Considerations in HPJava language design and implementation

April 15, 1998

Abstract

In this report, we discuss design and implementation issues in *HPJava* language, an instantiation of *HPspmd* programming model. Through example codes, we want to make it understood that why the new language is designed in this way and how to implement it. We also hope this may help real applications to be programmed in the new programming style and the valuable feedbacks may improve the language itself.

Keywords Parallel programming, SPMD, Java

1 Introduction

HPJava is a programming language extended from Java for parallel programming on message passing systems, from multi-processor system to workstation clusters.

Though its name comes directly from HPF[1], the design of HPJava does not inherit HPF completely. The language introduces a high level SPMD programming style, *HPspmd*, which can be summarized as following,

- **Structured SPMD programming** The programming language presented here depends on a well organized process group as control threads. As in a SPMD program, only the data owners are allowed to share the current control thread when the data items are accessed. The language offerers special constructs for programs to achieve this conveniently.
- **Global name space** Besides local variables, the language provides variables associated with *data descriptor*, offering a global name space, especially, global addressed array with different distribute patterns. This helps to relieve programmers of error-prone activities like local-to-global, global-to-local translations in most data parallel applications.
- **Hybrid of data and task parallel programming** The language provides both different data types and language constructs to facilitate both

data parallel and task parallel programming. Through language constructs, different processors can not only work simultaneously on global addressed data, but also execute independently complex procedures on their own local data. The conversion between these two phases is a seamless one.

- **Collective communication** Accessing data on different processors is through a powerful collective communication routines provided as kernel library with the language. With this library, data parallel applications may have dead-lock free communication. As in SPMD style, the language itself does not provide data movement semantics implicitly. This encourages the programmer to write algorithms that exploit locality and simplifies the task of compiler writers.

- **Flexible for future extension** The language itself only provides basic concepts to organize a group of process grid. Different communication patterns are implemented as library functions. This gives the possibility that when ever a new communication pattern is needed, it is relatively easy to be integrated.

Though these are "buzzwords" just like Java has, they are trying to characterize a language independent programming style. In this report, we will not introduce the HPJava language itself. For the language specification, please refer to [2, 3]. In stead, we will discuss in depth the issues in the language design and implementation. With all the pros and cons explained, we hope the language can be better understood and appreciated not only by ourself, more important that different kinds of feedbacks will be helpful to improve it further.

In fact, it is easier to comment on the language design with the knowledge of its implementation first. So the document is organized as followings, section 2 provides a simple yet complete introduction on implementation scheme of 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

Currently, HPJava compiler is implemented as a language translator to convert an HPJava program to a Java node program, which calls runtime functions. The runtime system are build based upon the PCRC runtime library[4], with its kernel consists of 20K lines of C++ code, and the Java interface on top of it about 10K lines of Java/C++ code. We introduce the library in a let-the-code-explain-itself manner, first, a pure 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

For the moment, the runtime interface for HPJava is called *adJava*. It consists of a set of standard Java packages, currently including *spmd* and *spmd.adlib*.

The first package is the mainframe of the runtime system, it has the following Java classes[1],

- **A package wrapper** This class provides functions to initialize and finalize underlining communication library, currently MPI. It also can be used to support multi-thread 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 processing group
    public ApgStack stack;         // a stack for apg operation
  }
  ```

  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.

  The class `ApgStack`, also provided in the package, is used instead of Java utility class `Stack`. Because its implementation is based on native functions, providing better performance for operating on `apg`, a native `Group` object in PCRC runtime library to record active processing group.

- **Classes for built-in concepts** These classes have the same functionalities as they are in the HPJava language. Actually, they are used to translate corresponding HPJava classes.

  The first category of the classes contains `Group`,

  ```
  class Group {
    // Default constructor
    public boolean on();             // function pair for
    public void no();                //+translating on construct
    public Group divide(Location i); // group restricted by a loc
  }
  ```

  The `Group` is a light weight object which has subclasses carries more information on its dimensions.

  ```
  class Procs extend Group {
    public Procs ();                    // constructors for
    public Procs (int id);              //+different subclass
    public Procs (int[] grid);
  ```

[1] In this report, we only list classes and their fields or methods helpful to understand the language implementation. For more complete information, please refer to our references [4].

```
    public Procs (int[] grid, int[] ids);
    public Range dim(int r);              // range for r-th dim
  }
```

and further subclasses as `Procs0`, `Procs1`, `Procs2`, ..., which will provide simpler constructor for each kind of processor grid.

The second category contains `Range`,

```
  class Range {
    // Default constructor
    public Location location(int i);         // the i-th location
    public Range triplet(int l,int u,int s); // subrange triplet
    public int size;                         // extend of the range
  }
```

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

The third category of the built-in classes contains `Location`,

```
  class Location {
    public Location(Range r, int i); // constructor
    public boolean at();             // function pair for
    public void ta();                //+at construct
  }
```

- **Classes for global data** This part is a little more complex than the above ones. Unlike those built-in classes in HPJava, there are no corresponding class names in the package for global data declared by `#` or `[[ ]]`, they are replaced by the following Java classes,

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

  Generally speaking `Array`*nType* is used to represent an n-dimension global data of Java *Type*[2]. When n equals zero, the global data is a scalar reference, the one declared by `#` in HPJava.

  We will illustrate their constructors in examples later. Here are only some important fields,

  public *Type* `data[]`;

  is where the array elements are stored. And a method

[2]Temporally it is restricted to primary type in Java.

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

will return the local address in `data` field from the element position to be accessed.

Also section operations can be performed on a non-zero dimension array, with a lower dimension array returned. For example, in `Array2Int`, there are,

```
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);
```

These functions are used to support section operation on a global array reference.

In the second package *spmd.adlib*, communication and some simple I/O functions are provided, these include,

- **Communication schedules** There are different classes corresponding to various collective communication schedules provided in PCRC kernel. Most of them consist of a constructor, which will establish a schedule, and a `execute` method, which will carry out the data movement specified by the schedule.

  The communication schedules provided in this package are based on PCRC runtime library. Different communication model may be added as other packages.

  The collective communication schedules are covered by a wrapper class introduced next. Of cause, one can still use them directly to allow possible *reuse* of a communication schedule established.

- **Wrapper class** A class named `Adlib` is defined, with most of its member methods `static`, to perform communication schedules and simple I/O functions. For example, there are following methods in the class, with each one implemented through establishing the corresponding schedule and then executing it. We will further illustrate their usage in 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);
```

  Here `Section` is a class defined in package *spmd*. It is a super class of `Array`*nType*, 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;   // data owner group
public Range range[]; // range for different dimensions
```

which can be accessed publicly.

2.2 Programming in the adJava interface

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

Recall that if A and B are square matrix 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,

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

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

Here, `matmul` is a subroutine for local matrix multiplication, which will be given in the next section. We can rewrite this part in pure Java language with our adJava interface.

```
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);
      //input a, b here;
      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);
            //Broadcast submatrix in 'a' ...
            Adlib.remap(sub,
                 a.section(ii,a.range[1].location((i+k)%P),a.range[2],a.range[3]));
            for (int j=0; j<P; j++) {
              Location jj = y.location(j);
              if (jj.at()) {
                //Local matrix multiplication ...
                matmul(c.section(ii,jj,c.range[2],c.range[3]),sub,
                       b.section(ii,jj,b.range[2],b.range[3]));
              } jj.ta();
            }
          } ii.ta();
        }
        //Cyclic shift 'b' in first dim, amount 1 ...
        Adlib.shift(tmp, b, 1, 0, 0);
        Adlib.copy(b, tmp);
      }
    }
  }
}
```

It should be noticed that this is an executable Java program. One may 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` loop plus `at` construct to simulate `overall` construct. Besides, `range[1]`, `range[2]`, `range[3]` are range `y`, `z`, `z` respectively, they are recorded when the global array were constructed.

2.3 Improving the performance

In this section, we will add the `matmul` function to complete the Fox algorithm. 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];
}
```

Written in the adJava interface, this will be,

```
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))];
}
```

Here both `element` and `location` are both method functions introduced previously. Note that the information all three arrays have collapsed ranges can not be derived from the type signatures of the input arguments, we still use the arrays as if they do. Otherwise, an `overall` construct, not `for` loop, has to be used in each array dimension, and the code will be more complicated[3]. Even though, this segment of code does not have any run time performance. Because a series of functions are called to calculate actual address for each array element access.

Thinking of that the array data is stored as a continuous memory block, these calculations are not really necessary. All one need to do is to find the address of the first array element as initial value, then try to make other address as a linear expression of the loop variable and this initial value. This is the fundamental idea of improving the performance of the code. In fact, functions to calculate the initial value and the correct step-offset pair in the linear expression is designed in the adJava interface. So one can rewrite the above code in this format, and the nested loops may finally become,

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

Where the variables denoted by `_bas` are different initial address values and those by `_stp` are steps of the corresponding loops. They can be calculated from inquiry functions in runtime interface as the execution control enter the procedure.

[3] It should be mentioned that this information is given in HPJava program. We will show later that it can help to improve performance of the translated code.

This segment of Java code will make the program run much more faster. The only drawback is that, in contract to the previous Java procedure, the code is hardly readable. This is the place where compiler work begin to involve.

The idea of this optimization can be generalized to an `overall` construct too. As we introduced in [3], the semantic meaning of the `overall` construct can be expressed by a `for` loop plus a `at` construct. Alternatively speaking, it allows all the machines in this dimension simultaneously execute the construct body for each location inside the range specified by the construct, in the order that is the same as these locations in the range; and inside the construct body, the active processing group will be restricted by the location currently processed. Then one can safely move the operation on active processing group out of the loop which emulates all the locations to be processed on a node processor. Further more, emulation of the locations and calculation of the address can be combined together, as in the case discussed above, making the access pattern of the array element a linear expression of a loop variable, instead of function calls. The only difference is that for an `overall` construct, the translated code will be nested loops not one. Detailed translation patterns and runtime functions to support them will be introduced elsewhere.

3 Issues in the language design

Once the underlining implementation structure of the language is revealed, we can have a better understanding of the language design itself.

3.1 Extending Java language

Above all, the first question to answer is that why it is extended from Java?

Actually, the HPspmd programming model introduced is language independent. it can be binded with other languages like C/C++ and Fortran too.

Java as a new programming language has been 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].

Clearly Java needs many improvements both to the language and the support environment to achieve the required linkage of high performance expressively. This can not be guaranteed but we have helped set in motion a community activity involving academia, government and industry (including IBM, Intel, Microsoft, Oracle, Sun and perhaps most importantly James Gosling from Javasoft) which is designed to both address language changes and the establishment of standards for numerical libraries and distributed scientific objects. The Java environment is still malleable and we are optimistic that this effort will be successful and Java will emerge as a premier language for large scale computation.

If Java is going to be extended, then the question may be that if we can extend the Java language though standard Java packages just as the adJava interface, instead of "messing around" with the language syntax.

There are two reasons we did not do so. The first one is that the interface provided by standard packages is cumbersome to code with. This is clearly demonstrated in the adJava interface. For example, since Java does not support templet and operator overloading like C++ does, the runtime system needs all the class names like

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

to express HPJava types

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

Let along there are corresponding ones for `char`, `float`, ...

Further more, for a two dimensional array, to express type like `int[[*,]]`, `int[[,*]]` or `int[[*,*]]`.

```
Array2IntCollapsedDistributed
Array2IntDistributedCollapsed
Array2IntCollapsedCollapsed
```

as subclasses of `Array2Int`[4].

If this is not enough, recall how 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. Since the array elements are actually stored in the `data` field of the representing class, `Array1Int`.

The second reason to develop special syntax is that even though one may want to write very long class names to make the program "easy" to understand, there is no execution performance for a program compiled by Java directly from this kind of coding, as we discussed about the interface. Because all the local address of the global array is expressed by functions such as `element`. An optimization pass is needed to transform the accessing into a more "clever" pattern.

So, if a translator is going to be implemented any way, it makes more sense to design a set of syntax to express the concepts of the programming model more concisely.

[4] which is equivalent to `Array2IntDistributeDistribute`. `Distributed` is used as a place holder here. `Collapsed` is a special case of `Distributed`. It is not difficult to imagine that for a distributed array of more dimensions, class names will be even longer.

3.2 Why not HPF

The design of HPJava language is certainly, not only for its name, influenced by HPF. In fact, the language is the result of our practice to implement an HPF compiling 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 conterpart 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. And both of them can specify how data are mapped on to a processor grid. The difference between the two lies in their communication part. In HPF, a single assign 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, this communication is done though explicit function calls[5],

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

As the experience from compiling HPF language, though there may be different kinds of algorithms to detect communication automatically, it is not clear we can make generated node program have good performance all the time. In HPF, the need to decide on which processor the compuation should be carried on further complicated the situation, even one may apply "owner computs" or "majority computs" rule to partition computation.

In HPJava, the SPMD programming model is emphasized. The global addressed data only provide the convineice that one do not need to do error-prone address translation. While the task of computation partition and communication is still the responsiblity of a programmer. This may looks like an extra onus, and makes the langauge difficult to program than HPF[6]. But it helps programmer to understand the performance of the program much better than

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

[6] The program need to meet SPMD constrains, ie. only the owner of the data can access the data. Runtime checking may be add to ensure this condition, we will disscuss this elsewhere.

HPF, so algorithms exploiting locality and parallelism are encouraged. It also helps to simplify the work of compiler.

In addition, as the communication part is considered the "add-on" parts of the language, HPJava is more smooth than HPF to cooperate with successfully developed SPMD libraries, including MPI, CHAOS, Global Arrays, DAGH, etc.

3.3 Data type in HPJava

In a parallel language, it is certainly 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 one provides flexiblity and is suitable for task parallel programming; the second one provides conveneince, especially for data parallel programming. As summurized in the introduction, this is one goal in our language design.

In HPJava, the variable names are divided into two parts, those originally in Java are used as the first kind of variables. and those declared with `[[ ]]` or `#` are the second kind. The two parts are orthogonaic.

The global variales are supported through *data descriptor*, a concept is emphasized more than ever in the design of HPJava.

The meaning of the data descriptor can be best understood when dealing with procedure calls. Actually, we can consider the simplest form of data descriptor as memory address, as this is the case in C and Fortran77 language, where array is passed by reference. Then the passing of array sections in procedure references represents an important part of the array processing facility in Fortran90[8], which means the data descriptor in Fortran90 can describe the stride information in each array dimension, though the the concept is not explicitly mentioned. One can assume that HPF needs to support a more complex kind of data descriptor to allow passing distributed array. Finally, Java has the most explicit data descriptor concept, its arrays are always considered as *reference*, with one of its field `length` always ready to be accessed publicly.

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

It should be noticed that in HPJava, an array can be finally sectioned to a zero-dimension array, which 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 is why "`[[ ]]` is used for section operation and "`[ ]` is used for array element access. Besides, symble "`#`" is introduced to represent the type signature of a global scalar reference to distinguish an ordinary Java scalar.

Keeping two data sets seems complicated the language and its syntax. But this provides convenience for both task and data parallel processing. Also it actually simplifies interfaces of procedure calls. There is no need for things like `LOCAL` mechenism in HPF to call a local procedure on the node processor. Because the descriptors for ordinary Java variables can be kept the same as before. On each node processor the ordinary Java data will be used as local varables, like an MPI program can have. It even allow the combination of two different arrays. 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 show as in figure 1.

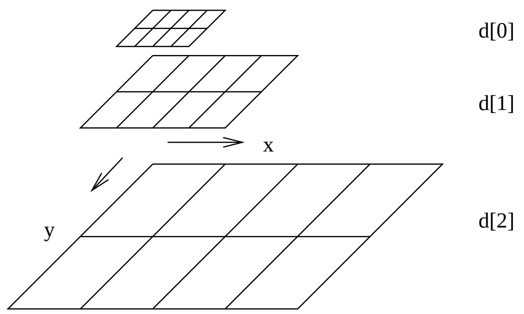

Figure 1: Array of distributed array

One more thing worth to mention is that the HPJava multi-dimension array is Fortran-like, not C-like array of arrays. Given the array captured by a data discriptor should be in continous memory block, this format has more concise information for describing a array distributed on a multi-dimension processor grid.

3.4 Performance enhancement

HPjava is a parallel programming language, performance is one of the most important factors we considered during its design. This can be reflected by the following issues.

Logically a `Location` reference is a map from `Range` by an integer, actually a global address of the array element. So the basic concept `Range` will be enough for any program need. Yet, in the programming model, a built-in class `Location` is still added.

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 also helps program to meet the SPMD constrains in elementatory array access. For example, in HPF, there is nothing wrong to write

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

In HPJava, this is most likely written as,

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

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

Further more, `Location` class has its subclass `Index`, on which the `overall` construct is defined. From languages point of view, this subclass may be not well defined. Actually, it represents an *iterator*, a programming technique rather than a language concept. It may be much clear just overloading **at** constuct and "=" symbol to let,

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

express the `overall` construct. But making a seperate subclass may allow one to define,

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

so as to put an index constructor out side a loop, which a clever compiler only can do after a thorough *flow analysis*. After extensive discussing, we decide to leave the more complicated work in compiler in the future.

Finally, the symbol `*` in global data type signature is also a sign to remind compiler that this is a collapsed range, so only one loop is needed in the generated code is necessary, instead of nested loops for a general case. For more details please refer to [4].

3.5 Convenience for programming

The language provides some special syntax for programmers' convenience in coding. Unlike the syntax for data type declaration, which has profound meanings in the programming model, this part is pure syntactical regulation.

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 maped 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 shortcuts of different construtors for the corresponding classes. Just like Java has special syntax support for String class constructor.

Secondly, the language defined 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 meaning, one can see this kind of conversion only happened when the integer is used to accesss array element, and it should be not used extensively, because it does not help to keep SPMD constrains. Perhaps the really useful thing is its extention, one can use a triplete expression to section a distributed data directly.

a[[$l : u : s$]]

is equavlent to

a[[a.range[0].triplet($l : u : s$)]]

Sometimes, it is also desired to convert an index or location reference to an integer to have a global address,

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

This conversion can only heppens when the location to be converted is guarded by an `at` or `overall` construct which uses the reference itself.

Unlike the conversions defined between Java primary types, both conversions intruduced here can be completed only under certain circumstances. So it is also called *restricted type conversion*. If one feels confusing about this, functions in the interface can always be used for the same purpose, of cause, with more typing work. For example, both conversions introduced above are used in the HPJava code to create array section in Fox algorithm,

```
a[[i, (i+k)%P, :, :]]
```

without the conversions, one needs to write,

```
a[[i, a.range[1].location((i.idx()+k)%P), a.range[2],a.range[3]]]
```

to express the same idea.

Here, `idx` is a method of class `Index`, which return the order of the index reference in the creating range.

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, which will be shown in the examples in the next section. It also is a *restricted* operation, which has its meaning only in certain programm patterns. It should be noticed that, in fact, the restricted type conversion can be used to achieve the same effect. So if shift operation is defined on "+" operator, it should override the type conversion condition.

4 Example programs and performance

In this part, we give more intriguing examples of HPJava code, and their preliminary performance. The HPJava compiler is being developed, which will be introduced elsewhere. So all the performance data comes from hand translated code.

The first example is Choleski decomposition, in which one need to update n times submatrix 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.

We write the above algorithm in the following HPJava program. In the code, a cyclic range is used to allocate the original array on multiprocessors, the result L can overwrite part of its storage. During the computation, collective communication `remap` is used for broadcasting updated columns.

```
Procs1 p = new Procs1(4);
on(p) {
  Range x = new CyclicRange(size, p.dim(0));
  float a[[*,]] = new float [[size, x]];
  // initialize the array here;
  float b[[*]] = new float [[size]]; // as a buffer

  at(j=x[0])
    a[0,j]=Math.sqrt(a[0,j]);

  for(int k=0; k<size-1; k++) {
    for(int s=k+1; s<size; s++)
      at(j=x[k])
        a[s,j]/=a[j,j];

    Adlib.remap(b[[k+1:]],a[[x[k+1:], k]]);

    overall (j=x|k+1:)
      for (int i=x; i<size; i++)
        a[i,j]-=b[i]*b[j];

    at(j=x[k+1])
      a[k+1,j]=Math.sqrt(a[k+1,j]);
```

```
  }
}
```

The second example is Jacobi iteration. The algrithm 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.

```
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];
}
```

In the above code, there is only one iteration. And runtime function `writeHalo` will update the array distributed on different process to make it ready for the iteration.

Figure 2 shows a preliminary benchmark for hand translated codes of our examples. The parallel programs are executed on 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 C++ version of the code. As shown in the figure.

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. Yet the overall performance is not as good, because the JIT compiler on irix. The whole system are also being ported to Windows NT.

5 Concluding remarks

In this report, we introduced the design and implementation issues in HPJava programming, a new programming style we proposed. We can see that the language presented here has the flexibility of SPMD programming, and the convenience of HPF. (Other related languages may include F–, Spar, ZPL and Titanium, which are all different from our approach.) More important, its implementation has a straight forward support from runtime library. In the next step, we will streamline the HPJava translator and identify the optimization techniques required. At the same time, try to incorporate more 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

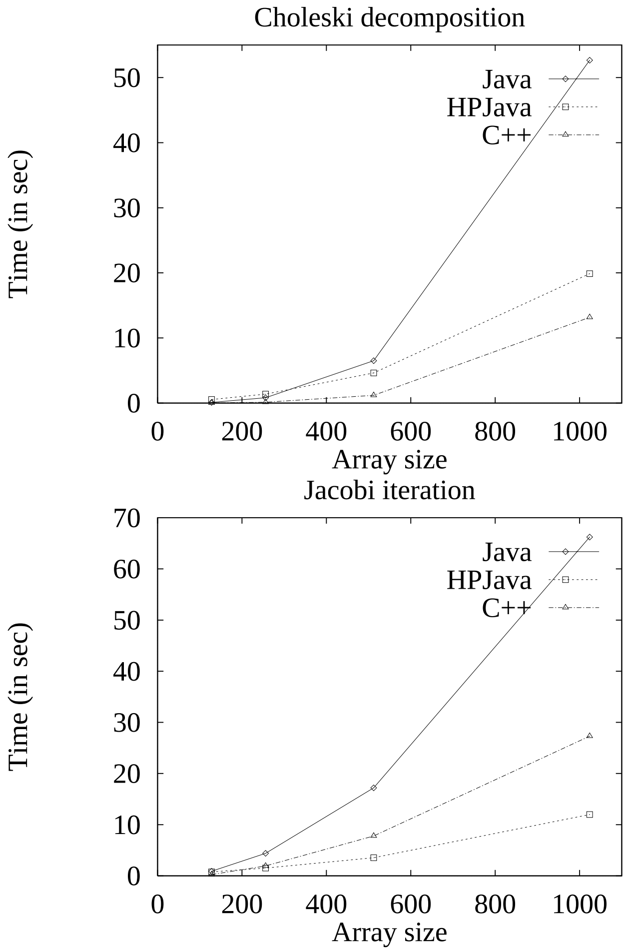

Figure 2: Preliminary performance