the processes of the grid. An `on` construct restricts control to processes in its parameter group. So in

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

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

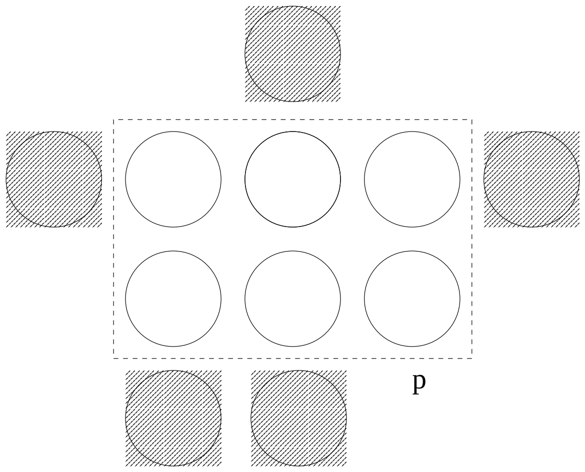


Figure 2.1: The process grid represented by `p`.

the code inside the construct is only executed by processes belonging to `p`. In Figure 2.1, the five processors outside the grid would skip this block of code.

This `on` construct establishes `p` as the *active process group* within its body. In several of the operations introduced later, the current active process group will act as a "default" group parameter.

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 are called *process dimensions*. The process dimensions of a grid are available through the inquiry method `dim(`r`)`. The argument r is in the range $0, \ldots, R-1$, where in general R is the rank (dimensionality) of the grid.

The `Dimension` class in turn has a method `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)
```

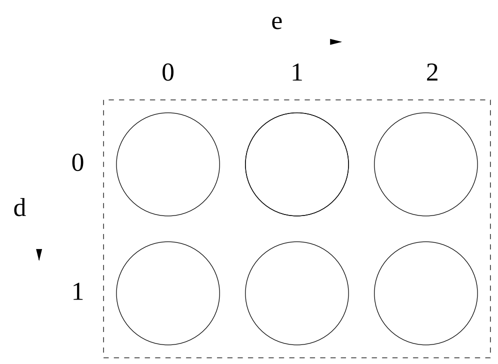


Figure 2.2: The process dimensions and coordinates in `p`.

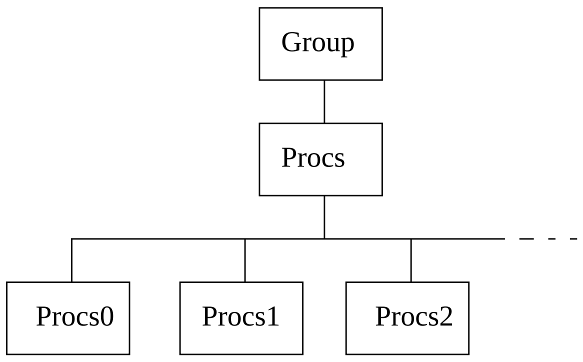


Figure 2.3: The `Group` hierachy of HPJava.

The dimensions of `p` are illustrated in Figure 2.2. 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 applied `crd()` to `d` or `e` in a process outside `p` (such as one of the shaded processes in Figure 2.1) we should expect an exception.

There is nothing special about 2-dimensional grids. The full `Group` hierarchy of HPJava includes the classes of Figure 2.3. Not surprisingly, `Procs1` is a one-dimensional process “grid”. Less obviously, `Procs0` is a group containing exactly one process. Higher dimensional grids are also allowed.

Note that so far the only special *syntax* we have added to the Java language is the `on` construct. The `Group` class has special status in HPJava[2], but syntactically it is just a class.

[2]Much as `Throwable` has a special status in normal Java—it is the only class that can parametrize a `catch` clause.

2.2 Distributed Arrays

The biggest new feature HPJava adds to Java is the *distributed array.* A distributed array is a *collective 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 similarities between the way HPJava distributed arrays are used and the way ordinary Java arrays are used. There are also a number of 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.

Bearing in mind these distinctions it does not seem a good idea to try and force the new HPJava arrays into a syntax directly reminiscent of ordinary Java arrays. Instead, HPJava distributed arrays look and feel quite different from standard arrays. Of course HPJava includes Java as a subset, and ordinary Java arrays can and should be used freely in an HPJava program. But they do not have a close relationship to the new distributed arrays.

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 another class that has a special status in 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 2.4. The choice of subclass `BlockRange` for the index ranges of the array means 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].

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

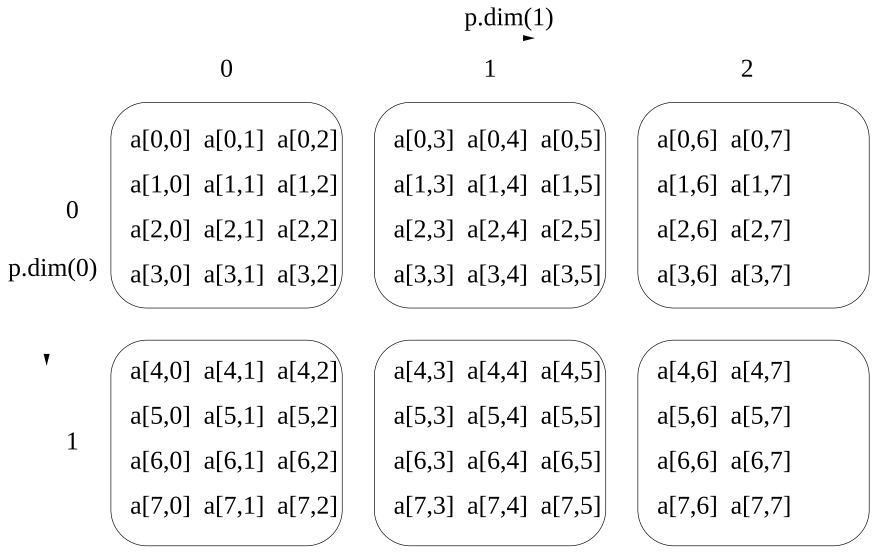

p.dim(1)
0
1
2
p.dim(0)
0
1
a[0,0] a[0,1] a[0,2] a[0,3] a[0,4] a[0,5] a[0,6] a[0,7]
a[1,0] a[1,1] a[1,2] a[1,3] a[1,4] a[1,5] a[1,6] a[1,7]
a[2,0] a[2,1] a[2,2] a[2,3] a[2,4] a[2,5] a[2,6] a[2,7]
a[3,0] a[3,1] a[3,2] a[3,3] a[3,4] a[3,5] a[3,6] a[3,7]
a[4,0] a[4,1] a[4,2] a[4,3] a[4,4] a[4,5] a[4,6] a[4,7]
a[5,0] a[5,1] a[5,2] a[5,3] a[5,4] a[5,5] a[5,6] a[5,7]
a[6,0] a[6,1] a[6,2] a[6,3] a[6,4] a[6,5] a[6,6] a[6,7]
a[7,0] a[7,1] a[7,2] a[7,3] a[7,4] a[7,5] a[7,6] a[7,7]

Figure 2.4: A two-dimensional array distributed over `p`.

2.3 Parallel Programming

The previous section included a simple example of how to create a distributed array. How do we use this kind of array? Figure 2.5 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)
```

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 :)
```

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

```
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 2.5: A parallel matrix addition.

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 2.6 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 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[6].

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

[6]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.

```
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(s, a,  1, 0) ;
  Adlib.shift(n, a, -1, 0) ;
  Adlib.shift(w, a,  1, 1) ;
  Adlib.shift(e, 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 2.6: A parallel stencil update program.

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` is called a *distributed index*. The value associated with a distributed index is a *location*, *not* an integer value.

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

Figure 2.7 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 distributed index—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). it is not allowed to write simply

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

because 1 and 4 are not distributed indices, 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 ;
```

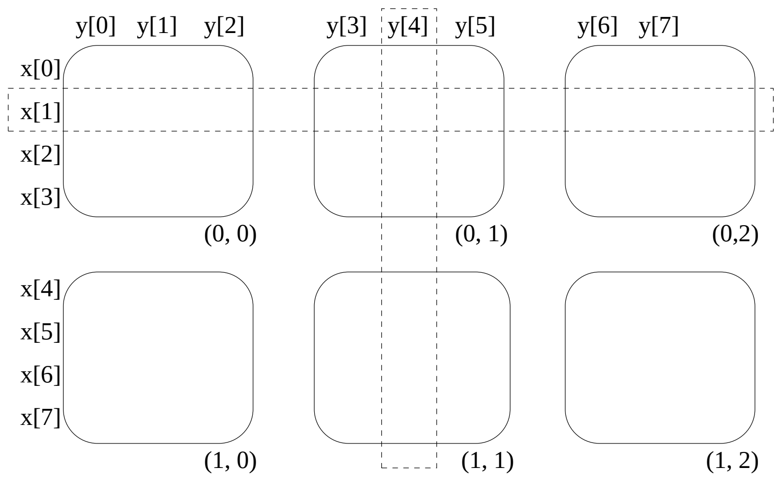


Figure 2.7: Mapping of `x` and `y` locations to the grid `p`.

The symbols `i` and `j`, scoped within the construct bodies, are distributed indices.

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 2.7, 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])
```

then restricts execution down to just process $(0,1)$. This is exactly the process that owns element `a[1,4]`. The odd restriction that subscripts must be distributed indices 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. If `s` is greater than zero, 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]) {
    ...
  }
```

If `s` is less than zero, it is equivalent to

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

The bodies of the `at` constructs are 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 distributed index, thus:

```
i`
```

This expression is read "i-primed", and 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 more complete examples.

2.5 A Complete Example

The example of Figure 2.8 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 distributed indices 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, composi-

[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.

```
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 2.8: Solution of Laplace equation by Jacobi relaxation.

tional 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
```

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 2.6 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 dimensions 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‘) ;
}
```

The point here is that the `overall` constructs only traverse half the ranges of the array they process. As shown in Figure 3.2, this leads to a very poor

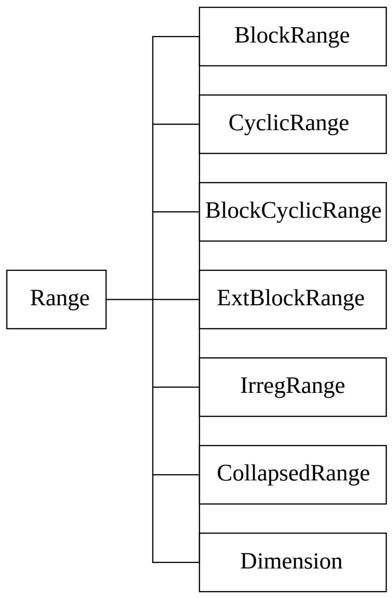


Figure 3.1: The `Range` hierachy of HPJava.

distribution of workload. The process with coordinates $(0, 0)$ does nearly all the work. The process at $(0, 1)$ 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 not nearly as 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 while leaving much of the code that processes them unchanged. HPJava does not guarantee this property in the same way that

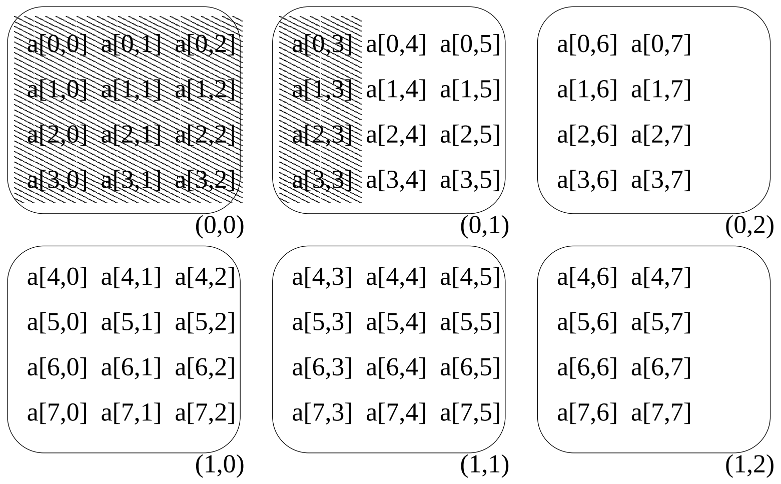


Figure 3.2: Work distribution for an example with block distribution.

HPF does, and fully optimized HPJava programs are unlikely to be so easily redistributed. But it is still a useful property.

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*.

[1]Notably ScaLAPACK.

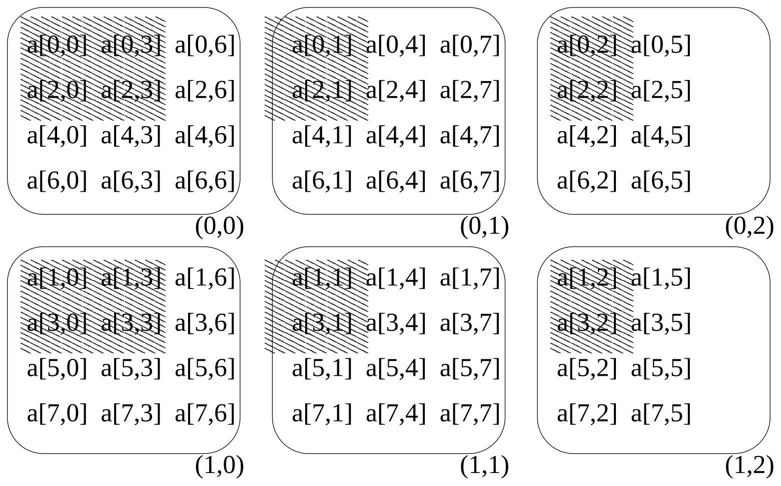


Figure 3.3: Work distribution for an example with cyclic distribution.

```
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.

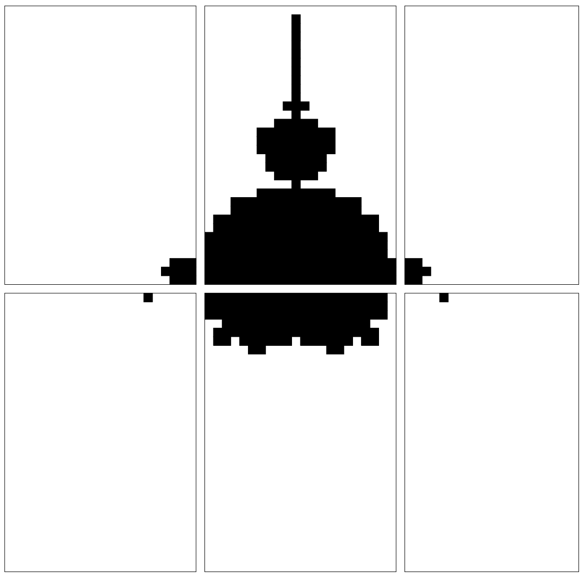


Figure 3.5: Blockwise decomposition of the Mandelbrot set (black region).

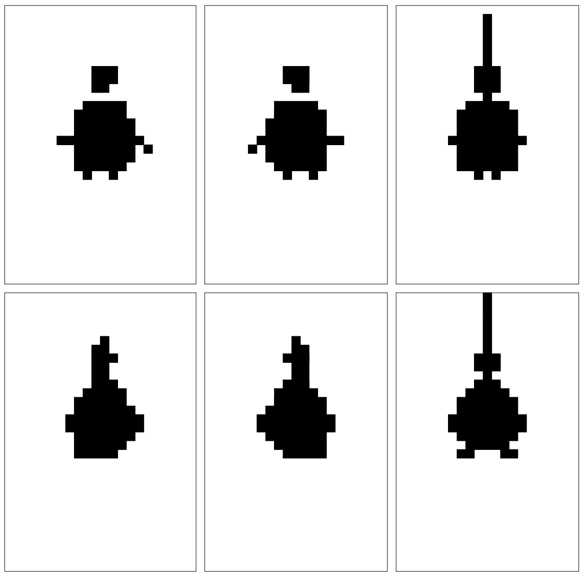


Figure 3.6: Cyclic decomposition of the Mandelbrot set.

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 distributed index is relaxed. In a special, slightly idiosyncratic, piece of syntax, the following expression

$$\textit{name} \pm \textit{expression}$$

is a legal subscript if *name* is a distributed index and *expression* is an integer expression—in practice usually a small constant. This is called a *shifted index.* The significance of the shifted index is that an element displaced from the original location will be accessed. If the shift puts the location outside the local block *plus* surrounding ghost region, an exception will occur. Using this syntax 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]) ;
```

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

Ghost regions are not magic. The values cached around the edges of a local block can only be made consistent with the values held in blocks on adjacent processes by a suitable communication. A library function called `writeHalo` updates the cached values 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 omitted code is unchanged from Figure 2.8. 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

[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 distributed index `i`. Let `j` be the location `x [i' + e]`. If `j` is mapped to the same processes as `i`, and the shifted index `i + e` is used as a subscript, it behaves just like the 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`.

elements between two *aligned* arrays (`copy` does not implement communication; if it is passed non-aligned arrays, an exception occurs).

As a matter of fact, if we removed the temporary, `b`, and reassigned the `a` elements on-the-fly in terms of the other partially updated `a` elements, 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.)

```
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 ;
      }

    HPspmd.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).

```
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.

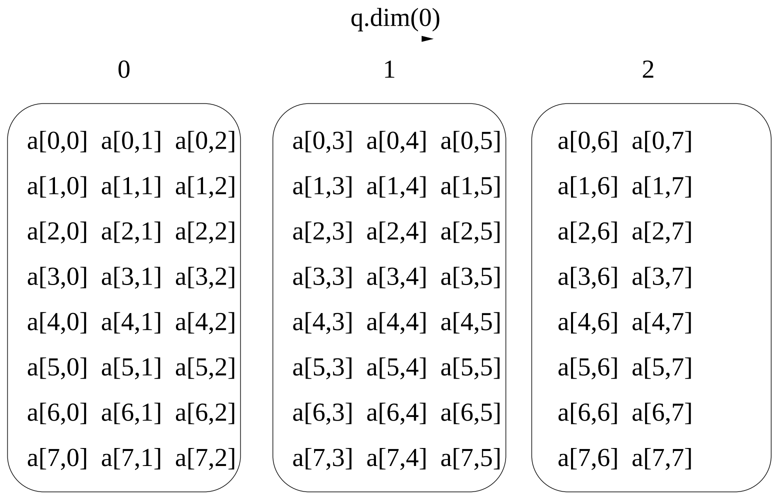


Figure 3.11: Two-dimensional array, `a`, distributed over the one-dimensional grid, `q`.

3.3 Collapsed Distributions and Sequential Dimensions

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".)

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

```
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 distributed indices, 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 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) ;
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 distributed indices 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,

```
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, 0) ;
    HPspmd.copy(b, tmp) ;
  }
}
```

Figure 3.12: A pipelined matrix multiplication program.

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.

3.4 Replication and Distribution Groups

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 whose rank is larger than the array.

```
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 require 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] ;
```

```
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.

The properties of `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.

In the last section we saw a “pipelined” matrix multiply algorithm. 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 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.

We would be very lucky to come across three arrays with such a special *alignment relation* (distribution format relative to one another). 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 and create a general purpose matrix multiplication routine. Besides the `remap` function, this example introduces the two inquiry methods `grp()` and

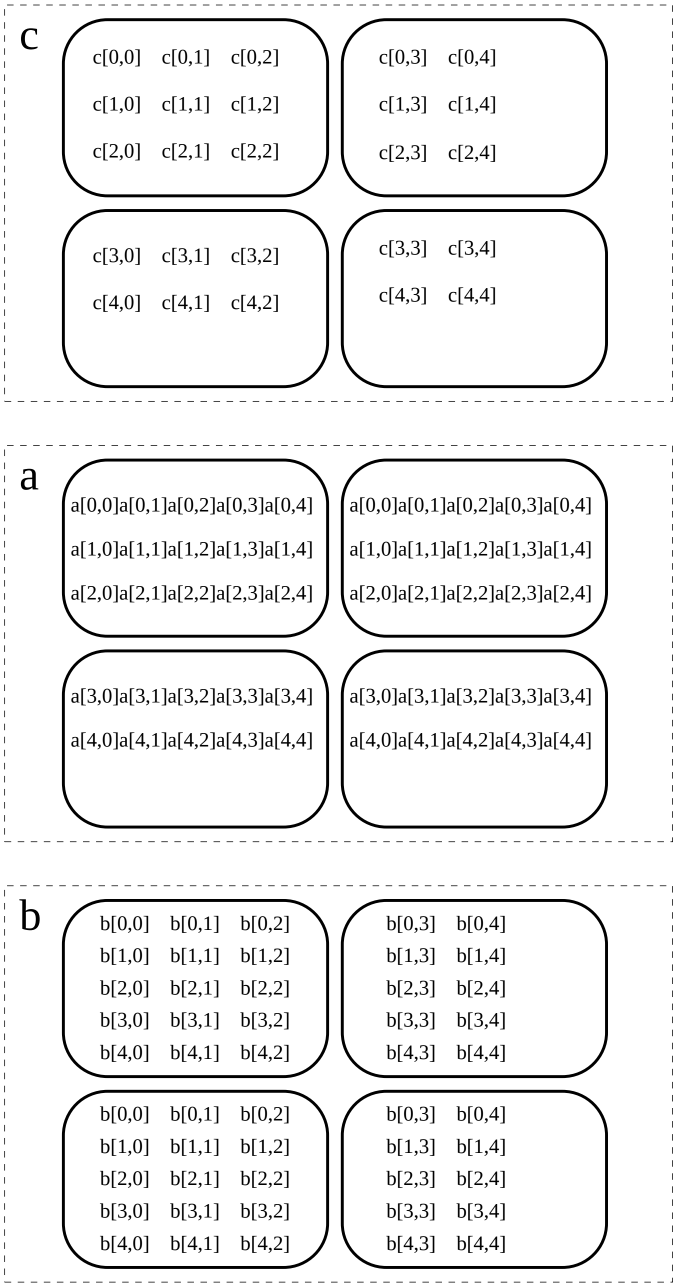


Figure 3.14: Distribution of array elements in example of Figure 3.13. Array `a` is replicated in every column of processes, array `b` is replicated in every row.

```
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.

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

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

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

on(p) {
  ... compute elements in 2d block-distributed ‘a’ ...
}


Procs1 q = new Procs1(Q) ;

Range z = new BlockRange(N, q.dim(0));

float [[-,*]] b = new float [[z, N]] on q;


Adlib.remap(b, a);  // copy elements of ‘a’ to row-distributed ‘b’

on(q) {
  ... process elements of ‘b’ ...
}
```

Figure 3.16: Sketch example exchanging data between different grids.

`rng()` which are defined for any distributed array. The inquiry `grp()` returns the distribution group of the array, and the inquiry `rng(`r`)` returns the rth range of the array. The argument r is in the range $0, \ldots, R-1$, where R is the rank (dimensionality) of the array.

The example also illustrates the most general form of the distributed array constructor. In all earlier examples arrays were distributed over the whole of the active process group, defined by an enclosing `on` construct. In general an *on clause* attached to an array constructor itself can specify that the array is distributed over some subset of the active group. This allows one to create an array outside the `on` construct that will processes its elements. Through communication functions like `remap`, values can then be exchanged between different process grids. A sketch example is given in Figure 3.16.

To allow for this kind of situation, where arguments might be distributed over distinct process groups—not the active process group—the generic matrix multiplication of Figure 3.15 included `on p` clauses in the constructors of its temporary arrays, and explicitly restricts control with an `on(p)` construct before processing. As we will see in the next section, this refinement also allows the arguments of `matmul` to be arbitrary array *sections*.

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 does not contain new elements. It 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. For each triplet subscript a section expression has an array dimension. So in a normal 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, :]]

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

[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

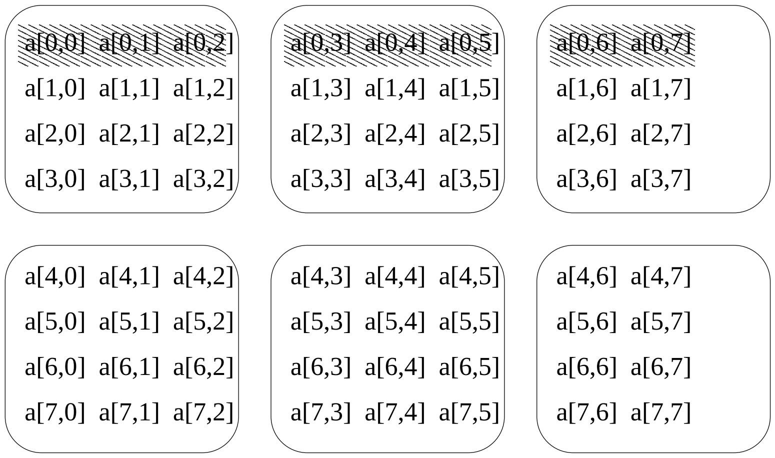


Figure 4.1: A one-dimensional section of a two-dimensional array (shaded area).

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). In an array section expression, *unlike* in an array element reference, a scalar subscript is always allowed to be an integer expression[2]. 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.

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 transformations 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 sketched 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 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.

[2]Distributed indices are allowed as well, if the location is in the proper range.

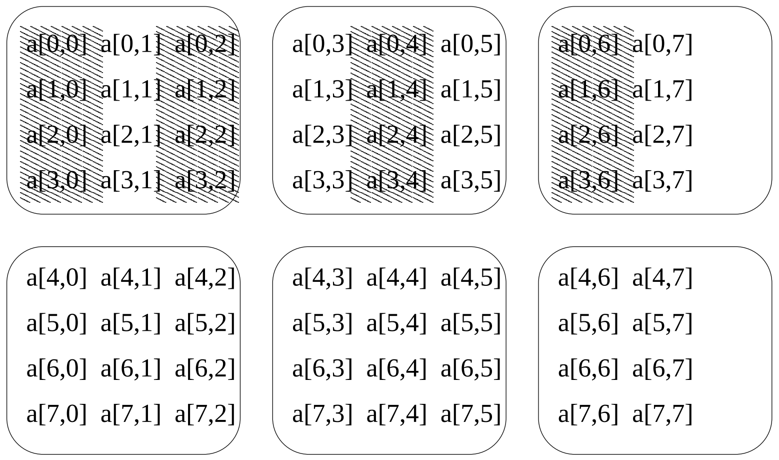


Figure 4.2: A two-dimensional section of a two-dimensional array (shaded area).

```
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.

4.2 Cholesky decomposition

If A is a symmetric positive definite matrix, associated linear equations are often solved 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

```
$For\ k = 1\ to\ n - 1$
  $l_{kk} = a_{kk}^{1/2}$
  $For\ s = k + 1\ to\ n$
    $l_{sk} = a_{sk}/l_{kk}$
  $For\ j = k + 1\ to\ n$
    $For\ i = j\ to\ n$
      $a_{ij} = a_{ij} - l_{ik}l_{jk}$
$l_{nn} = a_{nn}^{1/2}$
```

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

[Discuss the expression `b[j‘]` *—why it must be* `j‘`*.]*

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 arguments. 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 elements. It defines the largest block of locations of 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()
```

```
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: Cholesky decomposition.

```
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) {
    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.

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 becomes a credible basis for a library matrix multiplication routine, applicable to generic distributed arrays.

4.4 Subranges

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, 0 : N - 1 : 2]] ;
```

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

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 [0 : N - 1 : 2] ;
```

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

The global indices associated with the subrange v, for example, are in the range $0, \ldots, v$`.size()`. A subrange inherits locations from its parent range, but it specifically *does not* inherit global indices from the parent.

A *non-trivial subrange* is one for which the lower bound is not equal to zero, or the upper bound is not equal to $\texttt{size}() - 1$, or the stride is not equal to 1.

A non-trivial subrange is never considered to have ghost extensions, even if its parent range does. This avoids various ambiguities that might otherwise crop up.

What about the distribution groups of sections? Now 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`. But the example of figure 4.1 is 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`, like this:

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

it is understood to be 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?

4.5 Restricted Groups

In the last section 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 to be 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. Rather than 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.

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

Note that the inquiry function `dim()` is a member of the *Procs* class (the process grid class), *not* the superclass `Group`.

4.6 Mapping of array sections

Now we can give a formal definition of the mapping (distribution group and ranges) of a general array section.

If the rth dimension of array a is non-sequential, an integer section subscript, n, in this dimension behaves like a location-valued subscript, a`.rng(`r`)[`n`]`[3]. Suppose any such integer subscripts are replaced by their equivalent location subscripts in this way. If the set of all location subscripts is now $i, j, \ldots$, the distribution group of the section is

$$p/i/j/\ldots$$

where p is the distribution group of the parent array. For a shifted index, as a matter of definition,

$$\texttt{p / (i} \pm \textit{expression}\texttt{) = p / i}$$

This makes sense—a shifted index 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.

The kth range of the section is determined by the kth triplet-valued subscript. If the kth triplet-valued subscript is l`:`u`:`s in dimension r, the kth range of the section is a`.rng(`r`)[`l`:`u`:`s`]`.

Note that, because non-trivial subranges are never considered to have ghost extensions, a section constructed with non-trivial triplet subscripts in some dimensions is not be considered to have accessible ghost extensions in those dimensions, even if its parent array had them.

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

[3] If the rth dimension is sequential, this equivalence is not strict; an integer subscript in a sequential dimension may have extended bounds in the peculiar case that the sequential dimension has ghost extensions. But there are no locations associated with subscript values in ghost extensions.

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 without discrimination[4].

The last three sections were unusual in that they introduced some 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.

4.7 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 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, if it is really needed, 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.

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 [] ;
```

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

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. Like any other distributed array, scalar constructors can have `on` clauses, specifying a non-default distribution group.

4.8 Array restriction

Library functions operating on distributed arrays often specify certain *alignment relations* between their array arguments. In HPJava it is atural to define two arrays to be *aligned* if they have the same distribution group and all their ranges are aligned[5]. The Adlib member `dotProduct`, 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. But experience suggests that the simple definition of alignment given above is easy to understand, and the specification and implementation of library functions are 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

[5]Later we will give more detailed definitions.

```
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.

versions of both these sections we reduce both groups down to `p/i/j`. Luckily this is also the home group of the array element `c [i, j]`, so 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 a few precautions are taken. These will be explained in section 6.

Chapter 5

Some Rules and Definitions

In the preceding chapter 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 some useful 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". One 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 group argument 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—recall the discussion in section 2.4. We can use the notations introduced in the last section to state their effect more formally

First, for completeness, we restate the effect of the `on` construct and give an associated rule. As explained in section 2.1, the construct

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

changes the active group to `p` inside its body. The associated rule is:

Rule 1 *The construct*

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

can only appear at a point in the program where `p` *is contained in the active process group.*

In other words, an `on` construct cannot add any new process to the active group.

Now, 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.

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 can conclude that:

Rule 2 *Unless* `x` *is a collapsed range, a control construct*

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

or

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

can only appear at a point in the program where the process dimension of `x` *occurs in the dimension set of the active process group.*

As an example of the application of this rule, notice that the the dimension set of the restricted group p/x[n] certainly *does not* include the process dimension associated with `x`. So one of the implications of rule 2 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 good, 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].

5.2 Rules for distributed array constructors

There are several restrictions on distributed array constructor, which we will group together in:

[1] Alternatively `x` can be a collapsed range, in which case `p/i` is defined to be equal to `p`.
[2] Strictly speaking this nesting is legal (but pointless) if `x` is collapsed.

Rule 3 *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 e_r is a (non-collapsed) range object, its process dimension must belong to the dimension set of p. No two range objects in $e_0, \ldots, e_r, \ldots$ can be distributed over the same *process dimension of p.*

If the "`on` p" clause is omitted, we identify the distribution group, p, with the active process group, and the remaining conditions must still apply.

5.3 Rules for access to distributed array elements

First we will collect together rules for distributed array element references implied or informally stated in 2.4 and subsequent sections. The first rule is part of the static semantics of the language—it can be enforced by the type checker:

Rule 4 *If a is a distributed array, then in the element reference*

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

the expression e_r is either an integer expression—allowed only if the corresponding dimension of a has the sequential attribute—or a (possibly shifted) distributed index.

Depending on whether e_r is in fact an integer, an index declared in an `at` construct, or an index declared in an `overall` construct, exactly one of the following three "run-time"[3] rules applies. To simplify the discussion, we first ignore ghost regions. The three rules are: either

Rule 5 *In the distributed array element reference:*

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

if e_r is an integer, its value must lie in the range

$$0 \leq e_r < x\texttt{.size()}$$

where $x = a\texttt{.rng(}r\texttt{)}$.

or

[3]Of course there is nothing to prevent a compiler applying these checks at compile-time if it can do so.

Rule 6 *The distributed array element reference in:*

```
at(i = x [n]) {
  ... a[e_0, ..., e_{r−1}, i, e_{r+1}, ...]  ...
}
```

is allowed if and only if

1. *The expression* a *is invariant in the* `at` *construct*[4].

2. *The location* `x[n]` *is an element of the range* `a.rng(r)`[5].

or

Rule 7 *The distributed array element reference in:*

```
overall(i = x for l : u : s) {
  ... a[e_0, ..., e_{r−1}, i, e_{r+1}, ...]  ...
}
```

is allowed if and only if

1. *The expression* a *is invariant in the* `overall` *construct.*

2. *All locations in* `x[l:u:s]` *are of elements* `a.rng(r)`*.*

A subtle and important point to appreciate is that rules 6 and 7 are statements about the `at` and `overall` constructs as a whole, not about the array accesses in isolation. They apply unconditionally to any access that appears textually inside the constructs, even if some conditional test in the body of the construct might prevent those accesses from actually being executed. This is very important because it allows any associated run-time checking code to be lifted outside the constructs, and in particular to be lifted outside the local loops implied by an `overall`.

[4]The interpretation of "invariant" will be discussed further in section 8.2.1. Actually this condition is more important for the case of an index subscript declared by an `overall` (next rule). It is imposed here to preserve the semantic relation between `at` and `overall` specified in 2.4.

[5]This 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.

5.3.1 Changes for ghost regions

If the array a appearing in rule 5 has ghost regions[6], the range of allowed subscripts is changed to

$$-x\texttt{.loExtension()} \leq e_r < x\texttt{.size()} + x\texttt{.hiExtension()}$$

If the array appearing in rule 6 has ghost regions, the index subscript may be shifted:

```
at(i = x [n]) {
  ... a[e0, ... ,e(r-1),i ± d,e(r+1), ...]  ...
}
```

and the following requirement is added:

3. The expression $\pm d$ is in the range

$$-a\texttt{.rng(}r\texttt{).loExtension()} \leq \pm d \leq a\texttt{.rng(}r\texttt{).hiExtension()}$$

Rule 7 is modified in a completely analogous way if the array appearing there has ghost regions.

5.3.2 A final rule for array element access

The rules on subscripts given in the last two subsections go a long way towards ensuring a crucial requirement of HPJava, namely that a process may only access locally held array elements. There are still odd cases—typically involving array sections—where those rules are insufficient. Consider the pathological example of Figure 5.1. 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, 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:

Rule 8 *An element reference in an array a can only be made by a process that is contained in the distribution group a*`.grp()`.

The error above is now exposed, because the distribution group of `b` is `p/x[0]`. This does not contain the active process group inside the `overall` construct, namely `p/x[N-1]/j`. So the processes executing the array access fail to meet the criterion of rule 8.

5.4 A recommendation for updating variables

Suppose the distribution group of a is p, and the list of subscripts $e_0, \ldots, e_r, \ldots$ in the element reference

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

[6]See section 7.3.1 for an example of a sequential dimension with ghost regions.

```
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!
}
```

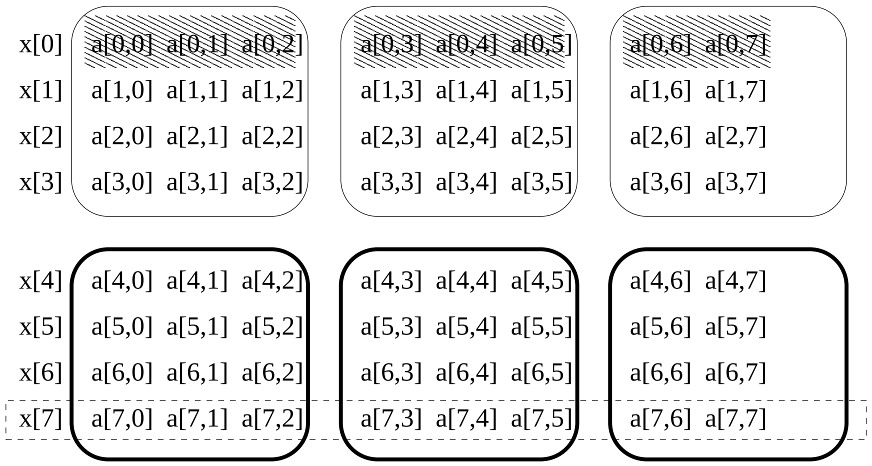

x[0]
x[1]
x[2]
x[3]
x[4]
x[5]
x[6]
x[7]
a[0,0] a[0,1] a[0,2] a[0,3] a[0,4] a[0,5] a[0,6] a[0,7]
a[1,0] a[1,1] a[1,2] a[1,3] a[1,4] a[1,5] a[1,6] a[1,7]
a[2,0] a[2,1] a[2,2] a[2,3] a[2,4] a[2,5] a[2,6] a[2,7]
a[3,0] a[3,1] a[3,2] a[3,3] a[3,4] a[3,5] a[3,6] a[3,7]
a[4,0] a[4,1] a[4,2] a[4,3] a[4,4] a[4,5] a[4,6] a[4,7]
a[5,0] a[5,1] a[5,2] a[5,3] a[5,4] a[5,5] a[5,6] a[5,7]
a[6,0] a[6,1] a[6,2] a[6,3] a[6,4] a[6,5] a[6,6] a[6,7]
a[7,0] a[7,1] a[7,2] a[7,3] a[7,4] a[7,5] a[7,6] a[7,7]

Figure 5.1: Access error discussed in text. Section `b` is shaded area at top. The at construct restricts control to the bottom row of processes.

includes locations $i, j, \ldots$, then the *home group* of the array element is defined to be

$$p/i/j/\ldots$$

If the array has a replicated distribution this group may contain several processes; otherwise it contains a single process.

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

A good rule of thumb for updating variables in general is

Recommendation 1 *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 interesting result 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 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 one reason why it is only advisory (another reason is that it is relatively expensive to enforce this rule by runtime checks).

We call the style of programming in which all variables are held coherent the *canonical HPspmd style*. Out of the examples given so far in this document, 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 algorithmic examples are coherent[7]. The canonical HPspmd style has a special affinity with the pure data parallel programming style of languages like HPF.

[7] There is a short example in section 2.1 that uses the `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 uniquely special 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; it does not 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—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* in the active process group.

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 3.4, 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 4.8, 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 as a shorthand 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 integer vectors are all of length R, the rank of the argument `a`. The values `wlo` and `whi` 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 `mode` values 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 (`WriteHalo.CYCL`), or whether to not update any ghost regions in this dimension at all (`WriteHalo.NONE`, equivalent to setting the corresponding elements of `wlo`, `whi` to zero).

Operation of `writeHalo` is visualized in figure 6.1.

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) ;
...
```

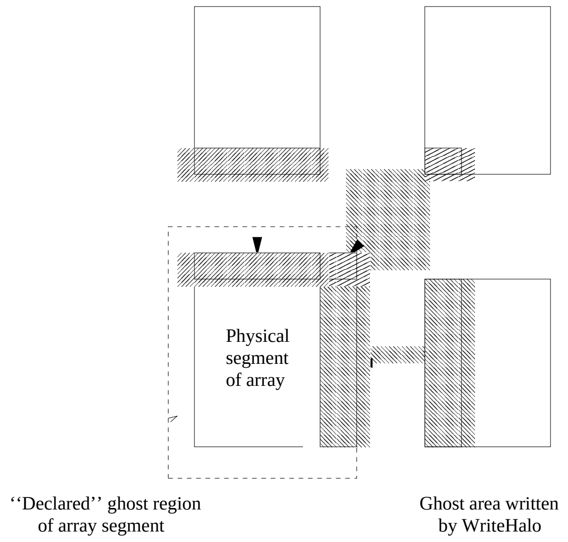


Figure 6.1: Illustration of the effect of executing the `writeHalo` function.

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 $\{0, \ldots, N-1\}$ 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 $\{0, \ldots, N-1\}$ 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

```
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.

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.

[Need to document `HPspmd.copy()` *somewhere.]*

Chapter 7

Low level SPMD programming

It often happens that some parts of a large parallel program cannot be written efficiently in the pure data parallel style, using `overall` constructs to process all elements of distributed arrays on essentially the same footing. Sometimes, for efficiency, a process has to be more "introspective"—it has to get down and do some procedure that combines the locally held array elements in a non-trivial way. The local results may be combined with off-processor results in a separate step.

7.1 An Example

We will consider a fragment from a parallel *N-body* classical mechanics problem. As the name suggests, this problem is concerned with the dynamics of a set of N interacting bodies. The total force on each body includes a contribution from all the other bodies in the system. The size of this contribution depends on the position, x, of the body experiencing the force, and the position, y, of the body exerting it. If the individual contribution is given by `force(`x, y`)`, the net force on body i is

$$\sum_j \text{force}(a_i, a_j)$$

where now a_j is the position of the jth body. The total force can be computed in parallel by the program given in 7.1. We repeatedly rotate a copy, `b`, of the position vector, `a`, using `cshift`. Every element in `b` thus passes by every element in the fixed vector `a`, and contributions to the force are accumulated as we go. The approach is similar to the pipelined matrix multiplication in Figure 3.12.

The trouble is that this involves N small shifts (Figure 7.2). Calling out to the communication library so many times (and copying a whole array so many times) is likely to produce an inefficient program.

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

  float [[-]] f = new float [[x]], a = new float [[x]],
              b = new float [[x]] ;

  ... initialize 'a' ...

  overall(i = x for :) {
    f [i] = 0.0 ;
    b [i] = a [i] ;
  }

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

    overall(i = x for :)
      f [i] += force (a [i], b [i]) ;

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

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

Figure 7.1: Data parallel version of the N-body force computation.

In fact we can achieve an equivalent effect—passing the value of every element by every other—if we do just P iterations of an outer loop, with each iteration shifting the *whole block* of locally held elements of the moving copy to the neighbouring process (Figure 7.3).

We can express the second algorithm straightforwardly enough in the language defined so far, but the price is that we have to change the way the distributed arrays are represented in the program.

One legitimate way to express the algorithm is in a direct SPMD message-passing style. Example code is given in Figure 7.4. The local block size is `B`, so the value of `N` is `P` $\times$ `B`. For the sake of being concrete, we have used the methods `Rank()` and `Sendrecv_replace()` from the *mpiJava* [?] binding of the Message Passing Interface, MPI [?]. Figure 7.4 is a valid SPMD Java program and thus a valid HPJava program. Unfortunately it is lives in a different universe of data structures and communication functions from the parallel-array, collective-communication oriented algorithms we have seen so far. We need to build some bridge between these two extreme styles of parallel programming.

One approach—perhaps not the most obvious, but quite natural in HPJava—

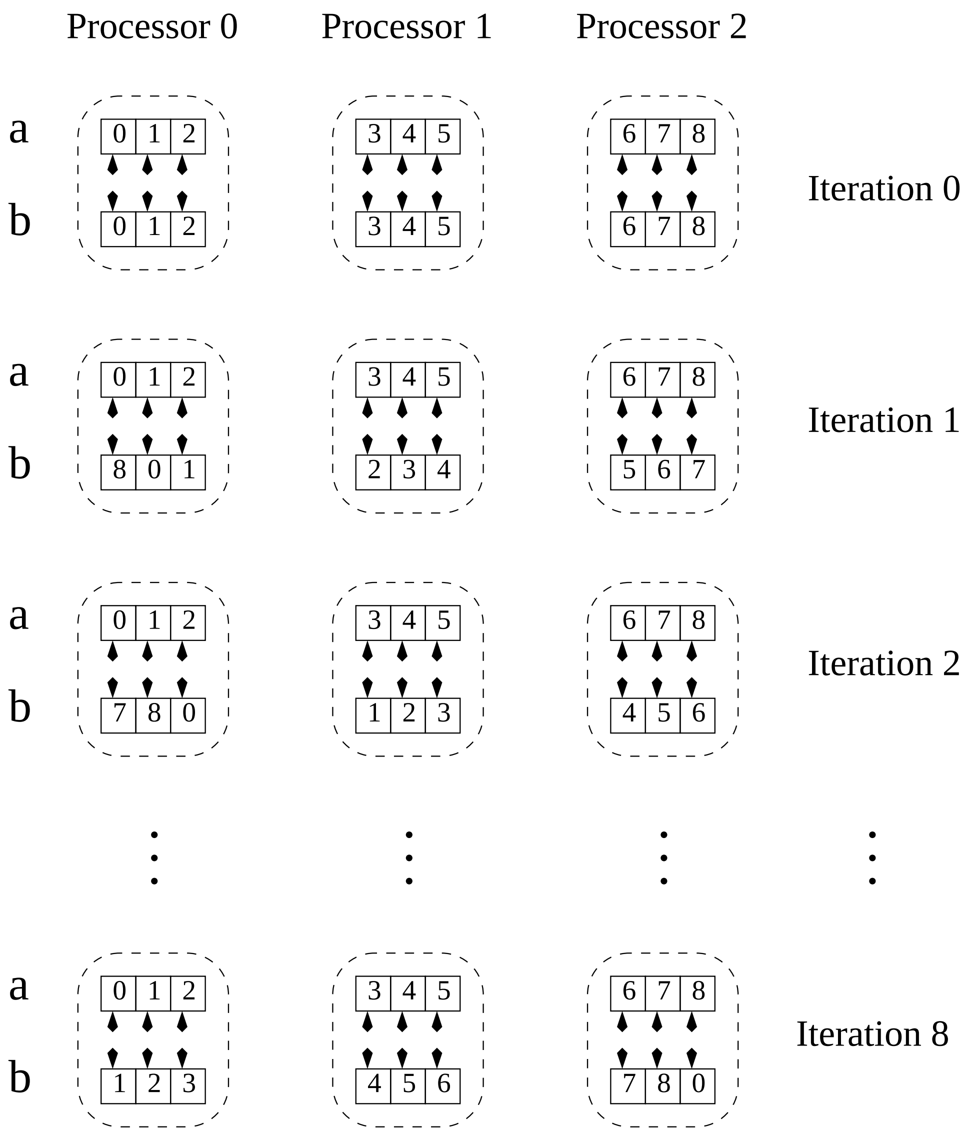


Figure 7.2: Simple "data parallel" N-body force computation. The array `b` is shifted one element to the right in each iteration. Arrows define element pairs combined in the iteration.

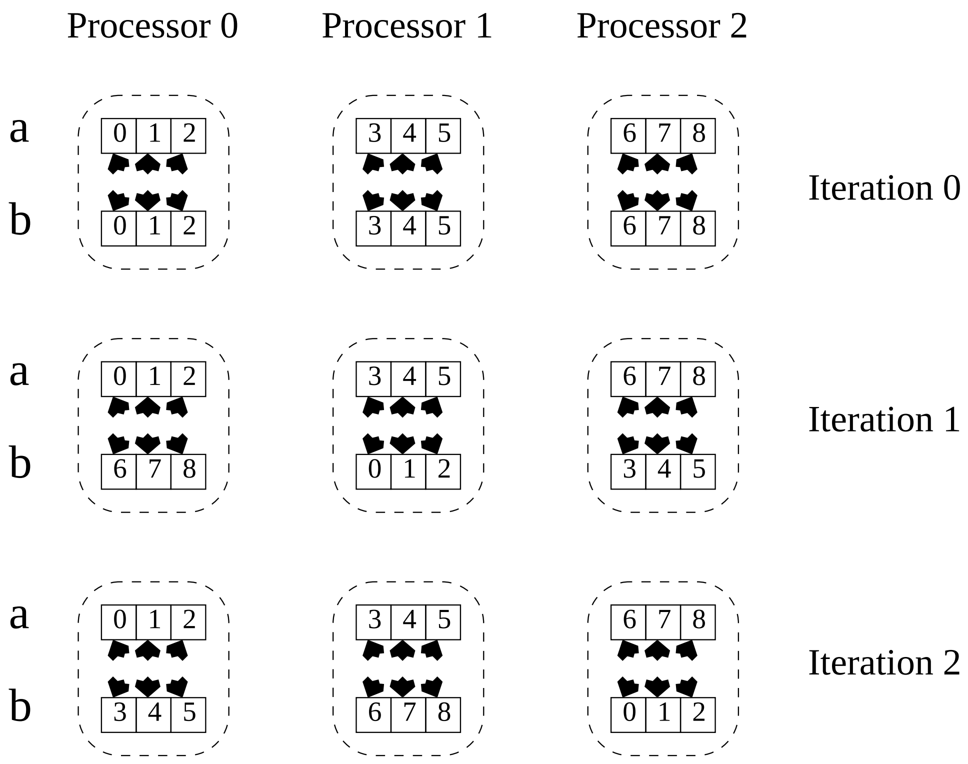


Figure 7.3: Efficient N-body force computation. The array `b` is shifted one *block* to the right in each iteration. Arrows define element pairs combined in the iteration.

```
int myID = MPI.COMM_WORLD.Rank();

float [] f = new float [B], a = new float [B], b = new float [B] ;

... initialize ‘a’ ...

for(int i = 0 ; i < B ; i++) {
  f[i] = 0.0 ;
  b[i] = a[i] ;
}

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

  for(int i = 0 ; i < B ; i++)  // B : local block size
    for(int j = 0 ; j < B ; j++)
      f [i] += force(a [i], b [j]) ;


  // cyclically shift ‘b’...

  int right = (myID + 1) % P, left = (myID + P - 1) % P ;

  MPI.COMM_WORLD.Sendrecv_replace(b, 0, B, MPI.FLOAT,
                                  right, 0, left, 0) ;
}
```

Figure 7.4: MPI version of the N-body force computation.

is to explicitly split the index space of the original one-dimensional arrays across two dimensions: a distributed dimension representing the process dimension itself, and a sequential dimension explicitly representing the local block. The code is given in Figure 7.5.

Although there is only one element of `d` associated with each process, the rules of HPJava force us to explicitly subscript in the associated array dimension. This leads to some extra verbosity, but this style of programming has some attractive features:

- As a practical matter, the fact we are dealing with true HPJava distributed arrays means we can continue to employ concise calls to collective library functions like `cshift()`, instead of relatively clumsy functions like `Sendrecv_replace()`.
- More esoterically, the program follows the *canonical HPspmd style*, described briefly in section 5.4. As defined in that section, all variables are *coherent*. The elements of the arrays in the program of Figure 7.4 are *not* coherent, because they take different values in each process, although their home group is the set of all processes. Respecting the canonical style may

```
Procs1 p = new Procs1(P) ;
Dimension d = p.dim(0) ;

on(p) {

  float [[-,*]] f = new float [[d, B]], a = new float [[d, B]],
                b = new float [[d, B]] ;

  ... initialize ‘a’ ...

  overall(i = d for :)
    for (int j = 0 ; j < B ; j++) {
      f [i, j] = 0.0 ;
      b [i, j] = a [i, j] ;
    }

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

    overall(i = d for :)
      for(int j = 0 ; j < B ; j++)
        for(int k = 0 ; k < B ; k++)
          f [i, j] += force(a [i, j], b [i, k]) ;

    // cyclically shift ‘b’ in ‘d’ dim...

    float [[-,*]] tmp = new float [[d, B]] ;

    Adlib.cshift(tmp, b, 1, 0) ;
    HPspmd.copy(b, tmp) ;
  }
}
```

Figure 7.5: Efficient HPJava version of the N-body force computation.

not have immediate practical advantages, but it is somehow aesthetically pleasing. In this style there is no need for incoherent functions like `Rank()` to get the local process id—instead one uses global index values associated with `overall` constructs.

7.2 Dimension Splitting

This style goes some way toward forging a link between low-level SPMD programming and the higher level data-parallel style of HPJava, but by itself it doesn't help if we are presented with an *existing* one-dimensional, block-distributed array, and required to do some low-level processing on its blocks. To allow for this situation, the language is extended to support *dimension splitting*. Dimension splitting is introduced as an extension of the array section mechanism described at length in Chapter 4. The extra syntactic baggage is minimal, but the repercussions are quite far-reaching.

First we note that a particular element in a distributed array can be identified in one of two ways. It can be identified by giving a *global* subscript in the distributed range, which is effectively what we have done in HPJava in earlier chapters. Alternatively it can be identified by giving a process coordinate and a *local subscript*—a subscript within the array segment held on the associated process. Dimension splitting provides a way of accessing an element through its process coordinate and local subscript, by allowing a distributed dimension of an array to be temporarily viewed as *two* dimensions—a coordinate dimension plus a local subscript dimension.

If the subscript in a particular dimension of a section expression is the special symbol `<>`, that dimension of the array is split. Whereas a triplet subscript in a section expression yields one dimension in the result array, a splitting subscript yields two—a distributed dimension and a sequential dimension. The range of the distributed dimension is the process dimension over which the original array dimension was distributed[1]; the local blocks of the original array are embedded in the the sequential dimension of the result. The two new dimensions appear consecutively in the signature of the result array, distributed dimension first.

Now we can combine the examples from Figures 7.1 and 7.5 of the last section. The version in Figure 7.6 initally creates the arrays as distributed, one-dimensional arrays, and uses this convenient form to initialize them. It then uses a split representation of the same arrays to compute the force array. Note that, because `as` and `fs` are semantically *sections* of `a` and `f`, they share common elements—they provide aliases through which the same element variables can be accessed. So when the computation loop is complete, the vector of forces can be accessed again through the one-dimensional array `f`. This is likely to be what is needed in this case.

As a similar but slightly more complicated example, Figure 7.7 contains an optimized version of the pipelined matrix multiplication from Figure 3.12. Here the arithmetic is done in local blocks by a method `matmul`, which implements the matrix multiplication $\texttt{c} = \texttt{a} \times \texttt{b}$ on sequential two-dimensional arrays. It

[1]Usually, but see Section 7.8.

could be written in elementary style as

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

  int l = c.rng(0).size(), m = c.rng(1).size(), n = a.rng(1).size() ;

  for (int i = 0 ; i < l ; i++)
    for (int j = 0 ; j < m ; j++) {
      c [i, j] = 0.0 ;
      for (int k = 0 ; k < n ; k++)
        c [i, j] += a [i, k] * b [k, j] ;
    }
}
```

or it could be an optimized library routine. The operation of the parallel algorithm for $\mathtt{P} = 2$ is illustrated in 7.8.

```
Procs1 p = new Procs1(P) ;
Dimension d = p.dim(0) ;

on(p) {
  Range x = new BlockRange(N, d) ;

  float [[-]] f = new float [[x]], a = new float [[x]],
              b = new float [[x]] ;

  ... initialize ‘a’ ...

  overall(i = x for :) {
    f [i] = 0.0 ;
    b [i] = a [i] ;
  }


  // split ‘x’ dimensions:

  float [[-,*]] fs = f [[<>]], as = a [[<>]], bs = b [[<>]] ;

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

    overall(i = d for :)
      for(int j = 0 ; j < B ; j++)
        for(int k = 0 ; k < B ; k++)
          fs [i, j] += force(as [i, j], bs [i, k]) ;

    // cyclically shift ‘bs’ in ‘d’ dim...

    Adlib.cshift(tmp, bs, 1, 0) ;
    HPspmd.copy(bs, tmp) ;
  }
}
```

Figure 7.6: Version of the N-body force computation using dimension splitting.

```
Procs1 p = new Procs1(P) ;
Dimension d = p.dim(0) ;

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]] ;

  ... initialize ‘a’, ‘b’


  // split ‘x’ dimensions:

  float as [[-,*,*]] = a [[<>, :]] ;
  float bs [[*,-,*]] = b [[:, <>]] ;
  float cs [[-,*,*]] = c [[<>, :]] ;

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

    overall (i = d for :) {
      const int base = B * ((i‘ + s) % P) ;

      matmul(cs [[i, :, base : base + B - 1]],
                              as [[i, :, :]], bs [[:, i, :]]) ;
    }

    // cyclically shift down ‘bs’ in ‘d’ dim...

    float tmp [[*,-,*]] = new float [[N, d, B]] ;

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

Figure 7.7: Pipelined matrix multiplication program using dimension splitting.

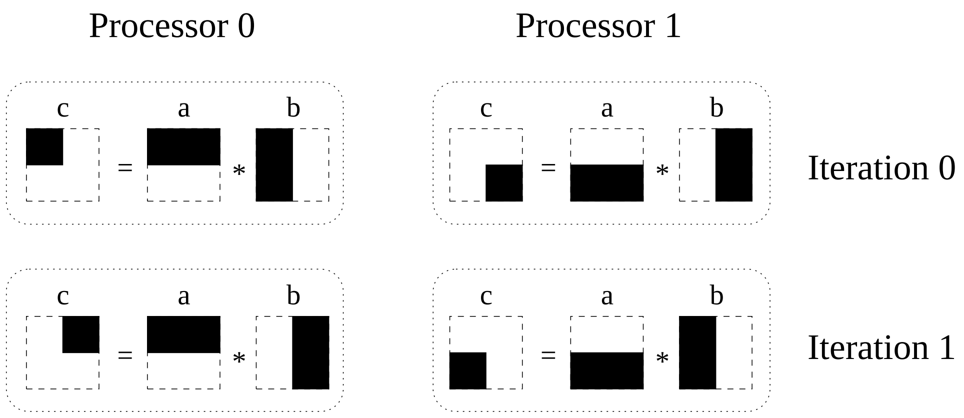


Figure 7.8: Operation of efficient pipelined matrix computation. The matrix `b` is shifted one block to the right in each iteration.

7.3 Block Parameters

That is quite neat, but unfortunately it isn't the end of the story. The examples of the previous section will only work properly if N is an exact multiple of the number of processes, P, so that the block size, B, is identical in all processes. In general we do not want to limit ourselves to this case.

A few of the possibilities for mapping a 50-element, one-dimensional array to 4 processes are illustrated in Figure 7.9. They presume the declarations:

```
Procs p = new Procs1(4) ;
Dimension d = p.dim(0) ;
```

In the first case the array a is divided into four contiguous blocks of sizes $(13, 13, 13, 11)$. In the second case the blocking is different—$(13, 13, 12, 12)$—and the formula for computing the global index value is quite different. The third case illustrates that a might actually be some section of an array. In this example the blocking is $(13, 12, 13, 12)$ and we must take into account that the subscripting into the local segment of the array is strided. Also there is an offset of the first element, which in some processes is zero and in others is one.

At this point one might feel inclined to abandon the idea of dimension splitting. These examples doesn't seem to fit at all with the idea of dividing a distributed dimension of extent N into P blocks of constant size B, which is what is needed if dimension splitting is to work.

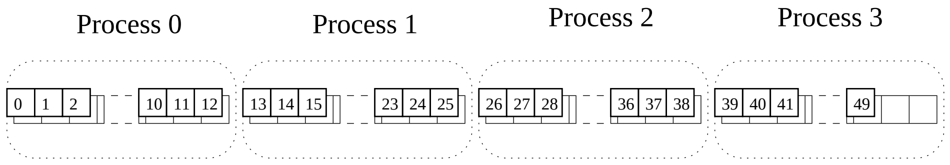
Persistence will pay off. For the sake of making progress, we note that for a given range there is a fixed *bound* on the number of array elements held by any process. We can now redefine the symbol B to refer to this constant bound. The operational assumption is that *all* processes allocate enough space to hold this bounding number of elements, although not all processes necessarily use all slots. For the dimension-split array, the extent of the the sequential dimension is the constant number, B, of locally allocated slots. The elements of the original array are embedded somehow in these slots.

In Figure 7.9 the most likely value for B in the first two examples is $\lceil 50/4 \rceil = 13$. For the third example the amount of space allocated will presumably be enough to hold the parent distributed array—most likely B is $\lceil 100/4 \rceil = 25$[2].

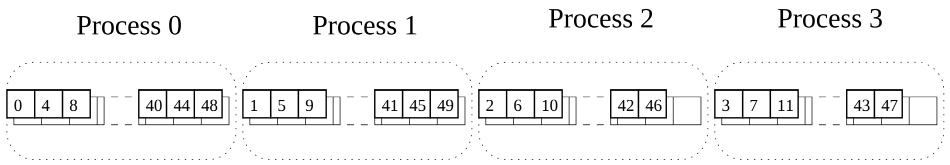
The embedding of actual elements is determined by a new method, `local-Block()`. This is a member of the `Range` class. It has no arguments, and returns

[2]In this case the dimension-split version of a is the same as the dimension-split version of the parent array b. A curious feature of a section with a dimension-splitting subscript is that the "section" may have *more* accessible elements than the parent array. This is a little counter-intuitive, and introduces some possibilities for abuse. But it is not a logically inconsistent situation, and, used carefully, it has various benefits.

```
Range x = new BlockRange(50, d) ;
float [[-]] a = new float [[x]] ;
```



```
Range x = new CyclicRange(50, d) ;
float [[-]] a = new float [[x]] ;
```



```
Range x = new BlockRange(100, d) ;
float [[-]] b = new float [[x]] ;

float [[-]] a = b [[0 : 99 : 2]] ;
```

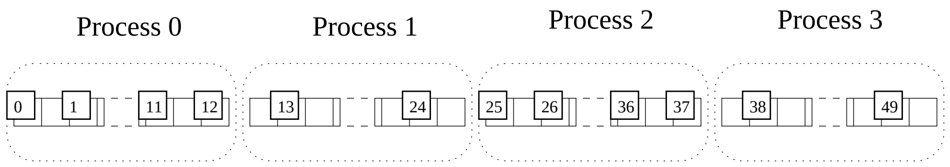


Figure 7.9: Example embeddings of array elements in local blocks

an object of class `Block`, declared as:

```
class Block {
  public int count ;

  public int sub_bas ;
  public int sub_stp ;

  public int glb_bas ;
  public int glb_stp ;
}
```

The value of `count` specifies the number of actual array elements in the selected block, and the pair `sub_bas`, `sub_stp` define a base and step for the local subscript values associated with those elements. The pair `glb_bas`, `glb_stp` define a base and step for the *global* index values associated with the elements.

(Note that unfortunately there is a overlap of nomenclature between the blocks of an arbitrary range, and the distribution format of one particular kind of range: `BlockRange`. The `Block` class and the `localBlock()` method are in no way specifically tied to the "block-wise" distribution format embodied in `BlockRange`. Equivalent methods are defined for any range.)

One of the most important applications of the `localBlock()` method is in a translation scheme for the `overall` construct, and this is a natural way to illustrate its use.

Consider this fragment of HPJava:

```
float [[-]] a;

Range x = a.rng(0);

overall(i = x for :)
  a [i] = (float) i` ;
```

By applying dimension splitting to the array `a`, the `overall` construct can be translated as illustrated in Figure 7.10.

At first sight we have just replaced one `overall` construct with another. But recall that the range `d` can be assumed to be a process dimension, so the role of the new `overall` construct is essentially "formal"—`k` only has one location in each process, so the new `overall` yields no local loop. Although the `k` subscript of `as` is required by the rules of the language, it is essentially does nothing, again because there is only one location. In fact the only substantive effect of the reduced construct is to change the active process group inside its body.

Effectively, this transformation has reduced the `overall` construct to a sequential local *for* loop. Subscripting with a distributed index has essentially been reduced to subscripting into the sequential local array `as[[k, :]]`. Moreover, the subscript expression and the global index expression on the right hand side have been reduced to expressions linear in the loop index. Such expressions can be translated efficiently by a compiler using a *strength reduction* op-

SOURCE:

```
float [[-]] a;

Range x = a.rng(0);

overall(i = x for :)
  a [i] = (float) i` ;
```

TRANSLATION:

```
float [[-]] a;

Range x = a.rng(0);

float [[-,*]] as = a [[<>]] ;

Range d = as.rng(0) ;

overall(k = d for :) {
  Block b = x.localBlock();

  for (int l = 0 ; l < b.count ; l++)
    as [k, b.sub_bas + b.sub_stp * l] =
                        (float) (b.glb_bas + b.glb_stp * l) ;
}
```

Figure 7.10: Recursive translation of a simple `overall` construct.

timization (replacing the linear expressions with an incremented accumulation variable).

Because it involves reducing an `overall` construct to a kind of lower-level `overall`, we call this general scheme *recursive translation*.

Note that technically the `localBlock()` inquiry is *incoherent* (see section 5.4). This could be fixed by requiring an argument that specified the local coordinate, similar to the argument of the coherent (though otherwise lower-level) `block()` method that will be introduced in section 7.7. This argument was omitted here to simplify usage, and also to allow the unique result of `localBlock()` to be computed once and cached inside the `Range` object. Effectively the incoherent `crd()` inquiry is used internally.

7.3.1 Ghost regions and dimension splitting

If the distributed range has ghost extensions, this does *not* affect the values in the block description returned by `localBlock()`. These values describe the layout of elements associated with the "physical" portion of the array, not elements in the ghost region. In this case, however, the range of legal subscripts in the local sequential array dimension is increased. In the absence of ghost regions that range may be, for example, $0, \ldots, B-1$, where typically B would be $\lceil N/P \rceil$. If the original range has lower and upper ghost extensions of width $w_{\mathrm{lo}}, w_{\mathrm{hi}}$, so does the new sequential range, exposed by dimension splitting. The allowed range of local subscripts will be $-w_{\mathrm{lo}}, \ldots, B + w_{\mathrm{hi}} - 1$[3]. Figure 7.11 gives the recursive translation of an `overall` construct involving a shifted index. It assumes `a` has a suitable ghost extensions.[4].

7.3.2 Local blocks of subranges

So far our recursive translation scheme does not apply to general `overall` constructions, which include some non-default triplet parameters. Overloaded versions of `localBlock()` that take `l`, `u`, `s` arguments are provided. Translation in this case is illustrated in Figure 7.12. There is also a version of `localBlock()` that omits the stride argument, `s`. This can be used in the case of unit stride.

[3]Note however that the `size()` inquiry applied to the associated collapsed range will still return the "physical" extent, B.

[4]This example actually assumes x has alignment stride of 1. In general the displacement in the translation should be multiplied by x`.str()`, where x is the range parametrizing the original `overall`.

SOURCE:

```
float [[-]] a;

Range x = a.rng(0);

overall(i = x for :)
  a [i + 1] = ... ;
```

TRANSLATION:

```
float [[-]] a;

Range x = a.rng(0);

float [[-,*]] as = a [[<>]] ;

Range d = as.rng(0) ;

overall(k = d for :) {
  Block b = x.localBlock();

  for (int l = 0 ; l < b.count ; l++)
    as [k, b.sub_bas + b.sub_stp * l + 1] = ... ;
}
```

Figure 7.11: Recursive translation of a shifted index subscript.

SOURCE:

```
float [[-]] a;

Range x = a.rng(0);

overall(i = x for l : u : s)
  a [i] = (float) i` ;
```

TRANSLATION:

```
float [[-]] a;

Range x = a.rng(0);

float [[-,*]] as = a [[<>]] ;

Range d = as.rng(0) ;

overall(k = d for :) {
  Block b = x.localBlock(l, u, s);

  for (int l = 0 ; l < b.count ; l++)
    as [k, b.sub_bas + b.sub_stp * l] =
                        (float) (b.glb_bas + b.glb_stp * l) ;
}
```

Figure 7.12: Recursive translation of `overall` construct with triplet index range.

7.4 Reduction to Java arrays

Dimension splitting allows one to access the local blocks of distributed arrays as sequential HPJava arrays. In many cases this may be all one needs to do low level SPMD programming. But the translation scheme for HPJava actually assumes that ultimately all Fortran-like arrays are implemented in terms of the standard arrays of Java. The spirit of HPspmd languages is not to conceal such things. In fact the spirit of HPspmd languages is to shamelessly expose internal workings at all levels. Hence the underlying Java arrays should be available if they are needed.

The inquiry `dat()` can be applied to any HPJava array. It returns a reference to a Java array with the same type of elements as the target array. This is the actual array in which the local elements are stored.

This gives us yet another way to optimize the original data-parallel N-body example of Figure 7.1. We can express the compute loop in the MPI style of Figure 7.4. The code is given in Figure 7.13. This particular implementation assumes that the process grid `p` coincides with the MPI group associated with the `COMM_WORLD` communicator.

Again this simple example hides the complexities that arise if we have to deal with a general distributed array. As we saw in the previous section, the local elements of a general distributed array are effectively stored in a multidimensional sequential array. The detailed embedding is defined by the `localBlock()` inquiry on the distributed array ranges.

The mapping of the local sequential multidimension array into the Java array is defined in turn by new inquiries `bas()` and `str()` on an HPJava array. These integer-valued methods define a base offset, and a stride for each dimension[5]. If an element of a sequential HPJava array, `a`, has integer subscripts $i_0, \ldots, i_{R-1}$ it is stored in element

$$\texttt{a.dat() [a.bas()} + i_0 \times \texttt{a.str(0)} + \ldots + i_{R-1} \times \texttt{a.str(}R-1\texttt{)]}$$

of the local Java array.

This formula can be extended to a formula for finding the local elements of distributed arrays. First we note that through dimension splitting any distributed array can be reduced to an array that has a mix of only sequential dimensions and level 0 ranges. The level 0 ranges contribute nothing to the total offset of the element in the local Java array. So if the subscript list is $i_0, \ldots, i_{R-1}$—a mix of integers and level 0 distributed index symbols—the local element is

$$\texttt{a.dat() [a.bas()} \quad + \sum_{\substack{r, \\ r\text{th subscript} \\ \text{an integer}}} i_r \times \texttt{a.str(}r\texttt{)]}$$

[5]Note these "memory" bases and strides are distinct from the subrange alignment parameters returned by similarly named inquiries on ranges.

To complete the story, we need to know how the values of the `str()` and `bas()` members are defined on a dimension split array. If `as` is a section defined by splitting dimension r of an array `a`, to yield dimensions r' and $r' + 1$ of `as`, then

- `as.bas()` is equal to `a.bas()`,
- `as.str(`$r' + 1$`)` (the stride associated with the new sequential dimension) is equal to `a.str(`r`)`, and
- `as.str(`r'`)` (the stride associated with the new distributed dimension) is equal to `a.str(`r`)` $\times$ `as.rng(`$r' + 1$`).volume()`.

The complicated final definition only really matters in the block cyclic case (section 7.8), where we have to apply dimension splitting a second time. Temporarily ignoring that case, we can use our new formulae to further simplify the recursive translation given earlier. The new translation is given in Figure 7.14.

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

  float [[-]] f = new float [[x]], a = new float [[x]],
              b = new float [[x]] ;

  ... initialize 'a' ...

  overall(i = x for :) {
    f [i] = 0.0 ;
    b [i] = a [i] ;
  }

  // extract the local vectors of elements:

  float [] f_blk = f.dat(), a_blk = a.dat(), b_blk = b.dat() ;

  int myID = MPI.COMM_WORLD.Rank();

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

    for(int i = 0 ; i < B ; i++)  // B : local block size
      for(int j = 0 ; j < B ; j++)
        f_blk [i] += force(a_blk [i], b_blk [j]) ;


    // cyclically shift 'b_blk'...

    int right = (myID + 1) % P, left = (myID + P - 1) % P ;

    MPI.COMM_WORLD.Sendrecv_replace(b_blk, 0, B, MPI.FLOAT,
                                    right, 0, left, 0) ;
  }
}
```

Figure 7.13: Version of the N-body force computation using reduction to Java arrays

SOURCE:

```
float [[-]] a;

Range x = a.rng(0);

overall(i = x for :)
  a [i] = (float) i` ;
```

TRANSLATION:

```
float [[-]] a ;

Range x = a.rng(0) ;

Block b = x.localBlock() ;

for (int l = 0 ; l < b.count ; l++)
  a.dat() [a.bas() + (b.sub_bas + b.sub_stp * l) * a.str(0)] =
                     (float) (b.glb_bas + b.glb_stp * l) ;
```

Figure 7.14: Reduction of subscripting in simple `overall` construct to local Java array accesses.

7.5 Local arrays

We have seen that the HPJava language provides two complementary ways to access the local part of a distributed array—through the dimension splitting syntax, and through the `dat()` inquiry.

Neither of these return exactly what one might originally have expected—a local sequental array containing exactly the local elements of the distributed array—no more and no less.

We refrained from complicating the language definition with this functionality, because it can now be implemented using a library function. For example, for a two dimensional array of `float` the following procedure would do the job[6]:

```
public static float [[*,*]] local(float [[-,-]] a) {
  Range x = a.rng(0), y = a.rng(1) ;

  Block b = x.localBlock(), c = y.localBlock() ;

  int b_sub_top = b.sub_bas + (b.count - 1) * b.sub_stp ;
  int c_sub_top = c.sub_bas + (c.count - 1) * c.sub_stp ;

  float [[-,*,-,*]] as = a [[<>, <>]] ;

  return as [[x.dim().crd(), b.sub_bas : b_sub_top : b.sub_stp,
              y.dim().crd(), c.sub_bas : c_sub_top : c.sub_stp]] ;
}
```

[6]This doesn't work if either array dimension has block-cyclic distribution

7.6 An extended example: prefix computation

In this section we will give a non-trivial example of how dimension splitting can be combined with the new inquiries on the `Range` class to write optimized parallel code that works for input arrays with general distribution formats.

A prefix computation (also sometimes known as the "scan" operation) takes an array as input, and outputs an array containing the set of partial sums of the elements of that array. If the input array is `a` and the result array is `r`, we want the outcome to be:

$$\texttt{r [}i\texttt{]} \quad = \quad \texttt{a [}0\texttt{]} + \ldots + \texttt{a [}i\texttt{]}$$

This is the *inclusive* form of prefix computation. There is also an *exclusive* form which would be defined by

$$\begin{aligned} \texttt{r [}0\texttt{]} \quad &= \quad 0 \\ \texttt{r [}i\texttt{]} \quad &= \quad \texttt{a [}0\texttt{]} + \ldots + \texttt{a [}i-1\texttt{]}; \quad 1 \leq i < N \end{aligned}$$

We will be interested in computing the inclusive form, but sometimes the exclusive form is needed in intermediate steps.

We can give a straightforward data-parallel algorithm for this computation using a *doubling* technique. Possible code is given in Figure 7.15. The algorithm is very simple. In a given iteration, the current value in the result array is shifted by an amount that doubles between iterations. The shifted array is added into the current array. The algorithm takes $\log_2(N)$ iterations to complete. Its operation is illustrated in Figure 7.16.

As often happens, the pure data parallel program is concise and quite readable. However it isn't very efficient. It requires a total of $N \times \log_2(N)$ floating point additions, whereas the naive sequential algorithm only needs N. So we can only expect useful parallel speedup if $P \gg \log_2(N)$. The pure data parallel version needs optimization.

7.6.1 Optimization for block distribution formats

For the most straightforward, block-based distribution formats—these include arrays parameterized `BlockRange`, `ExtBlockRange` and `IrregRange`, there is a fairly straightforward optimization. We can do prefix computation within individual blocks, then do a global exclusive prefix combining the sums of the blocks. Finally we add the global prefix back to the incomplete prefixes within the blocks. This is illustrated in Figures 7.17, 7.18.

This version has a hope to be reasonably efficient if $N \gg P$, so arithmetic costs have a chance to dominate communication cost. It still does about $2N$ total additions operations instead the N operations for the sequential code. But in principle if $P > 2$ it should be possible to compensate for this factor, and gain some parallel speedup.

```
void doublingPrefix(float [[-]] a) {
  Group p = a.grp() ;

  Range x = a.rng(0) ;
  int N = x.size();

  on(p) {
    float [[-]] t = new float [[x]] ;

    for(int s = 1 ; s < N ; s *= 2) {
      Adlib.shift(t, a, s) ;

      overall(i = x for s : N - 1)
        a [i] += t [i] ;
    }
  }
}
```

Figure 7.15: Simple doubling algorithm for parallel prefix

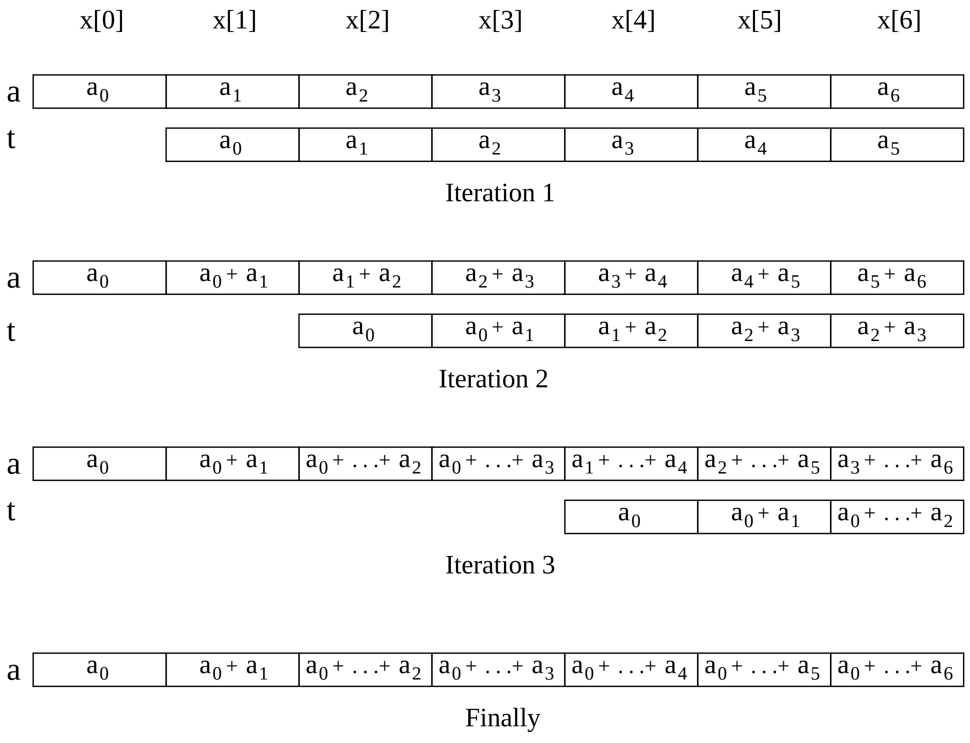


Figure 7.16: Illustration of doubling algorithm for parallel prefix for N = 7.

```
void blockPrefix(float [[-]] a) {
  Range x = a.rng(0) ;

  float [[-,*]] as = a [[<>]] ;

  Range z = as.rng(0) ;

  float [[-]] t = new float [[z]], s = new float [[z]] ;

  // 1. Do intra-block prefix.  Set ‘t’ elements to sums of blocks

  overall (k = z for :) {
    Block b = x.localBlock() ;

    float sum = 0.0f ;
    for (int l = 0 ; l < b.count ; l++) {
      int sub = b.sub_bas + b.sub_stp * l ;

      sum = (as [k, sub] += sum) ;
    }

    t [k] = s [k] = sum ;
  }

  // 2. Do "global prefix"

  prefix(t) ;

  // 3. Add exclusive global prefix to intra-block prefixes in ‘a’

  overall (k = z for :) {
    Block b = x.localBlock() ;

    float sum = t [k] - s [k] ;

    for (int l = 0 ; l < b.count ; l++) {
      int sub = b.sub_bas + b.sub_stp * l ;

      as [k, sub] += sum ;
    }
  }
}
```

Figure 7.17: Parallel prefix optimized for block-wise distribution formats.

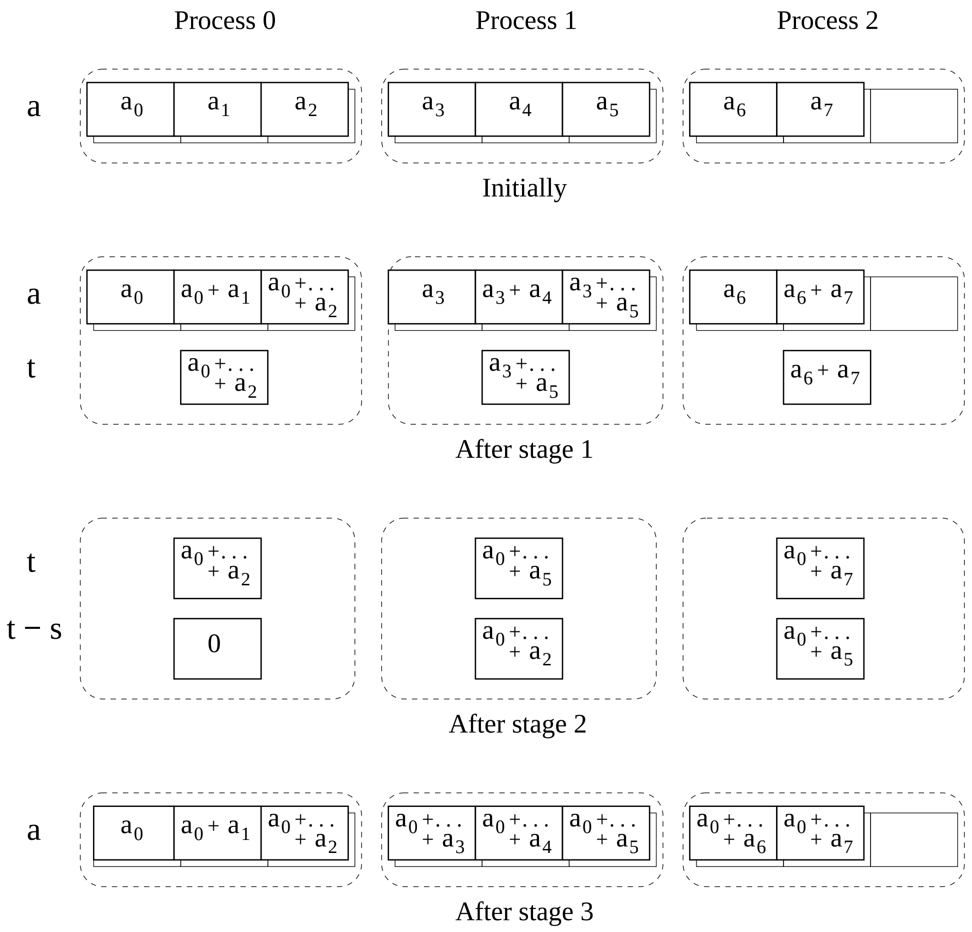


Figure 7.18: Prefix optimized for block distributions: illustration for N = 8, P = 3.

The code given here will work for the ranges mentioned above, and subranges of these[7]. It will not work for cyclic distributions.

7.6.2 Cyclic distribution formats

It seems difficult to give a truly efficient parallel prefix algorithm when the data has cyclic distribution format. It looks like an *IO bound* problem. It is certainly possible to find optimized algorithms that are *better* than our naive data parallel version—for example reducing the number of communication operations.

In general, even if an operation like this cannot be implemented with really good arithmetic performance, it does not follow that it is a useless operation, or that it is not worth optimizing. The operation in question might be a necessary step in larger program (for example the block-cyclic prefix that we will mention in section 7.8). So it may be worth our while to optimize the procedure so that it does not form a bottleneck in the larger context, even if we can't make it truly efficient as an arithmetic operation in its own right.

An improved scheme is illustrated in Figure 7.19. To make the example a bit more interesting, it covers the case of an array with a non-trivial (stride 2) alignment to a cyclic range.

The input array is copied to a temporary (dimension split) array `ts`, with zeros in positions for which there is no corresponding element of `a`. Then the global prefixes are formed across individual rows (as drawn here). With the the blocks (columns) treated as vectors, we would only need $\log_2$(`P`) shift-type operations to do this, if we used a naive doubling algorithm for this stage.

An exclusive prefix of the final column—the row sums—is broadcast to all processes. This exclusive prefix is added to the incomplete prefixes across the rows, computed previously. The results are copied back to `a`.

It is straightforward enough to implement this scheme in HPJava using dimension splitting. However the improvement in efficiency is probably not dramatic, and we will save space here by omitting the specialized code. To tell the truth the "naive" data parallel version is probably not much worse in practise.

7.6.3 Optimization for "general" distribution formats

We can combine the procedures given in the preceding sections to produce a single optimized `prefix` procedure that works for any distribution format by using the `format()` inquiry on `Range`. The code is given in Figure 7.20.

The inquiry `format()` returns the constant `DIST_DIMENSION` if the range is a process dimension and `DIST_CYCLIC` if it is a cyclically distributed range (or subrange). In these two cases we use the naive algorithm. In all other cases we use the improved `blockPrefix()`.

Notice that `blockPrefix()` will work OK for a collapsed range (corresponding to a sequential array). It is permitted to do dimension splitting on a collapsed array. The range of the resulting distributed dimension is a "degenerate"

[7]At least for subranges with positive alignment stride. We will return to this issue in section 7.6.3

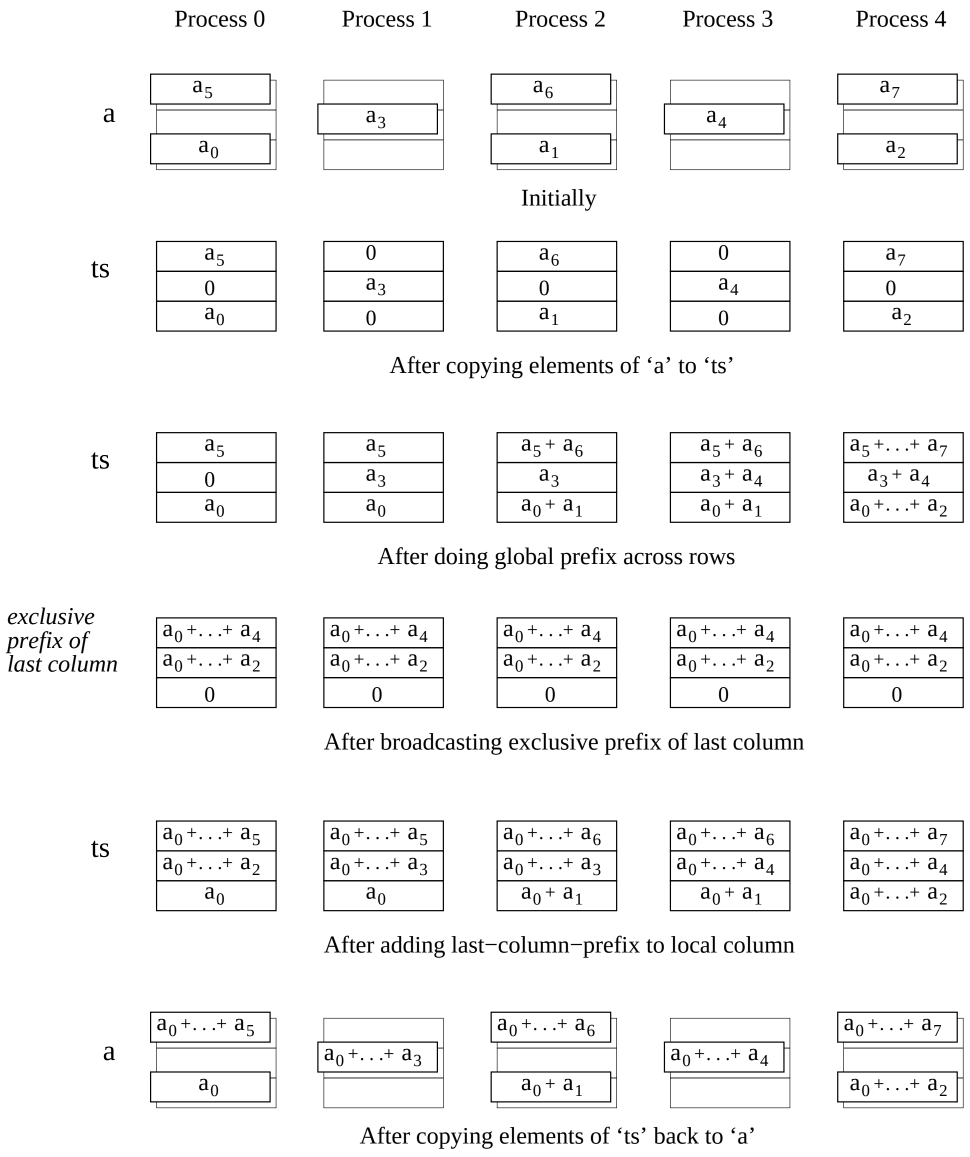

Process 0
Process 1
Process 2
Process 3
Process 4
a
a_5
a_0
a_3
a_6
a_1
a_4
a_7
a_2
Initially
ts
a_5 | 0 | a_0
0 | a_3 | 0
a_6 | 0 | a_1
0 | a_4 | 0
a_7 | 0 | a_2
After copying elements of ‘a’ to ‘ts’
ts
a_5 | 0 | a_0
a_5 | a_3 | a_0
$a_5 + a_6$ | a_3 | $a_0 + a_1$
$a_5 + a_6$ | $a_3 + a_4$ | $a_0 + a_1$
$a_5 +...+ a_7$ | $a_3 + a_4$ | $a_0 +...+ a_2$
After doing global prefix across rows
exclusive prefix of last column
$a_0 +...+ a_4$ | $a_0 +...+ a_2$ | 0
After broadcasting exclusive prefix of last column
ts
$a_0 +...+ a_5$ | $a_0 +...+ a_2$ | a_0
$a_0 +...+ a_5$ | $a_0 +...+ a_3$ | a_0
$a_0 +...+ a_6$ | $a_0 +...+ a_3$ | $a_0 + a_1$
$a_0 +...+ a_6$ | $a_0 +...+ a_4$ | $a_0 + a_1$
$a_0 +...+ a_7$ | $a_0 +...+ a_4$ | $a_0 +...+ a_2$
After adding last−column−prefix to local column
a
$a_0 +...+ a_5$
a_0
$a_0 +...+ a_3$
$a_0 +...+ a_6$
$a_0 + a_1$
$a_0 +...+ a_4$
$a_0 +...+ a_7$
$a_0 +...+ a_2$
After copying elements of ‘ts’ back to ‘a’

Figure 7.19: Possible optimization of prefix for cyclic distributions: illustration for stride 2 subrange of cyclic range with extent 15. `P` $= 5$.

```
void prefix(float [[-]] a) {
  Group p = a.grp() ;

  on(p) {
     Range x = a.rng(0) ;

     switch (x.format()) {

       case HPspmd.DIST_DIMENSION :
       case HPspmd.DIST_CYCLIC :

         doublingPrefix(a) ;
         break ;

       default :
         blockPrefix(a) ;
     }
   }
 }
```

Figure 7.20: Optimized parallel prefix for any distribution format.

internal process dimension of size 1. Everything will work, although it might be more efficient to test for `DIST_COLLAPSED` and handle this in a separate, optimized subroutine. *[Yes. Add the code to do this.]*

In the interests of simplifying the presentation, we left a bug in the algorithm of section 7.6.1. It will fail in the case where the original array range is a subrange with *negative* alignment stride. *[Give an example.]* To deal with this case the "recursive" call to compute global prefixes could be replaced with a call that computes the global *suffix*—partial sums of elements that start at the topmost element and increment downwards *[Give a formal definition]*. So stage 2 in Figure 7.17 could be replaced with something like[8]:

```
if(x.str() > 0)
  prefix(t) ;
else
  suffix(t) ;
```

The suffix computation procedure can use essentially the same algorithms as the prefixes, with some loop orders and shift directions reversed. A slightly more elegant solution to this problem will be given in section 7.7.1. Note that there is no need to change the individual block processing to take account of

[8]The inquiry `str()` on `Range` returns the alignment stride of an arbitrary range. There is a related inquiry, `bas()`, that returns the alignment base. These alignment parameters of ranges are distinct from "memory" base and strides returned by the similarly-named inquiries on distributed arrays.

negative alignment strides: the `localBlock()` inquiry does this automatically. The `sub_stp` field will be negative in this case.

There is one distribution format we have not considered here—*block cyclic* distribution format. Section 7.8 will explain one way HPJava can deal with this case.

7.7 Non-local blocks

Section 7.3 described the relatively easy-to-use `localBlock()` inquiry. This inquiry returned block parameters for the local coordinate value in the process dimension associated with the range.

There are other situations, for example inside the implementation of communication functions like `Adlib.remap`, where *all* blocks—local *and* remote—of a subrange must be enumerated.

To deal with these situations there is a general method called `block()`. This method takes one integer argument—a coordinate in the process dimension associated with the range. Like `localBlock()` it returns a `Block` object, defining the layout of distributed array elements in that process.

One might expect the `block()` inquiry should be well-defined for any valid process coordinate. However there are some array alignment options that such a scheme doesn't handle particularly well. Consider the example array sections illustrated in Figure 7.21, which presume the declarations:

```
  Procs p = new Procs1(6) ;
  Dimension d = p.dim(0) ;
```

The first example involves a narrow subrange of a block distributed range. Only two of the six processes hold any elements of the array section. This isn't a problem with for the simple version of block parametrization presented in section 7.3—the `localBlock()` inquiry will simply return "empty" blocks, with `count` set to zero. When translating simple `overall` constructs, the overhead of calling `localBlock()` unconditionally in all processes is not a problem—the method is called at most once in each process anyway. But in situations where *all* blocks—local *and* remote—of a subrange must be enumerated it may become genuinely inefficient to blindly work through every coordinate value, sifting through many empty blocks.

The second example in Figure 7.21, illustrates that a similar problem can arise for the case of a strided section of a *cyclic* range. In this particular example half the processes hold no elements. Again, blindly computing `localBlock()` for all coordinates can lead to inefficiencies, especially in communication functions (where this kind of situation actually arises quite naturally when one is dealing with cyclic ranges).

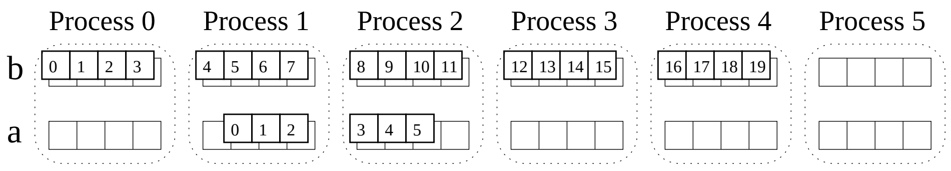
The final example of Figure 7.21 is rather different. It illustrates the case of an array section with a *negative* alignment stride. The natural enumeration order for coordinates of blocks is reversed here, and there are some situations where failure to take this into account can lead to wrong results. In fact we already saw one such example in section 7.6.

7.7.1 The `crds()` method

To allow for these kinds of situation, an new method `crds()` is added to the `Range` class. It takes no arguments, and returns an object of class `Triplet`,

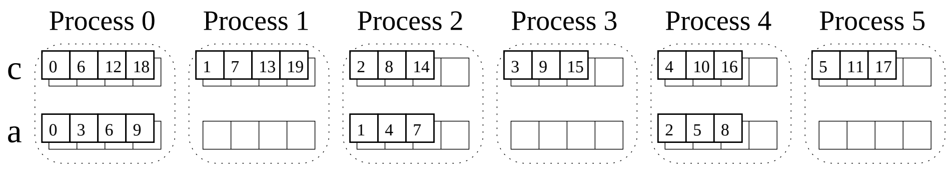
```
Range x = new BlockRange(20, d) ;
float [[-]] b = new float [[x]] ;

float [[-]] a = b [[5 : 10]] ;
```



```
Range x = new CyclicRange(20, d) ;
float [[-]] c = new float [[x]] ;

float [[-]] a = c [[0 : 18 : 2]] ;
```



```
Range x = new BlockRange(20, d) ;
float [[-]] b = new float [[x]] ;

float [[-]] a = b [[19 : 0 : -1]] ;
```

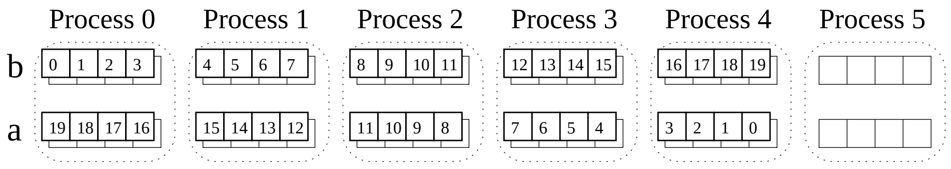


Figure 7.21: Example sections with unusual coordinate ranges.

declared as:

```
class Triplet {
  public int lo, hi ;
  public int stp ;

  public boolean inRange(int n) {...}
}
```

The values of `lo`, `hi` and `stp` define the parameters of some strided interval:

$$\texttt{lo}, \quad \texttt{lo} + \texttt{stp}, \quad \texttt{lo} + 2 \times \texttt{stp}, \quad \ldots, \quad \texttt{lo} + \left(\left\lfloor \frac{\texttt{hi} - \texttt{lo}}{\texttt{stp}} \right\rfloor + 1 \right) \times \texttt{stp}$$

The method `inRange()` returns `true` if and only if its argument is in this interval.

The `Triplet` object returned by the `crds()` method defines a triplet range of coordinates for which the `block()` method is well-defined. The method `localBlock()` introduced earlier can be defined in terms of the more primitive `block()` as follows:

```
Block localBlock() {
  int crd = dim().crd();
  if(crds().inRange(crd))
    return block(crd) ;
  else
    return Block.EMPTY ;
}
```

The `EMPTY` block can be assumed to have fields:

```
EMPTY.count   = 0 ;

EMPTY.sub_bas = 0 ;
EMPTY.sub_stp = 1 ;

EMPTY.glb_bas = 0 ;
EMPTY.glb_stp = 1 ;
```

Calling `block()` for an argument outside the range defined by `crds()` is an error.

As an immediate application of the new methods, consider the parallel prefix of Figure 7.17. We noted at the end of section 7.6.3 that this code fails for the case of a negative alignment stride. An ad hoc solution was given there. A more elegant and systematic approach would be to replace stage 2 with:

```
      Triplet crds = x.crds() ;

      prefix(t [[crds.lo : crds.hi : crds.stp]]) ;
```

If the original range has negative alignment stride, then the value of `crds.stp` will be negative. The recursive call to `prefix` on a negative alignment stride section of `t` effectively implements the suffix computation on `t` itself[9].

7.7.2 Blocks of subranges

As in the case of `localBlock`, there are overloaded versions of `crds()` and `block()` that take `l`, `u`, `s` arguments. Their use will be illustrated in a translation scheme for `overall` constructs parametrized by block cyclic ranges in the next section.

[9]This change will also improve performance of the recursive call if `x` has many empty blocks, as in the first two array sections of Figure 7.21.

7.8 Block cyclic distribution—up a level

High Performance Fortran included the *block cyclic* distribution format, whereby a distributed array range is divided into contiguous blocks of some fixed size, and these blocks are distributed over the process dimension in a cyclically wrapped way. There are various implementation problems associated with this distribution format, and it is not universally accepted that its benefits really justify the added complexity. But we will dedicate this section to describing how block cyclic distribution can be incorporated into the HPJava framework described so far.

An individual process now generally holds many primitive blocks for a range. A straightforward local translation of an `overall` construct may now involve a pair of nested for loops: an inner loop over a single block as before, and an outer loop enumerating the list of blocks allocated to the local process.

One idea for dealing with this is to leave the syntax for dimension splitting unchanged. *But*, when we split the array dimension in Figure 7.12, the distributed range, `d`, associated with dimension-split array is now no longer a process dimension (class `Dimension`). Instead it is a range of class `CyclicRange`. We call this cyclic range the *kernel* of the original block-cyclic range.

The `localBlock()` inquiry is ill-defined on a block-cyclic range, because there are usually many local blocks. We must generalize the recursive translation schemes presented in section 7.3 to use the `crds()` and `block()` inquires (Figure 7.22). In the current example, the "coordinates" returned by the `crds()` inquiry are "virtual coordinates": they are actually global subscripts in the cyclic kernel range. The `overall` construct in the transformed code is no longer a trivial single-pass local loop. On the other hand, it can be reduced by applying the original recursive translation tranformation to the generated `overall`, as illustrated in Figure 7.23[10].

The correctness of this translation in the case where the original array had *negative-stride alignment* (to a block cyclic range) relies on the fact that `t.stp` is negative in this case. Otherwise the locally held blocks would not be enumerated in the correct order, and we would not strictly implement the semantics for `overall` defined way back in section 2.4—viz that if `s` is negative then

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

is equivalent in behaviour to

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

[10]Use of the original recursive translation scheme using `localBlock()`, rather than the generalized scheme using `crds()` and `block()`, is not mandatory for the second application. But it is allowed, and is slightly more convenient.

SOURCE:

```
float [[-]] a;

Range x = a.rng(0);

overall(i = x for l : u : s)
  a [i] = (float) i‘ ;
```

TRANSLATION:

```
float [[-]] a;

Range x = a.rng(0);

float [[-,*]] as = a [[<>]] ;

Range d = as.rng(0) ;

Triplet t = x.crds(l, u, s);

overall(k = d for t.lo : t.hi : t.stp) {
  Block b = x.block(k‘, l, u, s);

  for (int l = 0 ; l < b.count ; l++)
    as [k, b.sub_bas + b.sub_stp * l] =
                        (float) (b.glb_bas + b.glb_stp * l) ;
}
```

Figure 7.22: Recursive translation of an `overall` construct, using `crds()` and `block()`.

SOURCE:

```
float [[-]] a;

Range x = a.rng(0);

float [[-,*]] as = a [[<>]] ;

Range d = as.rng(0) ;

Triplet t = x.crds(l, u, s);

overall(k = d for t.lo : t.hi : t.stp) {
  Block b = x.block(k`, l, u, s);

  for (int l = 0 ; l < b.count ; l++)
    as [k, b.sub_bas + b.sub_stp * l] =
                        (float) (b.glb_bas + b.glb_stp * l) ;
}
```

TRANSLATION:

```
float [[-]] a;

Range x = a.rng(0);

float [[-,*]] as = a [[<>]] ;

Range d = as.rng(0) ;

Triplet t = x.crds(l, u, s);

float [[-,*,*]] ass = as [[<>,:]] ;

Range e = ass.rng(0) ;

overall(m = e for :) {
  Block c = d.localBlock(t.lo, t.hi, t.stp) ;

  for(int n = 0 ; n < c.count ; n++) {
    Block b = x.block(c.glb_bas + c.glb_stp * n, l, u, s) ;

    for (int l = 0 ; l < b.count ; l++)
      ass [m, c.sub_bas + c.sub_stp * n, b.sub_bas + b.sub_stp * l] =
                        (float) (b.glb_bas + b.glb_stp * l) ;
  }
}
```

Figure 7.23: Applying recursive translation a second time, for an `overall` construct parametrized by a block cyclic range.

The parallel prefix implementation of section 7.6 needs only minor modifications to allow for arguments with block cyclic distribution. The enumeration of local blocks should be changed to use `crds()` and `block()` (Figure 7.24). The blocks of a block-cyclically distributed array will now be handled correctly by `blockPrefix()`, and the function will correctly recurse to the cyclic code to handle the kernel prefix[11].

We will sometimes refer to block cyclic ranges as "level 2" ranges. Most other distribution formats are associated with "level 1" ranges. Process dimensions are "level 0" ranges.

[11]It is a slightly esoteric point, but it is arguable that the reason we had to make any changes at all is because the original version used the incoherent `localBlock()` inquiry. If we had stuck to the canonical HPspmd style, we would have been forced to use the coherent `block()` inquiry from the start.

```
void blockPrefix(float [[-]] a) {
  Range x = a.rng(0) ;

  float [[-,*]] as = a [[<>]] ;

  Range z = as.rng(0) ;

  float [[-]] t = new float [[z]], s = new float [[z]] ;

  Triplet crds = x.crds();

  // 1. Do intra-block prefix.  Set ‘t’ elements to sums of blocks

  overall (k = z for crds.lo : crds.hi : crds.stp) {
    Block b = x.block(k‘) ;

    float sum = 0.0 ;
    for (int l = 0 ; l < b.count ; l++) {
      int sub = b.sub_bas + b.sub_stp * l ;

      sum = (as [k, sub] += sum) ;
    }

    t [k] = s [k] = sum ;
  }

  // 2. Do "global prefix"

  prefix(t [[crds.lo : crds.hi : crds.stp]]) ;

  // 3. Add exclusive global prefix to intra-block prefixes in ‘a’

  overall (k = z for crds.lo : crds.hi : crds.stp) {
    Block b = x.block(k‘) ;

    float sum = t [k] - s [k] ;

    for (int l = 0 ; l < b.count ; l++) {
      int sub = b.sub_bas + b.sub_stp * l ;

      as [k, sub] += sum ;
    }
  }
}
```

Figure 7.24: Block-wise version of parallel prefix modified to allow for block-cyclic distribution formats.

Chapter 8

Translation scheme

This chapter describes a basic translation scheme for HPJava. The translation scheme is given in some detail; in effect it represents the most formal and complete definition of the language itself.

8.1 Preliminaries

8.1.1 On distributed array types

A general distributed array type has the form:

$$T\texttt{[[}attr_0, \ldots, attr_{R-1}\texttt{]]}$$

where R is the rank of the array and each term $attr_r$ either consists of a single hyphen, `-`, or a single asterisk, `*`. In principle T can be *any* Java type.

In normal Java, arrays are not considered to be objects, but they are considered to have a class—a class representing an array type. We prefer to avoid making the statement "distributed arrays have a class". If we made this statement it would probably commit us to either:

A. extending the definition of class in the Java base language, or

B. creating genuine Java classes for each type of HPJava array that might be needed.

Class is such a fundamental concept in Java that option A looks like a hard road to follow. Would people expect us, for example, to integrate the complex runtime inquiries on HPJava distributed arrays into some extended version of the Java reflection API? Or into the Java Native Interface, JNI? Such fundamental extensions to Java don't seem very practical.

Option B has its own problems. Presumably the associated class types should capture the rather complicated system of array types we have described for HPJava. Because there is an infinite number of array types, the associated

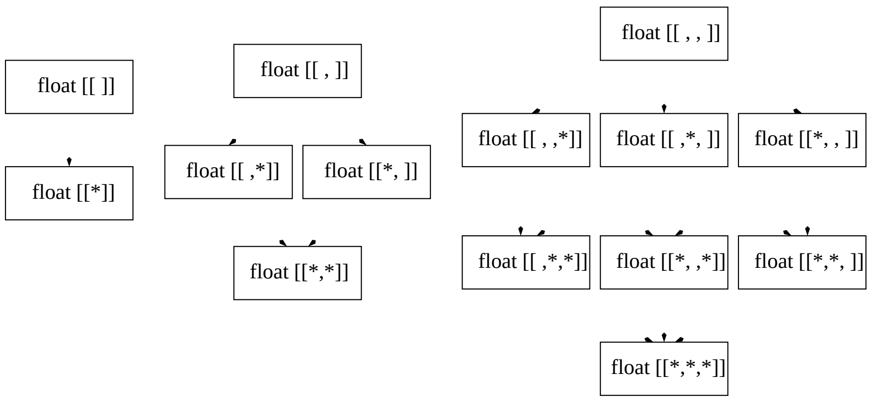


Figure 8.1: Part of the lattice of types for distributed arrays of `float`

classes would certainly have to be created on demand by the translator. Does the translator have to create class files for these automatically generated classes? If so how should these files be managed? Distributed array types have a rather complex, multiple-inheritance-like lattice of subtype relations, illustrated in Figure 8.1. This kind of type-lattice *can* be reproduced in Java by using interface types. But then, when we generate a new array class, we have to make sure all the interfaces it implements directly and indirectly are also defined.

We will finesse these issues by saying that an HPJava distributed array has an "extended class". The extended class concept embraces ordinary Java classes and HPJava distributed array types, as separate concepts.

The fact that a distributed array is not a member of any Java class has a real impact on how a distributed array can be used. For example, a distributed array *cannot* be an element of an ordinary Java array, nor can a distributed array reference be stored in a standard library class like `Vector`, which expects an `Object`. In practise this is not such a drastic limitation as it sounds, because the programmer can always create wrapper classes for particular types of distributed array. For example suppose we need a "stack" of two-dimensional distributed arrays of floating point numbers. We can set this up as follows:

```
class Level implements hpjava.lang.HPspmd {

  public Level(float [[-,-]] arr) {this.arr = arr ; }

  public float [[-,-]] arr ;
}


Range x, y ;

Level [] stack = new Level [S] ;
for (int l = 0 ; l < S ; l++)
  stack [l] = new Level(new float [[x, y]]) ;
```

So the fact that distributed arrays cannot be treated as normal objects is usually a minor inconvenience, not a fundamental limitation[1]. The interface `hpjava.lang.HPspmd` will be discussed in the next section.

8.1.2 HPspmd classes

We will define a translation scheme from HPJava class definitions to standard Java-language class definitions. The existing HPJava translator literally goes through these stages, generating Java source as its output. In the future a more advanced HPJava compiler might directly generate Java byte code, or even machine code. Nevertheless, the early phases of compilation will probably apply transformations similar to the ones described here.

In general only a subset of the classes in an HPJava program will actually use the special syntax of distributed arrays and distributed control constructs. Many of the classes used will be written in standard Java, or may be part of standard Java libraries.

Methods that *do* use HPJava syntax have some special properties. Apart from the fact that they may take distributed array arguments, they also assume that there is a well-defined active process group (APG) at their point of invocation. So in general it is problematic (although not impossible) to invoke HPJava code from a piece of ordinary Java. These problems reflect genuine limitations of the underlying SPMD programming model—it is difficult to *directly* invoke distributed parallel procedures from sequential code.

The HPJava translator tries to make a clear distinction between code that may use HPJava syntax extensions and Java code that may not. It introduces a special interface, `hpjava.lang.HPspmd`, which must be implemented by any class that uses the special syntax.

[1]Note that Fortran 90 also does not allow arrays of arrays to be declared directly. If they are needed they have to be simulated by introducing a derived datatype wrapper, just as we introduced a class here.

We will refer to a class that implements the `hpjava.lang.HPspmd` interface as an *HPspmd class*. Any other class is a *non-HPspmd class*. Likewise, an interface that extends the `hpjava.lang.HPspmd` interface is an *HPspmd interface*, and any other interface is a *non-HPspmd interface*. The extended syntax of HPJava can only be used in methods, constructors and fields declared in HPspmd classes and interfaces. To discourage invocation of HPspmd code from non-HPspmd code, the HPJava translator imposes the following limitations:

1. An HPspmd interface may not extend *any* non-HPspmd interface.
2. An HPspmd class may not implement *any* non-HPspmd interface.
3. An HPspmd class may not extend any non-HPspmd class, except for the mandatory base class, `Object`.
4. An HPspmd class may not override the non-final methods from the base class `Object`.

Ordinary Java code (not processed by the HPJava translator) *may* access members of HPspmd classes, but this requires detailed knowledge of the transformations the HPJava translator applies to function signatures[2] and distributed array variables, and in general this facility should be used very cautiously. The HPJava translator itself will prevent code in non-HPspmd classes from creating instances of, or using members from, HPspmd classes. Of course there is no restriction in the other direction—an HPspmd class can freely use any non-HPspmd Java class, limited only by the normal accessibility rules.

The HPJava translator will also not allow an HPspmd class to override methods from the base class `Object`. For example, an HPspmd class cannot define a `finalize()` method. This is just as well, because the `finalize()` method is invoked asynchronously by the garbage collector. In this context there is no well-defined active process group, and any attempt to, say, invoke a collective operation from a `finalize()` method would almost certainly be a disaster.

It should be very clear that the distinction between HPspmd and non-HPspmd classes is orthogonal to the discussion in section 8.1.1 about the types of distributed arrays themselves. Here we are essentially concerned with what kind of code is allowed to *use* distributed arrays. HPspmd classes are unequivocally Java classes, whereas distributed array types are not classes of any kind.

[2] We will use the term *function* to cover both methods and constructors.

8.2 Static Semantic Checking

8.2.1 Subscripting

In section 5.3 we gave a handful of special rules for distributed array element access. One example is that in

```
overall(i = x for l : u : s) {
  ... a[e_0, ..., e_{r-1}, i, e_{r+1}, ...]  ...
}
```

the expression a must be invariant in the `overall` construct.

In general, proving that an arbitrary expression is invariant throughout a particular block of code is an intractable problem for a compiler. The block of code (our `overall` construct) may, for example, contain method invocations. If the expression we are analysing involves fields or arrays, it is not generally possible to be prove that the method invocation does not change the value of the expression as a side-effect. On the other hand, if one could *not* prove that expressions like a above were invariant, the loss of efficiency in the translated code might be quite spectacular.

For these reasons the translator imposes the following, fairly sweeping, restrictions:

- *Every* distributed-array-valued variable in an HPJava program implicitly has the `final` attribute.

  Note that Java (and HPJava) support an idea of *blank final* variables. A blank final variable is a variable that is declared with the `final` attribute, meaning it is essentially constant, but its value is not initialized in its declaration. Instead it is initialized in a later assignment. Thus a blank final variable is a kind of *single-assignment variable*. A blank final *instance variable*, for example, can (and in fact must) be initialized in a constructor of the class to which it belongs. Restricting distributed array variables to be final variables is thus not as limiting as it may sound, because these array variables may be blank finals. On the other hand this restriction allows accesses to the special arrays of HPJava to be analysed with at least some of the simplicity of earlier, more static, languages like Fortran, and this is a very important advantage. Moreover we suspect that this limitation will rarely be noticed by the programmer—it merely enforces practices that are likely to be commonplace.

- To take full advantage of this limitation, we impose an additional ad hoc rule that the array reference expression in a distributed array element reference or an array section expression *must be a simple identifier*.

  This identifier can refer to a local variable or an instance variable of the current object. Again the actual inconvenience to the programmer due to this restriction is fairly minor. It seems only to enforce common practices. If however, a method really *does* need to process a distributed-array-valued

> field in another object (say) the reference just needs to be copied to a distributed-array-valued *local variable* prior to processing the elements (or computing the section).

We can now very cleanly enforce the original requirements on subscripting in `overall` and `at` constructs, by applying a simple static check that if a distributed index i appears in a subscript of a distributed array a, the variable a should have been declared *outside* the scope of i, and (if it is a blank final) a should not be assigned *inside* the scope of i.

8.2.2 Initialization of fields

Any HPspmd code needs to execute in the context of a well-defined active process group (APG). In the current implementation of HPJava, the APG is defined only in the bodies of methods and constructors of HPspmd classes. There is no APG in effect for code that executes outside the bodies of these functions. So in particular executable HPspmd code is not allowed in initializers for fields or class initialization blocks, including `static` initialization blocks.

This means that:

> Fields whose type is an HPspmd class type or an HPspmd interface type (including the built-in classes `Group` and `Range`) cannot have initializers in their declarations.

Similarly:

> Fields that have distributed array type cannot have initializers in their declarations.

Since distributed-array-valued fields, like all distributed-array-valued variables, are implicitly `final`, it follows that they are also implicitly *blank final*. In the case of instance variables, this means they must be initialized in the constructors of the class. In general Java requires that blank final *static* fields be initialized in static initialization blocks. Since these blocks cannot contain executable HPspmd code, it follows that a distributed-array-valued *static* field could never be initialized. Consequently:

> HPJava forbids distributed-array-valued fields from having the `static` attribute.

8.3 Pre-translation

The current HPJava translator has (at least) two translation phases. The first phase, *pre-translation*, reduces the input HPJava program to some other, equivalent HPJava program, coded in a restricted subset of the full HPJava language. The translation phase proper follows pre-translation, and converts the intermediate HPJava program to a standard Java program.

Roles of the pre-translator include:

- As explained above, the input program is reduced to a *restricted form* that is suitable for processing in the translation phase proper. This restricted form, which is described in the next section, disallows certain complex expressions involving distributed arrays. The pre-translator reduces these complex expressions to simpler expressions, by introducing temporaries if necessary.
- To do the required transformations in an effective way, the pre-translator will perform some scalar dependence analysis on the source program. The pre-translator may also annotate (the translator's internal representation of) the program with information obtained in this analysis, for use by later phases.
- The pre-translator may insert code for *run-time checking* of various HP-spmd rules described in chapter 5.

8.3.1 Restricted Form

So far as the HPJava translator is concerned, a program is in *restricted form* if it respects certain limits on the complexity of expressions involving distributed arrays. First we give two definitions:

- *Constructive distributed array expressions* are:

  a) distributed array creation expression,

  b) distributed-array-valued method invocation expression,

  c) array section expression, and

  d) array restriction expression.

- An expression is a *simple expression* if it is either a simple variable name (identifier) or a constant expression.

Now the restrictions on expressions are:

1. A constructive distributed array expression may *only* appear as the right-hand-side of a top-level assignment. Here, a *top-level assignment* is defined to be an assignment expression appearing as a *statement expression* in a program block. Note that this specifically excludes assignments appearing

in for-loop headers. Also note that a variable declarator with an initializer is *not* an assignment expression, so the restricted form does not allow constructive distributed array expressions to appear as initializers in declarations.

2. All of the following *must* be simple expressions:

   a) the target object in accesses to distributed-array-valued fields,

   b) the boolean expression in a distributed-array-valued conditional expressions, and

   c) the array operand in array-restriction expression,

   d) the range and group parameters of distributed array creation expressions,

   e) the range expression in the header of `overall` and `at` constructs.

3. The group expression in the header of an `on` construct must be a simple expression *that is invariant in the body of the construct.*

4. The index expression in the header of an `at` construct must be a simple expression that is invariant in the body of the construct.

In principle reducing a program to restricted form is a straightforward matter of precomputing subexpressions that break the rules above, and saving their values in temporary variables introduced by the pre-translator. Of course the subexpressions of the original expression are replaced by accesses to the temporary.

As a special case, an invocation of a method that returns a distributed array, but which appears alone as a statement expression (with the result value ignored), may be pre-translated to an assignment to a temporary variable whose value is subquently ignored.

One problematic area is arguments and prefixes in explicit constructor invocations. An explicit constructor invocation[3] must be the first statement in the body of a constructor, so we cannot place an assignment to a temporary before it. The easiest way to handle this situation is to add an ad hoc rule limiting the complexity of expressions that may appear in this context in the HPJava source program—for example we may insist that any distributed-array-valued expressions appearing as a subexpression in an explicit constructor invocation statement must be a simple expression.

Figures 8.2 and 8.3 illustrate the process of expression simplification. The source program is an example taken from section 4.8. In this example pre-translation shifts three kinds of constructive array expression—distributed array creation, array section and array restriction—into top-level assignments. In the case of the array creation expressions it simply has to split the array declarations with initializers to blank declarations plus an assignment. In the case of the sections and restrictions, it has to introduce temporaries.

[3]Which should not be confused with the implicit constructor invocation in an object creation expression.

```
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 8.2: Example source program prior to pre-translation.

8.3.2 Expression simplification

The transformations needed to simplify expressions to reduce a program to restricted form are relatively straightforward. For completeness we will outline in schematic form the algorithm applied by the current HPJava translator.

One principle of the HPspmd model is that parts of a program that are written purely in the base language—not using special HPspmd features—should, as far as practical, be compiled in exactly the same was as they would be if they appeared in a local sequential program. We will adapt this requirement as a guiding principle in defining an algorithm for simplifying expressions. Subexpressions that are pure Java expressions, not involving distributed arrays should, as far as possible, be copied unchanged to the translated program. In particular the generated standard Java program should not break up the evaluation of such subexpressions by introducting temporaries. Note however that a purely Java sub-expression may, as a whole, be *assigned* to a temporary, if it was originally embedded in another expression that *did* involve special HPJava terms.

In our scheme certain subexpressions are "marked" according to the simplification rules. These are expressions that *must* be precomputed, to meet the requirements of a subsequent translation phase. In general, if the marked subexpressions are to be precomputed, dependence relations amongst subexpressions may force other, unmarked subexpressions to also be precomputed.

Marked subexpressions

According to section 8.3.1 the marked subexpressions are:

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

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

    ... initialize ‘a’, ‘b’

    overall(i = x for :)
        overall(j = y for :) {
            float [[-]] __$t3 ;
            {
                float [[-]] __$t2 ;
                {
                    __$t2 = a [[i, :]] ;
                }
                __$t3 = __$t2 / j ;
            }
            float [[-]] __$t1 ;
            {
                float [[-]] __$t0 ;
                {
                    __$t0 = b [[:, j]] ;
                }
                __$t1 = __$t0 / i ;
            }

            c [i, j] = Adlib.dotProduct(__$t3, __$t1) ;
        }
}
```

Figure 8.3: Example source program after pre-translation.

1. Constructive array expressions that are *not* on the right-hand-side of a top-level assignment.
2. Any of the following that are *not* simple expressions:
   a) the target object in accesses to distributed-array-valued fields,
   b) the boolean expression in a distributed-array-valued conditional expressions, and
   c) the array operand in array-restriction expression,
   d) the range and group parameters of distributed array creation expressions,
   e) the range expression in the header of `overall` and `at` constructs.
3. The group expression in the header of an *on* construct, if it is *not* a simple expression that is invariant in the body of the construct.
4. The index expression in the header of an *at* construct, if it is *not* a simple expression that is invariant in the body of the construct.

Schematic algorithm

A schematic algorithm for simplification is given in Figure 8.4. The algorithm is written as a recursive function, *simplify*. The main input to this function is an expression term e, and the main output is a simplified expression term e'. This is accompanied by an output sequence of statements, *INITS*, which declares and initializes any temporaries that were introduced in the simplification of e.

The algorithm traverses sub-expressions in *right to left* order. If a sub-expression must be precomputed, the associated code will have to be moved in front of other sub-expressions currently to its left. If there are no dependencies between the sub-expression and left siblings, this is fine. But if there is a dependency, the sibling code must also be precomputed, to respect the original order of evaluation of the dependent subexpressions.

Dependencies are detected using *access sets*. These record uses and definitions of individual variables, or categories of variables, within expressions. The *simplify* function computes these access sets on the fly and returns two of them in the values *ACCESS* and *I_ACCESS*. The set *ACCESS* is the set of program variable accesses in the translated expression e'. The set *I_ACCESS* is the set of program variable accesses in the "precomputation" code—code for initialization of temporaries defined in *INITS* (both sets exclude accesses to temporaries themselves).

The final input to the *simplify* function—the parameter *I_ACESSS_RIGHT*—is the set of variable accesses in "precomputation" code already generated by subexpressions *to the right* of the current expression. If a dependency is discovered between the translated version of the *current* expression and the code we already *know* has to moved "to the left", the whole of the current expression

```
simplify($INITS$, $e'$, $I\_ACCESS$, $ACCESS$,
        $e$, $I\_ACCESS\_RIGHT$) {

    OUT: $INITS$, $e'$, $I\_ACCESS$, $ACCESS$
    IN: $e$, $I\_ACCESS\_RIGHT$

    $INITS$ = “”

    $e'$ = $e$

    $I\_ACCESS$ = {}
    $ACCESS$ = direct accesses associated with this node

    foreach (subexpression $e_i$ of $e$, enumerated right to left) {

        simplify($INITS_i$, $e'_i$, $I\_ACCESS_i$, $ACCESS_i$,
                $e_i$, $I\_ACCESS$)

        $INITS$ = $INITS_i$ + $INITS$

        Replace $e_i$ in $e'$ with $e'_i$

        $I\_ACCESS$ = $I\_ACCESS \cup I\_ACCESS_i$
        $ACCESS$ = $ACCESS \cup ACCESS_i$
    }

    if($e$ is a marked subexpression or
       there is a dependency between $ACCESS$ and $I\_ACCESS\_RIGHT$) {

        // $t$ is a new temporary name; $T$ is type of $e$

        $INITS$ = “$T$ $t$ ;
                  {
                       $INITS$
                       $t$ = $e'$ ;
                  } ”

        $e'$ = “$t$”

        $I\_ACCESS$ = $I\_ACCESS \cup ACCESS$
        $ACCESS$ = {}
    }
}
```

Figure 8.4: Schematic algorithm for expression simplification.

must also be "moved to the left", and precomputed *before* the code already scheduled for movement.

In processing a "top-level" expression we invoke the *simplify* algorithm, passing it the expression term e to be simplified, and an empty value for *I_ACCESS_-RIGHT*.

Data structures

In the pseudocode of Figure 8.4 code fragments are written as Java-like strings of program text (with `+` as the concatenation operator). In a practical implementation these would be manipulated as nodes in an abstract syntax tree.

Access sets will probably be implemented as two separate sets: the set of *uses* and the set of *defs*. In the initial HPJava translator, each set is in turn implemented using four components:

1. A hash set of local variable descriptors.
2. A hash set of field descriptors (which may be considered to represent "all instances" of the field involved if they are instance variable descriptors).
3. A flag specifying that "all fields" must be assumed to be included in the set. If this flag is `true`, the previous component can be discarded or ignored.
4. A flag specifying "all array elements" must be assumed to be included in the set.

We will consider two access sets to be *dependent* if they share a common variable (or category of variables), and at least one of the accesses is a *def*.

To complete the *simplify* algorithm we still need to define, for each kind of expression in the language, the relevant set of subexpressions (enumerated in the `foreach`), and also to define the "direct accesses associated with" this kind of expression.

Expression nodes

To apply the *simplify* algorithm we need to define the relevant sub-expressions of, and "direct accesses" in, the various kinds of expression node in the language. The non-trivial cases—the cases that have non-empty sets of direct accesses—are:

- Updates:
  - simple assignment
  - auto-increment/decrement (`++`, etc)
  - compound assignment (`+=`, etc)
- References:

  - named expression
  - field reference with expression prefix
  - field reference with `super` prefix
  - array element reference (distributed array or Java array)

- "Wildcards":
  - method invocations

The "reference" expressions here will be treated differently, according to whether they appear as the variable operand of an update, or anywhere else. A "named expression" is any expression with the syntax of one or more identifiers separated by periods.

No other kinds of expression involve direct accesses, and all other kinds of expression have "obvious" sub-expression lists, so far as the *simplify* algorithm is concerned. They will not be discussed explicitly in this section.

First we define the "subexpressions of a reference". These are:

- If the reference is a named expression, the subexpressions are:
  - the prefix of named expression, if it is non-null and is an expression (not a type).
- If the reference is a field reference with a general expression prefix, the subexpressions are:
  - the prefix expression of the field reference.
- If the reference is a field reference with a `super` prefix, there are no subexpressions.
- If the reference is an array element reference, the subexpressions are:
  - the target array of the element reference,
  - the integer subscript(s) of the element reference, and
  - the "shift" expressions in any shifted-distributed-index subscripts of the element reference.

Now table 8.1 enumerates the remaining nodes that have "non-obvious" subexpression lists so far as the *simplify* algorithm is concerned. Table 8.2 enumerates the nodes that have non-empty sets of "direct accesses". In these tables LHS stands for the left-hand-side operand of an assignment; RHS stands for the right-hand-side operand.

Expression node	**Subexpressions**
Simple assignment	Subexpressions of reference on LHS; RHS of assignment.
Compound assignment	Subexpressions of reference on LHS; RHS of assignment.
Autoincrement/decrement	Subexpressions of operand reference.

Table 8.1: Subexpressions of various update nodes, for *simplify* algorithm. See text for definition of "subexpressions of a reference".

Expression node	**Direct *uses***	**Direct *defs***
Simple assignment with named expression on LHS		Variable descriptor (local or field) associated with LHS.
Simple assignment with field reference as LHS		Field descriptor associated with LHS.
Simple assignment with array element on LHS		"All array elements".
Compound assignment with named expression as LHS	Variable descriptor (local or field) associated with LHS.	Variable descriptor (local or field) associated with LHS.
Compound assignment with field reference as LHS	Field descriptor associated with LHS.	Field descriptor associated with LHS.
Compound assignment with array element on LHS	"All array elements".	"All array elements".
Autoincrement/decrement with named expression as operand	Variable descriptor (local or field) associated with operand.	Variable descriptor (local or field) associated with operand.
Autoincrement/decrement with field reference as operand	Field descriptor associated with operand.	Field descriptor associated with operand.
Autoincrement/decrement with array element as operand	"All array elements".	"All array elements".
Named expression, not the variable of an update.	Variable descriptor (local or field) associated with named expression.	
Field reference, not the variable of an update.	Field descriptor associated with field reference.	
Array element reference, not the variable of an update.	"All array elements".	
Any method invocation	"All fields"; "All array elements".	"All fields"; "All array elements".

Table 8.2: Direct accesses of various expression nodes, for *simplify* algorithm.

Compound assignments

Compound assignment needs special treatment in the *simplify* algorithm. Assuming $\oplus$ is one of the supported operators, a problem arises in

$$v \ \oplus\texttt{=}\ e$$

if e includes a subexpression e_i that must be precomputed, *and* e_i may define v (i.e. subexpression e_i has a *def* of v in its access set). The underlying problem is that there is then an anti-dependence from v to e_i *and* an output dependence from e_i to v. Hence one cannot move the code for e_i without splitting the compound assignment.

In this case we are essentially forced to translate the assignment to

$$v \ \texttt{=} \ v \ \oplus \ e$$

thus allowing the use of v on the RHS to be precomputed. But then if v itself is a non-trivial expression you may have to mark its subexpressions to avoid their repeated computation.

If the expression e has the form $e_l \ \oplus\texttt{=}\ e_r$, then the `foreach` loop in the body of the *simplify* algorithm may be replaced by the code given in Figure 8.5.

8.3.3 Run-time checks

It may also be natural for the pre-translation phase to introduce at least some of the run-time checks associated with the rules in chapter 5. An example is shown in Figure 8.6. The *ASSERT* macro throws an exception if its argument is boolean `false`. In the figure we have not yet applied the other transformations (expression simplifications) associated with pre-translation.

Since the pre-translation phase is supposed to generate some *use-def* information for the source program it may be possible to eliminate many of these checks at this early stage. In the example given, it seems realistic to assume that automatic analysis could detect that, for example `x.dim()` is `p.dim(0)`, and that `apg` is `p` at the point the statement

```
ASSERT(apg.contains(x.dim()))
```

appear, and thus that the check is superfluous. In fact in this example a moderately sophisticated analysis may well prove that all the assertions are safe at compile-time.

```
$simplify$($INITS_r$,  $e'_r$,  $I\_ACCESS_r$,  $ACCESS_r$,
          $e_r$,  $I\_ACCESS$)

$INITS$ = $INITS_r$  +  $INITS$

Replace $e_r$ in $e'$ with $e'_r$

$I\_ACCESS$ = $I\_ACCESS \cup I\_ACCESS_r$
$ACCESS$ = $ACCESS \cup ACCESS_r$

if($I\_ACCESS_r$ contains a $def$ of LHS variable) {
    Mark any subexpressions of the reference $e_l$ that are not simple expressions

    $splitting$ = true
}

$e'_l$ = $e_l$

foreach (subexpression $e_i$ of $e_l$, enumerated $right\ to\ left$) {

    $simplify$($INITS_i$,  $e'_i$,  $I\_ACCESS_i$,  $ACCESS_i$,
              $e_i$,  $I\_ACCESS$)

    $INITS$ = $INITS_i$  +  $INITS$

    Replace $e_i$ in $e'_l$ with $e'_i$

    $I\_ACCESS$ = $I\_ACCESS \cup I\_ACCESS_i$
    $ACCESS$ = $ACCESS \cup ACCESS_i$
}

if($splitting$) {

    // $t_l$ is a new temporary name; $T_l$ is type of $e_l$

    $INITS$ = “$T_l$  $t_l$  ;
                 $t_l$  =  $e'_l$  ;  ”  +  $INITS$

    $e'$  =  “$e'_l$  =  $t_l \oplus e'_r$”

} else {
    Replace $e_l$ in $e'$ with $e'_l$
}
```

Figure 8.5: Replacement code for `foreach` loop in *simplify*, to handle the compound assignment $e_l \oplus$= e_r.

```
Procs3 p = new Procs3(P, P, P) ;

ASSERT(apg.contains(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’

    ASSERT(apg.contains(x.dim()))
    ASSERT(c.rng(0).containsLocations(x))
    ASSERT(a.rng(0).containsLocations(x))
    overall(i = x for :) {
        ASSERT(apg.contains(y.dim()))
        ASSERT(c.rng(1).containsLocations(y))
        ASSERT(b.rng(1).containsLocations(y))
        overall(j = y for :)
            c [i, j] = Adlib.dotProduct(a [[i, :]] / j, b [[:, j]] / i) ;
    }
}
```

Figure 8.6: Example source program after adding run-time checks.

8.4 Translation

8.4.1 Translation functions

We will define several "translation functions".

First, function, $\mathbf{T}\,[e]$, on expression terms returns the result of translating an expression e, assuming that the expression is not a distributed array.

Translation functions for distributed-array-expressions are more complicated. In section 8.3.1 we defined a subset of *constructive* distributed-array-valued expressions. The remaining *non-constructive* distributed-array-valued expressions are:

a) distributed-array-valued local variable access,

b) distributed-array-valued field access,

c) conditional expression, in which the second or third operands are distributed arrays.

d) assignment expression, in which the left-hand operand is a distributed-array-valued variable.

The constructive expressions only appear in restricted contexts and do not have translation functions in their own right (instead they are handled as part of the translation of a top-level assignment). For *non-constructive* distributed-array-valued expressions there are $2{+}R$ separate parts of the evaluation: $\mathbf{T}_{\mathrm{dat}}\,[e]$, $\mathbf{T}_{\mathrm{bas}}\,[e]$ and $\mathbf{T}_0\,[e]$, $\ldots$, $\mathbf{T}_{R-1}\,[e]$, where R is the rank of the array. The interpretation of these separate terms will be given in the following sections.

Finally the translation function for statements, $\mathbf{T}\,[S\,|p]$, translates the statement or block S in the context of p as active process group. In the schemas given below for translation of statements we will just use the name *apg* to refer to the effective active process group. Hence a schema of the form

SOURCE:

$$S$$

TRANSLATION:

$$S'$$

should be read more precisely as

SOURCE:

$$s \quad \equiv \quad S$$

TRANSLATION:

$$\mathbf{T}\,[s\,|\mathit{apg}] \quad \equiv \quad S'$$

SOURCE:

T `[[`$attr_0$`,` ... `,` $attr_{R-1}$`]]` a `=` e `;`

TRANSLATION:

T `[]` a'_{dat} `=` $\mathbf{T}_{\mathrm{dat}}\,[e]$ `;`

`ArrayBase` a'_{bas} `=` $\mathbf{T}_{\mathrm{bas}}\,[e]$ `;`

$\mathit{DIMENSION\_TYPE}(attr_0)$ a'_0 `=` $\mathbf{T}_0\,[e]$ `;`
...
$\mathit{DIMENSION\_TYPE}(attr_{R-1})$ a'_{R-1} `=` $\mathbf{T}_{R-1}\,[e]$ `;`

where:

T is a Java type, the element type of the declared array,
R is the rank of the declared array,
each term $attr_r$ is a single hyphen, `-`, or a single asterisk, `*`,
the identifier a is the name of the declared array in the source program,
the expression e is an optional initializer in the source program,
$a'_{\mathrm{dat}} = \mathit{TRANS\_ID}(a)$,
$a'_{\mathrm{bas}} = \mathit{TRANS\_ID\_BAS}(a)$,
$a'_r = \mathit{TRANS\_I\_DIM}(a, r)$, and
the macros *TRANS_ID*, *TRANS_ID_BAS*, *TRANS_ID_DIM* and
DIMENSION_TYPE are defined in the text.

Figure 8.7: Translation of a distributed-array-valued variable declaration.

8.5 Translating variable declarations

The general scheme for translating declaration of a field or local variable holding a distributed array reference is illustrated in Figure 8.7. The single variable in the source program is converted to $2+R$ variables in the output program, where R is the rank of the array.

The macro *TRANS_ID* transforms the identifier of the field in a way that encodes the dimension signature of the result. This encoding is necessary in order that the HPJava type signature of a field can be reconstructed from the class file of a translated (and compiled) class. Note that the *element type* of the distributed array field is known from the type of Java array in translated field—this information is automatically encoded in the Java class file.

The values returned by the *TRANS_ID* macro and the macros *TRANS_ID-_BAS* and *TRANS_ID_DIM* all start with the the original a string, followed by the suffix separator "`__$`". This is followed by a string characteristic of the individual macro, described next.

The characteristic string for *TRANS_ID*(a) is a string of R letters, each of which is "`D`" (for a distributed dimension) or "`S`" (for a sequential dimension). The characteristic string for *TRANS_ID_BAS*(a) is "`bas`". The characteristic string for *TRANS_ID_DIM*(a, r) is the decimal representation of the constant integer value r.

The variable *TRANS_ID*(a) will hold a reference to the Java array containing the locally held elements of the distributed array. This is the value that would be returned by the inquiry a`.dat()` in the source program.

The *TRANS_ID_BAS*(a) will hold an instance of the class `ArrayBase`. Instances of this class contain a group object (the value that would be returned by the inquiry a`.grp()` in the source program) and an offset from the start of the Java array, where the first locally held element of the distributed array is stored (the value that would be returned by the inquiry a`.bas()` in the source program).

The R variables *TRANS_ID_DIM*$(a, 0)$ ..., *TRANS_ID_DIM*$(a, R-1)$ will hold descriptors for the dimensions of the arrays. The macro *DIMENSION_-TYPE* is defined as

$$\mathit{DIMENSION\_TYPE}(\mathit{attr}_r) \equiv \texttt{ArrayDim}$$

if the term attr_r is a hyphen, `-`, or

$$\mathit{DIMENSION\_TYPE}(\mathit{attr}_r) \equiv \texttt{SeqArrayDim}$$

if the term attr_r is an asterisk, `*`. Instances of the class `ArrayDim` contain a distributed range object and an associated "memory stride"—the values that would be returned by the inquiries a`.rng(`r`)` and a`.str(`r`)`, respectively, in the source program. These values are stored in the fields `range` and `stride` of `ArrayDim`. The class `SeqArrayDim` is a subclass of `ArrayDim` that contains extra information that can be used to simplify the computations associated with subscripting a sequential array dimension.

If, for example, a class in the source program has a field:

```
float [[-,-,*]] bar ;
```

the translated class can be assumed to have the five fields:

```
float [] bar__$DDS ;

ArrayBase bar__$bas ;

ArrayDim bar__$0 ;
ArrayDim bar__$1 ;
SeqArrayDim bar__$2 ;
```

SOURCE:

$$T\ \ f\texttt{(}U_0\ \ v_0\texttt{,}\ \ \ldots\texttt{,}\ \ U_{N-1}\ \ v_{N-1}\texttt{)}\ \ \texttt{\{}S\texttt{\}}$$

TRANSLATION:

```
T  f(TRANS_PARAMS(U_0 v_0,...,U_{N-1} v_{N-1}), Group p) {
     T [S |p]
}
```

where:

T is a Java type,
the identifier f is the name of the method,
each of the terms U_0, ..., U_{N-1} is a Java type or a distributed array type,
v_0, ..., v_{N-1} are parameter names appearing in the original program,
S is a block of statements in the original program,
p is the name of a new parameter, and
the macro *TRANS_PARAMS* is defined in the text.

Figure 8.8: Translation of method declaration.

8.6 Translating method declarations

There are two cases to consider. The case where the result of the method is not a distributed array, and the case where the result is a distributed array. First we consider the case where the result is *not* a distributed array.

The general scheme is illustrated in Figure 8.8. This scheme is modified in trivial ways if the method has a `void` result, or involves other modifiers (they are copied to the translated code).

The macro *TRANS_PARAMS* evaluates to a list of formal parameters. We will define it in terms the simpler macro *TRANS_PARAM* which operates on a single formal parameter declaration:

$$\begin{aligned}&\mathit{TRANS\_PARAMS}(U_0\ \ v_0, \ldots, U_{N-1}\ \ v_{N-1}) \equiv \\ &\qquad \mathit{TRANS\_PARAM}(U_0\ \ v_0),\ \ \ldots,\ \ \mathit{TRANS\_PARAM}(U_{N-1}\ \ v_{N-1})\end{aligned}$$

Now, if U is not a distributed array type, we have

$$\mathit{TRANS\_PARAM}(U\ \ v) \equiv \quad U\ \ v$$

Otherwise, if U has the form

$$T\ \ \texttt{[[}\mathit{attr}_0\texttt{,}\ \ \ldots\texttt{,}\ \ \mathit{attr}_{R-1}\texttt{]]}\ \ a\ \ \texttt{;}$$

(where as usual T is a Java type and as usual each term attr_r is a single hyphen, `-`, or a single asterisk, `*`) then the macro *TRANS_PARAM* is defined by

$$\begin{aligned}&\mathit{TRANS\_PARAM}(U\ \ v) \equiv \\ &\qquad T\ \ \texttt{[]}\ \ a'_{\mathrm{dat}}\texttt{,}\ \texttt{ArrayBase}\ a'_{\mathrm{bas}}\texttt{,} \\ &\qquad\qquad \mathit{DIMENSION\_TYPE}(\mathit{attr}_0)\ \ a'_0\texttt{,}\ \ \ldots\texttt{,} \\ &\qquad\qquad\qquad \mathit{DIMENSION\_TYPE}(\mathit{attr}_{R-1})\ \ a'_{R-1}\end{aligned}$$

where a'_{dat}, a'_{bas}, a'_{grp}, and $a'_0, \ldots, a'_{R-1}$ are new formal parameter names that may be obtained from a using the prescription given in Figure 8.7. The macro *DIMENSION_TYPE* is defined in section 8.5.

In other words, each distributed array parameter is split into $2 + R$ parameters.

The final paramter p added by the translator will hold the value of the *active process group* in effect at the point of invocation of the method.

If, for example, a class in the source program has a method:

```
void foo(int [[-,-,*]] bar) {...}
```

the translated class can be assumed to have the method:

```
void foo(int [] bar__$DDS, ArrayBase bar__$BAS,
         ArrayDim bar__$0, ArrayDim bar__$1, SeqArrayDim bar__$2,
         Group p) {...}
```

SOURCE:

```
    $T$ [[$attr_0$, ..., $attr_{R-1}$]] $f$($U_0$ $v_0$, ..., $U_{N-1}$ $v_{N-1}$) {
        $S$
    }
```

TRANSLATION:

```
  $T$ [] $TRANS\_ID(f)$(DAD $d$,
                        $TRANS\_PARAMS(U_0\ v_0, \ldots, U_{N-1}\ v_{N-1})$,
                        Group $p$) {
      $\mathbf{T}[S\,|p]$
  }
```

where:

T is a Java type,
R is the rank of the returned array,
each term $attr_r$ is a single hyphen, -, or a single asterisk, *,
the identifier f is the name of the method,
each of the terms U_0, ..., U_{N-1} is a Java type or a distributed array type,
v_0, ..., v_{N-1} are parameter names appearing in the original program,
S is a block of statements in the original program,
d and p is the names of new parameters, and
the macros *TRANS_ID* and *TRANS_PARAMS* is defined in the text.

Figure 8.9: Translating declaration of method returning a distributed array.

8.6.1 Methods that return distributed arrays

The scheme for translating a distributed-array-valued method declaration is illustrated in Figure 8.9.

The macro *TRANS_ID* encodes the dimension signature of the returned array and is defined in section 8.5. The macro *TRANS_PARAMS* was defined earlier in this section.

Instances of the class `DAD` (the initials stand for *Distributed Array Descriptor*) contain a reference to an `ArrayBase` object, and a vector of R instances of `ArrayDim`. These values are stored in fields `base` and `dimensions`, respectively. The DAD object will be created in the calling program and passed to the translated method. Before the method returns, it will store the parameters describing the layout of the distributed array result in these fields (see sections 8.17.1 and 8.17.2).

SOURCE:

```
on (p) S
```

TRANSLATION:

```
if (p.member()) {
    T[S|p]
}
```

where:

p is a simple expression in the source, and
S is a statement in the source program.

Figure 8.10: Translation of `on` construct.

8.7 Translating `on` constructs

A translation for the `on` construct is given in Figure 8.10. Note that pre-translation will have reduced the expression p to a simple expression, which itself requires no translation.

SOURCE:

```
at (i = x[n]) S
```

TRANSLATION:

```
Location l =  x.location(n) ;

Dimension d = x.dim() ;
if (d.crd() == l.crd) {
    Group p = ((Group) apg.clone()).restrict(d) ;

    T [S |p]
}
```

where:

i is an index name in the source program,
x and n are simple expressions in the source program,
S is a statement in the source program, and
l, d and p are the names of new variables.

Figure 8.11: Translation of `at` construct.

8.8 Translating `at` constructs

A translation for the `at` construct is given in Figure 8.11. Note that pre-translation will have reduced x and n to simple expressions.

The coordinate and local subscript associated with the specified location is returned by the method, `location()`, which is a member of the `Range` class. It takes one argument, the global subscript, and returns an object of class `Location`, declared as:

```
class Location {
  public int crd ;
  public int sub ;
}
```

The *local subscript* for the index i is the value of l`.sub`. This value is used in subscripting distributed arrays.

The *global index* for the index i is the value of n. This value is used in evaluating the global index expression i`‘`. Note that pretranslation has reduced the expression n to a simple expression that is constant in the body of the construct (by introducing a temporary if necessary).

The *shift step* for the index i is defined to be the value of x`.str()`. This value is used in computation of offsets associated with shifted index subscripts.

SOURCE:

```
overall ($i$ = $x$ for  $e_{\mathrm{lo}}$ : $e_{\mathrm{hi}}$ : $e_{\mathrm{stp}}$) $S$
```

TRANSLATION:

```
Block $b$ =  $x$.localBlock($\mathbf{T}[e_{\mathrm{lo}}]$,  $\mathbf{T}[e_{\mathrm{hi}}]$,  $\mathbf{T}[e_{\mathrm{stp}}]$) ;

Group $p$ = ((Group) $apg$.clone()).restrict($x$.dim()) ;

for (int $l$ = 0 ; $l$ < $b$.count ; $l$++) {
    int $sub$ = $b$.sub_bas + $b$.sub_stp * $l$ ;
    int $glb$ = $b$.glb_bas + $b$.glb_stp * $l$ ;

    $\mathbf{T}[S\,|p]$
}
```

where:

i is an index name in the source program,
x is a simple expressions in the source program,
e_{lo}, e_{hi}, and e_{stp} are expressions in the source,
S is a statement in the source program, and
b, p, l, sub and glb are names of new variables.

Figure 8.12: Translation of `overall` construct.

8.9 Translating `overall` constructs

A translation for the `overall` construct is given in Figure 8.12. The `localBlock()` method and the `Block` class have been discussed at length in section 7.3.

The *local subscript* for the index i is the value of *sub*. This value is used in subscripting distributed arrays.

The *global index* for the index i is the value of *glb*. This value is used in evaluating the global index expression i`‘`.

The *shift step* for the index i is defined to be the value of x`.str()`. This value is used in computation of offsets associated with shifted index subscripts.

SOURCE:

$$e \quad \equiv \quad i`$$

TRANSLATION:

$$\mathbf{T}\,[e] \quad \equiv \quad glb$$

where:

i is an index name in the source program, and
glb is the global index variable for the index i.

Figure 8.13: Translation of global index for i.

8.10 Translating global index expression

The scheme is illustrated in Figure 8.13. The global index variable associated with a distributed index is defined in sections 8.8 and 8.9.

SOURCE:

$$e \equiv a$$

TRANSLATION:

$$\begin{aligned}
\mathbf{T}_{\mathrm{dat}}[e] &\equiv a'_{\mathrm{dat}} \\
\mathbf{T}_{\mathrm{bas}}[e] &\equiv a'_{\mathrm{bas}} \\
\mathbf{T}_0[e] &\equiv a'_0 \\
&\dots \\
\mathbf{T}_{R-1}[e] &\equiv a'_{R-1}
\end{aligned}$$

where:

> a is an array name in the source program, rank R, and
> a'_{dat}, a'_{bas}, and $a'_0, \dots, a'_{R-1}$ are corresponding
> names of variables in the translated program.

Figure 8.14: Translation of a distributed-array-valued variable access.

8.11 Translating variable accesses

We only need to consider specially the case where the variable is a distributed array. The general scheme is illustrated in Figure 8.14. This applies to the case of a variable that is a *simple identifier*—a local variable, method or constructor argument, or instance variable of the current object. The case of a field reference with an object prefix will be covered in ...

The names a'_{dat}, a'_{bas}, and $a'_0, \ldots, a'_{R-1}$ are the names introduced by the translator when translating the corresponding field, local variable declaration or formal parameter in the source program (see sections 8.5 and 8.6).

SOURCE:

$$e \quad \equiv \quad e_t \texttt{ = } e_s$$

TRANSLATION:

$$\begin{array}{rcl}
\mathbf{T}_{\text{dat}}\,[e] & \equiv & \mathbf{T}_{\text{dat}}\,[e_t] \texttt{ = } \mathbf{T}_{\text{dat}}\,[e_s] \\
\mathbf{T}_{\text{bas}}\,[e] & \equiv & \mathbf{T}_{\text{bas}}\,[e_t] \texttt{ = } \mathbf{T}_{\text{bas}}\,[e_s] \\
\mathbf{T}_{0}\,[e] & \equiv & \mathbf{T}_{0}\,[e_t] \texttt{ = } \mathbf{T}_{0}\,[e_s] \\
\ldots & & \\
\mathbf{T}_{R-1}\,[e] & \equiv & \mathbf{T}_{R-1}\,[e_t] \texttt{ = } \mathbf{T}_{R-1}\,[e_s]
\end{array}$$

where:

e_t has distributed array type,
e_s is assignment convertible to the type of e_t, and
R is the rank of both arrays.

Figure 8.15: Translation of a distributed-array assignment.

8.12 Translating assignment expressions

When the expresssions involved are distributed arrays, the general scheme is illustrated in Figure 8.15. This translation applies to assignments where the right-hand-side of the assignment is *not* a constructive expression. Assignments involving constructive expressions are handled in the following sections.

SOURCE:

a `= new` T `[[`e_0`, ..., `e_{R-1}`]] on` p `;`

TRANSLATION:

s `= 1 ;`
b `= 0 ;`
$\mathit{DEFINE\_DIMENSION}(\mathbf{T}_{R-1}[a], e_{R-1}, s, b)$
`...`
$\mathit{DEFINE\_DIMENSION}(\mathbf{T}_0[a], e_0, s, b)$
$\mathbf{T}_{\mathrm{dat}}[a]$ `=` p`.member() ? new` T `[`s`] : null ;`
$\mathbf{T}_{\mathrm{bas}}[a]$ `= new ArrayBase(`p`, `b`) ;`

where:

T is a Java type,
R is the rank of the created array,
each e_r is either a range-valued or an integer-valued, simple expression
in the source program,
p is a simple expression in the source program,
the expression a is the assigned array variable in the source program,
s and b are names of new temporaries, and
the macro *DEFINE_DIMENSION* is defined in the text.

Figure 8.16: Translation of distributed array creation expression.

8.13 Translating distributed array creation

The pre-translator ensures that distributed array creation only appears on the right-hand-side of a top-level assignment, so we only need to consider that case. The scheme is illustrated in Figure 8.16. If the "`on` p" clause is omitted in the source program, the value of e'_{grp} in the translation can be taken to be *apg*.

The macro *DEFINE_DIMENSION* is defined as follows:

```
DEFINE_DIMENSION(a'_r, e_r, s, b) ≡
        a'_r = e_r.arrayDim(s) ;
        b += s * e_r.loExtension() ;
        s *= e_r.volume() ;
```

if the expression e_r is a range, or

```
DEFINE_DIMENSION(a'_r, e_r, s, b) ≡
        a'_r = new SeqArrayDim(e_r, s) ;
        s *= e_r ;
```

if the expression e_r is an integer. As each dimension is processed, the memory stride for the next dimension is computed by multiplying the variable s by the number of locally held range elements in the current dimension. The final value of s is the total number of locally held elements. The variable b is incremented to allow space for a lower ghost regions, below the base of the physical array, if this is demanded by the ranges involved.

The method `arrayDim()` on the `Range` class creates an instance of `ArrayDim`, with the memory stride specified in its argument. It is used in place of a call to the `ArrayDim` constructor because `arrayDim()` has the property that if the range is actually a collapsed range, the returned object will be an instance of the `SeqArrayDim` subclass. This allows a new array created with a collapsed range to be cast to an array with a sequential dimension, should it prove necessary at a later stage (see section 8.23).

SOURCE:

$$e \quad \equiv \quad a\ \texttt{[}e_0\texttt{,}\ \ \ldots\texttt{,}\ \ e_{R-1}\texttt{]}$$

TRANSLATION:

$$\mathbf{T}\,[e] \quad \equiv \quad \mathbf{T}_{\mathrm{dat}}\,[a] \ \ \texttt{[}\mathit{OFFSET}(a, e_0, \ldots, e_{R-1})\texttt{]}$$

where:

> The expression a is the subscripted array,
> each term e_r is either an integer, a distributed index name,
> or a shifted index expression, and
> the macro *OFFSET* is defined in the text.

Figure 8.17: Translation of array access expression.

8.14 Translating element access

We only need to consider the case where the array reference is a distributed array. The general scheme is illustrated in Figure 8.17. The macro *OFFSET* is defined as

$$\mathit{OFFSET}(a, e_0, \ldots, e_{R-1}) \equiv$$

$$\begin{aligned} \mathbf{T}_{\text{bas}}[a]\texttt{.base} \ &\texttt{+}\ \mathit{OFFSET\_DIM}(\mathbf{T}_0[a], e_0) \\ &\ \ \ldots \\ &\texttt{+}\ \mathit{OFFSET\_DIM}(\mathbf{T}_{R-1}[a], e_{R-1}) \end{aligned}$$

There are three cases for the macro *OFFSET_DIM* depending on whether the subscript argument is a distributed index, a shifted index, or an integer subscripts (in a sequential dimension).

If e_r is a distributed index i, then

$$\mathit{OFFSET\_DIM}(a'_r, e_r) \equiv \quad a'_r\texttt{.stride * } \mathit{sub}$$

where *sub* is the local subscript variable for this index (see sections 8.8 and 8.9). Otherwise if e_r is a shifted index $i \pm d$, then

$$\mathit{OFFSET\_DIM}(a'_r, e_r) \equiv \quad a'_r\texttt{.stride * (}\mathit{sub} \ \pm\ \mathit{shf\_stp} \texttt{ * } \mathbf{T}[d]\texttt{)}$$

where *sub* is the local subscript variable and *shf_stp* is the shift step for i (again, see sections 8.8 and 8.9). Otherwise if e_r is an integer expression (which implies that a'_r has type `SeqArrayDim`), then

$$\mathit{OFFSET\_DIM}(a'_r, e_r) \equiv \quad a'_r\texttt{.off\_bas + } a'_r\texttt{.off\_stp * } \mathbf{T}[e_r]$$

The fields `off_bas` and `off_stp` are initialized by the constructors for `SeqArray-Dim`. They do not exist in the superclass `ArrayDim`.

SOURCE:

v `=` a `[[`$subs_0$`, ...,` $subs_{R-1}$`]] ;`

TRANSLATION:

$PROCESS\_SUBSCRIPTS(v, 0, a, 0)$
$\mathbf{T}_{\mathrm{dat}}[v]$ `=` $\mathbf{T}_{\mathrm{dat}}[a]$ `;`
$\mathbf{T}_{\mathrm{bas}}[v]$ `=` $\mathbf{T}_{\mathrm{bas}}[a]$ `;`

where:

The expression v is the assigned array variable in the source program,
the simple expression a is the subscripted array in the source program,
each term $subs_s$ is either an integer, a triplet, or `<>`,
b is the name of a new temporary, and
the macro *PROCESS_SUBSCRIPTS* is defined in the text.

Figure 8.18: Translation of array section with no scalar subscripts.

8.15 Translating array sections

The rules for translating array sections are more complicated than any other part of the basic translation scheme.

We will break it down into three cases: the case where there are no scalar subscripts—integer or distributed index; the case where integer scalar subscripts may appear in *sequential* dimensions; and the general case where scalar subscripts may appear in distributed dimensions. The scheme for translating the first case is illustrated in Figure 8.19.

The macro *PROCESS_SUBSCRIPTS* will be defined here in a tail-recursive way. The intention is that it should be expanded to a compile-time loop over the subscripts.

Let R be the rank of the subscripted array. If $s = R$, then the macro *PROCESS_SUBSCRIPTS* (v, r, a, s) is empty. Otherwise, if $subs_s$ is the degenerate triplet, `:`, then

```
PROCESS_SUBSCRIPTS (v, r, a, s) ≡
          T_r [v] = T_s [a] ;
          PROCESS_SUBSCRIPTS (v, r + 1, a, s + 1)
```

Otherwise, if $subs_s$ is the triplet, e_{lo}`:`e_{hi}`:`e_{stp}, and the sth dimension of a is distributed, then

```
PROCESS_SUBSCRIPTS (v, r, a, s) ≡
    T_r [v] = a'_s.range.subrng(e'_lo, e'_hi, e'_stp).arrayDim(a'_s.stride) ;
    PROCESS_SUBSCRIPTS (v, r + 1, a, s + 1)
```

where $a'_s = \mathbf{T}_s\,[a]$, $e'_{\mathrm{lo}} = \mathbf{T}\,[e_{\mathrm{lo}}]$, $e'_{\mathrm{hi}} = \mathbf{T}\,[e_{\mathrm{hi}}]$, and $e'_{\mathrm{stp}} = \mathbf{T}\,[e_{\mathrm{stp}}]$. Otherwise, if $subs_s$ is the triplet, e_{lo}`:`e_{hi}`:`e_{stp}, and the sth dimension of a is sequential, then

```
PROCESS_SUBSCRIPTS (v, r, a, s) ≡
  T_r [v] = new SeqArrayDim(a'_s.range.subrng(e'_lo, e'_hi, e'_stp), a'_s.stride) ;
  PROCESS_SUBSCRIPTS (v, r + 1, a, s + 1)
```

with definitions as above. Two similar cases using the two-argument form of `subrng()` take care of triplets of the form e_{lo}`:`e_{hi}. Otherwise, if $subs_s$ is the splitting subscript, `<>`, then

```
PROCESS_SUBSCRIPTS (v, r, a, s) ≡
        x = a'_s.range ;
        u = a'_s.stride ;
        z = x.shell() ;
        T_{r+1} [v] = new SeqArrayDim(z, u) ;
        T_r [v] = new ArrayDim(x.dim(), u * z.volume()) ;
        PROCESS_SUBSCRIPTS (v, r + 2, a, s + 1)
```

where x, u, and z are the names of new temporaries.

SOURCE:

v `=` a `[[`$subs_0$`, ...,` $subs_{R-1}$`]] ;`

TRANSLATION:

b `=` $\mathbf{T}_{\mathrm{bas}}[a]$`.base ;`
$PROCESS\_SUBSCRIPTS(v, 0, a, 0)$
$\mathbf{T}_{\mathrm{dat}}[v]$ `=` $\mathbf{T}_{\mathrm{dat}}[a]$
$\mathbf{T}_{\mathrm{bas}}[v]$ `= new ArrayBase(`$\mathbf{T}_{\mathrm{bas}}[a]$`.group,` b`) ;`

where:

The expression v is the assigned array variable in the source program,
the simple expression a is the subscripted array in the source program,
each term $subs_s$ is either an integer, a triplet, or `<>`,
b is the name of a new temporary, and
the macro *PROCESS_SUBSCRIPTS* is defined in the text.

Figure 8.19: Translation of array section without any scalar subscripts in *distributed* dimensions.

8.15.1 Integer subscripts in sequential dimensions

To handle the case where integer subscripts may appear in sequential dimensions (Figure 8.19) we add one new case for the definition of the macro *PROCESS_SUBSCRIPTS*.

If $subs_s$ is an integer expression, and the sth dimension of a is sequential, then

$$\begin{aligned} &\mathit{PROCESS\_SUBSCRIPTS}\,(v, r, a, s) \equiv \\ &\quad b \ \texttt{+=} \ \mathit{OFFSET\_DIM}\,(\mathbf{T}_s\,[a]\,, subs_s) \ \ ; \\ &\quad \mathit{PROCESS\_SUBSCRIPTS}\,(v, r, a, s+1) \end{aligned}$$

where the macro *OFFSET_DIM* is defined in section 8.14.

[Actually distributed index subscripts in sequential dimensions could also go here, although it is an odd case.]

SOURCE:

v `=` a `[[`$subs_0$`, ...,` $subs_{R-1}$`]] ;`

TRANSLATION:

b `=` $\mathbf{T}_{\mathrm{bas}}[a]$`.base ;`
p `= (Group)` $\mathbf{T}_{\mathrm{bas}}[a]$`.group.clone() ;`
$PROCESS\_SUBSCRIPTS(v, 0, a, 0)$
$\mathbf{T}_{\mathrm{dat}}[v]$ `=` p`.member() ?` $\mathbf{T}_{\mathrm{dat}}[a]$ `: null ;`
$\mathbf{T}_{\mathrm{bas}}[v]$ `= new ArrayBase(`p`,` b`) ;`

where:

The expression v is the assigned array variable in the source program,
the simple expression a is the subscripted array in the source program,
each term $subs_s$ is either an integer, a triplet, or `<>`,
b and p are the names of new temporaries, and
the macro *PROCESS_SUBSCRIPTS* is defined in the text.

Figure 8.20: Translation of array section allowing scalar subscripts in distributed dimensions.

8.15.2 Scalar subscripts in distributed dimensions

The scheme for translating array sections when scalar subscripts appear in some distributed dimension is illustrated in Figure 8.20.

We add two new cases for the definition of the macro *PROCESS_SUBSCRIPTS*. If $subs_s$ is the integer expression n, and the sth dimension of a is distributed, then

```
$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv$
    $x$ = $a'_s$.range ;
    $l$ = $x$.location($n'$) ;
    $b$ += $l$.sub * $a'_s$.stride ;
    $p$.restrict($x$.dim(), $l$.crd) ;
    $PROCESS\_SUBSCRIPTS(v, r, a, s + 1)$
```

where x and l are the names of new temporaries, $a'_s = \mathbf{T}_s\,[a]$, and $n' = \mathbf{T}\,[n]$. Otherwise, if $subs_s$ is a distributed index i or a shifted index $i \pm d$, then

```
$PROCESS\_SUBSCRIPTS(v, r, a, s) \equiv$
    $b$ += $OFFSET\_DIM(\mathbf{T}_s\,[a]\,, subs_s)$ ;
    $p$.restrict($x$.dim()) ;
    $PROCESS\_SUBSCRIPTS(v, r, a, s + 1)$
```

where in this case x is the range associated with i and the macro *OFFSET_DIM* is defined in section 8.14.

SOURCE:

$$e \equiv e_{\text{obj}} \,.\, a$$

TRANSLATION:

$$\begin{aligned}
\mathbf{T}_{\text{dat}}\,[e] &\equiv e_{\text{obj}} \,.\, a'_{\text{dat}} \\
\mathbf{T}_{\text{bas}}\,[e] &\equiv e_{\text{obj}} \,.\, a'_{\text{bas}} \\
\mathbf{T}_{0}\,[e] &\equiv e_{\text{obj}} \,.\, a'_{0} \\
\ldots & \\
\mathbf{T}_{R-1}\,[e] &\equiv e_{\text{obj}} \,.\, a'_{R-1}
\end{aligned}$$

where:

the simple expression e_{obj} has class type,
R is the rank of the distributed-array-valued field,
the identifier a is the name of the field in the source program, and
a'_{dat}, a'_{bas}, and $a'_0, \ldots, a'_{R-1}$ are corresponding names of fields
in the translated program.

Figure 8.21: Translation of a distributed-array-valued field access.

8.16 Translating field accesses

We only need to consider the case where the field is a distributed array. The general scheme is illustrated in Figure 8.21.

The names a'_{dat}, a'_{bas}, and $a'_0, \ldots, a'_{R-1}$ are the names introduced by the translator when translating the corresponding field in the source program (see section 8.5).

SOURCE:

$$e \quad \equiv \quad e_{\text{obj}}\texttt{.}f\texttt{(}e_0\texttt{,} \quad \ldots\texttt{,} \quad e_{N-1}\texttt{)}$$

TRANSLATION:

$$\mathbf{T}\,[e] \quad \equiv \quad \mathbf{T}\left[e_{\text{obj}}\right]\texttt{.}f\texttt{(}\mathit{TRANS\_ARGS}(e_0, \ldots, e_{N-1})\texttt{,} \;\; \mathit{apg}\texttt{)}$$

where:

The expression e_{obj} has class type,
the identifier f is the name of the method,
each term e_r is an actual argument in the original program,
the macro *TRANS_ARGS* is defined in the text.

Figure 8.22: Translation of method invocation expression, where return value is not distributed-array-valued.

8.17 Translating method invocations

There are two cases to consider. The case where the result of the method is not a distributed array, and the case where the result is a distributed array. First we consider the case where the result is not a distributed array. The general scheme is illustrated in Figure 8.22. This scheme is modified in trivial ways if the method is static, or is applied to the current object.

The macro *TRANS_ARGS* evaluates to a list of translated expressions. We will define it in terms of the simpler macro *TRANS_ARG* which operates on a single argument:

$$\begin{aligned}&\mathit{TRANS\_ARGS}(e_0, \ldots, e_{N-1}) \equiv \\ &\qquad \mathit{TRANS\_ARG}(e_0), \quad \ldots, \quad \mathit{TRANS\_ARG}(e_{N-1})\end{aligned}$$

Now, if e is not a distributed array, then

$$\mathit{TRANS\_ARG}(e) \equiv \quad \mathbf{T}\,[e]$$

Otherwise, if e is a distributed array expression of rank R, then

$$\begin{aligned}&\mathit{TRANS\_ARG}(e) \equiv \\ &\qquad \mathbf{T}_{\mathrm{dat}}\,[e]\,, \ \mathbf{T}_{\mathrm{bas}}\,[e]\,, \ \mathbf{T}_0\,[e]\,, \ \ldots, \ \mathbf{T}_{R-1}\,[e]\end{aligned}$$

In other words, each distributed array argument is split into $2 + R$ arguments.

SOURCE:

v = e_{obj}.id(e_0, ..., e_{N-1}) ;

TRANSLATION:

d = `new DAD(`R`)` ;
$\mathbf{T}_{\text{dat}}[v]$ = $\mathbf{T}\left[e_{\text{obj}}\right]$.*TRANS_ID*($f$)($d$, *TRANS_ARGS*($e_0, \ldots, e_{N-1}$), *apg*) ;
$\mathbf{T}_{\text{bas}}[v]$ = d.`base`
$\mathbf{T}_0[v]$ = *TRANS_DIM*($d, 0$)
...
$\mathbf{T}_{R-1}[v]$ = *TRANS_DIM*($d, R-1$)

where:

The expression e_{obj} has class type or is a class,
the identifier f is the name of the method,
each term e_r is an actual argument in the source program,
d is a new temporary,
R is the rank of the result, and
the macros *TRANS_ID*, *TRANS_ARGS*, and *TRANS_DIM*
are defined in the text.

Figure 8.23: Translation of array-valued method invocation expression.

8.17.1 Methods that return distributed arrays

The scheme for translating a distributed-array-valued method invocation is illustrated in Figure 8.23.

The macro *TRANS_ID* encodes the dimension signature of the returned array and is defined in section 8.5. The macro *TRANS_ARGS* was defined earlier in this section.

The macro *TRANS_DIM* is defined as

$$\mathit{TRANS\_DIM}(d, r) \equiv \quad d\texttt{.dimensions [}r\texttt{]}$$

if the rth dimension of the result is distributed, or

$$\mathit{TRANS\_DIM}(d, r) \equiv \quad \texttt{(SeqArrayDim) } d\texttt{.dimensions [}r\texttt{]}$$

if the rth dimension is sequential.

SOURCE:

```
return $a$ ;
```

TRANSLATION:

```
$d$.base  = $\mathbf{T}_{\mathrm{bas}}[a]$ ;
$d$.dimensions [0] = $\mathbf{T}_0[a]$ ;
...
$d$.dimensions [$R-1$] = $\mathbf{T}_{R-1}[a]$ ;

return $\mathbf{T}_{\mathrm{dat}}[a]$ ;
```

where:

a is the array-valued result expression,
R is its rank, and
d is the DAD passed as first argument of the translated method.

Figure 8.24: Translation of return statement in array-valued method.

8.17.2 Translation of `return` statement

The scheme for translating a `return` statement in the definition of an array-valued method is illustrated in Figure 8.24.

SOURCE:

$$e \quad \equiv \quad \texttt{new}\ T\texttt{(}e_0\texttt{,}\ \ldots\texttt{,}\ e_{N-1}\texttt{)}$$

TRANSLATION:

$$\mathbf{T}\,[e] \quad \equiv \quad \texttt{new}\ T\texttt{(}\mathit{TRANS\_ARGS}(e_0, \ldots, e_{N-1})\texttt{)}$$

where:

T is a Java class type,
each term e_r is an actual argument in the original program,
the macro *TRANS_ARGS* is defined in the text.

Figure 8.25: Translation of class instance creation expression.

8.18 Translating constructor invocations

The rules for translating constructor invocations follow directly from the rules for method invocations given in section 8.17.

Figure 8.22 illustrates the translation for a class instance creation expression. Explicit constructor invocations (specifying `this` or `super`) do not introduce any new features.

SOURCE:

v `=` a `/` e_{loc} `;`

TRANSLATION:

p `=` $\mathit{RESTRICT\_GROUP}(\mathbf{T}_{\text{bas}}[a]$`.group,`$e_{\text{loc}})$ `;`
$\mathbf{T}_{\text{dat}}[v]$ `=` p`.member() ?` $\mathbf{T}_{\text{dat}}[a]$ `: null ;`
$\mathbf{T}_{\text{bas}}[v]$ `= new ArrayBase(`p`,` $\mathbf{T}_{\text{bas}}[a]$`.base) ;`

where:

The expression v is the assigned array variable in the source program,
the simple expression a is the restricted array in the source program,
the expression e_{loc} is either a distributed index, a shifted index,
or a range element,
p is a new temporary, and
the macro *RESTRICT_GROUP* is defined in the text.

Figure 8.26: Translation of array restriction operation.

8.19 Translating array restriction

The scheme is illustrated in Figure 8.26.

If e_{loc} is a range element of the form e_{rng} `[`n`]`, the macro *RESTRICT_GROUP* is defined as

```
RESTRICT_GROUP(p, e_loc) ≡
    x = e'_rng ;
    l = x.location(n');
    ((Group) p.clone()).restrict(x.dim(), l.crd) ;
```

where x and l are the names of new temporaries, $e'_{\mathrm{rng}} = \mathbf{T}^{V}_{s}\left[e_{\mathrm{rng}}\right]$, and $n' = \mathbf{T}\left[n\right]$. Otherwise, if e_{loc} is a distributed index i or a shifted index $i \pm d$, it is defined as

```
RESTRICT_GROUP(p, e_loc) ≡
    ((Group) p.clone()).restrict(x.dim()) ;
```

where in this case x is the range associated with i.

SOURCE:

$$e \;\equiv\; p \;/\; e_{\text{loc}}$$

TRANSLATION:

$$\mathbf{T}_{\text{dat}}\left[v\right] \;\equiv\; \mathit{RESTRICT\_GROUP}(\mathbf{T}\left[p\right], e_{\text{loc}})$$

where:

The expression p is the group to be restricted,
the expression e_{loc} is either a distributed index, a shifted index,
or a range element, and
the macro *RESTRICT_GROUP* is defined in the text.

Figure 8.27: Translation of group restriction operation.

8.20 Translating group restriction

The scheme is illustrated in Figure 8.27. The macro *RESTRICT_GROUP* is defined in section 8.19.

SOURCE:

$$e \quad \equiv \quad x \ \texttt{[}e_{\mathrm{lo}} \ \texttt{:} \ e_{\mathrm{hi}}\texttt{]}$$

TRANSLATION:

$$\mathbf{T}\,[e] \quad \equiv \quad \mathbf{T}\,[x]\,\texttt{.subrng(}\mathbf{T}\,[e_{\mathrm{lo}}]\,\texttt{,}\ \ \mathbf{T}\,[e_{\mathrm{hi}}]\,\texttt{)}$$

SOURCE:

$$e \quad \equiv \quad x \ \texttt{[}e_{\mathrm{lo}} \ \texttt{:} \ e_{\mathrm{hi}} \ \texttt{:} \ e_{\mathrm{stp}}\texttt{]}$$

TRANSLATION:

$$\mathbf{T}\,[e] \quad \equiv \quad \mathbf{T}\,[x]\,\texttt{.subrng(}\mathbf{T}\,[e_{\mathrm{lo}}]\,\texttt{,}\ \ \mathbf{T}\,[e_{\mathrm{hi}}]\,\texttt{,}\ \ \mathbf{T}\left[e_{\mathrm{stp}}\right]\texttt{)}$$

where:

The expression x is the parent range, and
e_{lo}, e_{hi} and e_{stp} are integer-valued expressions in the source program.

Figure 8.28: Translation of subrange expressions.

8.21 Translating subrange expressions

The schemes are illustrated in Figure 8.28.

SOURCE:

$$e \quad \equiv \quad e_{\text{bool}} \ \texttt{?} \ a \ \texttt{:} \ b$$

TRANSLATION:

$$\begin{array}{rcl}
\mathbf{T}_{\text{dat}}\,[e] & \equiv & e_{\text{bool}} \ \texttt{?} \ \mathbf{T}_{\text{dat}}\,[a] \ \ \texttt{:} \ \mathbf{T}_{\text{dat}}\,[b] \\
\mathbf{T}_{\text{bas}}\,[e] & \equiv & e_{\text{bool}} \ \texttt{?} \ \mathbf{T}_{\text{bas}}\,[a] \ \ \texttt{:} \ \mathbf{T}_{\text{bas}}\,[b] \\
\mathbf{T}_{0}\,[e] & \equiv & e_{\text{bool}} \ \texttt{?} \ \mathbf{T}_{0}\,[a] \ \ \texttt{:} \ \mathbf{T}_{0}\,[b] \\
\ldots & & \\
\mathbf{T}_{R-1}\,[e] & \equiv & e_{\text{bool}} \ \texttt{?} \ \mathbf{T}_{R-1}\,[a] \ \ \texttt{:} \ \mathbf{T}_{R-1}\,[b]
\end{array}$$

where:

The simple expression e_{bool} has boolean type, and
a and b are assignment convertible to a distributed array type, and
the rank of this type is R.

Figure 8.29: Translation of conditional operator selecting distributed arrays.

8.22 Translating the conditional operator

The general scheme is illustrated in Figure 8.15. The definition of the result type, and the associated conditions for legality of selecting between a and b, follow from this translation together with the normal Java rules applied to the parts of the arrays.

SOURCE:

$$e \equiv a\texttt{.dat()}$$

TRANSLATION:

$$\mathbf{T}\left[e\right] \equiv \mathbf{T}_{\mathrm{dat}}\left[a\right]$$

SOURCE:

$$e \equiv a\texttt{.bas()}$$

TRANSLATION:

$$\mathbf{T}\left[e\right] \equiv \mathbf{T}_{\mathrm{bas}}\left[a\right]\texttt{.base}$$

SOURCE:

$$e \equiv a\texttt{.grp()}$$

TRANSLATION:

$$\mathbf{T}\left[e\right] \equiv \mathbf{T}_{\mathrm{bas}}\left[a\right]\texttt{.group}$$

where:

The expression a has distributed array type.

Figure 8.30: Translation of distributed array inquiries.

SOURCE:

$$e \quad \equiv \quad a\texttt{.rng(}r\texttt{)}$$

TRANSLATION:

$$\mathbf{T}\left[e\right] \quad \equiv \quad \mathbf{T}_r\left[a\right]\texttt{.range}$$

SOURCE:

$$e \quad \equiv \quad a\texttt{.str(}r\texttt{)}$$

TRANSLATION:

$$\mathbf{T}\left[e\right] \quad \equiv \quad \mathbf{T}_r\left[a\right]\texttt{.stride}$$

where:

The expression a has distributed array type, and
the term r is a compile-time constant integer expression
in the range $0 \leq r < R$, where R is the rank of a.

Figure 8.31: Translation of distributed array inquiries, continued.

SOURCE:

$$e \equiv \texttt{(}T\ \texttt{[[}attr_0\texttt{, ..., }attr_{R-1}\texttt{]])}\ a$$

TRANSLATION:

$$\begin{aligned}
\mathbf{T}_{\mathrm{dat}}\,[e] &\equiv \texttt{(}T\ \texttt{[])}\ \mathbf{T}_{\mathrm{dat}}\,[a] \\
\mathbf{T}_{\mathrm{bas}}\,[e] &\equiv \mathbf{T}_{\mathrm{bas}}\,[a] \\
\mathbf{T}_0\,[e] &\equiv \mathit{CAST\_DIMENSION}\,(attr_0, \mathbf{T}_0\,[a]) \\
\ldots & \\
\mathbf{T}_{R-1}\,[e] &\equiv \mathit{CAST\_DIMENSION}\,(attr_{R-1}, \mathbf{T}_{R-1}\,[a])
\end{aligned}$$

where:

T is a Java type,
each term $attr_r$ is a single hyphen, `-`, or a single asterisk, `*`,
a is an array expression in the source program, of rank R, and
the macro *CAST_DIMENSION* is defined in the text.

Figure 8.32: Translation of cast of array expression.

8.23 Translating casts

The scheme for translating a cast of a distributed-array valued expression is illustrated in Figure 8.32. The macro *CAST_DIMENSION* is defined as follows:

$$\mathit{CAST\_DIMENSION}(\mathit{attr}_r, a'_r) \equiv \texttt{(ArrayDim)}\ a'_r$$

if the term attr_r is empty, or

$$\mathit{CAST\_DIMENSION}(\mathit{attr}_r, a'_r) \equiv \texttt{(SeqArrayDim)}\ a'_r$$

if the term attr_r is an asterisk, `*`.

SOURCE:

$$e \quad \equiv \quad a \ \texttt{instanceof} \ T \ \texttt{[[}attr_0\texttt{,} \ \ldots\texttt{,} \ attr_{R-1}\texttt{]]}$$

TRANSLATION:

$$\begin{aligned}
\mathbf{T}\,[e] \quad \equiv \quad & \texttt{(}\mathbf{T}_{\mathrm{dat}}\,[a] \ \texttt{instanceof} \ T \ \texttt{[])} \ \texttt{\&\&} \\
& \texttt{(}\mathbf{T}_0\,[a] \ \texttt{instanceof} \ \mathit{DIMENSION\_TYPE}\,(attr_0)\texttt{)} \ \texttt{\&\&} \\
& \ldots \\
& \texttt{(}\mathbf{T}_{R-1}\,[a] \ \texttt{instanceof} \ \mathit{DIMENSION\_TYPE}\,(attr_{R-1})\texttt{)}
\end{aligned}$$

where:

a is an array expression in the source program, rank R,
T is a Java type,
each term $attr_r$ is a single hyphen, `-`, or a single asterisk, `*`,
the macro *DIMENSION_TYPE* is defined in the text.

Figure 8.33: Translation `instanceof` applied to array expression.

8.24 Translating `instanceof`

The scheme for translating an `instanceof` test applied to a distributed-array valued expression is illustrated in Figure 8.33. The macro *DIMENSION_TYPE* is in section 8.5.

SOURCE:

$$e \equiv a \texttt{ == } b$$

TRANSLATION:

$$\begin{aligned}\mathbf{T}[e] \equiv\ & \texttt{(}\mathbf{T}_{\mathrm{dat}}[a] \texttt{ == } \mathbf{T}_{\mathrm{dat}}[b]\texttt{) \&\&}\\ & \texttt{(}\mathbf{T}_{\mathrm{bas}}[a] \texttt{ == } \mathbf{T}_{\mathrm{bas}}[b]\texttt{) \&\&}\\ & \texttt{(}\mathbf{T}_{0}[a] \texttt{ == } \mathbf{T}_{0}[b]\texttt{) \&\&}\\ & \ldots\\ & \texttt{(}\mathbf{T}_{R-1}[a] \texttt{ == } \mathbf{T}_{R-1}[b]\texttt{)}\end{aligned}$$

SOURCE:

$$a \texttt{ != } b$$

TRANSLATION:

$$\begin{aligned}\mathbf{T}[e] \equiv\ & \texttt{(}\mathbf{T}_{\mathrm{dat}}[a] \texttt{ != } \mathbf{T}_{\mathrm{dat}}[b]\texttt{) ||}\\ & \texttt{(}\mathbf{T}_{\mathrm{bas}}[a] \texttt{ != } \mathbf{T}_{\mathrm{bas}}[b]\texttt{) ||}\\ & \texttt{(}\mathbf{T}_{0}[a] \texttt{ != } \mathbf{T}_{0}[b]\texttt{) ||}\\ & \ldots\\ & \texttt{(}\mathbf{T}_{R-1}[a] \texttt{ != } \mathbf{T}_{R-1}[b]\texttt{)}\end{aligned}$$

where:

Expressions a and b are assignment compatible with a distributed array type, and this type has rank, R.

Figure 8.34: Translation of reference equality tests, applied to array expressions.

8.25 Translating reference equality

The scheme for translating the reference equality operators `==` and `!=`, applied to distributed arrays is illustrated in Figure 8.34.

Because there is no single Java reference to a distributed array (or, equivalently, because a distributed array is not considered to be an object) there is no obvious, a priori definition for reference equality between distributed arrays. The translations given here can be read as definitions.

One consequence of these definitions, together with the translation for array sections given in section 8.15, is that an array section in which every subscript is the default triplet `:` is considered to be identical (in the sense of reference equality) to its parent array.

[Define the value of `null` *here?? Have to define it somewhere.]*

8.26 Optimization

8.26.1 On subscripting

8.26.2 Analysis of `overall` constructs

If an array is subscripted with a distributed index, as in:

```
at(i = x [n]) {
  ... a[e0, ..., er−1, i, er+1, ...]  ...
}
```

or

```
overall(i = x for l : u : s) {
  ... a[e0, ..., er−1, i, er+1, ...]  ...
}
```

then, according to section 5.3 the expression *a* should be an side-effect free invariant. In fact, according to the policies defined section 8.2.1 it should be simply named, final variable, declared outside the `overall` construct.

Consider the nested `overall` and loop constructs from Figure 3.13, which we reproduce here:

```
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 ;
  }
```

A correct but naive translation of this fragment, based on the translation scheme in Figure 7.14, is given in Figure 8.35.

Actually this translation isn't quite general because it exploits some specific assumptions about the sequential dimensions of `a` and `b`. But in this section we will concentrate on subscripting in *distributed* dimensions. The main problem we want to address here is the complexity of the associated terms in the subscript expressions. Specifically we want to replace the code of Figure 8.35 with something like the code in Figure 8.36, where the subscript expressions have been greatly simplified by application of *strength-reduction optimizations*.

The specific goal in this version was to eliminate complicated expressions involving multiplication from expressions in inner loops. This was achieved by introducing the *induction variables*:

```
vai_  ≡  a.bas() + (bi.sub_bas + bi.sub_stp * lx) * a.str(0)
vci_  ≡  c.bas() + (bi.sub_bas + bi.sub_stp * lx) * c.str(0)
vb_j  ≡  b.bas() + (bj.sub_bas + bj.sub_stp * ly) * b.str(1)
vcij  ≡  c.bas() + (bi.sub_bas + bi.sub_stp * ly) * c.str(0) +
                   (bj.sub_bas + bj.sub_stp * ly) * c.str(1)
```

```
Block bi = x.localBlock() ;

for (int lx = 0 ; lx < bi.count ; lx++) {
  Block bj = y.localBlock() ;

  for (int ly = 0 ; ly < bj.count ; ly++) {

    float sum = 0 ;
    for(int k = 0 ; k < N ; k++)
      sum += a.dat() [a.bas() +
                      (bi.sub_bas + bi.sub_stp * lx) * a.str(0) +
                       k * a.str(1)] *
             b.dat() [b.bas() +
                       k * b.str(0) +
                      (bj.sub_bas + bj.sub_stp * ly) * b.str(1)] ;

    c.dat() [c.bas() +
            (bi.sub_bas + bi.sub_stp * lx) * c.str(0) +
            (bj.sub_bas + bj.sub_stp * ly) * c.str(1)] = sum ;
  }
}
```

Figure 8.35: Naive translation of `overall` from Figure 3.13

which can be computed efficiently by incrementing at suitable points with the *induction increments*:

```
sia0   ≡   bi.sub_stp * a.str(0)
sic0   ≡   bi.sub_stp * c.str(0)
sjb1   ≡   bj.sub_stp * b.str(1)
sjc1   ≡   bj.sub_stp * c.str(1)
```

In Figure 8.36 we deliberately refrained from reducing the multiplications `k * a.str(1)` and `k * b.str(0)` associated with subscripting sequential dimensions. In general such an optimization depends on analysis of the sequential `for` loop involved; here we are specifically interested in `overall` constructs and distributed index subscripts.

In rest of this section we will to describe a compile-time algorithm for identifying the induction variables and induction increments need to optimize an arbitrary program in this manner.

We define an *indexing pattern* as an mapping of the members of a set of distributed indices to a subset of the dimensions of a particular array. A *dimension assignment*[4] an association of an individual distributed index with an individual dimension.

[4] *Not* in the sense of an assignment statement or expression!

```
Block bi = x.localBlock() ;

int vai_ = a.bas() + bi.sub_bas * a.str(0) ;
int vci_ = c.bas() + bi.sub_bas * c.str(0) ;

final int sia0 = bi.sub_stp * a.str(0) ;
final int sic0 = bi.sub_stp * c.str(0) ;

for (int lx = 0 ; lx < bi.count ; lx++) {
  Block bj = y.localBlock() ;

  int vb_j = b.bas() + bj.sub_bas * b.str(1) ;
  int vcij = vci_    + bj.sub_bas * c.str(1) ;

  final int sjb1 = bj.sub_stp * b.str(1) ;
  final int sjc1 = bj.sub_stp * c.str(1) ;

  for (int ly = 0 ; ly < bj.count ; ly++) {

    float sum = 0 ;
    for(int k = 0 ; k < N ; k++)
      sum += a.dat() [vai_ + k * a.str(1)] *
             b.dat() [vb_j + k * b.str(0)] ;

    c.dat() [vcij] = sum ;

    vb_j += sjb1 ;
    vcij += sjc1 ;
  }

  vai_ += sia0
  vci_ += sic0
}
```

Figure 8.36: Translation of `overall` from Figure 3.13, after applying strength reduction to distributed index subscript expressions

A dimension assignment of a particular index can be represented by a tuple (i, a, r) where i is an index, a is an array name and r is a dimension label in the range $0, \ldots, R-1$, where R is the rank of a. An indexing pattern can be conveniently represented by a list of dimension assignments.

Now we assume the compile-time description of an index i is annotated with a *partial indexing set* and a *dimension assignment set*. As the names suggest, the former is a set of indexing patterns, and the latter is a set of dimension assignments.

These sets are initialized to the empty set for every index in the program, then the program text is traversed to look for subscripting expressions—distributed array accesses or array section expressions.

When a subscripting expression is encountered the following procedure is executed. We assume the array name is a, and p is a variable in which the indexing pattern for this expression will be accumulated:

1. Set p to `null`.
2. Find the distributed index with *largest scope* appearing in any subscript of the expression.
3. Suppose this index is i and it appears as a subscript in dimension r. Look for the tuple (i, a, r) in the dimension assignment set of i. If it doesn't appear, add it to the set.
4. Set p to (i, a, r)++p, ie, a list with head (i, a, r) and tail p.
5. Look for the pattern p in partial indexing pattern set of i. If it doesn't appear, add it to the set.
6. Find the distributed index with *next* largest scope appearing in any subscript. Repeat steps 3 to 6 until all indexes appearing in the subscripting expression have been exhausted.
7. Annotate the compile-time description of the subscripting expression with p.

As a specific example, in the matrix multiplication code from the start of this section, the partial indexing patterns associated with `i` will be:

$$[(\mathtt{i}, \mathtt{a}, 0)]$$
$$[(\mathtt{i}, \mathtt{c}, 0)]$$

and the partial indexing patterns associated with `j` will be:

$$[(\mathtt{j}, \mathtt{b}, 1)]$$
$$[(\mathtt{i}, \mathtt{c}, 0), (\mathtt{j}, \mathtt{c}, 1)]$$

These might equivalently be written as:

$$\mathtt{a}[\mathtt{i}, -]$$
$$\mathtt{c}[\mathtt{i}, -]$$

and

$$\texttt{b}[-, \texttt{j}]$$
$$\texttt{c}[\texttt{i}, \texttt{j}]$$

respectively.

After the whole program has been processed in this way, the translation of the `overall` construct associated with a particular index involves defining one induction variable for every member of the partial indexing set of that index, and one induction increment for every member of the dimension assigment set. The details are given in the next section.

SOURCE:

```
overall ($i$ = $e_{\mathrm{rng}}$ for $e_{\mathrm{lo}}$ : $e_{\mathrm{hi}}$ : $e_{\mathrm{stp}}$) $S$
```

TRANSLATION:

```
Block $b$ = $e_{\mathrm{rng}}$.localBlock($\mathbf{T}[e_{\mathrm{lo}}]$, $\mathbf{T}[e_{\mathrm{hi}}]$, $\mathbf{T}[e_{\mathrm{stp}}]$) ;

Group $p$ = ((Group) $apg$.clone()).restrict($e_{\mathrm{rng}}$.dim()) ;

$DEFINE\_INDUCTION\_VARIABLES\,(i, b)$
$DEFINE\_INDUCTION\_INCREMENTS\,(i, b)$

for (int $l$ = 0 ; $l$ < $b$.count ; $l$++) {

    $\mathbf{T}[S\,|p]$

    $INCREMENT\_INDUCTION\_VARIABLES\,(i)$
}
```

where:

i is an index name in the source program,
e_{rng}, e_{lo}, e_{hi}, and e_{stp} are expressions in the source,
S is a statement in the source program,
b, p and l are names of new variables, and
the macros *DEFINE_INDUCTION_VARIABLES*,
DEFINE_INDUCTION_INCREMENTS, and
INCREMENT_INDUCTION_VARIABLES are defined in the text.

Figure 8.37: OptimizedTranslation of `overall` construct.

8.26.3 Translating `overall` constructs

A translation for the `overall` construct is given in Figure 8.37. This assumes the existence of several "translation functions". The function on expressions, $\mathbf{T}_{\mathrm{val}}[e]$, returns the result of translating an expression e. We may expect that in general such translation involves the execution of a series of statements as well as the evaluation of a final expression[5]. The sequence of statements that needs to be executed before $\mathbf{T}_{\mathrm{val}}[e]$ is evaluated is given by the function $\mathbf{T}_{\mathrm{pre}}[e]$.

The variants $\mathbf{T}^{V}_{\mathrm{val}}$, $\mathbf{T}^{V}_{\mathrm{pre}}$ defined a translation of e that always returns a simple name for $\mathbf{T}^{V}_{\mathrm{val}}[e]$, by having $\mathbf{T}^{V}_{\mathrm{pre}}[e]$ define an extra temporary variable for the result if necessary.

The macro *DEFINE_INDUCTION_VARIABLES* is defined as follows:

```
DEFINE_INDUCTION_VARIABLES (i, b) ≡
            v_i = b.glb_bas ;
            DEFINE_INDUCTION_VARIABLE (p_0, b)
            ...
            DEFINE_INDUCTION_VARIABLE (p_{P-1}, b)
```

where v_i is the name of a new temporary variable, $\{p_0, \ldots, p_{P-1}\}$ is the partial indexing set attached to i, and

```
DEFINE_INDUCTION_VARIABLE (p, b) ≡
            v_p = base + b.sub_bas * a.str(r) ;
```

Here v_p is the name of a new temporary, and the index/array/dimension-label tuple (i, a, r) is the dimension assignment in the *head* of the partial indexing pattern, p. If the tail, p', of p is empty, then

$$base = a.\texttt{bas()}$$

Otherwise, if p' is non-empty,

$$base = v_{p'}$$

Here $v_{p'}$ is the name of an induction variable associated with p'. By construction of the partial indexing patterns, $v_{p'}$ will have been defined earlier in the translated program.

The macro *DEFINE_INDUCTION_INCREMENTS* is defined by:

```
DEFINE_INDUCTION_INCREMENT (i, b) ≡
            s_i = b.glb_stp ;
            DEFINE_INDUCTION_INCREMENT (d_0, b)
            ...
            DEFINE_INDUCTION_INCREMENT (d_{D-1}, b)
```

where s_i is the name of a new temporary variable, $\{d_0, \ldots, d_{D-1}\}$ is the set of dimension assignments attached to i, and

```
DEFINE_INDUCTION_INCREMENT (d, b) ≡
            s_d = b.sub_stp * a.str(r) ;
```

[5]These statements may, for example, assign values to temporary variables

Here s_d is the name of a new temporary variable, and (i, a, r) is the index/array/dimension-label tuple in the dimension assignment, d.

The macro *INCREMENT_INDUCTION_VARIABLES* is defined as follows:

$$
\begin{aligned}
\mathit{INCREMENT\_INDUCTION\_VARIABLES}\,(i) \equiv\ & \\
& v_i \texttt{ += } s_i \texttt{ ;} \\
& \mathit{INCREMENT\_INDUCTION\_VARIABLE}\,(p_0) \\
& \ldots \\
& \mathit{INCREMENT\_INDUCTION\_VARIABLE}\,(p_{P-1})
\end{aligned}
$$

Here v_i, s_i are the names introduced above for the induction variable and increment of the global index value, as above $\{p_0, \ldots, p_{P-1}\}$ is the set of partial indexing patterns attached to i, and

$$\mathit{INCREMENT\_INDUCTION\_VARIABLE}\,(p) \equiv \quad v_p \texttt{ += } s_d \texttt{ ;}$$

Here v_p is the name of the induction variable associated with p, and s_d is the name of the temporary holding the increment for the dimension assignment, d, contained in the *head* of p.

Now the translation of a subscripting expression that is annotated with indexing pattern p will involve the value of the induction variable v_p. The details of this translation are properly speaking part of the translation functions for expressions, which will be described later.

The *shift step* for the index i is defined to be the value of e_{rng}`.str()`. This value is used in computation of offsets associated with shifted index subscripts.