HPJava: Java for Data Parallel Programming

Bryan Carpenter and Geoffrey Fox

Today we see a healthy diversity of projects aiming to harness Java for scientific (or "Grande") computing. A number of these approaches were reviewed in a recent article in this series. In this article we will focus instead on one particular project of this kind, ongoing at our labs in Indiana University.

The HPJava project aims to support scientific and parallel computing in a modern object-oriented, Internet-friendly environment—namely the Java platform. To a large extent we do this by leveraging language and library features that have been popularly associated with Fortran. So HPJava imports features like "scientific" multidimensional array syntax, and distributed arrays similar to those in High Performance Fortran. At a more language-independent level, HPJava also introduces a slightly unusual parallel programming model, somewhere in between the classical HPF and MPI extremes.

History

Many readers will be aware of the work of the High Performance Fortran Forum, which was very active in the early 1990s. The forum brought together leading practitioners in the field of high performance computing to define a common language for data parallel computing. Inspired especially by the success of parallel dialects Fortran like *Connection Machine Fortran*, the resulting language definition—High Performance Fortran, or HPF—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 was also supposed to compile efficiently to the SPMD (Single Program Multiple Data) style of execution required by a newer generation of more loosely coupled MIMD 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 standard, or MPI (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. It grew especially out of earlier work in our group on HPF. In a collaborative project with researchers from Peking University and other institutes we had developed run-time libraries to support compilation of HPF. 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 been hoping for. It was difficult to write the compilers, and also the underlying program model was more restrictive than some people liked. In the absence of HPF there was the option of making our libraries available for explicit call from multi-

processor 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 was going to be either ugly or inefficient. Adding language extensions, even at the level of "syntactic sugar", seemed like a daunting prospect if C++ with its notoriously complex structure was the base language.

It looked as if Java might provide a better solution. People were starting to talk about Java Grande, and the possibility that eventually Java could be a high performance language. The language was sufficiently similar to C++ that we should be able to 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. By the time this column appears in print (March [?], 2003), the first release of the software should be available for free download from our Web site, www.hpjava.org.

HPJava Philosophy

HPJava lifts some ideas wholesale from High Performance Fortran, and it's domain of applicability is likely to overlap 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). One can write HPJava programs that look very much like the corresponding HPF program. 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 the requirement to use these libraries conveniently. 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 program. An HPJava program is supposed to be viewed like a SPMD 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 responsibility of the programmer, similar to MPI. But since, by design, we support convenient calls to high-level libraries acting on distributed arrays, this task is typically easier than it would be in an MPI program.

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

In this sense, although we may 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 that 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.

Multiarrays

At a sort of "entry-level", users of HPJava may simply want to take advantage of the syntax the extended language provides for Fortran-like, multi-dimensional arrays, perhaps initially ignoring the parallel aspects of the language. In this area the syntax is compatible with recent proposals from the Java Grande Numerics Working Group. Following that group, we call the new arrays *multiarrays*. HPJava has both multiarrays and ordinary Java arrays available, and the programmer chooses which is most convenient according to context. A multiarray can have elements of any standard Java type and it can have any *rank* (dimensionality). Multiarrays have a rectangular aspect, unlike standard Java multidimensional arrays, which are implemented as arrays of arrays, and can be "ragged". In general this makes it hard for a compiler to analyze standard Java multidimensional arrays, and optimize their use.

Our syntax for multiarrays uses double brackets to distinguish type signatures from standard Java arrays (which of course 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 rank of the array. Note double brackets are only used here in type signatures. Once **a**, **b** and **c** are declared to have multiarray type we 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 one can form *regular sections*, as in Fortran. For example this code:

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#, by the way, also supports multidimensional arrays, but doesn't allow 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 use a utility method **HPutil.copy()**. For more complicated things explicit loops, or other library calls, are needed. We see limited demand for the more esoteric forms of array syntax introduced in Fortran 90, and arguably these only really helped compilers to get good performance on specialized architectures. So in HPJava we provided just enough syntax to make library calls convenient.

Parallel Programming

The sequential multiarray syntax was really a spin-off from our original goal, which was 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. When a distributed array is created, 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 a three-dimensional distributed array with two distributed dimensions and one sequential dimension could be created 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 given process grid or range object is reused to create many arrays in the program, so you don't have to write all this code every time an array is created—the first three lines might go in some "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; the third dimension is sequential, with extent **L**.

The language imposes some stringent limits on how distributed arrays can be subscripted. It is a discipline of our HPspmd programming model that if an operation is part of the built-in syntax of the language its implementation should never require communication with other processes. This constraint would certainly be violated if one could freely subscript elements of distributed arrays. The restrictions are imposed by a

mixture of compile-time and run-time checks. An example of the kind of access pattern that *is* legal would be something 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 assumes the distributed arrays **c** and **b** are aligned—they share the distributed ranges **x** and **y**. The overall construct is a distributed parallel loop: note the iteration also runs over a specific distributed range. For the shifted indexes **i-1**, **i+1** in this example to be legal, the range **y** must be created with ghost extensions (by using another specialized constructor for the object); the communication library method **writeHalo()** updates the ghost regions of the 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—the programmer has to be very aware of 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 many parallel algorithms can be written compactly and efficiently. And arguably the HPJava syntax is just making explicit programming strategies that a skilled HPF programmer, writing an efficient parallel program, would be using implicitly anyway.

Examples

The HPJava translator itself and the code it generates are pure Java. The communication libraries available today (a distributed array collective communication library called Adlib and a Java binding of MPI called mpiJava) have been initially 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. It is also possible to run Adlib-based programs on shared memory computers without going through MPI: in this mode the whole operation is "pure Java" and therefore platform independent.

The HPJava system is still new, and as yet 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. The comparison is with the native HPF compiler for that platform. Considering that the HPJava translator is using a simplistic translation scheme with no optimization (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. This is a less trivial CFD program. The program on the Web page isn't truly running in parallel: it runs in 4 independent applets in your browser, exploiting the shared memory model for Adlib communication. This configuration is strictly for fun, but the numerical kernel can

be run without modification on a true parallel computer (in fact a Swing version of the whole interactive program could be run on, say, an SP3 if interactive nodes were available). The source of the program, also available on the Web page, is an interesting example of what a larger program written in HPJava look like.

Prospects

The current HPJava compiler is implemented as a preprocessor from the extended language to intermediate Java source code, which is compiled by a standard Java compiler. This implementation is largely transparent to the user, who runs an *hpjavac* command and gets class files as output. We put a lot of effort into preserving this transparency: the preprocessor includes a complete front end for Java with essentially all the requirements of the Java Language Specification checked at this stage. So it should be unusual to get messages from the backend Java compiler. The current translator (which is not doing global optimizations) also preserves line numbers all through the preprocessor, so line numbers in run-time exception messages point back to the original source: this means 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 that HPJava will be most helpful for problems that have some degree of regularity. To a first approximation the application domain is similar to HPF. It is certainly true that many of the most challenging problems in modern computational science have a very irregular structure, and 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.

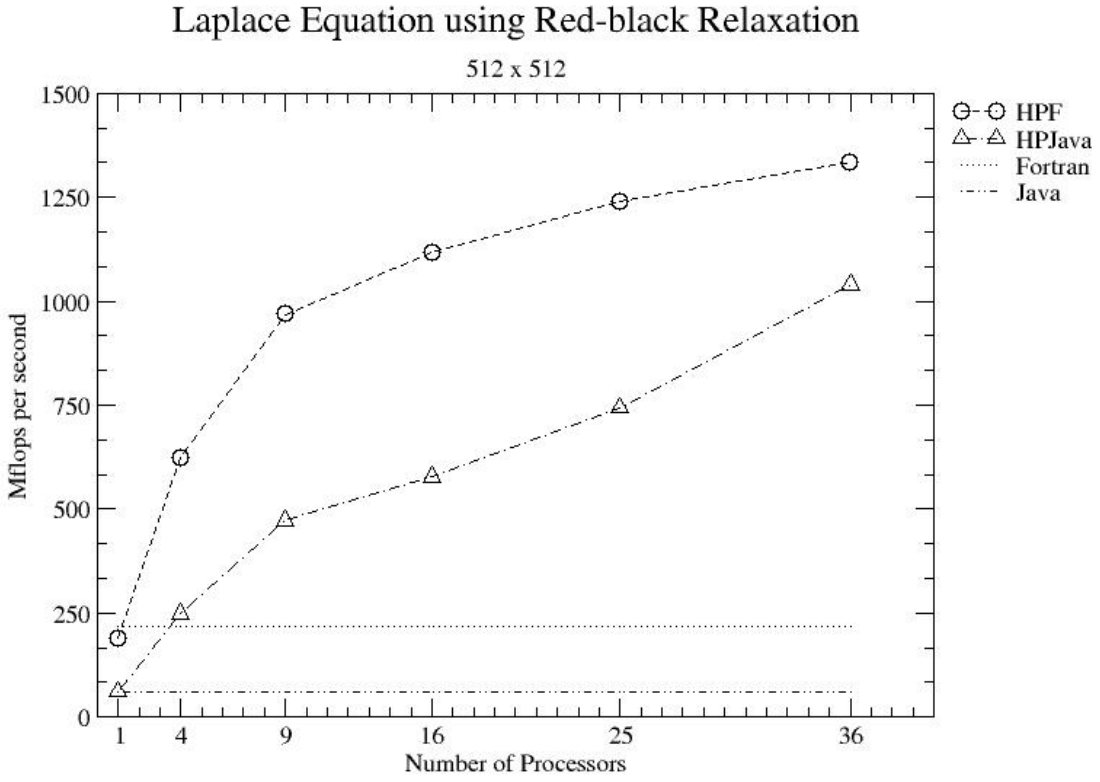

Figure 1: Performance on SP3 for red-black relaxation on the Laplace Equation.

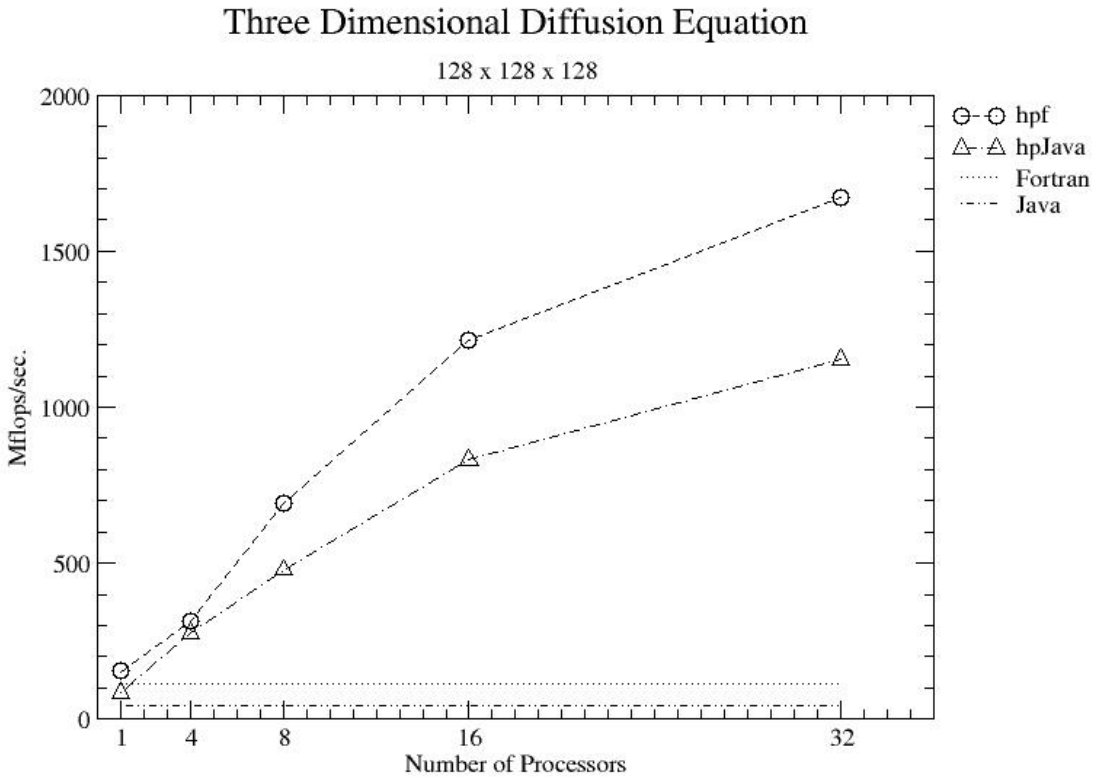

Figure 2: Performance on the SP3: 3-dimensional diffusion equation.

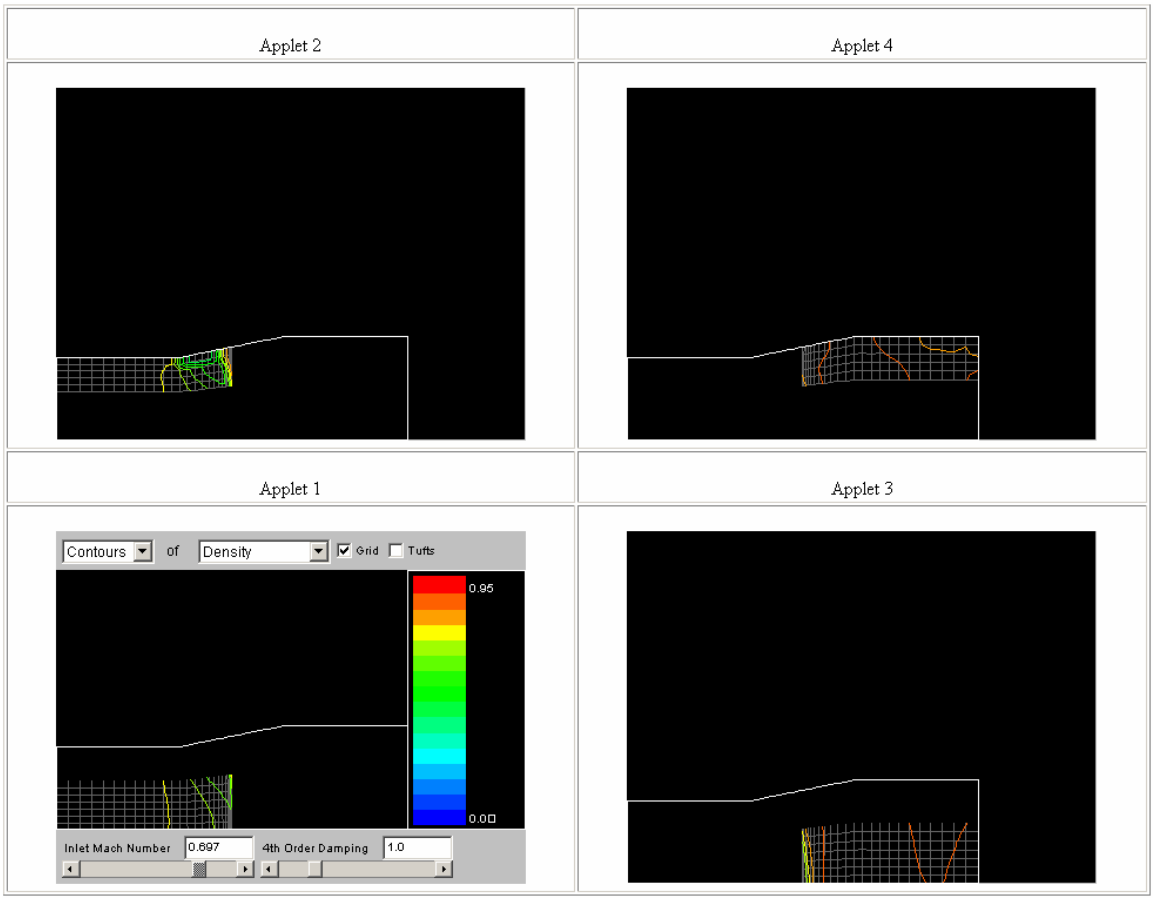

Figure 3. An interactive CFD demo.