Parallel Programming in HPJava

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

Pervasive Technology Labs, Indiana University
501 N. Morton St, Suite 224
Bloomington, IN 47404-3730

April 1, 2003
Contents

1 Introduction ......................................................... 7
  1.1 Getting Started ................................................ 8
    1.1.1 Installing hpjdk ....................................... 9
    1.1.2 Compiling and Running an HPJava program ............ 10
  1.2 Multidimensional Arrays ...................................... 12
    1.2.1 Multidimensional Arrays and Multidimensional Arrays 12
    1.2.2 Syntax .................................................. 13
    1.2.3 Some Semantic Issues .................................. 16
    1.2.4 Sections ................................................ 17
    1.2.5 Checking Subscripts .................................... 18
  1.3 Managing Expectations ........................................ 19

2 Processes and Distributed Arrays ................................. 21
  2.1 HPspmd Classes ............................................... 22
  2.2 Process Grids .................................................. 24
  2.3 Distributed Arrays ............................................ 28
  2.4 Parallel Programming ......................................... 30
  2.5 Locations ..................................................... 33
  2.6 A Complete Example ........................................... 35
  2.7 Multi-process and Multithreaded Execution Models ....... 37

3 More on Mapping Arrays ........................................... 41
  3.1 Other Distribution Formats .................................... 41
  3.2 Ghost Regions ................................................. 44
  3.3 Collapsed Distributions and Sequential Dimensions ....... 52
  3.4 Distribution Groups and Replication ......................... 53

4 Distributed Array Sections ....................................... 61
  4.1 Two-dimensional Fourier transform ......................... 63
  4.2 Cholesky decomposition ....................................... 63
  4.3 Matrix multiplication with reduced memory ................ 65
  4.4 Subranges ..................................................... 67
  4.5 Restricted Groups ............................................ 68
  4.6 Mapping of distributed array sections ....................... 69

3
4 7  Rank-0 Distributed Arrays ............................................. 70
4 8  Distributed Array Restriction ........................................ 71
5  Some Rules and Definitions .............................................. 73
  5.1  Rules for distributed control constructs ......................... 73
  5.2  Rules for distributed array constructors ......................... 74
  5.3  Rules for access to distributed array elements .................. 75
       5.3.1  Changes for ghost regions .................................. 77
       5.3.2  A final rule for array element access ...................... 77
  5.4  A recommendation for updating variables ......................... 79
6  A Distributed Array Communication Library ......................... 81
  6.1  Regular collective communications .................................. 82
  6.2  Reductions ............................................................ 86
  6.3  Irregular collective communications ................................ 88
  6.4  I/O ................................................................. 90
  6.5  Schedules ............................................................ 93
7  Low level SPMD programming ............................................. 97
  7.1  An Example .......................................................... 97
  7.2  Dimension Splitting ................................................ 103
  7.3  Block Parameters .................................................. 108
       7.3.1  Ghost regions and dimension splitting ...................... 112
       7.3.2  Local blocks of subranges .................................. 112
  7.4  Reduction to Java arrays ........................................... 115
  7.5  Local arrays ....................................................... 119
  7.6  An extended example: prefix computation ......................... 120
       7.6.1  Optimization for block distribution formats .............. 120
       7.6.2  Cyclic distribution formats ................................ 124
       7.6.3  Optimization for “general” distribution formats ........ 124
  7.7  Non-local blocks .................................................. 127
       7.7.1  The cards( ) method ....................................... 127
A  Translation scheme ...................................................... 131
  A.1  Preliminaries ....................................................... 131
       A.1.1  On multiaarray types ...................................... 131
       A.1.2  HPspmd classes .............................................. 133
  A.2  Static Semantic Checking ........................................... 136
       A.2.1  Subscripting .................................................. 136
  A.3  Pre-translation .................................................... 138
       A.3.1  Restricted Form ............................................. 138
       A.3.2  Expression simplification .................................. 140
       A.3.3  Applying the simplify algorithm .......................... 154
  A.4  Basic translation .................................................. 168
       A.4.1  Translation functions and schemas ......................... 168
       A.4.2  Translating variable declarations ......................... 171
Chapter 1

Introduction

HPJava is designed as a language for parallel programming. It extends the standard Java language with syntax for manipulating a new kind of parallel data structure—the distributed array. The extensions evolved out of work on Fortran 90 and High Performance Fortran (HPF), but in fact the parallel programming model of HPJava is different to HPF. We will start to discuss the parallel features of HPJava in Chapter 2. This introductory chapter will mainly show you how to get started with HPJava. It describes how to compile and run simple HPJava programs. It also describes the sequential part of HPJava language. HPJava only introduces one language extension for sequential programming, namely the multiarray.

Software. The HPJava software is available for free download from our Web site at www.hpjava.org. Full sources are available under a very liberal licensing agreement (at the moment no licensing agreement is required).

There are two parts to the software. The HPJava development kit, hpjdk contains the HPJava compiler and an implementation of the high-level communication library, Addb, used throughout this report. The only prerequisite for installing hpjdk is a standard Java development platform, like the one freely available for several operating systems from Sun Microsystems. Once hpjdk is installed (which is very straightforward because it is a pure Java package) you can start compiling HPJava programs. You can immediately run sequential HPJava programs using the standard java command. Parallel HPJava programs can also be run with the java command, provided they follow the multithreaded model described in the next chapter.

To distribute parallel HPJava programs across networks of host computers, or run them on supported distributed-memory parallel computers, you also need to install a second HPJava package—mpiJava. A prerequisite for installing mpiJava is the availability of an MPI installation on your system. MPI is a standard interface for interprocess communication on parallel computers (www.mpi-forum.org). MPI implementations are freely available for many platforms of interest. Section 2.7 describes how to use mpiJava to run HPJava
programs on multiple computers.

Installation of MPI and mpiJava is harder than installation of hpidk alone. Unless you have previous experience with mpiJava (the mpiJava package incorporates a standard Java interface to MPI, which some people have been using for parallel computing independently of the HPJava language extensions) you may want to familiarize yourself with HPJava using the multithreaded model first, before attempting to install mpiJava and thus run distributed memory parallel programs. Programs that follow the patterns described in this report can be developed and debugged using the multithreaded model. In principle turning such programs into distributed memory programs then only involves adding the mpiJava classes to your CLASSPATH environment variable. In reality there are quite likely to be unexpected issues, including performance-tuning issues, complicating the transition to a distributed memory program. So we certainly wouldn’t encourage you to do extensive development with hpidk before attempting this step!

**Programming examples.** This report contains a large number of programming examples. Larger examples appear in figures, and many smaller code fragments appear in the running text. All non-trivial examples here have been checked using the hpidk 1.0 compiler. Nearly all examples are available as working source code in the hpidk release package, under the directory hpidk/examples/PHPJ/.

### 1.1 Getting Started

We assume that you are somewhat familiar with the Java language, and have a Java programming environment installed on your computer. We recommend an environment compatible with Sun J2SE 1.4 (java.sun.com), although it should be possible to get by with 1.3 compatibility.

You will need to be able to create and edit a Java source program using a standard text editor of your choice, to compile it with the command `javac`, and to run it with the command `java`. Currently there is no GUI-based Integrated Development Environment for HPJava—the tools are command-based. This kind of environment will be familiar to most developers who work with Linux or UNIX operating systems. hpidk can be used directly under Microsoft Windows in a command prompt window. However if you are working under Windows you may well want to install a UNIX-like command shell such as Cygwin (www.cygwin.com) before proceeding. Certainly if you are planning to rebuild parts of hpidk from source, Cygwin or equivalent is likely to be very valuable.

To install HPJava, visit our Web site (www.hpjava.org) and download the archive file hpidk-1.0.zip or the archive file hpidk-1.0.tar.gz. The two archives have the same content; choose one or the other depending on what extraction tools you have available, or possibly on bandwidth considerations—the .tar.gz version is typically much smaller.
1.1. Getting Started

1.1.1 Installing hpjdk

For Linux or UNIX. Before trying to install hpjdk, ensure your environment is set up so that the `javac` and `java` commands are working correctly from the shell prompt.

Download `hpjdk-1.0.tar.gz`. In a suitable parent directory, execute a command such as

```
tar xvfz hpjdk-1.0.tar.gz
```
to unpack the archive. For illustration, we assume you run this command in the directory `/home/myid/`. If successful, this creates a directory `/home/myid/hpjdk/`.

Now you must set three environment variables correctly. First you need to create an environment variable `HPJAVA_HOME`, pointing to the directory under which hpjdk is installed. Then you need to add the directories `hpjdk/classes/` and `hpjdk/classes/multithreaded/` to your CLASSPATH environment variable. Finally, you should add the directory `hpjdk/bin/` to your PATH environment variable. If you are using the bash shell, one way to set these variables is to add the following lines to the file `.bashrc` in your home directory:

```
export HPJAVA_HOME=/home/myid/hpjdk
export CLASSPATH=.:$HPJAVA_HOME/classes:$HPJAVA_HOME/classes/multithreaded
PATH=$HPJAVA_HOME/bin:$PATH
```
If you are using the csh shell, you could add the following lines to the file `.cshrc` in your home directory:

```
setenv HPJAVA_HOME /home/myid/hpjdk
setenv CLASSPATH .:$HPJAVA_HOME/classes:$HPJAVA_HOME/classes/multithreaded
setenv PATH $HPJAVA_HOME/bin:$PATH
```
After making these changes, start a new shell.

For Windows command prompt. Before trying to install hpjdk, ensure your environment is set up so that the `javac` and `java` commands are working correctly from the command prompt.

Download `hpjdk-1.0.zip`. Using WinZip or an equivalent program, extract all files in this archive. For illustration, we assume you just extract to `c:\`. Make sure you have selected the option that preserves folder names. If successful, this will create a folder `c:\hpjdk`.

Now you must set three environment variables correctly. First you need to create an environment variable `HPJAVA_HOME`, pointing to the folder under which hpjdk is installed. Secondly, you need to add the folders `hpjdk\classes\` and `hpjdk\classes\multithreaded\` to your CLASSPATH environment variable. Finally, you should add the folder `hpjdk\bin\` to your PATH environment variable. One way to set these variables is to add the following lines to the file `c:\AUTOEXEC.BAT`:

```
set HPJAVA_HOME=c:\hpjdk
set CLASSPATH=.;%HPJAVA_HOME%\classes;%HPJAVA_HOME%\classes\multithreaded
set PATH=%HPJAVA_HOME%\bin;%PATH%
```
then reboot your computer (or log out and log in again, on modern versions of Windows). Alternatively you can use the Windows system menu to define these variables.

**For Cygwin.** This is similar to Linux or UNIX. The main difference is that typically one is using Java development software installed for Windows itself (not Cygwin). This means that the CLASSPATH value—which is interpreted by the Java platform—should identify directories and files through Windows-style paths, instead of the equivalent UNIX-style path.

Before trying to install hpjdk, ensure your environment is set up so that the javac and java commands are working correctly from the Cygwin prompt.

Download hpjdk-1.0.tar.gz. In a suitable parent directory, execute a command such as

```
tar xvfz hpjdk-1.0.tar.gz
```

to unpack the archive. For illustration, we assume you run this command in the directory `/home/myid/`. On success, it creates a directory `/home/myid/hpjdk/`.

Now you must set three environment variables correctly. First you need to create an environment variable `HPJAVA_HOME`, pointing to the directory under which hpjdk is installed. Then you need to add the directories `hpjdk/classes/` and `hpjdk/classes/multithreaded/` to your `CLASSPATH` environment variable. Finally, you should add the directory `hpjdk/bin/` to your `PATH` environment variable. For reasons discussed above, paths in `HPJAVA_HOME` and `CLASSPATH` must be translated to the equivalent Windows-style paths. For illustration we assume that Cygwin is installed in the folder `c:\cygwin\`. Then you can set the required variables by adding the following lines to the file `.bashrc` in your home directory:

```
export HPJAVA_HOME=c:/cygwin/home/myid/hpjdk
export CLASSPATH=":;hpjdk/classes;hpJAVA_HOME/classes/multithreaded"
export PATH=hpjdk/bin:$PATH
```

As advertised, the paths in `HPJAVA_HOME` and `CLASSPATH` are basically in Windows format. We took advantage of the leniency of typical Java systems, which tolerate either forward slash or backslash (using backslash would be more tedious, because backslash is the escape character of the UNIX shell). The value of `CLASSPATH` is quoted, to make sure that the Windows-style semicolon path separator is not interpreted as the shell command terminator. Notice that the `PATH` environment variable is not subject to these complications, because it is processed by the Cygwin shell itself, not by any Windows-based software. But for this reason we can't reuse the `HPJAVA_HOME` value in setting `PATH` here—it is in the wrong style.

After making these edits to `.bashrc`, start a new Cygwin shell.

### 1.1.2 Compiling and Running an HPJava program

We can now compile and run an HPJava program.
1.1. GETTING STARTED

As a first example, go to the subdirectory hpjdk/examples/fft2d/ (or the nested folder hpjdk/examples/fft2d\, under Windows). This contains a fairly sophisticated little program that combines a parallel algorithm for 2-dimensional Fourier Transform (discussed in section 4.1) with a simple Java Swing graphical user interface.

The main program is in the file Wolf.hpj. Compile the whole program by issuing the command

```
hpjavac Wolf.hpj
```

The main program depends on a couple of other source files in the same directory (or folder), but you don’t need to compile these explicitly. Like javac, the hpjavac command will detect these dependencies and automatically compile the other files that are needed.

You can now run the program by issuing the command

```
java -Dhpjava.numprocs=4 Wolf
```

Because we are running under the multithreaded model (this was determined by our setting for CLASSPATH) this command will run Wolf as a “parallel” program in 4 threads of a single Java Virtual Machine\(^1\).

A small Swing window should appear. Click to load the input file. A new window containing a picture of a wolf should appear. You can experiment with the image, exploring the effect of deleting different numbers of modes in Fourier space. The windows are illustrated in Figure 1.1.

In general running the hpjavac command generates one or more standard Java class files that can be executed by a standard Java Virtual Machine, or, as we will see later, by more than one JVM. (Section 2.7 describes how to run HPJava code on multiple JVMs, and thus on multiple computers.)

The source files passed as arguments to the hpjavac are normally HPJava programs with file extension “.hpj” (they can also be ordinary Java source files with extension “.java”).

The hpjavac command is actually implemented as a preprocessor. Every HPJava source program is converted to an intermediate Java file. These intermediate files are then fed to the standard javac compiler. If you look in the fft2d/ directory after successfully running the hpjavac command you will find several intermediate Java files. You can study the content of these files if you are interested. They typically contain sections of recognizable Java code from the source file, interspersed with some very long lines produced by translating the extended syntax. The preprocessor works hard to preserve the line numbering of the input file, to make debugging easier. Generally speaking it is a bad idea to manually edit these automatically generated files.

\(^{1}\)In general, when a parallel HPJava program is run under the multithreaded model, the number of threads is determined by the value of the Java system property `hpjava.numprocs`. The default (if no `-Dhpjava.numprocs` option is given) is to run the program in a single thread.
1.2 Multiarrays

HPJava was designed for parallel programming, but not every HPJava program has to be a parallel program. The language also has a useful sequential subset. This consists of all of standard Java plus some additional syntax for a special kind of array, called a multiarray. Because writing sequential programs that use multiarrays is likely to be a lot easier than writing parallel HPJava programs, and because it is even possible that some users might want to use HPJava for its multiarray syntax alone (without bothering with the parallel features) we will describe these features in this introductory chapter.

1.2.1 Multidimensional Arrays and Multiarrays

Standard Java provides multidimensional arrays. These multidimensional arrays are implemented as arrays of arrays. Amongst other things, this means they can be “ragged”—not all rows of a two-dimensional array, for example, need be the same length. Even if an array is initially created as rectangular, later assignments to the elements of the outer array can change this. The shape of a
1.2. MULTIARRAYS

Java multidimensional array is not immutable. Here are a couple of examples:

```java
int [][] t = new int [N] [] ;
for(int i = 0 ; i < N ; i++)
    t [i] = new int [i + 1] ;
// 't' is not rectangular (it is triangular).

float [][] s = new float [N] [N] ;
// 's' is rectangular here (it is square).

s [0][1] = new float [2 * N] ;
// 's' no longer rectangular (first row twice as long as others).
```

This flexibility might be exactly what is needed in certain contexts. But in other contexts—if we only ever needed a rectangular array—it is an overhead. It makes the program somewhat harder to analyse, both by the programmer and by the optimizer embedded in a compiler or virtual machine. Also it makes it hard to sensibly define the general kind of regular sections that Fortran programmers are accustomed to using in their programs.

Various people have pointed out this shortcoming of Java, including members of the Java Grande Numerics Working Group (math.nist.gov/javnumerics), who made a series of proposals to improve Java in this respect—both through class libraries and syntax extensions. The general idea is to introduce a new kind of multidimensional array, different from the existing multidimensional arrays of Java. The working group called the new arrays multiarrays. HPJava follows their terminology, and our syntax also closely parallels many of their recent proposals.

Another interesting development in this connection has been the emergence of Microsoft’s competitor language to Java, C#. Along with several other extensions relative to Java, Microsoft have added a kind of multiarray. One defect of C# multiarrays, however, is that they don’t currently support regular sections.

### 1.2.2 Syntax

The HPJava syntax for multiarrays uses double brackets to distinguish type signatures from standard Java arrays (which of course use single brackets). Here is a fragment that declares and creates a multiarray:

```java
float [*,*] r = new float [N, N] ;
// 'r' is rectangular here (and will always be).
```

The rank, or dimensionality, is determined by the number of asterisks that appear in the type signature, or by the number of comma-separated integer expressions that appear in the multiarray creation expression. These expressions define the extents, or number of elements, of the multiarray in each dimension.
The syntax is a little bit verbose compared with standard Java, but we don’t see this as a wholly bad thing: one can argue that there is some kind of virtue in having non-standard features stand out slightly from the rest of the code.

Here is a simple, sequential matrix multiplication written in HPJava, with the input and output matrices implemented as multiarrays:

```java
public static void matmul(double [*,*] c, double [*,*] a, double [*,*] b) {
    int M = c.rng(0).size();
    int N = c.rng(1).size();
    int L = a.rng(1).size();

    for(int i = 0 ; i < M ; i++)
        for(int j = 0 ; j < N ; j++) {
            c [i, j] = 0;
            for(int k = 0 ; k < L ; k++)
                c [i, j] += a [i, k] * b [k, j];
        }
}
```

There are a few things to notice here:

The first is that we access elements of a multiarray using a syntax very similar to the syntax for accessing an array element in ordinary Java. If the multiarray has rank greater than one\(^2\), then the subscripts are separated by commas. Double brackets aren’t needed here. They would soon get tiresome, and they are unnecessary because ordinary type checking can always determine (e.g. from the declaration of the array) whether or not the thing being subscripted is in fact a multiarray.

Another thing to notice is that of course multiarrays can be passed as arguments to methods. They are effectively passed by reference, exactly as for ordinary Java arrays. The formal parameter list must declare the rank of the multiarray expected. Methods can also return multiarrays as their result values.

Finally, notice the form of the inquiry methods that allow one to determine the shape of a given multiarray. Instead of providing a `length` field, as for Java arrays, a multiarray supports an inquiry operation `rng()` that looks syntactically like a method. The argument must be a constant expression in the range 0 to R - 1, where R is the rank of the multiarray. This `rng()` operation doesn’t return the integer extent of the array directly (as one might have expected). For reasons connected with the parallel origins of HPJava, `rng()` returns an object of type `hpJava.lang.Range`. One must apply the `size()` method to this object to get the extent itself\(^3\).

\(^2\) This is not necessarily the case. As we will see shortly, a one-dimensional multiarray might be useful in an algorithm that worked on regular sections of a one-dimensional vector.

\(^3\) A later version of HPJava may add an operation for directly obtaining the extent of the multiarray, and lift the restriction that the argument of these inquiries is a constant expression.
import java.text.DecimalFormat;

public class EgSequentialMatmul {

    public static void main(String[] args) {

        int N = 8;

        double [[*, *]] a = new double [[N, N]],
                     b = new double [[N, N]],
                     c = new double [[N, N]];

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

        matmul(c, a, b);

        DecimalFormat form = new DecimalFormat("##0.0");

        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                System.out.print(" \[ \) + form.format(c[i, j]) + " \]
                System.out.println("\n");

    }
}

public static void matmul(double [[*,*]] c,
                            double [[*,*]] a, double [[*,*]] b) {

    int M = c.rng(0).size();
    int N = c.rng(1).size();
    int L = a.rng(1).size();

    for(int i = 0; i < M; i++)
        for(int j = 0; j < N; j++) {
            c[i, j] = 0;
            for(int k = 0; k < L; k++)
                c[i, j] += a[i, k] * b[k, j];
        }
}

Figure 1.2: A complete, sequential HPJava program.
Figure 1.2 gives a complete HPJava program including the matmul method. The source is available in the hpjdk release as the file hpjdk/examples/PPHPJ/-EgSequentialMatmul.hpj. To compile the source one would execute the command:

```
hpjavac EgSequentialMatmul.hpj
```

To execute the program one would then simply run:

```
java EgSequentialMatmul
```

Because this is a sequential program, there is no need to define the hpjava.numprocs property. You should see the output:

```
0.0 140.0 280.0 420.0 560.0 700.0 840.0 980.0
0.0 168.0 336.0 504.0 672.0 840.0 1008.0 1176.0
0.0 196.0 392.0 588.0 784.0 980.0 1176.0 1372.0
0.0 224.0 448.0 672.0 896.0 1120.0 1344.0 1568.0
0.0 252.0 504.0 756.0 1008.0 1260.0 1512.0 1764.0
0.0 280.0 560.0 840.0 1120.0 1400.0 1680.0 1960.0
0.0 308.0 616.0 924.0 1232.0 1540.0 1848.0 2156.0
0.0 336.0 672.0 1008.0 1344.0 1680.0 2016.0 2352.0
```

### 1.2.3 Some Semantic Issues

Multidarrays can have any rank greater than or equal to zero. Their elements can have any standard Java type including primitive types, class types, and Java array types. They cannot, however, have multidarray type: multidarrays of multidarrays are specifically disallowed.

For multidarrays with elements of non-primitive type, conversions between multidarrays are allowed, completely analogous to the corresponding conversions between Java arrays. For example, a method with declaration:

```
void foo(Object [[*,*]] a);
```

accepts as argument any rank-two multidarray whose elements have Java reference type. So this would be legal:

```
foo [[*,*]] b;
```

where `foo` is the name of some class (or interface). These conversions exist only between multidarrays of the same rank. No conversion between a multidarray of rank \( R \) and a multidarray of different rank is possible.

In many respects multidarray types behave like Java reference types. A variable of multidarray type always holds a reference to a multidarray (or a null reference). In this example:

```
float [[*,*]] r = new float [[M, N]];
```

```
float [[*,*]] q = r;
```
1.2. MULTIARRAYS

q and r become two references to the same multiarray. Of course this is similar to the situation with Java arrays. Garbage collection for multiarrays operates on similar lines to the standard Java case.

Technically however, a multiarray reference is not equivalent to a Java reference. Multiarray references and Java references are not interconvertible. If the method foo had been declared instead as

```java
void foo(Object a) ;
```

then it would not be allowed to pass a multiarray as its argument. One consequence is that a reference to a multiarray cannot directly be stored in an instance of a Java container class like java.util.ArrayList or java.util.HashMap, because these classes accept entries of type Object.

There is another related limitation in this mold: one cannot declare standard Java arrays with elements of multiarray type.

These limitations (which exist to simplify the translation scheme) may be slightly inconvenient, but they are not a serious obstacle in practice. One can declare classes with fields of multiarray type. So the user can create a wrapper class for any particular multiarray type. A wrapper object, containing a multiarray, can then be stored in an instance of a container class or a Java array.

To summarize part of this discussion: multiarrays of Java arrays are allowed; Java arrays of multiarrays are not allowed (while multiarrays containing objects as elements, and objects containing multiarrays as fields, are both allowed).

1.2.4 Sections

One of the most interesting features of multiarrays is the possibility of forming a section of a multiarray. This is an idea that will be familiar to Fortran programmers. A section (of some parent multiarray) is a multiarray in its own right whose elements are a subset of the elements of the parent. In other words it is a subarray.

A section expression is a generalized subscripting operation, in which the subscripts may select some range of the allowed indexes of the parent multiarray. A range of indexes is represented by a term call a triplet. In HPJava a triplet has one of three forms:

```java
 l : u
 l : u : s
```

Here the symbols l, u and s stand for integer expressions. Any of the three kinds of triplet can appear as a subscript in a multiarray section expression. The first form, a single colon, means “all allowed index values” for the parent array. The second form means “all index values between l and u, inclusive” (the set is empty if u is less than l). In the third form s is a step value. If s is positive, the index range includes all values l, l + s, l + 2s, ... , less than or
equal to \( u \). If \( s \) is negative, the index range includes all values \( l, l + s, l + 2s, \ldots \), greater than or equal to \( u \).

The subscripts of a multiarray section expression can be any mixture of triplet subscripts and scalar subscripts\(^4\). A scalar subscript is in an ordinary integer expression, just like the subscript in an array element reference. Double brackets rear their head again, mainly to make the distinction between sections and element references very clear. Here are some examples—we assume \( a \) is a rank-2 multiarray:

\[
\begin{align*}
a & \quad [[0, :]] \\
a & \quad [[i, 0]] \\
a & \quad [[0:3, :]] \\
a & \quad [[0, 3]]
\end{align*}
\]

The first section evaluates to a rank-1 multiarray, containing the first row of \( a \). The second evaluates to a rank-1 multiarray containing the second column of \( a \). The third evaluates to a rank-2 multiarray with extent 4 in the first dimension; it contains the first 4 rows of \( a \). The last section evaluates to a rank-0 multiarray containing the single element \( a[0, 3] \). Rank-0 multiarrays are slightly exotic animals with a few interesting uses, but we won’t discuss them any more here.

A section is a normal expression in the HPJava language, and it can appear anywhere in an HPJava program that any other kind of multiarray-valued expression can appear. It is particularly common to see multiarray sections appearing as argument to methods. So, for example, this statement:

\[
\text{matmul}(c \quad [[i : i + B - 1, j : j + B - 1]], a, b); 
\]

presumably assigns the matrix product of \( a \) and \( b \) to \( c \) by \( B \) block of the elements of \( c \), starting at position \( i, j \).

The elements of an array section are variables; they are aliases (alternative names) for the elements of the parent array...if you modify an element of the section, then the corresponding element of the parent array is also changed. Note, however, that the array section expression is not itself a variable; an array section expression cannot appear on the left-hand-side of an assignment. So this:

\[
a \quad [[i, :]] = b; \quad // \text{ERROR}
\]

is illegal (this is different to Fortran).

1.2.5 Checking Subscripts

If the extent of a multiarray in one of its dimensions is \( N \), then any subscript \( i \) in that dimension should be in the range \( 0, \ldots, N - 1 \). It would be reasonable to

\(^4\)Unlike in Fortran, vector subscripts are not allowed in HPJava.
expect that if a subscript in any dimension is out of range then some exception—most likely a `java.lang.IndexOutOfBoundsException`—would automatically be thrown.

It turns out that this expectation is wrong. One of the goals of HPJava is performance, and the luxury of this level of checking would come at a cost in performance. What actually happens is that any element of a multiarray has a corresponding linear index in a notional underlying, flattened, 1-dimensional array. The formula for the linear index can be worked out easily enough given that the last index is “most rapidly varying”—we won’t bother to reproduce the math here. A `java.lang.IndexOutOfBoundsException` exception is thrown only if the linear index corresponding to the set of subscripts specified in the program is outside the allowed set of linear indexes for the whole multiarray (or parent multiarray, in the case of a section). If a subscript is out of range for its dimension, but the associated linear index is not out of range for the whole multiarray, the program silently accesses an unexpected element of the (parent) multiarray.

This is a compromise between ease of debugging and performance. Perhaps a future HPJava compiler will have an optimizer clever enough to make the more transparent form of bounds checking feasible, but we are not there quite yet.

### 1.3 Managing Expectations

In this chapter we have tried to show that it is quite simple to use HPJava in simple ways. In this section—our last before discussing parallel features—we feel we should issue a couple of disclaimers.

We will be happy if it turns out you find all of HPJava as easy as this introduction, but probably this won’t be the case. Parallel programming is fairly hard. HPJava was designed largely to benefit experienced scientific programmers, who are already aware of the kind of difficult issues that arise (just installing the software to run on distributed memory computers can be frustrating work, compared with installing hpjdk).

The style of this report is to present many elementary examples to illustrate principles of the language. Whether the techniques presented are actually effective in realistic situations depends on many factors. Because parallel programming introduces communication into programs, and because the speed of communication (at least in terms of its latency) lags far behind the speed of other kinds of basic operation on modern computers, it is too easy to “parallelize” a program and find it runs orders of magnitude more slowly than the program you started with. This doesn’t mean that parallel processing with HPJava “doesn’t work” (we hope). But it does mean you have to use it in the right way, on the right kinds of problem. Probably only experience and experimentation can help you guess what kinds of problem these are.

---

5In fact not so notional; the translator reduces a multiarray to a flat Java array.
Finally, we should mention that the HPJava software is still relatively new. Neither the compiler or the libraries are highly optimized yet, and there are performance penalties because of that. We hope and expect this situation will change with time.

Having said all that, if you are determined to learn parallel programming, we think HPJava is an excellent place to start. And if you are already an expert at parallel programming, we hope you find HPJava a useful new tool.
Chapter 2

Processes and Distributed Arrays

In this chapter we start to discuss parallel programming in HPJava.

The HPJava parallel programming model is one of explicitly cooperating processes. It is an implementation of the Single Program, Multiple Data (SPMD) model. In this model a group of processors or processes execute the same program text, but the 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 to access elements owned by their peers. At these points explicit communications are needed to enable this access.

This general scheme has been very successful in realistic applications. Probably most successful applications of parallel computing to large scientific and numerical problems are programmed in this style. So HPJava is attempting to add some support at the language level for the established practices of programmers. Perhaps more importantly, it also provides a framework for the development of libraries of subroutines operating on distributed data.

What HPJava is not is any kind of parallelizing compiler—the HPJava software certainly cannot take a sequential program and convert it automatically into a job that runs efficiently on a parallel computer. Before writing an HPJava program you must think about the kind of parallel algorithms you are going to employ, and about what kind of communications between processors these algorithms imply. HPJava is basically a notation for expressing and implementing the parallel algorithms you come up with after that.

HPJava was designed for writing programs for distributed memory parallel computers. In principle that could mean just about any collection of computers joined by a fast enough network connection. The “distributed memory” qualifier just means that the participating processors don’t need to share a common main memory. In other words, generally speaking, the default situation. It is relatively straightforward, for example, to run an HPJava program on a group
of Linux PCs connected by Ethernet. In practice such ad hoc clusters probably have a high\textit{ communication latency}, and getting good performance on these platforms may be hard. A dedicated cluster with a high performance interconnect, or a proprietary parallel computer like an IBM SP machine, may be more a promising platform for the average HPJava program.

With this kind of target in mind, the underlying programming model is one of communicating processes, where we use the term "process" in the usual sense of operating systems. A process is a self-contained context in which a program executes with its own thread (or threads) of control and its own, protected, memory, and associated address space. Generally speaking this memory is inaccessible to other processes. In HPJava a "process" in this sense is always a Java Virtual Machine.

This was the original programming model we had in mind, and it remains the main rationale for the design of HPJava. Fairly late in the day—and perhaps a little reluctantly—we accepted that, at least for development purposes, it was very convenient to be able to run HPJava programs in a different mode. In this mode, the processes of an HPJava program are mapped to the Java threads of a single JVM. This allows you debug and demonstrate your HPJava programs without facing the ordeal of installing MPI or running on a network. As a byproduct, it also means you can run HPJava programs on \textit{shared memory parallel computers}. These kinds of machines are quite widely available today—sold, for example, as high-end UNIX servers. Because the Java threads of modern JVMs are usually executed in parallel on this kind of machine, it is possible to get quite reasonable parallel speedup running HPJava programs in the multi-threaded mode.

So HPJava now has two execution models: the original \textit{multi-process model}, and the newer \textit{multithreaded model}. Throughout most of this report we will use the term "process" interchangeably to mean an actual process in the multi-process model, or a thread started when an HPJava program is run under the multi-threaded model. Section 2.7 discusses the relationship between the two models in more detail, and explains how to run code under the multi-process model.

### 2.1 HPspmd Classes

Not every HPJava program is a parallel program. In Chapter 1 we saw some examples of \textit{sequential} HPJava codes. In fact any standard Java class can be used by an HPJava program, and typically the methods of standard Java classes aren't parallel programs.

For various reasons, some of which should become apparent as we go along, the HPJava system needs an unambiguous way to tell whether a given piece of code is supposed to be executed in a parallel way—distributed across multiple processes. In HPJava this is flagged by implementing a \textit{marker interface}. Marker interfaces are a familiar mechanism in Java. Well known examples are the \texttt{java.io.Serializable} interface, which marks the instances of a particular
2.1. HPSPMD CLASSES

class as being serializable, and the java.rmi.Remote interface, which marks the
methods of a class as being remotely callable.

HPJava defines a marker interface hpjava.lang.HPspmd. Any class that
implements this interface is referred to as an HPspmd class. The methods
and constructors of HPspmd classes are supposed to be executed in a parallel
context—they are meant to be invoked concurrently by a group of cooperating
processes.

Consider these two examples:

    public class EgSeqHello {
        public static void main(String [] args) {
            System.out.println("Hello world");
        }
    }

and

    public class EgParHello implements HPspmd {
        public static void main(String [] args) {
            System.out.println("Hello world");
        }
    }

The only significant difference is that the second class implements the HPspmd
interface (we don’t need an explicit import declaration for the hpjava.lang
package; it is imported implicitly into all HPJava programs).

Suppose we save these fragments in source files EgSeqHello.hpj and EgPar-
Hello.hpj respectively, and compile them using the hpjavac command as
described in section 1.1.2. We assume that the class path is set up as described
there, so that we are running under the multithreaded model.

If we start either class without specifying the hpjava.numprocs property,
the result is the same

$ java EgSeqHello
Hello world
$ java EgParHello
Hello world

In the latter case it happens this way because the default value of the hpjava.-
umprocs property is 1. If now we give a non-default value to the hpjava.num-
procs property, the behavior is different:

$ java -Dhpjava.numprocs=4 EgSeqHello
Hello world
$ java -Dhpjava.numprocs=4 EgParHello
Hello world
Hello world
Hello world
Hello world

Hello world
Because EgParHello is an HPspmd class, it is run concurrently in 4 different threads. Meanwhile, the hpjava.numprocs property is ignored when starting up a non-HPspmd class.

2.2 Process Grids

So a parallel HPJava program starts concurrently in all members of some process set. From inside the HPJava program, the processes are organized and labelled through special objects representing process groups. In general the processes in an HPJava group are organized into 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

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

An instance of the class hpjava.lang.Procs2 describes a 2-dimensional grid of processes. The grid p is visualized in Figure 2.1. This figure assumes that the program was executed in 11 processes, e.g. by setting the hpjava.numprocs property to 11. The call to the Procs2 constructor selects 6 of the available processes and incorporates them into a grid. One could think of this, loosely, as an act of "federation". But actually members are selected from the available pool of processes by a simple-minded, deterministic procedure—the details don't matter here. A single program can introduce several grids, and a single process can belong to more than one grid.

If a Procs2 constructor is invoked and the requested number of processes is greater than the number of processes the program is running on, an hpjava.-lang.ProcArrayTooLargeException is thrown.

The Procs2 constructor is an example of a collective operation. It should be invoked concurrently by all members of the active process group. Perhaps it won't come as too much of a surprise that Procs2 is another example of an HPspmd class. The property of being an HPspmd class has implications beyond just ensuring that a main() method is started in multiple processes. Generally speaking, if a method or constructor is to be invoked in a collective way, it will be defined in a class that implements the HPspmd interface.

Procs2 is also subclass of the special class hpjava.lang.Group. The Group class has a privileged status in the HPJava language. An object that inherits from this class can be used in a couple of special contexts. 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 this particular grid. An on construct restricts control to processes

---

1Our use of the term "grid" isn't supposed to be confused with the same term as used by the Global Grid Forum. Perhaps it would be better to change the name, but "process grid" seems as natural as anything.

2Or, more generally, if the requested number of processes is greater than the size of the current active process group, defined shortly.
2.2. PROCESS GRIDS

Figure 2.1: The process grid represented by p. Circles represent processes. The white circles are the processes inside p.

in its parameter group. So in

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

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.

    If we save the following program:

    public class EgOutIn implements HPepmd {
        
        public static void main(String [] args) {
            System.out.println("Hello from an outsider");

            Procs2 p = new Procs2(2, 3);
            on(p) {
                System.out.println("Hello from an insider");
            }
        }
    }

    to a file EgOutIn.hpj, compile it, and run it by:

    java -Dhpjava.numprocs=11 EgOutIn
we might see something like

```
Hello from an outsider
Hello from an insider
Hello from an outsider
Hello from an insider
Hello from an outsider
Hello from an insider
Hello from an outsider
Hello from an insider
Hello from an outsider
Hello from an insider
Hello from an outsider
Hello from an outsider
Hello from an outsider
Hello from an outsider
Hello from an outsider
Hello from an outsider
```

The "outsider" message is printed 11 times; the "insider" message is printed 6 times. The exact order in which messages appear is non-deterministic, because there are 11 interleaved processes or threads.

The `on` construct in these examples establishes `p` as the *active process group* (or *APG*) within its body. The idea of an active process group that can change from line to line as the program is executed is an important concept in HPJava. The influence of the active process group will be seen in various places as we describe the language.

There is an auxiliary class `hpjava.lang.Dimension` associated with process grids. Objects of class `Dimension` describe a particular dimension or axis of a particular process grid. They will refer to them as *process dimensions*. The process dimensions of a grid are accessible through the inquiry method `dim(r)` of the `Proc` class. The argument `r` is in the range `0, \ldots, R-1`, where in general `R` is the rank (dimensionality) of the grid.

The `Dimension` class in turn has a method `crd()` that returns the local *process coordinate* associated with the dimension—i.e., the position of the local process within the dimension. If compile and execute the following HPJava
2.2. PROCESS GRIDS

![Diagram of a 2-dimensional grid with coordinates](image)

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

```java
public class EgCrds implements HPspmd {
    public static void main(String[] args) {
        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 see output something like:

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

Because the 6 processes are running concurrently there is no way to predict the order in which the messages appear. The dimensions of p are crudely illustrated in Figure 2.2. If we applied crd() to d or e in a process outside p (like one of the shaded processes in Figure 2.1) an `hjava.lang.ProcessNotInGroupException` would be thrown.

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, perhaps, Procs0 is a group containing exactly one process. This class is sometimes used to create a group
CHAPTER 2. PROCESSES AND DISTRIBUTED ARRAYS

that singles out a unique controlling process in a program. 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, but syntactically it is just a class. The status of Group in HPJava is comparable with the status of Throwable, say, in standard Java—only a Throwable object can appear in a throw statement, and only a Throwable class can appear in the header of a catch block or a throws clause. Likewise in HPJava, only a Group object can appear in the header of an on construct (or in the on clause of distributed arrays, which will be introduced later).

2.3 Distributed Arrays

Probably the most important feature HPJava adds to Java is the distributed array. A distributed array is a collective multiarray, shared by a number of processes. Like an ordinary multiarray, a distributed array has a rectangular, multidimensional 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.

The distribution of an index space is parametrized by objects belonging to another class that (like Group) has special status in the HPJava language. This is the hpjava.lang.Range class. Instances of Range classes are called distributed ranges.

The global index values for a dimension of a distributed array always lie in the range $0, \ldots, N - 1$, where $N$ is the extent of the dimension. This is the same as for a multiarray. The distributed range object for the dimension specifies the extent; it also specifies a process dimension over which the indexes are scattered,

---

In HPJava 1.0 the highest ProcN class is Proc3. But you can use a constructor of the base class Proc to create higher dimension grids if you actually need them.
2.3. DISTRIBUTED ARRAYS

and it specifies the format in which the indexes are distributed over the process
dimension.

Distributed arrays can be thought of as a generalization of multiarrays, and
their type signatures look quite similar. The only difference is that a distributed
array uses hyphens instead of asterisks in the dimension slots (we will see later
that actually you can mix hyphens and asterisks in a distributed array type
signature). Distributed array creation expressions look very similar to multiarray
creation expressions, except that distributed ranges appear in place of the
integer extent expressions.

In the following example we create a two-dimensional, \( N \) by \( N \), array of float-
ing point numbers, with elements distributed over the grid \( p \).

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

\begin{verbatim}
  on(p) {
    Range x = new BlockRange(N, p.dim(0));
    Range y = new BlockRange(N, p.dim(1));

    float [-1,-1] a = new float [[x, y]];
  }
\end{verbatim}
```

The decomposition of this array for the case \( N = 8 \) is illustrated in Figure 2.4.
The choice of subclass \( \text{BlockRange} \) 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. The constructor
for \( \text{BlockRange} \) takes two arguments: the extent of the range, and the process
dimension over which the range is distributed.

In this example the distributed array was created inside an \text{on}\,(p) \text{construct}.
This ensured that the \text{distribution group} for elements of \( a \) was the grid \( p \) (the
active process group at this point). We will see later that it is possible to create
distributed arrays outside such constructs, but in our initial examples will use
on constructs to implicitly set the distribution group.

In any case, the ranges of a distributed array must be distributed over
different dimensions of the array's distribution group\(^4\). If they are not, an
\text{hpjava.lang.DimensionNotInGroupException} will be thrown. Not surpris-
ingly, distributed array creation is a collective operation, and it can only appear
inside \text{HPspmd} code.

For illustration the distributed arrays appearing in examples in this report
are often quite small. We should point out that in practice it's unlikely that it
would worthwhile to distribute such small arrays. For data parallel program-
ing to be effective the distributed data structures should usually be \text{large}.

\(^4\) Unless they are collapsed ranges; see section 3.3.
2.4 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 distributed, parallel loop. So the statement

\[
\text{overall}(i = x \text{ for :})
\]

\[
\ldots
\]

can be read as “for all elements, \(i\), of the distributed range \(x\), do \ldots\”).

The colon in the overall headers of the example is shorthand for an index triplet (see section 1.2.4) so the forms

\[
\text{overall}(i = x \text{ for } l : u : s)
\]

\[
\ldots
\]

and

\[
\text{overall}(i = x \text{ for } l : u)
\]

\[
\ldots
\]

are also allowed. In the original form the lower bound defaults to 0 and the upper bound defaults to \(N - 1\), where \(N\) is the extent of the range before the for keyword. In other words it is equivalent to

\[
\text{overall}(i = x \text{ for } 0 : x.\text{size()} - 1 : 1)
\]
2.4. **PARALLEL PROGRAMMING**

```java
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 [x, y] 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.

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

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

    ... initialize 'a'

    Adlib.shift(a, 1, 0);  
    Adlib.shift(a, -1, 0);  
    Adlib.shift(x, 1, 1);  
    Adlib.shift(s, a, -1, 1);

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

Figure 2.6: A parallel stencil update program.
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 become clearer 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 in Figure 2.6 is not a new feature of the HPJava language. Instead it is a member of a particular library called Adlib. Adlib is a library of collective operations on distributed arrays. The function shift is overloaded to apply to various kinds of array. In this example we are using the version applicable to two dimensional distributed arrays with float elements:

```java
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. See section 6.1 for more information on the AdLib shift() method. To use the methods of the hpjava.adlib.Adlib class in the manner illustrated here there should be a suitable import declaration at the start of the 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:

```java
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 to this question relates to the nature of the symbols i and j. No declaration was given for these names, but it would be reasonable to guess that they stand for integer values. In fact they don’t.

---

5 Many of them, by the way, modelled on the array transformational intrinsic functions of Fortran.


2.5 Locations

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

\[
\text{Proc2} p = \text{new Proc2}(2, 3); \\
\text{on}(p) \{ \\
\text{Range} x = \text{new BlockRange}(N, p.\text{dim}(0)); \\
\text{Range} y = \text{new BlockRange}(N, p.\text{dim}(1)); \\
\ldots \\
\} \\
\]

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

\[
\text{overall}(i = x \text{ for } 1 : N - 2) \{ \\
\ldots \\
\} \\
\]

the symbol \( i \) is called a distributed index. The value associated with a distributed index is considered to be 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 of 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 \( x[0], x[1], \ldots, x[N - 1] \). 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 \( (1, 1) \).

There is one other construct in HPJava besides overall that defines a distributed index. This is the relatively simple 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

\[
\text{float } [\ldots] \ a = \text{new float } [x, y]; \\
\ldots \\
a[1, 4] = 73; \quad \text{// ERROR} \\
\]

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

\[
\text{float } [\ldots] \ a = \text{new float } [x, y]; \\
\ldots \\
\text{at}(i = x[1]) \\
\text{at}(j = y[4]) \\
a[1, j] = 73; \\
\]
The symbols \( i \) and \( j \), scoped within the construct bodies, are distributed indices. The effect 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 back to Figure 2.7, the outer

\[
\text{at}(i = x \ [1])
\]

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

\[
\text{at}(j = y \ [4])
\]

then restricts execution down to just process \((0, 1)\). This is exactly the process that owns element \( a[1, 4] \).

So we see that the odd-looking 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, like the `Adlib.shift()` operations used in the previous section.

We can now formally define the meaning of the `overall` construct in terms of the simpler `at` construct. If \( a \) is greater than zero, the construct

\[
\text{overall}(i = x \ \text{for} \ l : u : s ) \\
\text{...}
\]
is equivalent in behavior to

\[
\text{for(int } n = 1; n <= u ; n += s) \\
\text{at(i = x [n]) }
\]

\[\ldots\]

If \( s \) is less than zero, it is equivalent to

\[
\text{for(int } n = 1; n >= u ; n += s) \\
\text{at(i = x [n]) }
\]

\[\ldots\]

The bodies of the \text{at} constructs are skipped for values of \( n \) that don’t correspond to locally held elements. In practice of course the HPJava compiler translates the \text{overall} construct more efficiently than this definition suggests.

The \text{at} construct completes the contingent of special control constructs in HPJava. We sometimes refer to the three constructs on, \text{at} and \text{overall} as \text{distributed control} constructs.

The backquote ``, \( `\), 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 \text{overall} given above, this is the value called \( n \).

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

### 2.6 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 well-known to scientific programmers: solution of the two-dimensional Laplace equation with Dirichlet boundary conditions by the iterative Jacobi relaxation method. The boundary conditions of the equation are fixed by setting edge elements of an array. These elements don’t change during 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 2.4. Here we put it in the context of a working program.

The initialization is done with a pair of nested \text{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 explained at the end of the last section we, apply the operator \( ` \) to distributed indices to get the global loop index.
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;

    // 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 = 0.25F * (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.printf("%.3f\n" + N + "\n", a);
}

Figure 2.8: Solution of Laplace equation by Jacobi relaxation.
2.7. MULTI-PROCESS AND MULTITHREADED EXECUTION MODELS

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

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 practice. 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 Math.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 Adlib.maxval() finds the largest element of distributed array and broadcasts its value to all processes that call the function. (We didn’t initialize the edges of the array r. But, as for ordinary Java arrays, elements of multiarrays and distributed arrays get a default value: 0 for numbers.)

Finally a collective library function Adlib.sprintf() prints a formatted version of the array on the standard output stream. You can find details of how the output is formatted in section 6.4.

There is a working version of this example in the hpjdk release under the file hpjdk/examples/FPHPJ/EgJacobiLaplace.hpj. Compile it by:

hpjavac EgJacobiLaplace.hpj

and run it (under the multithreaded model) by, e.g.:

java -Dhpjava.numprocs=4 EgJacobiLaplace.hpj

2.7 Multi-process and Multithreaded Execution Models

As explained in the introduction to this chapter, an HPJava program can run under one of two models: the original multi-process model, and the multithreaded model.

Before you can run a program in multi-process mode you should install a suitable version of the mpiJava package. At the time of writing this is available for various Linux and UNIX platforms. It is possible to install mpiJava on Windows as well, but this is not well-supported in the current version. You need to go to the www.hpjava.org site and download mpiJava version 1.2.5x or

---

*We plan to improve support for Windows in mpiJava 1.3.*
CHAPTER 2. PROCESSES AND DISTRIBUTED ARRAYS

later. The “x” is important—the extended version includes an implementation of the mpi4j API that is used in the implementation of AdLib.

Unpack the mpiJava release, and carefully read the file mpiJava/README for installation instructions.

Once you are confident mpiJava is properly installed, you should change your CLASSPATH environment variable to remove the multithreaded implementation of mpi4j and add the mpiJava version, e.g.

```bash
export CLASSPATH=.:$HPJAVA_HOME/classes:/home/mpiid/mpiJava/
```

This assumes you are using the bash shell, and you installed mpiJava in the directory `/home/mpiid/mpiJava/`. It should then be possible to run HPJava programs in multi-process mode using the prun java script from mpiJava. For example go to the subdirectory `hpjdk/examples/fft2d` and execute the command

```
prun java 4 Wolf
```

This runs the Wolf demo in multiprocess mode in 4 processes. You can add a `machines` file to the current directory to specify which hosts these 4 processes run on. (You don’t need to recompile the Wolf program to switch models—the two implementations of mpi4j are binary compatible.)

If an HPJava program is written in the style originally intended—for execution on distributed memory parallel computers—then it may be possible to run the program in either model and have it behave the same way. This is a useful property—it is very convenient to be able to debug programs in a single JVM before launching them on a network or parallel computer. However this equivalence doesn’t happen automatically. It requires you to write your programs in a certain style.

First of all, and most obviously, you should avoid using standard Java thread synchronization mechanisms to communicate between the processes of an HPJava program. This may work under the multithreaded model, but it certainly won’t work under the multi-process model. Two HPJava processes should not share access to a common variable. Whether they do this under the protection of synchronized methods or the synchronized statement, or whether they use volatile fields—however they do it—none of this makes any difference: it just won’t work under the multi-process model.

This does not mean that individual, logical processes of an HPJava program should not be multithreaded. If a process is started up when an HPJava program is initiated, and it subsequently creates another local thread, the initiating process (or thread) can sensibly interact with the new thread through the ordinary thread synchronization features of Java. This will work under the multithreaded or the multi-process model (of course the new thread shouldn’t directly interact with any of the processes of the HPJava program other than its parent thread). You can see examples of technique at work in the hpjdk examples suite—one such is the Wolf program mentioned above, which starts and a second thread in the controlling process to deal with the GUI.
2.7. MULTI-PROCESS AND MULTITHREADED EXECUTION MODELS

These restrictions help to make sure that if a program works properly under the multithreaded model, it will subsequently work correctly under the multi-process model. There are other restrictions that go the other way.

A program running in the multi-process mode may rely on the of the per-process locality of certain "global" variables. In particular, in the multi-process model, each process has its own copy of every a static field (class variable) in the program. If a program is written in such a way that it depends on the process-locality of static fields, it will behave differently when the program is run under the multithreaded model. This different behavior usually manifests as a bug—typically some race condition associated with access to the unintentionally shared variable.

Our recommendation is to try to avoid all use of static fields in HPJava programs. If you need per-process global variables to share information across method invocations, define the global variables as instance fields, and have the main() method create an instance of your main class. Then access the instance fields in instance methods. Many of the examples in the hpjdk example suite simply avoid use of global variables, but see hpjdk/examples/pde2/PDE2.hpj for an example written in this style.

Sometimes you are using library functions, and it is difficult to know whether they internally use static fields. Well there is no very good solution to that problem. If you want to write code that runs under either model, you just have to be aware of the possible issues, and avoid libraries that seem likely to use and modify static fields.

It is clear that switching between the two models is not an entirely safe thing to do, and we aren't yet sure whether it is really something to be encouraged for large codes. But it is very convenient when it works. All the parallel codes in the hpjdk test suite and examples suite (except for a couple that explicitly use MPI calls) can be run in either mode.
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 should be quite 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 directly as array ranges. CyclicRange is directly analogous to the cyclic distribution format available in HPF.

Cyclic distributions are sometimes favored 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 favors 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

```java
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 [1,-1] 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
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

```java
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
3.1. OTHER DISTRIBUTION FORMATS

![Diagram of block distribution]

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

unchanged. HPJava does not guarantee this property in the same way that HPF does, and fully optimized HPJava programs are unlikely to be so easily redistributed. But it is still a useful feature.

As a more graphic example of how cyclic distribution can improve load balancing, consider the Mandelbrot set code of Figure 3.4.

The Mandelbrot set is defined in terms of points in a plane (actually the plane of complex numbers). Mathematically, the set is a subset of the points in the plane—those points for which a certain iteration does not converge. Figure 3.4 runs this iteration at each point of a grid embedded in the plane. If the iteration of the while loop terminates normally, the current point is not in the set. If the while loop would carry on indefinitely, the current point is in the set. In practice one approximates the set as being those points for which the while loop iterates a sufficiently large number of times—CUTOFF should be some moderately large integer.

We represent the set as a distributed array of char. On completion points outside the set have value ' ', and point inside the set have value ' # '. This representation is chosen just because it makes a recognizable display when the array is printed out with Adlib.printf().

Points away from the set are generally eliminated in a few iterations, whereas those inside the set or close to it take much more computation. 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
Figure 3.3: Work distribution for an example with cyclic distribution.

hold most of the set. Changing the class of the ranges to CyclicRange gives the much more even distribution shown in Figure 3.6.

We are not suggesting, by the way, that Mandelbrot computation is a serious application of HPJava. It is just a means as a simple illustration of an issue about distribution format and load balancing.

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

### 3.2 Ghost Regions

In a distributed array with ghost regions, the memory for the locally held block of elements is allocated with extra space around the edges. These extra locations can be used to cache some of the element values properly belonging to adjacent processors. The inner loop of algorithms involving stencil updates can then 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 piece of syntax, the following
Procs2 p = new Procs2(2, 3);
on(p) {
    Range x = new BlockRange(N, p.dim(0));
    Range y = new BlockRange(N, p.dim(1));
    char [[-,-]] set = new char [[x, y]] ;
    overall(i = x for ;)
    overall(j = y for ;) {
        double cr = (4.0 * i' - 2 * N) / N ;
        double ci = (4.0 * j' - 2 * N) / N ;
        double zz = cr, zi = ci ;
        set [i, j] = ' ' ;
        int k = 0 ;
        while (zz * zz + zi * zi < 4.0) {
            if(k++ == CUTOFF) {
                set [i, j] = '#' ;
                break ;
            }
            // z = c + z * z
            double newr = cr + zz * zz - zi * zi ;
            double newi = ci + 2 * zz * zi ;
            zz = newr ;
            zi = newi ;
        }
    }
    Adlib.printf("%c \%d + N + \"N", 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.
expression

\[ name \pm expression \]

is a legal subscript if \( name \) is a distributed index and \( expression \) is an integer expression—often a small constant. This kind of subscript is called a \textit{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 \textit{plus} surrounding ghost region, an exception may be thrown. Using this syntax the example at the end of section 2.3:

\begin{verbatim}
overall(i = x for 1 \text{ : } N - 2)
overall(j = y for 1 \text{ : } N - 2)
a[i, j] = 0.25F \ast (a[i - 1, j] + a[i + 1, j] +
a[i, j - 1] + a[i, j + 1]);
\end{verbatim}

is allowed \textit{provided} the array \( a \) has suitable ghost extensions.

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. The library function called \texttt{Adlib.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 \texttt{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.

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 \textit{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 \texttt{overall} construct.

The "footprint" of the stencil update can be more general. Figure 3.9 shows an implementation of the well-known cellular automaton, Conway's Game of Life. The local update involves dependences on diagonal neighbours. This example also shows the most general form of \texttt{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 \texttt{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.
Procs2 p = new Procs2(P, P);
on(p) {
    Range x = new ExtBlockRange(N, p.dim(O), 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 = 0.25F * (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 ;
            }

        overall(i = x for 1 : N - 2)
            overall(j = y for 1 : N - 2)
                a [i, j] = b [i, j] ;
    } while(Adlib.maxval(r) > EPS);

    /*
    */
}

Figure 3.7: Solution of Laplace equation using ghost regions.
3.2. GHOST REGIONS

```java
do {
    for(int parity = 0 ; parity < 2 ; parity++) {
        Adlib.writeHalo(a);

        overall(i = x for i \(=\) 1 : \(N - 2\))
            overall(j = y for i + (i' + parity) \% 2 : \(N - 2 : 2\)) {
                float newA;
                newA = 0.25F * (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).

using the Random class from the standard Java package java.util. 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 probably too naive for a reliable simulation, but as usual it illustrates a principle.)
int wlo[] = {1, 1}, whi[] = {1, 1}; // ghost widths for ‘writeHalo’
int mode[] = {Adlib.CYCL, Adlib.CYCL}; // boundary conds for "

Proc2 p = new Proc2(2, 2);
on(p) {
    Range x = new ExtBlockRange(W, p.dim(0), 1, 1);
    Range y = new ExtBlockRange(W, p.dim(1), 1, 1);

    int [[-1,-1]] state = new int [[x, y]];
    ...
    Define initial state of Life board
}

// Main update loop.

int [[-1,-1]] sums = new int [[x, y]];
for(int iter = 0; iter < NITER; iter++) {
    Adlib.writeHalo(state, wlo, whi, mode);
    ...
    Calculate neighbour sums.
    overall(i = x for :)
        overall(j = y for :)
            sums[i, j] =
                state[i - 1, j - 1] + state[i - 1, j] + state[i - 1, j + 1] +
                state[i, j - 1] + state[i, j + 1] +
                state[i + 1, j - 1] + state[i + 1, j] + state[i + 1, j + 1];
    ...
    Update state of board values.
    overall(i = x for :)
        overall(j = y for :)
            switch (sums[i, j]) {
                case 2: break;
                case 3: state[i, j] = 1; break;
                default: state[i, j] = 0; break;
            }
    ...
    Output final state
}

Figure 3.9: Conway’s Game of Life.
3.2. GHOST REGIONS

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

Proc2 p = new Proc2(P, P);
on(p) {
    Range x = new ExtBlockRange(N, p.dim(0), 1, 1);
    Range y = new ExtBlockRange(N, p.dim(1), 1, 1);
    int [[-, -]] latt = new int [[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.
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

```java
Process p = new Process(3);
for (int i = 0; i < p.dim(0); i++) {
    Range x = new CollapsedRange(N);
    Range y = new BlockRange(N, p.dim(0));

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

    ... // rest of the code
}
```

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, by the way, is our first example of a one-dimensional process “grid”.

Because of the limitations on what kinds of subscripts are allowed in accessing distributed array elements, the language defined so far doesn’t provide a good way to exploit the locality implied by a collapsed dimension. So in practice it is doubtful that one would explicitly create a distributed array with a collapsed dimension in the way just shown. Instead HPJava is extended to allow distributed arrays to have sequential dimensions.
3.4. DISTRIBUTION GROUPS AND REPLICATION

Given that we introduced sequential multiarrays in Chapter 1, sequential dimensions shouldn't seem like an unfamiliar idea. The type signature of a distributed array may have asterisks in place of hyphens in some of its dimensions. This indicates that these dimensions implicitly have a collapsed range. These dimension can be subscripted with integer expressions just like in a sequential multiarray dimensions. Here is an example:

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

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

The `at` construct deals with the distributed dimension, but there is no need for distributed indices in subscripting the sequential dimension. The distributed array creation expression is passed integer extent expressions for sequential dimensions. A `CollapsedRange` object will be created for the array, but generally the programmer does not need to be aware of its existence.

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

The standard library function `HPut.all.copy()` copies elements between two `aligned` arrays (unlike `Adlib.remap()`, `copy()` does not implement communication; if it is passed non-aligned arrays, an exception occurs). The `cshift()` method of `Adlib` is similar to the `shift()` method introduced earlier, except that it implements a cyclic shift—wrapping edge values around.

Distributed array types with sequential dimensions are technically subtypes of corresponding distributed array types without sequential dimensions. 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 Distribution Groups and Replication

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

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

  ... initialize 'a', 'b'

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

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

    // cyclically shift 'b' (by amount -1 in x dim)...  
    Adlib.cshift(tmp, b, -1, 1);
    HPutil.copy(b, tmp);
  }
}

Figure 3.12: A pipelined matrix multiplication program.
the array.

```java
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

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

on(p) {
    float [[x]] c = new float [[x]];
    ...
}
```

The range of `c` is sequential, and the array is replicated over both dimensions of `p`. This makes it essentially the same as the kind of sequential multiarray described in Chapter 1, although actually there is a subtle distinction between a multiarray created in HPspmd code and a multiarray created in non-HPspmd code. A multiarray created in HPspmd code always has a well-defined distribution group. For `c` this distribution group is `p`. A multiarray created in non-HPspmd code does not have a distribution group (in fact the way this is implemented is that the non-HPspmd multiarray has distribution group of null).

In the last section we gave 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
PROC2 p = new PROC2(P, P);
on(p) {
    Range x = new BlockRange(N, p.dim(0));
    Range y = new BlockRange(N, p.dim(1));

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

    ... initialize 'a', 'b'

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

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

        c[i, j] = sum;
    }
}

Figure 3.13: A direct matrix multiplication program.
3.4. DISTRIBUTION GROUPS AND REPLICATION

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.
static 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.

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

The example also illustrates the most general form of the distributed array creation expression. 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 distributed array creation expression itself itself can specify that the array is distributed over some subset of the active group. This allows one to create an array outside the on construct that will processes its elements. Through communication functions like remap, values
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.

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.

A special case of the pattern of multiple grids singles out one process of the
program as a “control” processor, responsible for things like I/O. This can be
done by creating a Procs0 singleton grid:

Procs0 control = new Procs0();

Code that is only to be executed by the control process is done in a suitable
on(control) construct. For example input or output of distributed data can
be done in such a construct, reading to or writing from an array with sequential
ranges, and distribution group control. This array can then be remapped to
or from a “real” distributed array with distributed ranges.
Chapter 4

Distributed Array Sections

In Chapter 1 we introduced the HP Java syntax for forming sections of multiarrays. This syntax carries over with very few changes to distributed arrays.

We have seen that the subscripts in a distributed array element reference are either generally either index symbols or (restrictedly) integer expressions. Options for subscripts in distributed array section expressions are wider. As for multiarrays a section subscript is allowed be a triplet. The section may also have some scalar subscripts, similar to those appearing in element references.

This fragment includes two examples of distributed array section expressions:

```java
Proc32 p = new Proc2(p, P);
for(p) {
    Range x = new BlockRange(N, p.dim(0));
    Range y = new BlockRange(N, p.dim(1));

    float [1, -1] a = new float [x, y];
    float [-] b = a [0, :];

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

The first array section expression appears on the right hand side of the definition of `b`. It specifies `b` as an alias for the first row of `a` (Figure 4.1). In an array section expression, unlike in an array element reference, a scalar subscript is always allowed to be an integer expression\(^\d\). 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.

Distributed 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.

\(^{1}\)Distributed indices are allowed as well, if the location is in the proper range.
Figure 4.1: A one-dimensional section of a two-dimensional array (shaded area).

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

In image processing applications Fast Fourier Transforms (FFTs) and related transformations are sometimes applied to two-dimensional images. We have used the Wolf program from the hpjdk release as an example in earlier chapters.

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 sequential multiarrays. A section dimension naturally inherits the sequential property (the asterisk in the type signature) from the associated dimension of the parent array. The method fftid() implements a standard sequential algorithm—we don’t reproduce it here. You can find the complete code in the hpjdk release under hpjdk/examples/FFHPJ/EgfFourier2d.hpj. Because Java doesn’t have complex numbers, we store real and imaginary parts in pairs of arrays whose names are prefixed re and im. Another possibility would be to add an extra dimension of extent 2 to the arrays. After processing all columns, the data is remapped so that each row is a sequential subarray, and these are processed in the same way.

The Wolf program itself, under hpjdk/examples/fft2d/ contains a generalized version for non-square images.

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 practice this is followed by forward and back substitutions:

$$Ly = b, \quad L^T x = y$$

to complete the solution of $Ax = b$. A pseudocode algorithm for Cholesky decomposition is

For $k = 1$ to $n - 1$

$$l_{kk} = \frac{a_{kk}}{l_{kk}}$$

For $s = k + 1$ to $n$

$$l_{sk} = \frac{a_{sk}}{l_{kk}}$$

For $j = k + 1$ to $n$

For $i = j$ to $n$

$$a_{ij} = a_{ij} - l_{jk}l_{jk}$$

$$l_{nn} = a_{nn}$$

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
static void fftid(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 ;)
        fftid(reA [[i, :]], imA [[i, :]]) ;

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

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

    ... result is in 'reB', 'imB'
}

Figure 4.3: A two-dimensional Fourier Transform.
Procs1 p = new Procs1(P);
on(p) {
  Range x = new CyclicRange(N, p.dim(0));
  float [[*,1]] a = new float [[N, x]];
  float [[*]]  m = new float [[N]]; // a buffer
  ...
  some code to initialise 'a'
  for(int k = 0 ; k < N - 1 ; k++) {
    at(j = x [k]) {
      float d = (float) Math.sqrt(a [k, j]) ;
      a [k, j] = d ;
      for(int i = k + 1 ; i < N ; i++)
        a [i, j] -= m [i] * m [j] ;
    }
    Adlib.remap(m [[k + 1 : N - 1]], a [[k + 1 : N - 1, k]]);
    overall(j = x for k + 1 : N - 1)
    for(int i = j' ; i < N ; i++)
      a [i, j] -= m [i] * m [j'] ;
  }
  at(j = x [N - 1])
  a [N - 1, j] = (float) Math.sqrt(a [N - 1, j]) ;
}

Figure 4.4: Cholesky decomposition.

in the lower part of the input array a. Note that the array m has a replicated
distribution, so the remap operation is a broadcast of the relevant part of column
k.

4.3 Matrix multiplication with reduced memory

One disadvantage of the program in Figure 3.15 of the last section 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 storing copies of elements
only 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.
static 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();
    ...
    Select "band-width", B
    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 = Math.min(B, N - base);
        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.
4.4. **Subranges**

For simplicity we assumed here that B is a compile-time constant. Alternatively we can compute this value dynamically. The `volume()` method on `Range` is used internally by array constructors to control allocation of memory for array elements. It defines the largest block of locations of the current range held by any processor. Hence an upper bound on the number of elements held by any processor for `ta` and `tb` combined is

\[
B = x.\text{volume}() + B * y.\text{volume()}
\]

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

\[
B = \text{MAX_TEMPORARY_SIZE} / (x.\text{volume()} + y.\text{volume()})
\]

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

### 4.4 Subranges

Let's take another look at the example of the array section in figure 4.2. We can capture this section in a named variable as follows

```java
float [-1] a = new float [x, y];
float [-1] 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

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

This syntax should look quite natural. It is modelled on the syntax for multi-array sections themselves.

The global indices associated with the subrange `v`, for example, are in the range \(0, \ldots, v.\text{size()}\). 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.
That covers the distributed ranges of sections. 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 distributed array \( a \). But the example of figure 4.1 is problematic. This was constructed using a scalar subscript, effectively as follows:

```plaintext
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:

```plaintext
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 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. The division operator is overloaded to describe these subgroups. The distribution group of \( b \) is equivalent to \( q \), defined by

```plaintext
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 was tacit in the definition of an abstract location. Without formally defining the idea, we used it implicitly in section 2.5. 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 be written as

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

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

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

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

```plaintext
```
or as


The restricted groups introduced here have a simple concrete representation. A restricted group can be 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 \texttt{Procs} object, plus just two \texttt{int} fields. It turns out that this representation is not only compact; it also leads itself to efficient computation of the most commonly used operations on groups.

Notice by the way that the inquiry function \texttt{dim()} is a member of the \texttt{Procs} class (the process grid class), \textit{not} the superclass \texttt{Group}, which also embraces restricted groups.

4.6 Mapping of distributed array sections

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

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

\[ p/i/j/\ldots \]

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

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

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

Because, as noted in section 4.4 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 did.

It shouldn't come as a surprise that subranges and restricted groups can be used in array constructors, on the same footing as the ranges and groups.
described in earlier sections. This means that temporary arrays can be constructed with identical mapping to any given section. This facility is useful when writing generic library functions, like the \texttt{matmul()} of Figure 3.15, which must accept complete distributed arrays or distributed array sections without discrimination\footnote{It also allows HPJava arrays to reproduce the full panoply of alignment options supported by the \texttt{LIKE} directive of High Performance Fortran.}.

In the last three sections we 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 closure to the language. In our experience it is fairly uncommon to see subgroups or subranges used explicitly in HPJava programs.

4.7 Rank-0 Distributed Arrays

There is no restriction that the list of subscripts in an array section expression must include some triplets. It is legitimate for all the subscript to be “scalar”. In this case the resulting distributed 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 \texttt{remap()}. If \( a \) and \( b \) are distributed arrays, we cannot usually write a statement like

\[
a \begin{bmatrix} 10, 10 \end{bmatrix} = b \begin{bmatrix} 30 \end{bmatrix};
\]

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

\[
\text{Adlib.remap}(a \begin{bmatrix} 10, 10 \end{bmatrix}, b \begin{bmatrix} 30 \end{bmatrix});
\]

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

There is a natural syntax for creating the new kind of object directly. Rank-0 distributed arrays (which we sometimes simply call “scalars”), can be created as in this example:

\[
\text{float } \begin{bmatrix} \end{bmatrix} c = \text{new float } \begin{bmatrix} \end{bmatrix};
\]

\[
\text{float } a \begin{bmatrix} -1, \end{bmatrix} = \text{new float } \begin{bmatrix} x, y \end{bmatrix};
\]

\[
\text{Adlib.remap}(c, a \begin{bmatrix} 10, 10 \end{bmatrix});
\]

\[
\text{float } d = c \begin{bmatrix} \end{bmatrix};
\]
4.8 Distributed Array Restriction

Library functions operating on distributed arrays often specify certain alignment relations between their array arguments. In HPJava it is natural to define two arrays to be aligned if they have the same distribution group and all their ranges are aligned\(^3\). The Adlib method 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 simplest 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 \( x \). 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

\[
c [ i, j ] = \text{Adlib.dotProduct}(a [[i, ]], b [[i, j]])
\]

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

\(^3\)Later we will give more detailed definitions.
```java
Procs3 p = new Procs3(P, P, P) ;
  on(p) {
    Range x = new BlockRange(W, p.dim(0)) ;
    Range y = new BlockRange(W, p.dim(1)) ;
    Range z = new BlockRange(W, p.dim(2)) ;

double [[-,-]] c = new double [[x, y]] ;

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

... initialize 'a', 'b'

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

Figure 4.6: A maximally parallel matrix multiplication program.

versions of both these sections we reduce both groups down to p/1/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.

Incidentally, this example illustrates some interesting principles, but it is not supposed to be a practical implementation of matrix multiplication. The overhead of making $N^2$ separate collective communication calls will far outweigh the notional advantage of their parallel execution.
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 have often 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—you may want to revisit the discussion in section 2.5. 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 2.2, the construct

\[
on(p) \{ 
\ldots 
\}
\]

changes the active group to \( p \) inside its body. The associated rule is:

**Rule 1** The construct

\[
on(p) \{ \ldots \}
\]

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

In other words, an on construct cannot add any new process to the active group. If this rule is not satisfied an \texttt{hpjava.lang.GroupNotContainedException} will be thrown on entering the construct.
Now, if the current active group is *p*, executing the construct construct
\[
\texttt{at(i = x [n])}
\]
\[
\ldots
\]
or
\[
\texttt{overall(i = x for l : u : s)}
\]
\[
\ldots
\]
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*. So we can conclude that:

**Rule 2**  Unless *x* is a collapsed range, a control construct
\[
\texttt{at(i = x [n])}
\]
\[
\ldots
\]
or
\[
\texttt{overall(i = x for l : u : s)}
\]
\[
\ldots
\]
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.

If this rule is not satisfied an `hpjava.lang.DimensionNotInGroupException` will be thrown on entering the construct.

As an example of the application of this rule, notice that 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:
\[
\texttt{at(i = x [n])}
\]
\[
\texttt{overall(i = x for l : u : s)} \quad \text{// ERROR}
\]
\[
\ldots
\]

This is good, because the outer construct already restricts control to a single coordinate value, and it doesn’t make sense to try distributing control across all coordinates of the same process dimension inside that construct\(^2\).

### 5.2 Rules for distributed array constructors

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

\(^1\) Alternatively *x* can be a collapsed range, in which case *p/i* is defined to be equal to *p*.

\(^2\) Strictly speaking this nesting is legal (but pointless) if *x* is collapsed.
Rule 3 The distributed array constructor expression

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

can only appear in a context where the distribution group, \( p \), is contained in the currently active process group. If \( e_r \) is a (non-collapsed) range object, its process dimension must belong to the dimension set of \( p \). No two range objects in \( e_0, \ldots, e_r, \ldots \) can be distributed over the same process dimension of \( p \).

If the first part of this rule is not satisfied an \texttt{hpjava.lang.GroupNotContained-Exception} will be thrown. If the other parts of the rule are not satisfied an \texttt{hpjava.lang.DimensionNotInGroupException} will be thrown.

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 2.5 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

\[ a[e_0, \ldots, e_r, \ldots] \]

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

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

Rule 5 In the distributed array element reference:

\[ a[e_0, \ldots, e_r, \ldots] \]

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

\(^3\)Of course there is nothing to prevent a compiler applying these checks at compile-time if it can do so.
**Rule 6** The distributed array element reference in:

\[
\text{at}(i = x [n]) \{ \\
\ldots a[e_0, \ldots, e_{r-1}, i, e_{r+1}, \ldots] \ldots \\
\}
\]

is allowed if and only if

1. The expression \(a\) is invariant in the \(\text{at}\) construct\(^4\).

2. The location \(x[n]\) is an element of the range \(a.\text{rng}(r)\)\(^5\).

or

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

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

is allowed if and only if

1. The expression \(a\) is invariant in the \(\text{overall}\) construct.

2. All locations in \(x \text{ [I:u:s]}\) are of elements \(a.\text{rng}(r)\).

A subtle point to appreciate is that rules 6 and 7 are statements about the \(\text{at}\) and \(\text{overall}\) 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 potentially 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 \(\text{overall}\).

We should point out that the run-time rules given in this section and the following subsection are not checked by the HPJava compiler in the hpjdk 1.0 release. This is a shortcoming that we hope will be rectified in future releases.

---

\(^4\)The interpretation of “invariant” will be discussed in section A.2.1. Actually this condition is more important for the case of an index subscript declared by an \(\text{overall}\) (next rule). It is imposed here to preserve the relation between \(\text{at}\) and \(\text{overall}\) given in 2.5.

\(^5\)This statement needs some interpretation, because locations in certain ranges may be identified with locations in others. For example, locations in a subrange may 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 \(\text{ExtBlockRange}\) can be identified with the corresponding locations of a \(\text{BlockRange}\) if the ranges have the same extent and process dimension. The \(\text{Range}\) class includes methods such as \(\text{isaligned()}\) that can be used to determine if two ranges are aligned, and thus logically share locations.
5.3. RULES FOR ACCESS TO DISTRIBUTED ARRAY ELEMENTS

5.3.1 Changes for ghost regions

If the array r appearing in rule 5 has ghost regions\(^6\), the range of allowed
subscripts is changed to

\[-x\.loExtension() ≤ e_r < x\.size() + x\.hiExtension()\]

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

\[
\text{at}(i = x[n]) \{ \\
\ldots a[e_0, \ldots, e_{r-1}, i \pm d, e_{r+1}, \ldots] \ldots \\
\}
\]

and the following requirement is added:

3. The expression \(\pm d\) is in the range

\[-a\.rng(r)\.loExtension() ≤ \pm d ≤ a\.rng(r)\.hiExtension()\]

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

5.3.2 A final rule for array element access

The rules on subscripts given in the last two subsections go a long way towards
ensuring a crucial requirement of HPJava, namely that a process may only access
locally held array elements. There are still odd cases—typically involving array
sections—where those rules are insufficient. Consider the pathological example
of Figure 5.1. The subscripts on the element reference \(b[j]\) are legal—\(j\) is
certainly a location in \(b\.rng(0)\) (which is equal to \(y\)). But, as illustrated,
the section \(b\) is localized to \(p/x[0]\)—the top row of processes in the figure—
whereas the \(ax\) construct specifies that the element assignments are performed
in the group \(p/x[N-1]\)—the bottom row of processes.

This kind of error can be excluded by the following rule:

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

If this rule is not satisfied a \(java\.lang\.NullPointerException\) will be thrown.

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

---

\(^6\)See section 7.3.1 for an example of a sequential dimension with ghost regions.
\textbf{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.
5.4 A recommendation for updating variables

We will define the concept of a *home group* for every kind of variable that can occur in an HPspmd code in an HPJava program.

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

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

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

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

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

The home group for a local variable is simply the active process group at the point where the variable is declared.

The home group for an instance variable is the active process group in effect when the parent instance is created.

The home group for an element of a standard Java array is the active process group in effect when the array is created.

The home group for a static field can be taken to be the set of processes in which the program is initiated\(^7\).

Now, 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 behavior through dependencies on values of global indexes in *overall* constructs, or values of locally held program variables) then we claim that all variables remain *coherent*. We say a variable is *coherent* if, at corresponding stages of execution of an HPspmd program, all processes in the home group of a variable hold identical values in their local copies of the variable\(^8\).

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 probably 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,\(^7\)

\(^7\) Because of the dynamic way Java classes are loaded and initialized, this definition is problematic. This may be another good reason for for avoiding static fields in HPJava programs.

\(^8\) These statements shouldn't exactly be taken as rigorous theorems. They haven't been proven, and the language is sufficiently complex it is almost certain somebody can find awkward cases to disprove them. But the general idea feels right.
the only one that doesn’t follow canonical style is the Monte Carlo program of Figure 3.10. In that program the variable \texttt{rand} has home group \texttt{p}, but it is updated inside nested \texttt{overall} constructs, where the active process group is \texttt{p}/i/j. Also, the initialization of \texttt{rand} involves a dependency on the \texttt{crd()} method of \texttt{Dimension}, which is intrinsically incoherent. All other variables in all other algorithmic examples are coherent\footnote{There is a short example in section 2.2 that uses the \texttt{crd()} inquiry, and therefore isn’t canonical.}. The canonical HPspmd style has a close affinity with the pure data parallel programming style of languages like HPF.
Chapter 6

A Distributed Array Communication Library

Many of the examples in this report use a communication library called Adlib. This library is not supposed to have a uniquely special status so far as the HPJava language is concerned. Eventually HPJava bindings for other communication libraries will probably be needed. For example, the Adlib library does not provide the one-sided communication functionality of libraries like the Global Arrays toolkit (www.emsl.pnl.gov:2080/docs/global/ga.html). It doesn't provide optimized numerical operations on distributed arrays like those in ScalAPACK (www.netlib.org/scalapack). And it doesn't provide highly optimized collective operations for irregular patterns of array access, like those in CHAOS (www.cs.umd.edu/projects/hpsl/chaos.html). These libraries (and others) work with distributed arrays more or less similar in concept to HPJava distributed arrays. Bindings to these or functionally similar libraries might one day be made available for HPJava. For now, this section summarizes essential features of the HPJava binding of Adlib.

The original Adlib was developed independently of the HPJava project, to support HPF translation. The first version of the library called Adlib¹ was implemented during the SHPF project (www.vcpc.univie.ac.at/information/-software/shpf) at Southampton University. The library was further developed during the PCRC project at Syracuse University (www.hpjava.org/pcrc). As part of the HPJava project it has been reimplemented again in Java.

All the collective functions in the HPJava version of Adlib are static methods of the class hpjava.adlib.Adlib. All may throw exceptions of type hpjava.adlib.AdlibException. This is an unchecked exception (it is a subclass of java.lang.RuntimeException), so catching these exceptions in the user program is optional. Uncaught exceptions will cause the program to abort, which, for better or worse, is fairly typical practice in parallel programming environments.

¹The name, Adlib, is due to John Merlijn.

81
6.1 Regular collective communications

There are three main families of collective operation in Adlib: regular communications, reduction operations, and irregular communications. It also provides a few I/O operations.

The regular communications are exemplified by the operations *remap()* , *writeHalo()* , *shift()* , and *cshift()* , introduced in earlier sections. The first of these, *remap()* , is a very characteristic example. The *remap()* method takes two distributed array arguments—a source array and a destination. These two arrays must have the same size and shape\(^2\) 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 communications necessary 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()* method is a static member of the *Adlib* class. Like most of the methods in Adlib, the *remap()* method is overloaded to apply to various ranks and types of array:

```java
void remap(int    [-1] destination, int    [-1] source)  
void remap(float  [-1] destination, float  [-1] source)  
void remap(double [-1] destination, double [-1] 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 to scalars:

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

For this family of operations the element-type overloading includes all Java *primitive* types, and the type *Object*. Object elements are communicated using the standard Java serialization mechanism, and the detailed interpretation of what it means to remap an array of objects (e.g. the extent to which referential integrity between the objects is preserved) follows from the semantics of those standard Java mechanisms\(^3\).

---

\(^2\) The *shape* of a multiarray or distributed array is the list of its extents, *(a.rng(0).size(), a.rng(1).size(), ...)*.

\(^3\) In a single call to a remap-like operation, a single *ObjectOutputStream* is used to serialize
6.1. REGULAR COLLECTIVE COMMUNICATIONS

The overloading with respect to rank is slightly more problematic, because there are infinitely many possible ranks, and for now the HPJava language has no way to express that an argument can have arbitrary rank. As currently implemented, we can summarize the available \texttt{remap()} signatures in the notation:

\begin{center}
\texttt{void remap(T \# destination, T \# source)}
\end{center}

where the variable \(T\) runs over all primitive types and \texttt{Object}, and the notation \(T \#\) means a multiarray of arbitrary rank, with elements of type \(T\).

This notation is quite terse, and perhaps cryptic at first sight, but you will need to get used to it to follow the rest of this chapter. Bear in mind that one should mentally expand this kind of prototype into a series of signatures similar to the ones given above.

You may notice that this shorthand signature contains slightly less information than the series given above, because it doesn’t impose as a compile-time constraint that source and destination arrays have the same rank. But as a matter of fact it is a more accurate reflection of the current implementation—although it is not possible to define multiarray parameters of unspecified rank \textit{within} the HPJava language (i.e. there is no such type as \(T \#\) in HPJava), the translator provides a special hook that allows methods of “externally defined” libraries—typically libraries written in standard Java—to behave as if they had such parameters.

There are four preconditions to a call to \texttt{remap()}:

1. All processes in the active process group must make the call, and they must pass \textit{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 (or, if the element types are \texttt{Object}, all objects referenced by elements of the source array must be assignable to elements of the destination array).

3. The arrays \texttt{source} and \texttt{destination} must not overlap—no element of \texttt{source} must be an alias for an element of \texttt{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 in the APG if the array’s distribution group is contained in the active process group. In other words, the requirement is that \textit{every copy of every element of both arrays is held on a processor involved in the collective operation.} \footnote{all objects sent from the local process to any given remote process. In the case where, say, two elements initially on the same process both hold references to a single object, and the two references are copied to different processes, referential integrity is explicitly \textit{not} preserved—the copied references are to different, local copies of the original object. But this is normal for Java serialization.}
If the second condition above is not satisfied an `hpjava.adlib.schedule.RankMismatchException` or an `hpjava.adlib.schedule.ShapeMismatchException` will be thrown. If the last condition is not satisfied an `hpjava.adlib.schedule.InaccessibleException` will be thrown.

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 in the APG, an exception occurs as described above.

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

```java
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 2.4. In general we have the signatures:

```java
void shift(T [[-]] destination, T [[-]] source,
           int shiftAmount)
```

and

```java
void shift(T # destination, T # source,
           int shiftAmount, int dimension)
```

with `T` defined as above. The first form applies only for one dimensional multiarrays. The second form applies to multiarrays of any rank. The `shiftAmount` argument, which may be negative, specifies the amount and direction of the shift. In the second form the `dimension` argument is in the range `0,…,R-1` where `R` is the rank of the arrays; it selects the array dimension in which the shift occurs. Again the source and destination arrays must have the same shape, but now there is an extra precondition—they must also be `identically aligned`
6.1. REGULAR COLLECTIVE COMMUNICATIONS

That is, their distribution groups must be identical and their corresponding ranges $x$, $y$ must be identical or at least satisfy the test $x$.isAligned$(y)$. By design, shift() implements a simpler pattern of communication than general remap(). The alignment relation allows for a more efficient implementation. The library incorporates runtime checks on alignment relations between arguments, where these are required. If the arguments of shift() are not correctly aligned an $\text{.hpjava.adlib.schedule.GroupMismatchException}$ or an $\text{hpjava.adlib.schedule.MisalignmentException}$ will be thrown.

The shift() operation does not copy values from source that would go past the edge of destination, and at the other extreme of the range elements of destination that are not targeted by elements from source are unchanged from their input value. The related operation cshift() is essentially identical to shift() except that it implements a circular shift, rather than an “edge-off” shift.

The function writeHalo() is applied to distributed arrays that have ghost regions. It updates those regions. The simplest versions have prototype

$$\text{void writeHalo}(T \# a)$$

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.

More general forms of writeHalo allow to specify that only a subset of the available ghost area is to be updated, or to select circular wraparound for updating ghost cells at the extreme ends of the array:

$$\text{void writeHalo}(T \# a, \text{int wlo} [], \text{int whi} [])$$

and

$$\text{void writeHalo}(T \# a, \text{int wlo} [], \text{int whi} [], \text{int [] mode})$$

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, dimension by dimension, whether to update in the normal way, leaving ghost edges at extreme edges of the arrays unwritten (value should be $\text{Adlib.EDGE}$), whether to update using circular wraparound ($\text{Adlib.CYCL}$), or whether to not update any ghost regions in this dimension at all ($\text{Adlib.NONE}$, equivalent to setting the corresponding elements of wlo, whi to zero).

Operation of writeHalo() is visualized in figure 6.1.

Finally we mention the function broadcast(), which is actually a simplified form of remap(). There are two signatures:

$$T \text{broadcast}(T [[\text{int [] source})$$
Figure 6.1: Illustration of the effect of executing the writeHalo function.

and

\[
T\ \text{broadcast}(T\ \text{source, Group root})
\]

The first form 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 to all members of the active process group, as in this fragment:

```java
int [i,-1] a = new int [i, y] ;

int n = 3 + Adlib.broadcast(a [[i0, i1]]) ;
```

The second form of broadcast() just takes an ordinary Java value as the source. This value should be defined on the process or group of processes identified by root. It is broadcast to all members of the active process group. Technically this version is only useful if the source expression is *incoherent*—see section 5.4—but it is quite handy in practical codes.

### 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 fairly large set of reduction operations, mimicking the reductions available as “intrinsic functions” in Fortran.

As a first example, the maxval operation simply returns the maximum of all elements of an array. It has prototypes

\[
\text{maxval}(t \# a)
\]
where \( t \) now runs over all Java numeric types—that is, all Java primitive types except boolean. The discovered result is broadcast to the active process group, and returned as the result of the \texttt{maxval()} method. Other reduction operations with similar interfaces are \texttt{minval}, \texttt{sum} and \texttt{product}. Of these \texttt{minval} is minimum value, \texttt{sum} adds the elements of \( a \) in an unspecified order, and \texttt{product} multiplies them together.

The boolean reductions:

\[
\begin{align*}
\text{boolean any(boolean } & \# a) \\
\text{boolean all(boolean } & \# a) \\
\text{int count(boolean } & \# a)
\end{align*}
\]

behave in a similar way: \texttt{any} returns true iff any element of \( a \) is true, \texttt{all} returns true iff all elements of \( a \) are true, and \texttt{count} returns a count of the number of true elements in \( a \).

The function \texttt{dotProduct} used in some earlier examples is also logically a reduction, but it takes two aligned 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 before multiplying elements. The prototype is

\[
t_3 \text{ dotProduct}(t_1 \# a, t_2 \# b)
\]

If either of \( t_1 \) or \( t_2 \) is a floating point type (\texttt{float} or \texttt{double}) the result type, \( t_3 \), is \texttt{double}. Otherwise the result type \( t_3 \) is long. The argument multiarrays must have the same shape and must be aligned. The result is broadcast to all members of the active process group. There is also a boolean form of \texttt{dotProduct}:

\[
\text{boolean dotProduct(boolean } \# a, \text{ boolean } \# b)
\]

This returns the logical “or” of all the pairwise logical “ands” of elements.

The methods \texttt{maxloc} and \texttt{minloc} return respectively the maximum and minimum values of all elements of an array—similar to \texttt{maxval} and \texttt{minval}—but also output the index tuple in the array at which the extreme value was found. The prototypical forms are:

\[
\begin{align*}
t \text{ maxval} & (\text{int } \# \text{ loc}, t \# a) \\
t \text{ minval} & (\text{int } \# \text{ loc}, t \# a)
\end{align*}
\]

where \( \text{loc} \) is an ordinary Java array of length \( R \), the rank of \( a \). On exit it contains the (broadcast) global subscripts of the extreme value.

For each of the simple reductions that combine all elements of an array into a single value, there is a corresponding “dimensional reduction”, which just reduces along a selected dimension of the input array, and outputs an array of values of rank one less than the input. The method \texttt{maxvalDim()}, for example, has the form:

\[
\text{void maxvalDim( } \# \text{ res, } t \# a, \text{ int dimension})
\]
The dimension argument takes a value, \( d \), in the range \( 0, \ldots, R - 1 \) where \( R \) is the rank of \( a \). The result array, \( res \), must have rank \( R - 1 \). It must be aligned with the input array \( a \), with replicated alignment in the \( d \)th dimension of \( a \). In other words, the distribution groups of \( a \) and \( res \) must be the same, and

\[
res.rng(i).isAligned(a.rng(i))
\]

for \( i < d \), and

\[
res.rng(i).isAligned(a.rng(i + 1))
\]

for \( d \leq i < R - 1 \). The reductions \( \text{minvalDim} \), \( \text{sumDim} \), \( \text{productDim} \), \( \text{anyDim} \), \( \text{allDim} \), \( \text{countDim} \) are defined in a similar way. The \( \text{maxloc} \) and \( \text{minloc} \) reductions have “dimensional” forms:

\[
\text{void maxlocDim}(f \# \text{res}, \text{int} \# \text{loc}, f \# a, \text{int} \text{ dimension})
\]

\[
\text{void minlocDim}(f \# \text{res}, \text{int} \# \text{loc}, f \# a, \text{int} \text{ dimension})
\]

where the array \( \text{loc} \) has the same rank and alignment as \( \text{res} \) (since the reduction is in a single dimension, only one index value—for the specified dimension—needs to be returned per extreme value). Currently there is no “Dim” form of \( \text{dotProduct}() \).

Finally all the numeric simple reductions and dimensional reductions all have “masked” variants. These take an extra boolean array aligned with the source array. For example

\[
\text{void maxval}(f \# a, \text{boolean} \# \text{mask})
\]

ignores all elements of \( a \) for which the corresponding element of \( \text{mask} \) is false.

A notable omission in the current version of the library is any facility for user-defined reduction operations. It also omits various arithmetic reductions that might seem natural in Java, such as bitwise, and, or, and xor. There is no fundamental reason for these omissions, and this might change in future releases.

### 6.3 Irregular collective communications

Adlib has some support for irregular communications in the form of collective \( \text{gather}() \) and \( \text{scatter}() \) operations. In general these methods have fairly unwieldy interfaces. Also their current implementation is far from optimal. If you find yourself using these methods extensively, it may be time to consider resorting to lower-level styles of parallel programming (as discussed for example in chapter 7). Or you may just need a different library...

The simplest form of the \( \text{gather}() \) operation for one-dimensional arrays has prototypes

\[
\text{void gather}(T \text{ [-] destination, } T \text{ [-] source,}
\]

\[
\text{int [-] subscripts)}
\]
The subscripts array should have the same shape as, and be aligned with, the destination array. In terms of pseudocode, the behavior of the gather operation is like:

\[
\text{for all } i \text{ in } \{0, \ldots, N-1\} \text{ in parallel do} \\
\quad \text{destination}[i] = \text{source}[\text{subscripts}[i]] ;
\]

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

\[
\text{for all } i \text{ in } \{0, \ldots, N-1\} \text{ in parallel do} \\
\quad a[i] = b[\text{fun}(i)] ;
\]

where \text{fun} is an arbitrary function, this can be expressed using Adlib as

\[
\begin{align*}
\text{int } [-1] \text{ tmp } &= \text{new int } [xj] \text{ on p ;} \\
\text{on(p)} &\quad \text{overall}(i = x \text{ for ;}) \\
\quad \text{tmp } [i] &= \text{fun}(i) ; \\
\text{Adlib.gather(a, b, tmp) ;}
\end{align*}
\]

where \( p \) and \( x \) are the distribution group and range of \( a \). The source array, \( b \), may have a completely unrelated mapping.

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

\[
\begin{align*}
\text{void } \text{gather}(T &\# \text{destination}, T [-1] \text{ source,} \\
&\text{int } \# \text{subscripts}) \\
\text{void } \text{gather}(T &\# \text{destination}, T [-,-] \text{ source,} \\
&\text{int } \# \text{subscripts0, int } \# \text{subscripts1}) \\
\text{void } \text{gather}(T &\# \text{destination}, T [-,-,-] \text{ source,} \\
&\text{int } \# \text{subscripts0, int } \# \text{subscripts1, int } \# \text{subscripts2})
\end{align*}
\]

Currently the highest rank of source array with a \text{gather()} method is 3\(^4\). The source and destination arrays can have different ranks (just as they have unrelated distribution formats). But the destination and subscripts arrays are all the same shape, and all are aligned with one another. The number of subscript array arguments is equal to the rank of the source array. If the rank of the destination array is \( R \), the rather complicated general behaviour of these methods is:

\[
\text{for all } i_0 \text{ in } \{0, \ldots, N_0 - 1\} \text{ in parallel do} \\
\quad \ldots \\
\text{for all } i_{R-1} \text{ in } \{0, \ldots, N_{R-1} - 1\} \text{ in parallel do} \\
\quad \text{destination}[i_0, \ldots, i_{R-1}] = \text{source}[\text{subscripts0}[i_0, \ldots, i_{R-1}], \\
\quad \text{subscripts1}[i_0, \ldots, i_{R-1}], \\
\quad \ldots] ;
\]

\(^4\)This could be increased by adding extra stub methods to the \text{Adlib} class, but eventually the excessive number of methods on this class becomes an issue.
CHAPTER 6. A DISTRIBUTED ARRAY COMMUNICATION LIBRARY

where \((N_0, \ldots, N_{R-1})\) is the shape of destination array.

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

\[
\text{void scatter}(T \ [\cdot] \ source, \ T \ [\cdot] \ destination, \\
\text{int} \ [\cdot] \ subscripts) 
\]

and it behaves like

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

Currently the HPJava version of Adlib does not support combining scatters, although these could be added in later releases.

All the gather() and scatter() operations take an optional final argument which is a boolean array acting as a mask, e.g.

\[
\text{void gather}(T \ [\cdot] \ destination, \ T \ [\cdot] \ source, \\
\text{int} \ [\cdot] \ subscripts, \ \text{boolean} \ [\cdot] \ mask)
\]

The mask should have the same shape as and be aligned with the subscript arrays. Assignment to a destination element is conditional on the value of the element of the mask associated with the subscripts.

6.4 I/O

Support for I/O in the current HPJava version of Adlib is rudimentary. In general any communications needed to input and output parallel data structures are left as the reponsibility of the HPJava programmer—such operations may, for example, be built up from standard Java I/O features, together with the broadcast and remap functions of Adlib. But to help you get started we provide a couple of “convenience methods”. The most interesting is aprintf(). This is modelled on the C printf() function. Its arguments are a control string and a list of distributed arrays. The effective prototypes are

\[
\text{void aprintf(String control, } T \ # \ \text{source})
\]

\[
\text{void aprintf(String control, } T_0 \ # \ \text{source0}, \\
T_1 \ # \ \text{source1})
\]

\[
\text{void aprintf(String control, } T_0 \ # \ \text{source0}, \\
T_1 \ # \ \text{source1}, \\
T_2 \ # \ \text{source2})
\]

For now a maximum of three input arrays is allowed (but this is just an interface issue). If more than one array is specified, all should have the same shape. But they can have any, unrelated distribution format.
6.4. I/O

The control string (like all arguments of Adlib methods) should be a “coherent” expression: it takes the same value for all processes in the active group. It has basically the same role and the same format as the control string of the printf() function in the C standard library. But the aprintf() operation performs its output “elementally”. If there is a single array argument it is as if a printf() function was called repeatedly, once for each element of the array. For example

```c
float [-l] a = new float [][x] ;
overall(i = x for ;)
  a [i] = 10.0F * i' ;

Adlib.aprintf("a[i] = %f\n", a) ;
```

would print out something like:

```
a [0] = 0.0
a [1] = 10.0
a [2] = 20.0
a [3] = 30.0
a [4] = 40.0
...
```

If there are two array elements, it is as if a printf() function was called once for each matching pair of array elements, passing two scalar arguments to the printf(). So

```c
int [-l] ix = new int [][x] ;
float [-l] a = new float [][x] ;
overall(i = x for ;) {
  ix [i] = i' ;
  a [i] = 10.0F * i' ;
}

Adlib.aprintf("a[%d] = %f\n", ix, a) ;
```

would print out something like:

```
a [0] = 0.0
a [1] = 10.0
a [2] = 20.0
a [3] = 30.0
a [4] = 40.0
...
```

Because it is quite common to want to print the current index value there is a special format sequence allowed in the control string to do just this: the

```
Adlib.aprintf("a[%R0] = %f\n", a) ;
```

produces the same output as the version above. The format %R0 interpolates the index value into the string without the need to initialize an extra array of integers.
All this works for multidimensional arrays as well. In a pseudo code notation, the general behaviour of `printf()` is like

```plaintext
for each i_0 in (0, ..., N_0 - 1) in sequence do
    ...
    for each i_{R-1} in (0, ..., N_{R-1} - 1) in sequence do
        printf(control, source0 [i_0, ..., i_{R-1}],
                source1 [i_0, ..., i_{R-1}],
                ...
)
```

where (N_0, ..., N_{R-1}) is the shape the arrays. The integer value i_0 is interpolated into the output wherever there is a `%R0` in the control string; the value i_1 is interpolated wherever there is a `%R1`; and so on (a field width modifier is also allowed with `%RD` format, just as with `%d` format). The imaginary elemental `printf` operation outputs to `System.out` on the root process of the active process group.

This code, for example:

```plaintext
float [[-,-]] a = new float [[x,y]]; 
overall(i = x for :)
    overall(j = y for :)
        a [i, j] = 10.0F * i + j;

Adlib.printf("a[%R0, %R1] = %f\n", a);
```

might print out something like:

```plaintext
a [0, 0] = 0.0
a [0, 1] = 1.0
a [0, 2] = 2.0
a [1, 0] = 10.0
a [1, 1] = 11.0
a [1, 2] = 12.0
```

where we assumed, for the sake of illustration, an unusually small 2 by 3 distributed array.

For large, possibly multidimensional, arrays one would probably prefer a more tabular layout. The `printf()` method provides one special format to support more creative line breaking. The `printf()` format `%N` by itself behaves exactly like the \n escape sequence—it inserts a newline into the output. But `%N` allows an integer modifier. This looks like a `printf()` field width modifier, but it is interpreted differently—it defines the frequency with which the newline is printed: if the value of the modifier is w, the newline is only printed in every wth elemental print operation. So if we replace the `printf()` call in the previous example with:

```plaintext
Adlib.printf("a[%R0, %R1] = %f  %N", a);
```
we get output something like

\[
\begin{align*}
& a [0, 0] = 0.0 & a [0, 1] = 1.0 & a [0, 2] = 2.0 \\
& a [1, 0] = 10.0 & a [1, 1] = 11.0 & a [1, 2] = 12.0
\end{align*}
\]

By specifying normal C-like field width modifiers on the %f format one could line up columns more beautifully, if so desired.

As a final trick, we can build the control string at run-time, allowing the number of columns to reflect the actual shape of the array:

\[
\text{Adlib.printf("%f %n + a.mg(1).size() + %n", a); }
\]

Now if the range \( y \) has size 8, say, the control string evaluates to "%f %8n", and a newline is printed after every 8 elemental outputs, i.e. we get a newline at the end of every row of the array, whatever the actual shape of the array. With a second modified %N format in the control string one might use similar tricks to insert a blank line to delimit the end of each "plane" of a three dimensional array. And so on.

The printf() method is useful for testing and demonstrating programs. It is less obvious that it is useful for production codes. We expect distributed arrays to be large, and in general it probably it doesn't make much sense to print thousands or millions of elements to standard output. With this in mind the current implementation has not been carefully optimized—it is quite slow.

For convenience there are also two much simpler methods in Adlib for printing a "global" String value. These are called printf() and println(). They have interfaces:

```
printf(String string)
println(String string)
```

All they do is output string to System.out on the root process of the active process group.

### 6.5 Schedules

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

Adlib provides a class of schedule objects associated with each of its communication functions. These schedules belong to the package hpjava.adlib.schedule. The classes generally have the same names as the static functions, with the first letter capitalized. Relevant type information (e.g. the type of argument
WriteHaloFloat writeHalo = new WriteHaloFloat(a);
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.

array elements) is usually appended to the end of the name. Each class has a
series of constructors with arguments identical to the method interfaces. Every
schedule class has one public method with no arguments called execute, which
executes the schedule. In some cases (e.g. reduction schedules) the execute
method returns a numeric value.

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

The complete set of schedule classes in the hpjava.adlib.schedule package
includes: RemapBoolean, RemapByte, RemapChar, RemapDouble, RemapFloat,
RemapInt, RemapLong, RemapObject, RemapShort, ShiftBoolean, ShiftByte,
ShiftChar, ShiftDouble, ShiftFloat, ShiftInt, ShiftLong, ShiftObject,
ShiftShort, WriteHaloBoolean, WriteHaloByte, WriteHaloChar, WriteHaloDouble,
WriteHaloFloat, WriteHaloInt, WriteHaloLong, WriteHaloObject, WriteHaloShort,
MaxvalByte, MaxvalChar, MaxvalDouble, MaxvalFloat, MaxvalInt, MaxvalLong,
MaxvalShort, MinvalByte, MinvalChar, MinvalDouble, MinvalFloat, MinvalInt,
MinvalLong, MinvalShort, SumByte, SumChar, SumDouble, SumFloat, SumInt,
SumLong, SumShort, ProductByte, ProductChar, ProductDouble, ProductFloat,
ProductInt, ProductLong, ProductShort, Any, All, Count, MaxlocByte, MaxlocChar,
MaxlocDouble, MaxlocFloat, MaxlocInt, MaxlocLong, MaxlocShort, MinlocByte,
MinlocChar, MinlocDouble, MinlocFloat, MinlocInt, MinlocLong, MinlocShort,
6.5. SCHEDULES

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 \textit{N-body} classical mechanics problem. As the name suggests, this problem is concerned with the dynamics of a set of \textit{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 \texttt{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 \texttt{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 [[-1]] f = new float [[x]], a = new float [[x]],
    b = new float [[x]], tmp = new float [[x]];

    ... initialize 'a' ...

    overall(i = x for ;) {
        f [i] = (float) 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);
        HPutil.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 approach 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 $\text{Rank}()$ and $\text{Sendrecvreplace}()$ from the mpiJava binding of MPI (discussed briefly on page 7 and in section 2.7). Figure 7.4 is a valid SPMD Java program and therefore a valid HPJava program. You can find a runnable version of this code in the hpdjk release under hpdjk/examples/PPHPJ/EpmpiNBody hpj. Unlike most examples in this report this code will only run under the multi-process model (see section 2.7)—currently there is no multithreaded version of the mpiJava mpi package available.
### 7.1. An Example

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
</table>
| a

![Diagram](image)

| b

![Diagram](image) |
|-----------------|

Iteration 0

| a

![Diagram](image) |
|-----------------|

| b

![Diagram](image) |

Iteration 1

| a

![Diagram](image) |
|-----------------|

| b

![Diagram](image) |

Iteration 2

| a

![Diagram](image) |
|-----------------|

| b

![Diagram](image) |

Iteration 8

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.
7.1. AN EXAMPLE

```java
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;
    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.

So this works, but 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—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. This code relies on the fact that the Dimension class representing a process dimension is actually a subclass of Range, so a process dimension can be used directly as a distributed range of an array.

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

for (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] = 0.0f;
            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);
        HPutil.copy(b, tmp);
    }
}

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

library functions like cshift(), instead of relatively clumsy functions like Sendrecv_replace().

● More esoterically, the program follows the canonical HPspmd style, described briefly in section 5.4. As defined in that section, all variables are coherent. The elements of the arrays in the program of Figure 7.4 are not coherent, because they take different values in each process, although their home group is the set of all processes. Respecting the canonical style may 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 we get a lot of new flexibility.

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 a *pair* of dimensions—a coordinate dimension plus a local subscript dimension.

If the subscript in a particular dimension of a section expression is the special symbol $\leftarrow$, 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; the local blocks of the original array are embedded in the the sequential dimension of the result. The two new dimensions appear consecutively in the signature of the result array, distributed dimension first.

Now we can combine the examples from Figures 7.1 and 7.5 of the last section. The version in Figure 7.6 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
could be written in elementary style as

```java
class Matmul {
  static 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++)
        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.
7.2. DIMENSION SPLITTING

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 [il] = 0.0F;
        b [il] = a [il];
    }

    // split 'x' dimensions:

    float [[-,*]] fs = f [[>]], as = a [[>]], bs = b [[>]];
    float [[-,*]] tmp = new float [[d, 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, i, 0);
        HPutil.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 :)
      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, B1]] ;
  Adlib.cshift(tmp, bs, -1, 1) ;
  HPutil.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:

```java
Proc p = new Proc1(4);
Dimension d = p.dim(0);
```

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

At this point one might feel inclined to abandon the idea of dimension splitting. These examples don’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.

We could give up and go home, but we won’t. Note that for a given range there is a fixed bound on the number of array elements held by any process. We can 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 $[50/4] = 13$. For the third example the amount of space allocated will presumably be enough to hold the parent distributed array—most likely $B$ is $[100/4] = 25$.

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

---

1 In this case the dimension-split version of $a$ is the same as the dimension-split version of the parent array $b$. An odd 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.
7.3. BLOCK PARAMETERS

Range $x = \text{new BlockRange}(60, d)$ ;
float $[\ldots] a = \text{new float}[[x]]$ ;

\begin{align*}
\text{Process 0} & \quad \text{Process 1} & \quad \text{Process 2} & \quad \text{Process 3} \\
0, 1, 2 & : & 10, 11, 12 & : \\
13, 14, 15 & : & 26, 27, 28 & : \\
29, 30, 31 & : & 39, 40, 41 & : \\
42 & : & & \\
\end{align*}

Range $x = \text{new CyclicRange}(60, d)$ ;
float $[\ldots] a = \text{new float}[[x]]$ ;

\begin{align*}
\text{Process 0} & \quad \text{Process 1} & \quad \text{Process 2} & \quad \text{Process 3} \\
0, 4, 8 & : & 40, 44, 48 & : \\
1, 5, 9 & : & 41, 45, 49 & : \\
2, 6, 10 & : & 52, 46 & : \\
7, 2, 11 & : & 43, 47 & : \\
\end{align*}

Range $x = \text{new BlockRange}(100, d)$ ;
float $[\ldots] b = \text{new float}[[x]]$ ;
float $[\ldots] a = b [[0 : 99 : 2]]$ ;

\begin{align*}
\text{Process 0} & \quad \text{Process 1} & \quad \text{Process 2} & \quad \text{Process 3} \\
0, 1, 2 & : & 11 & : \\
13 & : & 23 & : \\
25 & : & 36, 37 & : \\
39 & : & & \\
\end{align*}

Figure 7.9: Example embeddings of array elements in local blocks
an object of class `hpjava.lang.Block`, declared as:

```java
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.

(Unfortunately there is an overlap of terminology between the blocks of an arbitrary range, and the distribution format of one particular kind of range, viz. `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 the translation scheme for the `overall` construct, and this is a fairly natural way to illustrate its use.

Consider this fragment of HPJava:

```java
float [[-1]] a;

Range x = a.rng(0);

overall(i = x for :)
    a [il] = (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 the range `d` can be assumed to be a process dimension, so the role of the new `overall` construct is essentially a formality—`k` only has one location in each process, and the new `overall` doesn’t yield a local loop. Although the `k` subscript of `as` is required by the rules of the language, it is essentially does nothing, because there is only one locally held location. In fact the only real effect of the reduced construct is to change the active process group inside its body.

So 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, :]]`. 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
7.3. BLOCK PARAMETERS

SOURCE:

```plaintext
default [[-]] a;
    Range x = a.rng(0);
    overall(i = x for ;)
        a [i] = (float) i;
```

TRANSLATION:

```plaintext
default [[-]] a;
    Range x = a.rng(0);
    default [[-,*]] 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.subBas + b.subStp * l] =
                (float) (b.glbBas + b.glbStp * l);
```

Figure 7.10: Recursive translation of a simple overall construct.
be translated efficiently by a compiler using a strength reduction optimization (replacing the linear expressions with an incremented accumulation variable).

Because it involves reducing an overall construct to a kind of lower-level overall, we sometimes call this general scheme recursive translation. It's an interesting device formally, but note the actual HPJava translator doesn't work this way—it translates directly to Java, following the rules in Appendix A.

Note that technically the localBlock() inquiry is incoherent (see section 5.4). This can be fixed by adding 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 it uses the incoherent crd() inquiry internally.

### 7.3.1 Ghost regions and dimension splitting

If the distributed range has ghost extensions, this does not affect the values in the block description returned by localBlock(). These values describe the layout of elements associated with the "physical" portion of the array, not elements in the ghost region. In this case, however, the range of legal subscripts in the local sequential array dimension is increased. In the absence of ghost regions that range may be, for example, \(0, \ldots, B - 1\), where typically \(B\) would be \([N/P]\).

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 \(-w_{lo}, \ldots, B + w_{hi} - 1^2\). Figure 7.11 gives the recursive translation of an overall construct involving a shifted index. It assumes \(a\) has a suitable ghost extensions.

### 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.

---

2Note however that the size() inquiry applied to the associated collapsed range will still return the “physical” extent, \(B\).

3This 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:

```c
float [[-1]] a;

Range x = a.rng(0);

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

TRANSLATION:

```c
float [[-1]] a;

Range x = a.rng(0);

float [[-,*1]] 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 + l] = ... ;
```

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(1, u, s);
    for (int m = 0 ; m < b.count ; m++)
        as [k, b.sub_bas + b.substp * m] =
            (float) (b.glb_bas + b.glbstp * m) ;
}

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. HPJava does not attempt to conceal such things. Hence the underlying Java arrays should be available if they are needed.

The inquiry dat() can be applied to any HPJava multiarray or distributed 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 sequential multiarray. The detailed embedding is defined by the localBlock() inquiry on the distributed array ranges.

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

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

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

$$a.dat() \cdot [a.bas() + \sum_{r=0}^{R-1} i_r \times a.str(r)]$$

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 aa is a section defined by splitting dimension $r$ of an array a, to yield dimensions $r'$ and $r'+1$ of aa, then

$\text{Note:}$ These “memory” bases and strides are distinct from the subrange alignment parameters returned by similarly named inquiries on ranges.
- \texttt{as.bas()} is equal to \texttt{a.bas()}, and

- \texttt{as.str(r' + 1)} (the stride associated with the new sequential dimension)
  is equal to \texttt{a.str(r)}

We can use these formulae to further simplify the recursive translation given earlier. The new translation is given in Figure 7.14. This is quite similar to the scheme actually used by the HPJava translator.
7.4. REDUCTION TO JAVA ARRAYS

```java
Proc$p = new Proc$p(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;
    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:

```java
float [-] a;
Range x = a.rng(0);
overall(i = x for ;)
a [i] = (float) f[i];
```

TRANSLATION:

```java
float [-] a;
Range x = a.rng(0);
Block b = x.localBlock();
for (int i = 0 ; i < b.count ; i++)
a.dat()[a.bas() + (b.sub_bas + b.substp * i) + a.strb()] =
(float) (b.glb_bas + b.glbstp * i);
```

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

We have seen two complementary ways the HPJava language provides 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 be implemented using a library function. For example, for a two dimensional array of `float` the following procedure would do the job:

```java
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]]
}
```

Note that this is an *incoherent* function. The result array could not, for example, be passed as an argument to a collective method.
7.6 An extended example: prefix computation

In this section we will give a detailed 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 called a "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:

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

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

$$ r[0] = 0 $$

$$ r[i] = a[0] + \ldots + a[i-1]; \quad 1 \leq i < N $$

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.
### 7.6. AN EXTENDED EXAMPLE: PREFIX COMPUTATION

```java
static 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a₀</td>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>a₅</td>
</tr>
<tr>
<td>t</td>
<td>a₀</td>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>a₅</td>
</tr>
</tbody>
</table>

Iteration 1

<table>
<thead>
<tr>
<th>a</th>
<th>a₀</th>
<th>a₀+a₁</th>
<th>a₁+a₂</th>
<th>a₁+a₃</th>
<th>a₂+a₃</th>
<th>a₃+a₄</th>
<th>a₄+a₅</th>
<th>a₅+a₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>a₀</td>
<td>a₀+a₁</td>
<td>a₁+a₂</td>
<td>a₂+a₃</td>
<td>a₃+a₄</td>
<td>a₄+a₅</td>
<td>a₅+a₆</td>
<td></td>
</tr>
</tbody>
</table>

Iteration 2

<table>
<thead>
<tr>
<th>a</th>
<th>a₀</th>
<th>a₀+a₁</th>
<th>a₀+a₂</th>
<th>a₀+a₃</th>
<th>a₁+a₄</th>
<th>a₂+a₅</th>
<th>a₃+a₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>a₀</td>
<td>a₀+a₁</td>
<td>a₁+a₂</td>
<td>a₀+a₃</td>
<td>a₀+a₄</td>
<td>a₃+a₆</td>
<td></td>
</tr>
</tbody>
</table>

Iteration 3

Finally

Figure 7.16: Illustration of doubling algorithm for parallel prefix for N = 7.
static void blockPrefix(float [I] a) {
    Range x = a.rng(0);
    float [I,] as = a [[<]];
    Range z = as.rng(0);
    float [I] 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.subbas + b.substp * 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.subbas + b.substp * l;
            as [k, sub] += sum;
        }
    }
}

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

After stage 1

After stage 2

After stage 3

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

7.6. **AN EXTENDED EXAMPLE: PREFIX COMPUTATION**
The code given here will work for the ranges mentioned above, and subranges of these\(^5\). 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 schemes 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 may still be worth improving it. The operation in question might be a necessary step in larger program. 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 really 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 \(\text{ts}\), with zeros in positions for which there is no corresponding element of \(\mathbf{a}\). 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 \(\mathbf{a}\).

It is straightforward enough to implement this scheme in HPJava using dimension splitting. But 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 may not be much worse in practice.)

### 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” internal process dimension of size 1. Everything will work, although it would

---

\(^5\)At least for subranges with positive alignment stride. We will return to this issue in section 7.6.3
7.6. AN EXTENDED EXAMPLE: PREFIX COMPUTATION

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
<th>Process 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>a₀</td>
<td>a₁</td>
<td>a₀</td>
<td>a₁</td>
<td>a₂</td>
</tr>
</tbody>
</table>

Initially

<table>
<thead>
<tr>
<th>ts</th>
<th>0</th>
<th>a₄</th>
<th>0</th>
<th>a₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a₃</td>
<td>0</td>
<td>a₄</td>
<td>0</td>
</tr>
<tr>
<td>a₀</td>
<td>a₁</td>
<td>0</td>
<td>a₂</td>
<td>a₂</td>
</tr>
</tbody>
</table>

After copying elements of ‘a’ to ‘ts’

<table>
<thead>
<tr>
<th>ts</th>
<th>a₅</th>
<th>a₆</th>
<th>a₅+a₆</th>
<th>a₅+a₆</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a₃</td>
<td>0</td>
<td>a₄</td>
<td>a₄</td>
</tr>
<tr>
<td>a₀</td>
<td>a₁</td>
<td>a₀+a₁</td>
<td>a₀+a₁</td>
<td>a₀+a₁</td>
</tr>
</tbody>
</table>

After doing global prefix across rows

<table>
<thead>
<tr>
<th>exclusive prefix of last column</th>
<th>a₀+...+a₄</th>
<th>a₀+...+a₄</th>
<th>a₀+...+a₄</th>
<th>a₀+...+a₄</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a₀+...+a₂</td>
<td>a₀+...+a₂</td>
<td>a₀+...+a₂</td>
<td>a₀+...+a₂</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

After broadcasting exclusive prefix of last column

<table>
<thead>
<tr>
<th>ts</th>
<th>a₀+...+a₅</th>
<th>a₀+...+a₅</th>
<th>a₀+...+a₅</th>
<th>a₀+...+a₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₀</td>
<td>a₅</td>
<td>a₅</td>
<td>a₅</td>
<td>a₅</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
</tr>
</tbody>
</table>

After adding last-column-prefix to local column

<table>
<thead>
<tr>
<th>a</th>
<th>a₀+...+a₄</th>
<th>a₀+...+a₄</th>
<th>a₀+...+a₄</th>
<th>a₀+...+a₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₀</td>
<td>a₁</td>
<td>a₁</td>
<td>a₁</td>
<td>a₁</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
</tr>
<tr>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
</tr>
</tbody>
</table>

After copying elements of ‘ts’ back to ‘a’

Figure 7.19: Possible optimization of prefix for cyclic distributions: illustration for stride 2 subrange of cyclic range with extent 15. P = 5.
static void prefix(float [[-1] a) {
  Group p = a.grp();

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

    switch (x.format()) {
      case Range.DIST_DIMENSION:
        case Range.DIST_CYCLIC:

          doublingPrefix(a);
          break;

          default:
            blockPrefix(a);
          }
    }
  }
}

Figure 7.20: Optimized parallel prefix for any distribution format.

probably be more efficient to test for DIST_COLLAPSED and handle this in a separate, optimized subroutine.

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. 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. So stage 2 in Figure 7.17 could be replaced with something like⁶:

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. There is no need to change the individual block processing to take account of negative alignment strides: the localBlock() inquiry does this automatically. The substp field will be negative in this case.

⁶ 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 multiarrays.
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 `hpjava.lang.Block` object, defining the layout of distributed array elements in that process. The `Block` class was described on page 110.

One might expect the `block()` inquiry should be well-defined for any valid process coordinate. But there are some array alignment options that wouldn't be handled particularly well if that was true. Consider the example array sections illustrated in Figure 7.21, which presume the declarations:

```java
    Process p = new Process(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 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 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 striped 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 `hpjava.lang.-`
Range x = new BlockRange(20, d);
float [[-]] b = new float [[x1]];
float [[-]] a = b [[5 : 10]];

Range x = new CyclicRange(20, d);
float [[-]] c = new float [[x1]];
float [[-]] a = c [[0 : 18 : 2]];

Range x = new BlockRange(20, d);
float [[-]] b = new float [[x1]];
float [[-]] a = b [[19 : 0 : -1]];

Figure 7.21: Example sections with unusual coordinate ranges.
7.7. NON-LOCAL BLOCKS

Triplet, declared as:

```java
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,   lo + stp,   lo + 2 × stp,   ...,   lo + \left(\frac{hi - lo}{stp}\right) + 1 × 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:

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

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

```
EMPTY.count = 0;
EMPTY.sub_bas = 0;
EMPTY.sub_stp = 1;
EMPTY.glob_bas = 0;
EMPTY.glob_stp = 1;
```

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

As in the case of `localBlock`, there are overloaded versions of `crds()` and `block()` that take `l, u, s` arguments.
Appendix A

Translation scheme

This appendix describes a basic translation scheme for HPJava. The translation scheme is given in pedantic detail; in effect it represents the most complete definition of the HPJava language.

A.1 Preliminaries

A.1.1 On multiarray types

The most general multiarray type signature is:

\[ T \left[ \left[ \text{attr}_0, \ldots, \text{attr}_{R-1} \right] \right] \text{ bras} \]

where \( T \) is any Java type other than an array type, \( R \) is the rank of the multiarray, each term \( \text{attr}_r \) either consists of a single hyphen, -, or a single asterisk, *, and the term \( \text{bras} \) is a string of zero or more bracket pairs, \([\)]. \) This signature represents the type of a multiarray whose elements have Java type

\[ T \ \text{bras} \]

In this appendix we will use “multiarray” as a general term including sequential multiarrays and distributed arrays.

In standard Java, arrays are not considered to be objects, but they effectively have a class—a class representing an array type. We prefer to avoid making the statement “multiarrays 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 infeasibly hard; would people expect us, for example, to integrate the complex runtime inquiries
on HPJava multiarrays into some extended version of the Java reflection API? Or into the Java Native Interface, JNI? Such fundamental extensions to the Java platform look impractical.

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 multiarray types, the associated 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 A.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.

To avoid these problems an HPJava multiarray is not considered to have any class. Multiarray types are treated as a new kind of reference type in HPJava, separate from standard Java reference types.

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

```java
class Level {
    public Level(float [-,1] arr) {this.arr = arr ; }

    public float [-,1] arr ;
}
...

Level [] stack = new Level [S] ;
for (int l = 0 ; l < S ; l++) {
    Group p ;
    Range x, y ;
    ...
    stack [l] = new Level(new float [[x, y]] on p) ;
}
```

So the fact that multiarrays cannot be treated as normal objects is usually a minor inconvenience, not a fundamental limitation.

### A.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. Nevertheless, the early phases of compilation will presumably apply transformations similar to the ones described in this appendix.

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. Other classes may use sequential multiarrays, but not distributed arrays or parallel constructs.

Methods that do use parallel HPJava syntax have some special properties. They require a well-defined active process group (APG) at their point of invocation. So in general it is problematic (although not impossible) to invoke parallel HPJava code from a piece of ordinary Java. These problems reflect real 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 uses HPJava parallel extensions and code that does not. It introduces a special interface, hpjava.lang.HPspmd, which must be implemented by any class that uses the parallel syntax.

---

1Incidentally, 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 (directly or indirectly) implements the HPSmpd interface as an HPSmpd class. Any other class is a non-HPSmpd class. Likewise, an interface that extends the HPSmpd interface is an HPSmpd interface, and any other interface is a non-HPSmpd interface. We define HPSmpd code to be any statement or expression that includes the following:

- an on construct,
- an overall construct,
- an at construct,
- a distributed array creation expression (i.e., a multiarray constructor specifying any distributed ranges, or having an on clause),
- an invocation of an HPSmpd method, or
- creation of an instance of an HPSmpd class.

Now it is required that HPSmpd code only appear in methods and constructors defined in HPSmpd classes². This restriction is quite specific, and in particular it forbids HPSmpd code in initialization expressions for fields, or in static or instance initialization blocks, even if these are within HPSmpd classes—our current feeling is that the implementation complexity needed to make the active process group available in these contexts isn’t warranted by the modest advantages it provides.

To discourage invocation of HPSmpd code from non-HPSmpd code, the HPJava translator imposes the following limitations:

1. If a class or interface inherits a method with the same signature from more than one of its superclasses and superinterfaces, either all declarations of the method must be in HPSmpd classes and interfaces, or all declarations of the method must be in non-HPSmpd classes and interfaces. So an inherited method can always be unambiguously classified as an HPSmpd method or a non-HPSmpd method.

2. An HPSmpd class or interface may not override a non-HPSmpd method.

Ordinary Java code (not processed by the HPJava translator) may access members of HPSmpd classes, but this requires detailed knowledge of the transformations the HPJava translator applies to function signatures³ and distributed array variables, and in general this facility should be used very cautiously. The HPJava translator itself will prevent code in non-HPSmpd classes from creating instances of, or invoking methods from, HPSmpd classes. Of course there is no restriction in the other direction—an HPSmpd class can freely use any non-HPSmpd Java class, limited only by the normal accessibility rules.

²According to these rules it is allowed for a non-HPSmpd code to hold a reference to a distributed array, although it is limited in what operations it can perform on the array. Also note that non-HPSmpd code can manipulate sequential multiarrays unrestrictedly.

³We will use the term function to cover both methods and constructors.
It should be very clear that the distinction between HPspmd and non-HPspmd classes is orthogonal to the discussion in section A.1.1 about the types of distributed arrays themselves. Here we are essentially concerned with what kind of code is allowed to use distributed arrays. HPspmd classes are unequivocally Java classes, whereas distributed array types are not classes of any kind.
A.2 Static Semantic Checking

A.2.1 Subscripting

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

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

the expression \( a \) must be invariant in the \texttt{overall} construct.

Proving that an arbitrary expression is invariant throughout a particular block of code can be a very hard problem for a compiler. The block of code (our \texttt{overall} construct) may, for example, contain method invocations. If the expression we are analysing involves fields or array elements, it is not generally possible to be prove that the method invocation does not change the value of the expression as a side-effect. On the other hand, if one \texttt{failed} to prove that expressions like \( a \) above were invariant, the loss of efficiency in the translated code might be quite spectacular.

For these reasons the translator imposes the following, fairly sweeping, restrictions \textit{[Actually these aren't implemented in the current translator—neither are the run-time checks that should be associated with subscripting multiarrays and distributed arrays.]}:

- Every distributed-array-valued variable in an HPJava program implicitly has the \texttt{final} attribute.

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

- We impose an ad hoc rule that the array reference expression in a distributed array element reference or an array section expression must be a \texttt{simple identifier}.

This identifier can refer to a local variable or a field of the current class (or of a superclass or lexically enclosing class). Again the actual inconvenience
A.2. STATIC SEMANTIC CHECKING

to the programmer due to this restriction is fairly minor. It seems only to enforce common practices. If, however, code in one object really does need to subscript a distributed-array-valued field in another object (say), the array reference just needs to be copied to a distributed-array-valued local variable prior to subscripting.

We can now very cleanly enforce the original requirements on subscripting in overall and at constructs, by applying a simple static check that if a distributed index $i$ appears in a subscript of a distributed array $a$, the variable $a$ should have been declared outside the scope of $i$, and (if it is a blank final) $a$ should not be assigned inside the scope of $i$. The Java constraints on assignments to blank final make the latter check easy.
A.3 Pre-translation

The current HPJava translator has two major translation phases. The first phase, pre-translation, reduces the input HPJava program to some other, equivalent HPJava program, coded in a restricted subset of the full HPJava language. The translation phase proper follows pre-translation, and converts the intermediate HPJava program to a standard Java program. Thus the restricted form emitted by the pretranslator should be suitable for processing in the translation phase. This restricted form, which is described in the following subsection, disallows certain complex expressions involving multiarrays. The pretranslator reduces these complex expressions to simpler expressions, by introducing temporaries if necessary.

While straightforward in principle, in general the simplification of expressions induces non-trivial reorganizations of declarations and control constructs appearing in the program. These transformations are documented in detail later in this section. It turns out that the separation of the pre-translator from the translator helps keep issues related to intricacies in the Java base language (which are largely the province of the pre-translator) cleanly separated from issues related to translation of the extended syntax of HPJava (which are the province of the translator).

A.3.1 Restricted Form

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

- **Composite multiarray expressions are:**
  - a) multiarray creation expressions,
  - b) multiarray-valued method invocation expressions,
  - c) multiarray section expressions, and
  - d) multiarray restriction expressions.

- **Simple expressions are**
  - a) Constant expressions.
  - b) Local variable references.
  - c) Fields referenced using the syntax of a simple identifier, \( f \), assuming the referenced field is not volatile. The current HPJava translator also treats fields referenced through the syntax
    - \( \text{this}.f \),
    - \( \text{super}.f \), or
    - \( T.f \), where \( T \) is a type name,
  as being simple, assuming the referenced field are not volatile.
A.3. PRE-TRANSLATION

   d) The keyword this.

Now the restrictions on expressions are:

1. A composite multiarray expression may only appear as the right-hand-side of a top-level assignment (or, in the case of multiarray-valued method invocation, as a standalone statement expression). Here, a top-level assignment is defined to be an assignment expression appearing as a statement expression in a program block. This specifically excludes assignments appearing in for-loop headers. Also note that a variable declarator with an initializer is not an assignment expression, so the restricted form does not allow composite multiarray expressions to appear as initializers in declarations.

2. All of the following must be simple expressions:
   a) the target object in accesses to multiarray-valued fields,
   b) the boolean expression in a multiarray-valued conditional expressions, and
c) the array operand in array-restriction expression,
d) the array operand of multiarray inquiry operations (rng(), grp(), etc);
   e) the range and group parameters of multiarray creation expressions,
   f) the range expression in the header of overall and at constructs.

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

Figures A.2 and A.3 illustrate the process of expression simplification. The source program is a slightly esoteric example involving array restriction, taken from section 4.8. In this example pre-translation shifts three kinds of composite array expression—multiarray creation, array section and array restriction—into top-level assignments. In the case of the array creation expressions it simply has to split the array declarations with initializers to blank declarations plus an assignment. In the case of the sections and restrictions, it has to introduce temporaries.

Besides simplifying variables, the pre-translation phase also breaks up multiarray declarations that declare several variables, outputting a series of declarations of individual multiarray variables. This slightly simplifies the work of the translation phase.

\[\text{There is one situation in which a general array expression would cause a problem here, and that is expressions like } (a = e).\text{grp(). All components of the expression } a = e \text{ have side effects, but the translation scheme of Figures A.39 and A.40 would cause some components of the expression to be improperly discarded. If preferred, this situation could be addressed by a more specific rule. [This should be changed. It's probably more natural to handle this by marking multiarray-valued assignment subexpressions as composite expressions.]}\]
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));

      double [[-, -]] c = new double [[x, y]] on p;
      double [[-, -]] a = new double [[x, z]] on p;
      double [[-, -]] b = new double [[x, y]] on p;

      ... initialize ‘a’, ‘b’

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

Figure A.2: Example source program prior to pre-translation.

A.3.2 Expression simplification

For the most part, the transformations needed to simplify expressions and reduce
a program to restricted form are straightforward. For completeness, however,
we will outline in some detail the algorithm applied by the current HPJava
translator. We should remark that much of the discussion in this section is
not very specific to HPJava; similar strategies could be used in other Java
preprocessors that need to conservatively simplify expressions, as a pre-requisite
to translating some subset of those expressions.

One principle of the HPspin model is that parts of a program that are
written entirely in the base language—not using special HPspin features—
should, as far as practical, be compiled in exactly the same way as if they
appeared in a local sequential program. We will take this requirement as a
guiding heuristic in defining our transformations. In a rather literal and mi-
crocosmic way, we interpret the heuristic to mean that sub-expressions that are
pure Java expressions—not involving multiarrays—should, as far as possible, be
copied unchanged to the translated program. Ideally the generated Java pro-
gram should not break up these subexpressions by introducing temporaries.
(Necessarily, a “pure Java” sub-expression may as a whole be precomputed and
stored in a temporary, if it was originally embedded in an expression node that
also included special HPJava terms.)

In our scheme certain subexpressions are singled out according to the sim-
plification rules. These are expressions that must be precomputed, to meet
the requirements of a subsequent translation phase. In general, if the these
subexpressions are precomputed, dependence relations may force some other,
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))

  double [[-,-]] c;
  c = new double [[x, y]] on p;

  double [[-,-]] a;
  a = new double [[x, z]] on p;
  double [[-,-]] b;
  b = new double [[z, y]] on p;

  ... initialize 'a', 'b'

  overall(i = x for ;)
    overall(j = y for ;) {
      double [[-]] _$t1;
      _$t1 = a [[i, :]]

      double [[-]] _$t2;
      _$t2 = _$t1 / j;

      double [[-]] _$t3;
      _$t3 = b [[:, j]]

      double [[-]] _$t4;
      _$t4 = _$t3 / i;

      c [[i, j]] = Adlib.dotProduct(_$t2, _$t4);
    }
}

Figure A.3: Example source program after pre-translation.
“ordinary” subexpressions to be precomputed as well.

**Special subexpressions**

We will identify three classes of expressions that need special treatment: *composite expressions*, *multiply referenced values*, and *multiply referenced variables*.

- A composite expression is an expression that requires some preliminary statements to be executed before the expression can be evaluated. Typically composite expressions are special expressions of the extended language. In general, the pretranslation scheme will treat composite expressions differently according to whether they appear as the right-hand side of a top-level assignment (or as standalone statement expressions), or anywhere else. In the former cases it not usually necessary for the *pre-translation phase* to introduce temporaries.

- A multiply referenced value is any expression—possibly an ordinary Java expression—that appears in a context requiring the translated code to use the value of the expression more than once. Often multiply referenced values are subexpressions of composite expressions.

- A multiply referenced variable is an expression that has the syntax of a program variable, and appears in a context that means the translation scheme will use the expression more than once, *with at least one of these uses being a definition* (for example, one of the uses may be as the left-hand side of an assignment). In the scheme described here, the only multiply referenced variables that need special treatment are those on the left-hand side of compound assignments (`+=`, etc).

The singled-out expressions in the current HPJava translation scheme are listed in Figure A.4. To a large extent this table encapsulates the HPJava-specific features of the pre-translation scheme described in this section. With minor changes here and there, the rest of the specification of the pretranslator could probably be adapted easily to any source-to-source translator that was adding new aggregate types to the base Java language.

**Schematic algorithm**

A generic version of the algorithm for simplification—applicable to most of the expressions in the language—is given in Figure A.5. The algorithm is written as a recursive function, `simplify`. The main input to this function is an expression term `e`, which is modified by the function. Along with the modified version of `e`, the `simplify` function outputs a sequence of statements, `INITS`, which declares and initializes any temporaries that were introduced in the simplification of `e`. It also outputs “access sets” for `e` and `INITS`.

The algorithm traverses sub-expressions in *right to left* order. If a sub-expression must be precomputed, the associated code will have to be moved in front of sibling sub-expressions currently to its left. If there are no dependencies
Composite expressions:

1. Multiarray creation expressions.
3. Multiarray section expressions.
4. Multiarray restriction expressions.

Multiply referenced values:

1. The target object in accesses to multiarray-valued fields.
2. The boolean expression in a multiarray-valued conditional expressions.
3. The multiarray operand in array element reference and array section expression\(^6\).
4. The multiarray operand in array-restriction expression.
5. The array operand of multiarray inquiry operations (\texttt{rng()}, \texttt{grp()}, etc).
6. The range and group parameters of multiarray creation expressions.
7. The range expression in the header of \texttt{overall} and \texttt{at} constructs.

\(^6\)Assuming the language enforces the restriction discussed in section A.2.1—
that the target of multiarray subscripting be the name of a final variable—the pretranslating will not need to do any simplification to ensure this operand is simple.

Figure A.4: Special expressions in the current HPJava translation scheme.
Figure A.5: Schematic algorithm for expression simplification.
between the moved sub-expression and its left siblings, this is fine. But if there
is a dependency, the sibling expression must also be precomputed, to preserve
the original order of evaluation.

Dependencies are detected using access sets. These record uses and defini-
tions of individual variables, or categories of variables, within expressions. The simplify function computes these access sets on the fly and returns two of them
in the values ACCESS and INIT_ACCESS. The set ACCESS is the set of pro-
gram variable accesses in the translated expression $e'$. The set INIT_ACCESS
is the set of program variable accesses in the “precomputation” code—code for
initialization of temporaries defined in INIT (both sets exclude accesses to
temporaries themselves).

At the time a subexpression is visited, the variable INIT_ACCESS holds
accesses in “precomputation” code already generated by subexpressions to the
right of the current expression. If a dependency is discovered between the trans-
lated version of the current expression and the code we already know must move
“to the left”, the whole of the current expression must also “move to the left”,
to be precomputed before the code already scheduled to move.

Multiply referenced values and composite expressions are handled in different
parts of the algorithm, but currently they are a treated in exactly the same way.

Data structures

In the pseudocode of Figure A.5 code fragments are written as Java-like strings
of program text (with + as the concatenation operator). In the practical imple-
mentation these are be manipulated as nodes in an abstract syntax tree.

Access sets will probably be implemented as two separate variable sets: the
set of uses and the set of defs. A straightforward (possibly inefficient) approach
is to implement each of these two sets using five components:

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

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

To complete the simplify algorithm we still need to define, for each kind
of expression in the language, the relevant set of subexpressions (enumerated
in the foreach), and also—for dependence analysis within expressions—the
“direct accesses associated with” this kind of expression.
Expression nodes

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

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

- Variable expressions:
  - named expression
  - field reference with expression prefix
  - field reference with super prefix
  - array element reference (multiarray or Java array)

- “Wildcards”:
  - method invocations
  - class instance creation (named class or anonymous class)

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

Table A.1 gives suitable definitions for the “direct accesses” in these nodes. In these tables LHS stands for the left-hand-side operand of an assignment; RHS stands for the right-hand-side operand.

No other kinds of expression involve direct accesses, and most other kinds of expression have reasonably obvious sub-expression lists. For example, a binary expression node has two subexpressions and no direct accesses; a triplet subscript $l : u : s$ contributes three subexpressions to a section expression; an (ordinary Java) array initializer term is treated as a kind of expression with a variable number of subexpressions (and no direct accesses); etc, etc. In the case of anonymous class instance creation, the algorithm only treats constructor arguments as subexpressions; it does not treat the class body as a subexpression. Instead the “direct accesses” conservatively include all variables that could be used or modified in any anonymous class body. Note that unlike a method invocation, the body of an anonymous class can use local variables of the enclosing environment (at least if the local variables have the final modifier).

There are a handful of expression types for which the generic form of simplify in Figure A.1 is not straightforwardly applicable. The first set of odd cases
<table>
<thead>
<tr>
<th>Expression node</th>
<th>Direct <code>uses</code></th>
<th>Direct <code>defs</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple assignment with named expression on LHS</td>
<td></td>
<td>Variable descriptor (local or field) associated with LHS.</td>
</tr>
<tr>
<td>Simple assignment with field reference as LHS</td>
<td></td>
<td>Field descriptor associated with LHS.</td>
</tr>
<tr>
<td>Simple assignment with array element on LHS</td>
<td></td>
<td>“All array elements”.</td>
</tr>
<tr>
<td>Compound assignment with named expression as LHS</td>
<td>Variable descriptor (local or field) associated with LHS.</td>
<td>Variable descriptor (local or field) associated with LHS.</td>
</tr>
<tr>
<td>Compound assignment with field reference as LHS</td>
<td>Field descriptor associated with LHS.</td>
<td>Field descriptor associated with LHS.</td>
</tr>
<tr>
<td>Compound assignment with array element on LHS</td>
<td>“All array elements”.</td>
<td>“All array elements”.</td>
</tr>
<tr>
<td>Autoincrement/decrement with named expression as operand</td>
<td>Variable descriptor (local or field) associated with operand.</td>
<td>Variable descriptor (local or field) associated with operand.</td>
</tr>
<tr>
<td>Autoincrement/decrement with field reference as operand</td>
<td>Field descriptor associated with operand.</td>
<td>Field descriptor associated with operand.</td>
</tr>
<tr>
<td>Autoincrement/decrement with array element as operand</td>
<td>“All array elements”.</td>
<td>“All array elements”.</td>
</tr>
<tr>
<td>Named expression, not the variable of an update.</td>
<td>Variable descriptor (local or field) associated with named expression.</td>
<td></td>
</tr>
<tr>
<td>Field reference, not the variable of an update.</td>
<td>Field descriptor associated with field reference.</td>
<td></td>
</tr>
<tr>
<td>Array element reference, not the variable of an update.</td>
<td>“All array elements”.</td>
<td></td>
</tr>
<tr>
<td>Any method invocation</td>
<td>“All fields”; “All array elements”.</td>
<td>“All fields”; “All array elements”.</td>
</tr>
<tr>
<td>Class instance creation (named class)</td>
<td>“All fields”; “All array elements”.</td>
<td>“All fields”; “All array elements”.</td>
</tr>
<tr>
<td>Anonymous class instance creation</td>
<td>“All variables”.</td>
<td>“All fields”; “All array elements”.</td>
</tr>
</tbody>
</table>

Table A.1: Direct accesses of various expression nodes, for `simplify` algorithm.
is the update operations: simple assignments, increment and decrement operations, and compound assignments. These are singled out because they must handle variable expressions in a careful way. The second set is of odd cases is the expressions with “conditional evaluation”. These are: conditional expression \((b ? x : y)\), conditional “and” \((b && c)\), and conditional “or” \((b || c)\). These are special because evaluation of some subexpressions is conditional on the results of others. Computation of initialization expressions lifted from a conditionally evaluated subexpression must be controlled by the same condition that controls evaluation of the original subexpression.

**Updates**

To deal with updates, we first define, in Figure A.6, an abstracted *visitChildren* algorithm, convenient particularly for variable expressions (i.e. expressions that can appear on the left-hand-side of an assignment). The subexpressions of a variable expression are mostly self-explanatory. They are:

- If the variable is a named expression, the subexpressions are:
  - the prefix of named expression, if it is non-null and is an expression (not a type).

- If the variable is a field reference with a general expression prefix, the subexpressions are:
  - the prefix expression of the field reference.

- If the variable is a field reference with a `super` prefix, there are no subexpressions.

- If the variable is an array element reference, the subexpressions are:
  - the target array of the element reference,
  - the integer subscripts, if any, of the element reference, and
  - the “shift” expressions in any shifted-distributed-index subscripts of the element reference.

The *visitChildren* function is applied to the left-hand-side of a simple assignment in Figure A.7. An important point is that one does not recursively apply the *simplify* algorithm to the expression representing the variable itself; the recursion is on the *subexpressions* of the variable expression. The case of an autoincrement or autodecrement expression is also straightforward, and it is given in A.8. There is a complication in the case of compound assignments; they will be discussed in the next section.

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

\[ v \oplus= e \]
visitChildren(e, INITS, ACCESS, INITS ACCESS) {
    INOUT: e
    OUT: INITS
    INOUT: ACCESS, INITS ACCESS

    INITS = ""

    foreach (subexpression e_i of e, enumerated right to left) {
        simplify(e_i, INITS_i, ACCESS_i, INITS ACCESS_i)

        if(e_i is a multiply referenced value that is not simple, or
            e is a multiply referenced variable, and e_i is not simple, or
            there is a dependency between ACCESS_i and INITS ACCESS)
            { // t is a new temporary name; T is type of e_i
                INITS = INITS_i + " T t ; t = e_i ; " + INITS
                e_i = "t"
                INITS ACCESS = INITS ACCESS_i \cup INITS ACCESS \cup ACCESS_i
            } else {
                INITS = INITS_i + INITS
                ACCESS = ACCESS_i \cup ACCESS
                INITS ACCESS = INITS ACCESS_i \cup INITS ACCESS_i
            }
    }
}

Figure A.6: Standalone function for visiting subexpressions of a variable.
simplify(e, INITS, ACCESS, INITS_ACCESS) {
    INOUT: e
    OUT: INITS, ACCESS, INITS_ACCESS

    // e is a simple assignment, e_l = e_r.
    simplify(e_l, INITS, ACCESS, INITS_ACCESS)
    visitChildren(e, INITS, ACCESS, INITS_ACCESS)
    INITS = INITS + INITS
    ACCESS = ACCESS ∪ \{def of LHS variable\}
}

Figure A.7: Simplification of simple assignment expressions.

simplify(e, INITS, ACCESS, INITS_ACCESS) {
    INOUT: e
    OUT: INITS, ACCESS, INITS_ACCESS

    // e is one of e+++, e, --, ++e_r, or --e_r.
    ACCESS = {}
    INITS_ACCESS = {}

    visitChildren(e, INITS, ACCESS, INITS_ACCESS)
    ACCESS = ACCESS ∪
    \{use of operand variable, def of operand variable\}
}

Figure A.8: Simplification of autoincrement/decrement expressions.
if \( e \) includes a subexpression \( e_i \) that must be precomputed, and \( e_i \) may define \( v \) (i.e. subexpression \( e_i \) has a def of \( v \) in its access set). The underlying problem is that there is then an anti-dependence from \( v \) to \( e_i \) and an output dependence from \( e_i \) to \( v \). Hence one cannot move the code for \( e_i \) without splitting the compound assignment.

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

\[
v = v \oplus e
\]

thus allowing the use of \( v \) on the RHS to be precomputed. Moreover if \( v \) itself is a non-trivial expression, you may then have to mark its subexpressions as being multiply-referenced to avoid their repeated computation.

It is fairly difficult to imagine realistic code in which this situation would arise, but here is a contrived example:

```plaintext
a [n++] += sum(blackBox())
```

We will suppose `blackBox()` is a method that returns a multiarray result, but whose behavior is otherwise unknown. The method `sum()` might add the elements of a multiarray. On the left-hand-side, \( a \) is an array—a local variable, say—and we will assume \( n \) is also a local variable.

According to our simplification rules, the composite multiarray expression `blackBox(a)` must be precomputed, and assigned to a temporary. So a naive transformation might be:

```plaintext
_type1 = blackBox();
a [n++] += sum(_type1);
```

where `_type1` is the name of a temporary. In general this transformation is illegal, because `blackBox()` could modify the value of `a [n]` as a side-effect (even if the array reference `a` is a local variable, the referenced array may be accessible by other methods). In this situation, the correct semantics of the original code is to increment the initial, unmodified value of `a [n]`. But our naively transformed code increments the value modified by `blackBox()`.

To solve this problem, we should precompute the initial value of the left-hand-side of the assignment, something like

```plaintext
_type1 = a [n++];
_type2 = blackBox();
a [n++] = _type1 + sum(_type2);
```

Of course this still isn't correct due to the repetition of the subscript expression in the variable `a [n++]`. A correct transformation is:

```plaintext
_type1 = n++;
_type2 = a [_type1];
_type3 = blackBox();
a [_type1] = _type2 + sum(_type3);
```
simplify(e, INITS, ACCESS, INITS, ACCESS) {
    INOUT: e
    OUT: INITS, ACCESS, INITS, ACCESS

    // e is a compound assignment, \( e_l \oplus e_r \).
    simplify(e_r, INITS, ACCESS, INITS, ACCESS)
    if(INITS, ACCESS contains a def of LHS variable) {
        Mark \( e_l \) as a "multiply referenced variable"
        visitChildren(e_l, INITS_l, ACCESS, INITS, ACCESS)
        // t is a new temporary name; T is type of \( e_l \)
        INITS = INITS_l + " t t ; t = e_l ; " + INITS
        e = "e_l = t \oplus e_r"
    } else {
        visitChildren(e_l, INITS_l, ACCESS, INITS, ACCESS)
        INITS = INITS_l + INITS
    }

    ACCESS = ACCESS \cup \{ use of LHS variable, def of LHS variable \}
}

Figure A.9: Simplification of compound assignment expressions.

An algorithm is given in Figure A.9. Note that \( e_l \) may be marked as a multiply referenced variable. Such expressions were mentioned in section A.3.2. The effect of marking a variable expression in this way is to alter the behavior of the visitChildren function defined in Figure A.6, causing it to treat all the subexpressions, \( e_i \), as if they are multiply referenced values.

Expressions with conditional evaluation

Figures A.10, A.11, and A.12 give simplification schemes for the three kinds of expression in the language that require conditional evaluation of some subexpressions. Temporaries introduced in these subexpressions must themselves be initialized conditionally. So we need more complex initialization code, involving if statements.

In Figure A.10 we choose to replace any conditional expression returning a multiarray by an assignment to a temporary within an if statement. As an
simplify(e, INITS, ACCESS, INITS.ACCESS) {
    INPUT: e
    OUT: INITS, ACCESS, INITS.ACCESS

    // e is a conditional expression, e₀ ? eᵣ : eᵓ.

    simplify(eᵣ, INITSᵣ, ACCESSᵣ, INITS.ACCESSᵣ)
simplify(eᵣ, INITSᵣ, ACCESSᵣ, INITS.ACCESSᵣ)
simplify(eᵓ, INITSᵓ, ACCESSᵓ, INITS.ACCESSᵓ)

    INITS.ACCESS =
        INITSᵣ.ACCESSᵣ \cup INITSᵣ.ACCESSᵣ \cup INITSᵓ.ACCESSᵓ
ACCESS = ACCESSᵣ \cup ACCESSᵣ \cup ACCESSᵓ

    if (e is a multiarray or
        INITSᵣ is non-empty or INITSᵣ is non-empty) {

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

        INITS = INITSᵓ + "T t ;
            if(eᵓ) {
                + INITSᵓ + "t = eᵓ ;
            } else {
                + INITSᵣ + "t = eᵣ ;
            }"

        e = "e"

        INITSᵣ.ACCESS = INITSᵣ.ACCESS \cup ACCESS
ACCESS = \emptyset
    }
else
    INITS = INITSᵓ
}

Figure A.10: Simplification of conditional expressions.
optimization one may anticipate this transformation, and arrange to have the

simplify algorithm treat \( e_j \) and \( e_f \) as if they were originally on the right-hand
sides of a top-level assignments. This will avoid the introduction of unnecessary
temporaries if either of these subexpressions is composite.

On exact exceptions

Of course we want our transformations to be conservative in the sense that they
strictly preserve the meaning of the input program. The rules we give for trans-
formation of source code, based on the simplify algorithm just described, would
probably be considered conservative in “traditional” languages. They may not
be considered conservative in transforming strictly Java-like languages because
they do not preserve the exact model of exceptions—they allow reorderings of
code that could be detected by inspecting the values of program variables after
an exception occurred.

In our framework this shortcoming could be remedied by modelling the
throwing of an exception as a use of “all variables”. Any expression that could
throw any exception (e.g. an arithmetic division, a subscripting operation, a
derreferencing operation, a method invocation) would be considered to directly
read the value of all variables. An intuitive justification for this approach is that
throwing an exception can cause control to pass to a catch or finally block
that might read essentially any program variable that might have been assigned
in the expression. In any case, the effect is that an operation that may throw an
exception will never be reordered relative to a subexpression that might assign
any variable.

Actually we are converting HPJava to Java, not Java to Java. HPJava is
not exactly Java, and it is not a clear a priori requirement that it must treat
exceptions in the rigid Java way. For now the HPJava translator follows the
unmodified scheme described in this section.

A.3.3 Applying the simplify algorithm

In a Java or HPJava program, expressions can appear in the context of various
kinds of statement. This section enumerates (in gory detail) the transformations
of the input program that follow from simplification of those expressions.

Statement expressions

To start with the easiest case, consider the statement

\[ e ; \]

where \( e \) is an expression. Suppose \( \text{INITS} \) and \( e' \) are the results of applying the
simplify algorithm to \( e \). If \( e' = e \), i.e. no simplification is required, the input
simplify(e, INITs, ACCESS, INITs\_ACCESS) { 
  \textbf{INPUT}: e 
  \textbf{OUT}: INITs, ACCESS, INITs\_ACCESS 
  // e is a conditional or expression, \(e_1 \lor e_2\).
  simplify(e_2, INITs_2, ACCESS_2, INITs\_ACCESS_2) 
  simplify(e_1, INITs_1, ACCESS_1, INITs\_ACCESS_1) 
  \textbf{INITs\_ACCESS} = INITs\_ACCESS_1 \cup \textbf{INITs\_ACCESS}_2 
  ACCESS = ACCESS_1 \cup ACCESS_2 
  if(INITs_2 is non-empty) { 
    // t is a new temporary name
    INITs = INITs_1 + " boolean t; 
    if(e_1) 
      t = true; 
    else {" 
      \textbf{INITs}_2 + " t = e_2 ;" 
    }" 
    e = "t" 
    \textbf{INITs\_ACCESS} = \textbf{INITs\_ACCESS} \cup ACCESS 
    ACCESS = \{\} 
  } 
  else 
    INITs = INITs_1 
}

Figure A.11: Simplification of conditional or expressions.
simplify(e, INITS, ACCESS, INITS\_ACCESS) {
  INPUT: e
  OUT: INITS, ACCESS, INITS\_ACCESS

  // e is a conditional and expression, e₁ \&\& e₂.

  simplify(e₂, INITS₂, ACCESS₂, INITS\_ACCESS₂)
simplify(e₁, INITS₁, ACCESS₁, INITS\_ACCESS₁)

  \[\text{INITS\_ACCESS} = \text{INITS\_ACCESS₁} \cup \text{INITS\_ACCESS₂}\]
  \[\text{ACCESS} = \text{ACCESS₁} \cup \text{ACCESS₂}\]

  if \(\text{INITS₂}\) is non-empty) {

    // t is a new temporary name

    INITS = INITS₁ + "boolean t ;
    if(e₁) {
      + INITS₂ + "t = e₁ ;"
    }
    else
      t = false ;"

    e = "t"

    \[\text{INITS\_ACCESS} = \text{INITS\_ACCESS} \cup \text{ACCESS}\]
    ACCESS = Ø
  }
  else
    INITS = INITS₁

  }

Figure A.12: Simplification of conditional and expressions.
A.3. PRE-TRANSLATION

statement is unchanged. Otherwise, it transformed to

\[
\{
INITS
\epsilon^i ;
\}
\]

The case of a statement expression appearing in the initialization or con-
tinuation section of a for-loop header needs some special treatment—see page
160.

Local variable declarations
Consider a local variable declaration of the form

\[
m \ T \ V_0 \ I_0, \ldots, V_{n-1} \ I_{n-1} ;
\]

where \( m \) is an optional final modifier, \( T \) is a type, each term \( V_i \) is a simple
identifier, \( v_i \), followed by zero or more bracket pairs, \( [ \), and each optional
initializer clause \( I_i \) is either empty, or a term \( \epsilon = h_i \), where \( h_i \) is an expression
or an array initializer. We must have \( n \geq 1 \).

For each variable declarator

\[
V_i = h_i
\]

with a non-empty initializer, apply the simplify algorithm to \( h_i \), yielding results
\( \text{INITS}_i \) and \( h_i' \).

If \( T \) is a multiarray type we unconditionally break the declaration into a
series of transformed, single-variable declarations, \( D_i \):

\[
D_0 \\
\ldots \\
D_{n-1}
\]

where the form of \( D_i \) will be described below.

If \( T \) is not a multiarray type, and all \( \text{INITS}_i \) are empty, the whole declaration
is unchanged in the transformed code.

In any other case, suppose \( h_r \) is the first variable initializer for which \( \text{INITS}_i \)
is not empty. We will take the transformed declaration to be:

\[
m \ T \ V_0 \ I_0, \ldots, V_{r-1} \ I_{r-1} ;
\quad D_r \\
\ldots \\
D_{n-1}
\]

with the \( D_i \) defined as follows.

If \( \text{INITS}_i \) is empty and \( h_i \) is not a composite expression, then \( D_i \) is just:

\[
m \ T \ V_i \ I_i ;
\]
Otherwise $D_i$ is:

$$m \ T \ V_i ;$$

$$\text{INIT}_i$$

$$v_i = e_i^j$$

Here, if $h_i^j$ is not an array initializer, $e_i^j$ is just equal to $h_i^j$. If $h_i^j$ is an array initializer, $e_i^j$ is the array creation expression:

$$\text{new } U_i \ h_i^j$$

where $U_i$ is the array type of $v_i$.

If $h_i$ is composite expression $e_i$, one may anticipate this transformation, and arrange to have the simplify algorithm treat $e_i$ as if it was originally on the right-hand side of a top-level assignment. This will avoid the introduction of an unnecessary temporary.

The case of a local variable declaration appearing in the initialization section of a for-loop header may need some special treatment—see page 160.

**Field declarations**

Expressions can appear as initializers in field declarations. If these expressions need simplification, and the field is declared in a class, the field initialization can be moved into an instance initializer block (if the field is an instance variable) or a static initializer block (if the field is a class variable) in the class. The case of a field declared in an interface will be considered at the end of this section.

We visit every field declaration in a source class in textual order. Consider a field variable declaration of the form

$$ms \ T \ V_0, \ldots, V_{n-1} \ L_{n-1} ;$$

where $ms$ is an optional list of modifiers, and other declarations are as in section A.3.3. We assume this is an instance variable (respectively static variable) declaration. Visit each variable declarator in this declaration in exactly the same way as described in the preceding section on local variable declarations. The field declaration is processed in essentially the same way. The only significant change is in the definition of the $D_i$.

If $\text{INIT}_i$ is empty, then $D_i$ is just:

$$ms \ T \ V_i ;$$

Otherwise, we break the declaration into an uninitialized single-variable declaration and an instance initializer block:

$$ms \ T \ V_i ;$$

$$\{$$

$$\text{INIT}_i$$

$$v_i = e_i^j ;$$

$$\}$$
A.3. PRE-TRANSLATION

(respectively static initializer block:

\[
\begin{align*}
& ms \ T \ V_i \ ; \\
& \text{static} \ \{ \ \\
& \quad \text{INITS,} \\
& \quad v_i = e'_i \ ; \\
& \}
\end{align*}
\]

).

**Interfaces.** The fields of interfaces must be static and final. These fields are often used as to hold constants, but the Java language does not impose any special restrictions on what kind of expressions can appear as variable initializers in interfaces. In particular the variable initializer may require simplification according to our rules.

We cannot perform the precomputation and assignment of the field in an initializer block of the interface, because this kind of initializer is not allowed in interfaces. However we can create a nested private, static member class in an interface, and compute variable initializers in static initializer blocks of the nested class:

So, if the initializer of any field, \( v_i \), of an interface requires simplification, we create a nested class \( C \). Add a field of the same name, type, and initial value in \( C \). Transform the declaration of the class field as described in the previous section. Then replace the initializer of the interface field \( v \) with \( C.v \).

**if, switch, and synchronized statements**

Consider the statement

\[
\text{if}(e) \ S
\]

where \( e \) is an expression and \( S \) is a statement. As usual, \( \text{INITS} \) and \( e' \) are the results of applying the simplify algorithm to \( e \). If \( e' = e \), i.e. no simplification is required, the transformed version is just

\[
\text{if}(e) \ S'
\]

where \( S' \) is the pre-translated version of \( S \). Otherwise the transformed version is

\[
\{ \quad \text{INITS} \\
\quad \text{if}(e') \ S' \\
\}
\]

The if-else, switch and synchronized statements follow exactly the same pattern.
while statements

We consider the statement

\[
\text{while}(e)\ S
\]

where \(e\) is an expression and \(S\) is a statement. Again let \(\text{INITS}\) and \(e'\) be the results of applying the \textit{simplify} algorithm to \(e\). If \(e' = e\), the transformed version is just

\[
\text{while}(e)\ S'
\]

where \(S'\) is the pre-translated version of \(S\). Otherwise the transformed version is

\[
\text{while(true)}\ {\ \\
\{}\ \\
\ \ \ \ \ \text{INITS} \ \\
\ \ \ \ \text{if}(!e')\ \text{break} ; \ \\
\ \ \ \} \ \\
S' \ \\
\}
\]

do statements

We consider the statement

\[
\text{do}\ S\ \text{while}(e)\ ;
\]

where \(e\) is an expression and \(S\) is a statement. Again let \(\text{INITS}\) and \(e'\) be the results of applying the \textit{simplify} algorithm to \(e\). If \(e' = e\), the transformed version is just

\[
\text{do}\ S'\ \text{while}(e)\ ;
\]

where \(S'\) is the pre-translated version of \(S\). Otherwise the transformed version is

\[
\text{do}\ {\ \\
\{}\ \\
\ \ \ \ S' \ \\
\ \ \ \ {\ \\
\ \ \ \ \ \ \text{INITS} \ \\
\ \ \ \ \text{if}(!e')\ \text{break} ; \ \\
\ \ \ \} \ \\
\}\ \text{while(true)}\ ;
\]

for statements

The for statement is quite complicated. Consider

\[
\text{for}(a; b; c)\ S
\]
where $a$ is either a single declaration or a (possibly empty) comma-separated list of statement expressions, $b$ is an expression, $c$ is a (possibly empty) comma-separated list of statement expressions, and $S$ is a statement.

Suppose $a$ is the local variable declaration:

$$
\begin{array}{llll}
m & T & V_0 & I_0, \ldots, V_{n-1} & I_{n-1}
\end{array}
$$

where definitions are as in the section beginning page 157. Apply the transformation described in that section to the statement "$a;$". If this transformation does not produce any changes, and $T$ is not a multiarray type, the transformed version of the for construct is:

$$\text{for}(a; \ B; \ C) \ T$$

where $B$, $C$ and $T$ will be defined below. Otherwise, if the transformed version of "$a;$" is $A$, we may take the transformed version of the for construct to be:

$$
\begin{array}{l}
\{ \\
\quad A \\
\quad \text{for}(\ ; \ B; \ C) \ T \\
\}
\end{array}
$$

Suppose, alternatively, that $a$ is the possibly-empty list of expressions:

$$a_0, \ldots, a_{n-1}$$

where $n \geq 0$. Again, apply the simplify algorithm to each expression in this list, and assume now that $\text{INITS}_i$ and $a'_i$ are the results of applying the algorithm to $a_i$. If the simplify algorithm doesn’t change any of the expressions, and none of the $a_i$ are multiarrays, the transformed version of the for construct is again:

$$\text{for}(a; \ B; \ C) \ T$$

Otherwise, we may take the transformed version of the for construct to be:

$$
\begin{array}{l}
\{ \\
\quad \text{INITS}_0 \\
\quad a_0 \ ; \\
\quad \ldots \\
\quad \text{INITS}_{n-1} \\
\quad a'_{n-1} \ ; \\
\quad \text{for}(\ ; \ B; \ C) \ T \\
\}
\end{array}
$$

Now we need to define $B$, $C$ and $T$. We also introduce the symbols $s_b$ and $s_c$ to stand for possibly-empty statements added at the start and end of the loop body.
Apply the simplify algorithm to \( b \), and let \( \text{INITS} \) and \( b' \) be the results. If the algorithm doesn’t change \( b \), then

\[
B = b
\]

and the statement \( s_b \) is empty. If the algorithm does change \( b \), then

\[
B = \text{true}
\]

and the statement \( s_b \) is:

\[
\{ \\
\text{INITS} \\
\text{if} (! (b')) \text{ break;} \\
\}
\]

Assume \( c \) is the possibly-empty list of expressions:

\( c_0, \ldots, c_{p-1} \)

Apply the simplify algorithm to all these expressions. If the algorithm doesn’t change any \( c_i \), and none of the \( c_i \) is a multiarray, then

\[
C = c
\]

and the statement \( s_c \) is empty. If it does change some \( c_i \), or any \( c_i \) is a multiarray, then \( C \) is empty and we will take the statement \( s_c \) to be:

\[
\{ \\
\text{INITS}_0 \\
\quad c'_0 ; \\
\quad \ldots \\
\text{INITS}_{p-1} \\
\quad c'_{p-1} ; \\
\}
\]

where now \( \text{INITS}_i \) and \( c'_i \) are the results of applying the algorithm to \( c_i \).

Finally, if \( s_b \) and \( s_c \) are both empty, the value of \( T \) is just \( S' \), the pre-translated version of \( S \). Otherwise the block \( T \) has the form

\[
\{ \\
\quad s_b \\
\quad S' \\
\quad s_c \\
\}
\]
A.3. PRE-TRANSLATION

**return and throw statements**

Consider the statement

```
return e ;
```

where e is an expression. As usual, INITs and e' are the results of applying the simplify algorithm to e. If e' = e, i.e. no simplification is required, the statement is untransformed. Otherwise the transformed version is

```
{ 
  INITs  
  return e' ;  
}
```

The throw statement follows exactly the same pattern.

**Explicit constructor invocation statements**

Explicit constructor invocations appear as the first statement in the body of some other constructor. Typically they invoke a constructor of a superclass, although they may also invoke an alternative constructor of the current class. The syntax is either

```
super(e_0, \ldots, e_{n-1}) ;
```

or

```
this(e_0, \ldots, e_{n-1}) ;
```

or

```
e_0 . super(e_1, \ldots, e_{n-1}) ;
```

The last form may be used if the superclass is an inner class.

Devising transformations of the source program that allow for simplification of expressions in explicit constructor invocations is complicated by two restrictions. The first is that no statement in the body of the constructor can appear *before* the invocation. Ultimately any scheme that allows for simplification of composite expressions will require the declaration of auxiliary variables *before* the explicit constructor invocation is processed. At first sight this looks impossible.

There is at least one solution, however. Java allows a constructor to "dispatch" some initializations to another constructor in the same class by using the *this* form of explicit constructor invocation. If we divide the function of the original constructor across *two* constructors, a formal parameter of the second constructor can fulfill the role of the required auxiliary variable. The first constructor does the precomputations; the formal parameter of the second constructor is the name through which precomputed results are accessed—the name of the temporary, so to speak.

For this to work our "first" constructor must somehow embed the precomputations in a *this* explicit constructor invocation. It is fairly clear that putting
general initialization code in the explicit constructor invocation of the first constructor will require invocation some sort of auxiliary method. At this point the second restriction alluded to above becomes important: in Java the subexpressions $e_0, \ldots, e_{n-1}$ that appear in explicit constructor invocations are considered to be evaluated in a static context. This means that they cannot, for example, access instance variables or instance methods of the object being created. (They can, however, use static variables and methods of the current class, its superclasses, and its lexically enclosing classes.)

The static context means that any auxiliary method invoked cannot be an instance method of the current class. However the method must have access to the static context of this class. It could be a static method (or, say, a constructor of a nested static class). But we have to consider the possibility that the current class is an inner class; such static members cannot be defined in inner classes.

Luckily Java allows us to embed a definition of a method within the explicit constructor invocation itself; but using an anonymous class creation expression. This gives us access to the static context of the original class, while avoiding the problems of invoking instance methods of the original class.

In the following discussion our “first constructor” will be called the stub constructor. The “second constructor” will be called the private constructor.

To begin, suppose $C$ is a class which has a constructor of the form

$$
m\ C(m_0\ U_0\ V_0, \ldots, m_{r-1}\ U_{r-1}\ V_{r-1})\ \text{throws}\ ts\ \{\ 
\text{super}(e_0, \ldots, e_{n-1})\ ;
S
\}
$$

where $m$ is an optional access modifier, each $m_i$ is an optional final modifier, each term $U_i$ is a type, each term $V_i$ is a simple identifier, $v_i$, followed by zero or more bracket pairs, [$]$, $ts$ is a list of exceptions in the optional throws clause, each $e_i$ is an expression (which we will assume has type $T_i$), and $S$ is a sequence of statements. The other two forms of explicit constructor invocation can be handled by straightforward modifications to our basic scheme, which will be indicated as we go.

In a trivial way we can generalize the simplify algorithm to apply to an ordered list of expressions (treat the ordered list like a non-composite expression with no direct accesses, and the elements of the list like singly-referenced subexpressions). Apply this algorithm to the ordered list of expressions $[e_0, \ldots, e_{n-1}]$, yielding $INITS$ and $[e'_0, \ldots, e'_{n-1}]$. If $INITS$ is non-empty we transform the program as follows.

To support the transformation we will need to introduce an auxiliary “tuple class” used to store multiple results of the precomputation. This class can have
the general form:

    class A {
        A(S_0 f_0, \ldots, S_{m-1} f_{m-1}) {
            this.f_0 = f_0;
            \ldots
            this.f_{m-1} = f_{m-1};
        }

        final S_0 f_0;
        \ldots
        final S_{m-1} f_{m-1};
    }

where the \( S_i \) are the types of the \( m \) fields \( f_i \). In our case \( m \) will be \( n + r \), the fields \( f_0 \) to \( f_{n-1} \) will be used to store the values of the expressions \( e_0 \) to \( e_{n-1} \) (so \( S_0 \) to \( S_{n-1} \) should be \( T_0 \) to \( T_{n-1} \)), and the fields \( f_n \) to \( f_{m-1} \) will be used to store the values of the parameters \( \tau_0 \) to \( \tau_{r-1} \) of the original constructor after the \( e_i \) have been evaluated (so the types \( T_n \) to \( T_{m-1} \) will be the types of those parameters)\(^5\).

The new private constructor of \( C \) now has the form:

    private C(A a) throws ts {
        super(a.f_0, \ldots, a.f_{n-1});

        m_0 U_0 V_0 = a.f_n;
        \ldots
        m_{r-1} U_{r-1} V_{r-1} = a.f_{m-1};

        S'
    }

where \( S' \) is the pre-translated version of \( S \). If the original explicit constructor invocation had the alternate form

\[
\text{this}(e_0, \ldots, e_{n-1});
\]

the explicit constructor invocation in the new private constructor would be

\[
\text{this}(a.f_0, \ldots, a.f_{n-1});
\]

\(^5\)Exactly where \( A \) is declared is not very important to the following discussion. It could be declared as a top-level class, but that would have the disadvantage of polluting the global namespace of classes. The current HPJava translator introduces \( A \) as a private, static member class of the most closely lexically enclosing class of \( C \) that is not an inner class. If \( C \) is not an inner class, \( A \) will be a member of \( C \) itself.
If it had the form
\[ e_0 \cdot \text{super}(e_1, \ldots, e_{n-1}) \];
the form in the new private constructor would be
\[ a \cdot f_0 \cdot \text{this}(a \cdot f_1, \ldots, a \cdot f_{n-1}) \];

The new stub constructor has the form
\[ m C(m_0 U_0 V_0, \ldots, m_{r-1} U_{r-1} V_{r-1}) \text{ throws ts} \{
\text{this(new Object)} \{
  A \text{ eval}(m_0 U_0 V_0, \ldots, m_{r-1} U_{r-1} V_{r-1}) \text{ throws ts} \{
    \text{INITS}
    \text{return new A}(e'_0, \ldots, e'_{n-1}, v_0, \ldots, v_{r-1}) ;
  \}
}\text{. eval}(v_0, \ldots, v_{r-1}) ;\}
\}
\]

As promised, the this explicit constructor invocation includes an anonymous 
class creation expression that declares a method eval() in which the precomputation 
occurs. This method is immediately invoked to return a tuple object 
containing the results of the precomputation (and the values of the formal 
parameters of the constructor).

Noteworthy features of this transformation are:

- The expressions in the body of eval() are evaluated in a context 
esentially indistinguishable from that of the original explicit constructor 
invocation. They experience the full static context of C. If C is a local 
class, final local variables visible in its enclosing block are also visible 
inside eval(). The only difference is that the visible \( v_i \) variables are new 
variables. However their values are copies of those of the original variables.

- If evaluation of any \( e_i \) modifies values of any of the parameters \( v_i \), the 
value of \( v_i \) visible in \( S' \) is the modified value, as it should be.

Note Added. It now appears that in the JDK reference implementation of 
Java the interpretation of "static context" for a general expression in an 
exPLICIT constructor invocation is different to the static context experienced in 
the body of an anonymous class instance creation expression embedded in an 
exPLICIT constructor invocation. General expressions can (sometimes, under precise 
conditions we don’t clearly understand) access enclosing instances, whereas a 
special rule says specifically that anonymous class instance creation expressions 
embedded in explicit constructor invocations can not. So the above translation 
will not work if the original explicit constructor invocation contains a legitimate 
(?) reference to an enclosing instance. The correct rules for static context in 
inner classes are not very clear to us, and for now the HPJava translator follows 
scheme described above. With significant extra work, other interpretations of 
the static context can be accomodated.
on statements
We consider the statement
\[ \text{on}(p) \ S \]
where \( p \) is an expression and \( S \) is a statement. Pre-translation follows the same pattern as for if statements, see section A.3.3.

at and overall statements
We consider the statement
\[ \text{at}(i = x \ [n]) \ S \]
where \( x \) and \( n \) are expressions and \( S \) is a statement.

Apply the simplify algorithm to the ordered list \([x, n]\) (in this case treating \( x \) as a multiply reference value) and let the results be \( \text{INITS} \) and \([x', n']\). If \( \text{INITS} \) is empty the transformed version of our at statement is just
\[ \text{at}(i = x \ [n]) \ S' \]
where \( S' \) is the pre-translated version of \( S \). Otherwise the transformed version is
\[
\begin{cases}
\text{INITS} \\
\text{at}(i = x' \ [n']) \ S'
\end{cases}
\]

The overall statement follows the same pattern, if the triplet \( t \) in
\[ \text{overall}(i = x \ for \ t) \ S \]
is treated as a generalized kind of expression.
A.4 Basic translation

After the pre-translation phase reduces a program to a suitable restricted form, the translation phase proper begins. This section describes the basic, unoptimized form of this translation.

A.4.1 Translation functions and schemas

We will specify several translation functions. The detailed definitions will be presented through a series of schema in the following subsections. First we give some general definitions.

A function, \( T[e] \), on expression terms returns the result of translating an expression \( e \), assuming that the expression is not a multiarray.

Translation functions for multiarray-expressions are more complicated. In section A.3.1 we defined a subset of composite multiarray-valued expressions. The remaining non-composite multiarray-valued expressions are:

a) multiarray-valued local variable access,
b) multiarray-valued field access,
c) assignment expression, in which the left-hand operand is a multiarray-valued variable.

(this assumes that the pretranslator eliminates multiarray-valued conditional expression, as specified in Figure A.10).

The composite expressions only appear in restricted contexts and do not have translation functions in their own right (instead they are effectively handled as part of the translation of a top-level assignment statements). For non-composite multiarray-valued expressions there are \( 2 + R \) separate parts of the evaluation: \( T_{dat}[e], T_{has}[e] \) and \( T_0[e], \ldots, T_{R-1}[e] \), where \( R \) is the rank of the array. The interpretation of these separate terms will be given in the following sections.

Finally the translation function for statements, \( T[S; p] \), 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

\[
\text{SOURCE:} \\
S
\]

\[
\text{TRANSLATION:} \\
S'
\]

should be read more precisely as

\[
\text{SOURCE:} \\
s \equiv S
\]

\[
\text{TRANSLATION:} \\
T[s; apg] \equiv S'
\]
SOURCE:

\[ T \left[\left[ \text{attr}_0, \ldots, \text{attr}_{R-1} \right] \right] \text{bras} \ a = e ; \]

TRANSLATION:

\[ T \left[\begin{array}{c}
\end{array}\right] \text{bras} \ a'_{\text{dat}} = T_{\text{dat}}[e] ;
\]

\[ \text{ArrayBase} \ a'_{\text{bas}} = T_{\text{bas}}[e] ;
\]

\[ \text{DIMENSION\_TYPE}(\text{attr}_0) \ a'_0 = T_0[e] ;
\]

\[ \ldots
\]

\[ \text{DIMENSION\_TYPE}(\text{attr}_{R-1}) \ a'_{R-1} = T_{R-1}[e] ;
\]

where:

- \( T \) is a Java type,
- \( R \) is the rank of the declared array,
- each term \( \text{attr}_i \) is a single hyphen, -, or a single asterisk, *,
- the term \( \text{bras} \) is a string of zero or more bracket pairs, \([]\),
- the identifier \( a \) is the name of the declared array in the source program,
- the expression \( e \) is an optional initializer in the source program,
- \( a'_{\text{dat}} = \text{TRANSJD}(a) \),
- \( a'_{\text{bas}} = \text{TRANSJD\_BAS}(a) \),
- \( a' = \text{TRANSJD\_DIM}(a, r) \), and
- the macros \( \text{TRANSJD}, \text{TRANSJD\_BAS}, \text{TRANSJD\_DIM} \) and \( \text{DIMENSION\_TYPE} \) are defined in the text.

Figure A.13: Translation of a multiarray-valued variable declaration.
### A.4.2 Translating variable declarations

The general scheme for translating declaration of a field or local variable holding a multarray reference is illustrated in Figure A.13. The single variable in the source program is converted to \(2 + R\) variables in the output program, where \(R\) is the rank of the array.

The macro `TRANSJD` transforms the identifier of the field in a way that encodes the dimension signature of the result. This encoding is necessary in order that the HPJava type signature of a field can be reconstructed from the class file of a translated (and compiled) class. Note that the `element type` of the multarray 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 `TRANSJD` macro and the macros `TRANSJD BAS` and `TRANSJD DIM` all start with the `the original a` string, followed by the suffix separator “$\_S$”. This is followed by a string characteristic of the individual macro, described next.

The characteristic string for `TRANSJD(a)` is a string of \(R\) letters, each of which is “D” (for a distributed dimension) or “S” (for a sequential dimension). The characteristic string for `TRANSJD BAS(a)` is “bas”. The characteristic string for `TRANSJD DIM(a, r)` is the decimal representation of the constant integer value \(r\).

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

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

The \(R\) variables `TRANSJD DIM(a, 0) \ldots, TRANSJD DIM(a, R - 1)` will hold descriptors for the dimensions of the arrays. The macro `DIMENSION\_TYPE` is defined as

\[
\text{DIMENSION\_TYPE}(\text{attr}_r) \equiv \text{ArrayDim}
\]

if the term `attr_ r` is a hyphen, -, or

\[
\text{DIMENSION\_TYPE}(\text{attr}_r) \equiv \text{SeqArrayDim}
\]

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.rng(r)` and `a.str(r)`, respectively, in the source program. These values are stored in the fields `range` and `stride` of `ArrayDim`. The class `SeqArrayDim` is a subclass of `ArrayDim` that contains extra information that can be used to simplify the computations associated with subscripting a sequential array dimension.

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

```c
float [r, -, *] bar;
```
the translated class can be assumed to have the five fields:

```c
float [1] bar_.$0DS ;
ArrayBase bar_.$bas ;
ArrayDim bar_.$0 ;
ArrayDim bar_.$1 ;
SeqArrayDim bar_.$2 ;
```
SOURCE:

\[ T \ f(U_0, \ldots, U_{N-1}, v_{N-1}) \ \{ S \} \]

TRANSLATION:

\[ T \ f(TRANS\_PARAMS(U_0, \ldots, U_{N-1}, v_{N-1})) \ \{ T[S] \} \]

or

\[ T \ f(TRANS\_PARAMS(U_0, \ldots, U_{N-1}, v_{N-1}), \text{final APGroup } p) \ \{ T[S[p] \} \]

where:

- \( T \) is a Java type,
- the identifier \( f \) is the name of the method,
- each of the terms \( U_0, \ldots, U_{N-1} \) is a Java type or a multiarray type,
- \( v_0, \ldots, v_{N-1} \) 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.

The first translation applies for non-HPspmd methods, the second for HPspmd methods.

Figure A.14: Translation of method declaration.
A.4.3 Translating method declarations

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

The general scheme is illustrated in Figure A.14. 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 \texttt{TRANS\_PARAMS} evaluates to a list of formal parameters. We will define it in terms the simpler macro \texttt{TRANS\_PARAM} which operates on a single formal parameter declaration:

\[
\texttt{TRANS\_PARAMS}(U_0\ v_0, \ldots, U_{N-1}\ v_{N-1}) \equiv \\
\texttt{TRANS\_PARAM}(U_0\ v_0), \ldots, \texttt{TRANS\_PARAM}(U_{N-1}\ v_{N-1})
\]

Now, if \( U \) is not a multiaarray type, we have

\[
\texttt{TRANS\_PARAM}(U\ v) \equiv U\ v
\]

Otherwise, if \( U \) has the form

\[
T\ [(\texttt{attr}_0, \ldots, \texttt{attr}_{R-1})]\ bras\ a;
\]

(where as usual \( T \) is a Java type and as usual each term \( \texttt{attr}_i \) is a single hyphen, \(-\), or a single asterisk, \(*\), and the term \( \texttt{bras} \) is a string of zero or more bracket pairs, \([\] )\) then the macro \texttt{TRANS\_PARAM} is defined by

\[
\texttt{TRANS\_PARAM}(U\ v) \equiv \\
\texttt{ArrayBase}\ a'_{\text{bas}}, \\
\texttt{DIMENSION\_TYPE}(\texttt{attr}_0)\ a'_0, \ldots, \\
\texttt{DIMENSION\_TYPE}(\texttt{attr}_{R-1})\ a'_{R-1}, \\
T\ [\]\ bras\ a'_{\text{dat}}
\]

where \( a'_{\text{bas}}, a'_0, \ldots, a'_{R-1} \), and \( a'_{\text{dat}} \) are new formal parameter names that may be obtained from \( a \) using the prescription given in Figure A.13 (for example—these are local names so the exact prescription is unimportant). The macro \texttt{DIMENSION\_TYPE} is defined in section A.4.2. In other words, each multiaarray parameter is split into \( 2 + R \) parameters.

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

\[
\text{void foo(int } [\cdot,\cdot,\cdot]\ 	ext{bar)} \{ \ldots \}
\]

the translated class can be assumed to have the method:

\[
\text{void foo(int } [] \text{bar}_-\$DDS, \texttt{ArrayBase}\ \text{bar}_-\$BAS, \\
\text{ArrayDim bar}_-\$0, \texttt{ArrayDim}\ \text{bar}_-\$1, \texttt{SeqArrayDim}\ \text{bar}_-\$2, \\
f\text{inal}\ \texttt{APGGroup}\ p) \{ \ldots \}
\]

The last parameter \( p \) added by the translator will hold the value of the \textit{active process group} (APG) in effect at the point of invocation of the method.
SOURCE:

\[
T\ [[attr_0, \ldots, attr_{R-1}]] \ bras \ f(U_0 \ v_0, \ldots, U_{N-1} \ v_{N-1}) \ \{ \\
\}
\]

TRANSLATION:

\[
T \ [\bras \ TRANSJD(f)(\D\D\ D, \\
TRANS_PARAMS(U_0 \ v_0, \ldots, U_{N-1} \ v_{N-1}), \\
final \ AP\ Group \ p) \ \{ \\
\}
\]

where:

- \( T \) is a scalar Java type,
- \( R \) is the rank of the returned array,
- each term \( attr_j \) is a single hyphen, - , or a single asterisk, * ,
- the term \( bras \) is a string of zero or more bracket pairs, [1],
- the identifier \( f \) is the name of the method,
- each of the terms \( U_0, \ldots, U_{N-1} \) is a Java type or a multiarray type,
- \( v_0, \ldots, v_{N-1} \) are parameter names appearing in the original program,
- \( S \) is a block of statements in the original program,
- \( d \) and \( p \) are the names of new parameters, and
- the macros \( TRANSJD \) and \( TRANS_PARAMS \) are defined in the text.

Figure A.15: Translation of declaration of method returning a multiarray.
A.4. BASIC TRANSLATION

Methods that return multiarrays

The scheme for translating a multiarray-valued method declaration is illustrated in Figure A.15.

The macro \texttt{TRANS.ID} encodes the dimension signature of the returned array and is defined in section A.4.2. The macro \texttt{TRANS.PARAMS} was defined earlier in this section.

Instances of the class \texttt{DAD} (the initials stand for \textit{Distributed Array Descriptor}) contain a reference to an \texttt{ArrayBase} object, and a vector of \( R \) instances of \texttt{ArrayDim}. These values are stored in fields \texttt{base} and \texttt{dimensions}, respectively. The DAD object will be created in the calling program and passed to the translated method. Before the method returns, it will store the parameters describing the layout of the multiarray result in these fields (see sections A.4.15 and A.4.15).

An alternative—in some ways more obvious—translation would place the element array in a field in the DAD, and have the method return a DAD. However this would require either we create a separate DAD class for every possible element type—which is hard because HPJava supports multiarrays with arbitrary Java types, including user-defined class and interface types, or the element array field in our extended DAD has the universal \texttt{Object} type. In the latter case one would cast the object back to the appropriate element-array type on return from the call. There is a fatal problem with this approach however; it does not encode the element type of the multiarray in the class file generated by compiling the translated code. So the return type of the method cannot be determined by HPJava programs that load the declaring class from a class file.
SOURCE:

\[ T(U_0, v_0, \ldots, U_{N-1}, v_{N-1}) \{ S \} \]

TRANSLATION:

\[ T(TRANS\_PARAMS(U_0, v_0, \ldots, U_{N-1}, v_{N-1}), \text{final APGroup } p) \{ \]
\[ T[S][p] \]
\}

where:

- \( T \) is a Java type, the name of the declaring class,
- each of the terms \( U_0, \ldots, U_{N-1} \) is a Java type or a multiarray type,
- \( v_0, \ldots, v_{N-1} \) 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 A.16: Translation of constructor declaration.
A.4.4 Translating constructor declarations

The scheme for translating declaration of constructors in HPspmd classes is illustrated in Figure A.16.

The macro \textit{TRANS\_PARAMS} was defined in section A.4.3.
SOURCE:

\texttt{on (e_{grp}) S}

TRANSLATION:

```java
final APGGroup p = T[e_{grp}].toAPG();
if (p != null) {
    T[S][p]
}
```

where:

- $e_{grp}$ is an expression in the source, and
- $p$ is the name of a new variable, and
- $S$ is a statement in the source program.

Figure A.17: Translation of \texttt{on} construct.
A.4.5 Translating on constructs

A translation for the on construct is given in Figure A.17. If the group $c_{grp}$ is not fully contained in the active process group at the time the method toAPG() is invoked, it will throw a GroupNotContainedException.
SOURCE:
\[ at \ (i = x[e_{glb}\}) \ S \]

TRANSLATION:

```cpp
int glb = T[e_{glb}] ;
Location l = x.location(glb) ;

int sub = l.sub ;
int shf = x.str() ;

Dimension dim = l.dim ;
if (dim.crd() == l.crd) {
    final APGGroup p = apg.restrict(dim) ;
    T[S\ y]
}
```

where:

- \( i \) is an index name in the source program,
- \( x \) is a simple expression in the source program,
- \( e_{glb} \) is an expression in the source program,
- \( S \) is a statement in the source program, and
- \( glb, l, sub, shf, dim \) and \( p \) are the names of new variables.

Figure A.18: Translation of \( at \) construct.
A.4.6 Translating at constructs

A translation for the at construct is given in Figure A.18. Note that pre-translation will have reduced $x$ to a simple expression.

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` (see Figure A.59).

The _global index_ for the index $i$ is the value of `glb`. This value is used in evaluating the global index expression $i^i$.

The _local subscript_ for the index $i$ is the value of `sub`. This value is used in computation of offsets generated when $i$ is used as an array subscript.

The _shift step_ for the index $i$ is the value of `shf`. This value is used in computation of offsets associated with shifted index subscripts.

The _dimension_ for the index $i$ is the value of `dim`. One possible later use of this value is in computation the distribution group of a section subscripted with $i$.

If the process dimension `dim` is not an active dimension of `apg` at the time the method `restrict()` is invoked, it will throw a `DimensionNotInGroupException`.

[Variables like `glb`, `sub`, etc., should be final, so that they can be accessed in local classes defined inside the at construct? Unclear whether distributed indices should be usable inside such classes, but no immediately obvious reason to forbid it.]
SOURCE:

\[ \text{overall} \ (i = x \text{ for } e_{io} : e_{hi} : e_{stp}) \ S \]

TRANSLATION:

\[ \text{Block } b = x.\text{localBlock}(T[e_{io}], T[e_{hi}], T[e_{stp}]) ; \]
\[ \text{int } shf = x.\text{str}() ; \]
\[ \text{Dimension } dim = x.\text{dim}() ; \]
\[ \text{final AFGGroup } p = \text{apg.restrict(dim)} ; \]
\[ \text{for (int } l = 0 ; l < b.\text{count} ; l++) \{ \]
\[ \text{int } sub = b.\text{sub_bas} + b.\text{sub_stp} \times l ; \]
\[ \text{int } glb = b.\text{glb_bas} + b.\text{glb_stp} \times l ; \]
\[ T[S[p]] \]
\[ \}
\]

where:

- \( i \) is an index name in the source program,
- \( x \) is a simple expression in the source program,
- \( e_{io}, e_{hi}, \) and \( e_{stp} \) are expressions in the source,
- \( S \) is a statement in the source program, and
- \( b, shf, dim, p, l, sub \) and \( glb \) are names of new variables.

Figure A.19: Translation of overall construct.
A.4.7 Translating overall constructs

A translation for the overall construct is given in Figure A.19. The `localBlock()` method and the `Block` class have been discussed at length in section 7.3.

The variables `shf`, `dim`, `sub` and `glob` play the same roles as the correspondingly named variables in the translation of the at construct.

If the process dimension `dim` is not an active dimension of `apg` at the time the method `restrict()` is invoked, it will throw a `DimensionNotInGroupException`. 
SOURCE:

\[ e \equiv i^c \]

TRANSLATION:

\[ T[e] \equiv glb \]

where:

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

Figure A.20: Translation of global index for \( i \).
A.4.8 Translating global index expression

The scheme is illustrated in Figure A.20. The global index variable associated with a distributed index is defined in sections A.4.6 and A.4.7.
SOURCE:

\[ e \equiv a \]

TRANSLATION:

\[ T_{\text{dat}}[e] \equiv a'_{\text{dat}} \]
\[ T_{\text{bas}}[e] \equiv a'_{\text{bas}} \]
\[ T_0[e] \equiv a'_0 \]
\[ \ldots \]
\[ T_{R-1}[e] \equiv a'_{R-1} \]

where:

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

Figure A.21: Translation of a multiarray-valued variable access.
A.4.9 Translating variable accesses

We only need to consider specially the case where the variable is a multiarray. The general scheme is illustrated in Figure A.21. This applies to the case of a variable that is a simple identifier—a local variable, method or constructor argument, or instance variable of the current object. The case of a field reference with an object prefix will be covered in section A.4.14.

The names \( a'_\text{dat}, a'_\text{bas}, a'_0, \ldots, a'_{R-1} \) are the names introduced by the translator when translating the corresponding field, local variable declaration or formal parameter in the source program (see sections A.4.2 and A.4.3).
SOURCE:

\[ e \equiv e_t = e_s \]

TRANSLATION:

\[
\begin{align*}
T_{\text{dat}}[e] &\equiv T_{\text{dat}}[e_t] = T_{\text{dat}}[e_s] \\
T_{\text{bas}}[e] &\equiv T_{\text{bas}}[e_t] = T_{\text{bas}}[e_s] \\
T_0[e] &\equiv T_0[e_t] = T_0[e_s] \\
\cdots
T_{R-1}[e] &\equiv T_{R-1}[e_t] = T_{R-1}[e_s]
\end{align*}
\]

where:

- \( e_t \) has multiarray type,
- \( e_s \) is assignment convertible to the type of \( e_t \), and
- \( R \) is the rank of both arrays.

Figure A.22: Translation of a multiarray assignment.
A.4.10 Translating assignment expressions

When the expressions involved are multiarrays, the general scheme is illustrated in Figure A.22. This translation applies to assignments where the right-hand-side of the assignment is not a composite expression. Assignments involving composite expressions are handled in the following sections.
SOURCE:

\[ a = \text{new } T \left[ [e_0, \ldots, e_{R-1}] \right] \text{ bras} ; \]

TRANSLATION:

```java
int s = 1;
DEFINE_DIMENSION(T_{R-1}[a], e_{R-1}, s)

\ldots

DEFINE_DIMENSION(T_0[a], e_0, s)
T_{dat}[a] = \text{new } T \left[ s \right] \text{ bras} ;
T_{bas}[a] = \text{new } ArrayBase(\text{null}, 0) ;
```

where:

- \( T \) is a Java type,
- \( R \) is the rank of the created array,
- each \( e_r \) is an integer-valued simple expression in the source program,
- the term \( \text{bras} \) is a string of zero or more bracket pairs, \( [] \),
- the expression \( a \) is the assigned array variable in the source program,
- \( s \) is the name of a new temporary, and
- the macro \( \text{DEFINE\_DIMENSION} \) is defined in the text.

Figure A.23: Translation of multiarray creation expression in non-HPspin code.
A.4.11 Translating multiarray creation

The pre-translator ensures that multiarray creation only appears on the right-hand-side of a top-level assignment, so we only need to consider that case. The scheme for the translation in non-HPxpmode code is illustrated in Figure A.23.

In this case we must be creating a sequential multiarray and all extent expressions must be integers. The macro $\text{DEFINE\_DIMENSION}$ is defined as follows:

\[
\text{DEFINE\_DIMENSION}(d'_r, e_r, s) \equiv \\
\begin{align*}
    d'_r & = \text{new SeqArrayDim}(e_r, s) \\
    s & \equiv e_r
\end{align*}
\]

As each dimension is processed, the memory stride for the next dimension is computed by multiplying the variable $s$ by the extent of the current dimension. The final value of $s$ is the total number of locally held elements.
SOURCE:

\[
  a = \text{new } T \left[ [e_0, \ldots, e_{R-1}] \right] \text{bras on } p ;
\]

TRANSLATION:

\[
\begin{align*}
  \text{int } & s = 1 ; \\
  \text{int } & b = 0 ; \\
  p.\text{checkContained}(\text{args}) ; \\
  \text{DimSet } & t = p.\text{getDimSet}() ; \\
  \text{DEFINE DIMENSION}(T_{R-1}[a], e_{R-1}, s, b, t) \\
  \ldots
\end{align*}
\]

\[
\begin{align*}
  \text{DEFINE DIMENSION}(T_0[a], e_0, s, b, t) \\
  T_{\text{dat}}[a] & = p.\text{anMember()} ? \text{new } T [s] \text{ bras : null} ; \\
  T_{\text{bas}}[a] & = \text{new ArrayBase}(p, b) ;
\end{align*}
\]

where:

- \( T \) is a Java type.
- \( R \) is the rank of the created array.
- each \( e_r \) is either a range-valued or an integer-valued simple expression in the source program.
- the term \( \text{bras} \) is a string of zero or more bracket pairs, [ ],
- \( p \) is a simple expression in the source program.
- the expression \( a \) is the assigned array variable in the source program.
- \( s \) and \( b \), and \( t \) are names of new temporaries, and
- the macro \text{DEFINE DIMENSION} is defined in the text.

Figure A.24: Translation of multiarray creation with \text{on} clause.
Creating distributed arrays

The scheme for distributed array creation with an \textit{on} clause is given in Figure A.21.

If the expression \( e_r \) is a range the macro \texttt{DEFINE\_DIMENSION} is defined as follows:

\[
\texttt{DEFINE\_DIMENSION}(a'_r, e_r, s, b, t) \equiv \\
a'_r = e_r\text{.arrayDim}(s) ; \\
b += s * e_r\text{.loExtent}() ; \\
s *= e_r\text{.volume}() ; \\
t\text{.remove}(e_r\text{.dim}()) ;
\]

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 variable \( b \) is incremented to allow space for lower ghost regions, below the base of the physical array, if this is required for the ranges involved.

If the expression \( e_r \) is an integer then \texttt{DEFINE\_DIMENSION}(\( a'_r, e_r, s, b, t \)) is evaluated the same as \texttt{DEFINE\_DIMENSION}(\( a'_r, e_r, s \)) in the previous subsection.

The call to \texttt{checkContained()} throws a \texttt{GroupNotContainedException} run-time exception if \( p \) is not contained in the current APG. The variable \( t \) is also introduced for run-time checking of correct usage. The calls to \texttt{remove()} will throw a \texttt{DimensionNotInGroupException} run-time exception if any range specified for the array is not distributed over a dimension in \( p \), or if any two specified ranges are distributed over the same dimension.

The method \texttt{arrayDim()} on the \texttt{Range} class creates an instance of \texttt{ArrayDim}, with the memory stride specified in its argument. It is used in place of a call to the \texttt{ArrayDim} constructor because \texttt{arrayDim()} has the property that if the range is actually a collapsed range, the returned object will be an instance of the \texttt{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 A.4.21).
SOURCE:
\[ a = \text{new } T \left[[e_0, \ldots, e_{R-1}]\right] \, \text{bras} ; \]

TRANSLATION:

```java
int s = 1;
int b = 0;
DimSet t = apg.getDimSet();
DEFINE_DIMENSION(T_{R-1} \left[a]\right), e_{R-1}, s, b, t)

\ldots

DEFINE_DIMENSION(T_{0} \left[a]\right), e_{0}, s, b, t)
T_{dat} \left[a\right] = \text{new } T \left[s\right] \, \text{bras} ;
T_{bas} \left[a\right] = \text{new } ArrayBase\left(\text{apg}, b\right) ;
```

where:
- \( T \) is a Java type.
- \( R \) is the rank of the created array.
- each \( e_r \) is either a range-valued or an integer-valued simple expression in the source program.
- the term bras is a string of zero or more bracket pairs, \([\right]\).
- the expression \( a \) is the assigned array variable in the source program.
- \( s, b, \) and \( t \) are names of new temporaries, and
- the macro \text{DEFINE\_DIMENSION} is defined in the text.

Figure A.25: Translation of multiarray creation expression without the \texttt{on} clause.
Distributed array creation without the on clause

If we are in HPspmd code, but there is no on clause, the distribution group defaults to the active process group, and the scheme simplifies as illustrated in Figure A.25. The macro $\textit{DEFINE\_DIMENSION}$ is defined as in the previous section.
SOURCE:

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

TRANSLATION:

\[ T[e] \equiv T_{\text{dat}}[a] \ [\text{OFFSET}(a, e_0, \ldots, e_{R-1})] \]

where:

- The expression \( a \) is the subscripted array.
- Each term \( e_r \) is either an integer, a distributed index name, or a shifted index expression, and
- the macro \( \text{OFFSET} \) is defined in the text.

Figure A.26: Translation of multiax element access expression.
A.4.12 Translating element access

We only need to consider the case where the array reference is a multiarray. The general scheme is illustrated in Figure A.26. The macro \textit{OFFSET} is defined as

\[
\text{OFFSET}(a, e_0, \ldots, e_{R-1}) \equiv \\
\text{T}_{\text{bas} [a]}.\text{base} + \text{OFFSET}\_\text{DIM} (\text{T}_0 [a], e_0) \\
\ldots \\
+ \text{OFFSET}\_\text{DIM} (\text{T}_{R-1} [a], e_{R-1})
\]

There are three cases for the macro \textit{OFFSET}\_\text{DIM} depending on whether the subscript argument is an integer subscripts (in a sequential dimension), a distributed index, or a shifted index.

If \(e_r\) is an integer expression (which implies that \(a'_r\) has type \textit{SeqArrayDim}), then

\[
\text{OFFSET}\_\text{DIM}(a'_r, e_r) \equiv \quad a'_r.\text{off_{bas}} + a'_r.\text{off_{stp}} \times \text{T}[e_r]
\]

The fields \textit{off_{bas}} and \textit{off_{stp}} are initialized by the constructors for \textit{SeqArrayDim}. They do not exist in the superclass \textit{ArrayDim}.

Otherwise, if \(e_r\) is a distributed index \(i\), then

\[
\text{OFFSET}\_\text{DIM}(a'_r, e_r) \equiv \quad a'_r.\text{stride} \times \text{sub}
\]

where \textit{sub} is the local subscript variable for this index (see sections A.4.6 and A.4.7).

Otherwise, if \(e_r\) is a shifted index \(i \pm d\), then

\[
\text{OFFSET}\_\text{DIM}(a'_r, e_r) \equiv \quad a'_r.\text{stride} \times (\text{sub} \pm \text{stp} \times \text{T}[d])
\]

where \textit{sub} is the local subscript variable and \textit{stp} is the shift step variable for \(i\) (again, see sections A.4.6 and A.4.7).
SOURCE:

\[ v = a [ [ sub_0, \ldots, sub_{m-1} ]] \]

TRANSLATION:

\[
\text{PROCESS\_SUBSCRIPTS}(v, 0, a, 0)\\
T_{\text{dat}}[v] = T_{\text{dat}}[a] ;\\
T_{\text{bas}}[v] = T_{\text{bas}}[a] ;
\]

where:

The expression \( v \) is the assigned array variable in the source program, the simple expression \( a \) is the subscripted array in the source program, each term \( sub_i \) is either an integer, a triplet, or \(<\), and the macro \( \text{PROCESS\_SUBSCRIPTS} \) is defined in the text.

Figure A.27: Translation of array section with no scalar subscripts.
A.4.13 Translating array sections

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

We will break it down into three cases: the case where there are no scalar subscripts—integer or distributed index; the case where integer scalar subscripts appear in **sequential** dimensions only; and the general case where scalar subscripts may appear in distributed dimensions. The scheme for translating the first case is illustrated in Figure A.27.

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

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

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

Otherwise, if `subs` is the triplet, `e sublime ; e step`, and the `s`th dimension of `a` is distributed, then

```
PROCESS_SUBSCRIPTS(v, r, a, s) ≡
    T_r [v] = a'_s . range . subrng (e'_1, e'_last, e'_step) . arrayDim (a'_s . stride) ;
    PROCESS_SUBSCRIPTS(v, r + 1, a, s + 1)
```

where `a'_s = T_s [a]`, `e'_1 = T [e_1]`, `e'_last = T [e_last]`, and `e'_step = T [e_step]`. Otherwise, if `subs` is the triplet, `e_1 ; e_last ; e_step`, and the `s`th dimension of `a` is sequential, then

```
PROCESS_SUBSCRIPTS(v, r, a, s) ≡
    T_r [v] = new SeqArrayDim (a'_s . range . subrng (e'_1, e'_last, e'_step), a'_s . stride) ;
    PROCESS_SUBSCRIPTS(v, r + 1, a, s + 1)
```

with definitions as above. Two similar cases using the two-argument form of `subrng()` take care of triplets of the form `e_1 ; e_last`. Otherwise, if `subs` is the splitting subscript, `←`, then

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

where `x`, `u`, and `z` are the names of new temporaries.
SOURCE:
\[ v = a \left[ [subs_0, \ldots, subs_{R-1}] \right] ; \]

TRANSLATION:
\[
\begin{align*}
\text{int } b &= T_{bas}[a].\text{base} ; \\
\text{PROCESS\_SUBSCRIPTS}(v, 0, a, 0) \\
T_{\text{dat}}[v] &= T_{\text{dat}}[a] \\
T_{\text{bas}}[v] &= \text{new ArrayBase}(T_{\text{bas}}[a].\text{group}, b) ;
\end{align*}
\]

where:

The expression \( v \) is the assigned array variable in the source program, the simple expression \( a \) is the subscripted array in the source program, each term \( subs_i \) is either an integer, a distributed index, a shifted index, a triplet, or \( \langle \rangle \), \( b \) is the name of a new temporary, and the macro \text{PROCESS\_SUBSCRIPTS} is defined in the text.

Figure A.28: Translation of array section without any scalar subscripts in \textit{distributed} dimensions.
Scalar subscripts in sequential dimensions

To handle the case where scalar subscripts may appear in sequential dimensions (Figure A.28) we must add one new case for the definition of the macro \textit{PROCESS\_SUBSCRIPTS}.

If \textit{subs}_s is an integer expression, a distributed index, or a shifted index\footnote{Although it is an odd case, there is nothing in the language definition to prevent one from declaring indexes in overall or at statements to belong to the collapsed range of a sequential array dimension, and then using the index as a subscript in that array dimension.}, and the \textit{s}th dimension of \textit{a} is sequential, then

\[
\text{PROCESS\_SUBSCRIPTS}(v, r, a, s) \equiv \\
b \ += \ \text{OFFSET\_DIM}(T_s[a], \text{subs}_s) \\
\text{PROCESS\_SUBSCRIPTS}(v, r, a, s + 1)
\]

where the macro \textit{OFFSET\_DIM} is defined in section A.4.12.
SOURCE:

\[ v = a \begin{bmatrix} subs_0, \ldots, subs_{n-1} \end{bmatrix} \; \]

TRANSLATION:

```c
int b = Tbas[a].base;
Group p = Tbas[a].group;
PROCESS_SUBSCRIPTS(v, 0, a, 0)
if(p != null)
    Tdat[v] = p.amMember() ? Tdat[a] : null;
else
    Tdat[v] = Tdat[a];
    Tbas[v] = new ArrayBase(p, b);
```

where:

The expression \( v \) is the assigned array variable in the source program,
the simple expression \( a \) is the subscripted array in the source program,
each term \( subs_i \) is either an integer, a distributed index, a shifted index,
a triplet, or \( \langle \rangle \).

\( b \) and \( p \) are the names of new temporaries, and
the macro \( \text{PROCESS\_SUBSCRIPTS} \) is defined in the text.

Figure A.29: Translation of array section allowing scalar subscripts in distributed dimensions.
Scalar subscripts in distributed dimensions

The scheme for translating array sections when scalar subscripts appear in some distributed dimension is illustrated in Figure A.29.

We add two new cases for the definition of the macro \texttt{PROCESS\_SUBSCRIPTS}. These new definitions apply only if the \textit{s}th dimension of \textit{a} has the distributed attribute.

If, under these conditions, \textit{subs} is the integer expression \textit{n}, then

\begin{verbatim}
PROCESS\_SUBSCRIPTS(v, r, a, s) ==
  Range x = a'_s.range ;
  Location l = x.location(n') ;
  b += l.sub * a'_s.stride ;
  if(p != null) p = p.restrict(x.dims(), l.crd) ;
  PROCESS\_SUBSCRIPTS(v, r, a, s + 1)
\end{verbatim}

where \textit{x} and \textit{l} are the names of new temporaries, \textit{a'_s} = \text{\texttt{T}}_s[a], and \textit{n'} = \text{\texttt{T}}[n].

Otherwise, if \textit{subs} is a distributed index \textit{i} or a shifted index \textit{i} \pm \textit{d}, then

\begin{verbatim}
PROCESS\_SUBSCRIPTS(v, r, a, s) ==
  b += OFFSET\_DIM(a'_s, subs) ;
  if(p != null) p = p.restrict(dim) ;
  PROCESS\_SUBSCRIPTS(v, r, a, s + 1)
\end{verbatim}

where the macro \texttt{OFFSET\_DIM} is defined in section A.4.12 and \textit{dim} is the dimension associated with \textit{i}.
SOURCE:

\[ e \equiv e_{\text{obj}} \cdot a \]

TRANSLATION:

\[ T_{\text{dat}}[e] \equiv e_{\text{obj}} \cdot a'_{\text{dat}} \]
\[ T_{\text{bas}}[e] \equiv e_{\text{obj}} \cdot a'_{\text{bas}} \]
\[ T_0[e] \equiv e_{\text{obj}} \cdot a'_0 \]
\[ \ldots \]
\[ T_{R-1}[e] \equiv e_{\text{obj}} \cdot a'_{R-1} \]

where:

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

Figure A.30: Translation of a multiarray-valued field access.
A.4.14 Translating field accesses

We only need to consider the case where the field is a multiaarray. The general scheme is illustrated in Figure A.30.

The names $a'_{dat}$, $a'_{bas}$, and $a'_0, \ldots, a'_{R-1}$ are the names introduced by the translator when translating the corresponding field in the source program (see section A.4.2).
SOURCE:

\[ e \equiv e_{\text{obj}} \cdot f(e_0, \ldots, e_{N-1}) \]

TRANSLATION:

\[ T[e] \equiv T[e_{\text{obj}}] \cdot f(\text{TRANSARGS}(e_0, \ldots, e_{N-1})) \]

where:

- The expression \( e_{\text{obj}} \) has reference type or is a class,
- the identifier \( f \) is the name of the method,
- each term \( e_i \) is an actual argument in the original program,
- the macro \( \text{TRANSARGS} \) is defined in the text.

Figure A.31: Translation of method invocation expression, where return value is not multiarray-valued.
A.4.15 Translating method invocations

There are two cases to consider. The case where the result of the method is not a multiarray, and the case where the result is a multiarray. First we consider the case where the result is not a multiarray. The general scheme is illustrated in Figure A.31. 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. If the invoked method (or, in later sections, constructor) is not an HPspmd method (or constructor) the definition is:

$$TRANS\_ARGS(e_0, \ldots, e_{N-1}) \equiv$$
$$TRANS\_ARG(e_0), \ldots, TRANS\_ARG(e_{N-1})$$

If the invoked method (or constructor) is an HPspmd method (or constructor), the definition is:

$$TRANS\_ARGS(e_0, \ldots, e_{N-1}) \equiv$$
$$TRANS\_ARG(e_0), \ldots, TRANS\_ARG(e_{N-1}), apg$$

Now, if $e$ is not a multiarray, then

$$TRANS\_ARG(e) \equiv \ T[e]$$

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

$$TRANS\_ARG(e) \equiv$$
$$T_{bas}[e], \ T_0[e], \ldots, T_{R-1}[e], \ T_{dat}[e]$$

In other words, each multiarray argument is split into $2 + R$ arguments.
SOURCE:

\[ v = e_{\text{obj}} \cdot f(e_0, \ldots, e_{N-1}) \]

TRANSLATION:

\[ \text{DAD } d = \text{new DAD}(R) ; \]
\[ T_{\text{dat}}[v] = T[e_{\text{obj}}].TRANSIDX(f)(d, TRANSARGS(e_0, \ldots, e_{N-1})) ; \]
\[ T_{\text{has}}[v] = d.\text{base} \]
\[ T_0[v] = TRANS_DIM(d, 0) \]
\[ \cdots \]
\[ T_{R-1}[v] = TRANS_DIM(d, R - 1) \]

where:

The expression \( v \) is the assigned array variable in the source program,
The expression \( e_{\text{obj}} \) has reference type or is a class,
the identifier \( f \) is the name of the method,
each term \( e_r \) is an actual argument in the source program,
\( d \) is a new temporary,
\( R \) is the rank of the result, and
the macros \( \text{TRANSIDX}, \text{TRANSARGS}, \) and \( \text{TRANS_DIM} \)
are defined in the text.

Figure A.32: Translation of multiarray-valued method invocation expression.
Methods that return multiarrays

The scheme for translating a multiarray-valued method invocation is illustrated in Figure A.32.

The macro $TRANS\_ID$ encodes the dimension signature of the returned array and is defined in section A.4.2. The macro $TRANS\_ARGS$ was defined earlier in this section.

The macro $TRANS\_DIM$ is defined as

$$TRANS\_DIM(d, r) \equiv d.d\text{imensions}[r]$$

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

$$TRANS\_DIM(d, r) \equiv (\text{SeqArrayDim}) d.d\text{imensions}[r]$$

if the $r$th dimension is sequential.

[Also need to support invocation as a statement expression, with source:

$$e_{\text{obj}}.f(e_0, \ldots, e_{N-1}) ;$$

and translation:

$$T[e_{\text{obj}}].TRANS\_ID(f) (\text{new DAD}(R), TRANS\_ARGS(e_0, \ldots, e_{N-1})) ;$$]
SOURCE:  
  return a ;

TRANSLATION:  
  d.base = T[beg][a] ;
  d.dimensions [0] = T[0][a] ;
  ...  
  d.dimensions [R - 1] = T[R-1][a] ;
  return T[dat][a] ;

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

Figure A.33: Translation of return statement in array-valued method.
A.4. BASIC TRANSLATION

Translation of return statement
The scheme for translating a return statement in the definition of an array-valued method is illustrated in Figure A.33.
SOURCE:

\[ e \equiv \text{new } T(e_0, \ldots, e_{N-1}) \]

TRANSLATION:

\[ T[e] \equiv \text{new } T(\text{TRANSARGS}(e_0, \ldots, e_{N-1})) \]

where:

- \( T \) is a Java class type,
- each term \( e_r \) is an actual argument in the original program,
- the macro \( \text{TRANSARGS} \) is defined in the text.

Figure A.34: Translation of class instance creation expression.
A.4.16 Translating constructor invocations

The rules for translating constructor invocations follow directly from the rules for method invocations given in section A.4.15.

Figure A.34 illustrates the translation for a class instance creation expression. Explicit constructor invocations (specifying this or super) do not introduce any new features.
SOURCE:

\[ v = a / e_{\text{loc}} ; \]

TRANSLATION:

\[
\text{RESTRICT\_DIST\_GROUP}(p, T_{\text{bas}}[a], \text{group}, e_{\text{loc}}) \\
T_{\text{dat}}[v] = p.\text{amMember()} ? T_{\text{dat}}[a] : \text{null} ; \\
T_{\text{bas}}[v] = \text{new ArrayBase}(p, T_{\text{bas}}[a], \text{base}) ; \\
\]

\[
\text{CHECK\_DIMENSION}(T_0[a], e_{\text{loc}}) ; \\
T_0[v] = T_0[a] ; \\
\]

\[
\text{CHECK\_DIMENSION}(T_{R-1}[a], e_{\text{loc}}) ; \\
T_{R-1}[v] = T_{R-1}[a] ; \\
\]

where:

The expression \( v \) is the assigned array variable in the source program, the simple expression \( a \) is the array in the source program to be restricted, \( R \) is the rank of \( a \), the expression \( e_{\text{loc}} \) is either a distributed index, a shifted index, or a range element, \( p \) is the name of a new temporary, and the macros \text{RESTRICT\_DIST\_GROUP} and \text{CHECK\_DIMENSION} are defined in the text.

Figure A.35: Translation of distributed array restriction operation.
A.4.17 Translating distributed array restriction

The scheme is illustrated in Figure A.35. The basic operation is simple. Most of the complications come from the run-time checks need to ensure one doesn’t produce an illegally distributed result.

If \( e_{\text{loc}} \) is a range element of the form \( e_{\text{rng}}[n] \), the macros \textsc{restrict(dist.group)} and \textsc{check.dimension} are defined as

\[
\textsc{restrict(dist.group)} \left( p, e'_{\text{grp}}, e_{\text{loc}} \right) \equiv \\
\text{Location } l = x.\text{location}(e'_{\text{rng}}) ; \\
\text{Group } p = e'_{\text{grp}}.\text{restrict}(l.\text{dim}, l.\text{crd}) ; \\
\text{Dimension } d = l.\text{dim} ;
\]

\[
\textsc{check.dimension}(a^i, e_{\text{loc}}) \equiv \\
a^i.\text{checkRestriction}(d) ;
\]

where \( e'_{\text{rng}} = T^V_s[e_{\text{rng}}] \), and \( n' = T[n] \), and \( d \) is the name of a new temporary. Otherwise, if \( e_{\text{loc}} \) is a distributed index \( i \) or a shifted index \( i \pm d \), they are defined as

\[
\textsc{restrict(dist.group)} \left( p, e'_{\text{grp}}, e_{\text{loc}} \right) \equiv \\
\text{Group } p = e'_{\text{grp}}.\text{restrict}(\text{dim}) ;
\]

\[
\textsc{check.dimension}(a^i, e_{\text{loc}}) \equiv \\
a^i.\text{checkRestriction}(\text{dim}) ;
\]

where in this case \( \text{dim} \) is the dimension associated with \( i \).

The \text{restrict()} calls may throw a \text{DimensionNotInGroupException} run-time exception if the location belongs to a range whose process dimension is not in \( a.\text{grp}() \). Similarly, one of the calls to \text{checkRestriction()} will throw a \text{DimensionNotInGroupException} if any of the original array ranges is distributed over this process dimension—the restriction is supposed to occur in a dimension over which the array is \text{replicated}.

If the array \( a \) is a sequential array that was created in non-HPspmd code, \( p \) will be null and a \text{NullPointerException} will be thrown.
SOURCE:

\[ e ; \]

TRANSLATION:

\[
\begin{align*}
T_{\text{dat}}[e] ; \\
T_{\text{bas}}[e] ; \\
T_0[e] ; \\
\ldots \\
T_{R-1}[e] ;
\end{align*}
\]

where:

\( e \) is a multiarray-valued expression in the source program.

Figure A.36: Translation of other statement expressions.
A.4.18 Translating other statement expressions

In earlier sections we have given a series of special rules for statement expressions (mostly assignments) involving composite multiarray expressions. Figure A.36 gives the trivial rule for translating any other multiarray expression appearing as a statement expression.

In fact there is only one remaining possibility, and that is for $e$ to be an assignment statement with a non-composite expression on the right-hand-side. The rule for translation of the expression itself is thus given in section A.4.10. For completeness we factor out the additional step of generating a sequence of statement expressions as a separate rule here, noting that the rule of A.4.10 also applies (of course) to assignment expressions that are not top-level statement expressions.
SOURCE:
\[ p / e_{loc} \]

TRANSLATION:
\[ \text{RESTRICT\_GROUP}(T[p], e_{loc}) \]

where:
- the expression \( p \) is the group to be restricted,
- the expression \( e_{loc} \) is either a distributed index, a shifted index, or a range element, and
- the macro \( \text{RESTRICT\_GROUP} \) is defined in the text.

Figure A.37: Translation of group restriction operation.
A.4.19 Translating group restriction

The scheme is illustrated in Figure A.37.

If \( e_{\text{loc}} \) is a range element of the form \( e_{\text{rng}}[n] \), the macro \( RESTRICT\_GROUP \) is defined as

\[
RESTRICT\_GROUP \left( e'_{\text{grp}}, e_{\text{loc}} \right) \equiv e'_{\text{grp}}.\text{restrict} \left( e'_{\text{rng}}, n' \right)
\]

where \( e'_{\text{rng}} = T_i^j \left[ e_{\text{rng}} \right] \), and \( n' = T[n] \). Otherwise, if \( e_{\text{loc}} \) is a distributed index \( i \) or a shifted index \( i \pm d \), it is defined as

\[
RESTRICT\_GROUP \left( e'_{\text{grp}}, e_{\text{loc}} \right) \equiv e'_{\text{grp}}.\text{restrict} \left( \text{dim} \right)
\]

where in this case \( \text{dim} \) is the dimension associated with \( i \).
SOURCE:
\[ e \equiv x \ [ [e_{lo} : e_{hi}] ] \]

TRANSLATION:
\[ T[e] \equiv T[x].\text{subrng}(T[e_{lo}], T[e_{hi}]) \]

SOURCE:
\[ e \equiv x \ [ [e_{lo} : e_{hi} : e_{stp}] ] \]

TRANSLATION:
\[ T[e] \equiv T[x].\text{subrng}(T[e_{lo}], T[e_{hi}], T[e_{stp}]) \]

where:

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

Figure A.38: Translation of subrange expressions.
A.4.20 Translating subrange expressions

The schemes are illustrated in Figure A.38.
SOURCE:
\[ e \equiv a.\text{dat}() \]

TRANSLATION:
\[ T[e] \equiv T_{\text{dat}}[a] \]

SOURCE:
\[ e \equiv a.\text{bas}() \]

TRANSLATION:
\[ T[e] \equiv T_{\text{bas}}[a] . \text{base} \]

SOURCE:
\[ e \equiv a.\text{grp}() \]

TRANSLATION:
\[ T[e] \equiv T_{\text{bas}}[a] . \text{group} \]

where:

The simple expression \( a \) has distributed array type.

Figure A.39: Translation of multiarray inquiries.
SOURCE:
\[ e \equiv a\text{.rng}(r) \]

TRANSLATION:
\[ T[e] \equiv T_r[a]\text{.range} \]

SOURCE:
\[ e \equiv a\text{.str}(r) \]

TRANSLATION:
\[ T[e] \equiv T_r[a]\text{.stride} \]

where:

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

Figure A.40: Translation of multiarray inquiries, continued.
SOURCE:

\[ e = (T \ [[attr_0, \ldots, attr_{R-1}]] \ bm) \ a \]

TRANSLATION:

\[
\begin{align*}
\mathbf{T}_{\text{dat}}[e] & \equiv (T \ [[\ \\ bms]) \ T_{\text{dat}}[a] \\
\mathbf{T}_{\text{bas}}[e] & \equiv \mathbf{T}_{\text{bas}}[a] \\
\mathbf{T}_0[e] & \equiv \text{CAST}\_\text{DIMENSION} \ (attr_0, \mathbf{T}_0[a]) \\
\ldots \\
\mathbf{T}_{R-1}[e] & \equiv \text{CAST}\_\text{DIMENSION} \ (attr_{R-1}, \mathbf{T}_{R-1}[a]) 
\end{align*}
\]

where:

- \( T \) is a Java type,
- each term \( attr \) is a single hyphen, \(-\), or a single asterisk, \(*\),
- the term \( bms \) is a string of zero or more bracket pairs, \([\ldots]\),
- \( a \) is an array expression in the source program, of rank \( R \), and
- the macro \( \text{CAST}\_\text{DIMENSION} \) is defined in the text.

Figure A.41: Translation of cast of multiarray expression.
A.4.21 Translating casts

The scheme for translating a cast of a multiarray valued expression is illustrated in Figure A.41. The macro \( CAST\_DIMENSION \) is defined as follows:

\[
CAST\_DIMENSION(\text{attr}_r, a'_r) \equiv \begin{cases} 
\text{ArrayDim} \ a'_r \\
\text{SeqArrayDim} \ a'_r 
\end{cases}
\]

if the term \( \text{attr}_r \) is empty, or

if the term \( \text{attr}_r \) is an asterisk, \(*\).
SOURCE:

\[ e \equiv a \text{ instanceof } T \left[ \left[ \text{attr}_0, \ldots, \text{attr}_{R-1} \right] \right] \text{ bras} \]

TRANSLATION:

\[ T[e] \equiv (T_{\text{dat}}[a] \text{ instanceof } T \left[ \left[ \text{bras} \right] \right]) \&\&
(T_0[a] \text{ instanceof } DIMENSION\_TYPE(\text{attr}_0)) \&\&
\ldots
(T_{R-1}[a] \text{ instanceof } DIMENSION\_TYPE(\text{attr}_{R-1})) \]

where:

- \( a \) is an array expression in the source program, rank \( R \),
- \( T \) is a Java type,
- each term \( \text{attr}_i \) is a single hyphen, \(-\), or a single asterisk, \(*\),
- the term \( \text{bras} \) is a string of zero or more bracket pairs, [ ],
- the macro \( \text{DIMENSION\_TYPE} \) is defined in the text.

Figure A.42: Translation of \text{instanceof} expression for multiarray type.
A.4.22 Translating instanceof

The scheme for translating an instanceof test applied to a multiarray valued expression is illustrated in Figure A.42. The macro \textit{DIMENSION\_TYPE} is in section A.4.2.
SOURCE:

\[ e \equiv a == b \]

TRANSLATION:

\[
T[e] \equiv (T_{\text{dat}}[a] == T_{\text{dat}}[b]) \&\& \\
(T_{\text{bas}}[a] == T_{\text{bas}}[b]) \&\& \\
(T_0[a] == T_0[b]) \&\& \\
\ldots \\
(T_{R-1}[a] == T_{R-1}[b])
\]

SOURCE:

\[ a !\equiv b \]

TRANSLATION:

\[
T[e] \equiv (T_{\text{dat}}[a] !\equiv T_{\text{dat}}[b]) \lor \\
(T_{\text{bas}}[a] !\equiv T_{\text{bas}}[b]) \lor \\
(T_0[a] !\equiv T_0[b]) \lor \\
\ldots \\
(T_{R-1}[a] !\equiv T_{R-1}[b])
\]

where:

Expressions \( a \) and \( b \) are assignment compatible with a multiarray type, and this type has rank, \( R \).

Figure A.43: Translation of reference equality tests, applied to array expressions.
A.4.23 Translating reference equality

The scheme for translating the reference equality operators == and !=, applied
to multiarrays is illustrated in Figure A.43.

Because there is no single Java reference to a multiarray (or, equivalently,
because a multiarray is not considered to be an object) there is no obvious, a
priori definition for reference equality between multiarrays. The translations
given here can be read as definitions.

One consequence of these definitions, together with the translation for array
sections given in section A.4.13, 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.
SOURCE:

\[ e \equiv \text{null} \]

TRANSLATION:

\[ T_{\text{dat}}[e] \equiv \text{null} \]

\[ T_{\text{bas}}[e] \equiv \text{null} \]

\[ T_0[e] \equiv \text{null} \]

\[ \ldots \]

\[ T_{R-1}[e] \equiv \text{null} \]

where:

\( R \) is the rank of the expected multiarray expression.

Figure A.44: Translation of \texttt{null} in multiarray contexts.
A.4.24 Translating null

The keyword null can appear in various contexts where it must be promoted to a multiarray-valued expression. Specifically these contexts are:

- on the right-hand-side of an assignment expression (section A.4.10), where the variable on the left-hand-side is a multiarray variable,
- as one operand of a reference equality test (section A.4.23), where the other operand is a multiarray value,
- as the operand of a cast (section A.4.21) to a multiarray type, or
- as the operand of an instanceof test (section A.4.22) for a multiarray type\(^7\).
- as the argument of a method or constructor, where a multiarray parameter is expected.
- as the expression in a return statement, where the method has multiarray return type.

In these contexts, the (trivial) translation functions are given in Figure A.44.

\(^7\)It evaluates trivially to false, but it is allowed.
A.4.25 Support for generic library methods

The HPJava language itself doesn't currently provide an integrated mechanism for dealing with multiarrays of unspecified rank. In principle the language could be extended to provide support for such arrays. The set of operations on the envisaged abstract multiarrays would be limited: one would not be able to create an instance of an abstract multiarray directly, and no normal subscripting operations would be allowed on an expression of this type. However the standard inquiries on multiarrays would be available. Through the kind of low-level programming described in chapter 7 it would be possible to directly manipulate abstract multiarrays within the HPJava language.

A proposed syntax for abstract multiarray type signatures has the form:

\[ T \# brs \]

where \( T \) is any Java type other than an array type, and the term \( brs \) is a string of zero or more bracket pairs, \([\] \). This signature represents the abstract multiarray type of a multiarray whose elements have Java type

\[ T \ brs \]

In general it would probably be allowed to declare variables of abstract multiarray type; references could be passed to and returned from methods and constructors; there would be a widening conversion from a concrete multiarray type to an abstract multiarray type (provided there was a widening conversion between the respective component types). It should be possible to test concrete types of referenced multiarrays with the instanceof operator, and to cast expressions of abstract multiarray type back to concrete multiarray types. Equality tests comparing an abstract multiarray reference to another, or to a concrete multiarray reference, may be provided, and interconvertible null references might be supported.

This level of support requires fairly extensive changes to the translation rules given in this appendix. For now the HPJava translator supports these abstract multiarray types in a much more restrictive way. There is no special syntax within the HPJava language, but it is possible for externally defined methods and constructors to have parameters of abstract multiarray type. Here "externally defined" presumably means "implemented in standard Java" (following HPJava-compliant calling conventions). This extension requires minor changes to the HPJava type checker, and only one change to the translation scheme: in the definition of the macro \( TRANS_ARG \) in section A.4.15, if \( e \) is a multiarray expression of rank \( R \), and the corresponding formal parameter of the method
or constructor is an abstract multiarray, then\(^8\)

\[
\text{TRANS\_ARG}(c) \equiv \\
T_{\text{bas}[c]}, \text{new ArrayDim} ([] \{ T_0[c], \ldots, T_{R-1}[c] \}, T_{\text{dat}[c]})
\]

The Java implementation should accept an array of ArrayDim objects for the multiarray.

Generic element type

The current HPJava translator also allows library routines to be written (in Java) in such a way that they are generic with respect to the element type of multiarray arguments. The Java implementation should present a formal parameter of type Object for the elements array. The corresponding actual argument (provided by the translator according to the scheme in section A.4.15) will be a Java array—the Java implementation of the method or constructor will presumably test the actual element type at execution time.

This feature was introduced specifically to provide better support for the Adlib \texttt{aprintf()} routine. It does not require any change to the translation scheme described so far—it only involves a change to the HPJava type checker.

\(^8\text{We already see a subtlety with null references. If }c\text{ is a null reference, then the resulting abstract multiarray ought to be interpreted as null. But here it involves an array of }R\text{ nulls. So the abstract multiarray resulting from two null arrays of different rank would have different structure, although both should presumably be interpreted as the null reference for abstract multiarrays.}\)
public interface Hpspmd {
}

Figure A.45: The Hpspmd interface.

public abstract class Group {
    public boolean amMember() {...}
    public boolean contains(Dimension d) {...}
    public Group restrict(Dimension d, int crd) {...}
    public Group restrict(Range x, int n) {...}
    public APOGroup restrict(Dimension d) {...}
    public APOGroup toAPG() {...}
    public void checkContained(APOGroup apg) {...}
    public DimSet getDimSet() {...}
    ...
}

Figure A.46: The Group class.

A.5 Support Classes

Figures A.45 through A.65 list the most important parts of the various classes and interfaces used in the HPJava translation scheme.

The classes defined in figures A.45 through A.60 can be freely used in an HPJava source program. The classes defined in figures A.61 through A.65 are effectively reserved by the HPJava translator: they will appear in generated code, but should not appear in a program input to the translator.
public class Procs extends Group implements HPSmd {
    public Procs(int [] shape) {...}
    public Procs(int rank) {...}
    public Dimension dim(int r) {...}
}

Figure A.47: The Procs class.

public class Procs0 extends Procs {
    public Procs0() {
        super(0);
    }
}

Figure A.48: The Procs0 class.

public class Procs1 extends Procs {
    public Procs1(int p) {
        super(new int [] {p});
    }
    public Procs1() {
        super(1);
    }
}

Figure A.49: The Procs1 class.

public class Procs2 extends Procs {
    public Procs2(int p1, int p2) {
        super(new int [] {p1, p2});
    }
    public Procs2() {
        super(2);
    }
}

Figure A.50: The Procs2 class.
public class DimSet {

    public void remove(Dimension d) {...}

} ...
public abstract class Range {
    public int size() {...}

    public int base() {...}
    public int str() {...}

    public int volume() {...}
    public int format() {...}

    public Dimension dim() {...}

    public Range shell() {...}

    public int loExtension() {...}
    public int hiExtension() {...}

    public Range subrng(int lo, int hi) {...}
    public Range subrng(int lo, int hi, int step) {...}

    public Location location(int n) {...}

    public Block localBlock() {...}
    public Block localBlock(int lo, int hi) {...}
    public Block localBlock(int lo, int hi, int step) {...}

    public Triplet crds() {...}
    public Block block(int crd) {...}

    public Triplet crds(int lo, int hi) {...}
    public Block block(int crd, int lo, int hi) {...}

    public Triplet crds(int lo, int hi, int step) {...}
    public Block block(int crd, int lo, int hi, int step) {...}

    public boolean isAligned(Range x);

    public boolean containsLocation(Range x, int n) {...}

    public boolean containsLocations(Range x) {...}
    public boolean containsLocations(Range x, int lo, int hi) {...}
    public boolean containsLocations(Range x, int lo, int hi, int step) {...}

    public ArrayDim arrayDim(int stride) {...}

    static final DIST_DIMENSION = ...;
    static final DIST_BLOCK = ...;
    static final DIST_CYCLIC = ...;
    static final DIST_COLLAPSED = ...;
}

Figure A.52: The Range class.
public class Dimension extends Range {
    public int crd() {...}
}

Figure A.53: The Dimension class.

public class BlockRange extends Range {
    public BlockRange(int size, Dimension d) {...}
}

Figure A.54: The BlockRange class.

public class ExtBlockRange extends Range {
    public ExtBlockRange(int size, Dimension d, int wLo, int wHi) {...}
}

Figure A.55: The ExtBlockRange class.

public class CollapsedRange extends Range {
    public CollapsedRange(int size) {...}
}

Figure A.56: The CollapsedRange class.

public class CyclicRange extends Range {
    public CyclicRange(int size, Dimension d) {...}
}

Figure A.57: The CyclicRange class.
public class Block {
    public final int count;
    public final int sub_bas;
    public final int sub_stp;
    public final int glb_bas;
    public final int glb_stp;

    public Block(int count, int sub_bas, int sub_stp, int glb_bas, int glb_stp) {
        this.count  = count;
        this.sub_bas = sub_bas;
        this.sub_stp = sub_stp;
        this.glb_bas = glb_bas;
        this.glb_stp = glb_stp;
    }

    public static final EMPTY = new Block(0, 0, 1, 0, 1);
}

Figure A.58: The Block class.

public class Location {
    public final int crd;
    public final int sub;
    public final Dimension dim;

    public Location(int crd, int sub, Dimension dim) {
        this.crd = crd;
        this.sub = sub;
        this.dim = dim;
    }
}

Figure A.59: The Location class
public class Triplet {
    public final int lo;
    public final int hi;
    public final int stp;

    public Triplet(int lo, int hi, int stp) {
        this.lo = lo;
        this.hi = hi;
        this.stp = stp;
    }

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

Figure A.60: The Triplet class.
public class APOGroup extends Group {
    ...
}

Figure A.61: The APOGroup class.

public class ArrayBase {
    public final Group group ;
    public final int base ;

    public ArrayBase(Group group, int base) {
        this.group = group ;
        this.base = base ;
    }
}

Figure A.62: The ArrayBase class.

public class ArrayDim {
    public final Range range ;
    public final int stride ;

    public ArrayDim(Range x, int stride) {
        this.range = x ;
        this.stride = stride ;
    }
}

Figure A.63: The ArrayDim class.
public class SeqArrayDim extends ArrayDim {
    public final int off_bas;
    public final int off_stp;

    public SeqArrayDim(Range x, int stride) {
        super(x, stride);
        off_bas = stride * (x.longExtent() + x.bas());
        off_str = stride * x.str();
    }

    public SeqArrayDim(int size, int stride) {
        super(new CollapsedRange(size), stride);
        off_bas = 0;
        off_stp = stride;
    }
}

Figure A.64: The SeqArrayDim class.

public class DAD {
    public ArrayBase base;
    public ArrayDim [] dimensions;

    public DAD(int rank) {
        dimensions = new ArrayDim [rank];
    }
}

Figure A.65: The DAD class.
Index

T # notation, 83, 234
<> subscript, 103

abstract syntax tree, 145
access set, 145
active process group, 26, 73, 83, 133
Adlib, 7, 32, 81
alignment, of distributed arrays, 70,
71, 76, 85, 87–89
all(), method of Adlib, 87
allDim(), method of Adlib, 88
anonymous class creation expression,
164
any(), method of Adlib, 87
anyDim(), method of Adlib, 88
APG, 26, 83, 133
APGGroup class, 243
aprintf(), method of Adlib, 90
array of multiarrays, unsupported,
16, 132
ArrayBase class, 243
ArrayDim class, 243
at construct, 74, 167, 183

bas(), inquiry on multiarray, 115,
224
bas(), method of Range, 126
Block class, 108, 241
block distribution format, 29, 41
block(), method of Range, 112, 127
BlockRange class, 29, 41, 240
bounds checking, on multiarrays, 18,
75
brackets, double, 13
broadcast, 58, 71, 87
broadcast(), method of Adlib, 85
C#, 13
canonical HPspmd style, 79
cast expression, 227
CHAOS, 81
class variables, avoiding use of, 39,
79
coherent expression, 79, 83, 90
coherent variable, 79
collapsed distribution format, 52
CollapsedRange class, 52, 240
Collective operation, 24, 29
communication schedule, 93
compilation, of HPJava programs,
10
composite expression, 142
composite multiarray expression, 138
compound assignment, 148
conditional evaluation, in expressions,
152
constructor declaration, 179
count process, 28, 38, 59
copy(), method of HPutil, 53
count(), method of Adlib, 87
countDim(), method of Adlib, 88
crds() method of Range, 127
cshift(), method of Adlib, 53, 82, 85
cyclic distribution format, 41
CyclicRange class, 41, 240
Cygwin, 8

DAD class, 244
dat(), inquiry on multiarray, 115,
224
dim(), method of Procs, 69
Dimension class, 26, 101, 240
dimension splitting, 103
DimensionNotInGroupException, 29, 74, 75, 183, 185
DimSet class, 195, 238
distributed array, 28
distributed array creation expression, 29
distributed array element reference, 33, 75, 199
distributed array restriction, 71, 217
distributed array; type signature, 29
distributed control construct, 35
distributed index, 33, 35, 137
distributed range, 28
distribution group, 29, 53, 55
do statement, 160
dotProduct(), method of Adlib, 71, 87
element types, of multiarrays, 16
examples, programming, 8
exceptions, exact, 154
explicit constructor invocation, 163
expression simplification, 140
ExtBlockRange class, 44, 47, 240
extent, of a multiarray dimension, 13

field declaration, 158, 171
for statement, 160
forall statement, Fortran, 32
format(), method of Range, 124
Fortran, 7, 13, 32, 86
Fourier transform, 10, 63
gather(), method of Adlib, 88
Gauss-Seidel relaxation, 47
ghost regions, 44, 85, 112
Global Arrays Toolkit, 81
Global Grid Forum, 24
global index expression, 35, 187
global subscript, 103
gprint(), method of Adlib, 93
gprintf(), method of Adlib, 93
grid, process, 24
Group class, 24, 28, 236
group restriction, 68, 221

GroupkMismatchException, 85
GroupNotContainedException, 73, 75, 181
grp(), inquiry on distributed array, 58, 224

High Performance Fortran, 7, 32
home group, 79
hpjava.numprocs, 11
hpjavac, 11
hpjdlk, 7
HPspmd class, 133
HPspmd code, 134
HPspmd interface, 133, 134, 236
HPspmd method, 134

I/O, 90
if statement, 159
InaccessibleException, 84
incoherent expression, 80, 86, 112, 119
index, distributed, 33, 35
inner class, 164
inspector-executor pattern, 93
instanceof expression, 229
irregular communications, 81, 88
isAligned(), method of Range, 85, 88

Ising model, 47
Java files, intermediate, 11
Java Grande, 13
Java Native Interface, 132
Java Virtual Machine, 11
JNI, 132
JVM, 11

Linux, 8
load balancing, 41
local subscript, 103
local variable declaration, 157, 171
Location class, 241

Mandelbrot set, 43
mapping, of distributed array, 41
INDEX

mapping of distributed array section, 69
mask, in Adlib operations, 88, 90
matrix multiplication, 53, 55, 65, 71
maxloc(), method of Adlib, 87
maxlocDim(), method of Adlib, 88
maxval(), method of Adlib, 37, 86
maxvalDim(), method of Adlib, 87
Merlin, John, 81
method declaration, 175
method invocation, 209
minloc(), method of Adlib, 87
minlocDim(), method of Adlib, 88
minval(), method of Adlib, 87
minvalDim(), method of Adlib, 88
MisalignmentException, 85
Monte Carlo simulation, 47
MPI, 7, 115
mpiJava, 7, 37, 98
mpiJava, installation, 37
mpjdev, 38
multi-process execution model, 22, 37
multiarray creation expression, 13, 193
multiarray element reference, 199
multiarray section, 13, 61, 201
multiarray subtype relations, 53, 132
multiarray type, 16, 131
multiarray, type signature, 13, 29
multiarrays, 12
 multidimensional array, Java, 12
 multiple grids, in same program, 24, 59
 multiply referenced value, 142
 multiply referenced variable, 142
 multi-threaded execution model, 7, 22, 37
N-body problem, 97
null multiarray reference, 16, 233
NullPointerException, 77
Object, communication of, 82
on clause, in distributed array creation, 58
on construct, 24, 73, 167, 181
output, 90
overall construct, 30, 74, 167, 185
overall, communication inside, 84
overloading, communication methods, 82
overloading, w.r.t. multiarray rank, 83, 234
parallel computer, 7
parallel prefix, 120
parallel scan operation, 120
PCRC project, 81
pre-translation, 138
process grid, 24
Procs class, 26, 28, 237
Procs0 class, 27, 237
Procs1 class, 27, 52, 237
Procs2 class, 24, 237
ProcsArrayTooLargeException, 24
product(), method of Adlib, 87
productDim(), method of Adlib, 88
random number generators, 49
range checking, on multiarrays, 18, 75
Range class, 14, 28, 239
rank, generic, in method signatures, 83, 234
rank, of a multiarray, 13
rank-0 multiarray, 18, 70, 86
RankMISMATCHException, 84
recursive translation, 112
red-black relaxation, 47, 94
reduction operation, 86
reference, equality, of multiarrays, 231
regular section, of a multiarray, 17
renap(), method of Adlib, 53, 55, 70, 82
replicated distribution, 53
restricted group, 68, 221
restriction, of distributed array, 71, 217
return statement, 163, 213
rng(), inquiry on multiarray, 14, 58, 225
run-time checks, unimplemented, 76

ScALAPACK, 81
scalar subscript, 18, 203, 205
scatter(), method of Adlib, 88
schedule classes, of Adlib, 94
schedule, communication, 93
section expression, 17
section, of a multiarray, 17, 61, 201
SeqArrayDim class, 244
sequential dimension, of distributed array, 52
serialization, use in communication, 82
shape, of multiarray, 82
ShapeMismatchException, 84
shift(), method of Adlib, 32, 82, 84
shifted index, subscript, 47
SHPF, 81
simple expression, 138
simplify algorithm, 144
size(), method of Range, 14, 32
skew operation, 84
splitting subscript, 103, 201
statement expression, 154
static context, 164, 166
static fields, avoiding use of, 39, 79
stencil updates, 44
str(), inquiry on multiarray, 115, 225
str(), method of Range, 112, 126
subarray, 17
subrange, 67, 223
sum(), method of Adlib, 87
sumDim(), method of Adlib, 88
Swing, Java, 11
switch statement, 159
synchronized statement, 159

throw statement, 163
top-level assignment, 139
translation function, 168
translation schema, 168
triplet, 17, 30
Triplet class, 129, 242
triplet subscript, 18, 201
type signature, multiarray, 13, 29, 53, 131

UNIX, 8

vector subscript, unsupported, 18
volume(), method of Range, 67, 195
while statement, 160
Windows, Microsoft, 8, 37
Wolf program, 10, 38, 63
writeHalo(), method of Adlib, 47, 82, 85