HPJava: data parallel extensions to Java

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

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

December 7, 1997

Abstract

We outline an extension of Java for programming with distributed arrays. The basic programming style is Single Program Multiple Data (SPMD), but parallel arrays are provided as new language primitives. Further extensions include three *distributed control* constructs, the most important being a data-parallel loop construct. Communications involving distributed arrays are handled through a standard library of collective operations. Because the underlying programming model is SPMD programming, direct calls to MPI or other communication packages are also allowed in an HPJava program.

1 Introduction

The idea that Java may enable new programming environments, combining attractive user interfaces with high performance computation, is gaining increasing attention amongst computational scientists. 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.

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. Other paradigms (for example, Linda or coarse-grained data-flow) may come later, together with bindings to higher-level libraries and application-specific libraries such as CHAOS [7], ScaLAPACK [1], Global Arrays [8] or DAGH [9].

This is a large 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. We will be concentrating on array-parallel programming.

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 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 [2] 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.

Our compiler will be implemented as a translator to ordinary Java with calls to a suitable run-time library. At the time of writing the underlying library is already available [5], and the Java interface needed by the translator is under development. The translator itself is being implemented in a compiler construction framework developed in the PCRC project [6, 12].

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[1].

The type-signatures and constructors of the multidimensional array use double brackets to distinguish them from ordinary arrays:

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

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[2].

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).

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

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

[1]The run-time representation of our multi-dimensional arrays includes extra descriptor information that would simply encumber the large class "non-scientific" Java applications.

[2]For example, `c` allows section subscripting, whereas `d` does not.

the assignment simply copies a handle to object referenced by `e` into `a`. There is no element-by-element copy involved. Similarly we introduce no idea of elemental arithmetic or elemental function application. If `e` and `a` are arrays, the expressions

```
e + a
Math.cos(e)
```

are type errors.

Our HPJava *does* import 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. This is slightly different to the situation in Fortran, where sections cannot usually be captured as named entities[3].

```
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. The subscripts of `e`, like any other array, start at 0, although the first element is identified with `a [2, 2]`.

In our language, unlike Fortran, it is not allowed to use vectors of integers as subscripts. The only sections recognized are regular sections defined through scalar and triplet subscripts.

The language provides a 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

[3]Unless a section appears as an actual argument to a procedure, in which case the dummy argument names that section, or it is the target of a pointer assignment.

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 may have to be overloaded to apply to some finite set of array types, eg they may be defined for arrays with elements of any suitable Java primitive type, up to some maximum rank of array. Alternatively the type-hierarchy for arrays can be defined in a way that allows these functions to be more polymorphic.

3 Process 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 pre-requisite to introducing distributed arrays we discuss the *process arrays* over which their elements are scattered.

An abstract base class `Procs` 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[4].

`Procs` has a member function called `member`, returning a boolean value. This is `true` if the local process is a member of the group, `false` otherwise.

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

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

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`:

[4]There is no cooperation between the two constructor calls for `p` and `q`, so an individual physical process might occur in both groups or in neither. As an option not illustrated here, vectors of ids can be passed to the `Procs` constructors to specify exactly which processes are included in a particular group.

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

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

The object returned by the `dim` inquiry has class `Dimension`. The members of this class include the inquiry `crd`. This returns the coordinate of the local process with respect to the process dimension. The result is only well-defined if the local process is a member of the parent process array. The inner body code in

```
if(p.member())
  if(x.crd() == 0)
    if(y.crd() == 0) {
      ...
    }
```

will only execute on the first process from `p`, with coordinates $(0, 0)$.

4 Distributed arrays

Some or all of the dimensions of a multi-dimensional array can be declared to be *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. In fact the `Dimension` class introduced in the previous section is a subclass of `Range`. In this case the integer range is just the range of coordinate values associated with the dimension. Each value in the range is mapped, of course, to the process (or slice of processes) with that coordinate. This kind of range is also called a *primitive range*. More complex subclasses of `Range` implement more elaborate maps from integer ranges to process dimensions. Some of these will be introduced in later sections. For now we concentrate on arrays constructed with `Dimension` objects as their distributed ranges.

The syntax of section 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.
- 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[5].

[5]The `on` clause can be omitted in some circumstances—see section 5.

Assume p, x and y are declared as in the previous section, then

```
float [[#,#,]] a = new float [[x, y, 100]] on p ;
```

defines a as a 2 by 2 by 100 array of floating point numbers. Because the first two dimensions of the array are distributed ranges—dimensions of p—a is actually realized as four segments of 100 elements, one in each of the processes of p. The process in p with coordinates i, j holds the section a [[i, j, :]].

The distributed array a is equivalent in terms of storage to four local arrays defined by

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

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

```
float [[#,#,]] a = new float [[x, y, 100]] on p,
               b = new float [[x, y, 100]] on p ;

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

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

There is a catch in this. When subscripting the distributed dimensions of an array it is *simply disallowed* to use subscripts that refer to off-processor elements. While this:

```
int i = x.crd(), j = y.crd() ;

a [i, j, 20] = a [i, j, 21] ;
```

is allowed, this:

```
int i = x.crd(), j = y.crd() ;

a [i, j, 20] = b [(i + 1) % 2, j, 20] ;
```

is forbidden. The second example could apparently be implemented using a nearest neighbour communication, quite similar to the shift example above. But our language imposes an strict policy distinguishing it from most data parallel languages: while library functions may introduce communications, language primitives such as array subscripting *never* imply communication.

If subscripting distributed dimensions is so restricted, why are the i, j subscripts on the arrays needed at all? In the examples of this section these subscripts are only allowed one value on each processor. Well, the inconvience of specifying the subscripts will be reduced by language constructs introduced later, and the fact that only one subscript value is local is a special feature of the *primitive ranges* used here. The higher level distributed ranges introduced later map multiple elements to individual processes. Subscripting will no longer look so redundant.

5 The *on* construct and the active process group

In the section 3 the idiom

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

appeared. Our language provides a short way of writing this construct

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

In fact the *on* construct provides some extra value. Informally we said in section 3 that the *active process group* is restricted to `p` inside the body of the `p.member()` conditional construct. The language incorporates a more formal idea of an 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*.

6 Higher-level ranges and locations

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[6]. 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.

```
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

[6] Other higher-level ranges include `CyclicRange`, which produces the equivalent of `CYCLIC` distribution format in HPF.

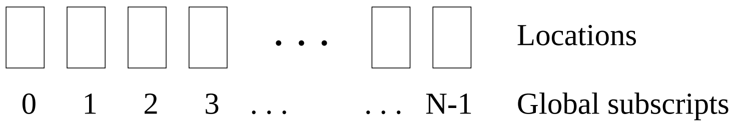

Figure 1: A range regarded as a set of locations, or slots.

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

REAL a(100, 200)

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

Subscripting distributed arrays with non-primitive ranges introduces some new problems. An array access such as

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

is perfectly legal *if* the local process holds the element in question. But deterimining whether an element is local is no longer so easy. When arrays had only *primitive* distributed ranges, it was straightforward to check that accesses were local—the subscript simply had to be equal to the local coordinate. With higher-level ranges, that simple condition no longer holds.

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. This model of a range is visualized in figure 1. 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.

Publically accessible fields of `Location` include `dim` and `crd`. The first is the process dimension of the parent range. The second is coordinate in that dimension to which the element is mapped. So the access to element `a [17, 23]` could now be guarded by conditionals as follows:

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

if(i.crd == i.dim.crd())
```

```
  if(j.crd == j.dim.crd())
    a [17, 23] = 13 ;
```

This is still quite verbose and error-prone. The language provides a second *distributed control construct* (analogous to *on*) to deal with this common situation. The new construct is called *at*, and takes a location as its argument. The fragment above can be replaced with

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

at(i)
  at(j)
    a [17, 23] = 13 ;
```

This is more concise, but still involves some redundancy because the subscripts 17 and 23 appear twice. A natural extension is to allow locations to be used directly as array subscripts:

```
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.

The range class has a member function

```
int Range.idx(Location i)
```

which can be used to recover the integer subscript, given a location in the range.

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.

7 Distributed loops

Good parallel algorithms don't usually expend many lines of code assigning to isolated elements of distributed arrays. The *at* mechanism of the previous section is often useful, but a more pressing need 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`. The class `Index` is a subclass of `Location`, so it is syntactically correct to use an index as an array subscript[7]. Here is an example of a pair of nested *over* loops:

[7]But the effect of such subscripting is only well-defined inside an *over* construct parametrised by the index in question.

```
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)[8]. An individual "iteration" executes on just those processors holding the location associated with the iteration. In a particular iteration, the location component of the index (the base class object) is equal to that location. 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 give expressions that depend on global index values.

With the *over* construct we can give some useful examples of parallel programs. Figure 2 is the famous Jacobi iteration for a two dimensionsional Laplace equation. We have used cyclic `shift` to implement nearest neighbour communications[9].

Copying whole arrays into temporaries is not an efficient way of accessing nearest neighbours in an array. Because this is such a common pattern of communication, the standard library supports *ghost regions*. Distributed arrays can be created in such a way that the segment stored locally is extended with some halo. This halo caches values stored in the segments of adjacent processes. The cached values are explicitly bought up to date by the library operation `writeHalo`.

An optimized version of the Jacobi program is give in figure 3. This version only involves a singe array temporary. A new constructor for `BlockRange` is provided. This allows the width of the ghost extensions to be specified. The arguments of `writeHalo` itself are an array with suitable extensions and two vectors. The first defines in each dimension the width of the halo that must actually be updated, and the second defines the treatment at the ends of the range—in this case the ghost edges are updated with cyclic wraparound. The new constructor and new `writeHalo` function are simply standard library extensions. One new piece of syntax is needed: the addition and subtraction operators are

[8]Formally `|` is being used here as an operator that combines a range and a triplet to return an object of the iterator class `Index`.

[9]Laplace's equation with cyclic boundary conditions is not particularly useful, but it illustrates the language features. More interesting boundary conditions can easily be incorporated later. Incidentally, this is a suitable place to mention that the array arguments of `shift` must be *aligned* arrays—they must have identical distributed ranges.

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

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

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

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

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

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

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

Figure 2: Jacobi iteration using `shift`.

```
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

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

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

  int [] widths = {1, 1} ;        // Widths actually updated
  Mode [] modes = {CYCL, CYCL} ;  // Wraparound at ends.

  HPJlib.writeHalo(u, widths, modes) ;

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

  Index i, j ;
  over(i = x | :)
    over(j = y | :)
      v [i, j] = 0.25 * (u [i + 1, j] + u [i - 1, j] +
                         u [i, j + 1] + u [i, j + 1]) ;

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

Figure 3: Jacobi iteration using `writeHalo`.

overloaded so that integer offsets can be added or subtracted to `Location` objects, yielding new, shifted, locations. The usual access rules apply—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.

8 Other features

We have already described 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 are created tacitly when a distributed array is subscripted with a triplet, and they can also be used directly 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. Again they may be created implicitly by section subscripting, this time using a scalar subscript. They also formally describe the state of the active process group inside *at* and *over* constructs.

The framework described is much more powerful than space allows us to demonstrate in this paper. This power comes in part from the flexibility to add features by extending the libraries associated with the language. We have only illustrated the simplest kinds of distribution format. But any HPF 1.0 array distribution format, plus various others, can be incorporated by extending the `Range` hierarchy in the run-time library. We have only illustrated `shift` and `writeHalo` operations from the communication library, but the library also includes much more powerful operations for remapping arrays and performing irregular data accesses. Our intention is to provide minimal language support for distributed arrays, just enough to facilitate further extension through construction of new libraries.

For a more complete description of a slightly earlier version of the proposed language, see [4].

9 Discussion and related work

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 of 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 [3] we suggest a concrete Java binding for MPI.

We will mention two related projects. Spar [11] is a Java-based language for array-parallel programming. Like our language it introduces multi-dimensional arrays, array sections, and a parallel loop. 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. Communication is more explicit than Spar, but not as explicit as in the language discussed in this article.

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. SIAM, 1997.

[2] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with HPJava. *Concurrency: Practice and Experience*, 9(6):633, 1997.

[3] Bryan Carpenter, Geoffrey Fox, Xinying Li, and Guansong Zhang. A draft Java binding for MPI. URL: http://www.npac.syr.edu/projects/pcrc/July97/doc.html.

[4] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xinying Li, and Yuhong Wen. Introduction to Java-Ad. URL: http://www.npac.syr.edu/projects/pcrc/July97/doc.html.

[5] Bryan Carpenter, Guansong Zhang, and Yuhong Wen. *NPAC PCRC Runtime Kernel Definition*, 1997. In preparation. For current draft, see http://www.npac.syr.edu/projects/pcrc/July97/doc.html.

[6] Parallel Compiler Runtime Consortium. Common runtime support for high-performance parallel languages. In *Supercomputing '93*. IEEE Computer Society Press, 1993.

[7] 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.

[8] 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.

[9] Manish Parashar and J.C. Browne. Systems engineering for high performance computing software: The HDDA/DAGH infrastructure for implementation of parallel structured adaptive mesh. In *Structured Adaptive Mesh Refinement Grid Methods*, IMA Volumes in Mathematics and its Applications. Springer-Verlag.

[10] Lawrence Snyder. A ZPL programming guide. Technical report, University of Washington, May 1997. URL: 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.