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

April 20, 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.

Keywords Parallel programming, SPMD, Java

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 variables that are accompanied by non-trivial *data descriptors*. 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 powerful 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 will not introduce the HPJava language itself. For outlines of the language specification, 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 outlines a simple but complete implementation scheme for the language; section 3 explains the language design issues based on its implementation; finally, the expected performance and test results are given.

2 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++ code and a Java interface implemented in Java and C++. Library functions are typically introduced by giving first a standard Java interface for HPspmd parallel programming, then an example of HPJava code, so one can see how to compile (translate) HPJava programs into this interface.

2.1 Java packages for HPspmd programming

The current runtime interface for HPJava is called *adJava*. It consists of a set of standard Java packages, including *spmd* and *spmd.adlib*. The first package is the HPspmd runtime proper. It includes the classes described[1] in the next three subsections 2.1.1, 2.1.2 and 2.1.3. The second package *spmd.adlib* provides communication and some simple I/O functions. It includes classes described in subsections 2.1.4 and 2.1.5.

2.1.1 An environment class.

This provides functions to initialize and finalize the underlying communication library (currently MPI). It also can be used to support multi-threaded programming in HPJava.

```
class public SpmdEnv {
  public SpmdEnv(String argv[]); // Constructor for main program
  public SpmdEnv(SpmdEnv spdm);  // Constructor for threads

  public Group apg;              // Active Process Group
  public ApgStack stack;         // A stack for APG operations
}
```

The constructors will call native functions to prepare the lower level communication package. A native `finalize` method is also needed to clean up the object.

2.1.2 Classes for HPJava built-in types.

These have the same functionalities as in the HPJava language and are used to translate corresponding HPJava classes.

The first hierarchy is based on `Group`:

```
class Group {
  public Group() ;                 // Default constructor

  public boolean on();             // Function pair for
  public void no();                //   translating on construct
```

[1] In this article, 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].

```
  public Group divide(Location i); // Restriction by location
}
```

The Group is a light weight object. The subclass Procs carries more information on process dimensions:

```
class Procs extend Group {
  public Procs();                   // Assorted constructors
  public Procs(int id);             //   ...
  public Procs(int[] grid);
  public Procs(int[] grid, int[] ids);

  public Range dim(int r);          // Range for r-th dim
}
```

This is further subclassed by Procs0, Procs1, Procs2, ... These provide simpler constructors for each rank of process grid.

The second hierarchy is based on Range:

```
class Range {
  public Range() ;                          // Default constructor

  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

  public int size;                          // Extent of the range
}
```

Range has subclasses such as DimRange, CollapsedRange, BlockRange, CyclicRange and BlockCyclikRange.

The third hierachy of HPJava built-in types is based on Location,

```
class Location {
  public Location(Range r, int i); // Constructor

  public boolean at();             // Function pair for
  public void ta();                //   at construct
}
```

2.1.3 Classes for global data.

This sector is a little more complex than the preceding ones. HPJava global data declared using [[ ]] (or #) will be represented by the following Java classes:

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

Generally speaking `Array`*nType* is used to represent an n-dimension 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.

```
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.

Section operations can be performed on an array, returning an array object 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);
```

2.1.4 Communication schedules.

There are different classes corresponding to various collective communication schedules provided in PCRC kernel. Most of them consist 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 PCRC runtime library. Different communication models may eventually be added through further packages.

The collective communication schedules can be invoked through the wrapper class introduced next. (Alternatively they can be used directly by the programmer, allowing *reuse* of the same schedule.)

2.1.5 Communication wrapper class.

A class named `Adlib` is defined, with mainly `static` members, to perform communication schedules and simple I/O functions. For example, the class includes the following methods in the class, each implemented by constructing the corresponding schedule and then executing it. Their use will be illustrated in later examples:

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

```
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);
```

Here `Section` is a class defined in package *spmd*. It is a super class of `Array`*nType*. It records a reference of the `data` field, so that communication schedule can access the array elements. It also records common low level field in `Array`*nType*, such as

```
public Group group;   // Group over which data is stored

public Range range[]; // Range for different dimensions
```

These can be accessed publicly.

2.2 Programming in the adJava interface

In this section we use an example—Fox's algorithm [5] for matrix multiplication—to show how to program in the adJava interface.

Recall that if A and B are square matrices of order n, then $C = AB$ is also a square matrix of order n, and c_{ij} is obtained by taking the dot product of the ith row of A and jth column of B. Fox's algorithm for multiplication organize A, B and C into sub-matrix on a P by P process array. It take P steps: in each step, broadcast corresponding sub-matrix of A on each row of the processes, do local computation and then shift array B for the next step computation.

We can program this algorithm in HPJava, and use `Adlib.remap` to broadcast sub-matrix, `Adlib.shift` to shift array B, and `Adlib.copy` to copy data back after shifting. The HPJava program is given in figure 1. Here, `matmul` is a subroutine for local matrix multiplication, which will be given in the next section. The HPJava program is slightly atypical, in that it uses arrays distributed explicitly over process dimensions, rather than using higher-level ranges such as `BlockRange` as dimensions 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 discussion.

We can rewrite the program in pure Java language using our adJava interface. The result is given in figure 2. This is an executable Java program. One can use `mpirun` to start Java VM on P by P processors and let them simultaneously load this class. In the program, we use function pair `on-no` and `at-ta` to switch `spmd.apg` with the original one push-popped in `spmd.ApgStack`. We also use `for` loops plus `at` constructs to simulate the `overall` constructs.

```
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 1: Algorithm for matrix multiplication in HPJava

2.3 Improving the performance

The program for the Fox algorithm is completed by the definition of the `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 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++)
```

```
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 2: Algorithm for matrix multiplication in adJava

```
        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 programbelow is, however, specified in the HPJava program. We will show below how this can help to improve performance of the translated code.

It is clear that this segment of code will have very poor run-time performance, because it involves many method invocations for each each array element access. Because the array data is actually stored in an contiguous block 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 express the other addresses as a linear expression of the loop variable and this initial value. So one can rewrite the above code in this format as follows:

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

This optimization depends on some features of the HPspmd runtime that were not introduced earlier. The array records contain additional `Stride` objects, akin to the range objects. These encode the layout of elements in local memory (where `Range` encodes their distribution across process space). The calls to the inquires `disp` and `disp_step` depend on some low-level features of the adJava interface which will not be discussed in detail here (see [4]). Their effect is to initialize 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, and 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, relative to the first Java procedure, the complete optimized code is hardly readable. This is a simple example of the need for compiler intervention in this style of programming.

This kind optimization can be generalized 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. Logically, though, the constuct allows all the machines across a process dimension to simultaneously execute the construct body for each location inside the range specified by the construct. Inside the construct body, the active process group will be restricted by the location currently processed. One can safely move the operation on active processing group out of the loop that handles all the locations to be processed on a node processor. Enumeration of locations calculation of the address can be combined, as in the case discussed above. This makes the local offset of the array element a linear expression in a loop variable, instead of function call. Detailed translation patterns and runtime functions to support them will be introduced elsewhere.

3 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.

3.1 Extending the Java language

The first question to answer is why HPJava is extended from Java? Actually, the HPspmd programming model introduced is largely language independent. It can bound to other languages like C/C++ and Fortran too. But Java is actually a particularly convenient base language for initial experiments, because it provides full object-orientation (very convenient for describing complex distributed data) implemented in a simple and familiar procedural setting (very conducive for implementation of source-to-source translators). The Java language is usually discussed for its use in distributed object and Web client based computing extensively. It also has many features that suggest it could be a very attractive language for scientific and engineering, or what we now term “Grande” applications [6]. Probably Java needs improvements both to the language and the support environment to achieve the required linkage of high performance and expressiveness.

If Java is to be extended, an obvious question is whether we can extend the language though standard Java packages such as the adJava interface, instead of changing the language syntax.

There are two reasons why we did not do so. The first one is that the interface of any package for parallel arrays as general as HPF is likely cumbersome to code with. This is illustrated by the adJava interface. For example, since Java does not support template and operator overloading like C++ does, the runtime system needs all the class names like

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

to express the HPJava types

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

as well as the corresponding ones for `char`, `float`, ... Furthermore, for a two dimensional array (say), to express types like `int[[*,]]`, `int[[,*]]` or `int[[*,*]]` one needs subclasses such as

```
Array2IntCollapsedDistributed
Array2IntDistributedCollapsed
Array2IntCollapsedCollapsed
```

as subclasses of `Array2Int`.

To access an array element of a global reference in HPJava, one can write

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

This is the same format used to access an ordinary Java array. In the adJava interface, it needs to be written as,

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

for the same assignment. The second reason to develop special syntax is that a program compiled by Java directly from this kind of coding will have very poor performance, as we discussed above, 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 be to do these optimizations anyway, it makes most sense to design a set of syntax to express the concepts of the programming model more naturally.

3.2 Why not HPF?

The design of HPJava language is strongly influenced by HPF. In fact, 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 global 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,

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

will cause communication between processor 1 and 2. In HPJava, similar communication is done though 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 not clear one can make generated node program have good performance all the time. In HPF, the need to decide on which processor the computation should be executed further complicates the situation, even though one may apply "owner computs" or "majority computes" rule to partition computation.

In HPJava, the SPMD programming model is emphasized. The global addressed data only provide the convenience that one need not do error-prone address translation. The tasks of computation partition and communication are still the responsiblity of the programmer. This is certainly an extra onus, which makes the langauge more difficult to program than HPF[4]. But it helps programmer to understand the performance of the program much better than HPF, so algorithms exploiting locality and parallelism are encouraged. It also helps to simplify the work of 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, etc.

[3] We use the convention that Fortran array addess starts from 1, while Java starts from 0.

[4] The program must meet the SPMD constraints, ie, only the owner of the data can access the data. Runtime checking may be added to ensure this condition. We will discuss this elsewhere.

3.3 Data types in HPJava

In a parallel language, it is desirable to have both *replicated name* variables (like the one in MPI programming) and *global name* variables (like the one in HPF programming). The first provides flexiblity and is suitable for task parallel programming; the second provides convenience, especially for data parallel programming. As summarized in the introduction, this is one goal in our language design.

In HPJava, variable names are divided into two sets. Those declared using ordinary Java represent the first kind of variables. Those declared with `[[ ]]` or `#` are the second kind. The two sectors are independent. The global variables are supported through a *data descriptor*, a concept that is emphasized in the design of HPJava.

The importance of the data descriptor can be best understood when dealing with procedure calls. The passing of array sections into procedure calls represents an important component of the array processing facility in Fortran90 [8], so the data descriptor in Fortran90 can describe the stride information in each array dimension, though the the concept is not explicit in the language definition. One can assume that HPF needs to support a more complex kind of data descriptor to allow passing distributed array. Java has a more explicit data descriptor concept; its arrays are always considered as *reference*, with one of its field `length` always publically accessible.

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

It should be noticed that in HPJava, an array can be sectioned to a zero-dimension array (by specifiying all scalar subscripts). This is still a reference, not a scalar value as in HPF. So, syntacally, there should be difference between an array sectioned to a scalar reference and an array elment. This one reason why "`[[ ]]` is used for 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 this provides convenience for both task and data parallel processing. Also, it simplifies interfaces of procedure calls. 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. 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 3.

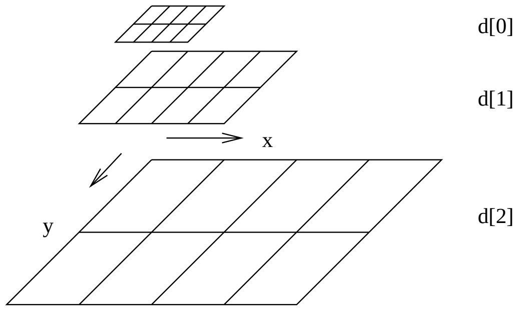

Figure 3: Array of distributed arrays

Finally, we remark that the HPJava multi-dimension array is Fortran-like—not a C-like array of arrays. This format is more appropriate for describing an array distributed on a multi-dimension processor grid.

3.4 Performance enhancement

Since HPJava is a parallel programming language, performance is one of the most important factors we considered during its design. This is reflected in the following issues.

Logically a `Location` is an element of a `Range` labelled by an integer. In principle the underlying concept `Range` should be sufficient. Nevertheless, `Location` has provided as an additional, built-in class. It not only clearly represents a physical concept in a process grid, but also is an intermediate concept to complete the translation from a global address to a local address. It helps program to meet the SPMD constraints in elementatory array access. For example, in HPF, one may well write:

```
FORALL (I=1,50) A(2*I,1)=1
```

In HPJava, this might be written as:

```
overall(i=x|::2)
  a[i,0]=1;
```

This spares the compiler work for analysing loop variable reference expressions in array access.

The `Location` class has a subclass `Index`, in terms of which the overall-construct is defined. The physical significance of this class is less obvious. Actually, it represents an *iterator*, a programming device rather than a language concept. It may appear clearer just to overload the at-constuct and "`=`" symbol to allow, say:

```
at (Location i = range_expression| triplet_expression) {
  ...
}
```

to express a distributed loop. Exposing a separate iterator subclass allows one to write, say:

```
Index i = x|::2 ;
Index j = y|::2 ;
overall(i)
  overall(j)
    a[i,j]=1;
```

The index constructors (written here using the idiosyncratically overloaded `|` operator) can then be lifted outside some enclosing loop. A clever compiler may achieve this optimization after flow analysis, but on the grounds that this analysis is more sophisticated than an initial HPspmd translator would generally need, it was decided to define the language in a way that allows the programmer to explicitly specify reuse of the iterator.

Finally, we recall that the symbol `*` in an array type signature reminds the compiler that a range is collapsed, so array subscripting operations with complex global subscripts can be translated straightforwardly with no extra calls to functions for global-to-local address translation. This is important because such random subscripting operations are very common for sequential array dimensions (whereas distributed dimensions are most often accessed, in parallel, inside overall-constructs, and a different class of optimizations to amortize run-time calls is applicable). For more details please refer to [4].

3.5 Convenience for programming

The language provides some special syntax for the programmers' convenience. Unlike the syntax for data type 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 "`/`" operation,
- a sub-range or location reference can be mapped by "`[ ]`" from a range by triplet expression or an integer,

- an index reference can be constructed from a range reference and a triplet expression by "`|`" operator, and

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

Secondly, the language defines conversions between an integer and a location reference. When accessing a global array element, one can use

```
a[n]
```

to represent

```
a[a.range[0].location(n)]
```

From its definition, one can see this kind of conversion is only meaningful when the integer appears as an array subscript. This idiom should not be used extensively, because it does not help to keep SPMD constraints. It is allowed for consistency with the natural idiom for subscripting collapsed dimensions, and for consistency with the following convenient idiom, which allows triplets to appear as subscripts:

```
a[[l : u : s]]
```

This is equavlent to

```
a[[a.range[0] [l : u : s]]]
```

Unlike the conversions defined between Java primary types, the conversions introduced here can be completed only under certain circumstances. This is sometimes called *restricted type conversion.*

The last special case is *location shift*, which is used to support *ghost region.* A shift operator "+" is defined on a location reference and an integer. It will be illustrated in the examples in the next section. It also is a *restricted* operation, which has its meaning only in an array subscript expression.

4 Example programs and performance

In this part, we give more interesting examples of HPJava code, and their preliminary performance. The HPJava compiler is still under development, so the performance data generally comes from hand translated code, following an established HPJava translation scheme.

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 4. In the code, a cyclic range is used to allocate the original array on multiprocessors. The result L 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' ...

  Location l ;
  Index m ;

  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]]);

    over(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 4: Choleski decomposition in HPJava

The second example is Jacobi iteration. The algorithm needs to calculate the average value of the neighboring elements. So a ghost area is defined when the global array reference is defined through a special `BlockRange` constructor. In the code of figure 5 there is only one iteration. The runtime function `writeHalo` will update the array distributed on different process 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 5: Jacobi iteration in HPJava

Figure 6 shows preliminary benchmarks for hand translated codes of our examples. The parallel programs, denoted by "HPJava", are executed on 1 4 sparc-sun-solaris2.5.1 with mpich MPI and Java JIT compiler in JDK 1.2Beta2. For Jacobi iteration, the timing is for about 90 iterations.

We also compared the sequential Java, C++ and Fortran version of the code, as shown in the figure. The sequential Java is coded in its natural way, using two-dimensial arrays—quite inefficient in Java. That is why it is worse than the single-processor HPJava program.

A similar test was made on an 8-node SGI challenge(mips-sgi-irix6.2), the communication time is much smaller than the one on solaris, due to MPI device using shared memory. Overall performance is not as good, due to the poor JIT compiler on IRIX. The whole system is also being ported to Windows NT.

5 Concluding remarks

In this report, we discussed design and implementation issues in HPJava programming, a new programming style we proposed. We can see that the language

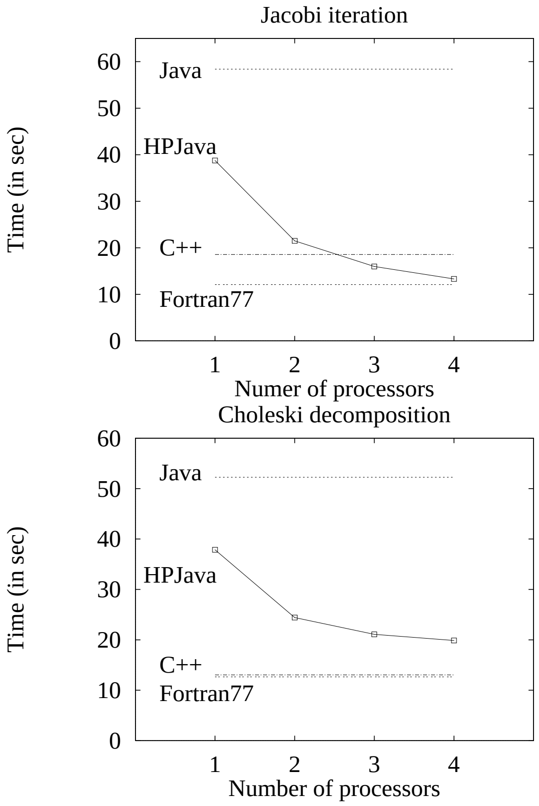

Figure 6: Preliminary performance

presented here has the flexibility of SPMD programming, and much of the convenience of HPF. (Other related languages include F−, Spar, ZPL and Titanium, which all take different approaches from ours.) Its implementation is straightforwardly support by a runtime library. In the next step, we will streamline the HPJava translator and identify further optimization techniques. At the same time, we will try 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