An evaluation of Java for numerical computing

Brian Blount * and Siddhartha Chatterjee
Department of Computer Science, The University of North Carolina, Chapel Hill, NC 27599-3175, USA
Tel.: +1 919 962 1953; Fax: +1 919 962 1799;
E-mail: {bblount,sc}@cs.unc.edu

This paper describes the design and implementation of high performance numerical software in Java. Our primary goals are to characterize the performance of object-oriented numerical software written in Java and to investigate whether Java is a suitable language for such endeavors. We have implemented JLAPACK, a subset of the LAPACK library in Java. LAPACK is a high-performance Fortran 77 library used to solve common linear algebra problems. JLAPACK is an object-oriented library, using encapsulation, inheritance, and exception handling. It performs within a factor of four of the optimized Fortran version for certain platforms and test cases. When used with the native BLAS library, JLAPACK performs comparably with the Fortran version using the native BLAS library. We conclude that high-performance numerical software could be written in Java if a handful of concerns about language features and compilation strategies are adequately addressed.

1. Introduction

The Java programming language [4] achieved rapid success due to several features key to the language. Java bytecodes are portable, which means that programs can be run on any machine that has an implementation of the Java Virtual Machine (JVM). Java provides garbage collection, freeing programmers from concerns about memory management and memory leaks. The language contains no pointers and dynamically checks array accesses, which help avoid common bugs in C programs. Because of such reasons, Java is establishing itself as a language of choice for many software developers.

Java is attractive to the scientific computing community for the very same reasons. However, several factors limit Java's inroads into this community. First, the performance of Java has been a continuous source of concern. Many of the attractive features of Java caused early interpreted versions of the JVM to perform poorly, especially when compared with compiled languages like Fortran and C. Second, the absence of a primitive complex type presents another obstacle, as many numeric codes make extensive use of complex numbers. Finally, several language features that make numeric codes less cumbersome to write, such as operator overloading and parametric polymorphism, are absent in Java.

Despite these issues, we believe that Java may be suitable for writing high-performance numerical software. The problems discussed above can be partially circumvented by careful programming techniques. Furthermore, certain language features, such as primitive complex types, may be included in future versions of Java. To test our hypothesis that good performance can be achieved in Java, we designed and implemented JLAPACK, a proof-of-concept version of LAPACK [3] in Java. LAPACK is a high-performance Fortran 77 library that solves common linear algebra problems. This library is well-suited for our study for several reasons: it is a standard library accepted and used by the scientific community; it is used to solve common and useful problems; and it is highly optimized, giving us a hard performance bound.

Our implementation of JLAPACK follows the Fortran version closely in spirit and structure. However, we did not write Fortran-style code in Java. JLAPACK employs object-oriented techniques such as inheritance, dynamic dispatch, and exception handling. We use classes to represent vectors, matrices, and other objects. We use exceptions to perform error handling. For performance analysis, we executed our model using a fully compliant JVM, with bounds checking and garbage collection enabled. JLAPACK performs within a factor of four of the optimized Fortran version for certain platforms and test cases.

The rest of the paper is organized as follows. Section 2 discusses the LAPACK library in more depth,

*Corresponding author.

Scientific Programming 7 (1999) 97–110
ISSN 1058-9244 / $8.00 © 1999, IOS Press. All rights reserved

and lists the portions of the library we have implemented in JLAPACK. Section 3 describes the design of JLAPACK. Section 4 presents the performance results of JLAPACK. Section 5 describes related work. Finally, Section 6 presents conclusions and directions for future work.

2. LAPACK

LAPACK [3] is a library of Fortran 77 routines for common linear algebra problems. The library contains driver routines, which are used to solve standard types of problems, and auxiliary routines, which are used to perform subtasks and provide common functionality. LAPACK provides driver routines for systems of linear equations, linear least square problems, eigenvalue problems, and singular value problems. Driver routines handle both real and complex numbers, with versions for both single and double precision representations. There are specialized routines for different types of matrices, such as banded matrices, tridiagonal matrices, and symmetric positive-definite matrices.

LAPACK uses the Basic Linear Algebra Subroutines (BLAS) [13–16,24,25] for many of its time-critical inner loops. Most vendors of high performance machines supply BLAS libraries with machine-specific optimizations, called *native BLAS*. Generic Fortran 77 BLAS code is available and is distributed with LAPACK. For JLAPACK, we provided two versions: one implemented in Java, and the other employing vender-supplied native BLAS. The latter version provides Java wrappers around the Fortran BLAS routines, using the `native` method call mechanism of Java. Bik and Gannon [7] have shown that native methods can be used to achieve good performance, and our findings support their results.

LAPACK uses block-oriented algorithms for many of its operations. A block-oriented algorithm is one that operates on blocks or submatrices of the original matrix. This provides more locality of reference and allows LAPACK to use Level 3 BLAS routines. The optimal block size varies based on both the problem size and on the architecture of the machine used. Both LAPACK and JLAPACK allow the block size to be set explicitly.

Two different types of driver routines are provided for solving systems of linear equations in LAPACK. One driver, the simple driver, solves the system $AX = B$ by factoring the coefficient matrix A and overwriting the right hand side B with the solution X. The other driver, the expert driver, provides additional functionality such as solving $A^T X = B$ or $A^H X = B$, estimating the condition number of A, and checking for near singularity. Because of time constraints, JLAPACK currently implements only the simple linear equation solver for general matrices (i.e., xGESV and the routines they require).

3. JLAPACK

JLAPACK and JBLAS are our Java implementations of the LAPACK and BLAS libraries, currently implementing the subset of the subroutines in both libraries that are used by the *simple general equation solver*. We follow the Fortran version in spirit and in structure, with every Java method corresponding to a Fortran subroutine. We retain the Fortran naming conventions, providing implementations for four data types: single precision real (S), double precision real (D), single precision complex (C), and double precision complex (Z).

Another project, the F2J [18] project, is also generating LAPACK and BLAS libraries in Java. They have developed a Fortran to Java translator, and are using this translator to transcribe the Fortran LAPACK source code into Java source code. This approach is very different from ours, as it does not take design issues into account when generating Java LAPACK code. We compare the performance of our version with the version generated by their translator in Section 5.5.

Several goals influenced the design of JLAPACK. Some of these goals are well established in object-oriented design [20]; others are specific to Java.

(1) Encapsulate all of the information specifying a vector or matrix into a class. This information fits into two categories: the data and its shape. This information should be kept orthogonal.

(2) Store matrix data in a one-dimensional array. The reasons for this are twofold. First, two-dimensional arrays in Java are not guaranteed to be contiguous in memory, so a one-dimensional array provides more locality of reference. The second reason involves bounds checking. Accessing an element in a two-dimensional array requires bounds checks on both indices, doubling this overhead.

(3) Allow matrices and vectors to share data. A vector object that represents a column of a matrix should be able to use the same data as the matrix itself.

(4) Limit the number of constructor calls. Excessive memory allocation is usually a large source of overhead in naive object-oriented programs. Gratuitous memory allocation should be avoided as much as possible.

In the rest of this section, we discuss in detail the design of JLAPACK and how it achieves these goals.

3.1. The components of JLAPACK

Our design contains three separate components: the `JLASTRUCT` package, the `JBLAS` package, and the `JLAPACK` package.

(1) The `JLASTRUCT` package supplies the vector, matrix, and shape classes used by the library. These are discussed in detail in Section 3.2.
(2) The `JBLAS` package contains the BLAS library code. It contains four classes, one for each data type. Because there are no instance members in this class, all the methods are static. Each method in the `JBLAS` classes corresponds to a subroutine in the BLAS library.
(3) The `JLAPACK` package contains the code for the general equation solvers. Like the `JBLAS` package, there are separate classes for the four data types and all methods in these classes are static. Again, each method in the `JLAPACK` classes corresponds to a subroutine in the Fortran 77 version of LAPACK.

3.2. The Vector, Matrix, and Shape classes

In Fortran 77, information about the shapes of vectors and matrices must be represented with multiple scalar variables, and be passed as extra arguments to every routine manipulating vectors and matrices. The vector and matrix classes in our design encapsulate all this information into the abstraction of *shape*. There are vector and matrix classes for each of the four data types.

The class `JLASTRUCT.Vector` implements two methods.

(1) **double eltAt(int i)**: This returns the ith element in the vector.
(2) **void assignAt(double val, int i)**: This stores val as the ith element of the vector.

The class `JLASTRUCT.Matrix` implements these methods.

(1) **double eltAt(int i, int j)**: This returns the element at location (i, j) of the matrix.
(2) **void assignAt(double val, int i, int j)**: This stores val at location (i, j) of the matrix.
(3) **void colAt(int i, Vector v)**: This aliases the vector v to the ith column of the matrix.
(4) **void rowAt(int i, Vector v)**: This aliases the vector v to the ith row of the matrix.
(5) **void submatrix(int i, int j, int r, int c, Matrix m)**: This aliases the matrix m to the submatrix of size (r, c) starting at location (i, j).

These classes contain two members: *data* and *shape*. The data member is a one-dimensional array of the appropriate type that is guaranteed to contain all the elements of the vector or the matrix. Note that the elements of the vector or matrix do not have to be dense within this array. The shape member of a vector is of type `JLASTRUCT.VShape`, and the shape member of a matrix is of type `JLASTRUCT.MShape`. The classes `JLASTRUCT.VShape` and `JLASTRUCT.MShape` are subclasses of the abstract class `JLASTRUCT.Shape`. The shape object defines the layout of the vector or matrix elements in the data array.

An object of type `JLASTRUCT.VShape` contains the following members.

(1) **start:** The index in *data* of the first element of the vector.
(2) **len:** The number of elements in the vector.
(3) **inc:** The step size in *data* between any two consecutive elements of the vector.

Thus, element i of a vector resides in slot j of its data array, where

$$j = start + i * inc. \tag{1}$$

Note that elements of a vector are evenly spaced in the data array.

An object of type `JLASTRUCT.MShape` contains the following members.

(1) **start:** The index in *data* of the first element of the matrix.
(2) **rows:** The number of rows in the matrix.
(3) **cols:** The number of columns in the matrix.
(4) **ld:** The leading dimension of the array. This is the distance in *data* between the first elements in two consecutive columns.

Therefore, element (i, j) of a matrix resides in location k of its data array, where

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data:	1.2	3.4	6.7	2.6	9.4	8.6	2.7	7.3	5.5	9.9	6.7	3.6	3.9	2.1	4.3	1.9

Operation:	**Shape:**	**Object:**
A = new DMatrix(data, 4, 4);	A.data = data A.shape.start = 0 A.shape.rows = 4 A.shape.cols = 4 A.shape.ld = 4	1.2 9.4 5.5 3.9 3.4 8.6 9.9 2.1 6.7 2.7 6.7 4.3 2.6 7.3 3.6 1.9
B = new DMatrix(); A.submatrix(1, 1, 2, 2, B);	B.data = data B.shape.start = 5 B.shape.rows = 2 B.shape.cols = 2 B.shape.ld = 4	8.6 9.9 2.7 6.7
C = new DVector(); B.rowAt(1, C);	C.data = data C.shape.start = 6 C.shape.inc = 4 C.shape.len = 2	2.7 6.7

Fig. 1. Sharing of data among multiple matrices and vectors.

$$k = start + ld * j + i. \tag{2}$$

Note that the matrix is stored in column-major order and is addressed with zero-based indexing of arrays. This fits the Fortran model, allowing JLAPACK to use the same optimizations as the Fortran version and enabling native BLAS to be incorporated easily.

This implementation allows objects to share data arrays. The example in Fig. 1 demonstrates how this may occur. The 4×4 matrix A uses all 16 elements of its data array. Matrix B is assigned to be a submatrix of A. It shares the same data object as A, but only uses 4 elements of the array. Vector C represents one row of matrix B. Again, it shares the data object with A and B, but only uses 2 elements. Note how the shape parameters specify exactly where the data is stored.

The ability to share member objects improves the performance of methods used to obtain rows, columns, and sub-matrices of matrices. We will use the $colAt()$ method as an example, as its implementation applies to the other two. A naive implementation of this method would allocate new memory for the vector and new memory for its shape. Instead, the $colAt()$ takes as a parameter a vector that has already been allocated. Then, all the method does is supply the vector's data member (by giving it a reference to its own data), and update its shape object. This approach eliminates unnecessary copying of data elements and allows reuse of storage for temporary vectors and matrices.

Boisvert et al. [8] discuss an implementation for numerical libraries in Java that does not encapsulate vectors and matrices in classes. They use two-dimensional arrays to represent matrices, and store information describing the shape of vectors and matrices in local variables, similar to the Fortran version. This approach has a significant side effect: they require several versions of each vector operation. One version must handle the case where a vector is stored in a one-dimensional array, and another must handle the case where a vector is a column of a matrix, and is stored in a two-dimensional array. They also discuss versions for special cases, such as when a vector is dense within a one-dimensional array. They claim [8, p. 41].

> If we are to provide the same level of functionality as the Fortran and C BLAS then we must provide *several* versions of each vector operation.

While this may be true of *implementations* of BLAS primitives, this should not clutter up the *interface* visible to the programmer. Our shape abstraction unifies

and encapsulates these various cases. For efficiency, an implementation can still provide specialized routines for common cases.

3.3. *Limiting constructor calls*

Excessive object creation is often a substantial source of performance loss in object-oriented programs. Therefore, we use a technique (similar to that described by Dingle and Hildebrandt [12]) that limits the number of temporary vectors and matrices created. Such objects are used locally in methods of the JBLAS and JLAPACK classes, so it is natural to place them within the methods. However, we make them private static class members. Thus, they are created once, instead of at every method invocation. Note that this approach works only because none of the methods in the library are recursive and because we are ignoring issues of thread safety.

3.4. *Method granularity*

Most of the work in the BLAS routines involves looping through columns of a matrix, accessing and modifying elements. An example is the `scale` routine, which scales a vector by a constant factor. Here is a natural implementation of this routine:

```
// Scale the vector col by the factor a
for(i = col.length(); i > 0; i--) {
   tmp = a * col.eltAt(i);
   col.assignAt(tmp, i);
}
```

Unfortunately, every call to *eltAt*() and *assignAt*() must use the shape object to calculate the address of an element from scratch. Equations (1) and (2) demonstrate the cost of these calculations. Boisvert et al. [8] observe that the use of such methods is five times slower than an ordinary array access. The vector and matrix classes employ two mechanisms to overcome this overhead: aggregate operations and incremental access methods.

3.4.1. *Aggregate operations*

We converted operations that were often performed to entire vector or matrix objects into methods in the the vector and matrix classes. These methods exploit the bulk nature of the updates to access successive elements using incremental address computations. The code for the scale method in the vector class is below. Note how the calculation of the index i consists only of an increment, instead of the multiplication and addition performed repeatedly in the *eltAt*() method.

```
//This is the scale method in the vector
class public void
scale(double a)
{
   int i = shape.start;
   int inc = shape.inc;
   int len = shape.len;
   int j;

   for(j = len; j > 0; j--) {
      data[i] *= a;
      i += inc;
   }
}
```

3.4.2. *Incremental access methods*

Another common type of operation in the library is to loop over a vector, accessing but not modifying its elements. Because the elements are being used instead of being modified, aggregate methods do not apply. An example of such an operation is the scaling of columns of a matrix by elements of a vector. Code for such an operation follows:

```
// A is a Matrix.
// v and col are Vectors.
for(j = v.length(); j > 0; j--) {
   tmp = v.eltAt(j);
   A.colAt(j, col);
   col.scale(tmp);
}
```

To limit the number of calculations in determining the index of an element, we include incremental access methods. The Vector class contains the following incremental methods:

(1) *beginning*(), *end*(): These methods tell the vector that incremental access is about to occur, and that access will start at the beginning or end of the vector.
(2) *next*(), *prev*(): These methods return the next or previous element in the vector.

The Matrix class contains similar methods to access columns and rows incrementally. Using these methods, the code above becomes:

```
// A is a Matrix.
// v and col are Vectors.
v.beginning();
A.beginningCol();
for(j = v.length(); j > 0; j--) {
   tmp = v.next();
   A.nextCol(col);
   col.scale(tmp);
}
```

These methods are similar to the methods defined by the `java.lang.Enumeration` [19] interface. However, `Enumeration` does not handle primitive types, so we could not implement this functionality with the `Enumeration` interface. In JDK 1.2, our incremental operations are similar to the methods of the `java.util.Iterator` interface [29].

3.5. *Error handling*

Every routine in LAPACK provides error checking of its parameters. Parameter values are checked to determine if they are appropriate for the given routine. If an error is discovered, an error value is returned. Identical error handling occurs in JLAPACK. Each parameter value is checked, and if it is not valid, an exception of class `JLASTRUCT.ParamException` is thrown.

3.6. *Complex numbers*

Currently, Java does not provide a primitive type for complex numbers. However, complex numbers are required within the LAPACK library, so we provide two implementations for them. The first approach uses a class `JLASTRUCT.Complex`, encapsulating complex values and arithmetic operations on them. While this object-oriented approach is attractive, the overhead of using many small objects and calling a method for every arithmetic operation makes it unusably slow.

Our second implementation of complex numbers simply inlines them, by making the data arrays of the vector and matrix classes twice as long, and storing the real and imaginary components contiguously in the array. Access methods change from *eltAt*() to *realAt*() and *imgAt*(), and all arithmetic is performed inline. While this is an unattractive approach from a software engineering point of view, it demonstrates the performance achievable with a primitive complex type.

3.7. *Discussion*

Certain aspects of Java made the development of JLAPACK difficult. In this section we discuss these aspects and how JLAPACK addresses them.

3.7.1. *Language issues*

Two language issues hinder the development of JLAPACK: the absence of parametric polymorphism and the absence of operator overloading. The absence of parametric polymorphism required us to create a version of the JLAPACK library for each data type, which results in code bloat and extra programmer effort. Several projects [2,27,28] have examined methods for providing parametric polymorphism, either by modifying the JVM or by a adding a preprocessing phase, and it is possible that the feature will be available in future versions of Java.

The lack of operator overloading required us to write many methods in unnatural forms. For example, the *colAt*() method intuitively should return a Vector object. Because we could not overload the assignment operator, we had to pass in the Vector object as a parameter to the method. Likewise, we had to write out in full detail mathematical operations such as scaling of vectors, instead of using a more natural and compact mnemonic form, such as the `*=` operator.

It is true that neither of these language features is fundamental, and that both represent "syntactic sugar" that would be removed in a preprocessing step. We ignored these issues while implementing JLAPACK, as our goal was to test our hypothesis about performance. However, the general user does not want to deal with such issues and is less apt to use a library that has such unnatural syntax. (Witness the success of Matlab, which virtually removes the difference between the linear algebraic description of an algorithm and its realization in code.) We feel that Java will not be attractive to the numerical computing community until these features are integrated into the language.

3.7.2. *Multithreaded programming*

Java provides full support for multithreading, and supplies a monitor locking mechanism for performing mutual exclusion with the `synchronized` keyword. A `synchronized` method must obtain a lock on its object in order to execute, so only one `synchronized` method can be executing on an object at any time. However, that does not guarantee that a method that is not `synchronized` will not modify data being used by the `synchronized` method. One way to ensure thread-safety is to make all methods that access data `synchronized` and to make all members private, so that subclasses cannot manipulate these members in methods that are not `synchronized`. Such a scheme introduces a huge overhead. Another way to ensure thread-safety is to make local copies of instance variables inside methods. This technique, while legal under current Java concurrency semantics, may not be the most intuitive. Further, Java's memory consistency model [21, Ch. 17], if implemented aggressively, could result in further unexpected behavior. In JLAPACK, we ignore the issues of thread safety.

3.7.3. Complex numbers

Complex numbers must be integrated into the Java environment before the language becomes commonly used for numerical computing. We have presented two methods of implementing complex numbers, and our results document the overhead of encapsulating complex numbers in classes. Manual inlining is not the correct solution either, as this detracts from the readability of the code, replicates common operations, and presents a common source of bugs. Thus, complex numbers must become part of the Java environment. This could take place at any of three points: compile time, load time, or run time.

- **Compile time:** Complex numbers could be introduced at the language level, leaving the JVM specification unchanged. The language would define a primitive complex type and all arithmetic operations on that type. At compile time, the operations on complex types would be translated into operations on real types, inlining the code in the same method we did by hand.
- **Load time:** Complex numbers could be introduced through load time transformations, using a bytecode restructuring tool such as JOIE [10]. Complex numbers would be represented using a Complex class at the language level. At load time, the class loader would modify classes using objects of type Complex, inlining the code. This method is attractive because it requires no modification to the Java language or the JVM.
- **Run time:** Another approach to introducing primitive complex numbers would be in both the language and the JVM. Complex types would become part of the JVM specification. This causes two problems. First, since all arithmetic instructions are specific to a primitive type, many new bytecodes would have to be introduced. Second, double precision complex types would require four words of memory, and current bytecodes only support single- and double-word arguments. The JVM would have to be modified to support extra long arguments.

Orthogonal to the above issue is the matter of exactly how best to integrate complex numbers. The Fortran approach introduces a single type `Complex`, with real and imaginary numbers being represented as special cases of this type with the appropriate component equal to zero. An alternate approach, similar to that under consideration for ANSI C, would introduce three types `Real`, `Imaginary`, and `Complex`. This latter scheme has the advantage of providing greater type discrimination, which allows for more optimization possibilities.

All of these approaches have strengths and weaknesses. While it is beyond the scope of this paper to determine the best mechanism for including primitive complex numbers in Java, this issue is under consideration by the Java Grande Forum [22], and must be resolved satisfactorily if Java is to be viable for numerical computing.

4. Related Work

Several other projects are investigating the use of Java in numerical computing. The Java Numerical Toolkit [8] is a set of libraries for numerical computing in Java. Its initial version contains functionality such as elementary matrix and vector operations, matrix factorization, and the solution of linear systems. HPJava [31] is an extension to Java that allows parallel programming. HPJava is somewhat similar to HPF and is designed for SPMD programming.

Several projects are developing optimizers for Java. Moreira et al. [26] are developing a static compiler that optimizes array bounds checks and null pointer checks within loops. Adl-Tabatabai et al. [1] have developed a JIT compiler that performs a set of optimizations, including subexpression elimination, register allocation, and the elimination of array bounds checking. Such optimizations may allow us to bridge the performance gap between our version with bounds checking and our version without bounds checking.

The F2J [18] project is also generating LAPACK and BLAS libraries in Java. They have developed a Fortran to Java translator, which they use to transcribe Fortran LAPACK source code into Java source code. They have experienced some difficulties handling differences in the languages, such as the absence of a `goto` statement in Java. Currently, f2j has generated Java class files for all of the double precision routines in the LAPACK and BLAS libraries.

5. Performance

Performance is an overarching concern for scientific computation. The Fortran version of LAPACK has been highly optimized and represents our target level of performance. Therefore, we compare JLAPACK with the optimized Fortran version (compiled

on the test platform with the vendor's optimizing Fortran77 compiler from LAPACK 2.0 source distribution downloaded from www.netlib.org) in all our results. In this section, we present the results from our experiments and discuss the reasons for both good and poor performance.

5.1. *Test cases*

We present performance results for solving the system of linear equations $AX = B$, using a coefficient matrix A and a right hand side matrix B whose entries are generated using a pseudorandom number generator from a uniform distribution in the range $[0, 1]$. The same seeds are used in both the Fortran and Java versions, to guarantee that both versions solve identical problems. The square matrix A has between 10 and 1000 columns. The matrix B has from 1 to 50 columns. In every case, the leading dimension of the matrix equals the number of rows of the matrix. We separately timed the triangular factorization (`xGETRF`) and triangular solution (`xGETRS`) stages. The two data types used in timing were double precision real numbers (x=D) and double precision complex numbers (x=Z). For the factorization stage, we used block sizes between 1 and 64.

5.2. *Testing environment*

Table 1 lists the platforms we used for timing. We ran Fortran versions for all Unix platforms, using the `-fast` optimization flag when compiling the Fortran library. We ran the version generated by the F2J translator on both the UltraSparc and the Pentium II. On the DEC, where native BLAS libraries were available through the `dxml` library [11], we measured performance with both the JBLAS classes and the native library. On the Sparcs, we ran two versions with `kaffe` [5,30]: one with dynamic array bounds checking turned on and the other with this feature turned off. We turned off array bounds checking in `kaffe` by modifying the native instructions that its JIT compiler emits. We measured performance without array bounds checking for two reasons. First, we wanted to quantify the cost of performing bounds checks, and to determine if it introduces significant overheads into the computation. Second, global analysis of our code could prove that instances of `java.lang.ArrayIndexOutOfBoundsException` could never be thrown. While this cannot always be determined from the structure of the program, and no current implementation of the JVM systematically eliminates runtime bounds checking in this manner, such an optimization is likely to appear in future generations of JVM implementations.

All of our test platforms are run in JIT mode rather than interpretive mode. The trend, as exemplified by current and short-term future releases of the JVM, is to replace interpretive implementations of the JVM with JIT-based implementations. We expect this trend to continue. While interpretation-based implementations will be useful for debugging, their performance is far too poor to be competitive.

We also measured the performance of JLAPACK when compiled with the High Performance Compiler for Java (HPCJ) [26] developed at IBM Watson. HPCJ is a static compiler that optimizes away some array bounds checks and null pointer checks within loops. It also has the capability of utilizing a fused multiply

Table 1

Testing environment

	SPARCstation 5	UltraSparc 170	DEC Personal Workstation	Pentium II
Processor	SPARC	Ultra 1	Alpha 21164	Pentium II
Processor Speed	110 MHz	170 MHz	500 MHz	300 MHz
Memory	40 MB	64 MB	512 MB	128 MB
Operating System	Solaris 2.5.1	Solaris 2.5.1	Digital Unix 4.0D	Windows 95
JVM	kaffe v0.9.2 JDK 1.2beta4	kaffe v0.9.2 JDK 1.2beta4	JDK 1.1.4	Visual Cafe
JIT Enabled	Yes	Yes	Yes	Yes
F77 Compiler Switches	-fast	-fast	-fast	N/A
ILP	No	Yes	Yes	Yes
Out-of-order execution	No	No	No	Yes

and add (FMA) operation. While this operation is currently illegal under Java semantics, it will become legal if a current proposal [23] dealing with numerics is adopted.

5.3. Performance optimizations

We manually compensated for certain deficiencies in `javac` to boost the performance of our code. The first modification was loop unrolling, a common technique used by optimizing compilers to achieve better performance. In our experiments, an unrolling depth of four gave the best performance. Unrolling does introduce a cost in code size. Unrolling loops in the `JLASTRUCT.Vector` class by factors of two, four, and eight increased class file sizes by 41%, 62%, and 104%.

The second technique optimizes field access. Whenever a member of an object is accessed, a *getfield* operation is performed. This operation has considerable overhead, as it must check access permissions. If a certain field is accessed repeatedly in a method, the *getfield* operation is performed repeatedly. The Java compiler did not optimize this away by leaving the reference in a local variable, so we did it by hand in the Java source code. This modification made little difference in `kaffe`, which uses JIT compilation, but made a significant difference for interpreted code.

5.4. Results

Table 2 shows the performance results for the four platforms listed in Table 1. Table 3 shows the performance results with the HPJC compiler.

5.5. Discussion

Analysis of the results reveals several interesting facts.

(1) The Java version with bounds checking enabled and inlined complex numbers performs within a factor of three of the Fortran version for certain architectures and problem sizes. On the SparcStation 5, the Java version is about two or three times worse than the Fortran version on the larger problem sizes for both the factorization and the triangular solve. As a side note, the interpreted Java implementation was unusably slow.

(2) On the UltraSparc, for most of the cases with bounds checking enabled and inlined complex numbers, there is less than a factor of four difference between the two versions. However, for the factorization with double precision numbers and blocking, the Fortran version performs about six times better than the Java version. This is because blocking significantly improves

Table 2

Performance results for double precision real (D) and double precision complex (Z) values. Entries represent the ratio of the JLAPACK running time to the LAPACK running time (lower is better). Results for the complex version that uses inlined complex numbers are denoted by (I), and results for the version that used classes for complex numbers are denoted by (C). The results for the triangular factorization without blocking are denoted by F(nb), the results for the triangular factorization with a blocking factor of 16 are denoted by F(b), and the results for the solve are by denoted S. The label r indicates a small matrix (100 by 100) was used so that the program could take advantage of caching. The label R indicates a large matrix (600 by 600) that could not fit into the system cache was used. A — label denotes a missing entry. (a) Performance on a SPARCstation 5. (b) Performance on an UltraSparc 170. (c) Performance on a DEC Personal Workstation. (d) Performance on a Pentium II.

	D		Z(I)		Z(C)	
	r	R	r	R	r	R
F(nb)	2.63	1.97	2.53	2.20	—	—
F(b)	2.93	1.64	2.81	2.44	—	—
S	2.27	1.71	2.94	2.46	—	—

(a)

	D		Z(I)		Z(C)	
	r	R	r	R	r	R
F(nb)	6.99	3.03	6.31	3.30	24.88	12.81
F(b)	7.62	5.96	4.33	3.22	16.27	12.60
S	8.09	3.19	2.72	2.05	9.61	7.37

(b)

	D		Z(I)		Z(C)	
	r	R	r	R	r	R
F(nb)	6.49	2.83	6.99	2.73	49.83	19.85
F(b)	8.37	6.22	8.10	6.30	52.82	48.09
S	4.78	2.82	5.04	3.53	33.65	26.31

(c)

	D		Z(I)		Z(C)	
	r	R	r	R	r	R
F(nb)	2.00	4.73	—	—	—	—
F(b)	2.00	2.34	—	—	—	—
S	2.00	1.49	—	—	—	—

(d)

Table 3

Performance results for double precision real (D) and double precision complex (Z) values using the HPCJ system. Entries represent the ratio of the JLAPACK running time to the LAPACK running time (lower is better). Results for the complex version that uses inlined complex numbers are denoted by (I), and results for the version that used classes for complex numbers are denoted by (C). The results for the triangular factorization without blocking are denoted by F(nb), the results for the triangular factorization with a blocking factor of 16 are denoted by F(b), and the results for the solve are by denoted S. The label r indicates a small matrix (100 by 100) was used so that the program could take advantage of caching. The label R indicates a large matrix (600 by 600) that could not fit into the system cache was used. (a) Performance on a PowerPC. (b) Performance on a Power2.

	D		Z(I)		Z(C)	
	r	R	r	R	r	R
F(nb)	1.53	1.01	0.99	1.01	2.78	3.00
F(b)	3.10	1.20	1.20	1.22	5.58	5.53
S	1.50	1.29	1.29	1.14	4.26	4.16

(a)

	D		Z(I)		Z(C)	
	r	R	r	R	r	R
F(nb)	2.90	2.52	1.70	1.62	10.20	10.00
F(b)	4.11	3.68	3.20	1.71	16.70	11.51
S	6.20	2.27	1.58	1.33	7.80	8.05

(b)

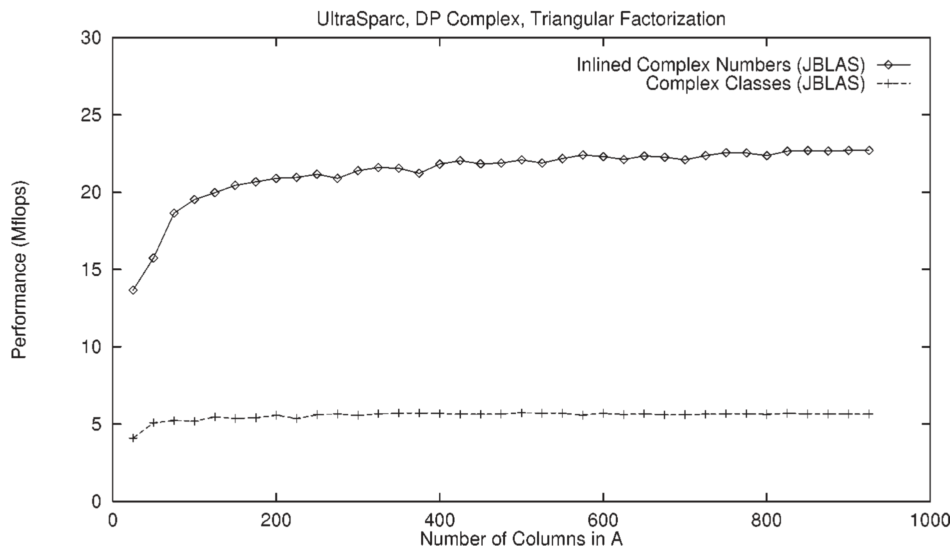


Fig. 2. The performance with double precision complex numbers. The performance with inlined complex numbers and with complex classes is shown.

the performance of the Fortran version, but not of the Java version. Our hypothesis is that the variations in performance represent instruction scheduling effects. We examined the assembly code generated by the Fortran compiler on the SparcStation 5 and on the UltraSparc, which represent different implementations of the same instruction set architecture. The code generated for the inner loops of several routines varied considerably, using different degrees of loop unrolling and different schedules. The `kaffe` and `JDK` JIT compilers generated identical instruction sequences for both platforms. We believe that the sub-optimal instruction schedule increases pipeline stalls and nullifies the improvements in spatial locality due to blocking.

(3) Figure 2 shows that using classes to represent complex numbers performs very poorly. On all the platforms tested, the version that uses the `Complex` class is more than twice as slow as the version that inlined complex numbers.

(4) The impact of bounds checking is shown in Fig. 3. Removing bounds checking increased performance by 15% to 25%. This figure is in line with the corresponding figure reported by Boisvert et al. [8]. While this impact is not significant, it has been shown by Moreira et al. [26] that removing bounds checks enables further optimizations, such as reordering loop

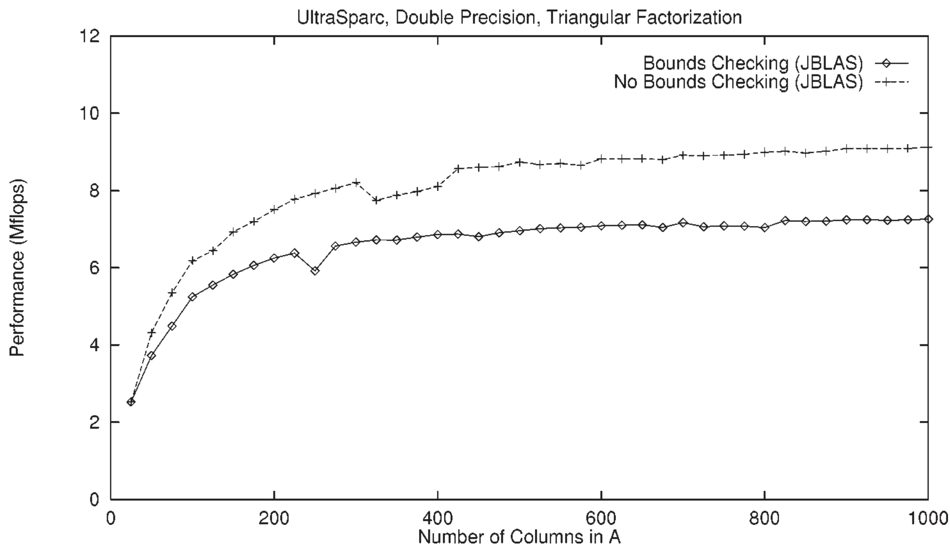


Fig. 3. The impact of bounds checking.

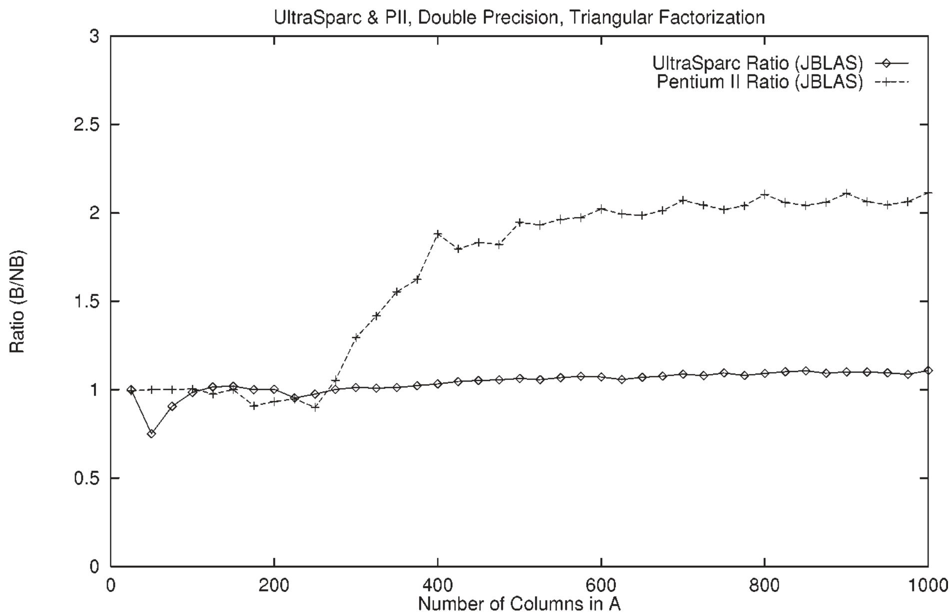


Fig. 4. Architectural effects. The ratio of the performance with blocking to the performance without blocking is shown for two platforms.

iterations, that do have a significant impact on performance.

(5) Figure 4 demonstrates the impact of blocking on both the Pentium II and the UltraSparc platforms. On the Pentium II, which provides instruction re-ordering in the hardware, blocking makes an impact. However, on the UltraSparc, which does not perform dynamic scheduling, blocking has no impact on performance.

(6) Figure 5 shows the performance when the native BLAS library was used. The native BLAS library made a significant impact on performance, especially for the cases where blocking was used. Because LAPACK heavily relies on BLAS for its computations, using the native BLAS library brought the performance of JLAPACK close to the performance of LAPACK (within 15% for sufficiently large problem sizes). This demonstrates that the object-oriented wrappers provided by JLAPACK were efficient. It also

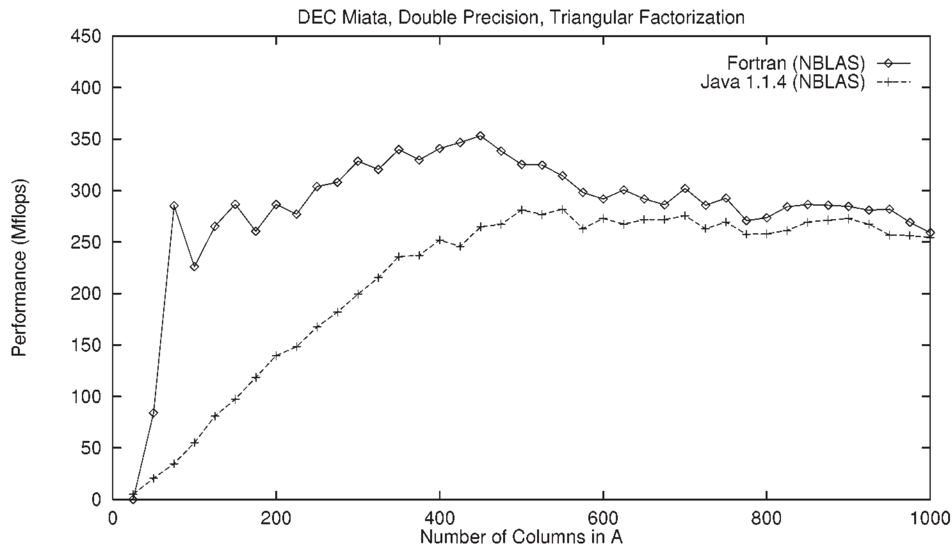


Fig. 5. Performance using the native BLAS library.

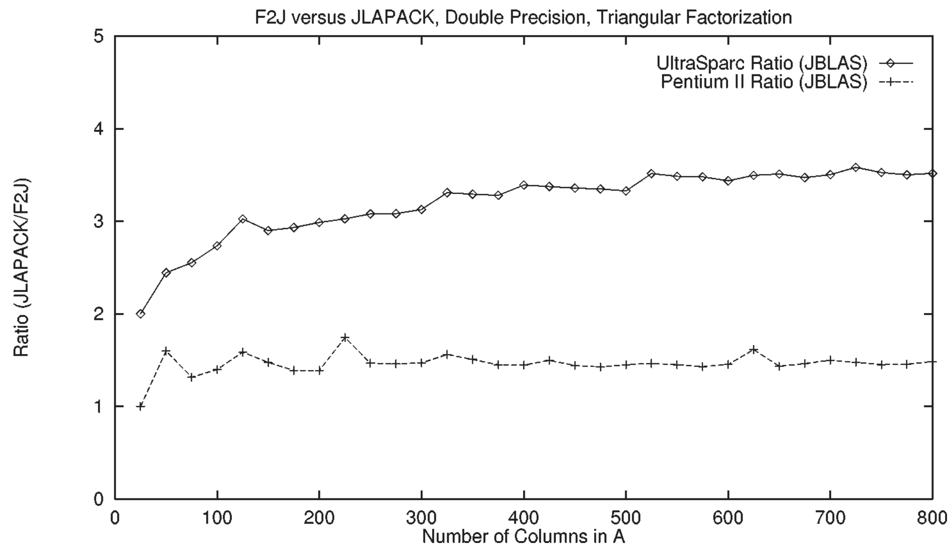


Fig. 6. Comparision of JLAPACK and F2J generated LAPACK code. The ratio of the performance of our version to the performance of the F2J version is shown for two platforms.

supports our hypothesis that poor instruction scheduling hurt performance in the pure Java version.

(7) Figure 6 compares the performance of our version with the performance of the version using the f2j translator. On the UltraSparc, our version performs around 3 times better. On the Pentium II, our version performs around 1.5 times better. This demonstrates that design issues are critical when developing high performance numerical software.

6. Conclusions and future work

Portability, security, and ease of use make Java an attractive programming environment for software development. Performance problems and the absence of several language features have hindered its use in high-performance numerical computing. While operator overloading and parametric polymorphism are indeed “syntactic sugar”, they will contribute significantly to the usability of the language and to the willingness of the numerical computing community to use

Java. We have quantified the difference between using a primitive type for complex numbers, which we have simulated, and using a class for complex numbers. As expected, there is strong evidence that a primitive type is needed.

Future work in the development of high-performance object-oriented numerical libraries in Java can be divided into the following categories.

(1) **Programming model changes.** The algorithms implemented in most numerical libraries today were designed for the Fortran programming model. These may not be the best algorithms when run under the object model of Java. We have discussed several object-oriented programming idioms to implement numerical libraries efficiently. Future work needs to explore these and other techniques such as expression templates [6].
(2) **Compiler changes.** We noted in Section 5 several desirable optimizations that `javac` does not perform. Much work remains to be done here to develop better compilation techniques for Java. Budimlic and Kennedy [9] are exploring such optimizations using object inlining techniques.
(3) **Just-In-Time compilation.** Current JIT compilers are in their early version, and have not been heavily optimized. As we discussed in Section 5, some do not take advantage of machine-specific optimizations and do not appear to schedule code effectively.
(4) **Architectural issues.** Current trends in processor implementation adds significant instruction re-ordering capabilities to the hardware. Engler [17] conjectures that this may reduce or obviate the need for instruction scheduling by JIT compilers. This is a reasonable conjecture whose range of applicability needs to be tested.
(5) **Experimentation with other codes.** LAPACK is obviously not representative of all numerical software. Further work needs to be done to determine if Java implementations of other numerical software behave similarly. We are currently investigating the performance of the expert general equation solver in LAPACK and plan to investigate the performance of sparse matrices in Java.

Our results demonstrate that Java may perform well enough to be used for numerical computing, if a handful of concerns about language features and compilation strategies are adequately addressed. While we have not yet met the goal of having Java perform as well as Fortran, we are beginning to get reasonably close. We speculate that a combination of techniques will narrow this gap considerably over the next few years, making Java a competitor for numerical computing in the near future.

Acknowledgments

This research is supported in part by the National Science Foundation under CAREER award CCR-9501 979 and grant CCR-9711438. We thank the Department of Computer Science at Duke University for machine time on their alpha machines and Sam Midkiff at IBM T.J. Watson Research Center for running test cases with the High Performance Compiler for Java (HPCJ).

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V.M. Parikh and J.M. Stichnoth, Fast, effective code generation in a just-in-time Java compiler, in: *Proceedings of PLDI'98*, Montreal, Canada, June 1998, pp. 280–290.

[2] O. Agesin, S. Freund and J. Mitchell, Adding type parameterization to the Java language, in: *Proceedings of the Symposium on Object Oriented Programming: Systems, Languages, and Applications*, 1997, pp. 49–65.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J.D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Oustrouchov and D. Sorenson, *LAPACK User's Guide*, 2nd edn., SIAM, Philadelphia, PA, 1995.

[4] K. Arnold and J. Gosling, *The Java™ Programming Language*, The Java™ Series, Addison-Wesley Publishing Company, 1996.

[5] M. Barr and J. Steinhorn, Kaffe, anyone? Implementing a Java Virtual Machine, *Embedded Systems Programming*, Feb. 1998, pp. 34–46.

[6] F. Bassetti, K. Davis and D. Quinlan, A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks, in: *Proceedings of ISCOPE'97*, 1997.

[7] A.J.C. Bik and D.B. Gannon, A note on native level 1 BLAS in Java, *Concurrency: Practice and Experience* **9**(11) (Nov. 1997), 1091–1099.

[8] R.F. Boisvert, J.J. Dongarra, R. Pozo, K.A. Remington and G.W. Stewart, Developing numerical libraries in Java, *Concurrency: Practice and Experience* **10**(11) (Sept. 1998), 1117–1129.

[9] Z. Budimlic and K. Kennedy, Optimizing Java: Theory and practice, *Concurrency: Practice and Experience* **9**(6) (June 1997), 445–463.

[10] G.A. Cohen, J.S. Chase and D.L. Kaminsky, Automatic program transformation with JOIE, in: *Proceedings of the 1998 USENIX Annual Technical Symposium*, June 1998.

[11] *DIGITAL Extended Math Library*, http://www.digital.com/hpc/software/dxml.html

[12] A. Dingle and T.H. Hildebrandt, Improving C++ performance using temporaries, *Computer*, Mar. 1998, pp. 31–41.

[13] J.J. Dongarra, J.D. Croz, S. Hammarling and I. Duff, Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs, *ACM Trans. Math. Softw.* **16**(1) (Mar. 1990), 18–28.

[14] J.J. Dongarra, J.D. Croz, S. Hammarling and I. Duff, A set of level 3 basic linear algebra subprograms, *ACM Trans. Math. Softw.* **16**(1) (Mar. 1990), 1–17.

[15] J.J. Dongarra, J.D. Croz, S. Hammarling and R.J. Hanson, Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs, *ACM Trans. Math. Softw.* **14**(1) (Mar. 1988), 18–32.

[16] J.J. Dongarra, J.D. Croz, S. Hammarling and R.J. Hanson, An extended set of basic linear algebra subprograms, *ACM Trans. Math. Softw.* **14**(1) (Mar. 1988), 1–17.

[17] D.R. Engler, VCODE: A retargetable, extensible, very fast dynamic code generation system, in: *Proceedings of PLDI'96*, 1996, pp. 160–170.

[18] *F2J*, *The Fortran 2 Java (f2j) compiler project*, http://www.cs.utk.edu/f2j/

[19] D. Flanagan, *Java in a Nutshell*, O'Reilly & Associates, Inc., Cambridge, MA, 1997.

[20] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, 1995.

[21] J. Gosling, B. Joy and G. Steele, *The Java™ Language Specification*, The Java™ Series, Addison-Wesley Publishing Company, 1996.

[22] Java Grande Forum, *The Java Grande Forum charter document*, http://www.npac.syr.edu/javagrande/jgfcharter.html

[23] Java Grande Forum, *Making Java work for high-end computing*, JGF Technical Report JGF-TF-1, 1998. Available at http://www.javagrande.org/reports.htm

[24] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, Algorithm 539: Basic linear algebra subprograms for Fortran usage, *ACM Trans. Math. Softw.* **5**(3) (Sept. 1979), 324–325.

[25] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, Basic linear algebra subprograms for Fortran usage, *ACM Trans. Math. Softw.* **5**(3) (Sept. 1979), 308–323.

[26] J.E. Moreira, S.P. Midkiff and M. Gupta, From flop to megaflops: Java for technical computing, in: *Proceedings of LCPC'98*, 1998, pp. 1–23.

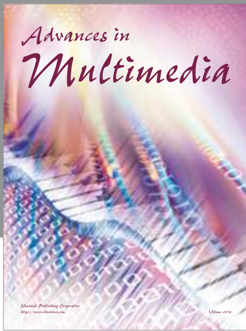
[27] A.C. Myers, J.A. Bank and B. Liskov, Parameterized types for Java, in: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1997, pp. 132–145.

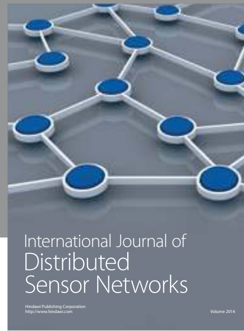
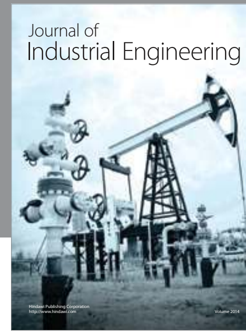
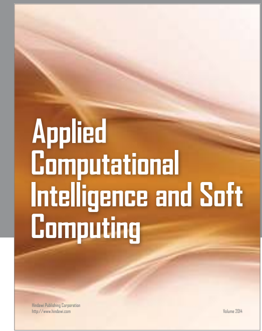
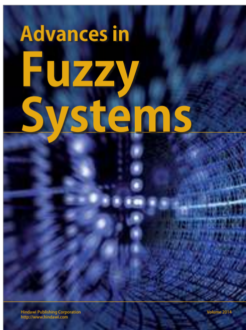
[28] M. Odersky and P. Wadler, Pizza into Java: Translating theory into practice, in: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1997, pp. 146–159.

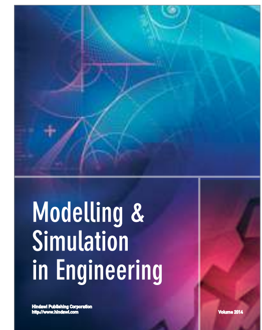
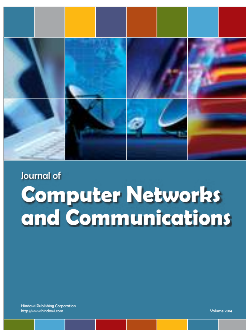
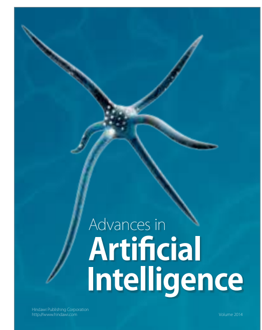
[29] *The Java collections framework*, http://java.sun.com/products/jdk/1.2/docs/guide/collections/index.html

[30] T.J. Wilkinson, The Kaffe homepage, http://www.transvirtual.com/kaffe.html

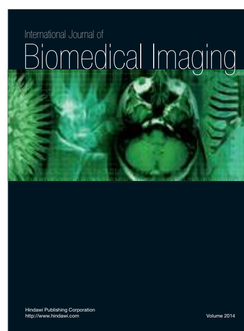
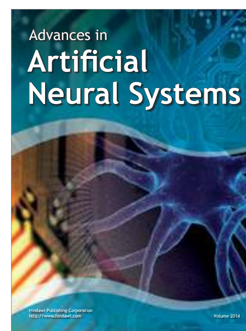
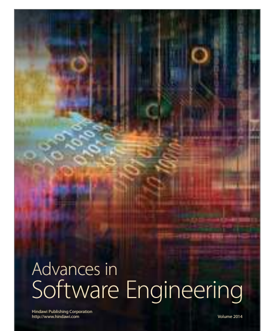
[31] G. Zhang, B. Carpenter, G. Fox, X. Li and Y. Wen, Considerations in HPJava language design and implementation, in: *Proceedings of LCPC'98*, 1998, pp. 24–43.


Advances in
Multimedia

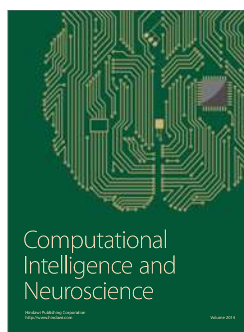
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

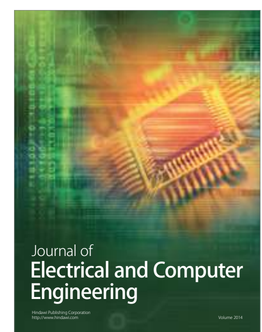
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering