HPJava Suggestions *[email dated 5 Aug 1996]*

Bryan Carpenter

For debate, some suggestions for a Java syntax (ie, class library interface).

I'm trying to incorporate the basic idea of everything being collective operations on array objects. I haven't defined FORALL "scripts", but the expression trees suggested here could be regarded as a sort of pre-parsed script. The advantage is that we don't need parsers and symbol tables in the run-time. The disadvantage is that the syntax is quite awkward. This is really a peculiarity of Java (see comments below).

One intention with this interface is that it is sufficiently high-level that it can be implemented in terms of a SIMD-style Java controller for an HPC back-end, or of a pure SPMD parallel Java model.

I'm using my 'Range' structures because they allow a simple definition of "conformance", and they are useful in indexed array expressions, which go some way to replacing FORALL.

Unlike HPF, the notation defined here forces communication operations to be explicitly specified. In that sense it is lower level than HPF—closer to a run-time library.

The syntax is quite awkward in practise. A big problem is still that Java doesn't allow library-defined operator overloading, so expressions quickly become difficult to read.

Index ranges

- An index range is specified by a member of the 'Range' class.
- A "primitive range" is either collapsed (sequential), or distributed over a particular dimension of a particular process array.

  ```
  Range x = new Range(100) ;
  Range y = new Range(50, BLK, p.dim(0)) ;
  Range z = new Range(50, BLK, p.dim(1)) ;
  ```

  `x` is collapsed. `y` is distributed blockwise over the first dimension of process array `p`, `z` over the second.

- Two distributed primitive ranges are "orthogonal" if they are distributed over different dimensions of the same process array. A collapsed primitive range is orthogonal to *any* primitive range except itself.

Arrays

- The shape and mapping of an array are defined by an ordered list of orthogonal index ranges.
- The types of array elements are restricted to one of the Java primitive types (cf, F77). There is a separate (sub)class of array for each primitive type.

```
ArrayInt a = new ArrayInt(x, y) ;
ArrayFloat b = new ArrayFloat(y, x) ;
ArrayDouble c = new ArrayDouble(y) ;
ArrayDouble d = new ArrayDouble(z) ;
```

  `a` and `b` are rank 2. `c` is rank 1.

Expressions

- An "expression" is an array expression.

  An expression node may correspond to a complete temporary array constructed at run-time, or through some process of interpretation (akin to interpretation of the mooted of forall scripts) it may simply correspond to a scalar temporary inside a loop in some "node code".

  The classes for expressions parallel the classes for arrays:

```
ExprInt, ExprFloat, ExprDouble, ...
```

- The shape and mapping of an expression are defined by an *unordered set* of orthogonal index ranges.
- An array is an expression. It's shape and mapping are defined by the set of ranges of the array.
- A range is an integer expression. It's shape and mapping are defined by the singleton set containing the range itself.
- Members on the expression class provide arithmetic, eg

```
b.plus(c)
```

corresponds to the array expression

```
b + c
```

The shape of the result is the union of the shape of the addend and the argument. *The operation is illegal if this union contains any non-orthogonal range pairs.*

Certain arithmetic coercions can be defined by overloading `plus`, eg

```
ExprInt   ExprInt :: plus(ExprInt) ;

ExprFloat ExprInt :: plus(ExprFloat) ;

ExprFloat ExprFloat :: plus(ExprFloat) ;

ExprFloat ExprFloat :: plus(ExprInt) ;
```

Other analogous members (`times`, `if`, and so on) are provided.

Simple operations on arrays

- An expression "conforms" with an array if its index range set is a subset of the index range list of the array.
- Assignment is through through suitable members, eg

```
void ArrayInt    :: assign(ExprInt) ;
void ArrayFloat  :: assign(ExprFloat) ;
void ArrayDouble :: assign(ExprDouble) ;
...
```

  Eg,

```
a.assign(b) ;    // a = b
b.assign(c) ;    // b = c
```

  The argument of `assign` must conform with the array being assigned.

```
c.assign(d) ;    // ILLEGAL
```

- The fact that a range is an expression provides some of the power of a "forall" statement...

```
a.assign(x.plus(y)) ;   // forall (x, y) a(x, y) = x + y ;
```

- A masked assignment operation 'where' could also be provided.

Communication

- The conformance rules for the previous operations mean no communication is needed in their implementation.
- The array member `remap` takes an argument of the same *shape* as the array (which does not, however, conform with the array) and copies its value into the array.

  ```
  void ArrayInt    :: remap(ArrayInt) ;
  void ArrayFloat  :: remap(ArrayFloat) ;
  void ArrayDouble :: remap(ArrayDouble) ;
  ...
  ```

  Eg

  ```
  c.remap(d) ;  // c = d
  ```

  This operation clearly *does* require communication.
- `shift` is similar to `remap`, except that the argument *does* conform with the array, but the copy is a shifted copy.

  Again, communication is required.
- Other analogous members are provided.

Subranges and Array sections

- A "subrange" is a triplet subrange of some primitive range, created by a suitable member function, eg

  ```
  x.sub(lb, ub, stride) ;
  ```

  The signature of `sub` would be

  ```
  Range Range :: sub(int, int, int) ;
  ```

  Subranges inherit the orthogonality properties of their parent primitive range.
- A section of an array is created by passing subranges of the array's range set to a suitable member, eg

  ```
  a.sect(x.sub(0, 99, 2), y) ;  // a(0 : 99 : 2, :)
  ```

The signatures of 'sect' would be

```
ArrayInt ArrayInt :: sect(Range x) ;
ArrayInt ArrayInt :: sect(Range x, Range y) ;
...
```

Example

The HPF/F90 program...

```
  integer n
  parameter(n = 8)

  real u(n, n)

  !hpf$ distribute u (block, block) onto p
  !hpf$ processors p (2, 2)

  integer i, j
  integer iter

  do i = 1, n
    do j = 1, n
      if(i == 1 .or. i == n .or. j == 1 .or. j == n) then
        u(i, j) = 1.0
      else
        u(i, j) = 0.0
      endif
    enddo
  enddo

  do iter = 1, 5
    u(2 : n - 1, 2 : n - 1) = &
        0.25 * (u(2 : n - 1, 1 : n - 2) + u(2 : n - 1, 3 : n) + &
                u(1 : n - 2, 2 : n - 1) + u(3 : n, 2 : n - 1))
  enddo

end
```

could be translated as...

```
  final integer n = 8 ;

  Procs p = new Procs(2, 2) ;
```

```
Range x = new Range(n, BLK, p.dim(0)) ;
Range y = new Range(n, BLK, p.dim(1)) ;

ArrayFloat u    = new ArrayFloat(x, y) ;
ArrayFloat mask = new ArrayInt(x, y) ;

mask = x.gt(0).and(x.lt(n - 1)).and(y.gt(0)).and(y.lt(n - 1)) ;

u.where(mask, 0.0, 1.0) ;

ArrayFloat tmp1 = new ArrayFloat(x, y) ;
ArrayFloat tmp2 = new ArrayFloat(x, y) ;
ArrayFloat tmp3 = new ArrayFloat(x, y) ;
ArrayFloat tmp4 = new ArrayFloat(x, y) ;

for(int iter = 1 ; iter <= 5 ; iter++) {
  tmp1.shift(u, -1, 0) ;
  tmp2.shift(u,  1, 0) ;
  tmp4.shift(u, -1, 1) ;
  tmp3.shift(u,  1, 1) ;

  u.where(mask, tmp1.plus(tmp2).plus(tmp3).plus(tmp4).
                   times(0.25)) ;

      // where(mask) u = 0.25 * (tmp1 + tmp2 + tmp3 + tmp4)
}
```

This assumes that the arithmetic members on arrays include boolean arithmetic, and that two variants of masked assignment ('where') are provided (one with two arguments corresponds to F90 simple WHERE; one with three arguments corresponds to F90 WHERE/ELSEWHERE).

Variation: the `where` operation in the loop could be replaced by use of sections...

```
Range i = x.sub(1, n - 2) ;  // subrange 1 : n - 2
Range j = y.sub(1, n - 2) ;  // subrange 1 : n - 2

  ...

  u.sect(i, j).assign(tmp1.sect(i, j).plus(tmp2.sect(i, j)).
               plus(tmp3.sect(i, j)).plus(tmp4.sect(i, j)).
                                         times(0.25)) ;

      // u(2 : n - 1, 2 : n - 1) = 0.25 *
```

```
    //      (tmp1(2 : n - 1, 2 : n - 1) +
    //       tmp2(2 : n - 1, 2 : n - 1) +
    //       tmp3(2 : n - 1, 2 : n - 1) +
    //       tmp4(2 : n - 1, 2 : n - 1))
```

Variation: the `shift` operations could be replaced by `remap` operations...

```
ArrayFloat tmp1 = new ArrayFloat(i, j) ;
ArrayFloat tmp2 = new ArrayFloat(i, j) ;
ArrayFloat tmp3 = new ArrayFloat(i, j) ;
ArrayFloat tmp4 = new ArrayFloat(i, j) ;

...

  tmp1.remap(u.sect(x.sub(2, n - 1), j)) ;
  tmp2.remap(u.sect(x.sub(0, n - 3), j)) ;
  tmp3.remap(u.sect(i, y.sub(2, n - 1))) ;
  tmp4.remap(u.sect(i, y.sub(0, n - 3))) ;

  // tmp3 = u(1 : n - 2, 2 : n - 1)
  // tmp2 = u(0 : n - 3, 1 : n - 2)
  // tmp1 = u(2 : n - 1, 1 : n - 2)
  // tmp4 = u(1 : n - 2, 0 : n - 3)

  u.sect(i, j).assign(tmp1.plus(tmp2).plus(tmp3).plus(tmp4).
                      times(0.25)) ;

      // u(2 : n - 1, 2 : n - 1) = 0.25 *
      //                           (tmp1 + tmp2 + tmp3 + tmp4)
```

This corresponds more directly to the original Fortran which uses arithmetic on array sections, rather than shift operations.