Introduction. **This article focuses on the potential of Java as a language for scientific parallel programming. We envisage a framework for parallel computing called** *HPJava*. **Here we describe some first steps towards a general framework, making specific proposals for the sector of HPJava most directly related to its namesake: High Performance Fortran. 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 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.**

Distributed arrays. **HPJava adds class libraries and some additional syntax for dealing with** *distributed arrays*. **Some or all of the dimensions of a these arrays can be declared as** *distributed ranges*. **A distributed range defines a range of integer subscripts, and specifies how they are mapped into a process grid dimension. It is represented by an object of base class** `Range`.

Process grids—equivalent to *processor arrangements* **in HPF—are described by suitable classes. A base class** `Group` **describes a general group of processes and has subclasses** `Procs1`, `Procs2`, **. . . , representing one-dimensional process grids, two-dimensional process grids, and so on. The inquiry function** `dim` **returns an object describing a particular dimension of a grid.**

In the example

```
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** `Range` **subclass** `BlockRange` **describes a simple block-distributed range of subscripts—analogous** `BLOCK` **distribution format in HPF. The arguments of the** `BlockRange` **constructor are the extent of the range and an object defining the process grid dimension over which the range is distributed. The fragment above is essentially equivalent to the HPF declarations**

```
!HPF$ PROCESSORS p(3, 2)
      REAL a(100, 200)
!HPF$ DISTRIBUTE a(BLOCK, BLOCK) ONTO p
```

In HPJava the type-signatures and constructors of distributed arrays use double brackets to distinguish them from ordinary Java arrays. If a particular dimension of an 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 (it defaults to the APG, see below). Distributed ranges of the array must be distributed over distinct dimensions of this group.

The language provides a library of functions for manipulating its arrays, closely analogous to the array transformational intrinsic functions of Fortran 90:

```
float [[#,#]] b = new float [[x, y]] on p ;
HPJlib.shift(b, a, -1, 0, CYCL) ;

float g = HPJlib.sum(b) ;
```

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. The `sum` operation simply adds all elements of its argument array. In general these functions imply inter-processor communication.

The *on* construct and the active process group. **Often in SPMD programming it is necessary to restrict execution of a block of code to processors in a particular group** `p`. **Our language provides a short way of writing this construct**

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

The language incorporates a 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.**

Locations and the *at* construct. 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 further constructs to alleviate the inconvenience of this restriction.

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`. Any location is mapped to a particular slice of a process grid. The syntax `x [n]` represents location `n` in range `x`. 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. The array access above can be safely written as:

```
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.

Distributed loops. **The last and most important distributed control construct in the language is called *over*. It implements a distributed parallel loop. 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. 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.

Figure 1 gives a parallel implementation of Choleski 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 Fortran-90-like sections of distributed arrays (using double brackets and triplet subscripts) and use of nontrivial triplets in the *over* construct.

Discussion. **We have covered the most important** *language* **features we propose to implement. 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.**

The language extensions described were devised partly to provide a convenient interface to a distributed-array library developed in the Parallel Compiler Runtime Consortium (PCRC) project. 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. 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 is nevertheless achievable, because we expect that in useful parallel algorithms most work on distributed arrays will occur inside *over* **constructs with large ranges. In normal usage, the formulae for address translation can then be linearized, and non-trivial**

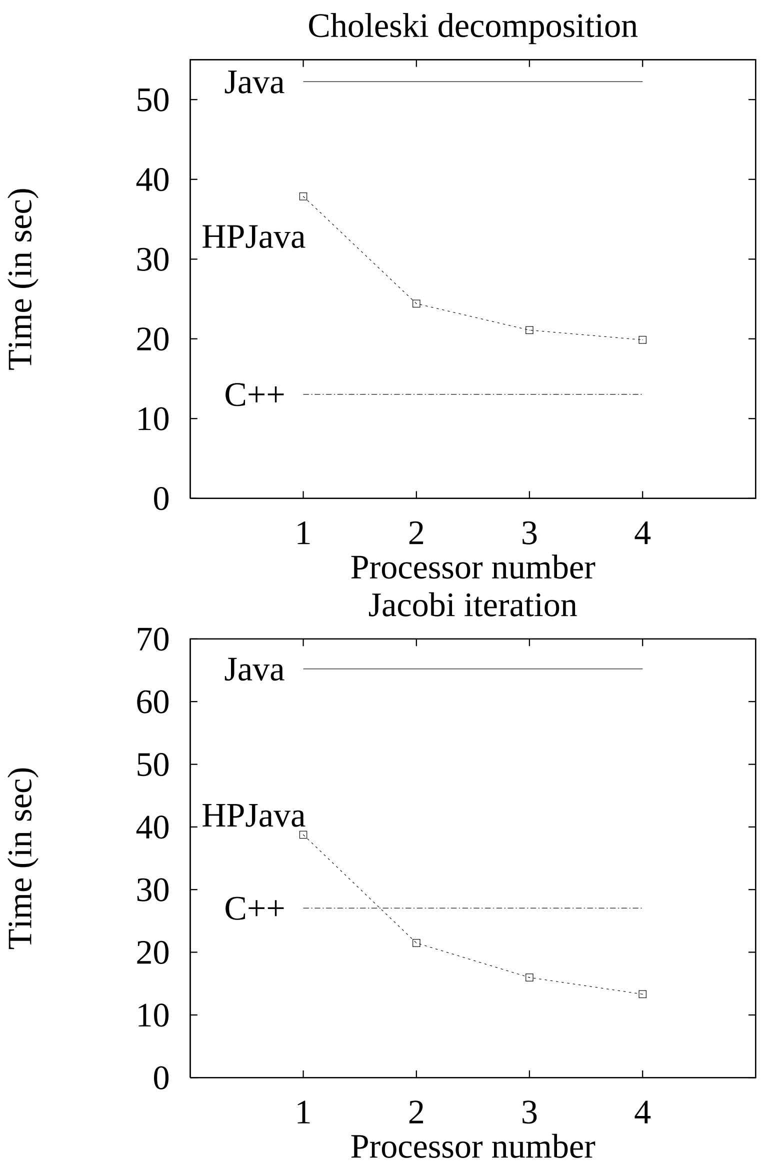
aspects of address translation (including array bounds checking) can be amortized in the startup overheads of the loop. Figure 2 shows a preliminary benchmark for hand translated code of our examples, following the translation scheme outlined above. The parallel programs are executed on 4 sparc-sun-solaris2.5.1 using MPICH and the Java JIT compiler in JDK 1.2Beta2, and Adlib for collective communications. In both cases arrrays are 1024 by 1024. For Jacobi iteration, the timing is for about 90 iterations. Timings are compared the sequential Java and C++ version of the code. (Poor scaling in the Cholesky case is attributable to the poor performance of MPICH on this platform, and will be much improved by using SunHPC MPI.)

If array accesses are genuinely irregular, the necessary subscripting cannot usually be *directly* expressed in our language; subscripts cannot be computed randomly in parallel loops without violating the SPMD restriction that 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). Of course a suitable binding of such a package is needed in our language.

Figure 1. Choleski decomposition.

```
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

  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]) ;
}
```

Figure 2. Benchmarks on hand-translated code.



Java MPI binding **In contrast to HPF, our programming model is explicitly SPMD, with independent threads of control in each processor. It is therefore possible to make calls to message-passing libraries such as MPI from any point in an HPJava program.**

We have proposed a Java binding for MPI, version 1.1. In the style of the C++ bindings in the MPI 2.0 standard, our interface is fully object-oriented, with communication operations defined as members of a hierarchy of communicator classes. With very few exceptions, it provides all the functions of MPI available to Fortran or C programmers.

The MPI binding has been initially implemented as a JNI wrapper on top of a standard C interface to MPI. A complete MPI test suite developed by IBM is being translated to Java, to validate the wrapper code.

References. **An extended version of this article, including references, is available from**

`http://www.npac.syr.edu/projects/pcrc/doc`