Towards a Java Environment for SPMD Programming

Bryan Carpenter, Guansong Zhang, Geoffrey Fox
Xiaoming Li*, Xinying Li and Yuhong Wen

NPAC at Syracuse University
Syracuse, New York,
NY 13244, USA
{dbc,zgs,gcf,lxm,xli,wen}@npac.syr.edu

Abstract. As a relatively straightforward object-oriented language, Java is a plausible basis for a scientific parallel programming language. We outline a conservative set of language extensions to support this kind of programming. The programming style advocated is Single Program Multiple Data (SPMD), with parallel arrays added as language primitives. Communications involving distributed arrays are handled through a standard library of collective operations. Because the underlying programming model is SPMD programming, direct calls to other communication packages are also possible from this language.

1 Introduction

Java boasts a direct simplicity reminiscent of Fortran, but also incorporates many of the important ideas of modern object-oriented programming. Of course it comes with an established track-record in the domains of Web and Internet programming. The idea that Java may enable new programming environments, combining attractive user interfaces with high performance computation, is gaining increasing attention amongst computational scientists [7, 8].

This article will focus specifically on the potential of Java as a language for scientific parallel programming. We envisage a framework called *HPJava*. This would be a general environment for parallel computation. Ultimately it should combine tools, class libraries, and language extensions to support various established paradigms for parallel computation, including shared memory programming, explicit message-passing, and array-parallel programming. This is a rather ambitious vision, and the current article only discusses some first steps towards a general framework. In particular we will make specific proposals for the sector of HPJava most directly related to its namesake: High Performance Fortran.

For now we do not propose to import the full HPF programming model to Java. After several years of effort by various compiler groups, HPF compilers are still quite immature. It seems difficult justify a comparable effort for Java

* Current address: Peking University

before success has been convincingly demonstrated in Fortran. In any case there are features of the HPF model that make it less attractive in the context of the integrated parallel programming environment we envisage. Although an HPF program *can* interoperate with modules written in other parallel programming styles through the HPF extrinsic procedure interface, that mechanism is quite awkward. Rather than follow the HPF model directly, we propose introducing some of the characteristic ideas of HPF—specifically its distributed array model and array intrinsic functions and libraries—into a basically SPMD programming model. Because the programming model is SPMD, direct calls to MPI [1] or other communication packages are allowed from the HPJava program.

The language outlined here provides HPF-like distributed arrays as language primitives, and new *distributed control* constructs to facilitate access to the local elements of these arrays. In the SPMD mold, the model allows processors the freedom to independently execute complex procedures on local elements: it is not limited by SIMD-style array syntax. 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 simplify error-prone tasks such as converting between local and global array subscripts and determining which processor holds a particular element. As in HPF, it is possible to write programs at a natural level of abstraction where the meaning is insensitive to the detailed mapping of elements. Lower-level styles of programming are also possible.

2 Multidimensional Arrays

First we describe a modest extension to Java that adds a class of true multidimensional arrays to the standard Java language. The new arrays allow regular section subscripting, similar to Fortran 90 arrays. The syntax described in this section is a subset of the syntax introduced later for parallel arrays and algorithms: the only motivation for discussing the sequential subset first is to simplify the overall presentation.

No attempt is made to integrate the new multidimensional arrays with the standard Java arrays: they are a new kind of entity that coexists in the language with ordinary Java arrays. There are good technical reasons for keeping the two kinds of array separate[2]. The type-signatures and constructors of the multidimensional array use double brackets to distinguish them from ordinary arrays:

```
int [[,]] a = new int [[5, 5]] ;
```

[2] For example, the run-time representation of our multi-dimensional arrays includes extra descriptor information that would encumber the large class "non-scientific" Java applications.

```
float [[,,]] b = new float [[10, n, 20]] ;

int [[]] c = new int [[100]] ;
```

`a`, `b` and `c` are respectively 2-, 3- and one- dimensional arrays. Of course `c` is very similar in structure to the standard array `d`, created by

```
int [] d = new int [100] ;
```

`c` and `d` are not identical, though. For example, `c` allows section subscripting (see below), whereas `d` does not. The value `c` would not be assignable to `d`, or vice versa..

Access to individual elements of a multidimensional array goes through a subscripting operation involving single brackets, for example

```
for(int i = 0 ; i < 4 ; i++)
  a [i, i + 1] = i + c [i] ;
```

For reasons that will become clearer in later sections, this style of subscripting is called *local subscripting*. In the current sequential context, apart from the fact that a single pair of brackest may include several comma-separated subscripts, this kind of subscripting works just like ordinary Java array subscripting. Subscripts always start at zero, in the ordinary Java or C style (there is no Fortran-like lower bound).

Our HPJava imports a Fortran-90-like idea of array *regular sections*. The syntax for *section subscripting* is different to the syntax for local subscripting. Double brackets are used. These brackets can include scalar subscripts or *subscript triplets*. A section is an object in its own right—its type is that of a suitable multi-dimensional array. It describes some subset of the elements of the parent array.

```
int [[]] e = a [[2,  2 :]] ;

foo(b [[ : , 0, 1 : 10 : 2]]) ;
```

`e` becomes an alias for the 3rd row of elements of `a`. The procedure `foo` should expect a two-dimensional array as argument. It can read or write to the set of elements of `b` selected by the section. As in Fortran, upper or lower bounds can be omitted in triplets, defaulting to the actual bound of the parent array, and the stride entry of the triplet is optional.

In general our language has no idea of Fortran-like array assignments. In

```
int [[,]] e = new int [[n, m]] ;
...
a = e ;
```

the assignment simply copies a handle to object referenced by `e` into `a`. There is no element-by-element copy involved. On the other hand the language provides a standard library of functions for manipulating its arrays, closely analogous to the array transformational intrinsic functions of Fortran 90:

```
int [[,]] f = new int [[5, 5]] ;
HPJlib.shift(f, a, -1, 0, CYCL) ;

float g = HPJlib.sum(b) ;

int [[]] h = new int [[100]] ;
HPJlib.copy(h, c) ;
```

The `shift` operation with shift-mode `CYCL` executes a cyclic shift on the data in its second argument, copying the result to its first argument—an array of the same shape. In the example the shift amount is -1, and the shift is performed in dimension 0 of the array—the first of its two dimensions. The `sum` operation simply adds all elements of its argument array. The `copy` operation copies the elements of its second argument to its first—it is something like an array assignment. These functions can be overloaded to apply to some finite set of array types. In the initial implementation of the language, the new arrays will be restricted to taking elements of primitive type. This is not regarded as an essential limit to the language, but it simplifies various aspects of the implementation, such as the communication library.

3 Distributed Arrays

HPJava adds class libraries and some additional syntax for dealing with *distributed arrays*. These arrays are viewed as coherent global entities, but their elements are divided across a set of cooperating processes. As a preliminary to introducing distributed arrays we discuss the *process arrays* over which their elements are scattered.

A base class `Group` describes a general group of processes. It has subclasses `Procs1`, `Procs2`, ..., representing one-dimensional process arrays, two-dimensional process arrays, and so on.

```
Procs2 p = new Procs2(2, 2) ;
Procs1 q = new Procs1(4) ;
```

These declarations set `p` to represent a 2 by 2 process array and `q` to represent a 4-element, one-dimensional process array. In either case the object created describes a group of 4 processes. At the time the `Procs` constructors are executed the program should be executing on four or more processes. Either constructor selects four processes from this set and identifies them as members of the constructed group.

The multi-dimensional structure of a process array is reflected in its set of *process dimensions*. An object is associated with each dimension. These objects are accessed through the inquiry member `dim`:

```
Dimension x = p.dim(0) ;
Dimension y = p.dim(1) ;

Dimension z = q.dim(0) ;
```

As indicated, the object returned by the `dim` inquiry has class `Dimension`.

Now, some or all of the dimensions of a multi-dimensional array can be declared as *distributed ranges*. In general a distributed range is represented by an object of class `Range`. A `Range` object defines a range of integer subscripts, and defines how they are mapped into a process array dimension. For example, the class `BlockRange` is a subclass of `Range` which describes a simple block-distributed range of subscripts. Like `BLOCK` distribution format in HPF, it maps blocks of contiguous subscripts to each element of its target process dimension[3]. The constructor of `BlockRange` usually takes two arguments: the extent of the range and a `Dimension` object defining the process dimension over which the new range is distributed.

The syntax of Sect. 2 is extended in the following way to support distributed arrays

- A distributed range object may appear in place of an integer extent in the "constructor" of the array (the expression following the `new` keyword).
- If a particular dimension of the array has a distributed range, the corresponding slot in the type signature of the array should include a `#` symbol. (From the point of view of the type hierarchy, the sequential multi-dimensional arrays of the last section are regarded as a specialization of the more general distributed distributed array class embellished with `#` symbols).
- In general the constructor of the distributed array must be followed by an `on` clause, specifying the process group over which the array is distributed. Distributed ranges of the array must be distributed over distinct dimensions of this group. The `on` clause can be omitted in some circumstances—see Sect. 4.

For example, in

```
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 ;
```

`a` is created as a 100×200 array, block-distributed over the 6 processes in `p`. The fragment is essentially equivalent to the HPF declarations

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

      REAL a(100, 200)

!HPF$ DISTRIBUTE a(BLOCK, BLOCK) ONTO p
```

Because `a` is declared as a collective object we can apply collective operations to it. The `HPJlib` functions introduced in Sect. 2 apply equally well to distributed arrays, but now they imply inter-processor communication.

[3] Other range subclasses include `CyclicRange`, which produces the equivalent of `CYCLIC` distribution format in HPF.

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

HPJlib.shift(a, b, -1, 0, CYCL) ;
```

At the edges of the local segment of `a` the `shift` operation causes the local values of `a` to be overwritten with values of `b` from a processor adjacent in the `x` dimension.

Subscripting operations on distributed arrays are subject to a strict restriction. As already emphasized, the HPJava model is explicitly SPMD. An array access such as

```
a [17, 23] = 13 ;
```

is legal, but *only* if the local process holds the element in question. The language provides several *distributed control* constructs to alleviate the inconvenience of this restriction.

4 The *on* Construct and the Active Process Group

The class `Group` (of which the process array classes are special cases) has a member function called `local`. This returns a boolean value which is `true` if the local process is a member of the group, `false` otherwise. In

```
if(p.local()) {
  ...
}
```

the code inside the conditional is executed only if the local process is a member `p`. We can say that inside this construct the *active process group* is restricted to `p`.

Our language provides a short way of writing this construct

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

The *on* construct provides some extra value. The language incorporates a formal idea of the active process group (APG). At any point of execution some process group is singled out as the APG. An `on(p)` construct specifically changes the value of the APG to `p`. On exit from the construct, the APG is restored to its value on entry.

Elevating the active process group to a part of the language allows some simplifications. For example, it provides a natural default for the `on` clause in array constructors. More importantly, formally defining the active process group simplifies the statement of various rules about what operations are legal *inside* distributed control constructs like *on*.

5 Locations and the *at* Construct

Returning to the example at the end of Sect. 3, we need a mechanism to ensure that the array access

```
a [17, 23] = 13 ;
```

is legal, because the local process holds the element in question. In general determining whether an element is local may be a non-trivial task.

In practise it is unusual to use integer values directly as local subscripts in distributed array dimensions. Instead the idea of a *location* is introduced. A location can be viewed as an abstract element, or "slot", of a distributed range. Conversely, a range can be thought of as a set of locations. An individual location is described by an object of the class `Location`. Each `Location` element is mapped to a particular slice of a process grid. In general two locations are identical only if they come from the same position in the same range. A subscripting syntax is used to represent location `n` in range `x`:

```
Location i = x [n]
```

This is an important idea in HPJava. By working in terms of abstract locations—elements of distributed ranges—one can usually respect locality of reference without resorting explicitly to low-level local subscripts and process ids. In fact the location can be viewed as an abstract data type incorporating these lower-level offsets. The data fields of `Location` include `dim` and `crd`. The first is the process dimension of the parent range. The second is the coordinate in that dimension to which the element is mapped.

Locations are used to parametrize a new distributed control construct called the *at* construct. This is analogous to *on*, except that its body is executed only on processes that hold the specified location. Locations can also be used directly as array subscripts, in place on integers. So the access to element `a [17, 23]` could now be safely written as follows:

```
Location i = x [17], j = y [23] ;

at(i)
  at(j)
    a [i, j] = 13 ;
```

Locations used as array subscripts must be elements of the corresponding ranges of the array.

There is a restriction that an `at(i)` construct should only appear at a point of execution where `i.dim` is a dimension of the active process group. In the examples of this section this means that an `at(i)` construct, say, should normally be nested directly or indirectly inside an `on(p)` construct.

The range class has a member function `idx` which can be used to recover the integer subscript, given a location in the range.

6 Distributed Loops

The *at* mechanism of the previous section is often useful, but in practice good parallel algorithms do not spend much time assigning to isolated elements of distributed arrays. A more urgent requirement is a mechanism for *parallel* access to distributed array elements.

The last and most important distributed control construct in the language is called *over*. It implements a distributed parallel loop. Conceptually it is quite similar to the `FORALL` construct of Fortran, except that the *over* construct specifies exactly where its parallel iterations are to be performed. The argument of *over* is a member of the special class `Index`. This class is a subclass of `Location`, so it is syntactically correct to use an index as an array subscript (the effect of such subscripting is only well-defined inside an *over* construct parametrised by the index in question). Here is an example of a pair of nested *over* loops:

```
float [[#,#]] a = new float [[x, y]],
              b = new float [[x, y]] ;
...
Index i, j ;
over(i = x | :)
  over(j = y | :)
    a [i, j] = 2 * b [i, j] ;
```

The body of an *over* construct executes, conceptually in parallel, for every location in the range of its index (or some subrange if a non-trivial triplet is specified). An individual "iteration" executes on just those processors holding the location associated with the iteration. The net effect of the example above should be reasonably clear. It assigns twice the value of each element of `b` to the corresponding element of `a`. Because of the rules about *where* an individual iteration iterates, the body of an *over* can usually only combine elements of arrays that have some simple alignment relation relative to one another. The `idx` member of range can be used in parallel updates to yield expressions that depend on global index values.

With the *over* construct we can give some useful examples of parallel programs.

Figure 1 gives a parallel implementation of Cholesky decomposition in the extended language. The first dimension of `a` is sequential ("collapsed" in HPF parlance). The second dimension is distributed (cyclically, to improve load-balancing). This a column-oriented decomposition. The example involves one new operation from the standard library. The function `remap` copies the elements of one distributed array or section to another of the same shape. The two arrays can have any, unrelated decompositions. In the current example `remap` is used to implement a broadcast. Because `b` has no range distributed over `p`, it implicitly has *replicated* mapping; `remap` accordingly copies identical values to all processors. This example also illustrates construction of sections of distributed arrays, and use of non-trivial triplets in the *over* construct.

Figure 2 gives a parallel implementation of red-black relaxation in the extended language. To support this important stencil-update paradigm, *ghost re-*

```
Procs1 p = new Procs1(P) ;
on(p) {
  Range x = new CyclicRange(N, p.dim(0));

  float [[,#]] a = new float [[N, x]] ;

  float [[]]   b = new float [[N]] ;  // buffer

  // ... some code to initialise 'a'

  Location l ;
  Index m ;

  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 ;
    }

    HPJlib.remap(b [[k + 1 : ]], a [[k + 1 : , k]]);

    over(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]) ;
}
```

Fig. 1. Choleksy decomposition.

gions are allowed on distributed arrays. Ghost regions are extensions of the locally held block of a distributed array, used to cache values of elements held on adjacent processors. In our case the width of these regions is specified in a special form of the `BlockRange` constructor. The ghost regions are explicitly brought up to date using the library function `writeHalo`. Its arguments are an array with suitable extensions and a vector defining in each dimension the width of the halo that must actually be updated.

Note that the new range constructor and `writeHalo` function are *library* features, not new language extensions. One new piece of syntax is needed: the addition and subtraction operators are overloaded so that integer offsets can be

```
Procs2 p = new Procs2(P, P) ;

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

  float [[#,#]] u = new float [[x, y]] ;

  int [] widths = {1, 1} ;        // Widths updated by 'writeHalo'

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

  for(int iter = 0 ; iter < NITER ; iter++) {
    for(int parity = 0 ; parity < 2 ; parity++) {

      HPJlib.writeHalo(u, widths) ;

      Index i, j ;
      over(i = x | 1 : N - 2)
        over(j = y | 1 + (x.idx(i) + parity) % 2 : N - 2 : 2)
          u [i, j] = 0.25 * (u [i - 1, j] + u [i + 1, j] +
                             u [i, j - 1] + u [i, j + 1]) ;
    }
  }
}
```

Fig. 2. Red-black iteration using `writeHalo`.

added or subtracted to locations, yielding new, shifted, locations. This kind of shifted access is illegal if it implies access to off-processor data. It only works if the subscripted array has suitable ghost extensions.

We have covered most of the important *language* features we propose to implement. Two additional features that are quite important in practice but have not been discussed are *subranges* and *subgroups*. A subrange is simply a range which is a regular section of some other range, created by syntax like `x [0 : 49]`. Subranges can be used to create distributed arrays with general HPF-like alignments. A *subgroup* is some slice of a process array, formed by restricting process coordinates in one or more dimensions to single values. Subgroups formally describe the state of the active process group inside *at* and *over* constructs. For a more complete description of a slightly earlier version of the proposed language, see [3].

7 Discussion

We have described a conservative set of extensions to Java. In the context of an explicitly SPMD programming environment with a good communication library, we claim these extensions provide much of the concise expressiveness of HPF, without relying on very sophisticated compiler analysis. The object-oriented features of Java are exploited to give an elegant parameterization of the distributed arrays in the extended language. Because of the relatively low-level programming model, interfacing to other parallel-programming paradigms is more natural than in HPF. With suitable care, it is possible to make direct calls to, say, MPI from within the data parallel program (in [2] we suggest a concrete Java binding for MPI).

The language extensions described were devised partly to provide a convenient interface to a distributed-array library developed in the PCRC project [5, 4]. Hence most of the run-time technology needed to implement the language is available "off-the-shelf". The existing library includes the run-time descriptor for distributed arrays and a comprehensive array communication library. The HPJava compiler itself is being implemented initially as a translator to ordinary Java, through a compiler construction framework also developed in the PCRC project [12].

The distributed arrays of the extended language will appear in the emitted code as a pair—an ordinary Java array of local elements and a Distributed Array Descriptor object (DAD). Details of the distribution format, including non-trivial details of global-to-local translation of the subscripts, are managed in the run-time library. Acceptable performance should nevertheless be achievable, because we expect that in useful parallel algorithms most work on distributed arrays will occur inside *over* constructs. In normal usage, the formulae for address translation can then be linearized. The non-trivial aspects of address translation (including array bounds checking) can be absorbed into the startup overheads of the loop. Since distributed arrays are usually large, the loop ranges are typically large, and the startup overheads (including all the run-time calls associated with address translation) can be amortized. This approach to translation of parallel loops is discussed in detail in [4].

Note that if array accesses are genuinely irregular, the necessary subscripting cannot usually be *directly* expressed in our language, because subscripts cannot be computed randomly in parallel loops without violating the fundamental SPMD restriction that all accesses be local. This is not regarded as a shortcoming: on the contrary it forces explicit use of an appropriate library package for handling irregular accesses (such as CHAOS [6]). Of course a suitable binding of such a package is needed in our language.

A complementary approach to communication in a distributed array environment is the one-sided-communication model of Global Arrays (GA) [9]. For task-parallel problems this approach is often more convenient than the schedule-oriented communication of CHAOS (say). Again, the language model we advocate here appears quite compatible with GA approach—there is no obvious

reason why a binding to a version of GA could not be straightforwardly integrated with the the distributed array extensions of the language described here.

Finally we mention two language projects that have some similarities. Spar [11] is a Java-based language for array-parallel programming. There are some similarities in syntax, but semantically Spar is very different to our language. Spar expresses parallelism but not explicit data placement or communication—it is a higher level language. ZPL [10] is a new programming language for scientific computations. Like Spar, it is an array language. It has an idea of performing computations over a *region*, or set of indices. Within a compound statement prefixed by a *region specifier*, aligned elements of arrays distributed over the same region can be accessed. This idea has certain similarities to our *over* construct.

References

1. Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with HPJava. *Concurrency: Practice and Experience*, 9(6):633, 1997.
2. Bryan Carpenter, Geoffrey Fox, Xinying Li, and Guansong Zhang. A draft Java binding for MPI. http://www.npac.syr.edu/projects/pcrc/doc.
3. Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xinying Li, and Yuhong Wen. Introduction to Java-Ad. http://www.npac.syr.edu/projects/pcrc/doc.
4. Bryan Carpenter, Guansong Zhang, and Yuhong Wen. NPAC PCRC runtime kernel 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.
5. Parallel Compiler Runtime Consortium. Common runtime support for high-performance parallel languages. In *Supercomputing '93*. IEEE Computer Society Press, 1993.
6. 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.
7. Geoffrey C. Fox, editor. *Java for Computational Science and Engineering—Simulation and Modelling*, volume 9(6) of *Concurrency: Practice and Experience*, June 1997.
8. Geoffrey C. Fox, editor. *Java for Computational Science and Engineering—Simulation and Modelling II*, volume 9(11) of *Concurrency: Practice and Experience*, November 1997.
9. 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.
10. Lawrence Snyder. A ZPL programming guide. Technical report, University of Washington, May 1997. http://www.cs.washington.edu/research/projects/zpl/.
11. Kees van Reeuwijk, Arjan J. C. van Gemund, and Henk J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, 1997.
12. 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.

This article was processed using the LaTeX macro package with LLNCS style