HPJava: Towards Programming Support for High-Performance Grid-Enabled Applications

Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, Sang Boem Lim
{hkl, dbc, gcf, slim}@grids.ucs.indiana.edu

Pervasive Technology Labs at Indiana University
Bloomington, IN 47404-3730

Computational Science and Information Technology at Florida State University
Tallahassee, FL 32306

Abstract

We start seeing the authors' HPspmd Programming Model [1] *towards the programming support for High-Performance Grid-Enabled Environments since the future grid computing systems will need to provide programming models. As a proper programming model for grid-enabled environments and applications, high performance on multi-processor systems is critical issue. We argue with simple experiments that we can in fact hope to achieve high performance in a similar ballpark to more traditional HPC languages.*

Keywords: *HPspmd Programming Model, HPJava, High-Performance Grid-Enabled Environment, Java*

1 Introduction

In the earlier publications such as [7, 6], we argued that HPJava should ultimately provide acceptable performance to make it a practical tool for HPC. Moreover, from [18], we proved that HPJava node performance is quite acceptable, compared with C, FORTRAN, and ordinary Java: especially Java is no longer much slower than C and FORTRAN; The performance is becoming comparable. Thus, we verified our library-based HPspmd programming language extensions can be implemented efficiently in the context of Java.

The next step is to start viewing our HPspmd programming model as *Programming Support for High-Performance Grid-Enabled Environments.*

In this paper, first, we will see what grid computing is, why we need grid computing, and how our HPspmd programming model can be adapted for high-performance grid-enabled environments. Through this approach, we can gradually change our dimension for viewing HPspmd programming from high-performance computing to high-performance grid-enabled environments.

Secondly, we will review some features, run-time library, and compilation strategies including optimization schemes for HPJava. Moreover, we will experiment on simple HPJava programs comparing against FORTRAN and Java programs.

2 High-Performance Grid-Enabled Environments

2.1 Grid Computing

Grid computing can be defined as computing environments that are fundamentally distributed, heterogeneous, and dynamic for resources and performance. As inspired by [10], the Grid will establish an huge environment, connected by global computer systems such as end-computers, databases, and instruments, to make a World-Wide-Web-like distributed system for science and engineering.

The majority of scientific and engineering researchers believe that the future of computing will

[1] This work was supported in part by the National Science Foundation Division of Advanced Computational Infrastructure and Research, contract number 9872125.

heavily depends on grid computing for efficient and powerful computing by improving legacy technology, increasing demand-driven access to computational power, increasing utilization of idle capacity, sharing computational results, and providing new problem-solving techniques and tools. Of course, substantially powerful Grids can be established using high-performance networking, computing, and programming support regardless of the location resources and users.

Then, what will be the biggest potential issues in terms of programming supports we have to concentrate on grid computing and its environments, which wants to simplify distributed heterogeneous computing in the same way that the World-Wide-Web simplified information sharing over the internet? The *high-performance* is one possible answer since a slow system which has a clever motivation is useless. The other answer could be the thirst for *grid-enabled applications*, hiding the "heterogeneity" and "complexity" of grid environments without losing performance.

2.2 HPspmd Programming Model Towards Grid-Enabled Applications

To support "high-performance grid-enabled applications", the future grid computing systems will need to provide *programming models* [10]. The main thrust of programming models is to hide and simplify complexity and details of implementing the system, while focusing on the application design that have a significant impact on program performance or correctness.

Generally, we can see different programming models in sequential programming and parallel programming. For instance, in sequential programming, commonly used programming models for modern high-level languages furnish applications with inheritance, encapsulation, and scoping. Moreover, in parallel programming, distributed arrays, message-passing, threads, condition variables, and so on. Thus, using each model's significant charateristics, sequential and parallel programmings must maximize their performance and correctness.

But, there is no clarity about what programming model is appropriate for a grid environment, although it seems clear that many programming models will be used.

One approach to grid programming is to adapt programming models that have already proved successful in sequential or parallel environments. For example, the data-parallel language model such as HPspmd Programming Model might be an ideal programming model for supporting and developing high-performance grid-enabled applications, allowing programmers to specify parallelism in terms of process groups and distributed array operations. A grid-enabled MPI – currently our HPJava project team provides a Java implementation of MPI, called *mpiJava* – would extend the popular message-passing models. Java new I/O API's dramatic performance improvement encourages us to focus on grids-enabled MPI implementations as well. Moreover, high-performance grid-enabled applications and run-time systems demand "adaptability", "security", and "ultra-portability", which can be simply supported by the HPJava language since it is implemented in the context of Java.

Despite tremendous potential, enthusiasm, and commitment to Grid, few software tools and programming models exist for high-performance grid-enabled applications. Thus, to make prospective high-performance grid-enabled environments, we need nifty compilation techniques, high-performance grid-enabled programming models, applications, and components, and a better and improved base language (i.e. Java).

We believe that the success of HPJava would make our HPspmd Programming Model a promising candidate for constructing high-performance grid-enabled applications and components.

3 The HPJava Language

3.1 HPspmd Programming Model

HPJava [13] is an implementation of what we call the *HPspmd programming language model*. HPspmd programming language model is a flexible hybrid of HPF-like data-parallel features and the popular, library-oriented, SPMD style, omitting some basic assumptions of the HPF [12] model.

To facilitate programming of massively parallel, distributed memory systems, it extends the Java language with some additional syntax and some pre-defined classes for handling distributed arrays, and with *Adlib* [8], the run-time communication library. HPJava supports a true multi-dimensional array, which is a modest extension to the standard Java language, and which is a subset of our syntax for distributed arrays. HPJava introduces some new control constructs such as `overall`, `at`, and `on` statements.

As mentioned in earlier section 2.2, our HPspmd programming model must be the nifty choice to support high-performance grid-enabled applications in science and engineering.

```
Procs2 p = new Procs2(P, P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  double [[-,-]] c = new double [[x, y]] on p ;

  double [[-,*]] a = new double [[x, N]] on p ;
  double [[*,-]] b = new double [[N, y]] on p ;

  ... initialize 'a', 'b'

  overall(i = x for :)
    overall(j = y for :) {

      double sum = 0 ;
      for(int k = 0 ; k < N ; k++)
        sum += a [i, k] * b [k, j] ;

      c [i, j] = sum ;
    }
}
```

Figure 1. Matrix Multiplication in HPJava.

3.2 Features

Figure 1 is a basic HPJava program for a matrix multiplication. It includes much of the HPJava special syntax, so we will take the opportunity to briefly review the featues of the HPJava language. The program starts by creating an instance `p` of the class `Procs2`. This is a subclass of the special base class `Group`, and describes 2-dimensional grids of processes. When the instance of `Procs2` is created, `P` × `P` processes are selected from the set of processes in which the SPMD program is executing, and labelled as a grid.

The `Group` class, representing an arbitrary HPJava process group, and closely analogous to an MPI group, has a special status in the HPJava language. For example the group object `p` can parametrize an `on(p)` construct. The `on` construct limits control to processes in its parameter group. The code in the `on` construct is *only* executed by processes that belong to `p`. The `on` construct fixes `p` as the *active process group* within its body.

The `Range` class describes a distributed index range. There are subclasses describing index ranges with different properties. In this example, we use the `BlockRange` class, describing block-distributed indexes. The first argument of the constructor is the global size of the range; the second argument is a *process dimension*—the dimension over which the range is distributed. Thus, ranges `x` and `y` are distributed over the first dimension (i.e. `p.dim(0)`) and second dimension (i.e. `p.dim(1)`) of `p`, and both have `N` elements.

The most important feature HPJava adds to Java is the *distributed array.* A distributed array is a collective object shared by a number of processes. Like an ordinary array, a distributed array has some index space and stores a collection of elements of fixed type. Unlike an ordinary array, the index space and associated elements are scattered across the processes that share the array. There are some similarities and differences between HPJava distributed arrays and the ordinary Java arrays. Aside from the way that elements of a distributed array are distributed, the distributed array of HPJava is a *true multi-dimensional array* like that of FORTRAN. Like in FORTRAN, one can form a *regular section* of an array. These features of FORTRAN arrays have adapted and evolved to support scientific and parallel algorithms.

With a process group and a suitable set of ranges, we can declare distributed arrays. The type signature of a distributed array is clearly told by double brackets. In the type signature of a distributed array, each slot holding a hypen, `-`, stands for a distributed dimension, and a astrisk, `*`, a sequential dimension. The array `c` is distributed in both its dimensions. Besides, Arrays `a` and `b` are also distributed arrays, but now each of them has one distributed dimension and one *sequential dimension.*

The `overall` construct is another control construct of HPJava. It represents a distributed parallel loop, sharing some characteristics of the *forall* construct of HPF. The symbol `i` scoped by the `overall` construct is called a *distributed index.* Its value is a *location*, rather an abstract element of a distributed range than an integer value. The indexes iterate over all locations. It is important to note that (with a few special exceptions) the subscript of a distributed array must be a distributed index, and the location should be an element of the range associated with the array dimension. This unusual restriction is an important feature of the model, ensuring that referenced array elements are locally held.

Figure 1 doesn't have any run-time communications because of the special choice of alignment relation between arrays. All arguments for the innermost scalar product are already in place for the computation. We can make a completely general matrix multiplication method by taking arguments with arbitrary distribution, and remapping the input arrays to have the correct alignment relation with the output array. Figure 2 shows the method. The method has two temporary arrays `ta`, `tb` with the desired distribution format. This

```
void matmul(double [[-,-]] c,
            double [[-,-]] a, double [[-,-]] b) {

  Group p = c.grp() ;

  Range x = c.rng(0) ;
  Range y = c.rng(1) ;

  int N = a.rng(1).size() ;

  double [[-,*]] ta = new double [[x, N]] on p ;
  double [[*,-]] tb = new double [[N, y]] on p ;

  Adlib.remap(ta, a) ;
  Adlib.remap(tb, b) ;

  on(p)
    overall(i = x for :)
      overall(j = y for :) {

        double sum = 0 ;
        for(int k = 0 ; k < N ; k++)
          sum += ta [i, k] * tb [k, j] ;

        c [i, j] = sum ;
      }
}
```

Figure 2. General matrix multiplication.

is determined from `c` by using DAD inquiry functions `grp()` and `rng()` to fetch the distribution group and index ranges of a distributed array. `Adlib.remap()` does the actual communication to remap.

This implementation has some performance issues associated with its memory usage. These issues can be patched up—see [7] for more details. Meanwhile the simple version given here encapsulates some interesting principles of library construction with HPJava—in particular how arrays can be created and manipulated, even though the distribution formats are only determined at run-time.

We will give another old favorite program, red-black relaxation. It is still interesting since it is a kernel in some practical solvers (for example we have an HPJava version of a multigrid solver in which relaxation it is a dominantly time-consuming part). Also it conveniently exemplifies a whole family of similar, local, grid-based algorithms and simulations.

We can see an HPJava version of red-black relaxation of the two dimensional Laplace equation in Figure 3. Here we use a different class of distributed range. The class `ExtBlockRange` adds *ghost-regions* [11] to distributed arrays that use them. A library function called `Adlib.writeHalo` updates the cached values in the ghost regions with proper element values from neighboring processes.

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Range x = new ExtBlockRange(N, p.dim(0)) ;
  Range y = new ExtBlockRange(N, p.dim(1)) ;

  double [[-,-]] a = new double [[x, y]] on p ;

  ... initialization for 'a'

  for(int iter=0; iter<count; iter++){

    Adlib.writeHalo(a, wlo, whi);

    overall(i=x for 1 : N - 2)
      overall(j=y for 1+(i`+iter)%2 : N-2 : 2) {
        a[i,j] = 0.25F * (a [i-1,j] + a [i+1,j] +
                          a [i,j-1] + a [i,j+1]);
      }
  }
}
```

Figure 3. Red-black iteration.

There are a few additional pieces of syntax here. The range of iteration of the overall construct can be restricted by adding a general triplet after the `for` keyword. The i` is read "i-primed", and yields the integer *global index* value for the distributed loop (`i` itself does not have a numeric value—it is a symbolic subscript). Finally, if the array ranges have ghost regions, the general policy that an array subscript must be a simple distributed index is relaxed slightly—a subscript can be a *shifted index*, as here. The value of the numeric shift—symbolically added to or subtracted from the index—must not exceed the width of the ghost regions, and the index that is shifted must be a location in the distributed range of the array, as before.

3.3 Run-time Communication Library

In this section, we mention Adlib, the HPJava run-time communication library, and the *mpjdev* API [19], which is designed with the goal that it can be implemented portably on network platforms and efficiently on parallel hardware. It needs to support communication of intrinsic Java types, including primitive types, and objects. It should transfer data between the Java program and the network while keeping the overheads of the Java Native Interface as low as practical.

Unlike MPI which is intended for the application developer, mpjdev is meant for library developers. Application level communication libraries like the Java

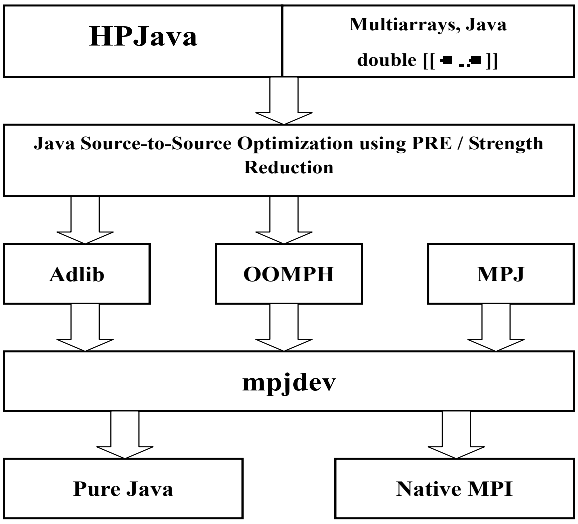

Figure 4. HPJava Architecture.

version of Adlib, or MPJ [5] might be implemented on top of mpjdev. mpjdev itself may be implemented on top of Java sockets in a portable network implementation, or—on HPC platforms—through a JNI interface to a subset of MPI. The positioning of the mpjdev API is illustrated in Figure 4.

The initial version of the mpjdev has been targeted to HPC platforms—through a JNI interface to a subset of MPI. A Java sockets version that provides more portable network implementation is included in HPJava 1.0.

4 Compilation Strategies for HPJava

In this section, we will see efficient compilation strategies for HPJava. The HPJava compilation system consist of three parts; Parser, Type-Analyzer, Translator, and Optimizer. HPJava adopted JavaCC [15] as a parser generator. Type-Analyzer, Translator, and Optimizer are reviewed in following subsections. Figure 4 is the overall HPJava hierarchy.

4.1 Type-Analysis and Translation

The implementation of type-analysis (i.e. type-checking) system for the HPJava language has been the most time-consuming part of implementing the entire system. Since the HPJava language is a superset of ordinary Java language, HPJava fully supports the *Java Language Specification* [16].

In stark distinction to HPF, the HPJava translation scheme *does not* require insertion of compiler-generated communications, making it relatively straightforward. The most complicated part is ensuring that node code works independently of the distribution format of arrays. The current translation schemes is documented in detail in the HPJava manual, called *Programming in HPJava* [7], and translation scheme [4].

4.2 Optimization

For common parallel algorithms, where HPJava is likely to be successful, distributed element access is generally located inside distributed `overall` loops. One main issue optimization strategies must address is the complexity of the associated terms in the subscript expressions for addressing local element for distributed arrays. Optimization strategies should remove overheads of the naive translation scheme (especially for `overall` construct), and speed up HPJava, i.e. produce a Java-based environment competitive in performance with existing FORTRAN programming environments.

To eliminate complicated distributed index subscript expressions in the inner loops, the HPJava compiler will adopt both of *Partial Redundancy Elimination* (PRE) algorithm from [17] and *Strength Reduction* (SR) algorithm from [2].

PRE is a very important optimization technique to remove partial redundancies in the program by analyzing data flow graph that solves code replacements. PRE is a powerful and proper algorithm for HPJava compiler optimization since `overall` loops are the right locations which have the complexity of the associated terms in the subscript expressions for addressing local element for distributed arrays, and since *loop invariants*, which are naturally partially redundant, are generally located in the subscript expression of distributed arrays. Moreover, PRE should be applied to a general or *Static Single Assignment* (SSA) [9] formed data flow graph after adding *landing pads* [20], representing entry to the loop from outside.

SR replaces expensive and slow operations by equivalent, efficient, cheaper, and fast ones on the target machine. SR is effective to be used to replace computations involving multiplications and additions with ones involving only additions. The combination of complex multiplications and additions are natural to the subscript expression of distributed arrays.

Thus, applying `PRE / SR` to the naively translated codes could make performance of HPJava faster, and could have HPJava comparable to C, FORTRAN, and Java. In the section 5, we will prove leaps of performance for HPJava using `PRE / SR` before adopting the

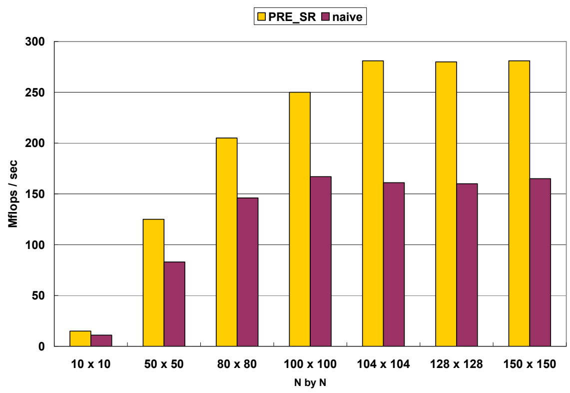

Figure 5. Experiment for matrix multiplication in HPJava with PRE/SR.

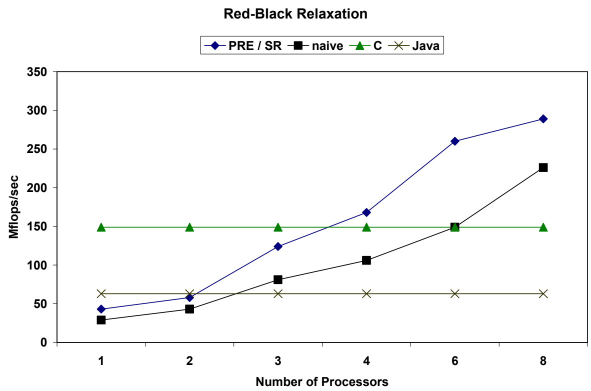

Figure 6. Laplace Equation with red-black relaxation with size of 512 x 512 on Sun Solaris 9 with 8 UltraSPARC III processors.

optimization strategies to HPJava.

5 Experiments

As we mentioned earlier, we proved that HPJava individual node performance is quite acceptable, and proved that Java itself can get 70 – 75% of the performance of C and FORTRAN from the previous publication [18]. Moreover, from Figure 5, we can see the dramatic result after applying `PRE/SR`. The results use the IBM Developer Kit 1.3 (JIT) with `-O` flag on Pentium4 1.5GHz Red Hat 7.2 Linux machines. Thus, now, we expect that the HPJava results will scale on suitable parallel platforms, so a *modest* penalty in node performance is considered acceptable.

5.1 Laplace Equation with Red-Black Relaxation

First, we experiment HPJava with a simple Laplace Equation with red-black relaxation on the Sun Solaris 9 with 8 UltraSPARC III Cu 900MHz Processors and 16GB of main memory.

Figure 6 shows the result of four different versions (HPJava with `PRE/SR` optimization, HPJava with naive translation, FORTRAN, and Java) of red-black relaxation of the two dimensional Laplace equation. After applying `PRE/SR` for the naive translation, HPJava can be improved up to 170% of the performance.

Second, The results of our benchmarks use an IBM SP3 running with four Power3 375MHz CPUs and 2GB of memory on each node. This machine uses AIX version 4.3 operating system and the IBM Developer Kit 1.3.1 (JIT) for the Java system. We are using the shared "css0" adapter with User Space(US) communication mode for MPI setting and -O compiler command for Java. For comparison, we also have completed experiments for sequential Java, Fortran and HPF version of the HPJava programs. For the HPF version of program, it uses IBM XL HPF version 1.4 with *xlhpf95* compiler commend and -O3 and -qhot flag. And XL Fortran for AIX with -O5 flag is used for Fortran version.

Figure 7 shows result of four different versions (HPJava, sequential Java, HPF and Fortran) of red-black relaxation of the two dimensional Laplace equation with size of 512 by 512. In our runs HPJava can outperform sequential Java by up to 17 times. On 36 processors HPJava can get about 78% of the performance of HPF. It is not very bad performance for the initial benchmark result. Scaling behavior of HPJava is slightly better then HPF. Probably, this mainly reflects the low performance of a single Java node compare to FORTRAN. We do not believe that the current communication library of HPJava is faster than the HPF libray because our communication library is built on top of the portablity layers, mpjdev and MPI, while IBM HPF is likely to use a platform specific communication library. But clearly future versions of Adlib could be optimized for the platform.

We see similar behavior on large size of three dimensional Diffusion equation benchark (Figure 8). In general we expect 3 dimensional problems will be more amenable to parallelism, because of the large problem size.

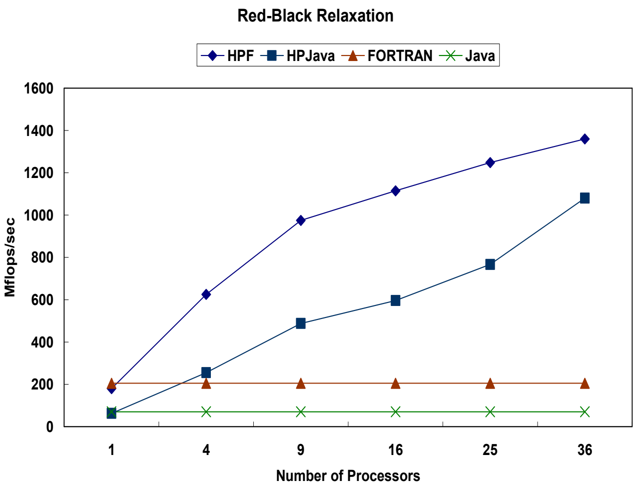

Figure 7. Laplace Equation with red-black relaxation with size of 512 x 512 on IBM SP3.

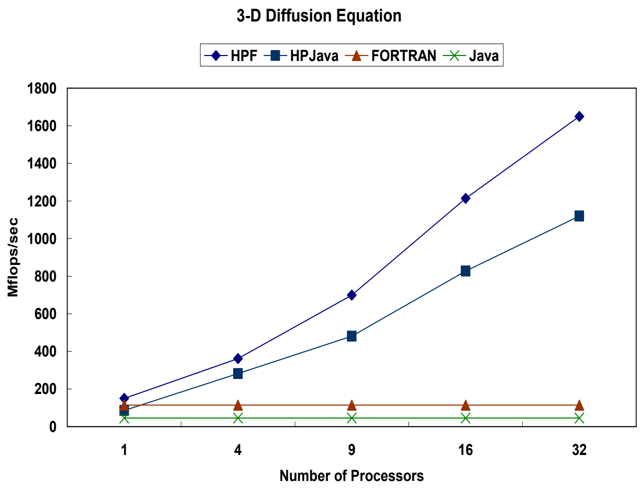

Figure 8. Three dimensional Diffusion equation with size of 128 x 128 x 128 on IBM SP3.

6 Related Works

HPJava is an instance of what we call the *HPspmd model*: arguably it is not exactly a high-level parallel programming language in the ordinary sense, but rather a tool to assist parallel programmers in writing SPMD code. In this respect the closest recent language we are familiar with is probably Co-Array FORTRAN [21], but HPJava and Co-Array FORTRAN have many obvious differences. In Co-Array FORTRAN, array subscripting is local by default, or involves a combination of local subscripts and explicit process ids. There is no analogue of global subscripts, or HPF-like distribution formats. In Co-Array FORTRAN the logical model of communication is built into the language—remote memory access with intrinsics for synchronization. In HPJava, there are no communication primitives in the language itself. We follow the MPI philosophy of providing communication through separate libraries.

In grid computing, the *GrADS Project* [3] is to simplify distributed computing in the same way that the World Wide Web simplified information sharing over internet. To that end, the project is exploring the scietific users to develop, execute, and tune application on the Grid. Even though the project shares the same purpose with our high-performance grid-enabled application using HPspmd programming model, it doesn't adapt programming models that have already proved successful in sequential or parallel environments.

7 Conclusions

The main purpose of this paper was to verify if our library-based HPspmd Programming Model can be efficiently adapted into and implemented for the programming support for high-performance grid-enabled applications in the context of Java. Though thr experiments, we proved that HPJava has quite acceptable performance on scientific and engineering (matrix) algorithms, which plays very important asset in high-performance grid-enabled applications such as search engines, using "parameter searching".

Now, the first fully functional HPJava is operational and can be downloaded from [13]. The system fully supports the Java Language Specification [16], and has tested and debugged against the HPJava test suites and *jacks* [14], an Automated Compiler Killing Suite. The current score is comparable to that of Sun jdk 1.4 and IBM Developer Kit 1.3.1. This means that the HPJava front-end is very conformant with Java. The HPJava test suites includes simple HPJava programs, and complex scientific algorithms and applications such as a multigrid solver, adapted from an existing FORTRAN program (called PDE2), taken from the Genesis parallel benchmark suite [1]. The whole of this program has been ported to HPJava (it is about 800 lines). A research application for fluid flow problems, CFD [2] (Computational Fluid Dynamics) has been ported to

[2]CFD simulates a 2-D inviscid flow through an axisymmetric nozzle. The simulation yields contour plots of all flow variables, including velocity components, pressure, mach number, density and entropy, and temperature. The plots show the location of

HPJava (it is about 1300 lines) as well.

References

[1] C. Addison, V. Getov, A. Hey, R. Hockney, and I. Wolton. *The Genesis Distributed-Memory Benchmarks*. Elsevier Science B.V., North-Holland, Amsterdam, 1993.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Pub Co, 1986.

[3] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *The International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[4] B. Carpenter, G. Fox, H.-K. Lee, and S. B. Lim. Translation Schemes for the HPJava Parallel Programming Language. In *11th International Workshop on Languages and Compilers for Parallel Computing 2001*, 2001.

[5] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[6] B. Carpenter, G. Zhang, G. Fox, X. Li, X. Li, and Y. Wen. Towards a Java environment for SPMD programming. In D. Pritchard and J. Reeve, editors, *4th International Europar Conference*, volume 1470 of *Lecture Notes in Computer Science*. Springer, 1998.

[7] B. Carpenter, G. Zhang, H.-K. Lee, and S. Lim. *Parallel Programming in HPJava*. Draft, 2001. http://www.hpjava.org.

[8] B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC Runtime Kernel Definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. Up-to-date version maintained at http://www.npac.syr.edu/projects/pcrc/doc.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficient Computing Static Single Assignment Form and the Control Dependence Grapg. *ACM Transactions on Programming Languages and Systems*, pages 451–490, 1991.

[10] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[11] M. Gerndt. Updating Distributed Variables in Local Computations. *Concurrency: Practice and Experience*, 2(3):171–193, 1990.

[12] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, Special Issue, 2, 1993.

[13] HPJava Home Page. htt://www.hpjava.org.

[14] Jacks (Java Automated Compiler Killing Suite). http://www-124.ibm.com/developerworks/oss/cvs/jikes/~checkout~/jacks/jacks.html.

[15] JavaCC – Java Compiler Compiler (Parser Generator). http://www.webgain.com/products/java_cc/.

[16] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley Pub Co, 2000.

[17] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. Partial Redundancy Elimination in SSA Form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.

[18] H.-K. Lee, B. Carpenter, G. Fox, and S. B. Lim. Benchmarking HPJava: Prospects for Performance. In *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers(LCR2002)*, Lecture Notes in Computer Science. Springer, March 2002.

[19] S. B. Lim, B. Carpenter, G. Fox, and H.-K. Lee. Collective Communication for the HPJava Programming Language. *To appear Concurrency: Practice and Experience*, 2003.

[20] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1988.

[21] R.W.Numrich and J.L.Steidel. F- -: A Simple Parallel Extension to Fortran 90. *SIAM News*, page 30, 1997.

any shock wave that would reside in the nozzle. Also, the code finds the steady state solution to the 2-D Euler equations.