HPJava: proposals for comment

Bryan Carpenter, Guansong Zhang

September 22, 1997

1 Overview

The HPJava programming model is a flexible hybrid of the data parallel and SPMD approaches. It provides HPF-like distributed arrays as language primitives, and new *distributed control* constructs to facilitate access to the local elements of these arrays[1]. In the SPMD mold, the model allows processors the freedom to independently execute complex procedures on local elements: it is not constricted by SIMD-style array syntax.

In HPJava, all access to *non-local* array elements must go through library functions—typically collective communication operations. This puts an extra onus on the programmer; but making communication explicit encourages the programmer to write algorithms that exploit locality, and simplifies the task of the compiler writer. On the other hand, by providing distributed arrays as language primitives we are able to relieve the programmer of error-prone tasks such as converting between local and global array subscripts and determining which processor holds a particular element. In the HPJava model the programmer never has to deal explicitly with local array subscripts. As in HPF, programs can be written at a natural level of abstraction where the meaning is insensitive to the detailed mapping of elements.

To describe the mapping of data and computation the model introduces three special classes:

Group. A *group* is a process group, typically with some grid structure and an associated set of *process dimensions*.

Range. A *range* maps an integer interval into a process dimension according to some HPF-like 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, or as a mapping from a global subscript to locations.

[1] HPF 2 defines an `ON` construct with a similar role.

These three classes are used in the construction of distributed arrays, and parametrize the three distributed control constructs, `on`, `at` and `over`:

```
Group p ;                Location i ;           Range x ;
...                      ...                    ...
on(p) {                  at(i) {                over(x) {
  ...                      ...                    ...
}                        }                      }
```

`on` and `at` restrict execution of the code in their bodies to particular subsets of processes. `over` is a distributed parallel loop.

2 Distributed Arrays

HPJava provides multi-dimensional distributed arrays. These are additional to—orthogonal to—the ordinary sequential arrays of Java.

The mapping of a distributed array is described through two classes with a special status in the language: `Group` and `Range`. A `Group` object describes the process group over which array elements are distributed. A `Range` describes the mapping of an array dimension to a process dimension. These basic classes have various subclasses some of which are illustrated below.

```
Procs2 p = new Procs2(3, 2) ;

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

float [[,]] a = new float [[x, y]] on p ;
int   [[]]  b = new int   [[x]] on p ;
```

`Procs2` is a subclass of `Group` describing a two-dimensional process grid. Declaration of `p` creates a new partition of six of the active processes into a three by two grid.

`BlockRange` is a subclass of `Range` describing an index space distributed block-wise over a process dimension. In the example `x` is a range of extent 100 distributed over the first dimension of `p`; `y` is a range of extent 200 distributed over the second dimension of `p`. *Global subscripts* associated with a range of extent N are always in the range $0, \ldots, N-1$.

Finally, `a` and `b` are distributed arrays. A distributed array is distinguished from a normal Java array by use of double brackets in the type signature (and constructor). `a` is a 200 by 100, two-dimensional array with elements of type `float`, distributed blockwise over `p`. Its dimension ranges are `x` and `y`. `b` is a 100-element, one-dimensional array. Its dimension range is `x`. Implicitly, it is replicated over the second dimension of `p`, because it has no range distributed over this dimension.

In general, the ranges of an array must be distributed over orthogonal dimensions of the process group over which the array as a whole is distributed. That group is specified in the `on` clause of the array constructor[2].

The declarations above are equivalent to the following declarations in HPF:

```
!HPF$ PROCESSORS p(3, 2) ;

REAL a(100, 200)
INTEGER b(100)

!HPF$ DISTRIBUTE a(BLOCK, BLOCK) ONTO p
!HPF$ ALIGN b(i) WITH a(i, *)
```

Syntax aside, the only substantive difference is that in HPJava it is individual *ranges* that are distributed, not arrays as a whole. HPF-style alignment relations between arrays then emerge automatically by allowing arrays to share ranges. The elevation of ranges to first-class language entities is important to the distributed control constructs described below, especially for the distributed loop.

Though rarely useful in their own right, for completeness HPJava allows *distributed scalars* to be declared as follows

```
int # c = new int # on p ;
```

This creates a scalar value replicated over `p`. A distributed scalar differs from an ordinary Java variable—it is embedded in a specific group, generally a subset of the active processes. Formally, it is equivalent to a zero dimensional distributed array.

3 Access to elements of distributed arrays

A range can be regarded as a set of *locations* or slots. This model of a range is visualized in figure 1. An individual slot is described by the third special class of HPJava, `Location`. Each `Location` element is mapped to a particular slice of a process grid. An array has an element, or slice of elements, for each location in each of its ranges. We emphasize that in general two locations are identical only if they come from the same position in the same range.

HPJava has a special syntax for the location `n` in range `x`—an overloading of the `|` operator:

```
Location i = x | n
```

This is one of the basic ideas of HPJava. By working in terms of abstract locations—elements of distributed ranges—we can respect locality of reference without resorting to low-level local subscripts and process ids.

[2]The `on` clause can be omitted, in which case the array is distributed over the *active process group*. This will be explained later.

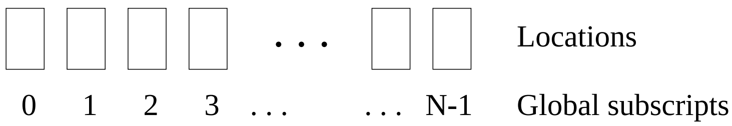

Figure 1: A range regarded as a set of locations, or slots.

Local subscripting of arrays[3] uses double brackets containg a list of *locations*. For example

```
a [[i, j]]

b [[i]]
```

The locations used as subscripts must be elements of the corresponding ranges of the array. These expressions are only meaningful on the processors that hold the elements concerned. The simplest way to make sure that statements using array elements are only executed on processors that hold copies of them is through the *at* construct. This is our first *distributed control* construct. The code fragment below illustrates a pair of nested *at* constructs.

```
Location i = x | n, j = y | 17 ;
at(i)
  at(j)
    a [[i, j]] = b [[i]] + 23 ;
```

The *at* construct is the HPJava abstraction of the *if* statement in a low-level SPMD program that tests whether the local processor contains a desired array element.

A special shorthand is provided to avoid explicit declaration of `Location` objects in this context. The fragment above can be replaced with

```
at(x | n)
  at(y | 17)
    a [[x, y]] = b [[x]] + 23 ;
```

In this notation, `x` and `y` are called the *parametric ranges* of the constructs. Within the body of an *at* construct that has `x` as the parametric range, the symbol `x` can be used to stand for the `Location` object defined in the argument of the construct.

Of course, compared with the HPF equivalent:

```
a (n, 17) = b (n) + 23 ;
```

our syntax looks awkward. But, in a stylized way, the *at* construct reflects what actually happens in an SPMD program. As a matter of observation, good parallel algorithms don't usually expend many lines of code assigning to isolated

[3] Distinguished from *section subscripting*, which will be introduced later.

elements of distributed arrays. Sequential access to elements of parallel arrays is not a particularly good idea. For this reason we leave the *at* mechanism as it stands, and turn to the more important idiom for dealing with *parallel* access to parallel arrays.

The *over* construct is a distributed loop. It is parametrized by a range object. For example

```
over(x)
  over(y)
    a [[x, y]] = 2 * b [[x]] + y ;
```

The body of an *over* construct executes, conceptually in parallel, for every element of the parametric range. In the shorthand form of the *at* construct, the parametric range could stand for a `Location` object inside the body of the construct. The same idea is at work here. In each particular parallel iteration the parametric range can be used as an array subscript, in which case it stands for the location of the range associated with this iteration. The example illustrates a second analogous conversion in the use of `y` on the right hand side: a range can be converted to a numeric value representing the current value of the global subscript. The result of this conversion is only well defined inside an *at* or *over* construct parametrized by the range concerned.

Invariably, if a range is used to parametrize an *at* or *over*, the only meaningful use of that range inside the body of the construct is through one of these conversions to a location or global subscript.

4 The active process group

The active process group is the set of processes sharing the current thread of control. This may, for example, be the whole of a particular process grid. The active process group can be set explicitly through the *on* construct, for example

```
on(p)
  # c = 34 ;
```

In the body of the construct the active process group is changed to the group `p`. (Recall that `c` was declared earlier as a scalar distributed over `p`. The analogue of local subscripting for a distributed scalar is application of the `#` operator. Formally `# c` is equivalent to `c [[]]`.) The *on* construct is our abstraction of the *if* statement in a low-level SPMD program that tests whether the local process is a member of a particular group.

The *on* construct can only *reduce* the active process group: in the example, `p` must be contained in the group active outside the construct. Similarly the *over* and *at* constructs defined in the previous section can only appear in a context where their parametric range is distributed over a dimension of the active process group. In fact the active process group is further modified inside the body of these constructs. It is reduced to some slice of the process grid. We will return to this point in appendix A.

5 Collective communication

Collective communications are movements of distributed arrays. They are static members of the `Adlib` class. The following fragment illustrates the use of one such member: `shift`.

```
Procs2 p(2, 2) ;

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

  int [[,]] u = new int [[x, y]] ;

  // ... some code to initialise 'u'

  int [[,]] unx = new int [[x, y]], upx = new int [[x, y]],
            uny = new int [[x, y]], upy = new int [[x, y]] ;

  Adlib.shift(unx, u,  1, 0, CYCL) ;
  Adlib.shift(upx, u, -1, 0, CYCL) ;
  Adlib.shift(uny, u,  1, 1, CYCL) ;
  Adlib.shift(upy, u, -1, 1, CYCL) ;

  over(x)
    over(y)
      u [[x, y]] = 0.25 * (unx [[x, y]] + upx [[x, y]] +
                           uny [[x, y]] + upy [[x, y]]) ;
}
```

The array arguments of `shift` must have the same type and have the same ranges (they must be aligned). The third argument is the shift amount, the fourth is a dimension selector, and the fifth is the mode of the shift—in this case cyclic shift is selected.

A more efficient version of this code, using ghost regions, will be given later.

The `Adlib` class will provide an interface to all the collective communication functions of the NPAC PCRC library.

6 Subranges

A *subrange* is a section of a range, parametrized by a global subscript triplet, in the Fortran style. A subrange is a `Range`. Logically, it is a subset of the locations of the original range.

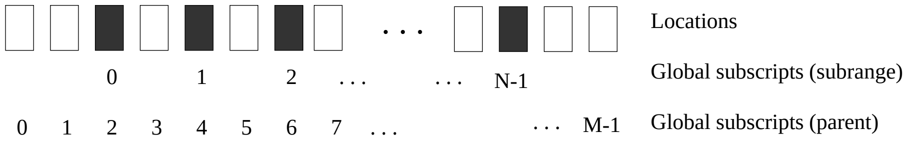

Figure 2: Locations of a subrange (shaded slots).

The syntax for creation of subranges is

```
x | 0 : 49
```

or

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

for a strided subrange. As in Fortran, an upper or lower bound may be omitted in which case it defaults to the bound of the parent range.

Subranges retain an *alignment relation* to their parent range. Locations in a subrange are locations of the parent range. Note, however, that the *global* subscripts for a subrange are in the range $0, \ldots, N-1$ where N is the extent of the *subrange*. See figure 2.

Arrays can be created with subranges, for example

```
int [[,]] e = new int [[(x | 0 : 50), (y | 1 : 198 : 2)]] on p ;
```

In HPF terms, `e` has a non-trivial linear alignment to the template spanned by `x` and `y`. By allowing subranges (and subgroups, see appendix A) to appear in array constructors we reproduce the two-level alignment model of HPF in full generality, at little cost in terms of syntax extensions. The HPJava version may be slightly less readable than the HPF version, but after all this is an advanced feature which is unlikely to be needed by the casual programmer.

Subranges can be used to parametrize `over` constructs

```
Range z = x | 1 : 98 : 2 ;

over(z)
  b [[z]] = z ;
```

Because a location in a subrange is a location in the parent range, `z` could be used to subscript `b` in the loop even though `b` was declared with range `x`. The alignment relation between `x` and `z` determines which element is selected.

The shorthand

```
over(x | 1 : 98 : 2)
  b [[x]] = x ;
```

is also allowed. This has a slightly different effect. The same locations are involved, but the global subscript value assigned to the array elements is now the global subscript in the parent range `x`, not the global subscript in the subrange.

7 Array sections

HPJava provides array sections directly analogous to Fortran 90 array sections. The syntax of section subscripting is similar to local subscripting except that single brackets are used. The subscripts can be locations or ranges.

```
a [i, z]

b [i]
```

Here `i` is a location in the first range of `a` and `b`, and `z` is a subrange of the second range of `a`. The first expression represents a one-dimensional distributed array providing an alias for a subset of elements of `a`. The second section contains a single element of `b`, but the result is a *distributed scalar*, *not* a simple variable. Section subscripting always yields an distributed array or distributed scalar—never an array element.

Formally, local subscripting can be defined in terms of section subscription. It is equivalent to a section subscripting operation returning a distributed scalar, followed by a `#` "dereferencing" operation to uncover the variable "inside" the scalar:

```
a [[i, j, ...]]  ≡  # a [i, j, ...]
```

The `#` operation is only meaningful if the local processor holds a copy of the variable.

8 Nested parallelism

Consider this code, which is supposed to implement a skewed shift of the data in `a`, putting the result in `b`:

```
float [[,]] a = new float [[x, y]] ;
float [[,]] b = new float [[x, y]] ;

over(y)
  Adlib.shift(b [x, y], a [x, y], y, 0, CYCL) ;
```

The shift operation is applied in parallel to the rows of `a` and `b`, with a shift amount given by the global index of the parallel loop.

A priori, there is an ambiguity in the use of `y` as a section subscript. Recall that ranges and locations are both valid as section subscripts. Is `y` to be used directly as a range or converted to a location? The second option is chosen: if the parametric range of an *at* or *over* appears as a subscript in the body of the construct, it is always converted to a location.

9 Collapsed ranges

A *collapsed* range is a sequential range, mapped entirely to the local process. A collapsed range is a `Range`.

The constructor for a collapsed range is illustrated by

```
Range z = new Range(50) ;
```

This creates a collapsed range of extent 50.

It is possible to treat a collapsed range in exactly the same way as any other range. But because this kind of range is very common, and the computations involved in subscripting a collapsed range are simpler than for a distributed range, it can pay to tell the compiler explicitly that a particular range of an array is collapsed. An asterisk in the corresponding slot of the distributed array type signature conveys this information[4]. As a convenience, an integer value can stand for a collapsed range in an array constructor. So one can write simply

```
float [[,*]]  a = new float [[x, 20]] on p ;
float [[*,*]] b = new float [[50, 50]] ;
```

`a` is an array distributed over `p` with one distributed range and one collapsed range (by definition, a collapsed range is orthogonal to any other range). `b` is an array replicated over the active process group with two collapsed ranges. For most practical purposes `b` is equivalent to a two dimensional sequential array.

As a further convenience, to make the analogy with sequential arrays even stronger, it is allowed to use simple integer values as locations for collapsed ranges. We emphasize that this is *only* allowed for collapsed ranges—it is a compile-time or run-time error to use an integer as a location in a range that is not collapsed. With the declarations above the following examples are now legal

```
on(p)
  over(x)
    a [[x, 10]] = x ;

b [[25, 25]] = 13.0 ;
```

To avoid ambiguity, we emphasize that if a range is used in a local subscripting operation, it is always converted to a location, never to an integer global subscript. If we wanted to assign the diagonal of the array `a` we would need an explicit conversion

```
over(x)
  a [[x, (int) x]] = x ;
```

[4]Array signtures with asterisks denote "subtypes" of the corresponding signatures without the asterisks. An array with a collapsed range can be passed as an argument to a function that expects an array with arbitrary distributed range in the corresponding position (but not vice versa).

10 Ghost regions

Ghost regions require some extension to the framework discussed above. A new constructor for `BlockRange` allows specification of ghost widths. Shift operators `>>` and `<<` applicable to ranges return locations a specified number of places to the right or left of the current position (the results are only defined inside an *at* or *over* construct parametrized by the range involved). A collective operation `writeHalo` updates the ghost regions of a distribute array, copying values from the physical segments of neighbouring processes.

```
Procs2 p(2, 2) ;

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 [[,]] u = new int [[x, y]] ;

  // ... some code to initialise 'u'

  Adlib.writeHalo(u) ;

  int [[,]] v = new int [[x, y]] ;
  over(x)
    over(y)
      v [[x, y]] = 0.25 * (u [[x << 1, y]] + u [[x >> 1, y]] +
                           u [[x, y << 1]] + u [[x, y >> 1]]) ;

  Adlib.copy(u, v) ;
}
```

This version only involves a single array temporary and a single array-copy operation (done here using the library function `copy`). By using suitable strided subranges to parametrize the loops this can be converted to a red-black form requiring no array temporaries, something like

```
  over(x)
    over(y | (x + parity) % 2 : : 2)
      u [[x, y]] = 0.25 * (u [[x << 1, y]] + u [[x >> 1, y]] +
                           u [[x, y << 1]] + u [[x, y >> 1]]) ;
```

It is a compile-time or run-time error to use `>>` or `<<` in ways that would imply access to off-processor data.

For the purposes of this example we have assumed that `writeHalo` implements cyclic boundary conditions.

11 Process dimensions

A rank-R process array has R associated *process dimensions*. A process dimension is a `Range`—a *primitive range* over which other ranges can be distributed. The global subscripts are in the range $0, \ldots, P-1$ where P is the extent of the process dimension.

Because process dimensions are ranges, they can appear directly in array declarations and distributed control constructs.

```
...
void task(int [[*]] result, int my_id) { ... }
...
Procs1 p(4) ;

on(p) {
  Range d = p.dim(0) ;

  int [[,*]] results = new int [[d, 100]] ;

  over(d)
    task(results [d, :], d) ;
}
```

The array `v` has 100 integer elements for every process in `p`. The *over* construct assigns the global subscript—in this case the local value of the process coordinate—to each element of `v`.

12 Cyclic ranges

`CyclicRange` is a subclass of `Range` that represents a cyclically distributed range. Ghost regions are not supported for cyclic ranges. Block cyclic distributions can be introduced by distributing a range blockwise over a cyclic range:

```
Range k = CyclicRange(20, p.dim(0)) ;
Range x = BlockRange(100, k) ;
```

13 Global arrays

The language features described above support data parallel programming based on a collective communication library. However the model trivally allows a lower level of SPMD programming through the idiom illustrated in section 11.

Eventually we can add GAs style non-local `get` and `put` members to the distributed arrays (and a handful of other GAs-style synchronization primitives) to support task parallel programming in this framework. Task parallel phases can be freely alternated with SPMD data parallel phases.

A Subgroups

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 array and a dimension set which is some subset of the dimensions of the parent array. The *restriction* operation on a group takes a slice in a particular dimension. This restriction procedure is conveniently expressed in terms of a *location*. 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`.)

For example

```
Procs2 p = new Procs2(3, 2) ;

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

Group q = p / (x | 20) ;

int [[]] d = new int [[y]] on q ;
```

The global subscript 20 maps to coordinate 0 of `p.dim(0)`, so `q` is a one dimensional slice of `p` formed by fixing the first coordinate to 0. In HPF terms, `d` has an *embedded* alignment in the template spanned by `x` and `y`.

Besides their role in specifying this unusual kind of alignment, subgroups allow us to formally state the effects of the *at* and *over* constructs on the active process group. If the current active process group is `p` and `i` is some location in a range distributed over a dimension of `p`, then in

```
at(i)
  S
```

the statement `S` executes in the context of an active process group `p / i`. If the current active process group is `p` and `x` is some range distributed over a dimension of `p`, then in

```
over(x)
  S
```

the statement `S` executes in the context of an active process group `p / x`, assuming the usual conversion from a range to a location.

B Usage rules

The *usage rules* for variables and distributed arrays are required to ensure that

- for ordinary variables, accesses are always local, and
- for distributed arrays, all elements of a distributed array are available in the active process group at the time a collective operation is performed on that array.

This section gives a formal definition of these rules. A programmer with a good intuitive understanding of the SPMD model will probably not need the formal statement given here, but it is provided for completeness.

[It should be possible to simplify these rules slightly now section and local subscripting has been unified.]

First we formally define the phrase *distributed over* for an array or section. If an array constructor specifies a process group `p`, the constructed array is distributed over `p`. If an array constructor doesn't specify a process group the constructed array is distributed over the active process group. For a section, assume the section is expressed in canonical form (see section 7), and the section subscripts include locations `i, j, ...` (interspersed with some range subscripts, which can be ignored here). If the parent array is distributed over `p`, the section is distributed over

```
p / i / j / ...
```

Now, for any ordinary variable (as opposed to a distributed array, which is a collection of variables) we define the *home group* as the group of processes contain a copy of the variable. In general this group includes multiple processes, because the variable may be stored in a replicated manner. For a variable that is not an array element or distributed scalar the home group is the group that was active at the time the variable was created or declared. For an array element, suppose `a` is an array distributed over `p`, and `i, j, ...` are appropriate locations, the home group of

```
a [[i, j, ...]]
```

is

```
p / i / j / ...
```

The usage rules for ordinary variables are now:

- the variable may only be accessed by value when the active process group is contained in the home group of the variable.
- the variable may only be assigned when the active process group is *identical* to the home group of the variable.

In other words, the variable can be read if the local process contains a copy of the variable. To write to a variable, we also require that the active process group—the group executing the write operation—includes the whole of the group over which the variable is replicated.

The rules for collective operations are slightly more complex. The essential idea is that an array can be updated by a collective operation if all copies of all its elements are contained in the active process group, and it can be accessed by value if *at least one copy* of all its elements are contained in the active process group.

More formally, an array is *accessible* if it is distributed over a group *contained in* the active process group. It is *value-accessible* if it is accessible *or* if the group, p, over which the array is distributed has the same parent process array as the active process group, *and* the dimension set of the intersection of p with the active process group contains all process dimensions over which the ranges of the array are distributed. (This is a complicated way of saying something obvious: p may have dimensions outside the active process group, but the array must be replicated over those dimensions.)

For example

```
on(p) {
  float [[,]] a = new float [[x, y]] ;
  float [[,]] b = new float [[x, z]] ;
  float [[,]] c = new float [[z, y]] ;

  over(x)
    over(y)
      a [[x, y]] = Adlib.dotProduct(b [x, z], c [z, y]) ;
}
```

We assume `x`, `y` and `z` distributed over orthogonal dimensions of `p` (otherwise the declarations are illegal). At the point of assignment the active group is `p / x / y`. The section `b [x, z]` is distributed over `p / x`, which means it is not *accessible* at this point. However it is *value-accessible* because the intersection of `p / x` with the active process group is `p / x / y`. The dimension set of this intersection contains the set of dimensions over which the ranges of the section are distributed, namely {`z.dim()`}.

The rules for accessibility in collective operations apply also to array constructors. An array must be accessible at the time it is created!

In some cases the compiler may be able to detect violation of usage rules automatically. This cannot be guaranteed however—in general, if a programmer writes an invalid program this will lead to a run-time error.

References

[1] Bryan Carpenter, "*Adlib* Language Reference",

URL: http://www.npac.syr.edu/users/dbc/Adlib

[2] Special edition of Concurrency, Practise and Experience,

[3] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, Xiaoming Li, "Experiments with *HPJava*"