HPJava – a Java language binding for *Structured SPMD* programming model

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

May 12, 1998

1 Introduction

In this report, we introduce *HPJava* language, a programming language extended from Java for parallel programming on message passing systems, from multiprocessor systems to workstation clusters.

Although it has a close relationship with HPF[1], the design of HPJava does not inherit HPF. The language introduces a high level structured SPMD programming style, *HPspmd*, which can be summarized as following,

- **Structured SPMD programming** Programs written in the programming language presented here explicitly coorinates a well-organized process group. As in a conventional distributed-memory SPMD program, only a process owning a data item is allowed to access the item directly. The language provides special constructs that allow programmers to meet this constraint conveniently.

- **Global name space** Besides the normal local variables of the sequential base language, the language provides classes of variables accompanied by non-trivial with *data descriptor*, providing a global name space in the form of globally subscripted arrays, with assorted distribute patterns. This helps to relieve programmers of error-prone activities such as the local-to-global, global-to-local address translations which occur in data parallel applications.

- **Hybrid of data and task parallel programming** The language also provides special constructs to facilitate both data parallel and task parallel programming. Through language constructs, different processors can either simultaneously work on global addressed data, or independently execute complex procedures on their own local data. The conversion between these phases is seamless.

- **Communication libraris** In the traditional SPMD model, the language itself does not provide implicit data movement semantics. Different communication patterns are implemented as library functions. This greatly simplifies the task of the compiler, and should encourage programmers to use alogorithms that exploit locality. Data on remote processors are accessed exclusively through explicit library calls. In particular, the initial HPJava implementation relies on a library of powerful collective communication routines. Other distributed-array oriented communication libraries may be bound to the language later. The low level MPI communication is always available as a fall-back. Since the language itself only provides basic concepts to organize data arrays and process groups, it allows the possibility that if a new communication pattern is needed, it can be relatively easy to integrate through new libraries.

The reason we don't follow HPF directly is that during the practice of compiling HPF[2], the idea of "runtime supporting" is emphasized in our approach, and we found the level of runtime system can be effectively raised to a programming language model. In fact, the new programming style introduced is language independent. It also can be binded with other programming languages such as C/C++ and Fortran.

Since Java language is simple, elegant, and more promising, we implemented our prototype based upon this language, and found the efforts are quite rewarding.

2 Java language Binding

String is a class in Java, yet there is language syntax to support it, such as its construction and concatenating operation. In HPJava, we add more this kind of *build-in* classes.

2.1 Basic concepts

The key concepts in the new programming style is built around process groups, which is used to support program execution control in a parallel program.

Process group A *Group* is defined to represents a process group, typically with a grid structure and an associated set of *process dimensions*. It has its subclasses to represent different grid shape, such as `Procs1`, `Procs2`, etc. For example,

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

Naturally, an HPJava program will be executed parallel in each process of a group grid.

Distributed dimension and index with position The elements of an ordinary array can be represented by an array name and an integer sequence. Here, we have two concepts reflected by `int` value, an index to access each array element and a range where the index can be chosen from. When describing a distributed array, we use two new build-in classes in HPJava to represent the above concepts,

- A *range* maps an integer interval into a process dimension according to certain distribution format. Ranges describe the extent and mapping of array dimensions.
- A *location*, or slot, is an abstract element of a range. A range can be regarded as a set of locations, actually it is a one-to-one mapping between the global index and locations. For example,

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

will create two ranges on the different process dimensions of the group `p`, one is block distributed, the other is cyclic distributed. And we can get 100 different items as `Location` references mapped by the range `x` from integers, for example, the first one is[1],

```
Location i = x[0];
```

Subgroup and Subrange A *subgroup* is some slice of a process array, formed by restricting the process coordinates in one or more dimensions to single values.

Suppose `i` is a location in a range distributed over a dimension of group `p`. The expression

```
p / i
```

represents a smaller group—the slice of `p` to which location `i` is mapped.

Similarly, a *subrange* is a section of a range, parameterized by a global index *triplet*. Logically, it represents a subset of the locations of the original range.

The syntax for a subrange expression is

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

The symbol "`:`" is a special separator. It is used to compose a *triplet* expression with optional `int` expressions to represent an integer subset. The default initial and finial value are zero and the extend of the range respectively. The default stride size is 1.

[1] Here "`[ ]`" is used to construct a new location, it is also used to construct a subrange in the following code.

Structured SPMD programming When a process group is defined, a set of `Range` and `Location` reference also have been defined, as in figure 1.

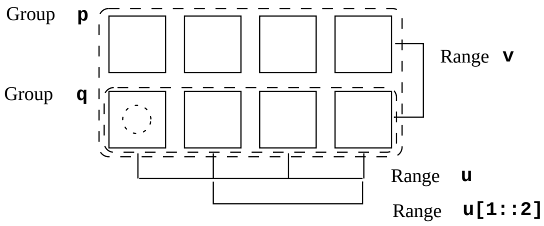

Figure 1: Structured processes group

The two ranges associated with the group `p` are,

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

`dim(int)` is a member function return a range reference, which corresponds to a processor dimension.

Further more, we can get a location in range `v`, and use it to create a new group,

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

As shown in the figure, group `p` is a highly *structured* concept, and all the notions introduced around it contribute to program execution control in the new programming language.

In a traditional SPMD program, execution control is based on `if` statements and process id or rank numbers. In the new programming language, switching execution control is based on the structured process group. For example, it is not difficult to defer, and we will see, that the following code,

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

will switch the execution control inside the bracket to processes in group `p`.

The language also provided well defined constructs to switch execution control among processes according to data items we want to access. This will be introduced later.

2.2 Global variables

As a SPMD programming language, when a program starts on a group of n processes, there will be n logical control threads, which mapping to utmost n physical processors.

On each control thread, the program can define variables in the same way as in a sequential one. The variables created in this way are *local variables*, they are *replicated* names on each process, which will be accessed individually.

Besides local variables, HPJava allows a program to define *global variables*, which are distributed on a process group. A global variable will be considered a single entity during the execution on each process which creates it.

The language has special syntax for the definition of global data. And the global variables are all defined by using the `new` operator from free storage. When a global variable is created, a *data descriptor* is also allocated to describe where the data are created.

Data descriptor and global data The concept of data descriptor is not entirely new. It exists in Java language itself. The field `length` in Java array reflects that Java array is accessed through a data descriptor.

On a single processor, an array variable can be labeled by a simple value like memory addresses and an `int` value as length. On a multi-processor, a more complicated structure is needed to label a distributed array. We also call it *data descriptor*.

The data descriptor portrays where the data is created, and how are they distributed. A logical structure of a descriptor is shown in figure 2.

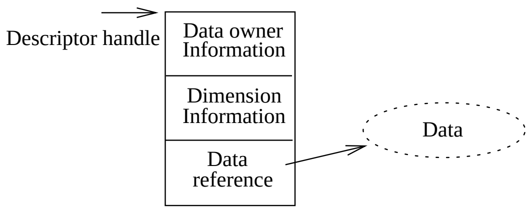

Figure 2: Descriptor

New syntax were added in HPJava to define data with descriptors.

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

creates a global scalar on the current executing process group. In the statement, `s` is a data descriptor handle, in HPJava term, a *global scalar reference*. And the scalar is of an integer value. Global scalar references can be defined for each primitive type and class type in Java.

The symbol `#` in the right hand side of the assignment indicates a data descriptor is allocated as the scalar is created.

For a scalar variable, a field `value` is used to retrieve the value.

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

Figure 3 shows a possible memory mapping for this scalar on different processes.

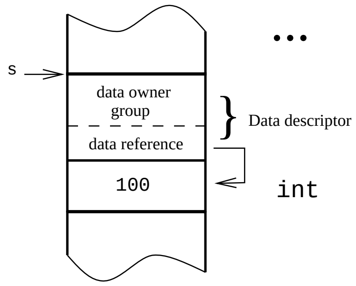

Figure 3: Memory mapping

Note, the value of `s` is *duplicated* on each process in the current executing processes. Duplicated variables are different from replicated local variables. The descriptors they have can be used to keep their value identical on each process during the program execution.

The group inside a descriptor is called *data owner group*, it defines where the global variable belongs.

```
on(p)
  int # s = new int # on q;
```

will set data owner field in the descriptor as group `q`, instead of the default `p`.

Note, the `on` clause only change the data owner, it does not change the *scope* of the variable. The above code is similar but different from the following one.

```
on(p)
  on(q)
    int # s = new int #;
```

When defining a global array, it is not necessary to allocate a data descriptor for each array element, so the syntax to define a global array is not derived directly from the one for a scalar, and it can be defined with different kinds of ranges we introduced before.

Suppose we still have

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

and the process group defined in figure 1, then

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

will create a global array with range y on group q. Here a is a descriptor handle, which describes an one dimension float type array. It is block distributed on group q[2]. In HPJava term, a is also called a *global or distributed array reference.*

The distributed array range can also be a *collapsed* one, where an integer is used.

```
on(p)
  float [[*]] a = new float [[100]];
```

Again, b is a descriptor handle, which describes an one dimension float type array of size 100.

When defining a global array, * is used to mark a collapsed dimension.

The accessing pattern of a global array element is not the same as a global scalar reference, neither exactly the same as a local array element. Since global arrays may have position information in their dimensions, we may need location references as their indexes when their dimensions are not collapsed.

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

Here the forth element of array a is assigned to 3. We will leave at construct and how to access array elements in section 2.3, and look at a simpler example here.

When a global array is defined with a collapsed dimension, accessing its element is as usual,

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

will assign the loop index to each corresponding element in the array.

When defining a multi-dimension global array, one descriptor can describe a rectangular array of any dimensions,

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

will create a two-dimension global array, with the first dimension block distributed and the second cyclic distributed. c is a global array reference, its element can be accessed by putting a single bracket with two location references inside.

The global array introduced here is Fortran-style multi-dimension arrays rather than Java-like array-of-arrays, hence the memory for the multi-dimension array is a continuous block, different from Java's memory scheme for arrays.

[2]The on clause restrict the data owner group of the array to q. If group p is used instead, the one dimenstion array will be duplicated in the first dimenstion of the group, while block distributed on the second dimension.

The array-of-arrays in Java is still useful. For example, one can define a distributed array of local 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 4.

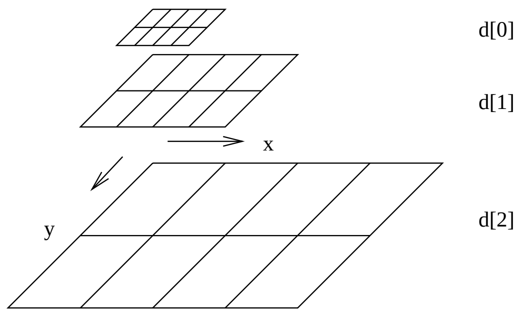

Figure 4: Array of distributed array

Array section and type signature HPJava provides array section of global arrays. The syntax of section subscripting uses double brackets. The subscripts can be locations or ranges[3].

Suppose we still have array `a` and `c` defined as above, then, `b[[i]]`, `c`, `c[[i, y[1::2]]]`, and `c[[i, z]]` are all array sections. Here `i` is a location in the first range of `a` and `c`, and `z` is a subrange of the second range of `c`.

Both expression `c[[i, y[1::2]]]` and `c[[i, z]]` represent a one-dimensional distributed array, providing an alias for a subset of elements of `c`. Expression `a[[i]]` contains a single element of `a`, yet the result is a global scalar reference, not a simple variable.

Array section expression will be used as arguments in function calls[4]. Table 1 shows the type relations of global data with different dimensions.

In the table, both `i` and `j` are location references.

[3]When section is made in a collapsed dimension, an integer or triplet expression may also be used directly.

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

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

Table 1: Section expression and type signature

Inquiry fields and functions The length of an array reference in Java can be get from its `length` field. Similarly, in HPJava, information like data owner group and distributed dimensions can be accessed from the following fields,

```
Group group;   //data owner group
Range range[]; //dimension array
```

And further inquiry functions may be used to get values such as dimension extent and distribution format.

For an array with collapsed dimensions, collapsed ranges will be created for the corresponding dimensions.

2.3 Program execution control

HPJava has all the Java statements for program execution control within a single process. It also introduces three new control constructs, `on`, `at` and `over` for execution control among processes.

A new concept, *active process group*, is introduced. It is the set of processes sharing the current thread of control.

In a traditional SPMD program, the concept of switching the active process group is reflected by `if` statements,

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

means that inside the brace bracket, the processes numbered as 0 to 3 share the control thread.

In HPJava, this concept is expressed by a `Group` reference. When a HPJava program starts, the active process group is a system pre-defined value. During the execution, the active process group can be changed explicitly through an `on` construct in the language.

In a shared memory program, accessing the value of a variable is straight forward. In a message passing system, only the process which holds data can read and write the data. A traditional SPMD program achieves this by using `if` statements. For example,

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

will make sure that only `my_data` on process 1 is assigned to 3.

In the language we present here, the same thing is required, and not only for local variables but also for global variables, i.e. when assigning data, the data owner must be the active process group.

Besides `on` construct, there is a more convenient way to change the active process group according to the array element we want to access, `at` construct.

Suppose we still have array `a` defined in previous section,

```
on (q) {
  Location i=x[1];
  at(i)
    a[i]=3; //correct

  a[i]=3;   //error
}
```

The assign statement guarded by an `at` construct is correct, the one without it may cause run time error if there is run time range checking.

In HPJava, a more powerful construct `overall` can be used to combine the switching of the active process group with a loop,

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

is semantically equivalent to[5]

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

Inside each iteration, the active process group is changed to `q/i`.

In section 3, we will use more programs to show that by using the `at` and `overall` construct it is quite convenient for a program to keep the active process group equal to the data owner group of the assigned data.

When accessing data on another process, HPJava needs explicit communication as in a ordinary SPMD program.

2.4 Communication library functions

Communication libraries are provided as packages in HPJava. Detailed function specifications will be introduced in other papers. Here we will only introduce a small number of top level collective communication functions.

[5]But a compiler can implement `overall` construct in a more efficient way, using linearized address calculation. For detail definition on `overall` construct, please refer to [4]

In the current design, the collective communications are member functions of a static class `Adlib` in a HPJava package. `Adlib.remap` will copy the corresponding element from one to another, regardless of their distribution format. `Adlib.shift` will shift certain amount in a specific dimension of the array in either cyclic or off-edge mode. `Adlib.writeHalo` is used to support ghost region.

Since the basic programming style is SPMD, it is also possible to allow other communication library be integrated as part of the communication packages of the language. We have already implemented Java MPI interface. Currently CHAOS [5] and GA [6] are being considered as "add-on" packages.

3 Programming examples

In this section we only give out example programs to show the new language features.

The first example is Choleski decomposition,

```
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]);

  }
}
```

Here, `remap` is used to broadcast one updated column to each process.

The second example is Jacobi 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];
}
```

In the above code, there is only one iteration, it is used to demonstrate how to define range reference with *halo* area, and how to use the `writeHalo` function.

4 Project in progress

The related projects of the work may include development of MPI, HPF, and other parallel languages such as ZPL and Spar, which are introduced else where[6]. Here we explain more backgrounds and future developments about our own project.

The work originated in our compilation practice in HPF. As introduced in [2], our compiler emphasize runtime system support. *Adlib*[4], as PCRC runtime kernel library, provides a rich set of collective communication functions. During the practice, it is realized that the runtime interface can be effectively raised to a higher level, and a rather straight-forward (compared to HPF) compiler can be developed to translate the high level language code to a node program calling this runtime interface functions.

Currently, Java interface has been implemented on top of the Adlib library. With classes such as `Group`, `Rang` and `Location` in the Java interface, one can write Java programs quite similar to HPJava we proposed here. Yet, the program executed in this way will have large overhead due to function calls (such as address translation) when accessing data inside loop constructs.

Given the knowledge of data distribution plus inquiry functions inside runtime library, one can substitute address translation calls with linear operation on the loop variable, and keep most of the inquiry function calls out side the loop construct. This is the basic idea of the HPJava compiler.

At present time, we are working on the design and implementation of the prototype of this kind of "translator". Further research works may include

[6]For more analysis, please refer to our documents at http://www.npac.syr.edu/projects/pcrc/doc

optimization and safety-checking techniques in the compiler for HPspmd programming.

Figure 5 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, the array size is 1024X1024.

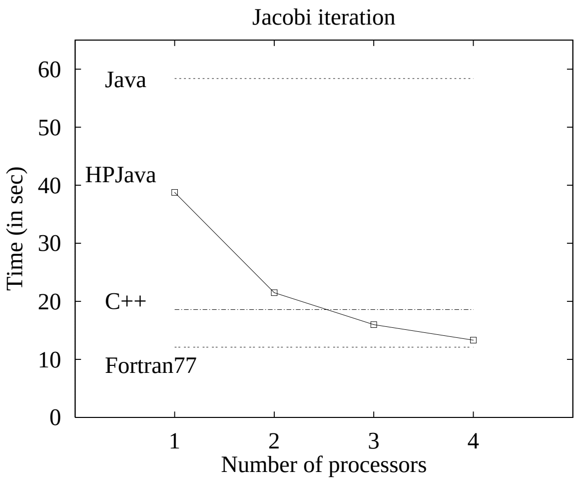

Figure 5: Preliminary performance

We also compared the sequential Java, C++ and Fortran version of the code, all with `-O` flag. Shown in the figure. Since Java program use language own mechanism for calculating array element address, it is slower than HPJava, which uses an optimized scheme.

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 Summary

Through the simple examples in the report, we can see the programming language presented here has the flexibility of a SPMD program, and the convenience of HPF. The language encourages programmers to express parallel algorithms in a more explicit way. We believe it will help programmers to solve real application problems easier compared with using communication packages such as MPI directly, and allow the compiler writer to implement the language compiler without the difficulties met in the HPF compilation.

The Java binding is only an introduction of the new programming style. (A Fortran binding is being developed.) It can be used as a software tool for teaching parallel programming. And as Java for scientific computation become more mature, it will be a practical programming language to solve real application problems in parallel and distribute environments.

References

[1] High Performance Fortran Forum, "High Performance Fortran Language Specification", version 2.0, Oct. 1996

[2] 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.

[3] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xinying Li, and Yuhong Wen. "Introduction to Java-Ad". 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] R. Das, M. Uysal, J.H. Salz, and Y.-S. Hwang. "Communication optimizations for irregular scientific computations on distributed memory architectures". Journal of Parallel and Distributed Computing, Sep. 1994

[6] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. The Global Array: Nonuniform memory access programming model for high-perfomance computers. The Journal of Supercomputing, 1996.