Structured SPMD programming
– *Java language binding*

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

NPAC at Syracuse University
Syracuse, NY 13244

{*zgs, dbc, gcf, xli, wen*}*@npac.syr.edu*

Oct. 1, 1997

1 Introduction

In this report, we introduce *HPJava* language, a programming language extended from Java [1] for parallel processing, targeting at multi-processor system with distributed memory.

The language introduces a new programming style, called *structured SPMD programming*(We will explain more about this term in section 2.1.4). It can be summarized as following[1],

- **Structured SPMD programming** The programming language presented here has a single thread control on a well organized process group. As 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, at the same time, presents a well defined loosely synchronization mechanism among the processors.
- **Global name space** Besides local variables, the language provides variables associated with *data descriptor*, a global name space, especially, global addressed array with different distribute patterns among processors in a process group. This helps to relieve programmers of error-prone activities like local-to-global, global-to-local translations in most data parallel applications.
- **Collective communication** Accessing data on different processors is through a powerful collective communication library provided 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.

[1]Though these are "buzzwords" just like Java has, they try to characterize a language independent programming style.

- **Hybrid of data and task parallel programming** The language provides constructs facilitating 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.
- **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.

In fact, the programming style introduced here is language independent, it can also be binded with other programming languages like C/C++ and Fortran.

However, we believe in the near future, Java will play a more important role in the programming world. In addition Java language itself is simple. So we explain the new language constructs inside Java language, and implementing our prototype based on it.

2 Java language Binding

String is a class, yet treated almost as a primary data-type in Java language, we call it as a build-in class. In HPJava, we add more build-in classes to represent basic concepts in our language.

2.1 Basic concepts

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

2.1.1 Process group

A new class, *Group*, is defined, it represents a process group, typically with a grid structure and an associated set of *process dimensions*.

Class `Group` has subclasses derived from it to represent different grid shape, such as `Procs1`, `Procs2`, etc. For example,

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

defines a group `p` of a 2X4 2-dimension process grid. These 8 processes will have 2 *processes dimensions* associate with them.

Naturally, an HPJava program will be executed parallel in each process of a group grid. The package provides a pre-defined group, `Procs1 Adlib.global`. It is an one dimension process grid, with the size of total number of logical processes the runtime environment provided. The program will starts on this group, and new process groups may be created as subsets of this group.

2.1.2 Distributed dimension and index with position

In a programming language, array elements are corresponding to an integer sequence, for example in Java, we have

```
int[] a = new int[10];
```

then, when accessing the array, we use an index in the sequence,

```
for (int i=0; i<a.length; i++)
  ...a[i]...
```

In fact, we have two concepts here, an index `i` and a range where the index can be chosen from, specified by `length`. They are both of type `int`.

When describing an array with position information, we need two new build-in classes in HPJava to represent those two concepts,

Range. A *range* maps an integer interval into a process dimension according to certain distribution format. Ranges describe the extent and mapping of array dimensions.

Location. 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 `Group p`, one is block distributed, the other is cyclic distributed. We will further explain the function `dim` and the meaning of block and cyclic range in section 2.1.4.

We can get 100 different items as `Location` references mapped by the range `x` from integers, for example, the first one is,

```
Location i = x|0;
```

here "`|`" is an operator defined on `Rang` and `int`.

When mapping a location reference from a range by an integer, its *order* or rank in the range is used. We can also get this integer value from an inquiry function in `Range` class.

```
int order = x.id(i);
```

will assign variable `order` to 0.

2.1.3 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. A subgroup is a `Group`.

Any subgroup has a parent process grid and a dimension set which is some subset of the dimensions of the parent grid. The *restriction* operation on a group takes a slice in a particular dimension. This restriction procedure is conveniently expressed in terms of a location reference.

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. (By the nature of the definition of the `/` operator we can see that `p / i / j` is equivalent to `p / j / i`.)

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

The syntax for a subrange expression is

```
x | 1 : 49
```

or

```
x | 1 : 98 : 2
```

for a strided subrange.

In the above examples, both "`/`" and "`|`" are overloaded operators in HPJava.

The symbol "`:`" is a special separator. It is used to compose a *triplet* expression, with three optional `int` expression to represent an integer subset by pointing out the initial position, final position and an optional stride size. Expression `:98:2`, `1:98` and `1::2` are all legal triplet. The default initial and finial value are zero and the maximum value of the array index respectively. The default stride size is 1. Unlike other expressions in Java, there is no type information associate with them.

2.1.4 Structured SPMD programming

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

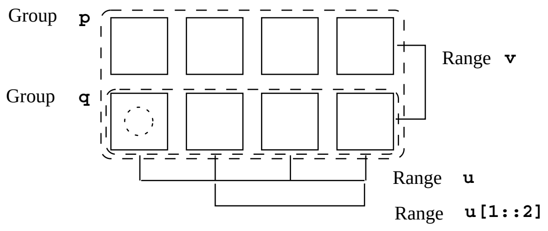

Figure 1: Structured processes

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

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

`Group.dim(int)` is a member function return a `Range` reference. It corresponds to a processor dimension, which reflects the basic meaning of a range. This range can be considered either a block range with its size as one, or a cyclic range of its extent the same as the processor number. More complicated ranges can be derived from it. Currently, the language supports *block range*, *cyclic range*. The distribution format for these two ranges can be show in figure 2. Besides a *collapsed range*, defined as duplicated positions on each process in the dimension is also provided.

Combining block range and cyclic range will get block-cyclic distribution. For example,

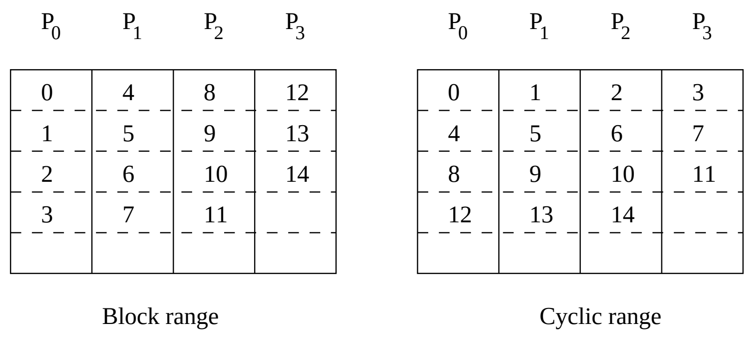

Figure 2: Range distribution pattern

```
Range v1=new CyclicRange(20, v);
Range v2=new BlockRange(100, v1);
```

will create a cyclic(5) distribution on the second dimension of the processor.

We define the range representing processor dimension as *level 0* range, and range based on it one *level* higher. So `u` and `v` are level 0 ranges, `v1` is a level 1 range and `v2` is a level 2 range[2].

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

```
Location i = u|1;
Group q = p/i;
```

We also can create a subrange of `v` by writing

```
v|1::2
```

Note that in `Group p`, it need two locations to locate the processor with a dotted circle. They are `i` mapped from `Range u`, and a new location `j` mapped from `Range v`.

```
Location j = v|0;
```

From the figure, we can see that `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 we introduced above, for example, it is not difficult to think that the following code

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

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

Further more, the language provided well defined constructs to switch execution control among processes according to data items we want to access. Therefor, we call the programming model introduced here *structured SPMD programming*.

[2]Currently, the language supports up to level 2 range. We don't see the need to support level 3 range yet.

2.2 Global variables

HPJava is a SPMD programming style 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 Java language. The variables created in this way are *local variables*, they are *replicated* names on each process, which will be accessed by each process individually.

Besides local variables, HPJava allows a program to define *global variables*, which are distributed on a process group. The global variables 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 obtained by using the `new` operator from free storage.

When a global variable is created, a *data descriptor* is also allocated to describe where it is created. We will further introduce this concept.

2.2.1 Global scalar and data owner

The following code,

```
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. Also it will be used to access this `int` value, as in the following, [3]

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

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

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.

Compared with a local variable,

```
on(p)
  int t = 10;
```

the memory layout for the two will be quite different. There is no data descriptor allocated with `t`, neither `t` is a handle.

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

For a local variable, the data owner is the process which holds the data, though there is no descriptor to record it. For example, there will be an integer `t` on each process in `Group p` when `t` is defined as above.

[3]Though the "`#`" is used as a prefix operator here, `s` is only a handle, it is not the descriptor itself.

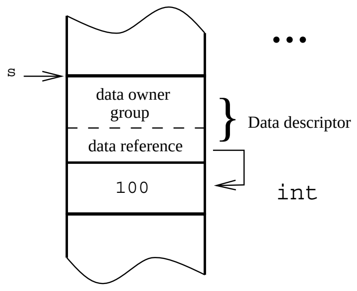

Figure 3: Memory mapping

For a global variable, the data owner group and the group on which the data are defined are two different concepts. By default, the data owner is the current executing group. We can also specify the data owner group explicitly by using an optional `on` clause after `new` operator.

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

will set data owner field in the descriptor of `s` as `Group q`.

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

In the previous fragment, the owner of `s` is set as `Group q`, but its descriptor is duplicated on each process in `Group p` where `s` is defined.

2.2.2 Global array

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 scalar.

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

will create a global array of size 100 on `Group p`. Here `a` is a descriptor handle, which describes an one dimension `float` type array. The distribution format is the same as a collapsed range. We call it *collapsed dimension*. In HPJava term, `a` is also called a *global or distributed array reference*.

A global array can also be created with different kinds of ranges we introduced before. Unlike the global array defined with collapsed dimension, array element in a range defined dimension is not duplicated in that process dimension.

Suppose we still have

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

and the process group defined in figure 1, then

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

will create a global array with `Range x` on `Group q`. Here `a` is a descriptor handle, which describes an one dimension `float` type array of size 100, distributed with block range on the first dimension of `Group p`.

Note, when defining a global array, if its dimension is not collapsed, then a symbol `#` is marked in double brackets.

The accessing pattern of a global array element is not the same as a global scalar reference, neither exactly 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)
  b[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 something simpler here.

When a global array is defined with a collapsed dimension, accessing its element is simpler.

```
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, there is also no need to allocate a descriptor on each dimension. One descriptor can describe a rectangular array of any dimensions[4].

```
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 array introduced here is Fortran-style multi-dimension arrays rather than C-like array-of-arrays, hence it can be clearly showed that which dimensions the descriptor is describing.

The array-of-arrays in Java is still useful, global arrays can be combined with 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]]];
}
```

[4]limitations may come from the size of one dimension array in Java. Since it's used to allocate memory for a multi-dimension array.

will create an array show as in figure 4.

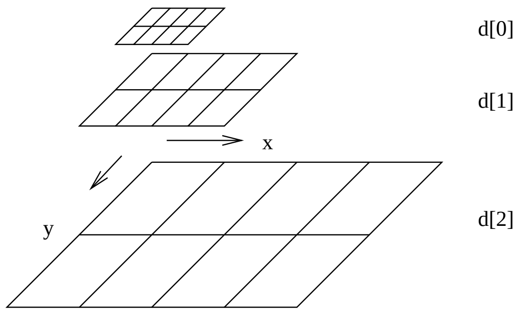

Figure 4: Array of distributed array

2.2.3 Array section and type signature

HPJava provides array section of global arrays. The syntax of section subscripting is similar to array definition, a double bracket is used. The subscripts can be locations or ranges.

Suppose we still have

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

then, `c`, `c[[i, y|1::2]]`, `c[[i, z]]` and `b[[i]]` are all array sections. Here `i` is a location in the first range of `b` and `c`, and `z` is a subrange of the second range of `c`.

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 `b[[i]]` contains a single element of `b`, yet the result is a global scalar reference, not a simple variable.

When section is made in a collapsed dimension, a triplet expression instead of a range reference is used directly. For example, if we have,

```
float [[ ]] a = new float [[100]];
```

then `a[[:98:2]]` is a strided array section.

In HPJava, array section expression will be used as arguments in member function calls. As in any programming language, only the dummy argument and actual argument have the same type signature, the function call can be completed. Table 1 shows the type relations of global data with different dimensions.

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

When used in method calls, `#` marked type is a *super-type* of the one without the symbol. 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 variable	**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

2.2.4 Data descriptor

When we introduced global data, we introduced a concept, *data descriptor*. We will explain more about the idea in this section.

Actually, the concept of data descriptor is not something entirely new. It exists in Java language itself.

In Java, array type is not of any class type, but a special reference type. In fact, it is exactly a handle to a data descriptor. We can see this point through the following example.

In C language, a `main` function may has the following prototype,

```
int main(int argc, char* argv[]);
```

From the function input, one can retrieve the information like how many command line arguments are passed, and where they are located.

But in Java, a `main` function is written as,

```
int main(String argv[]) {
  ...
};
```

Why does not Java language need to pass the number of arguments?

The answer is that you can always get this value from the array reference, `argv.length`. This means `argv` not only store where the array data is, but also record the length of the array. It is a simplest 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, to label a global variable, a more complicated structure is needed. We call it *data descriptor*. It portrays where the data is created, and how are they distributed. A logical structure of a descriptor is shown in figure 5.

The descriptor locates on each process which defines the global variable, whenever necessary, a process can look up those information to decide where the data is.

In HPJava, when global data are defined, such as

```
int # s = new int #;
float [[ ]] a = new float [[ ]];
float [[#]] b = new float [[x]];
float [[#,#]] c = new float [[x, y]];
```

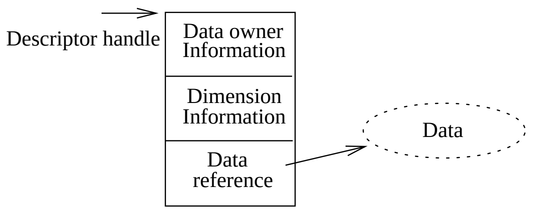

Figure 5: Descriptor

a descriptor shown in the figure is allocated with them.

Though they have different syntax looking and different names, such as global scalar reference an global array reference, in an implementation point of view, `s`, `a` `b` and `c` are all descriptor handles. They are not the descriptor themselves.

Therefor, syntactically , we use the descriptor handle to represent the whole data it describes; And we use different operators, such as `#` (when it describes scalar) or `[]` (when it describes array) to access the data value.

When a new array section is created, a new descriptor has to be created to depict a subset of the same array with some dimension restricted by locations or subranges. In this case, a double bracket is used.

2.2.5 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 on a single process. It also introduces three new control constructs, `on`, `at` and `over` for execution control among processes.

2.3.1 Active process group

A new concept, *active process group*, is introduced here. 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` statement,

```
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, `Adlib.global`. During the execution, the active process group can be set explicitly through an `on` construct in the language, for example

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

This will change the active process group of the code inside the bracket to `Group p`, which means that every statement inside the bracket is only executed by the processes inside `Group p`.

2.3.2 Accessing global variable

In a single process program, accessing the value of a variable is straight forward. Given a data reference, there is only one process can be used to read or write it. In a SPMD program, 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. Say it in HPJava terms: when assigning data, the data owner must be the active process group.

We have already introduced the `on` construct to switch the active process group, 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 `b` defined in previous section,

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

  b[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 general, `at` construct has the form,

```
at(Location i= ...) {
  array[i]=...
}
```

inside the brace bracket, the active process group will be restricted by `Location i`. If `i` has already been declared, `Location` can be omitted in the construct.

This is similar to a traditional SPMD program like,

```
if( own(array[global_local(i)]) ) {
  array[global_local(i)]=...
}
```

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

```
on(q)
  over(Location i= x|0:3)
    b[i]=3;
```

is semantically equivalent to[5]

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

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

In general, `over` construct has the form,

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

Note, the `=` inside `over` construct header does not represent an assignment expression. It is used to compose an `over` loop *control header*, with its left hand side as a loop index and its right hand side as loop range in which the loop index takes value.

Similar as in `at` construct, if the loop index has already been declared in the program, keyword `Location` can be omitted.

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

Generally speaking, when writing global data, the active process group should be equal to the data owner group. Bigger than the data owner group may cause runtime problem, smaller than the data owner group may cause the global variable have different values on different processes. When reading global data, the active process group should be not bigger than the data owner group. In any situations, the active process group must be contained in the data owner group. We call this *access rule* for global data. Some optional run-time checking may be added at compiling time to find code violating this rule, but it also should be kept in mind of the programmer to avoid the accessibility violation.

When accessing data on another process, HPJava needs explicit communication as in a ordinary SPMD program. The most important communication functions are introduced in section 2.5.

[5]But a compiler can implement `over` construct in a more efficient way.

2.4 Operators

Table 2 summarize the operators which is new to or overloaded from Java, and their functionalities in HPJava.

Operator	**Operand type(s)**	**Operation performed**
`/`	`Group`, `Location`	restrict a group to a subgroup by a location, used as the divide operator in Java
`|`	`Range`, triplet expression	restrict a range to a location by an integer expression or a subrange by a triplet expression; also in `over` construct control header. used as the bit-or operator in Java
`:`	`int`, `int`	form a triplet expression
`#`	global scalar reference	access values of the global scalar reference, also in definition of a global scalar reference, and a non collapsed dimension
`[[]]`	global array reference, `Location`, `Range` or triplet expression	section subscript a global array, also in definition of a global array reference
`[]`	global array reference, `Location`	access array element
`=`		in `over` construct control header. used as an assignment expression in Java

Table 2: Operator

The first three items are used upon built-in classes in HPJava. Though we list symbol `:` here, it is different from other operators. As we introduced earlier, when it is used on integer, no value in Java type is returned, only a triplet expression is formed. The next four items are concerned with defining and accessing global data. We consider `[]` as an operator on array reference here.

There are two things we need to pay attention to in this section.

Operator `new` When `new` is used to allocate global variables in the program, the new data are created on the current active process group. In HPJava, there is an optional `on` clause, which can be used after `new` operator to specify the data owner group. Here again, it should be noticeed that the active process group as an executor of `new` is different from the data owner group, the one specified by the `on` clause.

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

will create a global scalar similar but different to the one created in the following,

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

In the first fragment, the data descriptor is duplicated on all process in `Group p`, but for those not in `Group q`, the data reference field of the descriptor is undefined.

We also have accessibility problem here. One can not define,

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

Restricted type conversion When an array has position information in its dimensions, `Location` references, instead of `int` values are used as indexes. Given a global index we can get a location reference by `|` operation from the range. Sometimes it is also convenient that given a location reference, we can get the order of the location in the range it comes from. So a type conversion is defined from `Location` to `int` inside `at` and `over` constructs.

```
on(q)
  over(Location i= x|0:3) {
    b[i]=(int)i;
  }
```

will assign the location id to each array element. It is equivalent to,

```
on(q)
  over(Location i= x|0:3) {
    b[i]=x.id(i);
  }
```

Whenever necessary, this conversion can be an implicit one.

We call it as a *restricted conversion*, because it works only with `at` and `over` constructs. For example,

```
on(q)
  over(Location i= x|0:3) {
    Location j=i;
    b[i]=j;          //error, j can not be converted;

    Range y=x;
    Location k=y|i;  //correct, i converted to an integer;
    b[i]=k;          //error, k can not be converted;
  }
```

Another restricted type conversion is vice versa, from `int` to `Location`. This only happens when an integer used as an index in a dimension which is not a collapsed dimension, and it will be converted to the location mapping from the range which define the dimension of the array.

For example, suppose `b` is a distributed array,

```
int i=3;
b[(Location)i]=i;
```

is equivalent to,

```
int i=3;
b[b.range[0]|i]=i;
```

Similarly, a triplet expression can be converted to a subrange. Both conversions are valid only when their results expected to be a location or subrange. And they also can be implicit ones.

Adding integers and triplets as *access keys* for distributed array provides a "higher level" access for programmers. But they are not intend to be abused. A programmer should consider their code to exploit the possibility to reuse concepts such as location and subrange to improve efficiency, instead of depending compiler to find reusable patterns. One more thing need to be noticed is that though it is actually correct to use both location reference and integer to access array element in a non-collapsed dimension, we still use symbol `#` to mark non-collapsed dimension when defining a distributed array. The reason to do this is to help compiler generate better code for address caculation when an array dimension is collapsed.

These type conversions are defined to simplify program writing. If they look confusing, one can always use inquiry functions to get the desired results.

One typical use of the type conversion is location shift. It supports *ghost regions*, which require a new constructor for `BlockRange` with specification of ghost widths. A collective operation `Adlib.writeHalo` can update the ghost regions of a distribute array, copying values from the physical segments of neighboring processes. For example, the following is a version of Jacobi iteration with only one iteration inside.

```
Procs2 p(2, 4);

on(p) {
  Range x = new BlockRange(100, p.dim(0), 1);  // ghost width 1
  Range y = new BlockRange(200, p.dim(1), 1);  // ghost width 1

  int [[#,#]] a = new int [[x,y]] ;

  // ... some code to initialize 'a'

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

  Adlib.writeHalo(a);

  over(Location i=x|:)
    over(Location j=y|:)
      b[i,j] = (a[i-1,j] + a[i+1,j] + a[i,j-1] + a[i,j+1]) / 4;
      // location i, j converted to integer + or - one, converted back;

  over(Location i=x|:)
    over(Location j=y|:)
      a[i,j] = b[i,j];

}
```

2.5 Communication library functions

Communication libraries are provided as packages in HPJava. Detailed function specifications will be introduced in other papers. In this section, we will only introduce a small number of top level collective communication functions, through which data parallel applications may have dead-lock free communication.

It should be noticed that the SPMD programming style of the language also allow MPI library be integrated as part of the communication libraries of the language. This part will also be introduced else where.

2.5.1 Package `adlib`

As a Java language, HPJava has a special package `adlib`. Several important collective communications are member functions of a static class `Adlib` in the package.

`Adlib.remap` Given two global array of the same shape and element type, `Adlib.remap` will copy the corresponding element from one to another, regardless of their distribution format.

`Adlib.shift` This function will shift certain amount in a specific dimension of the array in either cyclic or off-edge mode. Explicit memory management for the shift is needed.

`Adlib.writeHalo` This function is used to support ghost region, it update the halo area of the global array from neighboring processes when is called.

3 Programming examples in HPJava

We already saw one example program, Jacobi iteration, in section 2.4. In this section, we have two more examples.

3.1 Choleski decomposition

In [2] different kinds of Choleski algorithm organization are discussed. Among them, column interleaved outer product form and row interleaved column Choleski form are considered more promising ones on a parallel computer system with distributed memory. Here we only write one of them, the outer product form of Choleski algorithm, which is based on the following mathematic equation:

$$A_1 = \begin{bmatrix} a_{11} & \mathbf{a_1}^T \\ \mathbf{a_1} & A_s \end{bmatrix} = \begin{bmatrix} a_{11}^{1/2} & 0 \\ \mathbf{l_1} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & A_2 \end{bmatrix} \begin{bmatrix} a_{11}^{1/2} & \mathbf{l_1}^T \\ 0 & I \end{bmatrix}$$

the updating of the submatrix A_2 is accomplished by subtracting the outer product of $\mathbf{l_1}\mathbf{l_1}^T$. A simple description of the algorithm in a sequential code may be,

```
$l_{11} = a_{11}^{1/2}$
$For\ k = 1\ to\ n - 1$
    $For\ s = k + 1\ to\ n$
        $l_{sk} = a_{sk}/l_{kk}$
    $For\ j = k + 1\ to\ n$
        $For\ i = j\ to\ n$
            $a_{ij} = a_{ij} - l_{ik}l_{jk}$
    $l_{k+1,k+1} = a_{k+1,k+1}^{1/2}$
```

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.

```
on(Adlib.global) {
  Range x = new CyclicRange(size, Adlib.global.dim(0));

  float a[[,#]] = new float [[size, x]];
  // initialize the array here;

  float b[[]] = new float [[size]]; // used as a buffer

  Location j;

  at(j=x|0)
    a[0,j]=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[[k,x|k+1:]]);

    over (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]=sqrt(a[k+1,j]);
  }
}
```

3.2 Fox's algorithm for matrix multiplication

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. That is,

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + ... + a_{i,n-1}b_{n-1,j}$$

Fox's algorithm [3] for multiplication organize A, B and C into submatrix on a P by P process array. It take P steps, in each step, broadcast corresponding submatrix of A on each row of the processes, do local computation and then shift array B for the next step computation. Write this algorithm in HPJava, we still use `Adlib.remap` to broadcast submatrix, `matmul` is a subroutine for local matrix multiplication. `Adlib.shift` is used to shift array B, and `Adlib.copy` copies data back after shift, it can also be implemented as nested `over` and `for` loops.

```
Group 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 [[#,#, , ]] temp = new float [[x,y,B,B]];

  for (int k=0; k<p; k++) {
    over(Location i=x|:) {
      float [[,]] sub = new float [[B,B]];

      //Broadcast submatrix in 'a' ...
      Adlib.remap(sub, a[[i, (i+k)%P, z, z]]);

      over(Location j=y|:) {
        //Local matrix multiplication ...
        matmul(c[[i, j, z, z]], sub, b[[i, j, z, z]]);
      }
    }

    //Cyclic shift 'b' in 'y' dimension ...
    Adlib.shift(tmp, b, 1, 0, 0); // dst, src, shift, dim, mode;
    Adlib.copy(b, tmp);
  }
}
```

4 Summary

Through the three simple examples in the report, we can see the programming language we present here has all the power of a SPMD program with MPI can have, yet, it has the convenience of programming in 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 here 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 when Java plays a more significant role in scientific and engineering computation, it can be an intermediate language to implement high level programming language such as HPF, or as a practical programming language itself to solve real application problems in parallel and distribute environments.

References

[1] Ken Arnold, James Gosling, "The Java programming language", Addison-Wesley Publishing Company, Inc., 1996.

[2] J. Ortega, "Introduction to parallel and vector solution of linear system", Plenum Press, 1989.

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