Contents

Chapter 1

The *HPspmd* Model and its Java Binding

GUANSONG ZHANG, BRYAN CARPENTER, GEOFFREY FOX,
XINYING LI, AND YUHONG WEN

NPAC at Syracuse University
Syracuse, NY, 13 244

Email: {*zgs,dbc,gcf,xli,wen*}*@npac.syr.edu*

1.1 Introduction

In this chapter we introduce the *HPJava* language, a programming language that extends Java for parallel programming on message passing systems — from multi-processor systems to workstation clusters.

HPJava owes much to High Performanace Fortran (HPF) [4]. Its model of data distribution is adapted directly from the HPF model. The heritage of HPF can be traced back to Fortran dialects that were implemented most successfully on SIMD and other tightly-coupled MPP architectures. While it was always a goal of the HPF designers that the language should be efficiently implementable on the more loosely coupled MIMD clusters that dominate today, the complexity of the language — and notably the design goal of emulating exactly the semantics of a sequential Fortran program — have made efficient implementation on today's architectures quite hard.

HPJava, in contrast, starts from the assumption that the target hardware is a set of interacting MIMD processors, and exposes that assumption explictly in its programming model. This greatly simplifies the task of the compiler, and increases the chance of obtaining efficient implementations on architectures including PC and workstation clusters. Instead of the HPF programming model, the language introduces a high-level structured SPMD programming style — the HPspmd model. A program written in this class of language explicitly coordinates well-defined process groups. These cooperate in a loosely synchronous manner, sharing logical threads of control. As in a conventional distributed-memory SPMD program, only a process owning a data item such as an array element is allowed to access the item directly. The language provides special constructs that allow programmers to meet this constraint conveniently.

Besides the normal variables of the sequential base language, the language model introduces classes of global variables that are stored collectively across process groups. Primarily, these are distributed arrays. They provide a global name space in the form of globally subscripted arrays, with assorted distribution patterns. This helps to relieve programmers of error-prone activities such as the local-to-global, global-to-local subscript translations which occur in data parallel applications.

In addition to special data types, the language provides special constructs to facilitate both data parallel and task parallel programming. Through these constructs different processors can work either simultaneously on globally addressed data, or independently to execute complex procedures on locally held data. The conversion between these phases is seamless.

In the traditional SPMD mold, the language itself does not provide implicit data movement semantics. This greatly simplifies the task of the compiler, and should encourage programmers to use algorithms that exploit locality. Data on remote processors is accessed exclusively through explicit library calls. In particular, the initial HPJava implementation relies on a library of collective communication routines originally developed as part of an HPF run-time library. Other distributed-array-oriented communication libraries may be bound to the language later. Due to the explicit SPMD programming model, low level MPI communication is always available as a fallback. The language itself only provides basic concepts to organize data arrays and process groups. Different communication patterns are implemented as library functions. This allows the possibility that if a new communication pattern is needed, it is relatively easily integrated through new libraries.

In our earlier work on HPF compilation [10] the role of run-time support was emphasized. Difficulties in compiling HPF efficiently suggested to make the run-time communication library directly visible in the programming model. Since Java is a simple, elegant language, we are implementing our prototype based upon this language.

Section 1.2 reviews the HPspmd model in the context of the HPJava language. Section 1.3 describes the class library packages used in code generated by the HPJava translator, and thus exposes many of the implementation issues. Some examples of simple algorithms expressed in HPJava are given in Section 1.4. Then Section 1.5 discusses the rationale of various design decisions in the language. The status of the project and future goals are summarized in Sections 1.6 and 1.7.

1.2 Java Language Binding

This section introduces the HPJava language. HPJava contains the whole of standard Java as a subset. It adds various built-in classes for describing process groups and index ranges, new global data types, and some syntax for accessing distributed data and specifying which processes execute particular statements.

1.2.1 Basic Concepts

Key concepts in the programming model are built around the process groups used to describe program execution control in a parallel program. `Group` is a class representing a process group, typically with a grid structure and an associated set of process dimensions. It has its subclasses that represent different grid dimensionalities, such as `Procs1`, `Procs2`, etc. For example,

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

`p` is a 2-dimensional, 2 x 4 grid of processes.

The second category of concepts is associated with distributed ranges. The elements of an ordinary array can be represented by an array name and an integer sequence. There are two parameters associated with this sequence: an index to access each array element, and the extent of the range this index can be chosen from. In describing a distributed array, HPJava introduces two new kinds of entity to represent the analogous concepts. A *range* maps an integer interval into a process dimension according to certain distribution format. Ranges describe the extent and the mapping of array dimensions. A *location*, or slot, is an abstract element of a range. For example,

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

creates two ranges distributed over the two process dimensions of the group `p`. One is block distributed, the other is cyclic distributed. There are 100 different locations in the range `x`. The first one, for example, is

```
x [0]
```

Additional related concepts are subgroups and subranges. A subgroup is some slice of a process array, formed by restricting the process coordinates in one or more dimensions to single values. Suppose `i` is a location in a range distributed over a dimension of group `p`. The expression

```
p / i
```

represents a smaller group—the slice of `p` to which location `i` is mapped. Similarly, a subrange is a section of a range, parameterized by a global index triplet. Logically, it represents a subset of the locations of the original range. The syntax for a subrange expression is

```
x [1 : 49]
```

The symbol "`:`" is a special separator. It is used to construct a Fortran-90 style triplet, with lower- and upper-bound expressions defining an integer subset. The optional third member of a triplet is a stride.

When a process grid is defined, certain ranges and locations are also implicitly defined. As shown in Figure 1.1, two primitive ranges are associated with dimensions of the group `p`:

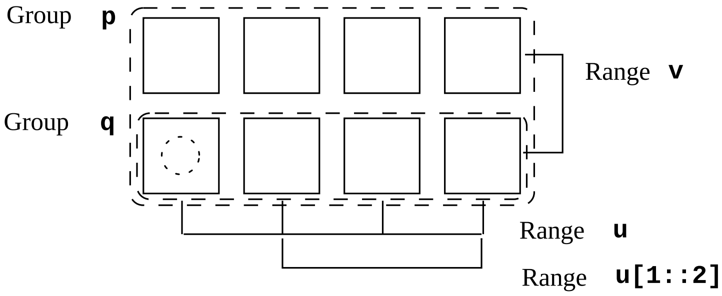

Figure 1.1 Structured process group.

```
Range u = p.dim(0);
Range v = p.dim(1);
```

`dim()` is a member function that returns a range reference, directly representing a processor dimension. We can obtain a location in range `v`, and use it to create a new group,

```
Group q = p / v [1] ;
```

In a traditional SPMD program, execution control is based on *if* statements and process ID or rank numbers. In the new programming language, switching execution control is based on the structured process group. For example, it is not difficult to guess that the following code:

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

will restrict the execution control inside the bracket to processes in group `p`. The language also provided well-defined constructs to split execution control across processes according to data items we want to access. This will be discussed later.

1.2.2 Global Variables

When an SPMD program starts on a group of n processes, there will be n control threads mapped to n physical processors. In each control thread, the program can define variables in the same way as in a sequential program. The variables created in this way are local variables. Their names may be common to all processes, but they will be accessed individually (their scope is local to a process).

Besides local variables, HPJava allows a program to define global variables, explicitly mapped to a process group. A global variable will be treated by the process group that created it as a single entity. The language has special syntax for the definition of global data. Global variables are all defined by using the `new` operator from free storage. When a global variable is created, a data descriptor is

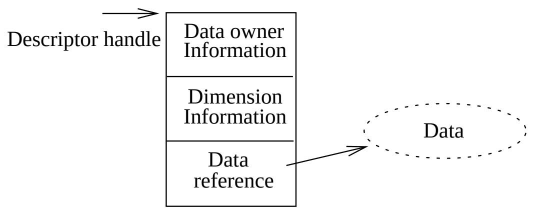

Figure 1.2 Descriptor.

also allocated to describe where the data are held. On a single processor, an array variable might be parametrized by a simple record containing a memory address and an `int` value for its length. On a multiprocessor, a more complicated structure is needed to describe a distributed array. The data descriptor specifies where the data is created, and how they are distributed. The logical structure of a descriptor is shown in Figure 1.2.

HPJava has special syntax to define global data. The statement

```
int # s = new int # on p ;
```

creates a global scalar replicated over process group `p`. In the statement, `s` is a data descriptor handle — a global scalar reference. The scalar contains an integer value. Global scalar references can be defined for any primitive type (or, in principle, class type) of Java. The symbol `#` in the type signature distinguishes a global scalar from a primitive integer. For a global scalar, a field `value` is used to access the value:

```
on(p) {
  int # s = new int # ;
  s.value = 100 ;
}
```

Note how the `on` clause can be omitted from the constructor: the whole of the active process group is the default distribution group. Figure 1.3 shows a possible memory

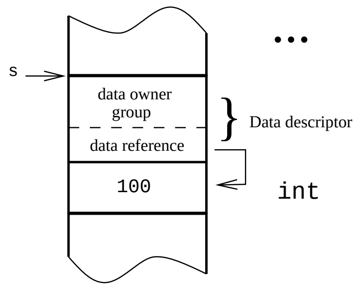

Figure 1.3 Memory mapping.

mapping for this scalar on different processes. Note, the value field of s is identical in each process in the distribution group. Replicated value variables are different from local variables with identical names. The associated descriptors can be used to ensure the value is maintained identically in each process, throughout program execution.

When defining a global array, it is not necessary to allocate a data descriptor for each array element—one descriptor suffices for the whole array. An array can defined with various kinds of range, introduced earlier. Suppose we have, as before,

```
Range x = new BlockRange(100, p.dim(0)) ;
```

and the process group defined in Figure 1.1, then

```
float [[]] a = new float [[x]] on q ;
```

will create a global array with range x on group q. Here a is a descriptor handle describing a one-dimensional array of float. It is block distributed on group q.[1] In HPJava a is called a global or distributed array reference.

A distributed array range can also be collapsed (or sequential). An integer range is specified, eg

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

When defining an array with collapsed dimensions an asterisk is normally added in the type signatures to mark the collapsed dimensions.

The typical method of accessing global array elements is not exactly the same as for local array elements, or for global scalar references. In distributed dimensions of arrays we must use named locations as subscripts, for example

```
at(i = x [3])
  a [i] = 3 ;
```

We will leave discussion of the *at* construct to Section 1.2.3, and give a simpler example here: if a global array is defined with a collapsed dimension, accessing its elements is modelled on local arrays. For example:

```
for(int i = 0 ; i < 100 ; i++)
  b [i] = i ;
```

assigns the loop index to each corresponding element in the array.

When defining a multidimensional global array, a single descriptor parametrizes a rectangular array of any dimension:

```
Range x = new BlockRange(100, p.dim(0)) ;
Range y = new CyclicRange(100, p.dim(1)) ;
float [[,]] c = new float [[x, y]];
```

[1] The on clause restricts the data owner group of the array to q. If group p is used instead, the one-dimenstional array will be replicated in the first dimenstion of the group, and block distributed over the second dimension.

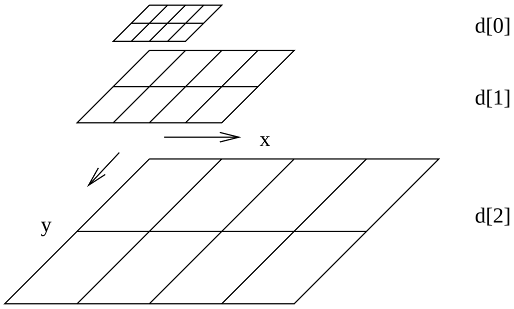

Figure 1.4 Array of distributed arrays.

This creates a two-dimension global array with the first dimension block distributed and the second cyclic distributed. Now `c` is a global array reference. Its elements can be accessed using single brackets with two suitable locations inside.

The global array introduced here is a true multidimensional array, not a Java-like array-of-arrays. Java-style arrays-of-arrays are still useful. For example, one can define a local array of distributed arrays:

```
int[] size = {100, 200, 400};
float [[,]] d[] = new float [size.length][[,]] ;
Range x[], y[];
for (int l = 0; l < size.length; l++) {
  const int n = size [l] ;
  x[l] = new BlockRange(n, p.dim(0)) ;
  y[l] = new BlockRange(n, p.dim(1)) ;
  d[l] = new float [[x[l], y[l]]];
}
```

This creates the stack of distributed arrays shown in Figure 1.4.

Like Fortran 90, HPJava allows construction of sections of global arrays. The syntax of section subscripting uses double brackets. The subscripts can be scalar (integers or locations) or triplets.

Suppose we have array `a` and `c` defined as above. Then `a[[i]]`, `c`, `c[[i, 1::2]]`, and `c[[i, :]]` are all array sections. Here, `i` is an integer in the appropriate interval (it could also be a location in the first range of `a` and `c`). Both the expressions `c[[i, 1::2]]` and `c[[i, :]]` represent one-dimensional distributed arrays, providing aliases for subsets of the elements in `c`. The expression `a[[i]]` contains a single element of `a`, but the result is a global scalar reference (unlike the expression `a[i]` which is a simple variable).

Array section expressions are often used as arguments in function calls.[2] Table

[2]When used in method calls, the collapsed dimension array is a *subtype* of the ordinary one, i.e., an argument of `float[[*,*]]`, `float[[*,]]` and `float[[,*]]` type can all be passed to a dummy of type `float[[,]]`. The converse is not true.

Table 1.1 Section Expression and Type Signature

global var	array section	type
2-dimension	`c`	`float [[,]]`
	`c[[:,:]]`	`float [[,]]`
1-dimension	`c[[i,:]]`	`float [[]]`
	`c[[i,1::2]]`	`float [[]]`
scalar(0-dim)	`c[[i,j]]`	`float #`

1.1 shows the type signatures of global data with different dimensions.

The size of an array in Java can be had from its `length` field. In HPJava, information like the distributed group and distributed dimensions can be accessed from the following inquiries, available on all global array types:

```
Group grp()       // distribution group

Range rng(int d)  // d'th range
```

Further inquiry functions on `Range` yield values such as extents and distribution formats.

1.2.3 Program Execution Control

HPJava has all the conventional Java statements for execution control within a single process. It introduces three new control constructs: *on*, *at*, and *overall* for execution control across processes. A new concept, the active process group, is introduced. It is the set of processes sharing the current thread of control.

In a traditional SPMD program, switching the active process group is effectively implemented by *if* statements such as:

```
if(myid >= 0 && myid < 4) {
  ...
}
```

Inside the braces, only processes numbered 0 to 3 share the control thread. In HPJava, this effect is expressed using a `Group`. When a HPJava program starts, the active process group has a system-defined value. During the execution, the active process group can be changed explicitly through an *on* construct in the program.

In a shared memory program, accessing the value of a variable is straightforward. In a message passing system, only the process which holds data can read and write the data. We sometimes call this SPMD constraint. A traditional SPMD program respects this constraint by using an idiom like

```
if(myid == 1)
  my_data = 3 ;
```

The *if* statement makes sure that only `my_data` on process 1 is assigned as 3.

In the language we present here, similar constraints must be respected. Besides *on* construct introduced earlier, there is a convenient way to change the active process group to access a required array element, namely, the *at* construct. Suppose array `a` is defined as in the previous section, then:

```
on(q) {
  a [1] = 3 ;     // error

  at(j = x [1])
    a [j] = 3 ;  // correct
}
```

The assignment statement guarded by an *at* construct is correct; the one without is likely to imply access to an element not held locally. Formally it is illegal because, in a simple subscripting operation, an integer expression cannot be used to subscript a distributed dimension. The *at* construct introduces a new variable `j`, a *named location*, with scope only inside the block controlled by the *at*. Named locations are the only legal element subscripts in distributed dimensions.

A more powerful construct called *overall* combines restriction of the active process group with a loop:

```
on(q)
  overall(i = x | 0 : 3)
    a [i] = 3 ;
```

is essentially equivalent to[3]

```
on(q)
  for(int n = 0 ; n < 4 ; n++)
    at(i = x [n])
      a [i] = 3;
```

In each iteration, the active process group is changed to `q / i`. In Section 1.4, we will illustrate with further programs how *at* and *overall* constructs conveniently allow one to keep the active process group equal to the data owner group for the assigned data.

1.2.4 Communication Library Functions

When accessing data on another process, HPJava needs explicit communication, as in a normal SPMD program. Communication libraries are provided as packages in HPJava. Detailed function specifications are given elsewhere. The next section will introduce a small number of top level collective communication functions.

[3]A compiler can implement overall construct in a more efficient way, using linearized address calculation. For detailed translation schemes for the overall construct, please refer to [2].

1.3 Java Packages for HPspmd Programming

The implementation of the HPJava compiler is based on a run-time system. It is actually a source-to-source translator converting an HPJava program to a Java node program, with function calls to the run-time library, called *adJava*.

The run-time interface consists of several Java packages. The most important one is the HPspmd runtime proper. It includes the classes needed to translate language constructs. Other packages provide communication and some simple I/O functions. Important classes in the first package include distributed array "container classes" and related classes describing process groups and index ranges. These classes correspond directly to HPJava built-in classes.

The first hierarchy is based on `Group`. A group, or process group, defines some subset of the processes executing the SPMD program. They can be used to describe how program variables, such as arrays, are distributed or replicated across the process pool, or to specify which subset of processes executes a particular code fragment. Important members of adJava `Group` class include the pair `on()`, `no()` used to translate the *on* construct.

The most common way to create a group object is through the constructor for one of the subclasses representing a process grid. The subclass `Procs` represents a grid of processes and carries information on process dimensions: in particular an inquiry function `dim(r)` returns a range object describing the r-th process dimension. `Procs` is further subclassed by `Procs0`, `Procs1`, `Procs2`, ... which provide simpler constructors for fixed dimensionality process grids.

The second hierarchy in the package is based on `Range`. A range is a map from the integer interval $0, \ldots, n-1$ into some process dimension (i.e., some dimension of a process grid). Ranges are used to parametrize distributed arrays and the *overall* distributed loop.

The most common way to create a range object is to use the constructor for one of the subclasses representing ranges with specific distribution formats. Simple block distribution format is implemented by `BlockRange`, while `CyclicRange` and `BlockCyclicRange` represent other standard distribution formats of HPF. The subclass `CollapsedRange` represents a sequential (undistributed range). Finally, a `DimRange` is associated with each process dimension and represents the range of coordinates for the process dimension itself—just one element is mapped to each process.

The related adJava class `Location` represents an individual location in a particular distributed range. Important members of the adJava `Range` class include the function `location(i)`, which returns the ith location in a range, and its inverse, `idx(l)`, which returns the global subscript associated with a given location. Important members of the `Location` class include `at()` and `ta()`, used in the implementation of the HPJava that *at* construct.

Finally, we have the rather complex hierarchy of classes representing distributed arrays. HPJava global arrays declared using `[[]]` are represented by Java objects belonging to classes such as:

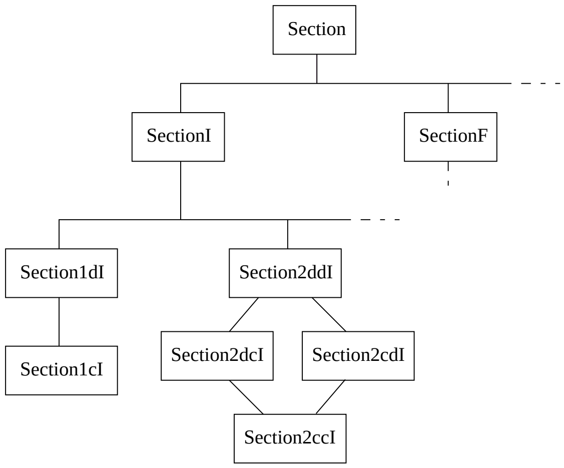

Figure 1.5 The adJava `Section` hierarchy

```
Array1dI, Array1cI,
Array2ddI, Array2dcI, Array2cdI, Array2ccI,
...
Array1dF, Array1cF,
Array2ddF, Array2dcF, Array2cdF, Array2ccF,
...
```

Generally speaking, the class "`Array`n`c|d`... T" represents n-dimensional distributed arrays with elements of type T—currently one of `I`, `F`, ..., meaning `int`, `float`, ... [4] The penultimate part of the class name is a string of n "c"s and "d"s specifying whether each dimension is collapsed or normally distributed. These correlate with presence or absence of an asterisk in slots of the HPJava type signature. The concrete `Array...` classes implement a series of abstract interfaces. These follow a similar naming convention, but the root of their names is `Section` rather than `Array` (so `Array2dcI`, for example, implements `Section2dcI`). The hierarchy of `Section` interfaces is illustrated in Figure 1.5. The need to introduce the `Section` interfaces should be evident from the hierarchy diagram. The type hierarchy of HPJava involves a kind of multiple inheritance. The array type `int [[*, *]]`, for example, is a specialization of both the types `int [[*, ]]` and `int [[, *]]`. Java allows "multiple inheritance" only from interfaces, not classes.

Important members of the `Section` interfaces include inquiry functions `dat()`, which returns an ordinary one dimensional Java array used to store the locally held elements of the distributed array, and the member `pos(i, ...)`, which takes n subscript arguments and returns the local offset of the element implied by those

[4] In the inital implementation, the element type is restricted to the Java primitive types.

subscripts. Each argument of `pos` is a location or an integer (only allowed if the corresponding dimension is collapsed). These functions are used to implement elemental subscripting. The inquiry `grp()` returns the group over which elements of the array are distributed. The inquiry `rng(d)` returns the dth range of the array.

Another package in adJava is the communication library. The adJava communication package includes classes corresponding to the various collective communication schedules provided in the NPAC PCRC kernel. Most of them provide a constructor to establish a schedule, and an `execute` method, which carries out the data movement specified by the schedule. Different communication models may eventually be added through further packages.

The collective communication schedules can be used directly by the programmer or invoked through certain wrapper functions. A class named `Adlib` is defined with static members that create and execute communication schedules and perform simple I/O functions. This class includes, for example, the following methods, each implemented by constructing the appropriate schedule and then executing it:

```
static void remap(Section dst, Section src)
static void shift(Section dst, Section src, int shift, int dim, int mode)
static void copy(Section dst, Section src)
static void writeHalo(Section src, int [] wlo, int [] whi, int [] mode)
```

`Adlib.remap` will copy the corresponding elements from one array to another, regardless of their respective distribution format. `Adlib.shift` will shift data by a certain amount in a specific dimension of the array, in either cyclic or edge-off mode. `Adlib.writeHalo` is used to update ghost regions.

Given the classes described above, one can program in the HPspmd style in pure Java program. Then the idea of the HPJava compiler is just to translate the HPJava program onto this interface, in the meanwhile, to use optimized address calculation instead of function calls to access elements of arrays. (For a detailed translation scheme, please refer to [11].)

1.4 Programming Examples

In this section we give two example programs to show the new language features. The first example is the Choleski decomposition, seen in Figure 1.6. Here, `remap` is used to broadcast one updated column to each process. The function `idx` gets the global index of location `m` relative to the parent range `x`. The second example is the Jacobi iteration, seen in Figure 1.7. In the displayed code there is only one iteration, but it demonstrates how to define range references with ghost areas, how to use the `writeHalo` function, and how to use shifted locations as subscripts.

1.5 Issues in the Language Design

With some of the implementation mechanisms exposed, we can better discuss the language design itself.

```
Procs1 p = new Procs1(4);
on(p) {
  Range x = new CyclicRange(n, p.dim(0));
  float a[[*,]] = new float [[n, x]];
  ... some code to initialise 'a' ...
  float b[[*]] = new float [[n]]; // buffer

  for(int k = 0 ; k < N - 1 ; k++) {
    at(l = x[k]) {
      float d =  Math.sqrt(a[k,l]) ;
      a[k,l] = d ;
      for(int s = k + 1 ; s < N ; s++)
        a[s,l] /= d ;
    }
    Adlib.remap(b[[k+1:]], a[[k+1:,  k]]);

    overall(m = x | k + 1 : )
      for(int i = x.idx(m) ; i < N ; i++)
        a[i,m] -= b[i] * b[x.idx(m)] ;
  }
  at(l = x [N - 1])
    a[N - 1,l] = Math.sqrt(a[N-1,l]) ;
}
```

Figure 1.6 Choleski decomposition.

```
Procs2 p = new Procs2(2, 4);
Range x = new BlockRange(100, p.dim(0), 1),
      y = new BlockRange(200, p.dim(1), 1);
on(p) {
  float [[,]] a = new float [[x,y]], b = new float [[x,y]];
  ... some code to initialize 'a'

  Adlib.writeHalo(a);

  overall(i = x | : )
    overall(j = y | : )
      b[i,j] = 0.25 * (a[i-1,j] + a[i+1,j] + a[i,j-1] + a[i,j+1]);
  overall(i = x | : )
    overall(j = y | : )
      a[i,j] = b[i,j];
}
```

Figure 1.7 Jacobi relaxation.

1.5.1 Extending the Java Language

The first question to answer is why use Java as a base language? Actually, the programming model embodied in HPJava is largely language-independent. It can be bound to other languages like C, C++, and Fortran. But Java is a convenient base language, especially for initial experiments, because it provides full object-orientation — convenient for describing complex distributed data — implemented in a relatively simple setting, conducive to development of source-to-source translators. It has been noted elsewhere that Java has various features suggesting it could be an attractive language for science and engineering [6].

With Java as a base language, an obvious question is whether we can extend the language by simply adding packages, instead of changing the syntax. There are two problems with doing this for data-parallel programming.

Our baseline is HPF, and any package supporting parallel arrays as general as HPF is likely to be cumbersome to code with. Our run-time system needs an (in principle) infinite series of class names

```
Array1dI, Array1cI, Array2ddI, Array2dcI, ...
```

to express the HPJava types

```
int [[]], int [[*]], int [[,]], int [[,*]] ...
```

as well as the corresponding series for `char`, `float`, and so on. To access an element of a distributed array in HPJava, one writes

```
a[i] = 3 ;
```

In the adJava interface, it must be written as

```
a.dat()[a.pos(i)] = 3 ;
```

This is only for simple subscripting. Constructing array sections will be even more complex using the raw class library interface.

The second problem is that a Java program using a package like adJava in a direct, naive way will have very poor performance, because all the local address of the global array are expressed by functions such as `pos`. An optimization pass is needed to transform offset computation to a more intelligent style. So if a preprocessor must do these optimizations anyway, it makes sense to design a syntax to express the concepts of the programming model more naturally.

1.5.2 Why not HPF?

The design of the HPJava language is strongly influenced by HPF. The language emerged partly out of practices adopted in our efforts to implement an HPF compilation system [10]. For example:

```
!HPF$ POCESSOR    P(4)
!HPF$ TEMPLET     T(100)
!HPF$ DISTRIBUTE  T(BLOCK) ONTO P
```

```
      REAL         A(100,100), B(100)
!HPF$ ALIGN        A(:,*) WITH T(:)
!HPF$ ALIGN        B WITH T
```

have their conterparts in HPJava:

```
Procs1 p = new Procs1(4);
Range x = new BlockRange (100, p.dim(0));
float [[,*]] a = new float [[x,100]] on p;
float [[ ]] b = new float [[x]] on p;
```

Both languages provide a globally addressed name space for data parallel applications. Both of them can specify how data are mapped on to a processor grid. The difference between the two lies in their communication aspects. In HPF, a simple assignment statement may cause data movement. For example, given the above distribution, the assignment

```
A(10,10) = B(30)
```

will cause communication between processor 1 and 2. In HPJava, similar communication must be done through explicit function calls: [5]

```
Adlib.remap(a[[9,9]], b[[29]]);
```

Experience from compiling the HPF language suggests that, while there are various kinds of algorithms to detect communication automatically, it is often difficult to give the generated node program acceptable performance. In HPF, the need to decide on which processor the computation should be executed further complicates the situation. One may apply "owner computes" or "majority computes" rules to partition computation, but these heuristics are difficult to apply in many situations.

In HPJava, the SPMD programming model is emphasized. The distributed arrays just help the programmer organize data, and simplify global-to-local address translation. The tasks of computation partition and communication are still under control of the programmer. This is certainly an extra onus, and the language may be more difficult to program than HPF; [6] but it helps programmers to understand the performance of the program much better than in HPF, so algorithms exploiting locality and parallelism are encouraged. It also dramatically simplifies the work of the compiler.

Because the communication sector is considered an "add-on" to the basic language, HPJava should interoperate more smoothly than HPF with other successful SPMD libraries, including MPI [5], Global Arrays [7], CHAOS [3], and so on.

1.5.3 Datatypes in HPJava

In a parallel language, it is desirable to have both local variables (like the ones in MPI programming) and global variables (like the ones in HPF programming). The

[5] By default Fortran array subscripts starts from 1, while HPJava global subscripts always start from 0.

[6] The program must meet SPMD constraints, e.g., only the owner of an element can access that data. Run-time checking can be added automatically to ensure such conditions are met.

former provide flexibility and are ideal for task parallel programming; the latter are convenient especially for data parallel programming.

In HPJava, variable names are divided into two sets. In general those declared using ordinary Java syntax represent local variables and those declared with `#` or `[[]]` represent global variables. The two sectors are independent. In the implementation of HPJava the global variables have special data descriptors associated with them, defining how their components are divided or replicated across processes. The significance of the data descriptor is most obvious when dealing with procedure calls. Passing array sections to procedure calls is an important component in the array processing facilities of Fortran90 [1]. The data descriptor of Fortran90 will include stride information for each array dimension. One can assume that HPF needs a much more complex kind of data descriptor to allow passing distributed arrays across procedure boundaries. In either case the descriptor is not visible to the programmer. Java has a more explicit data descriptor concept; its arrays are considered as objects, with, for example, a publicly accessible `length` field. In HP-Java, the data descriptors for global data are similar to those used in HPF, but more explicitly exposed to programmers. Inquiry functions such as `grp` and `rng` have a similar role in global data as the field `length` in an ordinary Java array.

Keeping two data sectors seems to complicate the language and its syntax. But it provides convenience for both task and data parallel processing. There is no need for things like the `LOCAL` mechanism in HPF to call a local procedure on the node processor. The descriptors for ordinary Java variables are unchanged in HPJava. On each node processor ordinary Java data will be used as local varables, as in an MPI program.

1.6 Projects in Progress

Projects related to this work include development of MPI, HPF, and other parallel languages such as ZPL[8] and Spar[9]. Here, we explain the background and future developments of our own project.

The work originated in our compilation practices for HPF. As described in [10], our compiler emphasizes run-time support. Adlib[2], a PCRC run-time library, provides a rich set of collective communication functions. It was realized that by raising the run-time interface to the user level, a rather straightforward (compared to HPF) compiler could be developed to translate the high level language code to a node program calling the run-time functions.

Currently, a Java interface has been implemented on top of the Adlib library. With classes such as `Group`, `Range`, and `Location` in the Java interface, one can write Java programs quite similar to HPJava proposed here. But a program executed in this way will have large overhead due to function calls (such as address translation) when accessing data inside loop constructs. Given the knowledge of data distribution plus inquiry functions inside the run-time library, one can substitute address translation calls with linear operations on the loop variable, and keep inquiry function calls "outside the loop."

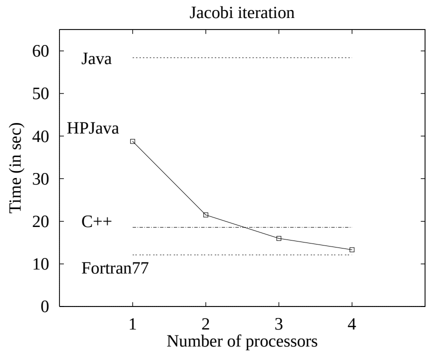

Figure 1.8 Preliminary performance.

At the present time, we are implementing the translator. Further research work will include optimization and safety-checking techniques in the compiler for HPspmd programming.

Figure 1.8 shows a preliminary benchmark for a hand-translated versions of our examples. The parallel programs are executed on 4 sparc-sun-solaris2.5.1 with MPICH and the Java JIT compiler in JDK 1.2Beta2. For Jacobi iteration, the timing is for about 90 iterations; the array size is 1024X1024.

We also compared the sequential Java, C++, and Fortran version of the code, all with `-O` flag when compiling. The dotted lines shown in the figure only represent times for the one processor case. We can see that on a single processor, Java program use language supported mechanism for calculating array element address; hence, it is slower than HPJava (as HPJava uses an optimized address calculation scheme). We emphasize again that in the picture we are comparing *sequential* Fortran, etc., with *parallel* HPJava. This is not supposed to be an comparative evaluation of the various languages. It is just supposed to give an impression of the performance ballpark Java is currently operating in.

1.7 Summary

Through the simple examples in this chapter we have tried to illustrate that the programming language presented here provides the flexibility of SPMD programming, and much of the convenience of HPF. The language helps programmers to express parallel algorithms in a more explicit way. We suggest it will help programmers to solve real application problems more easily, compared with using communication packages such as MPI directly, and allow the compiler writer to implement the language compiler without the difficulties met in the HPF compilation.

The overall structure of the system is shown in Figure 1.9. The central DAD stands for the Distributed Array Descriptor. Around it, we have different run-

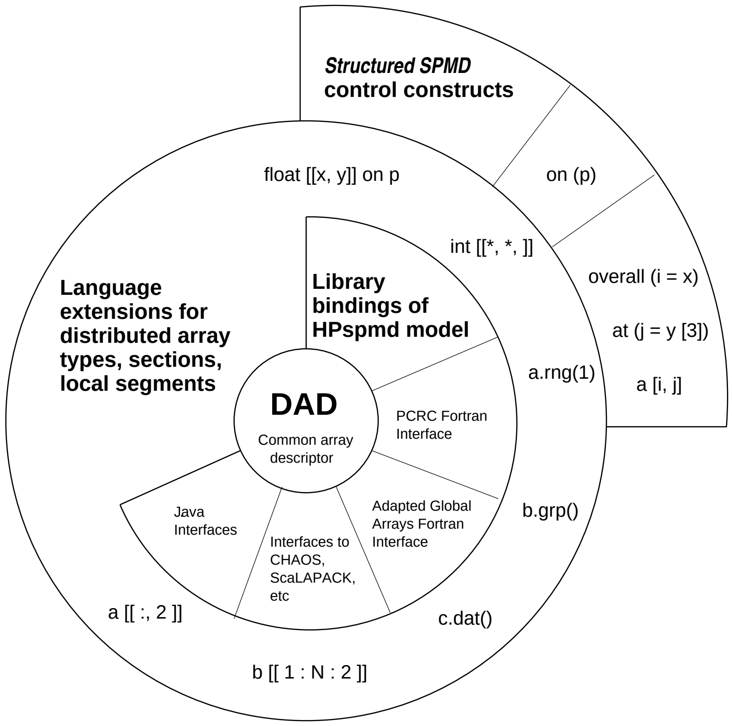

Figure 1.9 Layers of the HPspmd model.

time libraries. The Java interface is most relevant here. But the Java binding is only an introduction of the programming style. (A Fortran binding is being developed.) Initially, the Java version can be used as a software tool for teaching parallel programming. As Java for scientific computation becomes more mature, it will be a practical programming language to solve real application problems in parallel and distributed environments.

1.8 Bibliography

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, 1992.

[2] Bryan Carpenter, Guansong Zhang, and Yuhong Wen. NPAC PCRC Runtime Kernel Definition. Technical Report CRPC-TR97726. Center for Research on Parallel Computation, 1997. Up-to-date version maintained at `http://www.npac.syr.edu/projects/pcrc`.

[3] R. Das, M. Uysal, J.H. Salz, and Y.-S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Aemory Architectures. *Journal of Parallel and Distributed Computing*, vol. 22(3), pages 462–479, September 1994.

[4] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, special issue, February 1993.

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. `http://www.mcs.anl.gov/mpi`.

[6] Geoffrey C. Fox, editor. *ACM 1998 Workshop on Java for High-Performance Network Computing*. February 1998.

[7] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. The Global Array: Non-Uniform-Memory-Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, vol. 10, pages 197–220, 1996.

[8] Lawrence Snyder. A ZPL Programming Guide. Technical Report, University of Washington, May 1997. `http://www.cs.washington.edu/research/projects/zpl/`.

[9] Kees van Reeuwijk, Arjan J. C. van Gemund, and Henk J. Sips. Spar: A Programming Language for Semi-Automatic Compilation of Parallel Programs. *Concurrency: Practice and Experience*, vol. 9(11), pages 1193–1205, 1997.

[10] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. PCRC-Based HPF Compilation. In *10th International Workshop on Languages and Compilers for Parallel Computing*, August 1997. To appear in Lecture Notes in Computer Science.

[11] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xinying Li, and Yuhong Wen. Considerations in HPJava Language Design and Implementation. In *11th International Workshop on Languages and Compilers for Parallel Computing*, August 1998. To appear in Lecture Notes in Computer Science.