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

Han-Ku Lee

2035 Hughes Hall
Old Dominion University
Norfolk, VA 23529
U.S.A.

Bryan Carpenter, Geoffrey Fox, Sang Boem Lim

Pervasive Technology Labs at Indiana University
Indiana University
Bloomington, IN 47404-3730
U.S.A.

Abstract

The paper begins by considering what a Grid Computing Environment might be, why it is demanded, and how the authors' HPspmd programming fits into this picture. We then review our HPJava environment[1] *as a contribution towards programming support for High-Performance Grid-Enabled Environments. Future grid computing systems will need to provide programming models. In a proper programming model for grid-enabled environments and applications, high performance on multi-processor systems is a critical issue. We describe the features of HPJava, including run-time communication library, compilation strategies, and optimization schemes. Through experiments, we compare HPJava programs against FORTRAN and ordinary Java programs. We aim to demonstrate that HPJava can be used "anywhere"—not only for high-performance parallel computing, but also for grid-enabled applications.*

Keywords: *HPspmd, HPJava, High-Performance, Grids, Grid-Enabled Environments, Java*

1 Introduction

HPJava is an environment for parallel and scientific programming with Java, developed by the present authors [14]. It provides a programming model with various similarities to—and some notable differences from—the programming model of *High Performance Fortran*, (HPF) [13, 19]. In comparison with HPF, HPJava has a stronger emphasis on the role of *libraries* that operate on distributed data. This emphasis sits naturally with the object-oriented features of the base language—Java rather than Fortran[2]. In other respects HPJava provides a lower-level programming model than HPF—with multiple SPMD threads of control. But it retains important distributed data abstractions from HPF. It is most specifically aimed at *distributed memory* architectures, although of course HPJava can also run on shared memory computers. We call our general programming model the *HPspmd model*—so HPJava is a realization of the HPspmd model.

An initial release of the HPJava environment was placed in the public domain last year (2003). This includes a reasonably sophisticated translator implementing the full HPJava language (which

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

[2]Though HPJava does not *enforce* a completely object-oriented programming style—it consciously allows a more "Fortran-like" style, where appropriate.

includes the full Java language as a subset), and a set of supporting libraries. In earlier publications we argued that HPJava should eventually provide acceptable performance, making it a practical tool for HPC [7, 8]. We showed in [20] that HPJava node performance is quite reasonable, compared with C, Fortran, and ordinary Java: in particular on many platforms Java is no longer much slower than C or Fortran. Thus we verified our library-based HPspmd programming language extensions can be implemented quite efficiently in the context of Java.

In part because the SPMD programming model is relatively loosely coupled (compared with HPF), and in part because of its foundation in a ubiquitous Internet programming language (Java), we can make the case that our HPspmd programming model provides promising programming support for high-performance grid-enabled environments. This paper begins with an interpretation of what "grid computing" means, why it is needed, and how our HPspmd programming model can be adapted to this end—shifting the focus from high-performance computing to high-performance grid-enabled environments. We then review some features of HPJava including run-time library and compilation strategies, and optimization schemes. We present some experiments on simple HPJava programs, comparing against Fortran and ordinary Java programs. We hope to demonstrate a promising future for HPJava, which can be used "anywhere"—not only for high-performance parallel computing, but also for *grid-enabled* applications.

2 High-Performance Grid-Enabled Environments

2.1 Grid Computing

Grid computing environments have been defined as computing environments that are fundamentally distributed, heterogeneous, and dynamic, in their resources and performance. According to [11], Grid initiatives will establish a huge environment—connecting global computer systems, including end-computers, databases, and instruments, into a World-Wide-Web-like distributed system for science and engineering.

Many researchers in science and engineering believe that the future of computing will heavily depend on the Grid for efficient and powerful computing, 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. Substantially powerful Grids can be established using high-performance networking, computing, and programming support, regardless of the location resources and users.

We can ask what will be the biggest *hurdles*—in terms of programming support—to simplify distributed heterogeneous computing—in the same way that the World Wide Web simplified information sharing over the Internet. One possible answer is *high performance*—a slow system that has a clever motivation is not very useful. There will be a pressing need for *grid-enabled applications* that hide the heterogeneity and complexity of the underlying grid, without losing performance.

Today, programmers often write grid-enabled application in what is in effect an assembly language: sometimes using explicit calls to the Internet Protocol's User Datagram Protocol (UDP) or Transmission Control Protocol (TCP), with hard-coded configuration decisions for specific computing systems. We are a far from portable, efficient, high-level languages.

2.2 HPspmd Programming Model: Towards Grid-Enabled Applications

To support "high-performance grid-enabled applications", future grid computing systems will need to provide *programming models* [11]. The main thrust of programming models is to hide and simplify complexity and details of implementing the system, while focusing on application design issues that have a significant impact on program performance or correctness.

Despite tremendous commitment to the Grid, few software tools and programming models exist for high-performance grid-enabled applications. To make excellent high-performance grid-

enabled environments, we probably need compilation techniques, programming models, applications, components—and also modern base languages.

Generally, we 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. In parallel programming we have distributed arrays, message-passing, threads, condition variables, and so on. There is much less clarity about what programming model is appropriate for a grid environment, although it seems certain that many programming models will be used.

One approach is to adapt programming models that have already proved successful in sequential or parallel environments. For example a data-parallel language model like our HPspmd programming model might be an excellent 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. High-performance grid-enabled applications and run-time systems demand "adaptability", "security", and "ultra-portability". These should be relatively easy to provide in HPJava, since it is implemented in the context of Java, which has always had such design goals.

The HPJava language has quite acceptable performance on scientific and engineering algorithms, which also play important roles in grid-enabled applications—from search engines to "parameter searching". Another interesting Grid problem where HPJava may have a role is coordinating the execution and information flow between multiple Web services, where each Web service has a WSDL style interface. For example we plan to use HPJava as middleware in the project called "BioComplexity Grid Environments" [4] at Indiana University. So we believe HPJava and the HPspmd Programming Model are promising candidates for constructing high-performance grid-enabled applications and components.

3 The HPJava Language

3.1 HPspmd Programming Model

HPJava [14] is an implementation of what we call the *HPspmd programming language model.* This is a flexible hybrid of HPF-like data-parallel features and the popular, library-oriented, SPMD style, omitting some basic assumptions of the HPF [13] model.

To facilitate programming of massively parallel, distributed memory systems, we extend the Java language with some additional syntax and some pre-defined classes for describing distributed arrays, and for passing these as arguments to library calls. Besides distributed arrays, HPJava also includes true multi-dimensional "sequential" arrays—a modest extension to the standard Java language. HPJava introduces three new control constructs: the `overall`, `at`, and `on` statements.

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 features 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 labeled 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 parameterize 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.

```
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.

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 *multiarray*. There are two kinds of multiarray in HPJava: *distributed arrays* and sequential *multidimensional arrays*[3]. A distributed array is a *collective array*, 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. A sequential multi-dimensional array is similar to an ordinary Fortran array. Like in Fortran (but unlike ordinary Java arrays) one can form a *regular section* of a multiarray.

With a process group and a suitable set of ranges, we can declare distributed arrays. The type signature of a distributed array is clearly distinguished by double brackets. In the type signature of a distributed array each slot holding a hyphen, `-`, stands for a distributed dimension, while an asterisk, `*`, stands for a sequential dimension. The array `c` is distributed in both its dimensions. 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 with the *forall* construct of HPF. The symbol `i` scoped by the `overall` construct is called a *distributed index*. Its value is a *location*—an abstract element of a distributed range. In general 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 does not 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 argu-

[3]These aren't really different types—to be precise sequential multiarrays are a subset of distributed arrays.

```
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.

ments with arbitrary distribution, and remapping the input arrays to have the correct alignment in relation to the output array. Figure 2 shows the method. It has two temporary arrays `ta`, `tb` with the desired distribution format. This 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. *Adlib* [9], described in section 3.3, is a communication library available for use with HPJava.

This simplified example 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 remains interesting since it is a kernel in some practical solvers (for example we have an HPJava version of a multigrid solver in which relaxation is a dominantly time-consuming part). Also it conveniently exemplifies a 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* [12] 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.

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.

```
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.

3.3 Run-time Communication Library

The examples in this paper use the communication library called Adlib [9]. Earlier versions of this library were developed some years ago by the current authors, as run-time systems for HPF translators. The current version is a reimplementation in Java, designed as an *application level* library for HPJava programming. It provides a family of collective operations that work directly with the distributed arrays of our HPspmd model.

The new version of Adlib is implemented on top of a low-level API called *mpjdev* [21], 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 version of Adlib, or MPJ [6] might be implemented on top of mpjdev. The positioning of the mpjdev API is illustrated in Figure 4. The initial version of the mpjdev has been targeted to HPC platforms—implemented through a JNI interface to a subset of MPI. A Java "New I/O" based version, providing a more portable network implementation, is under development.

4 Compilation Strategies for HPJava

In this section, we will discuss efficient compilation strategies for HPJava. The HPJava compilation system consists of four parts; Parser, Type-Analyzer, Translator, and Optimizer. HPJava adopted JavaCC [16] as a parser generator. Type-Analyzer, Translator, and Optimizer are reviewed in following subsections. Figure 4 is the overall architecture of the HPJava translator.

4.1 Type-Analysis

The current version of the HPJava type-checker (front-end) has three main phases; *type-analysis*, *reachability analysis*, and *definite (un)assignment analysis*. Development of this type-checker was one of the most time-consuming parts during the implementation of the whole HPJava compiler.

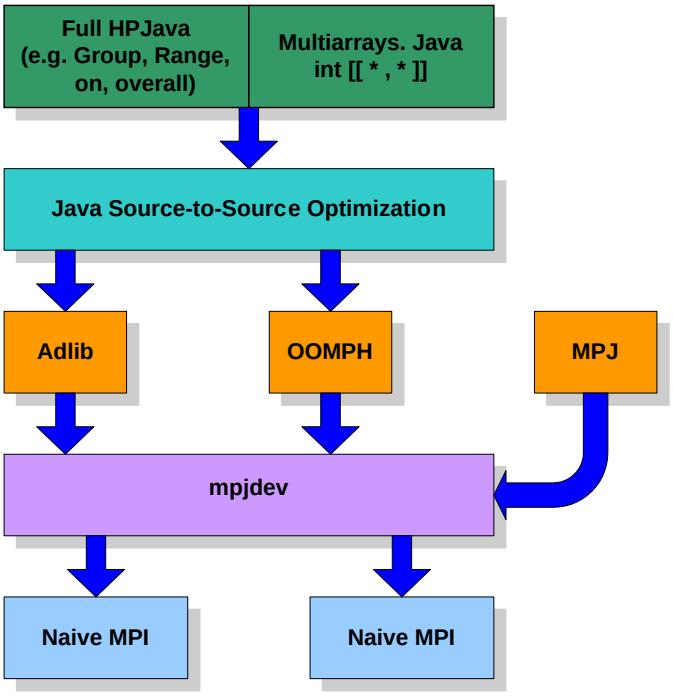

Figure 4. HPJava Architecture.

The first phase is *type-analysis*. It has five subordinate phases: *ClassFinder*, *ResolveParents*, *ClassFiller*, *Inheritance*, and *HPJavaTypeChecker*.

1. `ClassFinder` collects some simple information about top-level and nested class or interface declarations, such as names of the classes, the names of super class, and the names of super interfaces.

2. `ResolveParents` resolves class's proper super class and super interfaces using the information from ClassFinder.

3. `ClassFiller` fulfills a more complicated mission. It collects all of the rest of the information about top-level and nested class or interface declarations, such as field declarations, method declarations, constructor declarations, anonymous class allocations, and so on. `ClassFiller` also collects and resolves single-type-import declarations and type-import-on-demand declarations.

4. `Inheritance` collects and resolves the method inheritance, overriding, and hiding information to be used in `HPJavaTypeChecker`.

5. `HPJavaTypeChecker` does type-checking on statements, statement expressions, and expressions, including all ordinary Java, and the newly introduced HPJava constructs—multiarrays, etc.

The second phase is *reachability analysis*, carrying out a conservative flow analysis to make sure all statements are reachable. The idea of `Reachability` is that there must be some possible execution path from the beginning of the constructor, method, instance initializer or static initializer that contains the statement to the statement itself. The analysis takes into account the structure of statements. Except for the special treatment of while, do, and for statements whose condition expression has the constant value `true`, the values of expressions are not taken into account in the flow analysis.

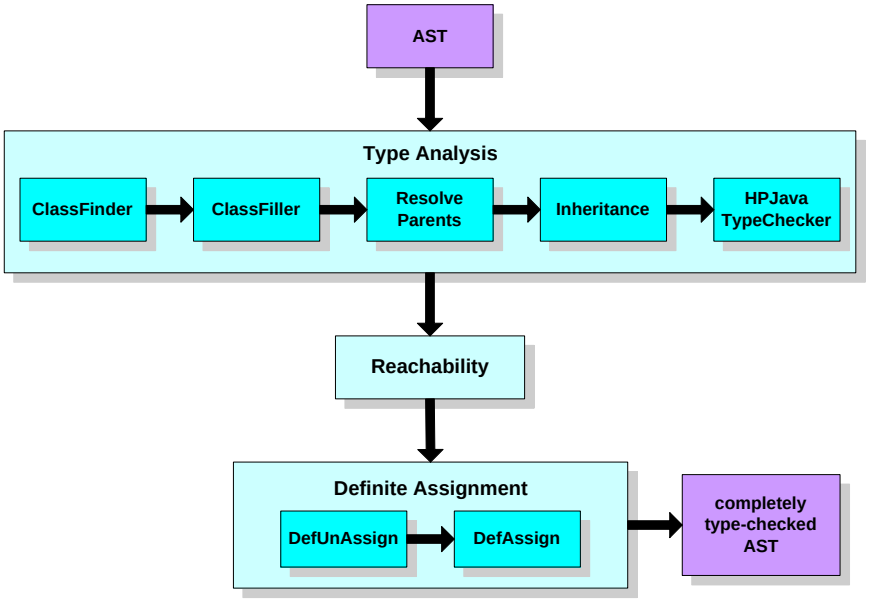

Figure 5. The Architecture of HPJava Front-End.

The third phase is *definite (un)assignment analysis*. It consists of two parts, *DefAssign* and *DefUnassign*, which checks the definite (un)assignment rules for Java, implementing the flow analysis of the Java language specification.

The three phases are basically inspired by carefully organizing *The Java Language Specification, Second Edition* (JLS) [17]. The current HPJava type-checker system only supports exactly what JLS says (plus the HPJava extensions, of course). But the Java language itself keeps evolving in a slow manner. For example, Sun's SDK 1.4 Java compiler supports some changes to JLS, and the 1.5 release will add many more. We expect to update the HPJava type-checker system when the new edition of JLS is eventually published. Figure 5 shows the complete architecture of HPJava front-end.

4.2 Translation

Currently, the HPJava translator has two phases, *pre-translation* and *translation*. Pre-translation reduces the source HPJava program to an equivalent program in a *restricted form* of the full HPJava language. The translation phase transforms the pre-translated program to a standard Java program. The main reason the current HPJava translator has two phases is to make the basic translation scheme simple and clear, by transforming certain complex expressions involving multiarrays into simple expressions in the pre-translation phase. Thus, the restricted form generated by the pre-translator should be suitable for being processed in the translation phase.

Unlike with HPF, the HPJava translation scheme *does not* require insertion of compiler-generated communications, making it relatively straightforward. The current translation schemes is documented in detail in the HPJava manual, called *Parallel Programming in HPJava* [7], and also in the paper [5].

4.3 Optimization

For common parallel algorithms, where HPJava is likely to be effective, distributed element access is generally located inside distributed `overall` loops. Optimization strategies must address

the complexity of the terms in the subscript expressions for addressing local elements of distributed arrays. When an `overall` construct is translated, the naive translation scheme generates 4 control variables outside the loop (refer to [7]). A control variable is often a dead code, and is partially redundant for the outter `overall` for nested `overalls`.

Optimization strategies should remove overheads of the naive translation scheme (especially for `overall` construct), and speed up HPJava, to produce a Java-based environment competitive in performance with existing Fortran programming environments.

4.3.1 Partial Redundancy Elimination

Partial Redundancy Elimination (PRE) [18] is a very important optimization technique to remove partial redundancies in the program by solving a data flow problem that yields code replacements. PRE is a powerful and proper algorithm for HPJava compiler optimization, since *loop invariants*, which are naturally partially redundant, often occur in the subscript expression of distributed arrays.

PRE should be applied to a general or *Static Single Assignment* (SSA) [10] formed data flow graph after adding *landing pads* [24], representing entry to the loop from outside.

PRE is a complicated algorithm to understand and to implement, compared with other well-known optimization algorithms. However, the basic idea of PRE is simple. Basically, PRE converts partially redundant expressions into redundant expressions. The key steps of PRE are:

Step 1: *Discover where expressions are partially redundant using data-flow analysis.*

Step 2: *Solve a data-flow problem that shows where inserting copies of a computation would convert a partial redundancy into a full redundancy.*

Step 3: *Insert the appropriate code and delete the redundant copy of the expression.*

4.3.2 Strength Reduction

Strength Reduction (SR) [2] replaces expensive operations by equivalent cheaper ones from the target machine language. Some machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For instance, x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine. Moreover, an additive operator is generally cheaper than a multiplicative operator.

SR is a very powerful algorithm to transform computations involving multiplications and additions together into computations only using additions. It is a natural candidate to optimize the current HPJava compiler because of the complexity of the terms in the subscript expression of a multiarray element access.

4.3.3 Dead Code Elimination

Dead Code Elimination (DCE) [2] is an optimization technique to eliminate some variables not used at all. An `overall` construct generates 6 variables outside the loop according to the naive translation scheme. Since these control variables are often unused, and the methods are specialized methods known to the compiler—side effect free—we don't have any side effects from applying DCE with data flow analysis to them.

We assume that DCE should be applied after all optimization techniques we discussed earlier. Moreover, we assume we narrow the target of DCE for the HPJava optimization strategy. That is, for the moment, DCE will target only control variables and control scripts for `overall` constructs.

4.3.4 Loop Unrolling

In conventional programming languages some loops have such a small body that most of the time is spent to increment the loop-counter variables and to test the loop-exit condition. We can make these loops more efficient by *unrolling* them, putting two or more copies of the loop body in a row. This technique is called *Loop Unrolling* (LU) [2].

Our experiments [20] did not suggest that this is a very useful source-level optimization for HPJava, except in one special but important case. LU will be applied for `overall` constructs in an HPJava Program of Laplace equation using red-black relaxation. For example,

```
overall (i = x for 1 + (i‘ + iter) % 2 : N-2 : 2) { ... }
```

The `overall` is repeated in every iteration of the loop associated with the outer `overall`. Because red-black iteration is a common pattern, HPJava compiler should probably try to recognize this idiom. If a nested overall includes expressions of the form

```
(i‘ + expr) % 2
```

where `expr` is invariant with respect to the outer loop, this should be taken as an indication to unroll the outer loop by 2. The modulo expressions then become loop invariant and arguments of the call to `localBlock()`, the whole invocation is a candidate to be lifted out of the outer loop.

4.3.5 HPJOPT2

To eliminate complicated distributed index subscript expressions and to hoist control variables in the inner loops, we adopted the following algorithm;

Step 1: *(Optional) Apply Loop Unrolling.*

Step 2: *Hoist control variables to the outermost loop by using compiler information if loop invariant.*

Step 3: *Apply Partial Redundancy Elimination and Strength Reduction.*

Step 4: *Apply Dead Code Elimination.*

We call this algorithm *HPJOPT2* (**HPJ**ava **OPT**imization Level **2**)[4]. Applying Loop Unrolling is optional since it is only useful when a nested `overall` loop involves the pattern, `(i‘ + expr) % 2` such as in Figure 3. We don't treat Step 3 of HPJOPT2 as a part of PRE. It is feasible for control variables and control subscripts to be hoisted by applying PRE. But using information available to the compiler, we often know in advance they are loop invariant without applying PRE. Thus, without requiring PRE, the compiler hoists them if they are loop invariant.

In the section 5, we will see the performance of HPJava adopting HPJOPT2 optimization strategies.

5 Experiments

5.1 Node Performance

We showed in a previous publication [20] that HPJava individual node performance is quite acceptable, and Java itself can get 70 – 75% of the performance of C and Fortran.

The "direct" matrix multiplication algorithm in Figure 1 is relatively easy and potentially efficient since the operand arrays have carefully chosen replicated/collapsed distributions. Figure

[4] In later benchmarks we will take PRE alone as our "Level 1" optimization.

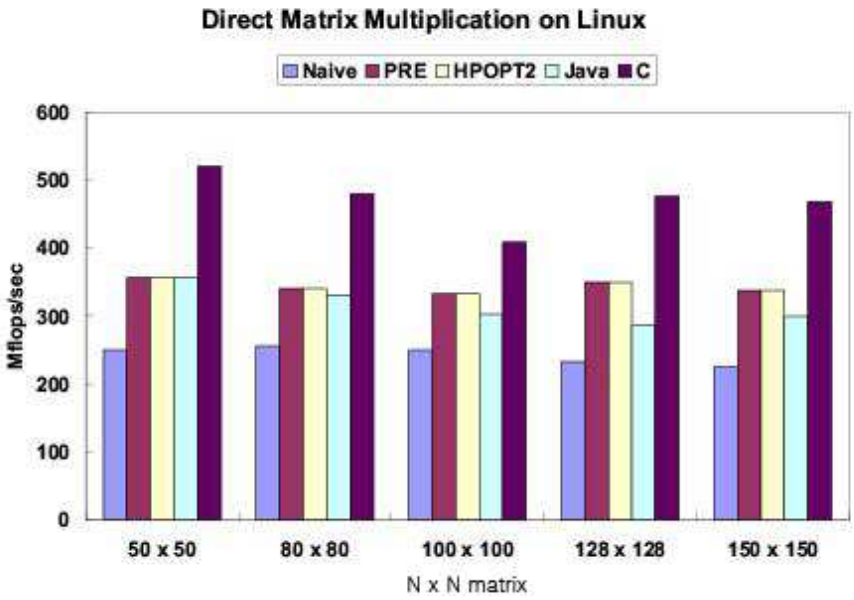

Figure 6. Matrix multiplication on the Linux machine.

6 shows the performance of the direct matrix multiplication programs in `Mflops/sec` with the sizes of 50 × 50, 80 × 80, 100 × 100, 128 × 128, and 150 × 150 in HPJava, Java, and C on the Linux machine.

From Figure 6, we can see the significant benefit of applying PRE or HPJOPT2[5]. 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.

5.2 Laplace Equation with Red-Black Relaxation

First, we experiment with HPJava on 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 7 shows the result of five different versions (HPJava with HPJOPT2 optimization, HPJava with PRE optimization, HPJava with naive translation, Java, and C) of red-black relaxation of the two dimensional Laplace equation. After applying HPJOPT2 for the naive translation, the speedup of HPJava is 177% on a single processor and 138% on 8 processors.

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 flag for Java. For comparison, we have also completed experiments for sequential Java, Fortran and HPF version of the HPJava programs. For the HPF version of program, we use IBM XL HPF version 1.4 with *xlhpf95* compiler command and `-O3` and `-qhot` flag. And XL Fortran for AIX with `-O5` flag is used for Fortran version.

[5]We can see that HPJOPT2 has no advantage over simple PRE in this benchmark. The reason is the innermost loop for the direct matrix multiplication algorithm in HPJava is a "for" loop, i.e. "sequential" loop. This means the more specialized HPJOPT2 scheme has nothing to optimize.

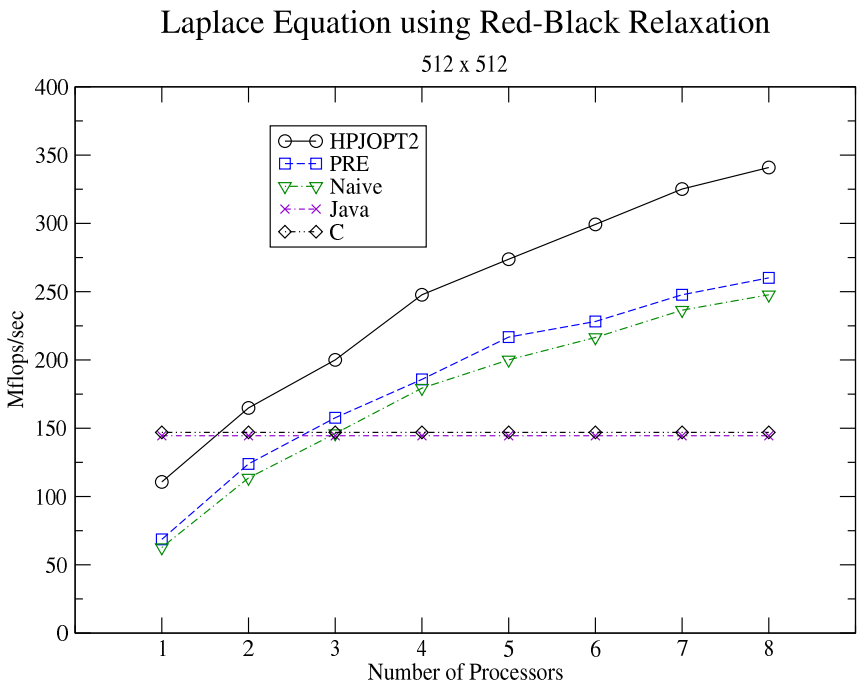

Figure 7. Laplace Equation with red-black relaxation with size of 512 x 512 on SMP.

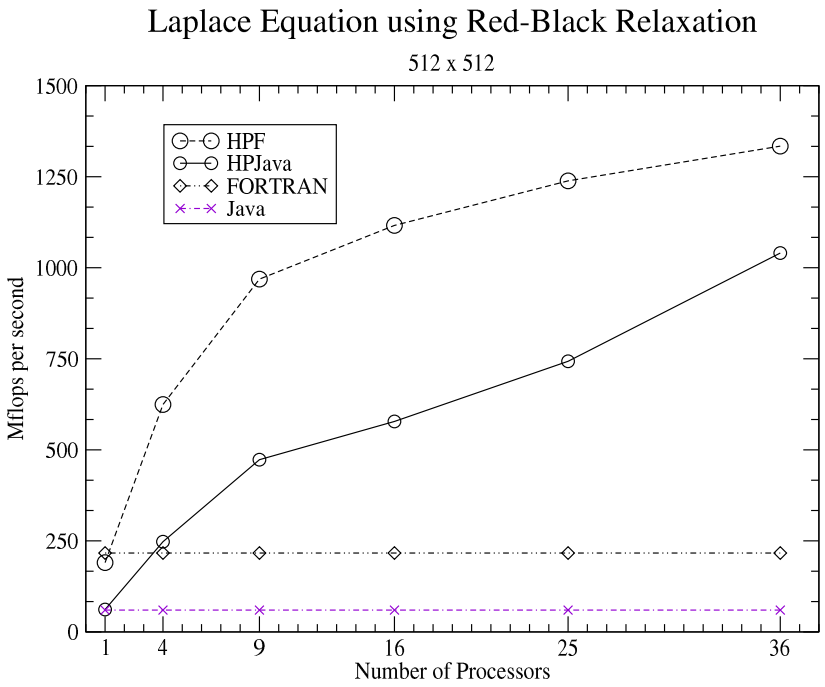

Figure 8. Laplace Equation with red-black relaxation with size of 512 x 512 on IBM SP3.

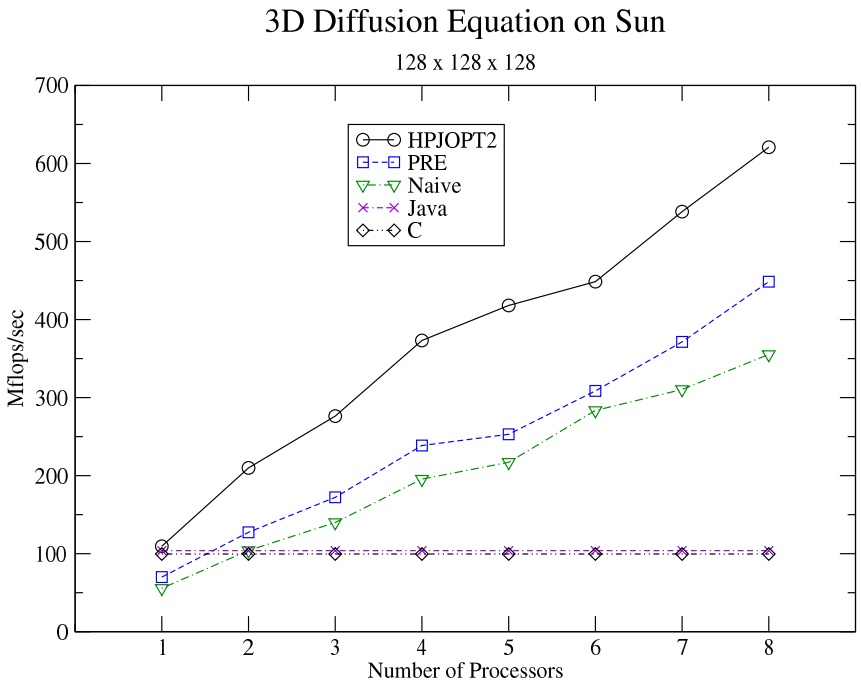

Figure 9. Three dimensional Diffusion equation with size of 128 x 128 x 128 on SMP.

Figure 8 shows the results from four different versions (HPJava, sequential Java, HPF and Fortran) of red-black relaxation for the two dimensional Laplace equation with size of 512 by 512. In our runs parallel HPJava can out-perform sequential Java by up to 17 times. On 36 processors HPJava can get about 78% of the performance of HPF. This is satisfactory performance for the initial benchmark result. Scaling behavior of HPJava is slightly better than HPF. Probably this mainly reflects the low performance of a single Java node compared to Fortran[6].

5.3 3-Dimensional Diffusion Equation

We see similar and better behavior on a larger three dimensional Diffusion Equation benchmark (Figure 9 and 10). After applying HPJOPT2 for the naive translation, the speedup of HPJava is 192% on a single processor and 567% on 8 processors. In general we expect 3 dimensional problems will be more amenable to parallelism, because of the large problem size.

5.4 Discussion

In this section, we have explored the performance of the HPJava system on two machines using the efficient node codes previously discussed in [20]. In the results here, the HPJOP2 algorithm was applied systematically *by hand* to the code generated by the existing translator. The next stage of this work is to include this algorithm in the translator.

The speedup of each HPJava application is satisfactory even with expensive communication methods such as `Adlib.writeHalo()` and `Adlib.sumDim()`. Moreover, performances on both machines show consistent behavior similar to that seen on the Linux machine. One machine does not have a big advantage over others. Performance of HPJava is good on all machines we have

[6] We do not believe that the current communication library of HPJava is faster than the HPF library because our communication library is built on top of the portability 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.

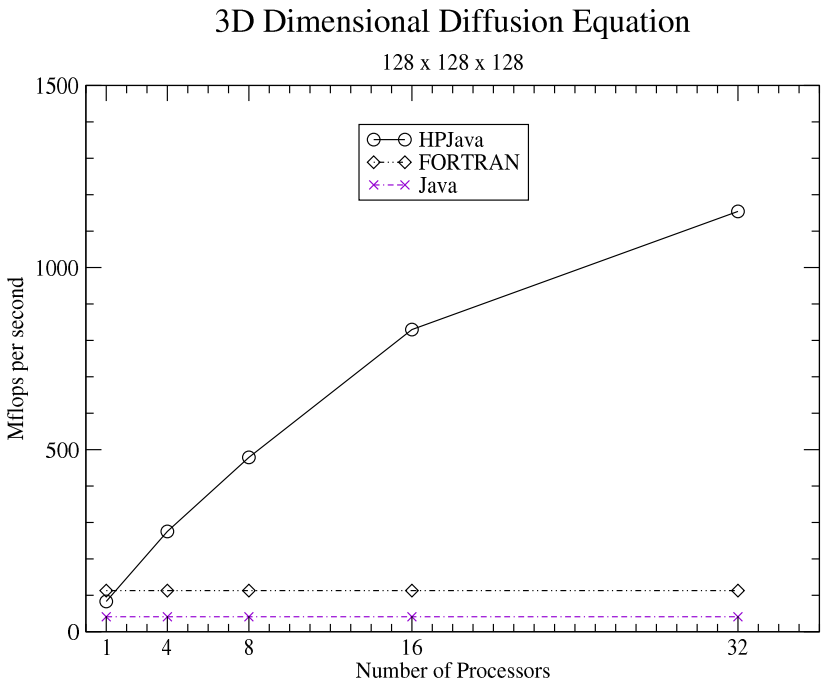

Figure 10. Three dimensional Diffusion equation with size of 128 x 128 x 128 on IBM SP3.

benchmarked. HPJava has an advantage over some systems because performance of HPJava on Linux machines, shared memory machines, and distributed memory machines are consistent and promising. Thus, we hope that HPJava has a promising future, and can be used anywhere to achieve not only high-performance parallel computing but also grid-enabled applications.

6 Related Works

6.1 Co-Array Fortran

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 [22]. 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.

6.2 ZPL

ZPL [25] is an array programming language designed from first principles for fast execution on both sequential and parallel computers for scientific and engineering computation. ZPL has an idea of performing computations over a region, or set of indices. Within a compound statement

prefixed by a region specifier, aligned elements of arrays distributed over the same region can be accessed. This idea has similarities to our `overall` construct. In ZPL, parallelism and communication are more implicit than in HPJava. The connection between ZPL programming and SPMD programming is not explicit. While there are certainly attractions to the more abstract point of view, HPJava deliberately provides lower-level access to the parallel machine.

6.3 JavaParty

JavaParty [23] allows easy ports of multi-threaded Java programs to distributed environments such as clusters. Regular Java already supports parallel applications with threads and synchronization mechanisms. While multi-threaded Java programs are limited to a single address space, JavaParty extends the capabilities of Java to distributed computing environments. It adds remote objects to Java purely by declaration, avoiding disadvantages of explicit socket communication and the programming overhead of RMI.

JavaParty is for parallel cluster programming in Java. Because the only extension is the `remote` keyword, it is quite simple and easy to use. Conversely, HPJava provides lower-level access to the parallel machine. Compared to the HPJava system, the basic approach of JavaParty, remote objects with RMI hooks, might become a bottleneck because of unavoidable overhead of RMI.

6.4 Timber

Timber [26] is a Java-based programming language for semi-automatic array-parallel programming, in particular for programming array-based applications. The language has been designed as part of the Automap project, in which a compiler and run-time system are being developed for distributed-memory systems. Apart from a few minor modifications, Timber is still a superset of Java.

Like HPJava, Timber introduces multidimensional arrays, array sections, and a parallel loop. They have some similarities in syntax, but semantically Timber is very different to HPJava. Although Timber supports parallel operations such as `each`, `foreach` constructs, it maps less directly to massively parallel distributed memory computing. Timber does not include HPF-like multiarrays, or support low-level access to the parallel machine. For high-performance, Timber chose C++ as its target language. But in the meantime Java performance has been improved greatly, while C++ remains less portable and secure.

6.5 Titanium

Titanium [27] is another Java-based language (not a strict extension of Java) for high-performance computing. Its compiler translates Titanium into C. Moreover, it is based on a parallel SPMD model of computation.

Titanium is originally designed for high-performance computing on both distributed memory and shared memory architectures. (Support for SMPs is not based on Java threads.) The system is a Java-based language, but, its translator finally generates C-based codes for perceived performance reasons. Titanium does not provide any special support for distributed arrays, and the programming style is quite different to HPJava.

6.6 GrADS

In grid computing, the *GrADS Project* [3] aims to simplify distributed computing in the same way that the World Wide Web simplified information sharing over the Internet. To that end, the project is working with scientific users to develop, execute, and tune applications on the Grid. That

is, the main interest of the GrADS is in application technologies that make it easy to construct and execute applications with reliable performance in the constantly-changing environment of the Grid. The main goals of the GrADS project are in constructing libraries for Grid-ware components and developing innovative new science and engineering applications to run effectively in Grid Environments.

To achieve these goals, the GrADS project focuses on prototype forms of important scientific applications, together with programming systems and problem-solving environments to support the development of Grid applications by end users. It emphasizes execution environments that dynamically match Grid applications to available resources for consistent performance, hardware and software for experimentation with GrADS programs. This emphasis is distinguished from (though perhaps complementary with) our HPJava-based approach.

6.7 Discussion

One edge the HPJava system has compared to other systems is that HPJava not only is a Java extension but also is translated from a HPJava code to a pure Java byte code. Moreover, because HPJava is totally implemented in Java, it can be deployed any systems without any changes. Java is object-oriented and highly dynamic. That can be as valuable in scientific computing as in any other programming discipline. Moreover, it is evident that SPMD environments are successful in high-performance computing, although programming applications in SPMD-style is relatively difficult. Unlike other systems our HPspmd programming model targets SPMD environments, providing lower-level access to the parallel machines.

Many systems adopted implicit parallelism which have the benefit of simplicity for users. But this results in the difficulty of implementing the system. Some systems chose C, C++, and Fortran as their target language for high-performance. But, as mentioned earlier, Java is now a competitive language for scientific computing. The choice of C or C++ makes the systems less portable and secure in the modern computing environment.

7 Conclusions

The first fully functional HPJava is operational and can be downloaded from [14]. The system fully supports the Java Language Specification [17], and has tested and debugged against the HPJava test suites and *jacks* [15], a Java Automated Compiler Killing Suite. 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. Also, a Computational Fluid Dynamics (CFD) application for fluid flow problems[7] has been ported to HPJava. An applet version of this application can be viewed at `www.hpjava.org`.

There are two main parts to the software. The HPJava development kit, *hpjdk* contains the HPJava compiler and an implementation of the high-level communication library, *Adlib*. The only prerequisite for installing hpjdk is a standard Java development platform, like the one freely available for several operating systems from Sun Microsystems. The installation of hpjdk is very straightforward because it is a pure Java package. Sequential HPJava programs can immediately be run using the standard `java` command. Parallel HPJava programs can also be run with the `java` command, provided they follow a *multithreaded model*.

To distribute parallel HPJava programs across networks of host computers, or run them on

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

supported distributed-memory parallel computers, one should install the second HPJava package—*mpiJava*. A prerequisite for installing mpiJava is the availability of an *MPI* installation on your system.

The main purpose of this paper was to investigate whether our library-based HPspmd Programming Model can be efficiently adapted into programming support for high-performance *grid-enabled* applications. Through the experiments, we proved that HPJava has quite acceptable performance on "scientific" algorithms, which play very important roles in high-performance grid-enabled applications, including "search engines" and "parameter searching".

Although not illustrated here, an interesting Grid application where HPJava may be adoptable is "coordinating" the execution and information flow between multiple "web services" where each web service has WSDL style interface and some high level information describing capabilities. In the near future, HPJava will be used as middleware to support "complexity scripts" in the project called "BioComplexity Grid Environment" at Indiana University.

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] BioComplexity Institue Home Page. http://biocomplexity.indiana.edu/.

[5] B. Carpenter, G. Fox, H.-K. Lee, and S. B. Lim. Translation Schemes for the HPJava Parallel Programming Language. *Lecture Notes in Computer Science*, 2624(2):18–32, 2003.

[6] 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.

[7] B. Carpenter, H.-K. Lee, S. Lim, G. Fox, and G. Zhang. *Parallel Programming in HPJava.* Draft, 2003. http://www.hpjava.org.

[8] 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. http://www.hpjava.org.

[9] 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.

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

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

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

[13] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.

[14] HPJava Home Page. http://www.hpjava.org.

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

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

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

[18] 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.

[19] C. Koelbel, D. Loveman, R. Schreiber, J. G.L. Steel, , and M. Zosel. The High Performance Fortran Handbook. *MIT Press*, 1994.

[20] 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.

[21] 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. http://www.hpjava.org.

[22] R. Numrich and J. Steidel. F- -: A simple parallel extension to Fortran 90. *SIAM News*, page 30, 1997. http://www.co-array.org/welcome.htm.

[23] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225 – 1242, 1997.

[24] 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.

[25] L. Snyder. A ZPL Programming Guide. Technical report, University of Washington, May 1997. http://www.cs.washington.edu/research/projects/zpl.

[26] K. van Reeuwijk, A. J. C. van Gemund, and H. J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, 1997.

[27] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11-13):825 – 836, 1998.