Editor: Geoffrey Fox, gcf@indiana.edu

HPJAVA: A DATA PARALLEL PROGRAMMING ALTERNATIVE

By Bryan Carpenter and Geoffrey Fox

TODAY, THERE IS A HEALTHY DIVERSITY OF PROJECTS AIMING TO HARNESS JAVA FOR SCIENTIFIC (OR "GRANDE") COMPUTING. IN FACT, THIS DEPARTMENT LOOKED AT SEVERAL OF THESE APPROACHES IN A RECENT ARTICLE.[1] HERE, WE WILL FOCUS ON A particular project that's ongoing at our Pervasive Technology Lab at Indiana University.

The HPJava (high-performance Java) project aims to support scientific and parallel computing in a modern, object-oriented, Internet-friendly environment—the Java platform. HPJava leverages popular high-performance Fortran (HPF) language and library features such as "scientific" multidimensional array syntax and distributed arrays, while at a more language-independent level, it introduces a slightly unusual parallel programming model, somewhere in between the classical HPF and message-passing interface (MPI) extremes. We'll say more about this later. See the "HPJava History" sidebar for some of the language's developmental roots.

HPJava Philosophy

HPJava lifts some ideas directly from HPF, and its domain of applicability is likely to overlap HPF to a large extent. It has an almost equivalent model of distributed arrays (the only very significant difference in this respect is that we eventually abandoned the "block cyclic" distribution format); you can write HPJava programs that look very much like corresponding HPF programs. But the philosophies of the languages differ in some significant ways. HPJava was designed in a "bottom up" manner.

We started with some parallel libraries, and built the language around making these libraries convenient to use. Arguably, this library-centric approach is more modern, more "object oriented" in spirit. The parallel programming model is also different. An HPF program logically has a single thread of control—similar to a sequential or SIMD (single instruction, multiple data) program. You can view an HPJava program like an SPMD (single program, multiple data) program with multiple threads of control—more similar to an MPI program. This means HPJava maps more straightforwardly onto the most popular parallel architectures. The HPJava compiler itself does not insert any of the communications necessary to access remote data; all this is the programmer's responsibility and resembles an MPI environment. But HPJava supports convenient calls to high-level libraries acting on distributed arrays, so this task typically is easier than it would be in an MPI program.

It was important to be 100 percent compatible with the standard Java platform. This did not necessarily mean that the implementation of all HPJava libraries had to be pure Java in a narrow sense. An acceptable compromise was to provide access to native libraries (MPI, for example) through the Java Native Interface (JNI). What was more important to us was that all existing Java libraries (for networking, GUIs, database access, and so on) could be invoked from an HPJava program directly—without recompiling those libraries. This strongly suggested we should target standard Java Virtual Machines as our execution platform. As it turned out, an HPJava program or library compiles to completely standard Java byte-code files.

Although we have a few extensions at the relatively superficial level of syntax, we can claim that HPJava is unequivocally a Java technology. This contrasts with some other comparable projects, such as UC Berkeley's Titanium and the University of Delft's Timber. These projects extend the Java language for scientific computing, but "compile down" to C or C++. Those approaches offer the hope of tuning for higher ultimate performance, but sacrifice various benefits of the full Java platform.

Now, let's get started using HPJava.

Multiarrays

Entry-level HPJava users might want to take advantage of the syntax the extended language provides for Fortran-like, multidimensional arrays, perhaps initially ignoring the language's parallel aspects. In this area, the syntax is com-

HPJava History

Active in the early 1990s, the High Performance Fortran Forum (www.crpc.rice.edu/HPFF or www.vcpc.univie.ac.at/information/mirror/HPFF/) brought together leading high-performance computing practitioners to define a common language for data parallel computing. Inspired by the success of parallel dialects of Fortran such as Connection Machine Fortran, the resulting high-performance Fortran (HPF) language definition extended the then-standard Fortran 90 with several parallel features, most notably a comprehensive set of directives for describing distributed arrays.

An HPF distributed array behaves programmatically like a normal Fortran array, but its elements are distributed across a collection of processors (instead of being stored in a single address space). The general approach had been very successful in compilers for SIMD (single instruction, multiple data) parallel architectures, but HPF claimed to compile efficiently to the SPMD (single program, multiple data) style of execution required by a newer generation of more loosely coupled MIMD (multiple instruction, multiple data) parallel computers. In particular, HPF was expected to remove the need for the explicit message-passing style of programming for those computers, as embodied in the message-passing interface (MPI) standard. (By a quirk of history, the MPI standard emerged one year after the HPF standard, but the general ideas were older.)

Our HPJava project started around 1997, growing out of our group's earlier HPF work. In a collaborative project with researchers from Peking University and other institutes, we developed runtime libraries to support HPF compilations. The support libraries (some inherited from an earlier, related project at Southampton University) worked quite well, but the HPF language didn't seem to be making the headway people had hoped for. Writing the compilers was difficult, and the underlying program model was more restrictive than some people liked.

In the absence of HPF, we had the option of making our libraries available for explicit call from multiprocessor parallel programs written in C++ (an original implementation language of the libraries). But, without some "syntactic sugar" to represent HPF-like distributed data structures, that approach was going to be either ugly or inefficient. Adding language extensions, even at the syntactic-sugar level, seemed like a daunting prospect if C++—with its notoriously complex structure—was the base language.

We thought Java might be a better solution. People were starting to talk about Java Grande and the possibility that eventually Java could be a high-performance language. The two languages were similar enough that we could call our libraries from Java. On the other hand, the syntax was considerably simpler, and perhaps there was scope to extend it with the features we wanted for data-parallel computing. We threw around a lot of proposals for language extensions and estimated we could produce a prototype translator for the extended language in "a few months."

That first prototype was working some time in 2000. We immediately started to rewrite it with a more effective translation scheme and better compile-time checking. The first release of the software is available for free download from our Web site, www.hpjava.org.

patible with recent proposals from the Java Grande Numerics Working Group (http://math.nist.gov/javanumerics/). Following that group's lead, we call the new arrays *multiarrays*. HPJava has multiarrays and ordinary Java arrays, and you can choose which is most convenient according to context. A multiarray can have elements of any standard Java type as well as any *rank* (dimensionality).

Standard Java provides multidimensional arrays, but they are implemented as arrays of arrays. This means that they can be "ragged" (for example, not all rows of a two-dimensional array must be the same length). This can be useful in some applications but it makes it harder for a compiler to analyze and optimize array use and difficult to define general array sections. In contrast, the new HPJava multiarrays have a fixed rectangular aspect, similar to Fortran arrays.

Our syntax for multiarrays uses double brackets to distinguish type signatures from standard Java arrays (which use single brackets). Here is a simple, sequential matrix multiplication written in HPJava:

```
public static void matmul
    (double [[*,*]] c,
    double [[*,*]] a,
    double [[*,*]] b) {
  int M = c.rng(0).size() ;
  int N =  c.rng(1).size() ;
  int L = a.rng(1).size() ;
  for(int i = 0 ; i < M ;
    i++)
   for(int j = 0 ; j < N ;
     j++) {
    c [i, j] = 0 ;
    for(int k = 0 ; k < L ;
       k++)
     c [i, j] += a [i, k] *
       b [k, j] ;
    }
}
```

The number of asterisks in the type signature defines the array's rank. Note that double brackets are used here only in type signatures. Once you declare `a`, `b`, and `c` to have multiarray type, you can use single brackets to subscript them. Double brackets also make an appearance in multiarray *creation* expressions, which look like:

```
double [[*,*]] a =
      new double [[N, M]] ;
```

A useful feature of multiarrays is that

like Fortran you can form *regular sections*. For example,

```
matmul(c [[i : i + B - 1,
        j : j + B - 1]],
    a, b) ;
```

assigns the matrix product of `a` and `b` to a `B` by `B` block of elements of `c`, starting at position `i`, `j`. A multiarray section is a first-class expression in the language—it can appear anywhere any other multiarray-valued expression can appear (but not on the left-hand side of an assignment). (Microsoft's C# supports multidimensional arrays, but not sections.)

This is as far as we go with array syntax. We don't provide Fortran 90-like elemental operations on whole arrays. If you want to copy one array or section to another, you must use a utility method `HPutil.copy()`. For more complicated things like elemental arithmetic or transformational operations on whole arrays, you'll need explicit loops or other library calls. Arguably the more esoteric forms of array syntax introduced in Fortran 90 only really helped compilers improve performance on specialized architectures. In HPJava we tried to provide just enough syntax to make library calls convenient.

Parallel Programming

The sequential multiarray syntax was a spin-off from our original goal to get *distributed arrays* into the language. In HPJava, a distributed array is a special kind of a multiarray whose elements are distributed over a group of cooperating processes. The type signatures are similar to ordinary multiarrays, but if the index range of a particular dimension is to be shared across processors, the corresponding position in the signature gets a hyphen instead of an asterisk. In a *distributed* array creation expression, a *distributed range object* replaces the integer extent in the creation expression. Distributed range objects play a role similar to *templates* in HPF. Before we can create one, we must define the *process grid* (for HPF aficionados, this is equivalent to the *processor arrangement*). So, you could create a three-dimensional distributed array with two distributed dimensions and one sequential dimension like this:

```
Procs p = new Procs2(P, Q) ;
Range x = new BlockRange(M,
          p.dim(0)) ;
Range y = new CyclicRange(N,
          p.dim(1));
double [[-,-,*]] a = new
double [[x, y, L]] on p ;
```

This is a lot of code, but it packs a lot of information. Normally, a program reuses a given process grid or range object to create many arrays, so you don't have to write all this code every time you create an array. (The first three lines might go in a "setup" section.) In our example, the processor arrangement, `p`, is a `P`-by-`Q` grid; the first dimension of `a` has extent `M` and is distributed block-wise over the first dimension of the grid; the second dimension has extent `N` and is distributed cyclically; and the third dimension is sequential, with extent `L`.

A mixture of compile-time and runtime checks imposes some stringent limits on how distributed arrays are subscripted. It is a discipline of our HPspmd programming model that if an operation is part of the built-in language syntax, its implementation should never require communication with other processes. You would violate this constraint if you could freely subscript distributed array elements. An example of a legal access pattern looks like this:

```
Adlib.writeHalo(b) ;
overall(i = x for :)
  overall(j = y for :)
    c [i, j] = 0.25 *
             (b [i-1, j] +
             b [i+1, j]) ;
```

This code assumes that distributed arrays `c` and `b` are aligned—they share the distributed ranges `x` and `y`. The overall construct is a distributed parallel loop—the iteration also runs over a specific distributed range. For the shifted indexes, `i-1` and `i+1` (in this example) to be legal, you must create the range `y` with ghost extensions (by using another specialized constructor for the object); the communication library method `writeHalo()` updates the ghost regions of array `b`. If you need a more irregular pattern of access, you must explicitly use a different communication method.

The parallel processing features of HPJava are relatively specialized—a programmer must understand how array elements and computations are distributed—in some respects more like MPI than HPF. But our experience has been that once the language is mastered, programmers can write many parallel algorithms compactly and efficiently. Arguably, the HPJava syntax is only making explicit programming strategies that a skilled HPF programmer—writing an efficient parallel program—would be using implicitly anyway.

Practical Examples

The HPJava translator itself and the code it generates are pure Java. The communication libraries—a distributed array collective communication library called Adlib and a Java binding of MPI called mpiJava (both available at www.hpjava.org)—were implemented using a JNI interface to native MPI, so they can be ported to most platforms where an MPI implementation is available. For MPI-based operation, the best-tested platforms currently are Linux using MPICH, Solaris using SunHPC, and the IBM SP series. You also can run Adlib-based programs on shared-memory computers without going through MPI: in this mode, the whole operation is "pure Java" and, thus, is platform independent.

The HPJava system is still new, and we don't have very many running applications. Figures 1 and 2 present some early benchmark results for simple parallel algorithms running on the SP3 installation at Florida State University, a 42-node supercomputing system; each node has four 375-MHz Power3 processors and 2 Gbytes of shared memory. The comparison is with the native HPF compiler for that platform. Considering that the HPJava translator is using a simplistic, nonoptimized translation scheme (the HPF codes were compiled at a high level of optimization), HPJava competes surprisingly well, at least on large, regular problems. It would be naïve to expect this level of performance in every case, but the results are encouraging.

Figure 3 is a screen capture from a demo you can play with on our Web page (www.hpjava.org/demo.html). This computational fluid dynamics program is more complicated than the Laplace or diffusion examples. The program actually isn't running in parallel: it runs in four independent applets in your browser, exploiting the shared-memory model for Adlib communication. This configuration is strictly for fun, but we could run the numerical kernel without modification on a true parallel computer (for example, a Java Swing version of the whole interactive program could be run on an SP3 if interactive nodes were available). The program's source code also is available on our Web page, and is an interesting example of what a larger program written in HPJava looks like.

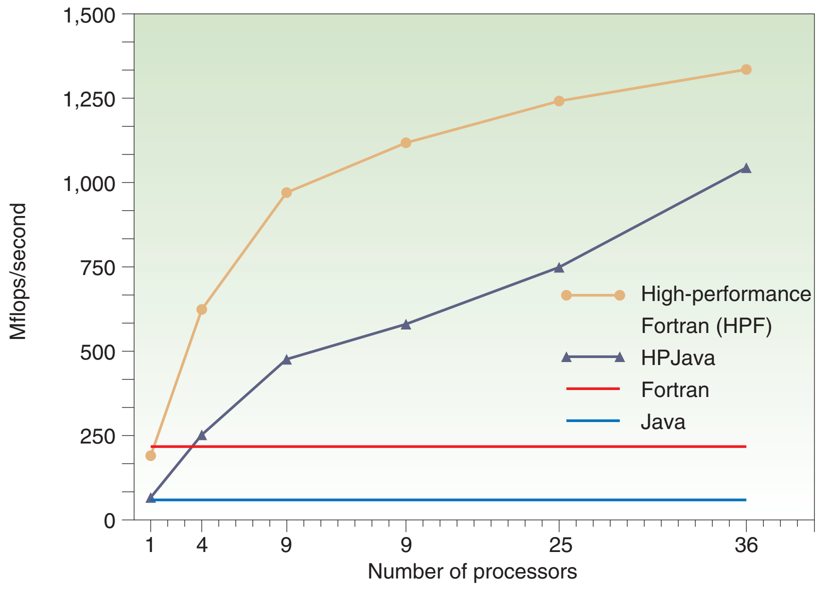

Figure 1. Performance on Florida State University's SP3 of a parallel HPJava program implementing red–black relaxation on a 512 × 512 Laplace equation. Execution is on up to 36 processors. Comparison is with IBM HPF, and also with standard (sequential) Fortran and Java.

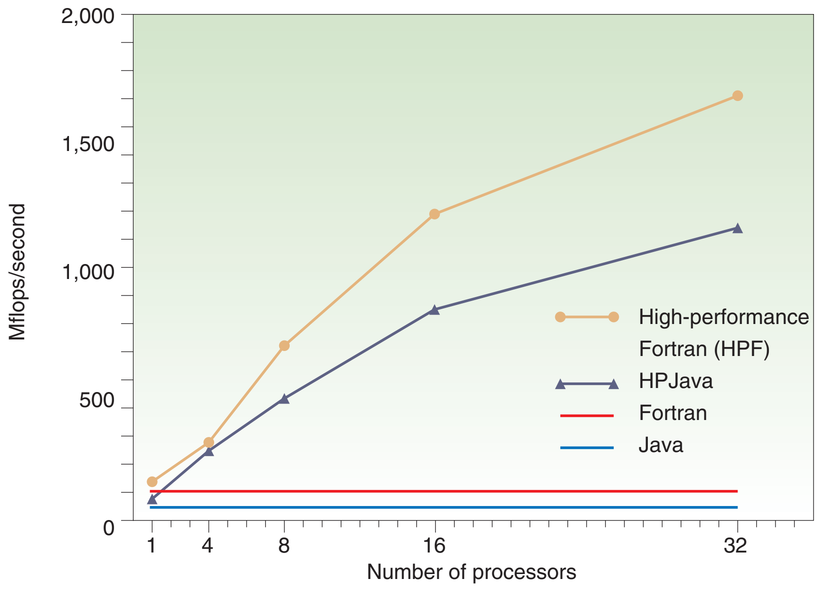

Figure 2. Performance on an SP3 of a parallel HPJava program solving the three-dimensional diffusion equation on a 128 × 128 × 128 grid. There can be up to 32 processors.

We implemented the current HPJava compiler as a preprocessor from the extended language to intermediate Java source

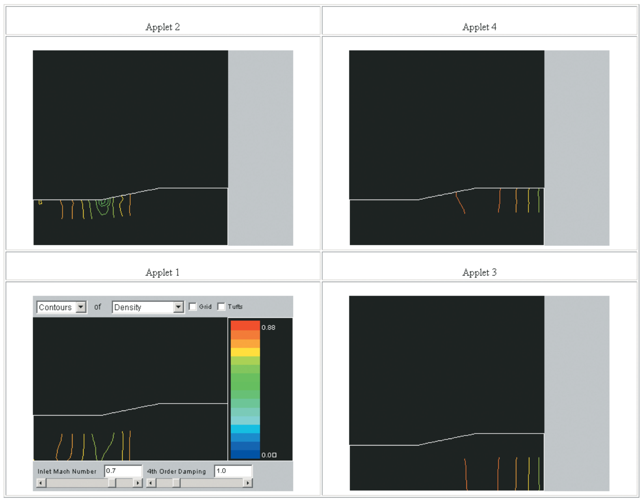

Figure 3. Simulation of a parallel program describing inviscid flow through an axisymmetric nozzle. This applet cluster lets you display contour and line plots of all flow variables. You can view the demo at www.hpjava.org/demo.html.

code, which is compiled by a standard Java compiler. This implementation is largely transparent to users, who run an `hpjavac` command and get class files as output. We put a lot of effort into preserving this transparency: the preprocessor includes a complete front end for Java, and we've checked most of the Java Language Specification requirements. So, it should be unusual to get messages from the back-end Java compiler.

The current translator (which is not doing global optimizations) also preserves line numbers throughout the preprocessor, so line numbers in runtime exception messages point back to the original source. This means that debugging is no harder than it should be for a pure compiler. The generated code incorporates various runtime checks associated with the constraints of the HPspmd programming model. Future work is likely to concentrate on optimization of the generated code and the development of additional libraries and applications.

We expect HPJava will be most helpful for problems that have some degree of regularity. To a first approximation, the HPJava's domain is similar to HPF's. Many of the most challenging problems in modern computational science have a very irregular structure, so the value of our language features in those domains is more controversial. Nevertheless, we believe HPJava has potential as a practical tool in many important areas. At the very least, it should be a good classroom tool for teaching parallel programming. CiSE

Reference

1. G. Fox, "Java and Grande Applications," *Computing in Science & Engineering*, vol. 5, no. 1, 2003, pp. 60–62.

Bryan Carpenter is a research scientist in the Pervasive Technology Labs at Indiana University. He has a PhD in physics from London University. He's worked in the field of parallel computing since 1985, initially on physics applications, but in recent years on more general library and language support for parallel computing, especially Java-based. Contact him at dbcarpen@indiana.edu.

Geoffrey Fox is director of the Community Grids Lab at Indiana University. He has a PhD in theoretical physics from Cambridge University. Contact him at gcf@indiana.edu; www.communitygrids.iu.edu/IC2.html.

Feedback?

Geoffrey Fox wants to hear from you. Comments? Applause? Criticism? Suggestions? What topic shall we tackle next? We want to hear about interesting topics and potential authors that could benefit Web Computing readers.