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

Han-Ku Lee

2035 Hughes Hall
Old Dominin 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

We review the authors' HPspmd Programming Model [1] *as a contribution towards the 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 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

We argued that HPJava should ultimately provide acceptable performance to make it a practical tool for HPC in [6, 7]. 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 in order to show a promising future of HPJava

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

which can be used anywhere to achieve not only high-performance parallel computing but also grid-enabled applications.

2 High-Performance Grid-Enabled Environments

2.1 Grid Computing

Grid computing environments 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 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. Of course, substantially powerful Grids can be established using high-performance networking, computing, and programming support regardless of the location resources and users.

What then will be the biggest potential issues 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? *High-performance* is one possible answer since a slow system which has a clever motivation is useless. Another answer could be the thirst for *grid-enabled applications*, hiding the "heterogeneity" and "complexity" of grid environments without losing performance.

Today, grid-enabled application programmers write applications in what, in effect, an assembly language: sometimes using explicit calls to the Internet Protocol's User Datagram Protocol (UDP) or Transmission Control Protocol (TCP), explicit or no management of failure, hard-coded configuration decisions for specific computing systems. We are a bit far from portable, efficient, high-level languages.

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. In parallel programming, distributed arrays, message-passing, threads, condition variables, and so on. Thus, using each model's significant charateristics, sequential and parallel program must maximize their performance and correctness.

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 our 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 [2] 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 (e.g. Java).

The HPJava language has quite acceptable performance on scientific and engineering (matrix) algorithms, which play very important roles in high-performance grid-enabled applications such as "search engines" and "parameter searching". Moreover, the most interesting Grid problem where HPJava is 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. It can even help parallel computing by specifying compiler hints. In the near future, HPJava will be mainly used as a middleware to support "complexity scripts" in the project, called "BioComplexity Grid Environment" at Indiana University.

Thus, 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*. The 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, we extend 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.

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, $\texttt{P} \times \texttt{P}$ 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

[2] currently our HPJava project team provides a Java implementation of MPI, called *mpiJava*

```
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.

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 *multiarray*. The HPJava multiarrays are divided into two parts: *distributed arrays* and sequential *true multidimensional arrays*. A distributed arrays is a *collective multiarray* 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 true multi-dimensional array is simialr to 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 asterisk, `*`, 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 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.

```
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.

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 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 [6] 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.

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

```
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.

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 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 discuss efficient compilation strategies for HPJava. The HPJava compilation system consists of four 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.

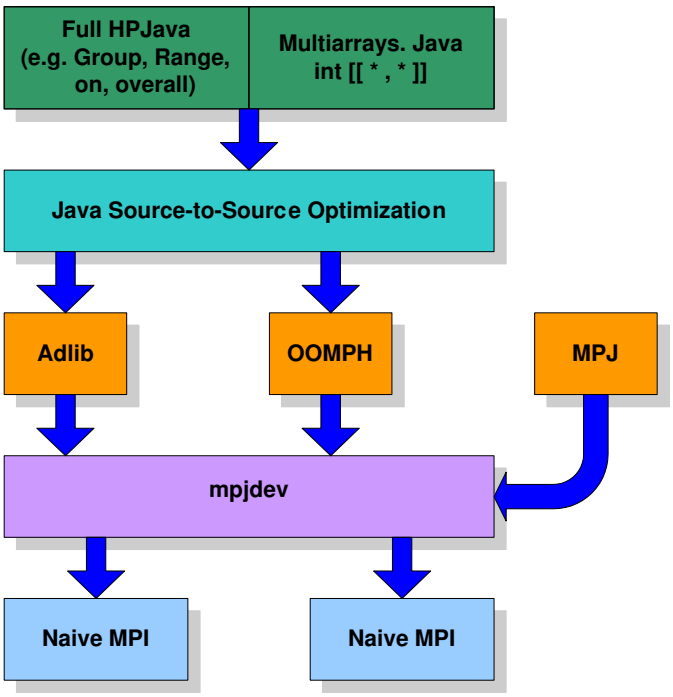

Figure 4. HPJava Architecture.

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 has been one of the most time-consuming parts during the implementation of the whole HPJava compiler.

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 missions. It collects all of the rest 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 newly introduced HPJava constructs, multiarrays, etc.

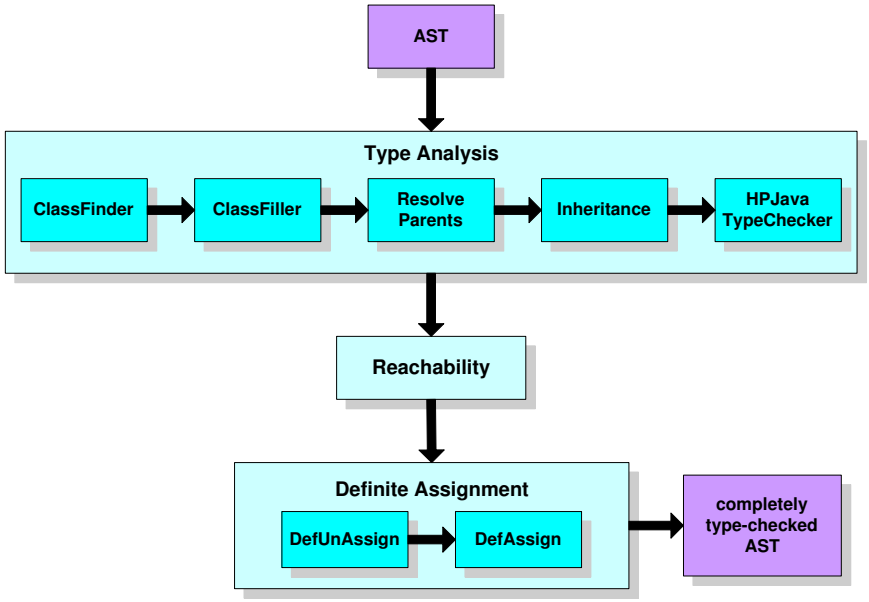

Figure 5. The architecture of HPJava Front-End.

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.

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 JLS.

The three phases are basically inspired by carefully organizing "The Java Language Specification Second Edition" (JLS) [16]. The current HPJava type-checker system only supports exactly what JLS says. However, The Java language itself keeps evolving in a slow manner. For example, Sun's SDK Java compiler supports some changes to JLS. For the moment, the HPJava type-checker system will not be changed until the new edition of JLS is released. 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 HPJava 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 a basic translation scheme easy 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. The basic translation schema will be applied to the pre-translated HPJava programs.

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* [6], and translation scheme [4].

4.3 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. Moreover, when an `overall` construct is translated, the naive translation scheme generates 4 control variables outside the loop (refer to [6]). A control variable is often a dead code, and is partially redundant for the outter `overall` for nested `overall`s.

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.

4.3.1 Partial Redundancy Elimination

Partial Redundancy Elimination (PRE) [17] 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.

PRE should be applied to a general or *Static Single Assignment* (SSA) [9] formed data flow graph after adding *landing pads* [22], representing entry to the loop from outside.

PRE is a complicated algorithm to understand and to implement, compared with other well-known optimization alogithms. 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.*

Since PRE is a global and powerful optimization algorithm to eliminate partial redundancy, (especially to get rid of loop-invariants generated by the current HPJava translator), we expect the optimized code to be very competitive and faster than the naively translated one.

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. Thus, it could be a very competitive candidate to optimize the current HPJava compiler because of the complexity of the associated 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. We call this technique, *Loop Unrolling* (LU) [2].

Our experiments [18] didn't 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`. This will be discussed again in section **??**. 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 will adopt 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 will call this algorithm *HPJOPT2* (**HPJ**ava **OPT**imization Level **2**)[3] . Applying Loop Unrolling is optional since it is only useful when a nested `overall` loop involves the pattern, `(i‘ + expr) % 2` such as 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.

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

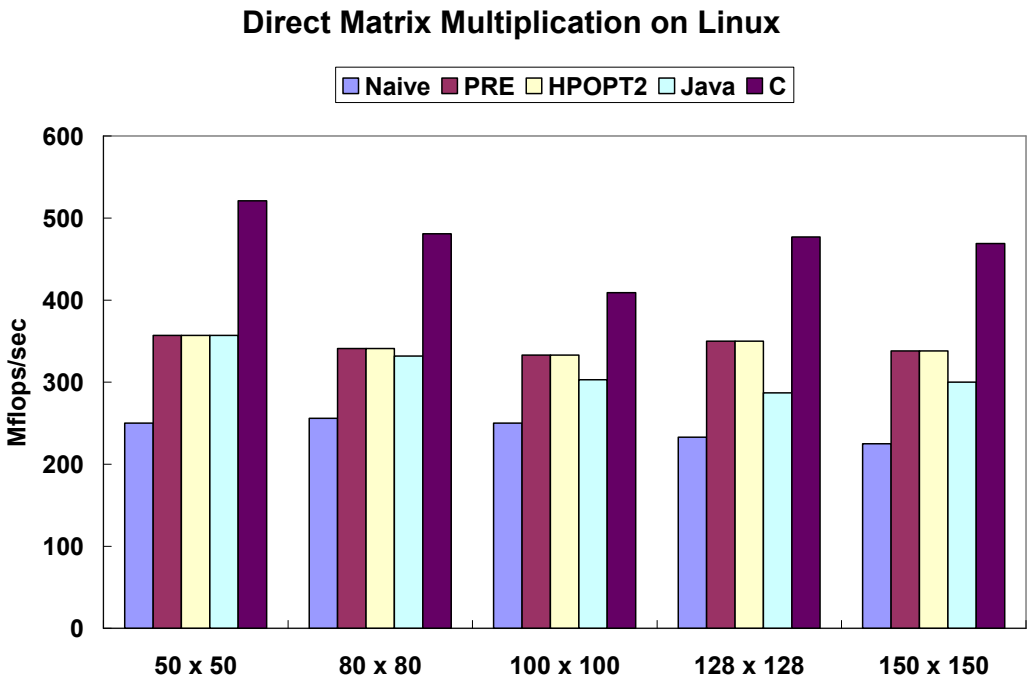

Figure 6. Matrix multiplication on the Linux machine.

Thus, applying `HPJOPT2` 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 see the performance of HPJava adopting `HPJOPT2` optimization strategies.

5 Experiments

5.1 Node Performance

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].

The "direct" matrix multiplication algorithm 1 is relatively easier and potentially more efficient since the operand arrays have carefully chosen replicated/collapsed distributions. Figure 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 dramatic result after applying `HPJOPT2` [4]. 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.

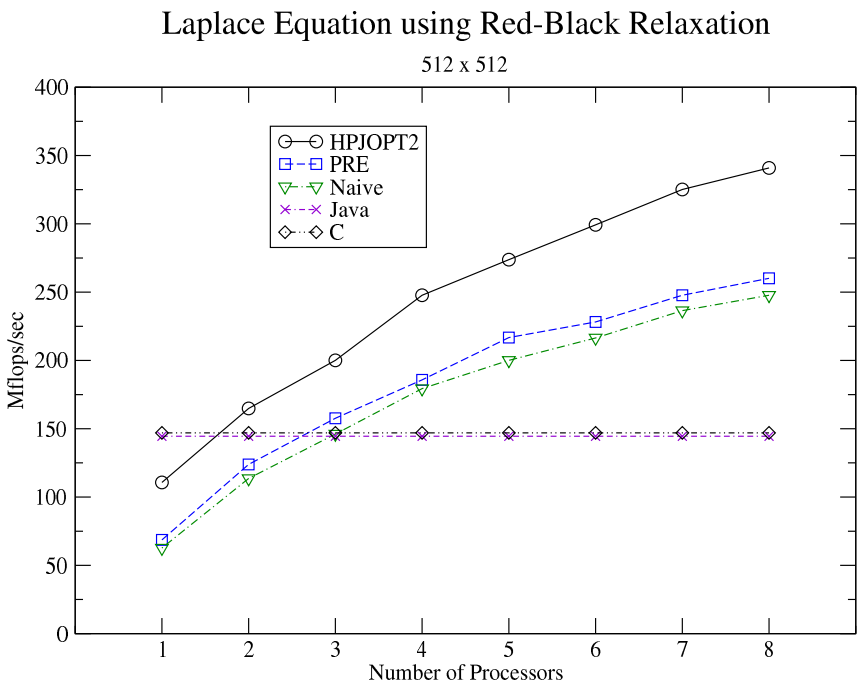

Figure 7. Laplace Equation with red-black relaxation with size of 512 x 512 on SMP.

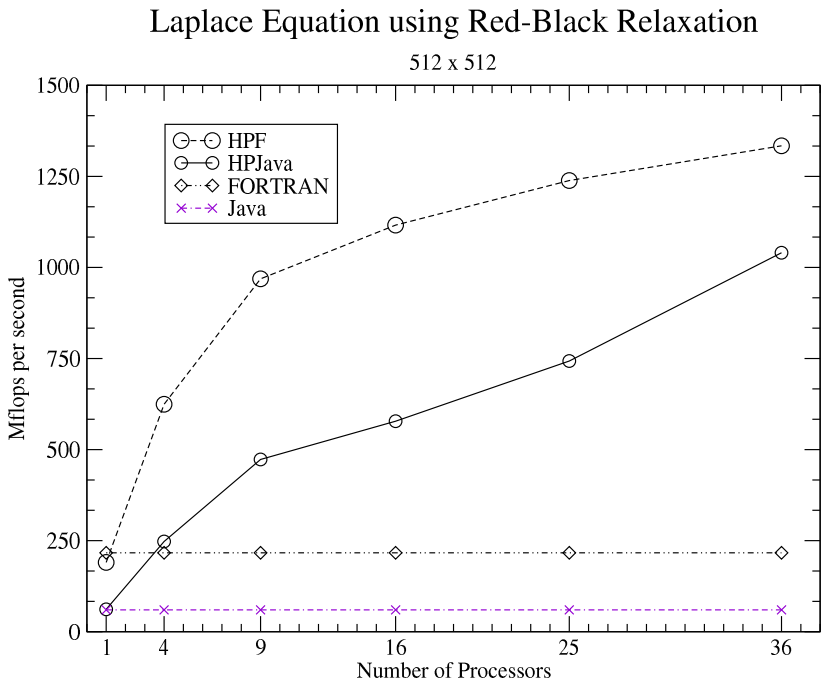

Figure 8. Laplace Equation with red-black relaxation with size of 512 x 512 on IBM SP3.

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 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 8 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 out-perform 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.

5.3 3-Dimensional Diffusion Equation

We see similar and better behavior on large size of 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 both machines using the efficient node codes we have benchmarked in [18].

The speedup of each HPJava application is very satisfactory even with expensive run-time communication libraries such as `Adlib.writeHalo()` and `Adlib.sumDim()`. Moreover, performances on both machines shows consistent and similar behaviour as we have seen on the Linux machine. One machine doesn't have a big advantage over others. Performance of HPJava is good on all machines we have benchmarked.

When program architects design and build high-performance computing environments, they have to think about what system they should choose to build and deploy the environments. There rarely exists machine-independent software, and performance on each machine is quite inconsistent. 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.

[4]We can see that HPJOPT2 has no advantage over simple PRE in the Figure 6. The reason is the innermost loop for the direct matrix multiplication algorithm in HPJava is "for" loop, i.e. "sequential" loop. This means HPJOPT2 scheme has nothing to optimize (e.g. hoisting control variables to the outermost overall construct) for this algorithm.

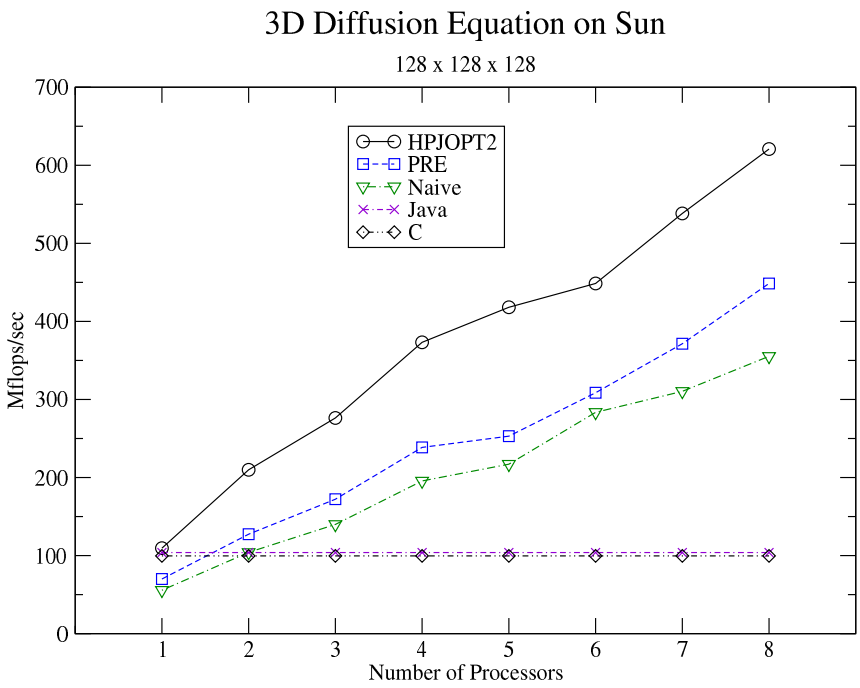

Figure 9. Three dimensional Diffusion equation with size of 128 x 128 x 128 on SMP.

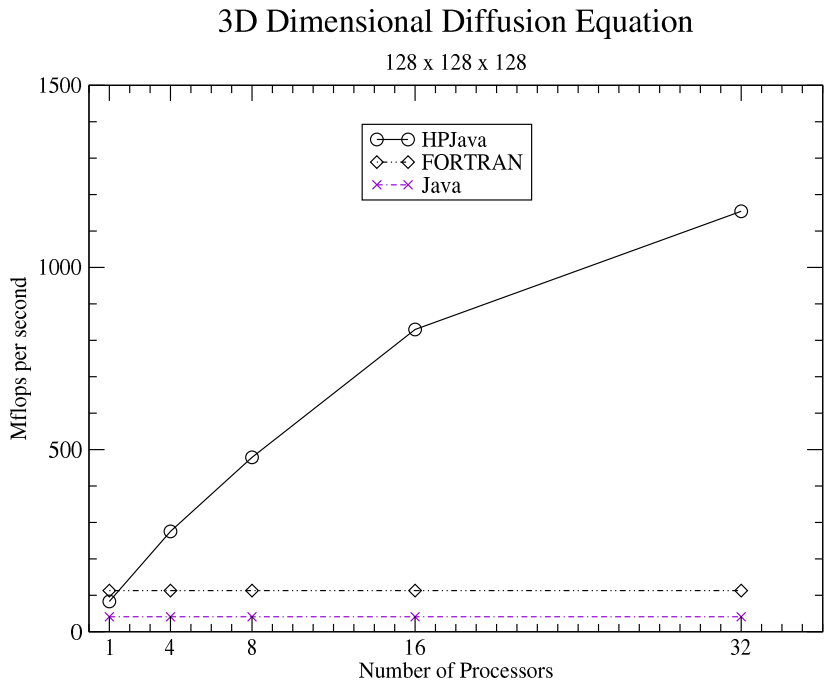

Figure 10. Three dimensional Diffusion equation with size of 128 x 128 x 128 on IBM SP3.

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 [20], 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 [23] 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 [21] 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. It has important contribution to research on Java-based parallel computing. Moreover, because the only extension is the `remote` keyword, it is systematically simple and easy to be used and to be implemented. However, 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 its bottleneck because of unavoidable overhead of RMI and limited evidence that the remote procedure call approach is good for SPMD programming.

6.4 Timber

Timber [24] 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 is difficult to say Timber targets massively parallel distributed memory computing, using HPF-like multiarrays, and supports lower-level access to the parallel machines. Moreover, for high-performance, Timber chose C++ as its target language. But it becomes its own bottleneck since Java has been improved and C++ is less portable and secure in the modern computing environment.

6.5 Titanium

Titanium [25] 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 performance reason. In the current stage development of Java, performance has almost caught up with C and Fortran, as discussed in section 5.1. Translating to C-based codes is no longer necessarily an advantage in these days. In addition, it doesn't 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] 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.

6.7 Discussion

In this section we have reviewed and compared some parallel language and library projects to our HPJava system. Compared against others, each language and library has its advantages and disadvantages.

A biggest edge the HPJava system has against 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, despite that programming applications in SPMD-style is relatively difficult. Our HPspmd programming model targets SPMD environments with lower-level access to the parallel machines unlike other systems.

Java has potential to displace established scientific programming languages. According to this argument, HPJava, an HPspmd programming model, has a high advantage over other parallel languages.

Many systems adopted implicit parallelism and 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, mentioned earlier, Java is a very competitive language for

scientific computing. The choice of C or C++ makes the systems less portable and secure in the modern computing environment.

Thus, because of the *popularity*, *portability*, *high-performance* of Java and SPMD-style programming, the HPJava system gets many advantages over other systems.

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 the experiments, we proved that HPJava has quite acceptable performance on scientific and engineering (matrix) algorithms, which play very important roles in high-performance grid-enabled applications such as "search engines" and "parameter searching". Moreover, the most interesting Grid problem where HPJava is 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. It can even help parallel computing by specifying compiler hints. In the near future, HPJava will be mainly used as a middleware to support "complexity scripts" in the project, called "BioComplexity Grid Environment" at Indiana University.

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). Also, Computational Fluid Dynamics research application for fluid flow problems, (CFD) [5] has been ported to HPJava (it is about 1300 lines). An applet version of this application can be viewed at `www.hpjava.org`.

There are two 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 using the standard `java` command can be immediately run. Parallel HPJava programs can also be run with the `java` command, provided they follow the *multithreaded model*.

To distribute parallel HPJava programs across networks of host computers, or run them on supported distributed-memory parallel computers, you also need to install a second HPJava package—*mpiJava*. A prerequisite for installing mpiJava is the availability of an *MPI* installation on your system.

hpjdk contains a large number of programming examples. Larger examples appear in figures, and many smaller code fragments appear in the running text. All non-trivial examples have been checked using the hpjdk 1.0 compiler. Nearly all examples are available as working source code in the hpjdk release package, under the directory `hpjdk/examples/PPHPJ/`. Figure 11 is running an HPJava Swing program, `Wolf.hpj`.

[5] 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.

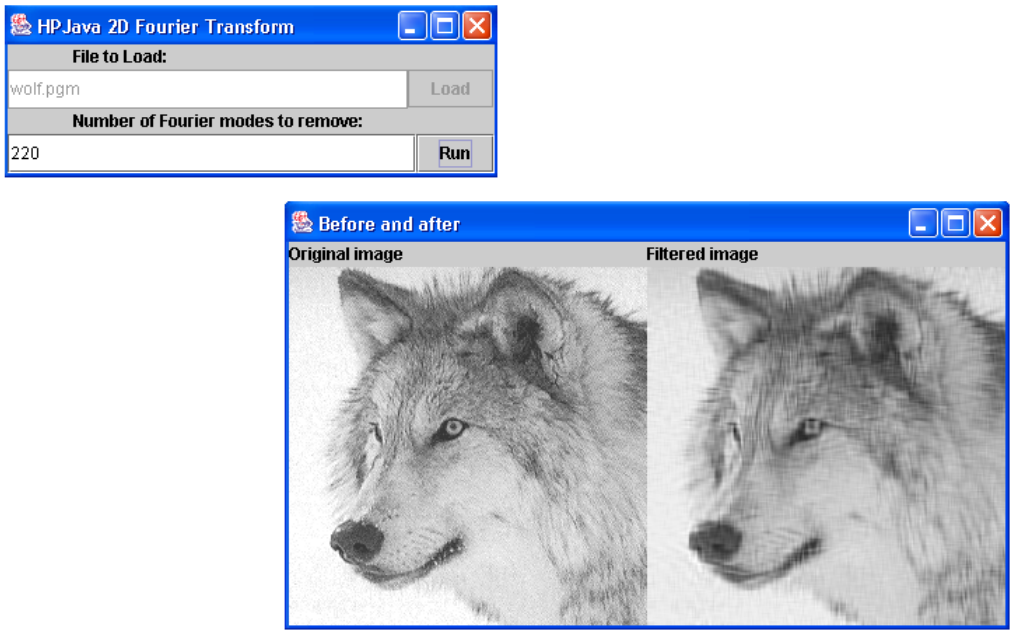

Figure 11. Running an HPJava Swing program, `Wolf.hpj`.

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 *14th 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, H.-K. Lee, S. Lim, G. Fox, and G. Zhang. *Parallel Programming in HPJava*. Draft, 2003. http://www.hpjava.org.

[7] 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.

[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. http://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. http://www.hpjava.org.
[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. http://www.hpjava.org.
[20] 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.
[21] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225 – 1242, 1997.
[22] 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.
[23] L. Snyder. A ZPL Programming Guide. Technical report, University of Washington, May 1997. http://www.cs.washington.edu/research/projects/zpl.
[24] 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.
[25] 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.