Applications of HPJava

Bryan Carpenter, Geoffrey Fox,
Han-Ku Lee and Sang Boem Lim

Pervasive Technology Labs at Indiana University
Bloomington, IN 47404-3730
{dbcarpen,gcf,hanklee,slim}@indiana.edu

Abstract. We describe two applications of our *HPJava* language for parallel computing. The first is a multigrid solver for a Poisson equation, and the second is a CFD application that solves the Euler equations for inviscid flow. We illustrate how the features of the HPJava language allow these algorithms to be expressed in a straightforward and convenient way. Performance results on an IBM SP3 are presented.

1 Introduction

The HPJava project [10] has developed translator and libraries for a version of the Java language extended to support parallel and scientific computing. Version 1.0 of the HPJava software was released earlier this year as open source software. This paper reports experiences using HPJava for applications, with some benchmark results. A particular goal here is to argue the case that our programming model is flexible and convenient for writing non-trivial scientific applications.

HPJava extends the standard Java language with support for "scientific" multidimensional arrays (multiarrays), and support for *distributed arrays*, familiar from High Performance Fortran (HPF) and related languages. Considerable work has been done on adding features like these to Java and C++ through class libraries (see for example [17], [8], [15]). This seems like a natural approach in an object oriented language, but the approach has some limits: most obviously the syntax tends to be inconvenient. Lately there has been widening interest in adding extra syntax to Java for multiarrays, often through preprocessors[1].

From a parallel computing point view of an interesting feature of HPJava is its spartan programming model. Although HPJava introduces special syntax for HPF-like distributed arrays, the language deliberately minimizes compiler intervention in *manipulating* distributed data structures. In HPF and similar languages, elements of distributed arrays can be accessed on essentially the same footing as elements of ordinary (sequential) arrays—if the element being accessed resides on a different processor, some run-time system is probably invoked transparently to "get" or "put" the remote element. HPJava does not have this feature. It was designed as a framework for development of *explicit* libraries operating on distributed data. In this mindset, the right way of accessing

[1] See, for example, the minutes of recent meetings at [12].

remote data is to explicitly invoke a communication *library* method to get or put the data.

So HPJava provides some special syntax for accessing locally held elements of multiarrays and distributed arrays, but stops short of adding special syntax for accessing non-local elements. Non-local elements can only be accessed by making explicit library calls. The language attempts to capture the successful library-based approaches to SPMD parallel computing—it is in very much in the spirit of MPI, with its explicit point-to-point and collective communications. HPJava raises the level of abstraction a notch, and adds excellent support for development of libraries that manipulate distributed arrays. But it still exposes a multi-threaded, non-shared-memory, execution model to to programmer. Advantages of this approach include flexibility for the programmer, and ease of compilation, because the compiler does not have to analyse and optimize communication patterns.

The basic features of HPJava have been described in several earlier publications. In this paper we will jump straight into a discussion of the implementation of some representative applications in HPJava. After briefly reviewing the compilation strategy in section 2, we illustrate typical patterns of HPJava programming through a multigrid algorithm in section 3. This section also serves to review basic features of the langauge. Section 4 describes another substantial HPJava application—a CFD code—and highlights additional common coding patterns. Section 5 collects together benchmark results from these applications.

1.1 Related Work

Other ongoing projects that extend the Java language to directly support scientific parallel computation include Titanium [3] from UC Berkeley, Timber/Spar [2] from Delft University of Technology, and Jade [6] from University of Illinois at Urbana-Champaign.

Titanium adds a comprehensive set of parallel extensions to the Java language. For example it includes support for a shared address space, and does compile-time analysis of patterns of synchronization. This contrasts with our HPJava, which only adds new data types that can be implemented "locally", and leaves all interprocess communication issues to the programmer and libraries.

The Timber project extends Java with the Spar primitives for scientific programming, which include multidimensional arrays and tuples. It also adds task parallel constructs like a *foreach* construct.

Jade focuses on message-driven parallelism extracted from interactions between a special kind of distributed object called a *Chare*. It introduces a kind of parallel array called a *ChareArray*. Jade also supports code migration.

HPJava differs from these projects in emphasizing a lower-level (MPI-like) approach to parallelism and communication, and by importing HPF-like distribution formats for arrays. Another significant difference between HPJava and the other systems mentioned above is that HPJava translates to Java byte codes, relying on clusters of conventional JVMs for execution. The systems mentioned above typically translate to C or C++. While HPJava may pay some price in

performance for this approach, it tends to be more fully compliant with the standard Java platform (e.g. it allows local use of Java threads, and APIs that require Java threads).

2 Features of the HPJava System

HPJava adds to Java a concept of multi-dimensional arrays called "multiarrays" (consistent with proposals of the Java Grande Forum). To support parallel programming, these multiarrays are extended to "distributed arrays", very closely modelled on the arrays of High Performance Fortran. The new distributed data structures are cleanly integrated into the syntax of the language (in a way that doesn't interfere with the existing syntax and semantics of Java—for example ordinary Java arrays are left unaffected).

In the current implementation, the source HPJava program is translated to an intermediate standard Java file. The preprocessor that performs this task is reasonably sophisticated. For example it performs a complete static semantic check of the source program, following rules that include all the static rules of the Java Language Specification [9]. So it shouldn't normally happen that a program accepted by the HPJava preprocessor would be rejected by the backend Java compiler. The translation scheme depends on type information, so we were essentially forced to do a complete type analysis for HPJava (which is a superset of standard Java). Moreover we wanted to produce a practical tool, and we felt users would not accept a simpler preprocessor that did not do full checking.

The current version of the preprocessor also works hard to preserve line-numbering in the conversion from HPJava to Java. This means that the line numbers in run-time exception messages accurately refer back to the HPJava source. Clearly this is very important for easy debugging.

A translated and compiled HPJava program is a standard Java class file, ready for execution on a distributed collection of JIT-enabled Java Virtual Machines. All externally visible attributes of an HPJava class—e.g. existence of distributed-array-valued fields or method arguments—can be transparently reconstructed from Java signatures stored in the class file. This makes it possible to build libraries operating on distributed arrays, while maintaining the usual portability and compatibility features of Java. The libraries themselves can be implemented in HPJava, or in standard Java, or as JNI interfaces to other languages. The HPJava language specification documents the mapping between distributed arrays and the standard-Java components they translate to.

Currently HPJava is supplied with one library for parallel computing—a Java version of the *Adlib* library of collective operations on distributed arrays [18]. A version of the *mpiJava* [1] binding of MPI can also be called directly from HPJava programs. Of course we would hope to see other libraries made available in the future.

3 A Multigrid Application

The multigrid method [5] is a fast algorithm for solution of linear and nonlinear problems. It uses a hierarchy or stack of grids of different granularity (typically with a geometric progression of grid-spacings, increasing by a factor of two up from finest to coarsest grid). Applied to a basic relaxation method, for example, multigrid hugely accelerates elimination of the residual by restricting a smoothed version of the error term to a coarser grid, computing a correction term on the coarse grid, then interpolating this term back to the original fine grid. Because computation of the correction term on the fine grid can itself be handled as a relaxation problem, the strategy can be applied recursively all the way up the stack of grids.

In our example, we apply the multigrid scheme to solution of the two-dimensional Poisson equation. For the basic, unaccelerated, solution scheme we use red-black relaxation. An HPJava method for red-black relaxation is given in Figure 1. This looks something like an HPF program with different syntax. One obvious difference is that the base language is Java instead of Fortran. The HPJava type signature `double [[-,-]]` means a two dimensional distributed array of `double` numbers[2]. So the arguments passed to the method `relax()` will be distributed arrays

The inquiry `rng()` on the distributed array `f` returns the `Range` objects `x`, `y`. These describe the distribution format of the array index (for the two dimensions).

The HPJava `overall` construct operates like a `forall` construct, with one important difference. In the HPJava construct one *must* specify how the iteration space of the parallel loop is distributed over processors. This is done by specifying a `Range` object in the header of the construct.

The variables `i`, `j` in the figure are called *distributed index* symbols. Distributed indexes are scoped by the overall constructs that use them. They are *not* integer variables, and there is no syntax to declare a distributed index *except* through an overall construct (or an *at* construct—see later). The usual Java scoping rules for local variables apply: one can't for example use `i` as the index of an overall if there is already a local variable `i` in scope—the compiler doesn't allow it.

An unusual feature of the HPJava programming model is that the subscripts in a distributed array element reference usually *must* be distributed index symbols. And these symbols must be distributed with the essentially *same* format as the arrays they subscript. As a special case, *shifted* index expressions like `i+1` are allowed as subscripts, but only if the distributed array was created with *ghost regions*. Information on ghost regions, along with other information about

[2] The main technical reason for using double brackets here is that it is useful to support an idea of *rank-zero distributed arrays*: these are "distributed scalars", which have a localization (a *distribution group*) but no index space. If we used single brackets for distributed array type signatures, then `double []` could be ambiguously interpretted as either a rank-zero distributed array or an ordinary Java array of `double`s.

```
static void relax(int itmax, int np,
                  double[[-,-]] u, double[[-,-]] f) {

  Range x = f.rng(0), y = f.rng(1);

  for(int it = 1; it <= itmax * 2; it++) {

    Adlib.writeHalo(u);

    overall(i = x for 1 : np - 2)
      overall(j = y for 1 + (i‘ + it) % 2 : np - 2 : 2) {

        u [i, j] = 0.25 * (f [i, j] +
                   u [i - 1, j] + u [i + 1, j] +
                   u [i, j - 1] + u [i, j + 1]);
     }
  }
}
```

Fig. 1. Red black relaxation on array `u`.

distribution format, is captured in the `Range` object associated with the array dimension or index.

These requirements ensure that a subscripting operation in an overall construct only accesses locally held elements. They place very stringent limitations on what kind of expression can appear as a subscript of a distributed array. We justify this by noting that this restricted kind of data parallel loop is a frequently recurring pattern in SPMD programs in general, and it is convenient to have it captured in syntax. A glance at the full source of the applications described in this paper should make this claim more plausible[3].

The method `Adlib.writeHalo()` is a communication method (from the library called Adlib). It performs the edge-exchange to fill in the ghost regions. As emphasized earlier, the compiler is not responsible for inserting communications—this is the programmer's responsibility. We assume this should be acceptable to programmers currently accustomed to using MPI and similar libraries for communication.

Because of the special role of distributed index symbols in array subscripts, it is best not to think of the expressions `i`, `j`, `i+1`, etc, as having a numeric value: instead they are treated as a special kind of thing in the language. We use

[3] When less regular patterns of access are necessary, the approach depends on the locality of access: if accesses are irregular but local one can extract the locally-held blocks of the distributed array by suitable inquiries, and operate on the blocks as in an ordinary SPMD program; if the accesses are non-local one must use suitable library methods for doing irregular remote accesses.

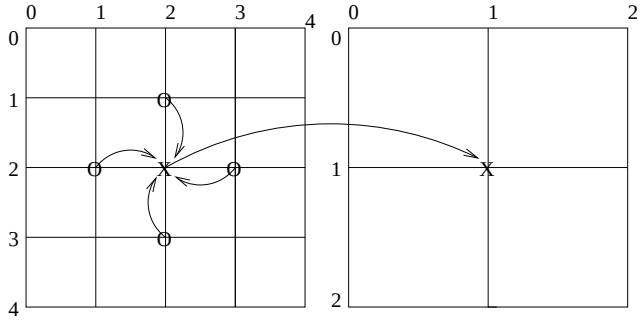

Fig. 2. Illustration of restrict operation

the notation `i‘` to extract the numeric global index associated with `i`, say[4]. In particular, use of this expression in the modulo 2 expression in the inner overall construct in Figure 1 implements the red-black pattern of accesses.

This completes the description of most "non-obvious" features of HPJava syntax. Remaining examples in the paper either recycle these basic ideas, or just introduce new library routines; or they import relatively uncontroversial syntax, like a syntax for array sections.

Figure 2 visualizes the "restrict" operation that is used to transfer the error term from a fine grid to a coarse grid. The HPJava code is given in Figure 3. The restrict operation here computes the residual term at sites of the fine grid with *even* coordinate values, then sends these values to the coarse grid. In multigrid the restricted residual from the fine grid becomes the RHS of a new equation on the coarse grid. The implementation uses a temporary array `tf` which should be aligned with the fine grid (only a subset of elements of this array are actually used). The last line introduces two new features: distributed array *sections*, and the library function `Adlib.remap()`. Sections work in HPJava in much the same way as in Fortran—one small syntactic difference is that they use double brackets. The bounds in the `fc` section ensure that edge values, corresponding to boundary conditions, are not modified. The stride in the `tf` section ensures only values with even subscripts are selected. The `Adlib.remap()` operation is needed because in general there is no simple relation between the distribution format of the fine and coarse grid—this function introduces the communications necessary to perform an assignment between any two distributed arrays with unrelated distribution format. As another example, the interpolation code of Figure 4 performs the complementary transformation from the coarse grid to the fine grid.

The basic pattern here depends only on the geometry of the problem. More complex (perhaps non-linear) equations with similar geometry could be tackled

[4] Early versions of the language used a more conventional "pseudo-function" syntax rather than the "primed" notation. The current syntax arguably makes expressions more readable, and emphasizes the unique status of the distributed index in the language.

```
static void restr(int npc, int npf,
                  double fc [[-,-]], double uf [[-,-]],
                  double ff [[-,-]], double tf [[-,-]]) {

  Range xf = ff.rng(0), ff = ff.rng(1);

  int nc = npc - 1, nf = npf - 1;

  Adlib.writeHalo(uf);

  overall(i = xf for 2 : nf - 2 : 2)
    overall(j = yf for 2 : nf - 2 : 2)
      tf [i, j] += 2.0 *
                   (ff [i, j] - 4.0 * uf [i, j] +
                    uf [i - 1, j] + uf [i + 1, j] +
                    uf [i, j - 1] + uf [i, j + 1]);

  Adlib.remap(fc [[1 : nc - 1, 1 : nc - 1]],
              tf [[2 : nf - 2 : 2, 2 : nf - 2 : 2]]);
}
```

Fig. 3. HPJava code for restrict operation.

by similar code. Problems with more dimensions can also be programmed in a similar way.

4 A CFD Application

In this section we discuss another significant HPJava application code. This code solves the Euler equations for inviscid fluid flow by a finite volume approach. One version of this code, viewable at `http://www.hpjava.org/demo.html` also has a novel parallel GUI implemented in HPJava[5].

The Euler equations are a family of conservation equations, relating the time rates of change of various densities to divergences of associated flow fields. In two dimensions there are four densities—the ordinary matter density, densities of the two components of momentum, and the energy density. The Euler equations can be summarized as a conservation equation for four-component vectors U, f and g:

$$\frac{\partial U}{\partial t} + \frac{\partial f}{\partial x} + \frac{\partial g}{\partial y} = 0 \tag{1}$$

The flow variables (f, g) are related to the dependent variables U by simple (but non-linear) algebraic equations. So the set of differential equations is closed.

[5] The code is adapted from a version of an original Java code by David Oh of MIT [16], modified by Saleh Elmohamed and Mike McMahon of Syracuse University. It is almost identical to the CFD benchmark in the Java Grande Benchmark suite, which came from the same original source.

```
static void interp(int npf, double[[-,-]] uc,
                   double[[-,-]] uf, double [[-,-]] tf) {

  Range xf = uf.rng(0), yf = uf.rng(1);

  int nf = npf - 1;

  Adlib.remap(tf [[0 : nf : 2, 0 : nf : 2]], uc);

  Adlib.writeHalo(tf);

  overall(i = xf for 1 : nf - 1 : 2)
    overall(j = yf for 2 : nf - 2 : 2)
      uf [i, j] += 0.5 * (tf [i - 1, j] + tf [i + 1, j]);

  overall(i = xf for 2 : nf - 2 : 2)
    overall(j = yf for 1 : nf - 1 : 2)
      uf [i, j] += 0.5 * (tf [i, j - 1] + tf [i, j + 1]);
}
```

Fig. 4. HPJava code for interpolate operation.

Two important quantities that figure in the equations are the pressure, p, and the enthalpy per unit mass, H, which can be computed from the components of U using the equations of state for the fluid.

4.1 Discretization and numerical integration

The system of partial differential is discretized by a finite volume approach—see for example [7] or [11]. Space is divided into a series of quadrilateral (but not necessarily rectangular) cells labelled (i, j). This reduces the PDEs to a large coupled system of ordinary differential equations. These are integrated by a variant of the well-known 4th order Runge Kutta scheme. A single time-step involves four stages like:

$$U'_{i,j} = U_{i,j} - \alpha \frac{\delta t}{\Omega_{i,j}} R_{i,j}(U) \tag{2}$$

where α is a fractional value characteristic of the scheme, and

$$R_{i,j}(U) = \sum_{\text{faces of cell}} (f\,\delta y - g\,\delta x) \tag{3}$$

Here $\Omega_{i,j}$ is the volume a cell and δx, δy are coordinate differences between end-points of the face. Since the dependent variables and fluxes are defined at cell centers, their values at a cell face in equation 3 is are approximated as the average of the values from the two cells meeting at the face.

So at its most basic level the program for integrating the Euler equations consists of a series of steps like:

1. Calculate p, H from current U (via equations of state).
2. Calculate f from U, p, H.
3. Calculate g from U, p, H.
4. Calculate R from f, g.
5. Update U.

To parallelize in HPJava, the discretized field variables are naturally stored in distributed arrays. All the steps above become *overall* nests. As a relatively simple case, the operation to calculate f (step 2) looks like:

```
Statevector [[-,-]] U, f, ... ;
double [[-,-]] p, H, ... ;
...
overall(i = x for 0 : imax)
    overall(j = x for 0 : imax) {

        double u = U [i, j].b / U [i, j].a ;  // velocity component

        f [i, j].a = U [i, j].b ;
        f [i, j].b = U [i, j].b * u  +  p [i, j] ;
        f [i, j].c = U [i, j].c * u ;
        f [i, j].d = H [i, j] * u ;
  }
```

The four fields `a`, `b`, `c`, `d` of `Statevector` correspond to the four conserved densities. A general observation is that the bodies of overall statements are now more complex than those in the (perhaps artificially simple) Poisson equation example of the previous section. We expect this will often happen in "real" applications. It is good for HPJava, because it means that various overheads associated with starting up a distributed loop are amortized better.

Another noteworthy thing is that these overall statements work naturally with aligned data—no communication is needed here. Out of the five stages enumerated above, only computation of `R` involves non-local terms (formally because of the use of averages across adjacent cells for the flow values at the faces). The code can be written easily using ghost regions, shifted indices, and the `writeHalo()` operation. Again it involves a single overall nest with a long body. A much-ellided outline is given in Figure 5. The optional arguments `wlo`, `whi` to `Adlib.writeHalo()` define the widths of the parts ghost regions that need updating (the default is to update the whole of the ghost regions of the array, whatever their width). In the current case these vectors both have value `[1, 1]`—because shifted indices displace one site in positive and negative x and y directions.

The arrays `xnode` and `ynode` hold coordinates of the cell vertices. Because these arrays are constant through the computation, the ghost regions of *these* arrays are initialized once during startup.

```
Adlib.writeHalo(f, wlo, whi) ;
Adlib.writeHalo(g, wlo, whi) ;

overall(i = x for 1 : imax - 1)
    overall(j = y for 1 : jmax - 1) {

        ... Set fields of r [i, j] to zero ...

        // East face
        hy = 0.5 * (ynode [i, j] - ynode [i, j - 1]) ;
        r [i, j].a += hy * (f [i, j].a + f [i + 1, j].a) ;
        r [i, j].b += hy * (f [i, j].b + f [i + 1, j].b) ;
        r [i, j].c += hy * (f [i, j].c + f [i + 1, j].c) ;
        r [i, j].d += hy * (f [i, j].d + f [i + 1, j].d) ;

        hx = 0.5 * (xnode [i, j] - xnode [i, j - 1]) ;
        r [i, j].a -= hx * (g [i, j].a + f [i + 1, j].a) ;
        r [i, j].b -= hx * (g [i, j].b + f [i + 1, j].b) ;
        r [i, j].c -= hx * (g [i, j].c + f [i + 1, j].c) ;
        r [i, j].d -= hx * (g [i, j].d + f [i + 1, j].d) ;

        ... Add similar contributions for S, W, N faces ...
    }
```

Fig. 5. Outline of computation of R.

We will briefly discuss two other interesting complications: handling of so-called *artificial viscosity*, and imposition of boundary conditions.

Artificial viscosity (or artifical smoothing) is added to damp out a numerical instability in the Runge Kutta time-stepping scheme, which otherwise causes unphysical oscillatory modes associated with the discretization to grow. An accepted scheme adds small terms proportional to 2nd and 4th order finite difference operators to the update of U. From the point of view of HPJava programming one interesting issue is that 4th order damping implies an update stencil requiring source elements offset two places from the destination element (unlike Figure 5, for example, where the maximum offset is one). This is handled by creating the U array with ghost regions *of width 2*.

Implementing numerically stable boundary conditions for the Euler equations is non-trivial. In our implementation the domain of cells is rectangular, though the grid is distorted into an irregular pipe profile by the choice of physical coordinates attached to grid points (`xnode`, `ynode` distributed arrays). HPJava has an additional control construct called *at*, which can be used to update edges (it has other uses). The *at* statement is a degenerate form of the overall statement. It only "enumerates" a single location in its specified range. To work along the

line $x = 0$, for example, one may write code like:

```
at(i = x [0])
    overall(j = y for 1 : jmax - 1) {
        ... assign U [i, j] in terms of U [i + 1, j], etc ...
    }
```

The actual code in the body is a fairly complicated interpolation based on Riemann invariants. In general access to `U [i+1,j]` here relies on ghost regions being up-to-date, exactly as for an index scoped by an overall statement.

5 Benchmark Results

For the two applications described above, we have sequential and parallel programs to compare performance. The sequential programs were written in Java and/or Fortran 95. The parallel programs, of course, were written in HPJava. For multigrid we also compare with an available HPF code (taken from [4]).

The experiments were performed on the SP3 installation at Florida State University. The system environment for SP3 runs were as follows:

- System: IBM SP3 supercomputing system with AIX 4.3.3 operating system and 42 nodes.
- CPU: A node has Four processors (Power3 375 MHz) and 2 gigabytes of shared memory.
- Network MPI Settings: Shared "css0" adapter with User Space(US) communication mode.
- Java VM: IBM 's JIT
- Java Compiler: IBM J2RE 1.3.1

For best performance, all sequential and parallel Fortran and Java codes were compiled using -O5 or -O3 with -qhot or -O (i.e. maximum optimization) flag.

5.1 Multigrid results

First we present some results for the the computational kernel of the multigrid code, namely unaccelerated red-black relaxation algorithm of Figure 1. Figure 6 gives our results for this kernel on a 512 by 512 matrix. The results are encouraging. The HPJava version scales well, and eventually comes quite close to the HPF code (absolute megaflop performances are modest, but this feature was observed for all our codes, and seems to be a property of the hardware)[6].

The flat lines at the bottom of the graph give the sequential Java and Fortran performances, for orientation. We did not use any auto parallelization feature here.

Corresponding results for the complete multigrid code are given in Figure 7. The results here are not as good as for simple red-black relaxation—both

[6] We do not know why the HPJava result on 25 processors appears to be below the general trend. However the result was repeatable.

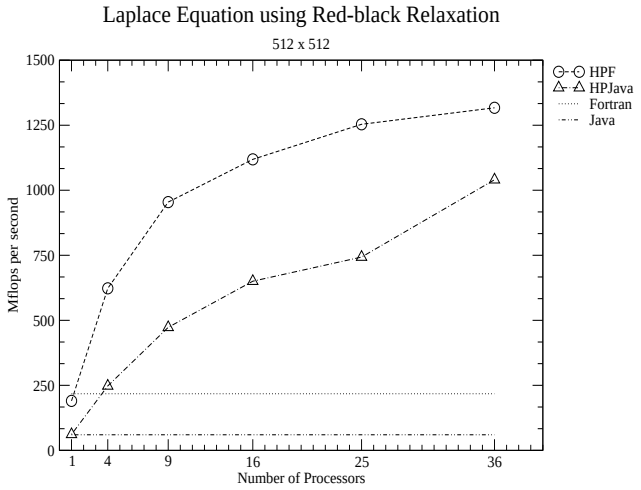

Fig. 6. Red-black relaxation of two dimensional Laplace equation with size of 512^2.

HPJava speed relative to HPF, and the parallel speedup of HPF and HPJava are less satisfactory.

The poor performance of HPJava relative to Fortran in this case can be attributed largely to the naive nature of the translation scheme used by the current HPJava system. The overheads are especially significant when there are many very tight overall constructs (with short bodies). We saw several of these in section 3. Experiments done elsewhere [13] lead us to believe these overheads can be reduced by straightforward optimization strategies which, however, are not yet incorporated in our source-to-source translator[7].

The modest parallel speedup of both HPJava and HPF is due to communication overheads. The fact that HPJava and HPF have similar scaling behavior, while absolute performance of HPJava is lower, suggests the communication library of HPJava is slower than the communications of the native SP3 HPF (otherwise the performance gap would close for larger numbers of processors). This is not too surprising because Adlib is built on top of a portability layer called *mpjdev*, which is in turn layered on MPI. We assume the SP3 HPF is more carefully optimized for the hardware. Of course the lower layers of Adlib could be ported to exploit low-level features of the hardware (we already did some experiments in this direction, interfacing Java to LAPI [14]).

[7] There are also likely to be inherent penalties in using a JVM vs an optimizing Fortran compiler, but other experiments suggest these overheads should be smaller than what we see here. The communication overheads are probably aggravated by a choice we made in the data distribution format in these experiments. All levels are distributed blockwise. A better choice may be to distribute only the finest levels, and keep the coaser levels sequential. This doesn't require any change to the main code—only to initialization of the grid stack. However this wasn't what was done in these experiments.

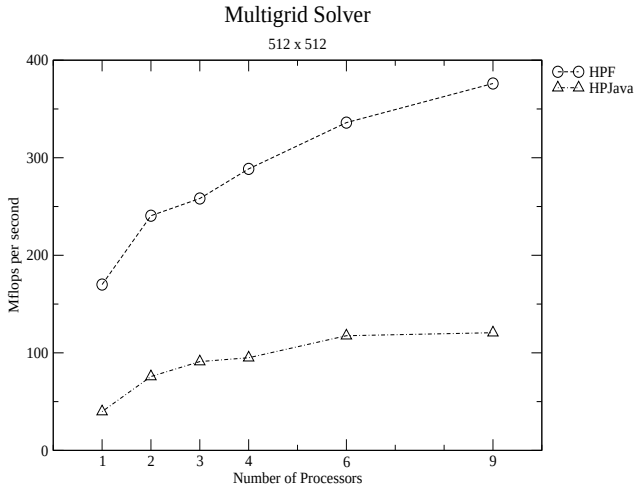

Fig. 7. Multigrid solver with size of 512^2.

5.2 CFD results

Figure 8 and gives some performance results for a version of the CFD code. The speedup results are quite reasonable, even for small problem sizes. Presumably this reflects the intrinsically greater granularity of this problem, compared with the multigrid case. (In this case unfortunately we don't have a Fortran version to compare with.)

6 Discussion

We illustrated, by a detailed discussion of the coding of two parallel applications, that the parallel primitives introduced in HPJava are a good match to the requirements of various applications. The limitations imposed on distributed control constructs like overall, and especially the strict rules for subscripting distributed arrays, may look strange from a language design perspective. But these features are motivated by patterns observed in practical parallel programs.

In particular the language provides a good framework for the development of SPMD libraries operating on distributed arrays. The collective operations of high-level libraries like Adlib, operating directly on distributed arrays, abstract and generalize the popular collective operations of MPI and other SPMD libraries. They also follow in the spirit of the array intrinsics and libraries of Fortran 90/95 and HPF. The language resembles HPF in various ways. But the programming model is closer to the MPI style. MPI programming seems to have been more popular in practice than HPF, perhaps because it gives the programmer control over communication, and it allows the programmer to estimate the cost of his program by looking at the code. We claim these as advantages for HPJava, too.

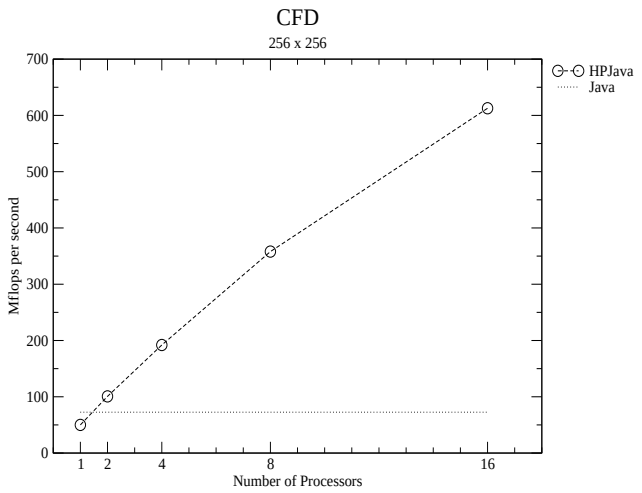

Fig. 8. CFD with size of 256^2.

In its current stage of development HPJava, like HPF, seems most naturally suited for problems with some regularity. This is not to say that more irregular problems can't be tackled. But doing so will at least need more specialized communication library support.

We have also shown that the performance of the initial implementation of HPJava is quite promising[8]. The current implementation provides full functionality, but it has not been seriously optimized. There is scope for dramatic improvements in efficiency [13]

7 Acknowledgement

This work was funded in part by National Science Foundation Division of Advanced Computational Infrastructure and Research, under contract number 987-2125. We are very grateful to Saleh Elmohamed for donating the original Java version of the CFD code, and for help with understanding and parallelizing it. All software discussed in this article, including the demonstration codes, is freely available, with full source, from `www.hpjava.org`.

References

1. mpiJava Home Page. http://www.hpjava.org/mpiJava.html.
2. Timber Compiler Home Page. http://pds.twi.tudelft.nl/timber.

[8] Java vs Fortran on the IBM machine is a relatively tough case. The IBM Fortran compilers tend to be better than those on important commodity platforms. On PCs the inherent performance of Java is typically more competitive with C and Fortran.

3. Titanium Project Home Page. http://www.cs.berkeley.edu/projects/titanium.
4. C.A. Addison, V.S. Getov, A.J.G. Hey, R.W. Hockney, and I.C. Wolton. *The Genesis Distributed-Memory Benchmarks*. Elsevier Science B.V., North-Holland, Amsterdam, 1993.
5. William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. The Society for Industrial and Applied Mathematics (SIAM), 2000.
6. Jayant DeSouza and L. V. Kale. Jade: A parallel message-driven java. In *Proceedings of the 2003 Workshop on Java in Computational Science*, Melbourne, Australia, 2003. Available from `http://charm.cs.uiuc.edu/papers/ParJavaWJCS03.shtml`.
7. E. Dick. Introduction to finite volume techniques in computational fluid dynamics. In John F. Wendt, editor, *Computational Fluid Dynamics: An Introduction*, pages 261–288. Springer-Verlag, 1992.
8. Jose E.Moreira, Samuel P. Midkiff, and Manish Gupta. A standard Java array package for technical computing. Technical Report RC21313, IBM Research, 1999. Available from `http://www.research.ibm.com/resources/`.
9. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification,* Second Edition. Addison-Wesley, 2000.
10. HPJava project home page. www.hpjava.org.
11. A. Jameson, W. Schmidt, and E. Turkel. Numerical solutions of the Euler equations by finite volume methods using Runge-Kutta time-stepping schemes. In *AIAA 14th Fluid and Plasma Dynamics Conference*. American Institute of Aeronautics and Astronautics, June 1981.
12. Java Grande Numerics Working Group home page. http://math.nist.gov/javanumerics/.
13. Han-Ku Lee. *Towards Efficient Compilation of the HPJava Language for High Performance Computing*. PhD thesis, Florida State University, June 2003.
14. Sang Boem Lim. *Platforms for HPJava: Runtime Support for Scalable Programming in Java*. PhD thesis, Florida State University, June 2003.
15. J. E. Moreira, S. P. Midkiff, M. Gupta, and R. Lawrence. High Performance Computing with the Array Package for Java: A Case Study using Data Mining. In *Supercomputing 99*, November 1999.
16. David Oh. The Java virtual wind tunnel. http://raphael.mit.edu/Java/.
17. R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference calculations. In *Object Oriented Numerics Conference*, 1994.
18. Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. PCRC-based HPF compilation. In Zhiyuan Li et al, editor, *10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*. Springer, 1997. http://www.hpjava.org/pcrc/npacWork.html.