Considerations in HPJava language design and implementation

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

111 College Place
NPAC at Syracuse University
Syracuse, NY 13244

{zgs, dbc, gcf, xli, wen}@npac.syr.edu
Fax: (315)4431973

September 14, 1998

Abstract

This report discusses some design and implementation issues in the *HPJava* language. Through example codes, we will illustrate how various language features have been designed to facilitate efficient implementation. This may help programming of real applications in the new style.

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 [1], 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, 3]. 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 subscripting operations (distinct from *section* 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. Finally, 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, 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 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[4], 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 package 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 packages are outlined in the next two sections.

3.1.1 HPJava language support

An environment class `SpmdEnv` provides functions to initialize and finalize the underlying communication library (currently MPI). It can also be used to support multi-threaded programming in HPJava[1]:

[1]In this section, we only list classes and fields or methods most helpful to an understanding of the language implementation. For more complete information, please refer to reference [4].

```
class public SpmdEnv {
  public SpmdEnv(String argv[]) {} // Initialization in main program
  public SpmdEnv(SpmdEnv spdm) {}  // Initialization in threads

  public Group apg ;               // Active Process Group
  ...
}
```

The constructors call native functions to prepare the lower level communication package. The 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.

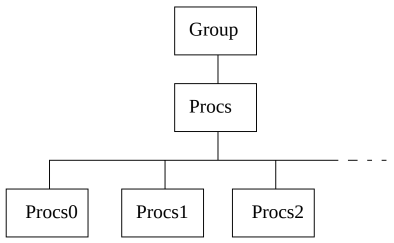

Figure 1: The HPJava `Group` hierarchy

The adJava class corresponding to the HPJava `Group` class is

```
class Group {
  ...
  public boolean on() {}           // Function pair for
  public void no() {}              // translating on construct
}
```

There are several ways to create group objects. The most common is to use 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 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 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. The simple block distribution format implemented by `BlockRange`. `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.

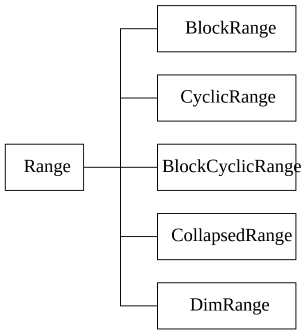

Figure 2: The HPJava `Range` hierarchy

The adJava class `Range` class includes members:

```
class Range {
  ...
  public Location location(int i) {}  // The i-th location
  public int idx(Location loc) {}     // Index of location in range

  public Range triplet(int l,int u,int s) {} // Subrange triplet
}
```

The related adJava class `Location` represents an individual location in a particular distributed range. It has members `at()` and `ta()` used in the implementation of the HPJava that *at* construct.

Finally we have classes for representing global data. HPJava global data declared using `[[ ]]` or `#` is represented by the following Java classes:

```
Array0Int, Array1Int, Array2Int, ...
Array0Float, Array1Float, Array2Float, ...
...
```

Generally speaking `Array`*nType* is used to represent an n-dimensional distributed array with elements of type *type*[2]. As a special case, if n equals zero, the global data is a scalar reference (declared using `#` in HPJava).

We will illustrate the constructors in later examples. Here we list some important fields and members:

```
public type data[];
```

is an ordinary Java array used to store the locally held elements of the distributed array. The member

```
public long element( Location loc0, Location loc1, Location loc2, ...)
```

returns the local address in `data` field from the element of the distributed array specified by the list of locations.

Sectioning operations, akin to the regular array sections of Fortran 90, can be performed on any array. These returne array objects of the same or lower rank. For example, `Array2Int` has members

```
public Array0Int section(Location loc0, Location loc1)
public Array1Int section(Range rng0, Location loc1)
public Array1Int section(Location loc0, Range rng1)
public Array2Int section(Range rng0, Range rng1)
```

In a `section` member a `Location` argument represents a scalar subscript, a `Range` object represents a "triplet subscript".

All `Array`*nType* classes share a base class `Section`. This includes the common fields of `Array`*nType*, such as

```
public Group group ;   // Group over which data is stored
public Range range[] ; // Range for different dimensions
```

These fields can be accessed publicly.

3.1.2 Collective communication library

The adJava commonication 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.

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

The collective communication schedules can be used directly by the programmer or invoked through 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. For example, the class includes the following methods, each implemented by constructing the appropriate schedule and then executing it. Their use will be illustrated in later examples:

```
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)
```

Polymorphism is achieved by using arguments of class `Section`. This class includes a reference to the `data` field of the concrete array subclass, so the communication schedule can access the array elements.

3.2 Programming in the adJava interface

In this section we illustrate through an example—Fox's algorithm [5] 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 3. 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 (process dimensions) and two collapsed range (the local block). This simplifies the initial discussion.

We can rewrite the program in pure Java language using our adJava interface. A translation is given in figure 4. 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`

```
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);
  }
}
```

Figure 3: Algorithm for matrix multiplication in HPJava

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.

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];
}
```

```
import spmd.*;
import spmd.adlib.*;

class Fox {
  final static int P=2;
  final static int B=4;
  final static Range z = new CollapsedRange(B);

  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()) {
      Array4Float a = new Array4Float(x,y,z,z,spmd.apg);
      Array4Float b = new Array4Float(x,y,z,z,spmd.apg);

      ... initialize a, b elements ...

      Array4Float c = new Array4Float(x,y,z,z,spmd.apg);
      Array4Float tmp = new Array4Float(x,y,z,z,spmd.apg);

      for (int k=0; k<P; k++) {
        for (int i=0; i<P; i++) {
          Location ii = x.location(i);
          if (ii.at()) {
            Array2Float sub = new Array2Float(z,z,spmd.apg);
            Adlib.remap(sub, a.section(ii,
                                       a.range[1].location((i+k)%P),
                                       a.range[2],a.range[3]));
                        // Broadcast sub-matrix of 'a'
            for (int j=0; j<P; j++) {
              Location jj = y.location(j);
              if (jj.at()) {
                matmul(c.section(ii,jj,c.range[2],c.range[3]),sub,
                       b.section(ii,jj,b.range[2],b.range[3]));
                        // 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);
      }
    }
  }
}
```

Figure 4: Algorithm for matrix multiplication in adJava

Translated naively to the adJava interface, this becomes:

```
public static void matmul(Array2Float c,
                          Array2Float a, Array2Float b) {
  for (int i=0; i<B; i++)
    for (int j=0; j<B; j++)
      for (int k=0; k<B; k++)
        c.data[c.element(c.range[0].location(i),
                         c.range[1].location(j))] +=
          a.data[a.element(a.range[0].location(i),
                           a.range[1].location(k))] *
          b.data[b.element(b.range[0].location(k),
                           b.range[1].location(j))];
}
```

The methods `element` and `location` were introduced earlier. The information that all three arrays have collapsed ranges is not specified in the adJava program. It was, however, specified in the HPJava program. This extra information can be used to dramatically improve the performance of the translated code.

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 5. This optimization exposes various low-level inquiries (and one auxilliary class, `Stride`) in the adJava runtime. The details are not particularly important here (see [4]). 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 only drawback is that, compared with the first Java procedure, the optimized code is hardly readable. This is a simple example of the need for compiler intervention if this style of programming is to be made acceptable.

Similar optimizations can be applied to the overall construct. As described in [3], a trivial implementation of the general overall construct is through 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.

```
public static void matmul(Array2Float c, Array2Float a, Array2Float b) {
  Range c_r0=c.range[0];
  Range c_r1=c.range[1];
  Stride c_u0=c.stride[0];
  Stride c_u1=c.stride[1];

  final int i_c_bas=c_u0.disp(c_r0.bas());
  final int i_c_stp=c_u0.disp_step(c_r0.str());
  final int j_c_bas=c_u1.disp(c_r1.bas());
  final int j_c_stp=c_u1.disp_step(c_r1.str());

  ... 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];
      }
    }
  }
}
```

Figure 5: Optimized translation of `matmul`

4 Issues in the language design

Once the underlying implementation mechanisms of the language is exposed, a better understanding of the language design itself is possible.

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 [6].

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. The runtime system needs all the class names

```
Array0Int, Array1Int, Array2Int ...
```

to express the HPJava types

```
int #, int[[ ]], int[[,]] ...
```

as well as the corresponding ones for `char`, `float`, and so on. Even if we restrict to two dimensional arrays, expressing subtypes like `int[[*,]]`, `int[[,*]]` or `int[[*,*]]` may require further subclasses of `Array2Int` such as

```
Array2IntCollapsedDistributed
Array2IntDistributedCollapsed
Array2IntCollapsedCollapsed
```

To access an element of a distributed array in HPJava, one writes

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

In the adJava interface, it needs to be written as,

```
a.data[a.element(i)] = 3 ;
```

The second problems is that a Java program using a package like adJava in a direct way will have very poor performance, because all the local address of the global array are expressed by functions such as `element`. 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 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 [7]. 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[3]:

```
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[4]. But this 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, CHAOS, Global Arrays, DAGH, 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 `[[ ]]` or `#` represent global variables. The two sectors are independent. In

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

[4]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.

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 [8]. 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 HPJava, the data descriptors for global data are similar to those used in HPF, but more explicitly exposed to programmers. Inquiry fields such as `group`, `range[]` have the same standing in global data as the field `length` in an ordinary Java array.

In HPJava, an array can be sectioned to yield a "zero-dimensional array" by specifiying all scalar subscripts in double brackets. The result is a global data entity *containing* the associated array element; the result is *not* the element itself. The difference between an array sectioned to a global scalar and an array elment is that the global scalar has a descriptor specifying the process group that holds the element. This one reason why `[[ ]]` is used for the section operation and `[ ]` is used for array element access. The symbol `#` is introduced to augment the type signature of a global scalar reference (a zero-dimensional array) to distinguish it from an ordinary Java scalar.

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.

It is allowed to combine the two different kinds of array (standard Java and distributed) of the language. For example:

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

will create an array like the one shown in figure 6. This facility is useful for multigrid and multiblock algorithms.

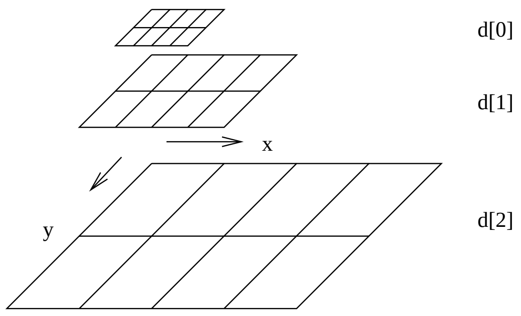

Figure 6: Array of distributed arrays

4.4 Convenience of programming

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, this part is pure syntactic convenience.

First there are a limited number of Java operators overloaded,

- a group reference can be restricted by a location reference with the `/` operation,
- a sub-range or location reference can be mapped by `[ ]` from a range by triplet expression or an integer,

These two items can be considered as shorthand for using suitable constructors in the corresponding classes. This is similar to the way Java provides special syntax support for String class constructor.

Another overloading occurs *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 Example programs

In this section, we give some more interesting examples of HPJava code. The first example is Choleski decomposition, in which one needs to update n submatrices by subtracting an outer product of two vectors. On a parallel computer with a distributed memory, the array may have a column-interleaved storage, then each processor can update its own column after getting the first updated column of the matrix or submatrix.

The above algorithm is written in HPJava in figure 7. In the code, a cyclic range is used to allocate the original array on multiprocessors. The result lower triangular matrix overwrites part of its storage. During the computation, the collective communication `remap` is used for broadcasting updated columns.

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

  float [[*,]] a = new float [[N, x]] ;

  float [[*]]  b = new float [[N]] ;

  ... some code to initialise 'a' ...

  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(l = x[k + 1 : ])
      for(int i = x.idx(l) ; i < N ; i++)
        a[i,l] -= b[i] * b[x.idx(l)] ;
  }

  at(l = x [N - 1])
    a[N - 1,l] = Math.sqrt(a[N-1,l]) ;
}
```

Figure 7: Choleski decomposition in HPJava

The second example is Jacobi iteration. The algorithm calculates the average value of the neighboring elements. A ghost area is defined when the global array is defined through a special `BlockRange` constructor. In the code of figure 8 there is only one iteration. The library function `writeHalo` performs the necessary communications to update ghost edges to make it ready for the iteration.

```
Procs2 p = new Procs2(2, 4);
Range x = new BlockRange(100, p.dim(0), 1);
Range y = new BlockRange(200, p.dim(1), 1);
on(p) {
  float [[,]] a = new int [[x,y]] ;

  ... some code to initialize ‘a’ ...

  float [[,]] b = new int [[x,y]];

  Adlib.writeHalo(a);

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

Figure 8: Jacobi iteration in HPJava

6 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–, Spar, ZPL and Titanium. 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] High Performance Fortran Forum, “High Performance Fortran Language Specification”, version 2.0, Oct. 1996

[2] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xinying Li, and Yuhong Wen. “Introduction to Java-Ad”. http://www.npac.syr.edu/projects/pcrc/doc.

[3] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xinying Li, and Yuhong Wen. "A high level SPMD programming model: *HPspmd* and its Java language binding". http://www.npac.syr.edu/projects/pcrc/doc.

[4] Bryan Carpenter, Guansong Zhang and Yuhong Wen, "NPAC PCRC Runtime Kernel (Adlib) definition", http://www.npac.syr.edu/projects/pcrc/doc

[5] E Pluribus Unum, "Programming with MPI", Morgan Kaufmann Publishers, Inc. 1997.

[6] Geoffery C. Fox, editor. ACM 1998 Workshop on Java for High-Performance Network Computing, Concurrency: Practice and Experience (to appear). Palo Alto, California, Feb. 28 and Mar. 1, 1998. http://www.cs.ucsb.edu/conferences/java98

[7] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong When. "PCRC-based HPF compilation", 10th International Workshop on Languages and Compilers for Parallel Computing, 1997.

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