Considerations in HPJava language design and implementation

Guansong Zhang, Bryan Carpenter, Geoffrey Fox
Xinying Li and Yuhong Wen

NPAC at Syracuse University
Syracuse, New York,
NY 13244, USA
{zgs,dbc,gcf,xli,wen}@npac.syr.edu

Abstract. This paper discusses some design and implementation issues in the *HPJava* language. The language is briefly reviewed, then the class library that forms the foundation of the translation scheme is described. Through example codes, we illustrate how HPJava source codes can be translated straightforwardly to ordinary SPMD Java programs calling this library. This is followed by a discussion of the rationale for introducing the language in the first place, and of how various language features have been designed to facilitate efficient implementation.

1 Introduction

HPJava is a programming language extended from Java to support parallel programming, especially (but not exclusively) data parallel programming on message passing and distributed memory systems, from multi-processor systems to workstation clusters.

Although it has a close relationship with HPF [5], the design of HPJava does not inherit the HPF programming model. Instead the language introduces a high-level structured SPMD programming style—the *HPspmd* model. A program written in this kind 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 either work simultaneously on globally addressed data, or independently 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 runtime 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 fall-back. 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.

The preceding paragraphs attempt to characterize a language independent programming style. This report only briefly sketches the HPJava language. For further details, please refer to [2, 15]. Here we will discuss in more depth some issues in the language design and implementation. With the pros and cons explained, the language can be better understood and appreciated.

Since it is easier to comment on the language design with some knowledge of its implementation, this document is organized as follows: section 2 briefly reviews the HPJava language extensions; section 3 outlines a simple but complete implementation scheme for the language; section 4 explains the language design issues based on its implementation; finally, the expected performance and test results are given.

2 Overview of HPJava

Java already provides parallelism through threads. But that model of parallelism can only be easily exploited on shared memory computers. HPJava is targetted at distributed memory parallel computers (most likely, networks of PCs and workstations).

HPJava extends Java with class libraries and some additional syntax for dealing with *distributed arrays*. Some or all of the dimensions of a these arrays can be declared as *distributed ranges*. A distributed range defines a range of integer subscripts, and specifies how they are mapped into a process grid dimension. It is represented by an object of base class `Range`. Process grids—equivalent to processor arrangements in HPF—are described by suitable classes. A base class `Group` describes a general group of processes and has subclasses `Procs1`, `Procs2`, ..., representing one-dimensional process grids, two-dimensional process grids, and so on. The inquiry function `dim` returns an object describing a particular dimension of a grid. In the example

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

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

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

`a` is created as a 100×200 array, block-distributed over the 6 processes in `p`. The `Range` subclass `BlockRange` describes a simple block-distributed range of subscripts—analogous to `BLOCK` distribution format in HPF. The arguments of the `BlockRange` constructor are the extent of the range and an object defining the process grid dimension over which the range is distributed.

In HPJava the type-signatures and constructors of distributed arrays use double brackets to distinguish them from ordinary Java arrays. Selected dimensions of a distributed array may have a collapsed (sequential) ranges rather than a distributed ranges: the corresponding slots in the type signature of the array should include a `*` symbol. In general the constructor of the distributed array is followed by an `on` clause, specifying the process group over which the array is distributed. (If this is omitted the group defaults to the APG, see below.) Distributed ranges of the array must be distributed over distinct dimensions of this group.

A standard library, *Adlib*, provides functions for manipulating distributed arrays, including functions closely analogous to the array transformational intrinsic functions of Fortran 90. For example:

```
float [[,]] b = new float [[x, y]] on p ;
Adlib.shift(b, a, -1, 0, CYCL) ;

float g = Adlib.sum(b) ;
```

The `shift` operation with shift-mode `CYCL` executes a cyclic shift on the data in its second argument, copying the result to its first argument. The `sum` operation simply adds all elements of its argument array. In general these functions imply inter-processor communication.

Often in SPMD programming it is necessary to restrict execution of a block of code to processors in a particular group `p`. Our language provides a short way of writing this construct

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

The language incorporates a formal idea of an active process group (APG). At any point of execution some group is singled out as the APG. An `on(p)` construct specifically changes its value to `p`. On exit from the construct, the APG is restored to its value on entry.

Subscripting operations on distributed arrays are subject to some restrictions that ensure data accesses are local. An array access such as

```
a [17, 23] = 13 ;
```

is forbidden because typical processes do not hold the specified element. The idea of a *location* is introduced. A location can be viewed as an abstract element, or "slot", of a distributed range. The syntax `x [n]` stands for location `n` in range `x`. In simple array subscripting operations, distributed dimensions of arrays can only be subscripted using locations (not integer subscripts). These must be locations in the appropriate range of the array. Moreover, locations appearing in simple subscripting operations must be *named locations*, and named locations can only be scoped by *at* and *overall* constructs.

The *at* construct is analogous to *on*, except that its body is executed only on processes that hold the specified location. The array access above can be safely written as:

```
at(i = x [17])
  at(j = y [23])
    a [i, j] = 13 ;
```

Any location is mapped to a particular slice of a process grid. The body of the *at* construct only executes on processes that hold the location specified in its header.

The last *distributed control* construct in the language is called *overall.* It implements a distributed parallel loop, and is parametrized by a range. Like *at*, the header of this construct scopes a named location. In this case the location can be regarded as a parallel loop index.

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

overall(i = x)
  overall(j = y)
    a [i, j] = 2 * b [i, j] ;
```

The body of an *overall* construct executes, conceptually in parallel, for every location in the range of its index. An individual "iteration" executes on just those processors holding the location associated with the iteration. Because of the rules about use of subscripts, the body of an *overall* can usually only combine elements of arrays that have some simple alignment relation relative to one another. The `idx` member of `Range` can be used in parallel updates to yield expressions that depend on global index values.

Other important features of the language include Fortran-90-style regular array sections (*section construction* operations look similar to simple subscripting operations, but are distinguished by use of double brackets), an associated idea of *subranges*, and *subgroups*, which can be used to represent the restricted APG inside *at* and *overall* constructs.

The language extensions are most directly targetted at data parallelism. But an HPJava program is implicitly an SPMD Java program, and task parallelism is available by default. A structured way to write a task parallel program is to write an overall construct parametrized by a process dimension (which is a particular kind of range). The body of the loop executes once in each process. The body can execute one or more "tasks" of arbitrary complexity. Task parallel programming with distributed arrays can be facilitated by extending the standard library with one-sided communication operations to access remote patches of the arrays, and we are investigating integration of software from the PNNL Global Array Toolset [8] in this connection.

3 Translation scheme

The initial HPJava compiler is implemented as a source-to-source translator converting an HPJava program to a Java node program, with calls to runtime functions. The runtime system is built on the NPAC PCRC runtime library [3], which has a kernel implemented in C++ and a Java interface implemented in Java and C++.

3.1 Java packages for HPspmd programming

The current runtime interface for HPJava is called *adJava*. It consists of two Java packages. The first is the HPspmd runtime proper. It includes the classes needed to translate language constructs. The second package provides communication and some simple I/O functions. These two packages will be outlined in this section.

The classes in the first package include an environment class, distributed array "container classes", and related classes describing process groups and index ranges. The environment class `SpmdEnv` provides functions to initialize and finalize the underlying communication library (currently MPI). Constructors call native functions to prepare the lower level communication package. An important field, `apg`, defines the group of processes that is cooperating in "loose synchrony" at the current point of execution.

The other classes in this package 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. Groups have two important roles in HPJava. First they are used to describe how program variables such as arrays are distributed or replicated across the process pool. Secondly they are used to specify which subset of processes execute 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 class hierarchy of groups and process grids is shown in figure 1.

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 (ie, some dimension of a process grid). Ranges are used to parametrize distributed arrays

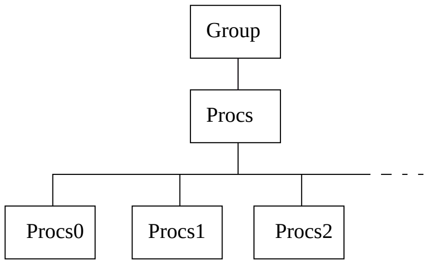

Fig. 1. The HPJava `Group` hierarchy

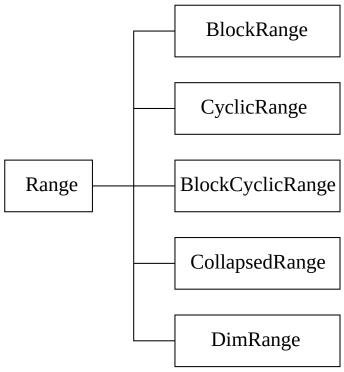

Fig. 2. The HPJava `Range` hierarchy

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*. The current class hierarchy is given in figure 2. 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, `DimRange` represents the range of coordinates of a 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 in this package 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:

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

Generally speaking the class `Array`$ndc\ldots T$ represents n-dimensional distributed array with elements of type T, currently one of `I`, `F`, ..., meaning `int`, `float`, ...[1]. The penultimate part of the class name is a string of n "c"s and "d"s specifying whether each dimension is collapsed or 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 3. The need to introduce the `Section`

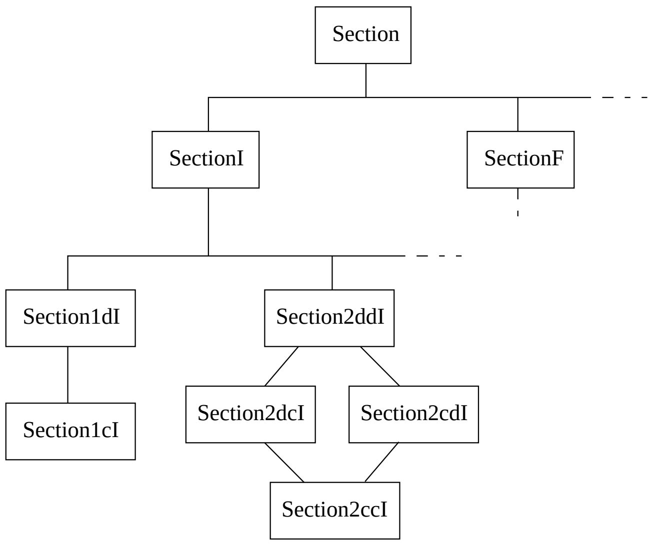

Fig. 3. The adJava `Section` hierarchy

interfaces should be evident from the hierarchy diagram. The type hierarchy of HPJava involves a kind of multiple inheritance. The array type `int [[*, *]]`,

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

for example, is a specialization of *both* the types `int [[*, ]]` and `int [[, *]]`. Java allows "multiple inheritance" only from interfaces, not classes.

We will illustrate constructors of the `Array` classes in later examples. Here we mention some important members of the `Section` interfaces. The inquiry `dat()` returns an ordinary one dimensional Java array used to store the locally held elements of the distributed array. The member `pos(i, ...)`, which takes n arguments, returns the local offset of the element specified by its list of arguments. Each argument is either a location (if the corresponding dimension is distributed) or an integer (if it is collapsed). 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.

The second 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 of a constructor to establish a schedule, and an `execute` method, which carries out the data movement specified by the schedule. The communication schedules provided in this package are based on the NPAC runtime library. 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 public void remap(Section dst, Section src)
static public void shift(Section dst, Section src,
                         int shift, int dim, int mode)
static public void copy(Section dst, Section src)
static public void writeHalo(Section src,
                             int[] wlo, int[] whi, int[] mode)
```

Use of these functions will be illustrated in later examples. Polymorphism is achieved by using arguments of class `Section`.

3.2 Programming in the adJava interface

In this section we illustrate through an example—Fox's algorithm [11] for matrix multiplication—how to program in the adJava interface. We assume A and B are square matrices of order n, so $C = AB$ is also a square matrix of order n. Fox's algorithm organizes A, B and C into sub-matrices on a P by P process array. It takes P steps. In each step, a sub-matrix of A is broadcast across each row of the processes, a local block matrix product is computed, and array B is shifted for computation in the next step.

We can program this algorithm in HPJava, using `Adlib.remap` to broadcast submatrices, `Adlib.shift` to shift array B, and `Adlib.copy` to copy data back after shifting. The HPJava program is given in figure 4. The subroutine `matmul` for local matrix multiplication will be given in the next section.

This HPJava program is slightly atypical: it uses arrays distributed explicitly over process dimensions, rather than using higher-level ranges such as `BlockRange` to describe the distribution of the arrays. Hence, two-dimensional matrices are represented as four dimensional arrays with two distributed ranges (actually process dimensions) and two collapsed ranges (spanning the local block). This simplifies the initial discussion.

```
Procs2 p = new Procs2(P,P);
Range x = p.dim(0), y = p.dim(1);
on(p) {
  float [[,,*,*]] a = new float [[x,y,B,B]];
  float [[,,*,*]] b = new float [[x,y,B,B]];

  ... initialize a, b elements ...

  float [[,,*,*]] c = new float [[x,y,B,B]];
  float [[,,*,*]] tmp = new float [[x,y,B,B]];

  for (int k = 0; k<P; k++) {
    overall(i = x) {
      float [[*,*]] sub = new float [[B,B]];
      Adlib.remap(sub, a[[i, (x.idx(i) + k) % P, :, :]]);
                         // Broadcast sub-matrix of 'a'
      overall(j = y)
        matmul(c[[i, j, :, :]], sub, b[[i, j, :, :]]);
                         // Local matrix multiplication
    }
    Adlib.shift(tmp, b, 1, 0, CYCLIC);
                         // Cyclic shift 'b' in first dim, amount 1
    Adlib.copy(b, tmp);
  }
}
```

Fig. 4. Algorithm for matrix multiplication in HPJava

We can rewrite the program in pure Java language using our adJava interface. A translation is given in figure 5. This is an executable Java program. One can use (for example) `mpirun` to start Java virtual machines on P^2 processors and let them simultaneously load the `Fox` class. This naive translation uses *for* loops plus *at* constructs to simulate the *overall* constructs. The function pairs `on`,`no` and `at`,`ta` adjust the field `spmd.apg`, which records the current active process group. The dynamic alteration of this group plays an non-trivial role in this program. The call to `remap` implements a broadcast because the temporary `sub` is replicated over the process group active at it's point of declaration. Within the `overall(i = x)` construct, the locally effective APG is a row of the process grid. The rather complex code for section construction exposes various low-level inquiries (and one auxilliary class, `Map`) from the adJava runtime. The details are not particulary important here.

3.3 Improving the performance

The program for the Fox algorithm is completed by the definition of `matmul`. First in HPJava:

```
void matmul (float[[*,*]] c, float[[*,*]] b, float[[*,*]] c) {
  for (int i=0; i<B; i++)
    for (int j=0; j<B; j++)
      for (int k=0; k<B; k++)
        c[i,j]+=a[i,k]*b[k,j];
}
```

Translated naively to the adJava interface, this becomes:

```
public static void matmul(Section2ccF c, Section2ccF a, Section2ccF b) {

  for (int i=0; i<B; i++)
    for (int j=0; j<B; j++)
      for (int k=0; k<B; k++)
        c.dat()[c.pos(i, j)] +=
            a.dat()[a.pos(i, k)] * b.dat()[b.pos(k, j)];
}
```

The methods `dat` and `pos` were introduced earlier.

It is clear that the segment of code above will have very poor run-time performance, because it involves many method invocations for each array element access. Because the array data is actually stored in a certain regularly strided section of a Java array, these calls are not really necessary. All that is needed is to find the address of the first array element, then write the other addresses as a linear expression in the loop variable and this initial value. The code above can be rewritten in the form given in figure 6. This optimization again exposes various low-level functions in the runtime—we omit details (see [3]). The effect is to compute the parameters of the linear expressions for the local address offsets. This allows inlining of the `element` calls. In this case the resulting expressions are linear in the induction variables of the *for* loops. If necessary the multiplications can be eliminated by standard compiler optimizations.

This segment of Java code will certainly run much faster. The drawback is that, compared with the first Java procedure, the optimized code is less readable. This is a simple example of the need for compiler intervention if the HPJava style of programming is to be made acceptable. Similar and more compelling examples arise in optimization of the overall construct. As described in [15] and illustrated in the example of the last section, a trivial implementation of the general overall construct is by a *for* loop surrounding an *at* construct. More sensibly, all the machines across a process dimension should simultaneously execute the body for all locally held locations in the relevant distributed range. Computation of the local offset of the array element can again be reduced to a linear expression in a loop variable instead of a function call.

```
class Fox {
  final static int P=2;
  final static int B=4;

  public static void matmul(Array2Float c,Array2Float a,Array2Float b) {
    ... implemented in next section ...
  };

  public static void main(String argv[]) {
    SpmdEnv spmd = new SpmdEnv(argv);
    Procs2 p=new Procs2(P,P);
    Range x=p.dim(0); Range y=p.dim(1);
    if(p.on()) {
      Section4ddccF a = new Array4ddccF(spmd.apg,x,y,B,B);
      Section4ddccF b = new Array4ddccF(spmd.apg,x,y,B,B);

      ... initialize a, b elements ...

      Section4ddccF c = new Array4ddccF(spmd.apg,x,y,B,B);
      Section4ddccF tmp = new Array4ddccF(spmd.apg,x,y,B,B);

      for (int k=0; k<P; k++) {
        for (int i=0; i<P; i++) {
          Location ii = x.location(i);
          if (ii.at()) {
            Section2ccF sub = new Array2ccF(spmd.apg,B,B);
            Location kk = a.rng(1).location((i + k) % P) ;
            Adlib.remap(sub,
                new Array2ccF(a.grp().restrict(ii).restrict(kk),
                              a.map(2), a.map(3), a.dat(),
                              a.map(0).offset(ii) + a.map(1).offset(kk)) ;
                                  // Broadcast sub-matrix of 'a'
            for (int j=0; j<P; j++) {
              Location jj = y.location(j);
              if (jj.at()) {
                matmul(new Array2ccF(c.grp().restrict(ii).restrict(jj),
                                     c.map(2), c.map(3), c.dat(),
                                     c.map(0).offset(ii) + c.map(1).offset(jj)),
                       new Array2ccF(b.grp().restrict(ii).restrict(jj),
                                     b.map(2), b.map(3), b.dat(),
                                     b.map(0).offset(ii) + b.map(1).offset(jj))) ;
                          // Local matrix multiplication
              } jj.ta();
            }
          } ii.ta();
        }
        Adlib.shift(tmp, b, 1, 0, 0);
                          // Cyclic shift 'b' in first dim, amount 1
        Adlib.copy(b, tmp);
      }
    }
  }
}
```

Fig. 5. Algorithm for matrix multiplication in adJava

```
public static void matmul(Section2ccF c, Section2ccF a, Section2ccF b) {

  Map c_u0=c.map(0);
  Map c_u1=c.map(1);

  final int i_c_bas=c_u0.disp();
  final int i_c_stp=c_u0.step();
  final int j_c_bas=c_u1.disp();
  final int j_c_stp=c_u1.step();

  ... similar inquiries for a and b ...

  for (int i=0; i<B; i++) {
    for (int j=0; j<B; j++) {
      for (int k=0; k<B; k++) {
        c.data[i_c_bas + i_c_stp * i + j_c_bas + j_c_stp * j] +=
          a.data[i_a_bas + i_a_stp * i + k_a_bas + k_a_stp * k] *
          b.data[k_b_bas + k_b_stp * k + j_b_bas + j_b_stp * j];
      }
    }
  }
}
```

Fig. 6. Optimized translation of `matmul`

4 Issues in the language design

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

4.1 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 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 implementation 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 [7].

With Java as 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 cumbersome to code with. The examples given earlier using the adJava interface illustrate this point. Our runtime 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 for *simple* subscripting. In passing in section 3.2 we noted how even more complex Fortran-90 style regular section construction appeared using the raw class library interface.

The second problems 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, as suggested in section 3.3. So if a preprocessor must do these optimizations anyway, it makes most sense to design a set of syntax to express the concepts of the programming model more naturally.

4.2 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 [14]. 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[2]:

```
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 is more difficult to program than HPF[3]; but it helps programmer 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 [6], CHAOS [4], Global Arrays [8], and so on.

4.3 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 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 `[[ ]]` 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

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

[3] The program must meet SPMD constraints, eg, only the owner of an element can access that data. Runtime checking can be added automatically to ensure such conditions are met.

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 HPJava, the data descriptors for global data are similar to those used in HPF, but more explicitly exposed to programmers. Inquiry functions such as `grp()`, `rng()` have a similar role in global data to 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, like in an MPI program.

4.4 Programming convenience

The language provides some special syntax for the programmer's convenience. Unlike the syntax for data declaration, which has fundamental significance in the programming model, these extensions are purely provide syntactic conveniences.

There are a limited number of Java operators overloaded. A group object can be *restricted* by a location using the `/` operation, and a sub-range or location can be obtained from a range using the `[ ]` operator enclosing a triplet expression or an integer, These pieces of syntax can be considered as shorthand for suitable constructors in the corresponding classes. This is comparable to the way Java provides special syntax support for String class constructor.

Another kind of overloading occurs in *location shift*, which is used to support *ghost regions*. A shift operator `+` is defined between a location and an integer. It will be illustrated in the examples in the next section. This is a restricted operation—it has meaning (and is legal) only in an array subscript expression.

5 Concluding remarks

In this report, we discussed design and implementation issues in HPJava, a new programming language we have proposed. We claim that the language has the flexibility of SPMD programming, and much of the convenience of HPF. Related languages include F– [9], Spar [12], ZPL [10] and Titanium [13]. They all take different approaches from ours. The implementation of HPJava is straightforwardly supported by a runtime library. In the next step, we will complete the HPJava translator and implement further optimizations. At the same time, we plan to integrate further SPMD libraries into the framework.

References

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, Geoffrey Fox, Xinying Li, and Yuhong Wen. Introduction to Java-Ad. http://www.npac.syr.edu/projects/pcrc/HPJava.
3. 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/docs.html.
4. R. Das, M. Uysal, J.H. Salz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.
5. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.
6. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. http://www.mcs.anl.gov/mpi.
7. Geoffrey C. Fox, editor. *ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998. To appear in Concurrency: Practice and Experience.
8. 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*, 10:197–220, 1996.
9. R.W. Numrich and J.L. Steidel. F–: A simple parallel extension to Fortran 90. *SIAM News*, page 30, 1997.
10. Lawrence Snyder. A ZPL programming guide. Technical report, University of Washington, May 1997. http://www.cs.washington.edu/research/projects/zpl/.
11. E Pluribus Unum. *Programming with MPI*. Morgan Kaufmann, 1997.
12. 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*, 9(11):1193–1205, 1997.
13. Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, Concurrency: Practice and Experience, 1998. To appear.
14. 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*, volume 1366 of *Lecture Notes in Computer Science*. Springer, 1997.
15. Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xinying Li, and Yuhong Wen. A high level SPMD programming model: HPspmd and its Java language binding. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, July 1998.

This article was processed using the LaTeX macro package with LLNCS style