A High Level SPMD Programming Model: HPspmd and its Java Language Binding

Guansong Zhang, Bryan Carpenter, Geoffrey Fox
Xinying Li and Yuhong Wen
111 College Place
NPAC at Syracuse University
Syracuse, NY 13244

Abstract *This report introduces a new language,* HPJava*, for parallel programming on message passing systems. The language provides a high level SPMD programming model. Through examples and performance results, the features of the new programming style, and its implementation, are illustrated.*

Keywords: Java, data parallel programming, SPMD

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 follow HPF directly. Instead it introduces a high level structured SPMD programming style, *HPspmd*, which can be summarized as follows:

- **Structured SPMD programming.** Programs written in the programming language presented here explicitly coordinate 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 *data descriptors*, providing a global name space in the form of globally subscripted arrays, with assorted distributed 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 libraries.** 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 algorithms 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 when a new communication pattern is needed, it should be relatively easy to integrate through new libraries.

In our earlier work on HPF compilation [2] the role of runtime support was emphasized. Difficulties in compiling HPF efficiently suggested to make the runtime communication library directly visible in the programming model. Since Java language is simple, elegant language, we implemented our prototype based upon this language.

2 Java language Binding

`String` is a class in Java, but there is language syntax, including construction and concatenation operations, to support it. In HPJava, we add several similar *built-in* classes.

2.1 Basic concepts

Key concepts in the programming model are built around *process groups*, used to describe program execution control in a parallel program.

Process group. *Group* is a class representing a process group, typically with a grid structure and an associated set of *process dimensions*. It has its subclasses that represent different grid dimensionalities, such as `Procs1`, `Procs2`, etc. For example,

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

An HPJava program will be executed in parallel across the processes of a 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` values: an index to access each array element and a range that index can be chosen from. In describing a distributed array, we use two new built-in classes in HPJava to represent the analogous 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)) ;
```

creates two ranges on the different process dimensions of the group `p`. One is block distributed, the other is cyclic distributed. There are 100 different `Location` items mapped by the range `x` from integers, for example, the first one is

```
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 final values are respectively zero and the extent of the range. The default stride size is 1.

Structured SPMD programming. When a process group is defined, a set of ranges and locations are also implicitly defined, as shown in figure 1. The two (primitive) ranges associated with dimensions of the group `p` are,

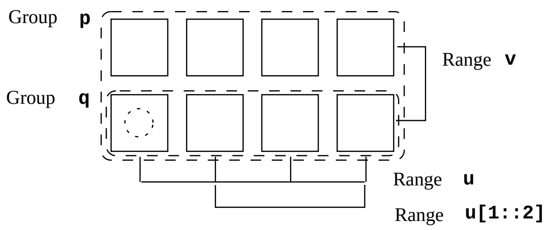

Figure 1: Structured processes group

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

`dim()` is a member function that returns a range reference, directly representing a processor dimension.

We can obtain 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 highly *structured*. 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 guess that the following code:

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

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

The language also provided well-defined constructs to split execution control across processes according to data items we want to access. This will be discussed later.

2.2 Global variables

When an SPMD program starts on a group of n processes, there will be n control threads mapped to n physical processors. In each control thread, the program can define variables in the same way as in a sequential program. The variables created in this way are *local variables*. Their *names* may be replicated across processes, but they will be accessed individually (their scope is local to a process).

Besides local variables, HPJava allows a program to define *global variables*, explicitly mapped to a process group. A global variable will be treated by the process group that creating it as a single entity. The language has special syntax for the definition of global data. 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 held.

Data descriptor and global data. The concept of data descriptor is not new. It exists in the Java language itself. For example, the field `length` in the Java array reflects the fact that an array is accessed through a data descriptor.

On a single processor, an array variable might be parametrized by a simple record containing a memory address and an `int` value for its length. On a multi-processor, a more complicated structure is needed to describe a distributed array. The data descriptor specifies where the data is created, and how they are distributed. The logical structure of a descriptor is shown in figure 2.

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

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

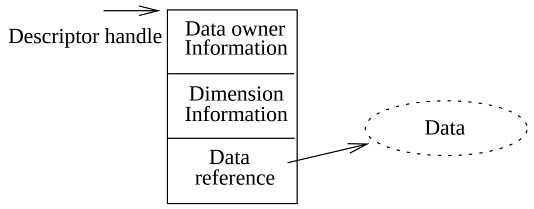

Figure 2: Descriptor

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*. The scalar contains an integer value. Global scalar references can be defined for any primitive type (or, in principle, class type) of 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. Note,

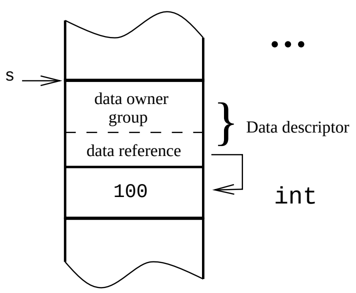

Figure 3: Memory mapping

the value field of `s` is identical in each process in the current executing processes. Replicated *value* variables are different from local variables with replicated *names*. The associated descriptors can be used to ensure the value is maintained identically in each process, throughout program execution.

The group inside a descriptor is called the *data owner group*, it defines where the global values are held.

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

will set data owner field in the descriptor to group `q`. In general this may be a subset of the default, `p` (the whole of the current active process group).

When defining a global array, it is not necessary to allocate a data descriptor for each array element. So the syntax for defining a global array is not derived directly from the one for a scalar. An array can defined with different kinds of ranges introduced earlier. 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 describing a one-dimensional array of `float`. It is block distributed on group `q`[1]. In HPJava term, `a` is also called a *global or distributed array reference*.

A distributed array range can also be *collapsed* (or *sequential*). An integer range is specified, eg

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

When defining an array with collapsed dimensions, `*` can optionally be added in a type signatures to mark these dimensions.

The typical method of accessing global array elements is not exactly the same as for local array elements, or for global scalar references. Since global arrays may have position information in their dimensions, we often use locations as their subscripts:

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

[1]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 replicated in the first dimenstion of the group, and block distributed over the second dimension.

Here the fourth element of array a is assigned the value 3. We will leave discussion of the at construct to section 2.3, and give a simpler example here: if a global array is defined with a collapsed dimension, accessing its elements can be modelled on local arrays. For example:

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

assigns the loop index to each corresponding element in the array.

When defining a multi-dimensional global array, a single descriptor parametrizes 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]];
```

This creates a two-dimension global array with the first dimension block distributed and the second cyclic distributed. Now c is a global array reference. Its elements can be accessed using single brackets with two suitable locations inside.

The global array introduced here is a Fortran-style multi-dimensional array, *not* a Java-like array-of-arrays. Java-style arrays-of-arrays are still useful. For example, one can define a local array of distributed arrays:

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

Array sections and type signatures. HPJava allows to construct *sections* of global arrays. The syntax of section subscripting uses double brackets. The subscripts can be scalar (integers or locations) or triplets.

Suppose we still have array a and c defined as above, then, a[[i]], c, c[[i, 1::2]], and c[[i, :]] are all array sections. Here i is a location in the first range of a and c (it could also be an integer in the appropriate interval). Both the expressions c[[i, 1::2]] and c[[i, :]] represent one-dimensional distributed arrays, providing aliases for subsets of the elements in c. The expression a[[i]] contains a single element of a, but the result is a global scalar reference (unlike the expression a[i] which is a simple variable).

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

Table 1: Section expression and type signature

Array section expressions are often used as arguments in function calls[2]. Table 1 shows the type signatures of global data with different dimensions.

Inquiry fields and functions The size of an array in Java can be had 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
```

Further inquiry functions on Range yield values such as extents and distribution formats.

2.3 Program execution control

HPJava has all the Java statements for execution control within a single process. It introduces three new control constructs, on, at and overall for execution control across processes. A new concept, the *active process group*, is introduced. It is the set of processes sharing the current thread of control.

[2] 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[[,]]. The converse is not true.

In a traditional SPMD program, switching the active process group is effectively implemented by `if` statements such as:

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

Inside the braces, only processes numbered 0 to 3 share the control thread. In HPJava, this effect is expressed using a `Group`. When a HPJava program starts, the active process group has a system-defined value. During the execution, the active process group can be changed explicitly through an `on` construct in the program.

In a shared memory program, accessing the value of a variable is straightforward. In a message passing system, only the process which holds data can read and write the data. We sometimes call this *SPMD constraint*. A traditional SPMD program respects this constraint by using an idiom like

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

The `if` statement makes sure that only `my_data` on process 1 is assigned to.

In the language we present here similar constraints must be respected. Besides `on` construct introduced earlier, there is a convenient way to change the active process group to access a required array element, namely `at` construct. Suppose array `a` is defined as in the previous section, then:

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

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

The assignment statement guarded by an `at` construct is correct; the one without it may cause run-time error.

A more powerful construct called `overall` combines switching of the active process group with a loop:

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

is essentially equivalent to[3]

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

In each iteration, the active process group is changed to `q/i`. In section 3, we will illustrate with further programs how `at` and `overall` constructs conveniently allow one to keep the active process group equal to the data owner group for the assigned data.

2.4 Communication library functions

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

In the current design, the collective communications are static methods of a class `Adlib`. `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 edge-off mode. `Adlib.writeHalo` is used to support ghost regions.

It is possible to integrate other communication library as communication packages of the language. We have already implemented a Java MPI interface. Currently CHAOS [4] and GA [5] 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.

[3] A compiler can implement `overall` construct in a more efficient way, using linearized address calculation. For detail definition on `overall` construct, please refer to [3]

The first example is Choleski decomposition,

```
Procs1 p = new Procs1(4);
on(p) {
  Range x = new CyclicRange(n, p.dim(0));
  float a[[*,]] = new float [[n, x]];
  ... some code to initialise 'a' ...
  float b[[*]] = new float [[n]]; // buffer

  for(int k = 0 ; k < N - 1 ; k++) {
    at(l = x[k]) {
      float d =  Math.sqrt(a[k,l]) ;
      a[k,l] = d ;
      for(int s = k + 1 ; s < N ; s++)
        a[s,l] /= d ;
    }
    Adlib.remap(b[[k+1:]], a[[k+1:,  k]]);

    overall(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]) ;
}
```

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),
    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] = 0.25 * (a[i-1,j] +
          a[i+1,j] + a[i,j-1] + a[i,j+1]);
  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

Projects related to this work include development of MPI, HPF, and other parallel languages such as ZPL[6] and Spar[7]. Here we explain the background and future developments of our own project.

The work originated in our compilation practices for HPF. As described in [2], our compiler emphasize runtime support. *Adlib*[3], a PCRC runtime kernel library, provides a rich set of collective communication functions. It was realized that by raising the runtime interface to the user level, a rather straightforward (compared to HPF) compiler could be developed to translate the high level language code to a node program calling the runtime functions.

Currently, a Java interface has been implemented on top of the Adlib library. With classes such as `Group`, `Range` 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 outside the loop. 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 translator. Further research works will include optimization and safety-checking techniques in the compiler for HPspmd programming.

Figure 4 shows a preliminary benchmark for hand translated versions 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.

We also compared the sequential Java, C++

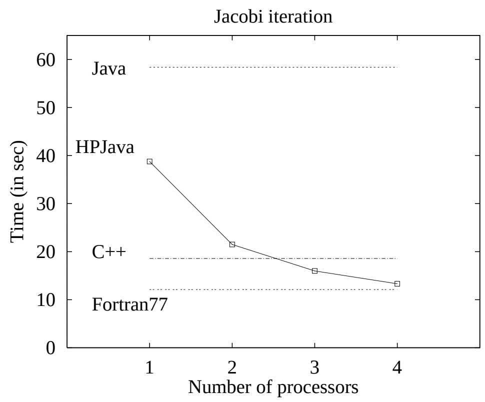

Figure 4: Preliminary performance

and Fortran version of the code, all with `-O` flag when compiling. 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.

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 suggest it will help programmers to solve real application problems more easily, 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 programming style. (A Fortran binding is being developed.) It can be used as a software tool for teaching parallel programming. 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, January 1997. http://www.crpc.rice.edu/HPFF/hpf2.

[2] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. PCRC-based HPF compilation. In *10th International Workshop on Languages and Compilers for Parallel Computing*, 1997. To appear in Lecture Notes in Computer Science.

[3] Bryan Carpenter, Guansong Zhang, and Yuhong Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. Up-to-date version maintained at http://www.npac.syr.edu/projects/pcrc/doc.

[4] R. Das, M. Uysal, J.H. Salz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[5] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. The Global Array: Non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[6] Lawrence Snyder. A ZPL programming guide. Technical report, University of Washington, May 1997. http://www.cs.washington.edu/research/projects/zpl

[7] K. van Reeuwijk, A.J.C. van Gemund, and H.J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, 1997.