

# Parallel Programming in HPJava

Bryan Carpenter, Guansong Zhang, Han-Ku Lee, . . .

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

March 20, 2001



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Processes and Arrays</b>	<b>7</b>
2.1	Processes and Process Grids . . . . .	7
2.2	Distributed Arrays . . . . .	10
2.3	Parallel Programming . . . . .	11
2.4	Locations . . . . .	14
2.5	A Complete Example . . . . .	16
<b>3</b>	<b>More on Mapping Arrays</b>	<b>19</b>
3.1	Other Distribution Formats . . . . .	19
3.2	Ghost Regions . . . . .	25
3.3	Collapsed Distributions and Sequential Dimensions . . . . .	31
3.4	Replication and Distribution Groups . . . . .	34
<b>4</b>	<b>Array Sections</b>	<b>39</b>
4.1	Two-dimensional Fourier transform . . . . .	40
4.2	Cholesky decomposition . . . . .	42
4.3	Matrix multiplication with reduced memory . . . . .	42
4.4	Subranges . . . . .	45
4.5	Restricted Groups . . . . .	46
4.6	Mapping of array sections . . . . .	47
4.7	Scalars . . . . .	48
4.8	Array restriction . . . . .	49
<b>5</b>	<b>Some Rules and Definitions</b>	<b>51</b>
5.1	Rules for distributed control constructs . . . . .	51
5.2	Rules for distributed array constructors . . . . .	52
5.3	Rules for access to distributed array elements . . . . .	53
5.3.1	Changes for ghost regions . . . . .	55
5.3.2	A final rule for array element access . . . . .	55
5.4	A recommendation for updating variables . . . . .	55

<b>6</b>	<b>A Distributed Array Communication Library</b>	<b>59</b>
6.1	Regular collective communications . . . . .	59
6.2	Reductions . . . . .	62
6.3	Irregular collective communications . . . . .	64
6.4	Schedules . . . . .	65
<b>7</b>	<b>Low level SPMD programming</b>	<b>67</b>
7.1	An Example . . . . .	67
7.2	Dimension Splitting . . . . .	73
7.3	Block Parameters . . . . .	78
	7.3.1 Ghost regions and dimension splitting . . . . .	82
	7.3.2 Local blocks of subranges . . . . .	82
7.4	Reduction to Java arrays . . . . .	85
7.5	Local arrays . . . . .	89
7.6	An extended example: prefix computation . . . . .	90
	7.6.1 Optimization for block distribution formats . . . . .	90
	7.6.2 Cyclic distribution formats . . . . .	94
	7.6.3 Optimization for “general” distribution formats . . . . .	94
7.7	Non-local blocks . . . . .	98
	7.7.1 The <code>crds()</code> method . . . . .	98
	7.7.2 Blocks of subranges . . . . .	101
7.8	Block cyclic distribution—up a level . . . . .	102
<b>8</b>	<b>Translation scheme</b>	<b>107</b>
8.1	Preliminaries . . . . .	107
	8.1.1 On distributed array types . . . . .	107
	8.1.2 HPspmd classes . . . . .	109
8.2	Translation functions . . . . .	111
8.3	Translating local variable declarations . . . . .	113
8.4	Translating field declarations . . . . .	115
8.5	Translating method declarations . . . . .	117
	8.5.1 Methods that return distributed arrays . . . . .	119
8.6	Translating <code>on</code> constructs . . . . .	121
8.7	Translating <code>at</code> constructs . . . . .	123
8.8	Translating <code>overall</code> constructs . . . . .	125
8.9	Translating global index expression . . . . .	127
8.10	Translating local variable accesses . . . . .	129
8.11	Translating assignment . . . . .	131
8.12	Translating distributed array creation . . . . .	133
8.13	Translating element access . . . . .	135
8.14	Translating array sections . . . . .	137
	8.14.1 Scalar subscripts in distributed dimensions . . . . .	139
8.15	Translating field accesses . . . . .	141
8.16	Translating method invocations . . . . .	143
	8.16.1 Methods that return distributed arrays . . . . .	145
	8.16.2 Translation of <code>return</code> statement . . . . .	147

8.17	Translating constructor invocations . . . . .	149
8.18	Translating array restriction . . . . .	151
8.19	Translating group restriction . . . . .	153
8.20	Translating subrange expressions . . . . .	155
8.21	Translating the conditional operator . . . . .	157
8.22	Translating casts . . . . .	161
8.23	Translating <b>instanceof</b> . . . . .	163
8.24	Translating reference equality . . . . .	165



# 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 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<sup>1</sup>. 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 ...  
}
```

---

<sup>1</sup>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.

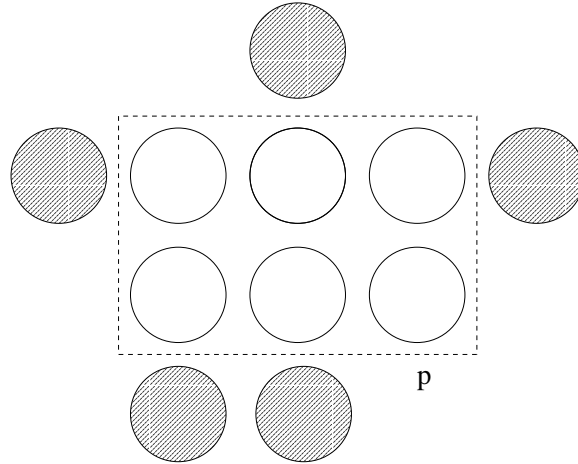


Figure 2.1: The process grid represented by `p`.

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 auxiliary class called `Dimension` is associated with process grids. Objects of this class describe a particular dimension or axis of a particular process grid. They will be called *process dimensions*. The process dimensions of a grid are available through the inquiry method `dim(r)`. The argument `r` is in the range  $0, \dots, R - 1$ , where in general  $R$  is the rank (dimensionality) of the grid.

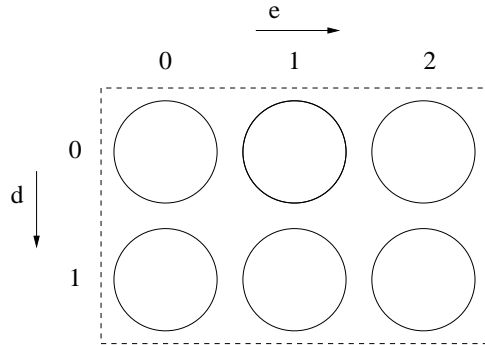
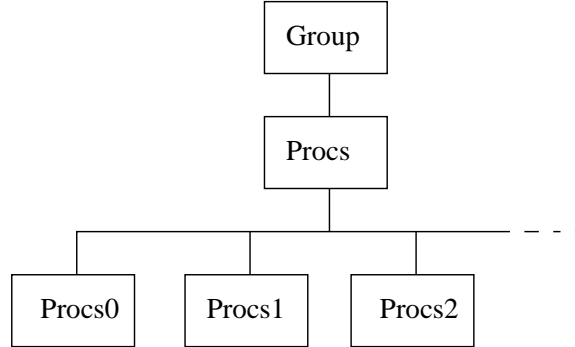
The `Dimension` class in turn has a method `crd`, which returns the local *process coordinate* associated with the dimension—ie, the position of the local process within the dimension. If we executed the following HPJava program

```
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` hierarchy 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<sup>2</sup>, but syntactically it is just a class.

---

<sup>2</sup>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`.

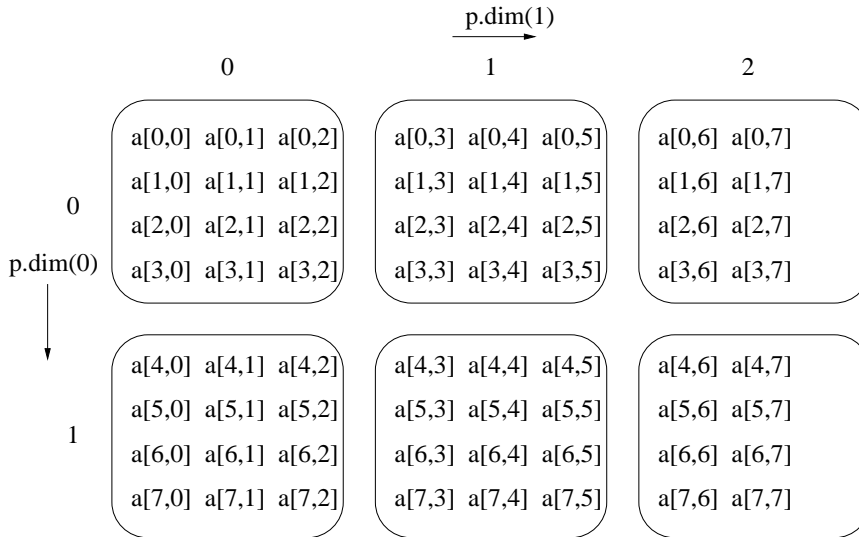
```
Procs2 p = new Procs2(2, 3) ;
on(p) {
    Range x = new BlockRange(N, p.dim(0)) ;
    Range y = new BlockRange(N, p.dim(1)) ;

    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<sup>3</sup>.

<sup>3</sup>Unless they are collapsed; see section 3.3.

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*<sup>4</sup>. 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 :)
```

<sup>4</sup>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<sup>5</sup>. 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<sup>6</sup>.

<sup>5</sup>Many of them are modelled on the array transformational intrinsic functions of Fortran 90.

<sup>6</sup>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 elements in parallel). it is not allowed to write simply

```

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

```

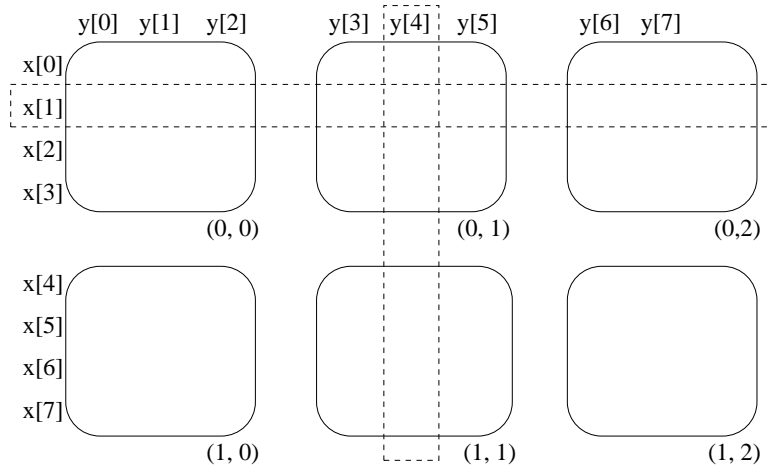
because 1 and 4 are not distributed indices, and therefore not legal subscripts. We *can* write:

```

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 = 1; 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<sup>7</sup>. 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-

<sup>7</sup>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.

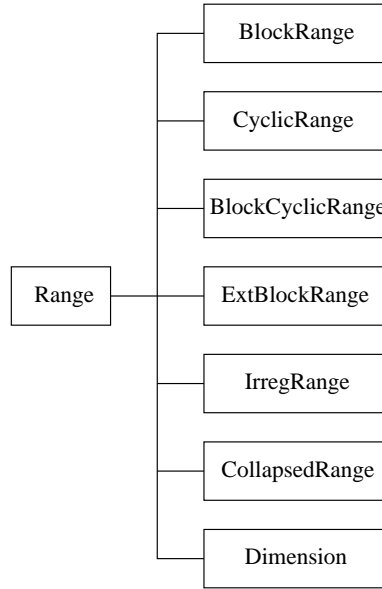
Cyclic distributions are sometimes favoured because they can lead to better *load balancing* than the simple block distributions introduced so far. Some algorithms (for example dense matrix algorithms) don't have the kind of locality that favours block distribution for stencil updates, but they do involve 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` hierarchy 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

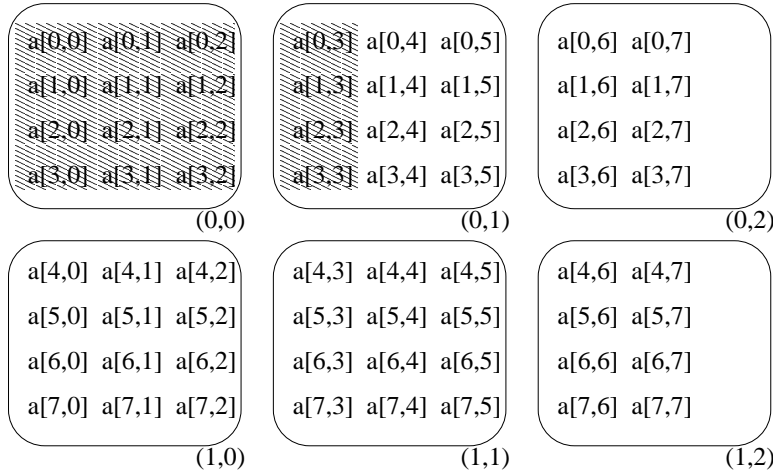


Figure 3.2: Work distribution for an example with block distribution.

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<sup>1</sup>. It will not be discussed further here.

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

---

<sup>1</sup>Notably ScaLAPACK.

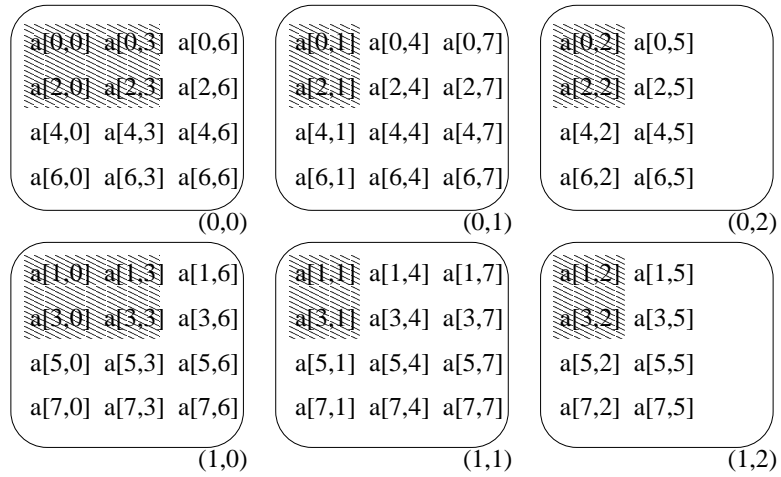


Figure 3.3: Work distribution for an example with cyclic distribution.



```

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.

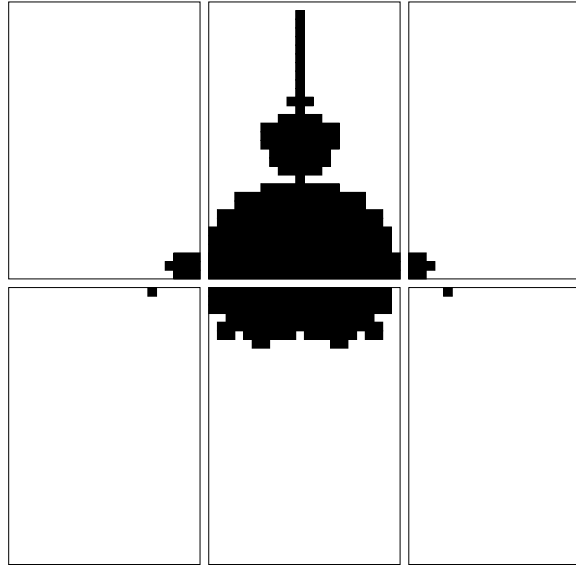


Figure 3.5: Blockwise decomposition of the Mandelbrot set (black region).

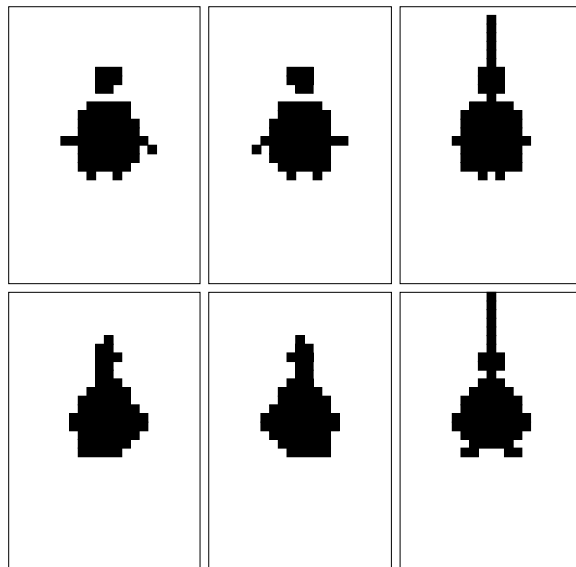


Figure 3.6: Cyclic decomposition of the Mandelbrot set.

## 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 be 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

$$name \pm expression$$

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<sup>2</sup>.

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 neighbouring 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 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

<sup>2</sup>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] + 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.



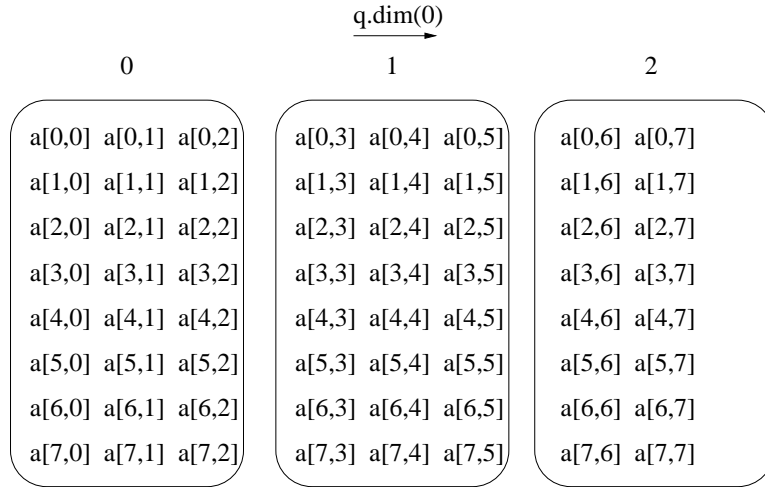


Figure 3.11: Two-dimensional array,  $a$ , distributed over the one-dimensional grid,  $q$ .

### 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 asterisk 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,

### 3.3. COLLAPSED DISTRIBUTIONS AND SEQUENTIAL DIMENSIONS35

```
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 [[*,j]] 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 expressions. 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

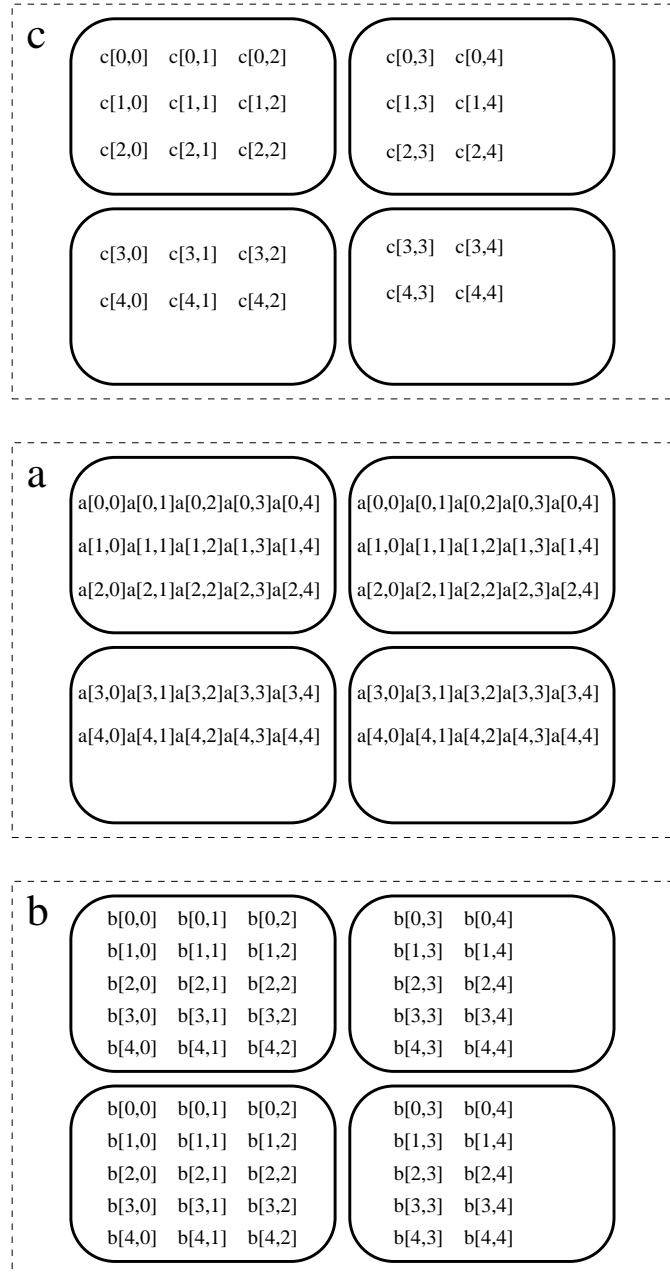


Figure 3.14: Distribution of array elements in example of Figure 3.13. Array **a** is replicated in every column of processes, array **b** is replicated in every row.

```
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 *r*th range of the array. The argument *r* is in the range  $0, \dots, R - 1$ , 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 process 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<sup>1</sup>.

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 to 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]]) ;
}
```

---

<sup>1</sup>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

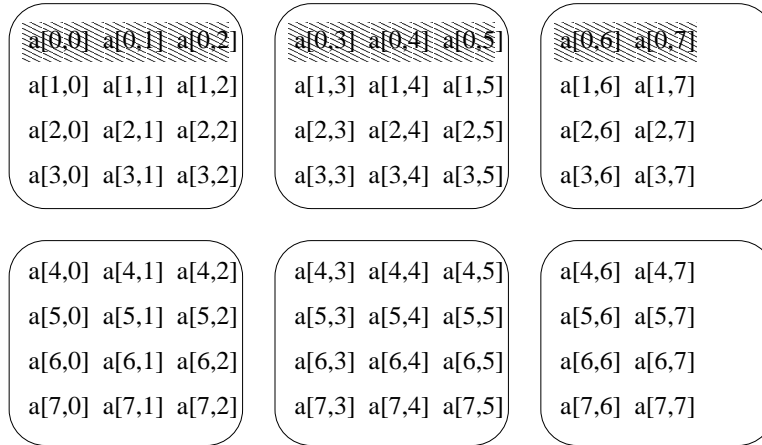


Figure 4.1: A one-dimensional section of a two-dimensional array (shaded area).

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<sup>2</sup>. 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.

<sup>2</sup>Distributed indices are allowed as well, if the location is in the proper range.

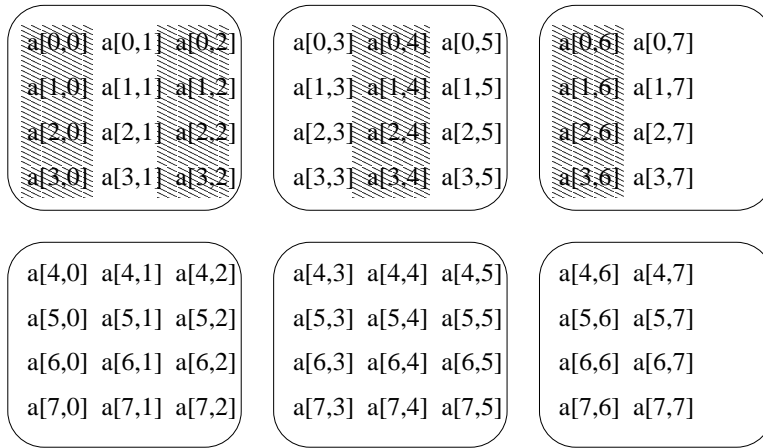


Figure 4.2: A two-dimensional section of a two-dimensional array (shaded area).

```

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:

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

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

```

For k = 1 to n - 1
  lkk = akk1/2
  For s = k + 1 to n
    lsk = ask/lkk
  For j = k + 1 to n
    For i = j to n
      aij = aij - likljk
  lnn = ann1/2

```

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  $\mathbf{a}$ . Note that the array  $\mathbf{b}$  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

$$B * \mathbf{x}.\text{volume}() + B * \mathbf{y}.\text{volume}()$$

```

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

$$p/i/j/\dots$$

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

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

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  $k$ th range of the section is determined by the  $k$ th triplet-valued subscript. If the  $k$ th triplet-valued subscript is  $l:u:s$  in dimension  $r$ , the  $k$ th range of the section is  $a.\text{rng}(r)[l:u:s]$ .

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

<sup>3</sup>If the  $r$ th 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<sup>4</sup>.

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.

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

The story of subranges and restricted groups repeats itself. The operation of array sectioning drives us to introduce a new kind of object into the language. Once that happens we should have a syntax for creating the new kind of object

---

<sup>4</sup>It also allows HPJava arrays to reproduce the full panoply of alignment options supported by the `ALIGN` directive of High Performance Fortran.

directly. Rank-0 distributed arrays, which we will also call simply “scalars”, can be created as follows

```
float # c = new float # ;

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

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

float d = c [] ;
```

This example illustrates one way to broadcast an element of a distributed array: remap it to a scalar replicated over the active process group. The element of the scalar is extracted through a distributed array element reference with an empty subscript list. 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<sup>5</sup>. 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 they 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

<sup>5</sup>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.

x. The sequential inner loop of Figure 3.13 is replaced by a call to `dotProduct` which directly forms the inner product of two sections with distributed range `z`.

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

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

The trouble is that according to the rules of the previous section the first argument of `dotProduct` has distribution group `p/i` whereas the second has distribution group `p/j`. So the arrays are not identically aligned. By forming restricted versions of both these sections we reduce both groups down to `p/i/j`. 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 the of the `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

```
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<sup>2</sup>.

## 5.2 Rules for distributed array constructors

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

<sup>1</sup>Alternatively  $x$  can be a collapsed range, in which case  $p/i$  is defined to be equal to  $p$ .

<sup>2</sup>Strictly speaking this nesting is legal (but pointless) if  $x$  is collapsed.

**Rule 3** *The distributed array constructor expression*

$$\text{new } T[[e_0, \dots, e_r, \dots]] \text{ on } p$$

*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  $e_0, \dots, e_r, \dots$  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*

$$a[e_0, \dots, e_r, \dots]$$

*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”<sup>3</sup> rules applies. To simplify the discussion, we first ignore ghost regions. The three rules are: either

**Rule 5** *In the distributed array element reference:*

$$a[e_0, \dots, e_r, \dots]$$

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

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

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

or

---

<sup>3</sup>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[e0, ..., er-1, i, er+1, ...] ...
}

```

*is allowed if and only if*

1. *The expression  $a$  is invariant in the **at** construct—it does not depend on  $i$ , or any variable declared or modified within the scope of  $i$ <sup>4</sup>.*
2. *The location  $x[n]$  is an element of the range  $a.\text{rng}(r)$ <sup>5</sup>.*

or

**Rule 7** *The distributed array element reference in:*

```

overall(i = x for l : u : s) {
    ... a[e0, ..., er-1, i, er+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  $a.\text{rng}(r)$ .*

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**.

---

<sup>4</sup>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.

<sup>5</sup>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 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 there 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 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<sup>6</sup>, the range of allowed subscripts is changed to

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

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

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

and the following requirement is added:

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

$$-x.\text{loExtension}() \leq \pm d \leq x.\text{hiExtension}()$$

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  $\mathbf{b}[\mathbf{j}]$  are legal— $\mathbf{j}$  is certainly a location in  $\mathbf{b}.\text{rng}(0)$  (which is equal to  $\mathbf{y}$ ). But, as illustrated, the section  $\mathbf{b}$  is localized to  $\mathbf{p}/\mathbf{x}[0]$ —the top row of processes in the figure—whereas the  $\mathbf{at}$  construct specifies that the element assignments are performed in the group  $\mathbf{p}/\mathbf{x}[\mathbf{N}-1]$ —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  $a.\text{grp}()$ .*

The error above is now exposed, because the distribution group of  $\mathbf{b}$  is  $\mathbf{p}/\mathbf{x}[0]$ . This does not contain the active process group inside the `overall` construct, namely  $\mathbf{p}/\mathbf{x}[\mathbf{N}-1]/\mathbf{j}$ . 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  $e_0, \dots, e_r, \dots$  in the element reference

$$a[e_0, \dots, e_r, \dots]$$

<sup>6</sup>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!
}

```

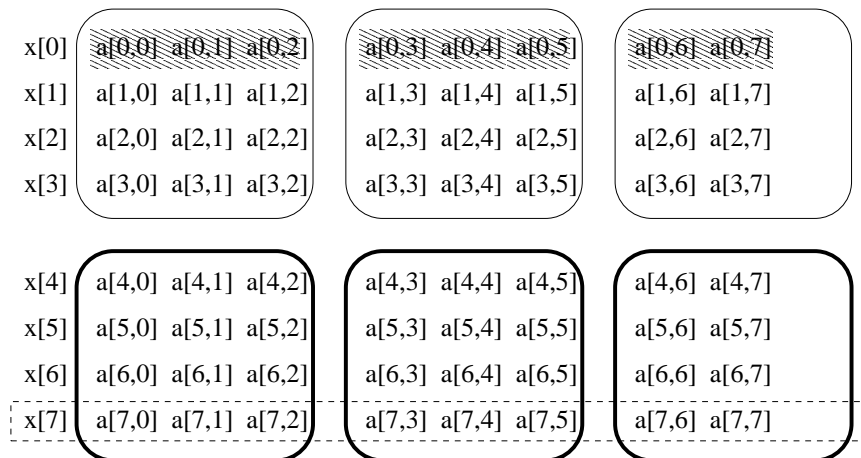


Figure 5.1: Access error discussed in text. Section b is shaded area at top. The at construct restricts control to the bottom row of processes.

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

$$p/i/j/\dots$$

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<sup>7</sup>. The canonical HPspmd style has a special affinity with the pure data parallel programming style of languages like HPF.

---

<sup>7</sup>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<sup>1</sup> 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

---

<sup>1</sup>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  $0, \dots, R-1$  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) ;
...
```



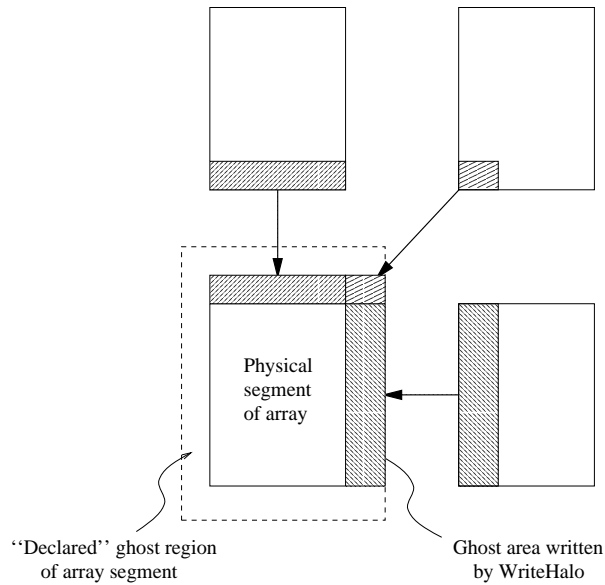


Figure 6.1: Illustration of the effect of executing the `writeHalo` function.

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 `broadcast` 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  $\{0, \dots, N-1\}$  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  $\{0, \dots, N-1\}$  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  $\{0, \dots, L-1\}$  in parallel do
  for all  $j$  in  $\{0, \dots, M-1\}$  in parallel do
    for all  $k$  in  $\{0, \dots, N-1\}$  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  $\{0, \dots, N-1\}$  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  $j$ th 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 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—

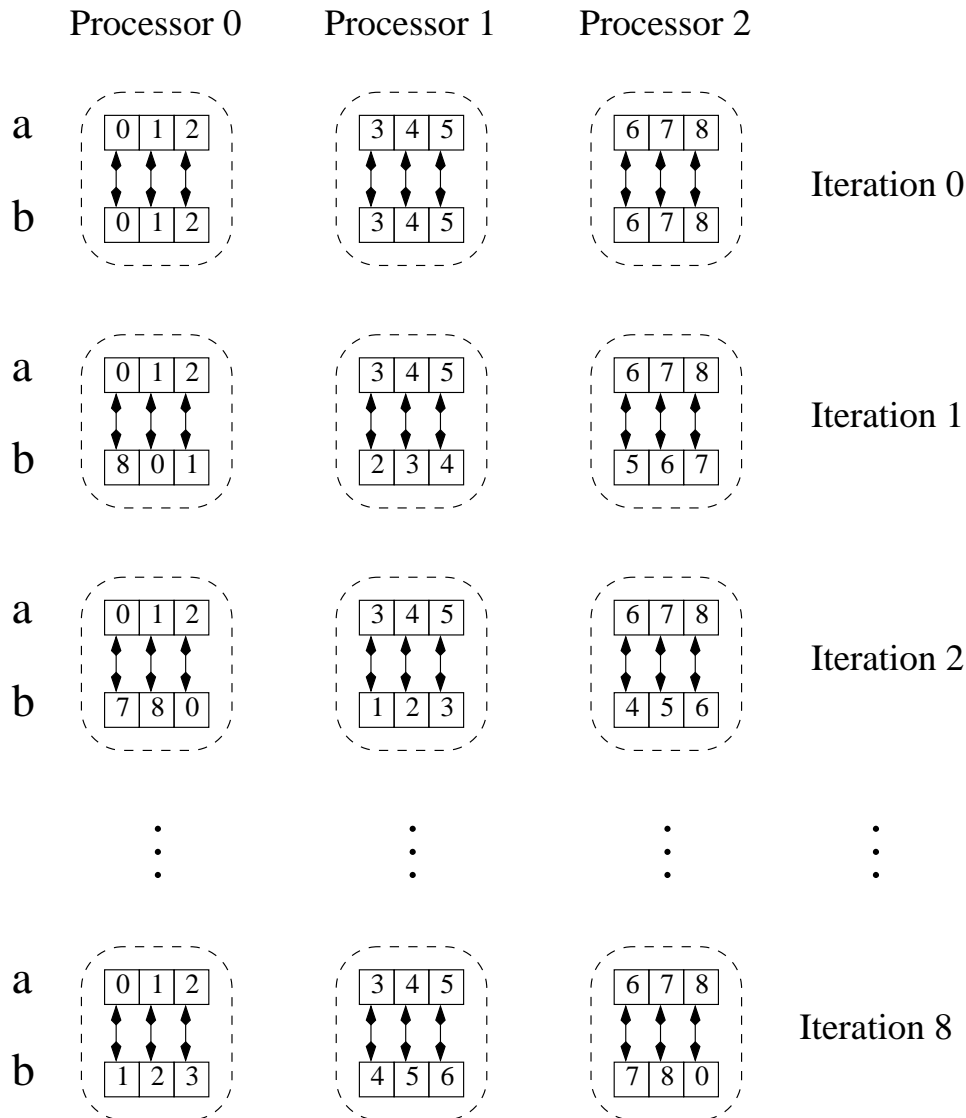


Figure 7.2: Simple “data parallel” N-body force computation. The array  $b$  is shifted one element to the right in each iteration. Arrows define element pairs combined in the iteration.

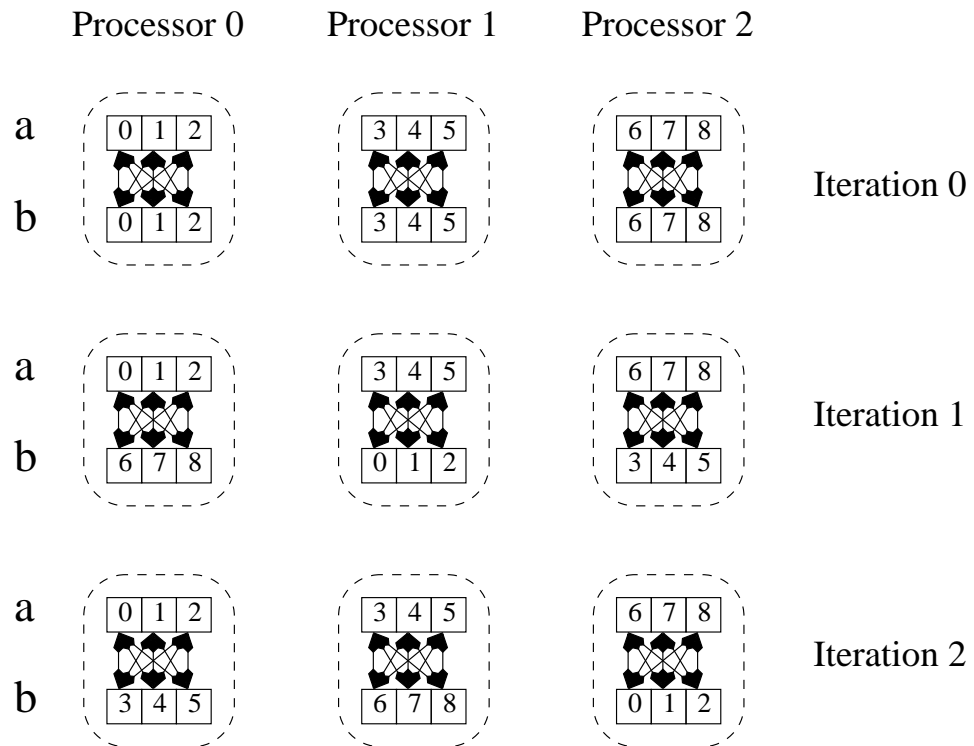


Figure 7.3: Efficient N-body force computation. The array *b* is shifted one *block* to the right in each iteration. Arrows define element pairs combined in the iteration.



```

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 HPsmd 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<sup>1</sup>; the local blocks of the original array are embedded in 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 initially 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  $c = a \times b$  on sequential two-dimensional arrays. It

---

<sup>1</sup>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  $P = 2$  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.

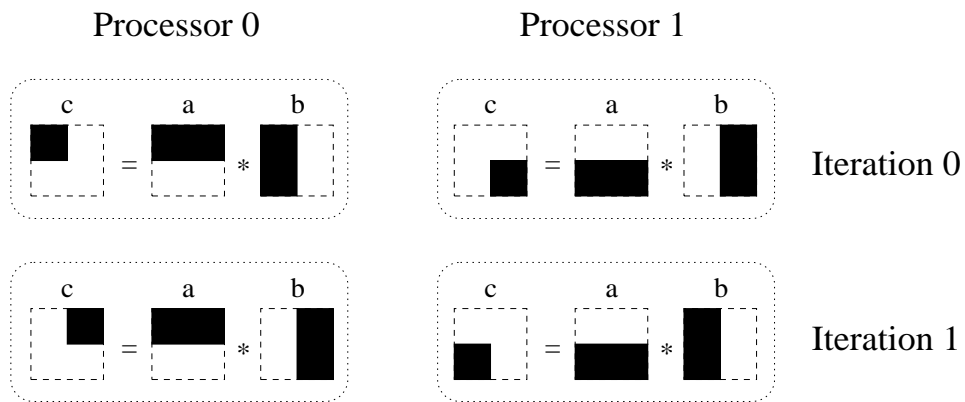


Figure 7.8: Operation of efficient pipelined matrix computation. The matrix  $b$  is shifted one block to the right in each iteration.

### 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  $\mathbf{a}$  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  $\mathbf{a}$  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  $\lceil 100/4 \rceil = 25^2$ .

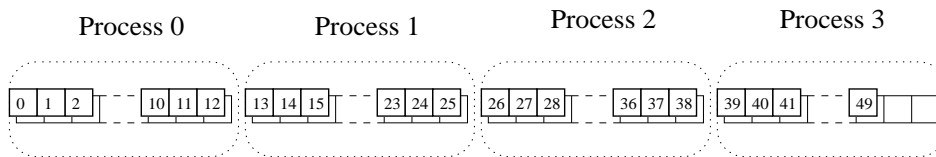
The embedding of actual elements is determined by a new method, `local-Block()`. This is a member of the `Range` class. It has no arguments, and returns

---

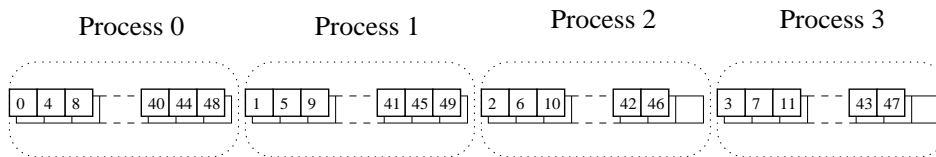
<sup>2</sup>In this case the dimension-split version of  $\mathbf{a}$  is the same as the dimension-split version of the parent array  $\mathbf{b}$ . A curious feature of a section with a dimension-splitting subscript is that the “section” may have *more* accessible elements than the parent array. This 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]] ;
```

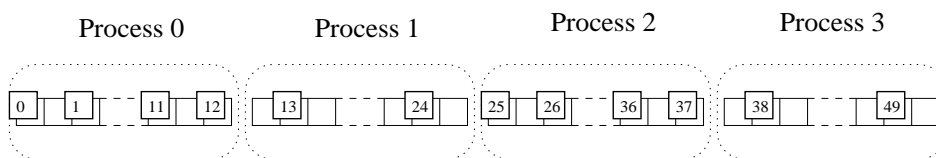


Figure 7.9: Example embeddings of array elements in local blocks

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) i4 ;
```

**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,  $0, \dots, B - 1$ , where typically  $B$  would be  $\lceil N/P \rceil$ . If the original range has lower and upper ghost extensions of width  $w_{lo}, w_{hi}$ , so does the new sequential range, exposed by dimension splitting. The allowed range of local subscripts will be  $-w_{lo}, \dots, B + w_{hi} - 1$ <sup>3</sup>. Figure 7.11 gives the recursive translation of an `overall` construct involving a shifted index. It assumes `a` has a suitable ghost extensions.<sup>4</sup>

### 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.

---

<sup>3</sup>Note however that the `size()` inquiry applied to the associated collapsed range will still return the “physical” extent,  $B$ .

<sup>4</sup>This example actually assumes  $x$  has alignment stride of 1. In general the displacement in the translation should be multiplied by  $x.\text{str}()$ , 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<sup>5</sup>. If an element of a sequential HPJava array, `a`, has integer subscripts  $i_0, \dots, i_{R-1}$  it is stored in element

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

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  $i_0, \dots, i_{R-1}$ —a mix of integers and level 0 distributed index symbols—the local element is

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

---

<sup>5</sup>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) × 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 sequential 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<sup>6</sup>:

```
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]]] ;
}
```

---

<sup>6</sup>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:

$$r[i] = a[0] + \dots + a[i]$$

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

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

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();

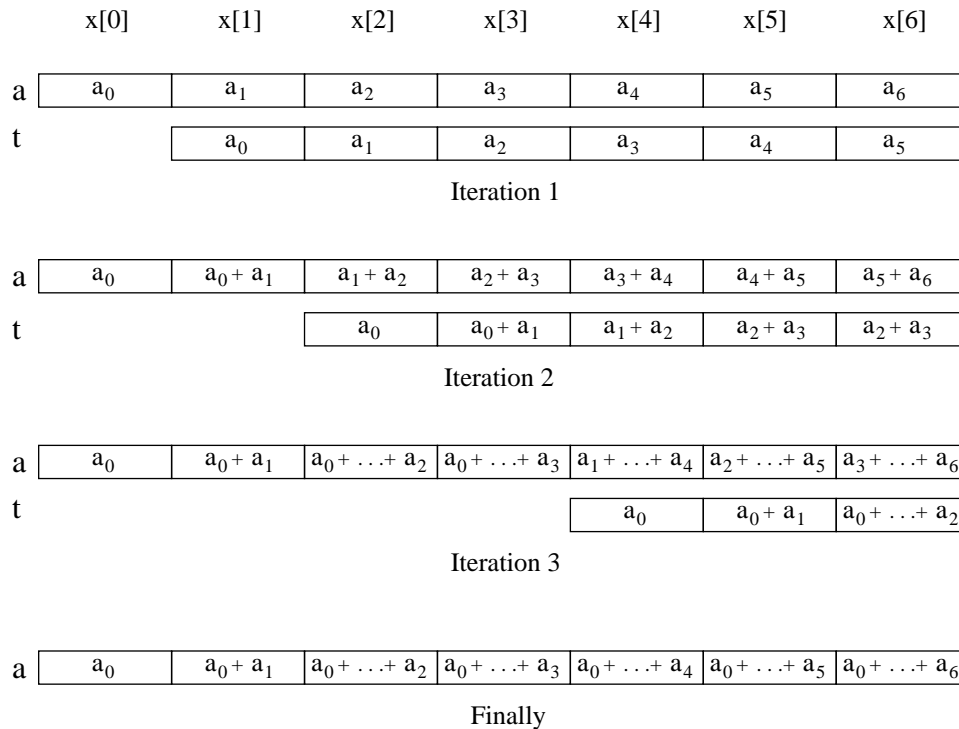
    on(p) {
        float [][] t = new float [[x]] ;

        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.

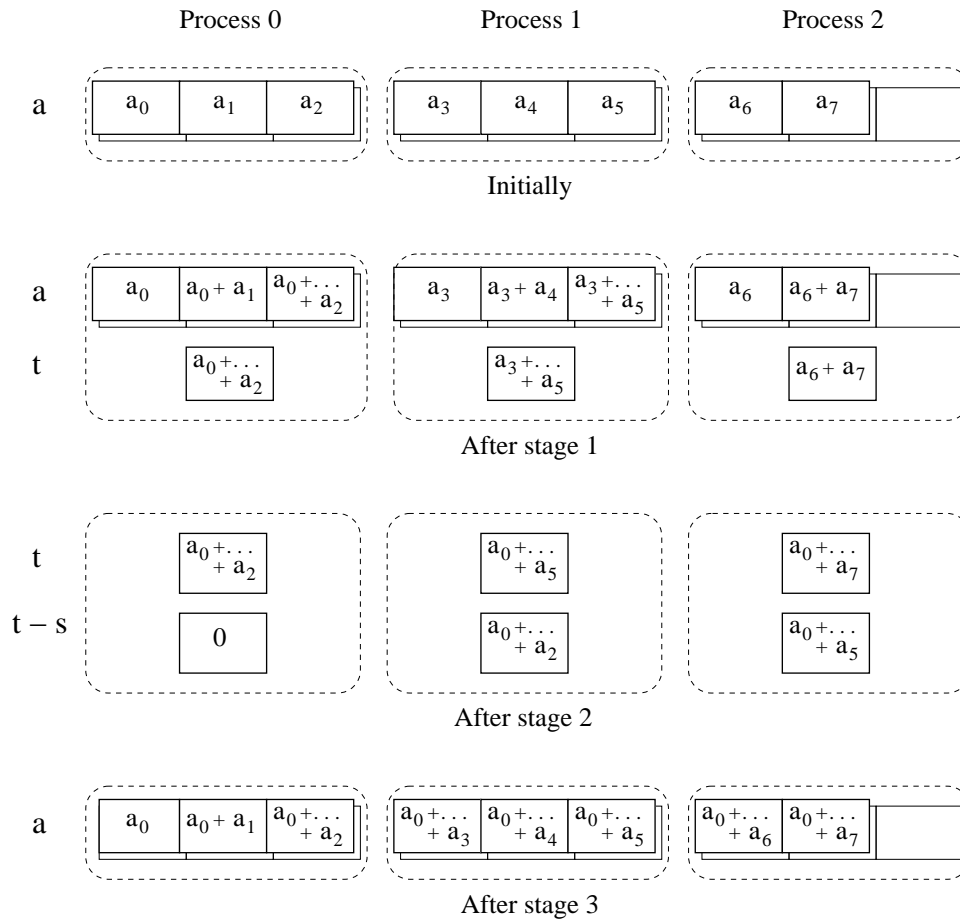


Figure 7.18: Prefix optimized for block distributions: illustration for  $N = 8$ ,  $P = 3$ .

The code given here will work for the ranges mentioned above, and subranges of these<sup>7</sup>. 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”

<sup>7</sup>At least for subranges with positive alignment stride. We will return to this issue in section 7.6.3



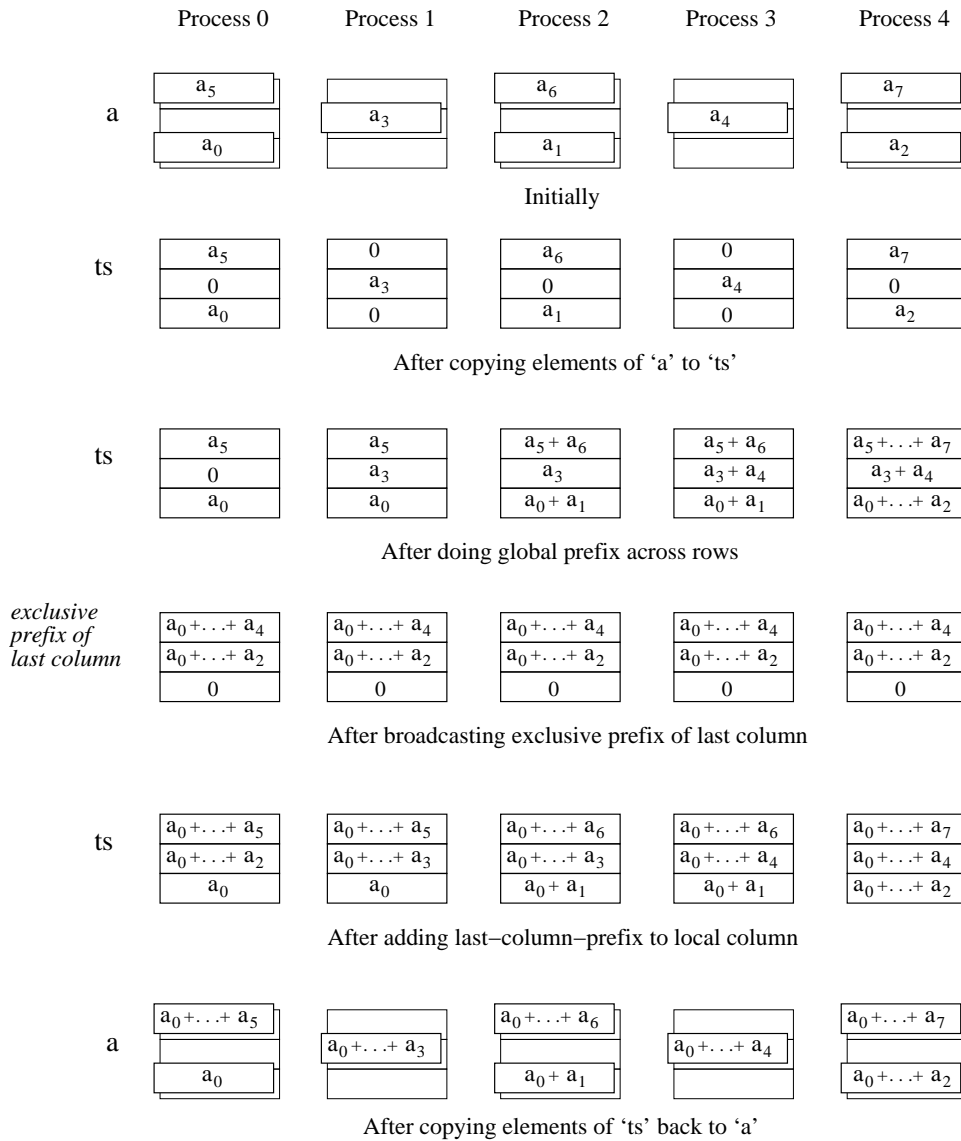


Figure 7.19: Possible optimization of prefix for cyclic distributions: illustration for stride 2 subrange of cyclic range with extent 15. P = 5.

```

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<sup>8</sup>:

```

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

---

<sup>8</sup>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, a new method `crds()` is added to the `Range` class. It takes no arguments, and returns an object of class `Triplet`,

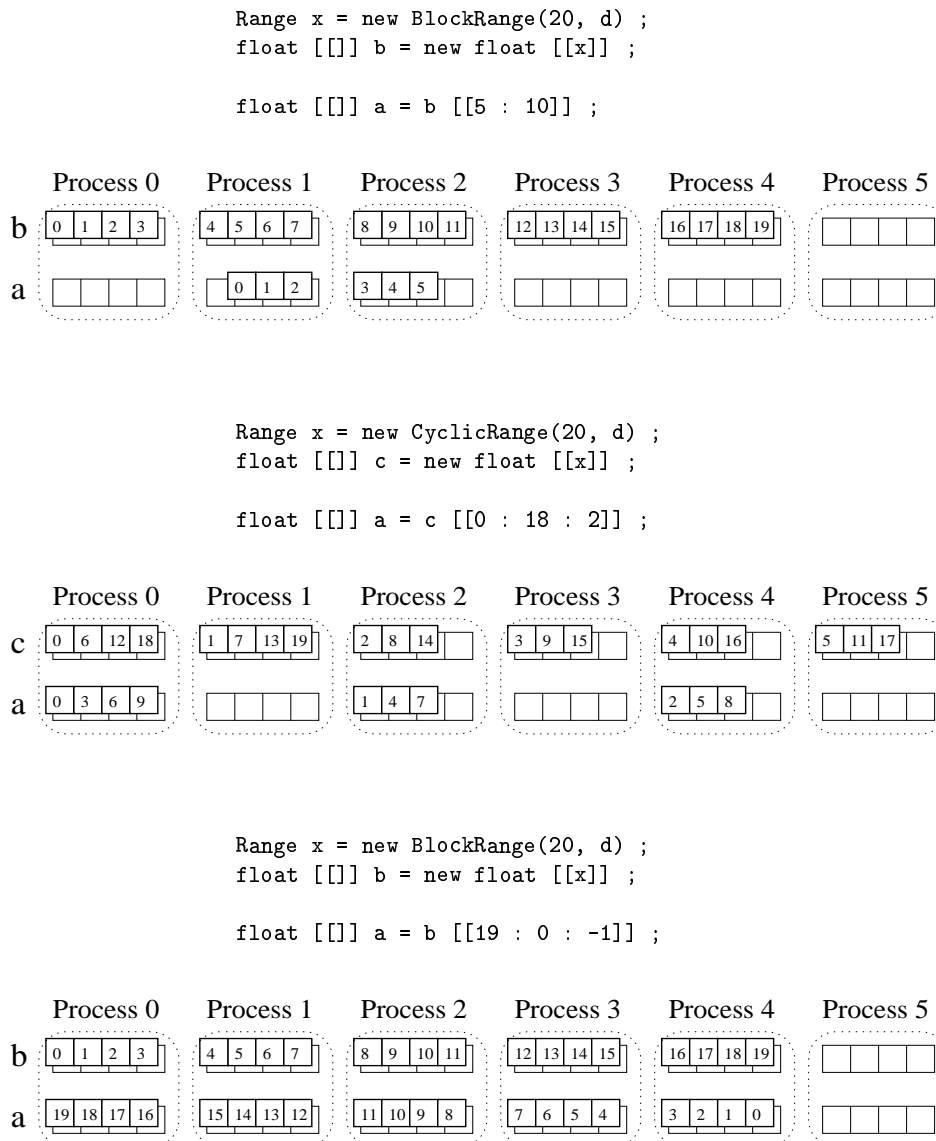


Figure 7.21: Example sections with unusual coordinate ranges.

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:

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

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(crd.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<sup>9</sup>.

### 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.

---

<sup>9</sup>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()` inquiries (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 transformation to the generated `overall`, as illustrated in Figure 7.23<sup>10</sup>.

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]) {
        ...
    }
```

---

<sup>10</sup>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.crd(1, u, s);

overall(k = d for t.lo : t.hi : t.stp) {
  Block b = x.block(k', 1, 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(1, u, s);

overall(k = d for t.lo : t.hi : t.stp) {
  Block b = x.block(k', 1, 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(1, 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, 1, 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<sup>11</sup>.

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.

---

<sup>11</sup>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 HPsmd 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:

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

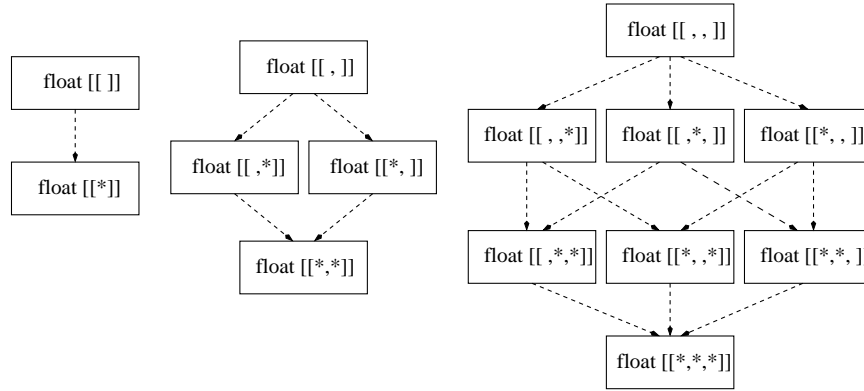
where  $R$  is the rank of the array and each term  $attr_r$  is either empty or consists of a single asterisk,  $*$ . In principle  $T$  can be *any* Java type.

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<sup>1</sup>. 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.

---

<sup>1</sup>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`.

Although it is *possible* for code in a non-HPspmd classes to use HPspmd members, it requires detailed knowledge of the transformations the HPJava translator applies to signatures and distributed array variables, and in general it is discouraged. Of course (emphatically!) an HPspmd class can freely *use* any non-HPspmd Java class, limited only by the normal accessibility rules.

The HPJava translator will 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 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 a kind of Java class, whereas distributed array types are not Java classes of any kind.



## 8.2 Translation functions

This assumes the existence of several “translation functions”. A function on expression terms,  $\mathbf{T}_{\text{val}}[e]$ , 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. These statements may, for example, assign values to temporary variables. The sequence of statements that needs to be executed before  $\mathbf{T}_{\text{val}}[e]$  is evaluated is given by the function  $\mathbf{T}_{\text{pre}}[e]$ .

For convenience, we introduce the variants  $\mathbf{T}_{\text{val}}^V$ ,  $\mathbf{T}_{\text{pre}}^V$ , which define a translation of  $e$  that is guaranteed to return a simple name for  $\mathbf{T}_{\text{val}}^V[e]$  (the value associated with which is invariant in the part of the program that uses the result) by having  $\mathbf{T}_{\text{pre}}^V[e]$  define an extra temporary variable to hold the expression value, if necessary.

Translation functions for array expressions are more complicated.  $\mathbf{T}_{\text{pre}}[e]$  works in the same way, but now there are  $3 + R$  separate parts of the evaluation:  $\mathbf{T}_{\text{dat}}[e]$ ,  $\mathbf{T}_{\text{bas}}[e]$ ,  $\mathbf{T}_{\text{grp}}[e]$  and  $\mathbf{T}_0[e], \dots, \mathbf{T}_{R-1}[e]$ , where  $R$  is the rank of the array. The interpretation of these separate terms will be given in the following sections.

As a shorthand we will allow the function  $\mathbf{T}_{\text{pre}}$  to be applied to more general syntactic terms, including lists of expressions and triplet terms. This should be interpreted as a sequence of applications to the subexpressions in the argument.

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:**

$s \equiv S$

**TRANSLATION:**

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

**SOURCE:**

$$T \text{ } [[attr_0, \dots, attr_{R-1}]] \text{ } a \text{ } ;$$
**TRANSLATION:**

$$T \text{ } [] \text{ } a'_{\text{dat}} ;$$

$$\text{int } a'_{\text{bas}} ;$$

$$\text{Group } a'_{\text{grp}} ;$$

$$\text{DIMENSION\_TYPE}(attr_0) \text{ } a'_0 \text{ } ;$$

$$\dots$$

$$\text{DIMENSION\_TYPE}(attr_{R-1}) \text{ } a'_{R-1} \text{ } ;$$

where:

$T$  is a Java type,

each term  $attr_r$  is either empty or a single asterisk,  $*$ ,

$a$  is an array name in the source program, rank  $R$ ,

$a'_{\text{dat}}$ ,  $a'_{\text{bas}}$ ,  $a'_{\text{grp}}$ , and  $a'_0, \dots, a'_{R-1}$  are names of new variables, and  
the macro  $\text{DIMENSION\_TYPE}$  is defined in the text.

Figure 8.2: Translation of local distributed array declaration.

### 8.3 Translating local variable declarations

The general scheme for translating declaration of a local variable holding a distributed array reference is illustrated in Figure 8.2. The single variable in the source program is converted to  $3 + R$  variables in the output program, where  $R$  is the rank of the array.

The variable  $a'_{\text{dat}}$  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  $a.\text{dat}()$  in the source program. The variable  $a'_{\text{bas}}$  will hold 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  $a.\text{bas}()$  in the source program). The variable  $a'_{\text{grp}}$  will hold a reference to the distribution group of the array (the value that would be returned by the inquiry  $a.\text{grp}()$  in the source program).

The  $R$  variables  $a'_0, \dots, a'_{R-1}$  will hold descriptors for the dimensions of the arrays. The macro *DIMENSION\_TYPE* is defined as

$$DIMENSION\_TYPE(attr_r) \equiv \text{ArrayDim}$$

if the term  $attr_r$  is empty, or

$$DIMENSION\_TYPE(attr_r) \equiv \text{SeqArrayDim}$$

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  $a.\text{rng}(r)$  and  $a.\text{str}(r)$ , 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.

**SOURCE:**

$$T \text{ } [[attr_0, \dots, attr_{R-1}]] \text{ } id \text{ } ;$$
**TRANSLATION:**

```

T [] TRANS_ID(id) ;

int TRANS_ID_BAS(id) ;
Group TRANS_ID_GRP(id) ;

DIMENSION_TYPE(attr_0) TRANS_ID_DIM(id, 0) ;
...
DIMENSION_TYPE(attr_{R-1}) TRANS_ID_DIM(id, R - 1) ;

```

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 either empty or a single asterisk, \*,  
the identifier  $id$  is the name of the declared field, and  
the macros  $TRANS\_ID$ ,  $TRANS\_ID\_BAS$ ,  $TRANS\_ID\_GRP$  and  
 $TRANS\_ID\_DIM$  and  $DIMENSION\_TYPE$  are defined in the text.

Figure 8.3: Translation of a distributed-array-valued field declaration.

## 8.4 Translating field declarations

The general scheme for translating declaration of a field holding a distributed array reference is illustrated in Figure Figure 8.16. This scheme is almost identical to the translation for a local variable declaration given in section 8.3, but in this case we have to be specific about the names of the generated fields, because they are exposed in (for example) the generated class files.

The macro *TRANS\_ID* transforms the identifier of the field in a way that encodes the dimension signature of the result. This information is necessary in order that the HPJava signature of the method can be reconstructed from the class file of the 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*, *TRANS\_ID\_GRP* and *TRANS\_ID\_DIM* all start with the prefix “\_\_HPJ\_”. This is followed by a string characteristic of the individual macro, described below. The characteristic string is terminated by one underscore, “\_”, and this underscore is followed by the original *id* string.

The characteristic string for *TRANS\_ID(id)* 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(id)* is “BAS”. The characteristic string for *TRANS\_ID\_GRP(id)* is “GRP”. The characteristic string for *TRANS\_ID\_DIM(id, r)* is the decimal representation of the constant integer value *r*.

The macro *DIMENSION\_TYPE* is defined in section 8.3.

If the original class has a field:

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

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

```
float [] __HPJ_DDS_bar ;

int __HPJ_BAS_bar ;
Group __HPJ_GRP_bar ;

ArrayDim __HPJ_0_bar ;
ArrayDim __HPJ_1_bar ;
SeqArrayDim __HPJ_2_bar ;
```

Unless the class containing the field declaration is an inner class, defined inside a method of an HPspmd class, there is no active process group in effect at the point at which the declaration appears. Consequently an initializer for a field usually *cannot* contain calls to methods of HPspmd classes, or distributed array creation expressions.

**SOURCE:**

$$T \text{ id}(U_0 \ v_0, \dots, U_{N-1} \ v_{N-1}) \{S\}$$
**TRANSLATION:**

$$\begin{array}{l} T \text{ id}(TRANS\_PARAMS(U_0 \ v_0, \dots, U_{N-1} \ v_{N-1}), \text{Group } p) \{ \\ \quad \mathbf{T}[S|p] \\ \} \end{array}$$

where:

$T$  is a Java type,  
 the identifier  $id$  is the name of the method,  
 each of the terms  $U_0, \dots, U_{N-1}$  is a Java type or a distributed array type,  
 $v_0, \dots, v_{N-1}$  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 8.4: Translation of method declaration.

## 8.5 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 8.4. 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:

$$\begin{aligned} \text{TRANS\_PARAMS}(U_0 \ v_0, \dots, U_{N-1} \ v_{N-1}) \equiv \\ \text{TRANS\_PARAM}(U_0 \ v_0), \ \dots, \ \text{TRANS\_PARAM}(U_{N-1} \ v_{N-1}) \end{aligned}$$

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

$$\text{TRANS\_PARAM}(U \ v) \equiv U \ v$$

Otherwise, if  $U$  has the form

$$T \ [[\text{attr}_0, \dots, \text{attr}_{R-1}]] \ a \ ;$$

where as usual  $T$  is a Java type and each term  $\text{attr}_r$  is either empty or a single asterisk, `*`, then the macro *TRANS\_PARAM* is defined by

$$\begin{aligned} \text{TRANS\_PARAM}(U \ v) \equiv \\ T \ [] \ a'_{\text{dat}}, \ \text{int} \ a'_{\text{bas}}, \ \text{Group} \ a'_{\text{grp}}, \\ \text{DIMENSION\_TYPE}(\text{attr}_0) \ a'_0, \ \dots, \\ \text{DIMENSION\_TYPE}(\text{attr}_{R-1}) \ a'_{R-1} \end{aligned}$$

where  $a'_{\text{dat}}$ ,  $a'_{\text{bas}}$ ,  $a'_{\text{grp}}$ , and  $a'_0, \dots, a'_{R-1}$  are new formal parameter names. The macro *DIMENSION\_TYPE* is defined in section 8.3.

In other words, each distributed array parameter is split into  $3 + R$  parameters.

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

**SOURCE:**

$$T \text{ } [[attr_0, \dots, attr_{R-1}]] \text{ } id(U_0 \text{ } v_0, \dots, U_{N-1} \text{ } v_{N-1}) \{$$

$$S$$

$$\}$$
**TRANSLATION:**

$$T \text{ } \square \text{ } TRANS\_ID(id)(DAD \text{ } d,$$

$$TRANS\_PARAMS(U_0 \text{ } v_0, \dots, U_{N-1} \text{ } v_{N-1}),$$

$$\text{Group } p) \{$$

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

$$\}$$

where:

$T$  is a Java type,  
 $R$  is the rank of the returned array,  
 each term  $attr_r$  is either empty or a single asterisk, \*,  
 the identifier  $id$  is the name of the method,  
 each of the terms  $U_0, \dots, U_{N-1}$  is a Java type or a distributed array type,  
 $v_0, \dots, v_{N-1}$  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 8.5: Translating declaration of method returning a distributed array.



### 8.5.1 Methods that return distributed arrays

The scheme for translating a distributed-array-valued method declaration is illustrated in Figure 8.5.

The macro *TRANS\_ID* encodes the dimension signature of the returned array and is defined in section 8.4. The macro *TRANS\_PARAMS* was defined earlier in this section.

Instances of the class **DAD** (the initials stand for *Distributed Array Descriptor*) contain an integer **base**, a reference to a group object, and a vector of *R* instances of **ArrayDim**. These values are stored in fields **base**, **group**, and **dimensions**, respectively. Before the translated method returns, it will store the parameters describing the layout of the distributed array result in these fields (see sections 8.16.1 and 8.16.2).

**SOURCE:**

on ( $e_{\text{grp}}$ )  $S$

**TRANSLATION:**

```

 $\mathbf{T}_{\text{pre}}^V [e_{\text{grp}}]$ 
   $ASSERT(apg.contains(e'_{\text{grp}}))$ 

  if ( $e'_{\text{grp}}.member()$ ) {
     $\mathbf{T} [S | e'_{\text{grp}}]$ 
  }

```

where:

$e_{\text{grp}}$  is an expression in the source,  
 $S$  is a statement in the source program,  
 $e'_{\text{grp}} = \mathbf{T}_{\text{val}}^V [e_{\text{grp}}]$ , and  
the macro  $ASSERT$  is defined in the text.

Figure 8.6: Translation of **on** construct.

## 8.6 Translating on constructs

A translation for the `on` construct is given in Figure 8.6.

The *ASSERT* macro throws an exception if its argument is boolean `false`.

**SOURCE:**

at ( $i = e_{\text{rng}}[e_{\text{glb}}]$ )  $S$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}^V [e_{\text{rng}}]$$

$$\mathbf{T}_{\text{pre}} [e_{\text{glb}}]$$

$$\text{Location } l = e'_{\text{rng}}.\text{location}(\mathbf{T}_{\text{val}} [e_{\text{glb}}]) ;$$

$$\text{Dimension } d = e'_{\text{rng}}.\text{dim}() ;$$

$$\text{if } (d.\text{crd}() == l.\text{crd}) \{$$

$$\quad \text{Group } p = ((\text{Group}) \text{ } \text{apg}.\text{clone}()).\text{restrict}(d) ;$$

$$\quad \mathbf{T}[S|p]$$

$$\}$$

where:

$i$  is an index name in the source program,  
 $e_{\text{rng}}$  and  $e_{\text{glb}}$  are expressions in the source,  
 $S$  is a statement in the source program,  
 $e'_{\text{rng}} = \mathbf{T}_{\text{val}}^V [e_{\text{rng}}]$ , and  
 $l$ ,  $d$  and  $p$  are the names of new variables.

Figure 8.7: Translation of **at** construct.

## 8.7 Translating at constructs

A translation for the `at` construct is given in Figure 8.7.

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 `restrict()` method will throw an exception if  $d$  is not an effective dimension of *apg*.

The *local subscript* for the index  $i$  is the value of  $l.\text{sub}$ . This value is used in subscripting distributed arrays.

The *global index* for the index  $i$  is the value of  $\mathbf{T}_{\text{val}}^V [e_{\text{glb}}]$ . 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  $e'_{\text{rng}}.\text{str}()$ . This value is used in computation of offsets associated with shifted index subscripts.

**SOURCE:**

```
overall (i = erng for elo : ehi : estp) S
```

**TRANSLATION:**

```

 $\mathbf{T}_{\text{pre}}^V [e_{\text{rng}}]$ 
 $\mathbf{T}_{\text{pre}} [e_{\text{lo}}, e_{\text{hi}}, e_{\text{stp}}]$ 
Block  $b = e'_{\text{rng}}.\text{localBlock}(\mathbf{T}_{\text{val}}[e_{\text{lo}}], \mathbf{T}_{\text{val}}[e_{\text{hi}}], \mathbf{T}_{\text{val}}[e_{\text{stp}}]) ;$ 
Group  $p = ((\text{Group}) \text{ app.clone()}).\text{restrict}(e'_{\text{rng}}.\text{dim}()) ;$ 
for (int  $l = 0 ; l < b.\text{count} ; l++$ ) {
    int  $sub = b.\text{sub\_bas} + b.\text{sub\_stp} * l ;$ 
    int  $glb = b.\text{glb\_bas} + b.\text{glb\_stp} * l ;$ 
     $\mathbf{T}[S|p]$ 
}

```

where:

$i$  is an index name in the source program,  
 $e_{\text{rng}}$ ,  $e_{\text{lo}}$ ,  $e_{\text{hi}}$ , and  $e_{\text{stp}}$  are expressions in the source,  
 $S$  is a statement in the source program,  
 $e'_{\text{rng}} = \mathbf{T}_{\text{val}}^V [e_{\text{rng}}]$ , and  
 $b$ ,  $p$ ,  $l$ ,  $sub$  and  $glb$  are names of new variables.

Figure 8.8: Translation of `overall` construct.

## 8.8 Translating overall constructs

A translation for the `overall` construct is given in Figure 8.8. 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  $e'_{\text{rng}}.\text{str}()$ . This value is used in computation of offsets associated with shifted index subscripts.

The `restrict()` method will throw an exception if  $e'_{\text{rng}}.\text{dim}()$  is not an effective dimension of *apg*.

**SOURCE:**

$$e \equiv i'$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv (\text{empty})$$

$$\mathbf{T}_{\text{val}}[e] \equiv glb$$

where:

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

Figure 8.9: Translation of global index for  $i$ .



## 8.9 Translating global index expression

The scheme is illustrated in Figure 8.9. The global index variable associated with a distributed index is defined in sections 8.7 and 8.8.

**SOURCE:**

$$e \equiv a$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv (\text{empty})$$

$$\mathbf{T}_{\text{dat}}[e] \equiv a'_{\text{dat}}$$

$$\mathbf{T}_{\text{bas}}[e] \equiv a'_{\text{bas}}$$

$$\mathbf{T}_{\text{grp}}[e] \equiv a'_{\text{grp}}$$

$$\mathbf{T}_0[e] \equiv a'_0$$

...

$$\mathbf{T}_{R-1}[e] \equiv a'_{R-1}$$

where:

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

Figure 8.10: Translation of a distributed-array-valued local variable access.

## 8.10 Translating local variable accesses

We only need to consider the case where the variable is a distributed array. The general scheme is illustrated in Figure 8.10.

The names  $a'_{\text{dat}}$ ,  $a'_{\text{bas}}$ ,  $a'_{\text{grp}}$ , and  $a'_0, \dots, a'_{R-1}$  are the names introduced by the translator when translating the corresponding local variable declaration or formal parameter in the source program (see sections 8.3 and 8.5).

**SOURCE:**

$$e \equiv e_t = e_s$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_t] \\ \mathbf{T}_{\text{pre}}[e_s]$$

$$\mathbf{T}_{\text{dat}}[e] \equiv \mathbf{T}_{\text{dat}}[e_t] = \mathbf{T}_{\text{dat}}[e_s]$$

$$\mathbf{T}_{\text{bas}}[e] \equiv \mathbf{T}_{\text{bas}}[e_t] = \mathbf{T}_{\text{bas}}[e_s]$$

$$\mathbf{T}_{\text{grp}}[e] \equiv \mathbf{T}_{\text{grp}}[e_t] = \mathbf{T}_{\text{grp}}[e_s]$$

$$\mathbf{T}_0[e] \equiv \mathbf{T}_0[e_t] = \mathbf{T}_0[e_s]$$

...

$$\mathbf{T}_{R-1}[e] \equiv \mathbf{T}_{R-1}[e_t] = \mathbf{T}_{R-1}[e_s]$$

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 8.11: Translation of a distributed-array assignment.

## 8.11 Translating assignment

When the expressions involved are distributed arrays, the general scheme is illustrated in Figure 8.11.

**SOURCE:**

$$e \equiv \text{new } T \ [[e_0, \dots, e_{R-1}]] \text{ on } e_{\text{grp}}$$
**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}^V[e_0] \\ &\dots \\ &\mathbf{T}_{\text{pre}}^V[e_{R-1}] \\ &\mathbf{T}_{\text{pre}}^V[e_{\text{grp}}] \\ \\ s &= 1 ; \\ b &= 0 ; \\ &DEFINE\_DIMENSION(a_{R-1}, \mathbf{T}_{\text{val}}^V[e_{R-1}], s, b) \\ &\dots \\ &DEFINE\_DIMENSION(a_0, \mathbf{T}_{\text{val}}^V[e_0], s, b) \\ \\ \mathbf{T}_{\text{dat}}[e] &\equiv e'_{\text{grp}}.\text{member}() ? \text{new } T \ [s] : \text{null} \\ \mathbf{T}_{\text{bas}}[e] &\equiv b \\ \mathbf{T}_{\text{grp}}[e] &\equiv e'_{\text{grp}} \\ \mathbf{T}_0[e] &\equiv a_0 \\ &\dots \\ \mathbf{T}_{R-1}[e] &\equiv a_{R-1} \end{aligned}$$

where:

$T$  is a Java type,  
 $R$  is the rank of the created array,  
each expression  $e_r$  is either range-valued or integer-valued,  
 $a$  is the assigned array name in the source program,  
 $s$ ,  $b$  and  $a_0, \dots, a_{R-1}$  are names of new temporaries,  
 $e'_{\text{grp}} = \mathbf{T}_{\text{val}}^V[e_{\text{grp}}]$ , and  
the macro *DEFINE\_DIMENSION* is defined in the text.

Figure 8.12: Translation of distributed array creation expression.

## 8.12 Translating distributed array creation

The general scheme for translating creation of a distributed array is illustrated in Figure 8.12. If the “on  $e_{\text{grp}}$ ” clause is omitted in the source program, the value of  $e'_{\text{grp}}$  in the translation can be taken to be *agg*.

The macro *DEFINE\_DIMENSION* is defined as follows:

```

DEFINE_DIMENSION( $a_r, e'_r, s, b$ ) ≡
     $a_r = e'_r.\text{arrayDim}(s)$  ;
     $b += s * e'_r.\text{loExtension}()$  ;
     $s *= e'_r.\text{volume}()$  ;

```

if the expression  $e'_r$  is a range, or

```

DEFINE_DIMENSION( $a_r, e'_r, s, b$ ) ≡
     $a_r = \text{new SeqArrayDim}(e'_r, s)$  ;
     $s *= e'_r$  ;

```

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 8.22).

**SOURCE:**

$$e \equiv e_{\text{arr}} [e_0, \dots, e_{R-1}]$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}} [e] \equiv \mathbf{T}_{\text{pre}}^{\text{V}} [e_{\text{arr}}] \\ \mathbf{T}_{\text{pre}} [e_0, \dots, e_{R-1}]$$

$$\mathbf{T}_{\text{val}} [e] \equiv \mathbf{T}_{\text{dat}}^{\text{V}} [e_{\text{arr}}] [\text{OFFSET}(e_{\text{arr}}, e_0, \dots, e_{R-1})]$$

where:

The expression  $e_{\text{arr}}$  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 8.13: Translation of array access expression.



### 8.13 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 8.13. The macro *OFFSET* is defined as

$$\begin{aligned} \text{OFFSET}(e_{\text{arr}}, e_0, \dots, e_{R-1}) \equiv & \\ & \mathbf{T}_{\text{bas}}[e_{\text{arr}}] + \text{OFFSET\_DIM}(\mathbf{T}_0^V[e_{\text{arr}}], e_0) \\ & \quad \dots \\ & + \text{OFFSET\_DIM}(\mathbf{T}_{R-1}^V[e_{\text{arr}}], e_{R-1}) \end{aligned}$$

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 subscript (in a sequential dimension).

If  $e_r$  is a distributed index  $i$ , then

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

where *sub* is the local subscript variable for this index (see sections 8.7 and 8.8). Otherwise if  $e_r$  is a shifted index  $i \pm d$ , then

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

where *sub* is the local subscript variable and *shf\_stp* is the shift step for  $i$  (again, see sections 8.7 and 8.8). Otherwise if  $e_r$  is an integer expression (which implies that  $a'_r$  has type *SeqArrayDim*), then

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

The fields *off\_bas* and *off\_stp* are initialized by the constructors for *SeqArrayDim*. They do not exist in the superclass *ArrayDim*.

**SOURCE:**

$$e \equiv e_{\text{arr}} [[\text{subs}_0, \dots, \text{subs}_{R-1}]]$$

**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}] \\ &\quad \mathbf{T}_{\text{pre}}[\text{subs}_0, \dots, \text{subs}_{R-1}] \\ b &= \mathbf{T}_{\text{bas}}[e_{\text{arr}}] \ ; \\ &\quad \text{PROCESS\_SUBSCRIPTS}(0, e_{\text{arr}}, 0) \end{aligned}$$

$$\begin{aligned} \mathbf{T}_{\text{dat}}[e] &\equiv \mathbf{T}_{\text{dat}}[e_{\text{arr}}] \\ \mathbf{T}_{\text{bas}}[e] &\equiv b \\ \mathbf{T}_{\text{grp}}[e] &\equiv \mathbf{T}_{\text{grp}}[e_{\text{arr}}] \\ \mathbf{T}_0[e] &\equiv a_0 \\ &\dots \\ \mathbf{T}_{R-1}[e] &\equiv a_{R-1} \end{aligned}$$

where:

The expression  $e_{\text{arr}}$  is the subscripted array,  
 each term  $\text{subs}_s$  is either an integer, a triplet, or  $\langle \rangle$ ,  
 $b$  and  $a_0, \dots, a_{R-1}$  are names of new temporaries,  
 the macro *PROCESS\_SUBSCRIPTS* is defined in the text.

Figure 8.14: Translation of array section without scalar subscripts in distributed dimensions.

## 8.14 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 two cases: the case where there are no scalar subscripts—integer or distributed index—in *distributed* dimensions (we *will* allow integer subscripts in sequential dimensions), and the general case. The first case is simpler because there is no non-trivial effect on the distribution group, so we will consider it first. The scheme for translating is illustrated in Figure 8.14.

The macro *OFFSET\_SEQUENTIAL* is defined as in section 8.13. The macro *PROCESS\_SUBSCRIPTS* will be defined 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*( $r, e_{\text{arr}}, s$ ) is empty. Otherwise, if  $subs_s$  is an integer expression, and the  $s$ th dimension of  $a$  is sequential, then

$$\begin{aligned} \text{PROCESS\_SUBSCRIPTS}(r, e_{\text{arr}}, s) &\equiv \\ b & += \text{OFFSET\_DIM}(\mathbf{T}_s[e_{\text{arr}}], subs_s) ; \\ \text{PROCESS\_SUBSCRIPTS}(r, e_{\text{arr}}, s + 1) & \end{aligned}$$

where the macro *OFFSET\_DIM* is defined in section 8.13. Otherwise, if  $subs_s$  is the degenerate triplet,  $:$ , then

$$\begin{aligned} \text{PROCESS\_SUBSCRIPTS}(r, e_{\text{arr}}, s) &\equiv \\ a_r & = \mathbf{T}_s[e_{\text{arr}}] ; \\ \text{PROCESS\_SUBSCRIPTS}(r + 1, e_{\text{arr}}, s + 1) & \end{aligned}$$

Otherwise, if  $subs_s$  is the triplet,  $e_{\text{lo}}:e_{\text{hi}}:e_{\text{stp}}$ , and the  $s$ th dimension of  $a$  is distributed, then

$$\begin{aligned} \text{PROCESS\_SUBSCRIPTS}(r, e_{\text{arr}}, s) &\equiv \\ a_r & = a'_s.\text{range.subrng}(e'_{\text{lo}}, e'_{\text{hi}}, e'_{\text{stp}}).\text{arrayDim}(a'_s.\text{stride}) ; \\ \text{PROCESS\_SUBSCRIPTS}(r + 1, e_{\text{arr}}, s + 1) & \end{aligned}$$

where  $a'_s = \mathbf{T}_s^V[e_{\text{arr}}]$ ,  $e'_{\text{lo}} = \mathbf{T}_{\text{val}}[e_{\text{lo}}]$ ,  $e'_{\text{hi}} = \mathbf{T}_{\text{val}}[e_{\text{hi}}]$ , and  $e'_{\text{stp}} = \mathbf{T}_{\text{val}}[e_{\text{stp}}]$ . Otherwise, if  $subs_s$  is the triplet,  $e_{\text{lo}}:e_{\text{hi}}:e_{\text{stp}}$ , and the  $s$ th dimension of  $a$  is sequential, then

$$\begin{aligned} \text{PROCESS\_SUBSCRIPTS}(r, e_{\text{arr}}, s) &\equiv \\ a_r & = \text{new SeqArrayDim}(a'_s.\text{range.subrng}(e'_{\text{lo}}, e'_{\text{hi}}, e'_{\text{stp}}), a'_s.\text{stride}) ; \\ \text{PROCESS\_SUBSCRIPTS}(r + 1, e_{\text{arr}}, s + 1) & \end{aligned}$$

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

$$\begin{aligned} \text{PROCESS\_SUBSCRIPTS}(r, e_{\text{arr}}, s) &\equiv \\ x & = a'_s.\text{range} ; \\ u & = a'_s.\text{stride} ; \\ z & = x.\text{shell}() ; \\ a_{r+1} & = \text{new SeqArrayDim}(z, u) ; \\ a_r & = \text{new ArrayDim}(x.\text{dim}(), u * z.\text{volume}()) ; \\ \text{PROCESS\_SUBSCRIPTS}(r + 2, e_{\text{arr}}, s + 1) & \end{aligned}$$

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

**SOURCE:**

$$e \equiv e_{\text{arr}} [[\text{subs}_0, \dots, \text{subs}_{R-1}]]$$

**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}^V[e_{\text{arr}}] \\ &\quad \mathbf{T}_{\text{pre}}[\text{subs}_0, \dots, \text{subs}_{R-1}] \\ b &= \mathbf{T}_{\text{bas}}[e_{\text{arr}}] ; \\ p &= (\text{Group}) \mathbf{T}_{\text{grp}}[e_{\text{arr}}].\text{clone}() ; \\ &\quad \text{PROCESS\_SUBSCRIPTS}(0, e_{\text{arr}}, 0) \end{aligned}$$

$$\begin{aligned} \mathbf{T}_{\text{dat}}[e] &\equiv p.\text{member}() ? \mathbf{T}_{\text{dat}}^V[e_{\text{arr}}] : \text{null} \\ \mathbf{T}_{\text{bas}}[e] &\equiv b \\ \mathbf{T}_{\text{grp}}[e] &\equiv p \\ \mathbf{T}_0[e] &\equiv a_0 \\ &\quad \dots \\ \mathbf{T}_{R-1}[e] &\equiv a_{R-1} \end{aligned}$$

where:

The expression  $e_{\text{arr}}$  is the subscripted array,  
 each term  $\text{subs}_s$  is either an integer, a distributed index name,  
 a shifted index expression, a triplet, or  $\langle \rangle$ ,  
 $b$ ,  $p$  and  $a_0, \dots, a_{R-1}$  are names of new temporaries, and  
 the macro *PROCESS\_SUBSCRIPTS* is defined in the text.

Figure 8.15: Translation of array section allowing scalar subscripts in distributed dimensions.

### 8.14.1 Scalar subscripts in distributed dimensions

The scheme for translating array sections when scalar subscripts appear in some distributed dimension is illustrated in Figure 8.15.

We add two new cases for the definition of the macro *PROCESS\_SUBSCRIPTS*. If *subs<sub>s</sub>* is the integer *e<sub>glb</sub>*, and the *s*th dimension of *e<sub>arr</sub>* is distributed, then

```

PROCESS_SUBSCRIPTS(r, earr, s) ≡
  x = a's.range ;
  l = x.location(e'glb) ;
  b += l.sub * a's.stride ;
  p.restrict(x.dim(), l.crd) ;
  PROCESS_SUBSCRIPTS(r, earr, s + 1)

```

where *x* and *l* are the names of new temporaries,  $a'_s = \mathbf{T}_s^V[e_{\text{arr}}]$ , and  $e'_{\text{glb}} = \mathbf{T}_{\text{val}}[e_{\text{glb}}]$ . Otherwise, if *subs<sub>s</sub>* is a distributed index *i* or a shifted index *i* ± *d*, then

```

PROCESS_SUBSCRIPTS(r, earr, s) ≡
  b += OFFSET_DIM( $\mathbf{T}_s[e_{\text{arr}}]$ , subss) ;
  p.restrict(x.dim()) ;
  PROCESS_SUBSCRIPTS(r, earr, s + 1)

```

where in this case *x* is the range associated with *i* and the macro *OFFSET\_DIM* is defined in section 8.13.

**SOURCE:**

$$e \equiv e_{\text{obj}} \cdot id$$

**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}^V[e_{\text{obj}}] \\ \mathbf{T}_{\text{dat}}[e] &\equiv e'_{\text{obj}} \cdot \text{TRANS\_ID}(id) \\ \mathbf{T}_{\text{bas}}[e] &\equiv e'_{\text{obj}} \cdot \text{TRANS\_ID\_BAS}(id) \\ \mathbf{T}_{\text{grp}}[e] &\equiv e'_{\text{obj}} \cdot \text{TRANS\_ID\_GRP}(id) \\ \mathbf{T}_0[e] &\equiv e'_{\text{obj}} \cdot \text{TRANS\_ID\_DIM}(id, 0) \\ &\dots \\ \mathbf{T}_{R-1}[e] &\equiv e'_{\text{obj}} \cdot \text{TRANS\_ID\_DIM}(id, R-1) \end{aligned}$$

where:

the expression  $e_{\text{obj}}$  has class type,  
the identifier  $id$  is the name of the field,  
 $R$  is the rank of the field,  
 $e'_{\text{obj}} = \mathbf{T}_{\text{val}}^V[e_{\text{obj}}]$ , and  
the macros  $\text{TRANS\_ID}$ ,  $\text{TRANS\_ID\_BAS}$ ,  $\text{TRANS\_ID\_GRP}$  and  
 $\text{TRANS\_ID\_DIM}$  are defined in the text.

Figure 8.16: Translation of a distributed-array-valued field access.

## 8.15 Translating field accesses

We only need to consider the case where the field is a distributed array. The general scheme is illustrated in Figure 8.16. This scheme is modified in trivial ways if the field is a class variable, or an instance variable of the current object.

The macros *TRANS\_ID*, *TRANS\_ID\_BAS*, *TRANS\_ID\_GRP* and *TRANS\_ID\_DIM* are defined in section 8.4.

**SOURCE:**

$$e \equiv e_{\text{obj}}.id(e_0, \dots, e_{N-1})$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_{\text{obj}}] \\ \mathbf{T}_{\text{pre}}[e_0, \dots, e_{N-1}]$$

$$\mathbf{T}_{\text{val}}[e] \equiv \mathbf{T}_{\text{val}}[e_{\text{obj}}].id(\text{TRANS\_ARGS}(e_0, \dots, e_{N-1}), \text{apg})$$

where:

The expression  $e_{\text{obj}}$  has class type,  
 the identifier  $id$  is the name of the method,  
 each term  $e_r$  is an actual argument in the original program,  
 the macro  $\text{TRANS\_ARGS}$  is defined in the text.

Figure 8.17: Translation of method invocation expression.



## 8.16 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 8.17. 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:

$$\begin{aligned} \text{TRANS\_ARGS}(e_0, \dots, e_{N-1}) \equiv \\ \text{TRANS\_ARG}(e_0), \dots, \text{TRANS\_ARG}(e_{N-1}) \end{aligned}$$

Now, if  $e$  is not a distributed array, then

$$\text{TRANS\_ARG}(e) \equiv e$$

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

$$\begin{aligned} \text{TRANS\_ARG}(e) \equiv \\ \mathbf{T}_{\text{dat}}[e], \mathbf{T}_{\text{bas}}[e], \mathbf{T}_{\text{grp}}[e], \mathbf{T}_0[e], \dots, \mathbf{T}_{R-1}[e] \end{aligned}$$

In other words, each distributed array argument is split into  $3 + R$  arguments.

**SOURCE:**

$$e \equiv e_{\text{obj}}.id(e_0, \dots, e_{N-1})$$

**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}[e_{\text{obj}}] \\ &\quad \mathbf{T}_{\text{pre}}[e_0, \dots, e_{N-1}] \\ v &= \mathbf{T}_{\text{val}}[e_{\text{obj}}].\text{TRANS\_ID}(id)(d, \text{TRANS\_ARGS}(e_0, \dots, e_{N-1}), \text{app}) ; \end{aligned}$$

$$\begin{aligned} \mathbf{T}_{\text{dat}}[e] &\equiv v \\ \mathbf{T}_{\text{bas}}[e] &\equiv d.\text{base} \\ \mathbf{T}_{\text{grp}}[e] &\equiv d.\text{group} \\ \mathbf{T}_0[e] &\equiv \text{TRANS\_DIM}(d, 0) \\ &\dots \\ \mathbf{T}_{R-1}[e] &\equiv \text{TRANS\_DIM}(d, R-1) \end{aligned}$$

where:

The expression  $e_{\text{obj}}$  has class type or is a class,  
the identifier  $id$  is the name of the method,  
each term  $e_r$  is an actual argument in the original program,  
 $v$  is a new temporary,  
 $d$  is a new temporary, referencing an allocated DAD object, and  
 $R$  is the rank of the result, and  
the macros  $\text{TRANS\_ID}$ ,  $\text{TRANS\_ARGS}$ , and  $\text{TRANS\_DIM}$   
are defined in the text.

Figure 8.18: Translation of array-valued method invocation expression.

### 8.16.1 Methods that return distributed arrays

The scheme for translating a distributed-array-valued method invocation is illustrated in Figure 8.18.

The macro *TRANS\_ID* encodes the dimension signature of the returned array and is defined in section 8.4. The macro *TRANS\_ARGS* was defined earlier in this section.

The macro *TRANS\_DIM* is defined as

$$TRANS\_DIM(d, r) \equiv d.dimensions [r]$$

if the *r*th dimension of the result is distributed, or

$$TRANS\_DIM(d, r) \equiv (SeqArrayDim) d.dimensions [r]$$

if the *r*th dimension is sequential.

**SOURCE:**

```
return  $e_{\text{arr}}$  ;
```

**TRANSLATION:**

```

 $\mathbf{T}_{\text{pre}} [e_{\text{arr}}]$ 

 $d.\text{base} = \mathbf{T}_{\text{bas}} [e_{\text{arr}}]$  ;
 $d.\text{group} = \mathbf{T}_{\text{grp}} [e_{\text{arr}}]$  ;
 $d.\text{dimensions} [0] = \mathbf{T}_0 [e_{\text{arr}}]$  ;
...
 $d.\text{dimensions} [R - 1] = \mathbf{T}_{R-1} [e_{\text{arr}}]$  ;

return  $\mathbf{T}_{\text{dat}} [e_{\text{arr}}]$  ;
```

where:

$e_{\text{arr}}$  is the array-valued result expression,  
 $R$  is its rank, and  
 $d$  is the DAD passed as first argument of the translated method.

Figure 8.19: Translation of return statement in array-valued method.

### 8.16.2 Translation of return statement

The scheme for translating a `return` statement in the definition of an array-valued method is illustrated in Figure 8.19.

**SOURCE:**

$$e \equiv \text{new } T(e_0, \dots, e_{N-1})$$

**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}[e_0] \\ &\dots \\ &\mathbf{T}_{\text{pre}}[e_{N-1}] \end{aligned}$$

$$\mathbf{T}_{\text{val}}[e] \equiv \text{new } T(\text{TRANS\_ARGS}(e_0, \dots, e_{N-1}))$$

where:

$T$  is a Java type,  
each term  $e_r$  is an actual argument in the original program,  
the macro  $\text{TRANS\_ARGS}$  is defined in the text.

Figure 8.20: Translation of class instance creation expression.

## 8.17 Translating constructor invocations

The rules for translating constructor invocations follow directly from the rules for method invocations given in section 8.16.

Figure 8.17 illustrates the translation for a class instance creation expression. Explicit constructor invocations (specifying `this` or `super`) do not introduce any new features.

**SOURCE:**

$$e \equiv e_{\text{arr}} / e_{\text{loc}}$$

**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}] \\ &\quad \mathbf{T}_{\text{pre}}[e_{\text{loc}}] \\ p &= \text{RESTRICT\_GROUP}(\mathbf{T}_{\text{grp}}[e_{\text{arr}}], e_{\text{loc}}) \end{aligned}$$

$$\begin{aligned} \mathbf{T}_{\text{dat}}[e] &\equiv p.\text{member}() ? \mathbf{T}_{\text{dat}}[e_{\text{arr}}] : \text{null} \\ \mathbf{T}_{\text{bas}}[e] &\equiv \mathbf{T}_{\text{bas}}[e_{\text{arr}}] \\ \mathbf{T}_{\text{grp}}[e] &\equiv p \\ \mathbf{T}_0[e] &\equiv \mathbf{T}_0[e_{\text{arr}}] \\ &\quad \dots \\ \mathbf{T}_{R-1}[e] &\equiv \mathbf{T}_{R-1}[e_{\text{arr}}] \end{aligned}$$

where:

The expression  $e_{\text{arr}}$  is the array to be restricted,  
the expression  $e_{\text{loc}}$  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 8.21: Translation of array restriction operation.



## 8.18 Translating array restriction

The scheme is illustrated in Figure 8.21.

If  $e_{\text{loc}}$  is a range element of the form  $e_{\text{rng}}[e_{\text{glb}}]$ , the macro *RESTRICT\_GROUP* is defined as

$$\begin{aligned} \text{RESTRICT\_GROUP}(p, e_{\text{loc}}) \equiv & \\ & x = e'_{\text{rng}} ; \\ & l = x.\text{location}(e'_{\text{glb}}); \\ & ((\text{Group})\ p.\text{clone}()).\text{restrict}(x.\text{dim}(), l.\text{crd}) ; \end{aligned}$$

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

$$\begin{aligned} \text{RESTRICT\_GROUP}(p, e_{\text{loc}}) \equiv & \\ & ((\text{Group})\ p.\text{clone}()).\text{restrict}(x.\text{dim}()) ; \end{aligned}$$

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

**SOURCE:**

$$e \equiv e_{\text{grp}} / e_{\text{loc}}$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}} [e] \equiv \begin{array}{l} \mathbf{T}_{\text{pre}} [e_{\text{grp}}] \\ \mathbf{T}_{\text{pre}} [e_{\text{loc}}] \end{array}$$

$$\mathbf{T}_{\text{dat}} [v] \equiv \text{RESTRICT\_GROUP}(\mathbf{T}_{\text{val}} [e_{\text{grp}}], e_{\text{loc}})$$

where:

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

Figure 8.22: Translation of group restriction operation.

## 8.19 Translating group restriction

The scheme is illustrated in Figure 8.22. The macro *RESTRICT\_GROUP* is defined in section 8.18.

**SOURCE:**

$$e \equiv e_{\text{rng}} [e_{\text{lo}} : e_{\text{hi}}]$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}} [e] \equiv \begin{array}{l} \mathbf{T}_{\text{pre}} [e_{\text{rng}}] \\ \mathbf{T}_{\text{pre}} [e_{\text{lo}}, e_{\text{hi}}] \end{array}$$

$$\mathbf{T}_{\text{dat}} [v] \equiv \mathbf{T}_{\text{val}} [e_{\text{rng}}] . \text{subrng} (\mathbf{T}_{\text{val}} [e_{\text{lo}}], \mathbf{T}_{\text{val}} [e_{\text{hi}}])$$

**SOURCE:**

$$e \equiv e_{\text{rng}} [e_{\text{lo}} : e_{\text{hi}} : e_{\text{stp}}]$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}} [e] \equiv \begin{array}{l} \mathbf{T}_{\text{pre}} [e_{\text{rng}}] \\ \mathbf{T}_{\text{pre}} [e_{\text{lo}}, e_{\text{hi}}, e_{\text{stp}}] \end{array}$$

$$\mathbf{T}_{\text{dat}} [v] \equiv \mathbf{T}_{\text{val}} [e_{\text{rng}}] . \text{subrng} (\mathbf{T}_{\text{val}} [e_{\text{lo}}], \mathbf{T}_{\text{val}} [e_{\text{hi}}], \mathbf{T}_{\text{val}} [e_{\text{stp}}])$$

where:

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

Figure 8.23: Translation of subrange expressions.

## **8.20 Translating subrange expressions**

The schemes are illustrated in Figure 8.23.

**SOURCE:**

$$e \equiv e_{\text{bool}} ? e_1 : e_2$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}} [e] \equiv \mathbf{T}_{\text{pre}}^V [e_{\text{bool}}] \\ \mathbf{T}_{\text{pre}} [e_1, e_2]$$

$$\mathbf{T}_{\text{dat}} [e] \equiv e'_{\text{bool}} ? \mathbf{T}_{\text{dat}} [e_1] : \mathbf{T}_{\text{dat}} [e_2]$$

$$\mathbf{T}_{\text{bas}} [e] \equiv e'_{\text{bool}} ? \mathbf{T}_{\text{bas}} [e_1] : \mathbf{T}_{\text{bas}} [e_2]$$

$$\mathbf{T}_{\text{grp}} [e] \equiv e'_{\text{bool}} ? \mathbf{T}_{\text{grp}} [e_1] : \mathbf{T}_{\text{grp}} [e_2]$$

$$\mathbf{T}_0 [e] \equiv e'_{\text{bool}} ? \mathbf{T}_0 [e_1] : \mathbf{T}_0 [e_2]$$

...

$$\mathbf{T}_{R-1} [e] \equiv e'_{\text{bool}} ? \mathbf{T}_{R-1} [e_1] : \mathbf{T}_{R-1} [e_2]$$

where:

$e_{\text{bool}}$  has boolean type, and  
 $e_1$  and  $e_2$  have distributed array type,  
 both have the same rank,  $R$ , and  
 $e'_{\text{bool}} = \mathbf{T}_{\text{val}}^V [e_{\text{bool}}]$ .

Figure 8.24: Translation of conditional operator selecting distributed arrays.

## 8.21 Translating the conditional operator

The general scheme is illustrated in Figure 8.11. The definition of the result type, and the associated conditions for legality of selecting between  $e_1$  and  $e_2$ , follow from this translation together with the normal Java rules applied to the parts of the arrays.

**SOURCE:**

$$e \equiv e_{\text{arr}}.\text{dat}()$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}]$$

$$\mathbf{T}_{\text{val}}[e] \equiv \mathbf{T}_{\text{dat}}[e_{\text{arr}}]$$

**SOURCE:**

$$e \equiv e_{\text{arr}}.\text{bas}()$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}]$$

$$\mathbf{T}_{\text{val}}[e] \equiv \mathbf{T}_{\text{bas}}[e_{\text{arr}}]$$

**SOURCE:**

$$e \equiv e_{\text{arr}}.\text{grp}()$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}]$$

$$\mathbf{T}_{\text{val}}[e] \equiv \mathbf{T}_{\text{grp}}[e_{\text{arr}}]$$

where:

The expression  $e_{\text{arr}}$  has distributed array type.

Figure 8.25: Translation of distributed array inquiries.



**SOURCE:**

$$e \equiv e_{\text{arr}}.\text{rng}(r)$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}]$$

$$\mathbf{T}_{\text{val}}[e] \equiv \mathbf{T}_r[e_{\text{arr}}].\text{range}$$

**SOURCE:**

$$e \equiv e_{\text{arr}}.\text{str}(r)$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}]$$

$$\mathbf{T}_{\text{val}}[e] \equiv \mathbf{T}_r[e_{\text{arr}}].\text{stride}$$

where:

The expression  $e_{\text{arr}}$  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  $e_{\text{arr}}$ .

Figure 8.26: Translation of distributed array inquiries, continued.

**SOURCE:**

$$e \equiv (T \ [\![attr_0, \dots, attr_{R-1}]\!] ) e_{\text{arr}}$$

**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}]$$

$$\mathbf{T}_{\text{dat}}[e] \equiv (T \ [\ ] ) \mathbf{T}_{\text{dat}}[e_{\text{arr}}]$$

$$\mathbf{T}_{\text{bas}}[e] \equiv \mathbf{T}_{\text{bas}}[e_{\text{arr}}]$$

$$\mathbf{T}_{\text{grp}}[e] \equiv \mathbf{T}_{\text{grp}}[e_{\text{arr}}]$$

$$\mathbf{T}_0[e] \equiv \text{CAST\_DIMENSION}(attr_0, \mathbf{T}_0[e_{\text{arr}}])$$

...

$$\mathbf{T}_{R-1}[e] \equiv \text{CAST\_DIMENSION}(attr_{R-1}, \mathbf{T}_{R-1}[e_{\text{arr}}])$$

where:

$T$  is a Java type,  
 each term  $attr_r$  is either empty or a single asterisk, \*,  
 $e_{\text{arr}}$  is an array expression in the source program, rank  $R$ , and  
 the macro  $\text{CAST\_DIMENSION}$  is defined in the text.

Figure 8.27: Translation of cast of array expression.

## 8.22 Translating casts

The scheme for translating a cast of a distributed-array valued expression is illustrated in Figure 8.27. The macro *CAST\_DIMENSION* is defined as follows:

$$CAST\_DIMENSION(attr_r, e'_r) \equiv (\text{ArrayDim}) e'_r$$

if the term  $attr_r$  is empty, or

$$CAST\_DIMENSION(attr_r, e'_r) \equiv (\text{SeqArrayDim}) e'_r$$

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

**SOURCE:**

$$e \equiv e_{\text{arr}} \text{ instanceof } T \text{ } [[\text{attr}_0, \dots, \text{attr}_{R-1}]]$$
**TRANSLATION:**

$$\mathbf{T}_{\text{pre}}[e] \equiv \mathbf{T}_{\text{pre}}[e_{\text{arr}}]$$

$$\begin{aligned} \mathbf{T}_{\text{val}}[e] \equiv & (\mathbf{T}_{\text{dat}}[e_{\text{arr}}] \text{ instanceof } T \text{ } []) \ \&\& \\ & (\mathbf{T}_0[e_{\text{arr}}] \text{ instanceof } \text{DIMENSION\_TYPE}(\text{attr}_0)) \ \&\& \\ & \dots \\ & (\mathbf{T}_{R-1}[e_{\text{arr}}] \text{ instanceof } \text{DIMENSION\_TYPE}(\text{attr}_{R-1})) \end{aligned}$$

where:

$e_{\text{arr}}$  is an array expression in the source program, rank  $R$ ,  
 $T$  is a Java type,  
 each term  $\text{attr}_r$  is either empty or a single asterisk, \*, and  
 the macro  $\text{DIMENSION\_TYPE}$  is defined in the text.

Figure 8.28: Translation `instanceof` applied to array expression.

## 8.23 Translating instanceof

The scheme for translating an `instanceof` test applied to a distributed-array valued expression is illustrated in Figure 8.28. The macro `DIMENSION_TYPE` is in section 8.3.

**SOURCE:**

$$e \equiv e_1 == e_2$$

**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}[e_1] \\ &\quad \mathbf{T}_{\text{pre}}[e_2] \\ \\ \mathbf{T}_{\text{val}}[e] &\equiv (\mathbf{T}_{\text{dat}}[e_1] == \mathbf{T}_{\text{dat}}[e_2]) \ \&\& \\ &\quad (\mathbf{T}_{\text{bas}}[e_1] == \mathbf{T}_{\text{bas}}[e_2]) \ \&\& \\ &\quad (\mathbf{T}_0[e_1] == \mathbf{T}_0[e_2]) \ \&\& \\ &\quad \dots \\ &\quad (\mathbf{T}_{R-1}[e_1] == \mathbf{T}_{R-1}[e_2]) \end{aligned}$$

**SOURCE:**

$$e_1 != e_2$$

**TRANSLATION:**

$$\begin{aligned} \mathbf{T}_{\text{pre}}[e] &\equiv \mathbf{T}_{\text{pre}}[e_1] \\ &\quad \mathbf{T}_{\text{pre}}[e_2] \\ \\ \mathbf{T}_{\text{val}}[e] &\equiv (\mathbf{T}_{\text{dat}}[e_1] != \mathbf{T}_{\text{dat}}[e_2]) \ || \\ &\quad (\mathbf{T}_{\text{bas}}[e_1] != \mathbf{T}_{\text{bas}}[e_2]) \ || \\ &\quad (\mathbf{T}_0[e_1] != \mathbf{T}_0[e_2]) \ || \\ &\quad \dots \\ &\quad (\mathbf{T}_{R-1}[e_1] != \mathbf{T}_{R-1}[e_2]) \end{aligned}$$

where:

$e_1$  and  $e_2$  have distributed array type, and both have the same rank,  $R$ .

Figure 8.29: Translation of reference equality tests, applied to array expressions.

## 8.24 Translating reference equality

The scheme for translating the reference equality operators `==` and `!=`, applied to distributed arrays is illustrated in Figure 8.29.

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 8.14, 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.]*