

LAPACKJ: A JNI-Based Approach to Reuse High Performance Linear Algebra Code

Eric Gamess ^(*,+) and Arturo J. Sánchez-Ruíz ^(∇)

E-mail: egamess@kanaima.ciens.ucv.ve

E-mail: asanchez@umassd.edu

^(*) Universidad del Valle
Departamento de Ciencias de la Computación
Cali, Colombia

⁽⁺⁾ Universidad Central de Venezuela
Postgrado en Ciencias de la Computación
Caracas, Venezuela

^(∇) University of Massachusetts - Dartmouth, CIS Department
285 Old Westport Road
North Dartmouth, MA 02747

Abstract

Because of its salient features, such as platform-independence and safety, Java is gaining acceptance as a language for scientific computing. In this paper, we describe *LAPACKJ*, a Java interface to *BLAS* and *LAPACK*, two well-known libraries for high performance linear algebra, fundamental tools used for developing many numerical applications. Our interface is simple to use, object-oriented, and the experiments we have performed indicate that it does not introduce a significant overhead with respect to native libraries.

Keywords: JNI, Code Reuse, High Performance Linear Algebra, BLAS, LAPACK, Java, Fortran, C

1 Introduction

There is a growing interest in using Java as a language for numerical computing, because it is platform-independent, object-oriented, safe, and recent results have shown that it has the potential to attain the performance of traditional scientific languages [1, 2]. Since current JDKs (Java Development Kits) lack of good support for scientific computation, specifically for numerical linear algebra, a number of projects, such as *Jama* [3], and *jlpack* [4] have been proposed. In this paper, we present *LAPACKJ*, a Java interface to *BLAS* (Basic Linear Algebra Subprograms) [5, 6], and *LAPACK* (Linear Algebra PACKage) [7, 8], two well-known Fortran 77 libraries for high performance linear algebra. *LAPACKJ* is object-oriented, and the performance assessment experiments we have developed, indicate that it outperforms both *Jama* and *jlpack* in time and space.

The rest of this paper has been structured as follows. Section 2 introduces *LAPACKJ*'s class hierarchy, comments on the functionality of its main classes, and gives some implementation details. Section 3 discusses the performance of different implementations of the JVM (Java Virtual Machine), and presents the relative performance of *LAPACKJ* with respect to *Jama*, and *jlapack*. Related work is discussed in section 4. We finish the paper with some conclusions that can be drawn after analyzing our preliminary results, and with some of the problems that we will be addressing in the near future.

2 *LAPACKJ*

LAPACKJ stands for “Linear Algebra PACKage for Java”, a set of Java classes that allows users to call native functions of *BLAS* and *LAPACK* from Java. *LAPACKJ* provides classes for constructing and manipulating real and complex dense matrices. It has basic and advanced methods, such as vector copy, vector addition, matrix-vector multiplication, matrix-matrix multiplication, and the numerical solution of systems of linear equations with several right-hand-side vectors, using either LU or Cholesky factorization.

LAPACKJ has been implemented in Java and the JNI (Java Native Interface) [9]¹. The JNI allows Java bytecode to interoperate with applications and libraries written in other languages such as C and C++. It was developed by SUN Microsystems to overcome well-known problems associated with multiple language applications (e.g. to reuse legacy code). The source code of *LAPACKJ* consists of both Java, and C programs. Since it is actually easier to reuse C code by using the JNI, than reusing Fortran 77 code, we opted for writing simple C wrappers for each Fortran 77 program unit, which are then invoked from Java via the JNI. Wrappers were written so that parameter passing is by reference, as dictated by Fortran 77. For more details, see *LAPACKJ*'s source code, which is available from <http://kuaimare.ciens.ucv.ve/lapackj>.

A diagram depicting the *generalization* UML relationship among *LAPACKJ*'s main classes is shown in figure 1. Class `UniArray` and its derived classes (`UniArrayInt`, `UniArrayFloat`, `UniArrayDouble`, `UniArraySComplex` and `UniArrayDComplex`) implement Java unidimensional arrays of `int`, `float`, `double`, and complex numbers (both in single, and double precision), respectively. Instance method `getLength` returns the length of an unidimensional array. Each of these classes provide “set” and “get” methods to modify, and retrieve a specific value associated with an array entry, respectively. In the case of complex entries, the corresponding classes also provide methods to selectively get/set both the real, and imaginary part of a complex number.

Class `BiArray` and its derived classes (`BiArrayFloat`, `BiArrayDouble`, `BiArraySComplex` and `BiArrayDComplex`) implement the bidimensional counterpart of `UniArray`, which relies on Java (unidimensional) arrays in a user-transparent way. Instance methods `getLength`, and `getWidth` return the number of rows, and columns, respectively, of a bidimensional array.

¹See also <http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html>

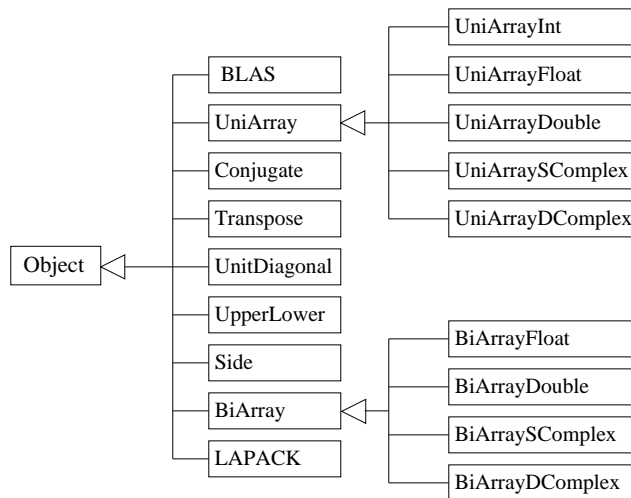


Figure 1: Main classes in *LAPACKJ*'s class hierarchy.

Class `UpperLower` defines two static final values (`UP_TRIAN` and `LOW_TRIAN`) that are used to specify whether a matrix is upper or lower triangular. Three static final values of class `Transpose` (`NO_TRANS`, `TRANS`, and `CONJ_TRANS`) indicate when a matrix object must be unchanged, transposed, or conjugated and transposed (hermitian matrix). Class `UnitDiagonal` is used to specify whether the main diagonal of a matrix is unitary or not.

Class `BLAS` is our way of encapsulating all the functionality of the homonymous linear algebra package (implemented in Fortran 77), as required by the JNI. For instance, native subroutines `sswap`, `dswap`, `cswap`, and `zswap`, swap the elements of two vectors of reals in simple precision, reals in double precision, complexes in single precision, and complexes in double precision, respectively. The Java version has just one overloaded method, named `BLAS.swap`, used to swap the elements of the two arrays passed as arguments.

Similarly, class `LAPACK` has advanced methods such as those used to solve systems of linear equations using either LU or Cholesky factorization. These methods are interfaces to native functions of *LAPACK* that are easier to use. For example, to solve a real (double precision) symmetric, positive-definite system of linear equations with *LAPACK* (in Fortran 77), the following subroutine should be used

```
dposv(uplo, n, nrhs, a, lda, b, ldb, info)
```

where `uplo`, `n`, `nrhs`, `lda`, and `ldb` specify whether the matrix `a` is upper or lower triangular, matrix size, number of right-hand-side vectors in matrix `b`, `a`'s leading dimension, and `b`'s leading dimension, respectively. Argument `info` indicates information about the success of the computation. Similarly, `sposv`, `cposv`, and `zposv` are subroutines to solve a symmetric, positive-definite system of linear equations for reals in simple precision, complexes in simple precision, and complexes in double precision, respectively. In *LAPACKJ*, there is just one method, namely

```
int posv(UpperLower uplo, BiArray a, BiArray b)
```

thus offering a more natural interface to users.

3 Experiments

In this section we present the experiments we have designed to estimate the performance of *LAPACKJ*. The first suite assesses the performance of the following implementations of the JVM, when used to solve a common linear algebra problem. JDK (Java Development Kit) from SUN Microsystems, and Kaffe² from Transvirtual, two well-known distributions. SUN Microsystems, IBM, and Blackdown³ had ported JDK to Linux. Kaffe 1.0.5 mostly implements Java 1.1 and parts of Java 2.

The second suite assesses the relative performance of *LAPACKJ*, *jlpack*, and *Jama*. These experiments were performed on a Pentium II (400 MHz) processor with 64 Mbytes of RAM, running RedHat Linux 6.0⁴, using both SUN JDK 1.2.2, and IBM JDK 1.1.8 ports of JVM to Linux.

3.1 Experiment 1: Performance of the different JVMs

In order to assess the performance of different JVM implementations, we used *jlpack* to solve randomly generated linear algebraic systems, with symmetric, positive-definite matrices having double precision entries. We did not use *LAPACKJ* because it is mainly implemented in native code. Both *Jama*, and *jlpack* are totally implemented in Java, but we decided to use *jlpack* because it handles bigger problems (see section 3.2). The method we used is based on Cholesky's decomposition. Table 1 shows the times in seconds.

These results indicate that JDK 1.1.7 from Blackdown has the poorest performance. This is mainly due to the fact that its virtual machine uses a plain bytecode interpreter, while the other implementations use a JIT (Just In Time) compiler. The majority of modern JVM implementations use a JIT compiler to translate class bytecode to native machine code at execution time. This increases application startup time, and introduces some delay when loading classes, but it can also significantly reduce the overall execution time. JDK 1.1.8 from IBM has the best performance for this experiment suite. A long dash represents out-of-memory errors. It seems that Blackdown ports of JDK have a weak memory manager which forces users to work with smaller linear algebra problems.

As suggested by these results, we decided to use JDK 1.1.8 from IBM, and JDK 1.2.2 from SUN Microsystems for our second suite of experiments. Apparently, users should use JDK 1.1.8 from IBM for small linear algebra problems for better performance, and JDK 1.2.2 from SUN

²<http://www.kaffe.org>

³<http://www.blackdown.org>

⁴<http://www.redhat.com>

Matrix size	Blackdown JDK 1.1.7	Blackdown JDK 1.2	SUN JDK 1.2.2	Transvirtual Kaffe 1.0.5	IBM JDK 1.1.8
250 × 250	7.40	1.27	1.41	1.84	0.83
500 × 500	58.31	9.69	10.78	14.51	5.49
750 × 750	196.14	32.81	35.61	49.38	17.98
1000 × 1000	463.81	78.04	85.49	117.66	40.96
1250 × 1250	—	152.92	170.25	231.10	82.31
1500 × 1500	—	—	293.17	400.61	147.83
1750 × 1750	—	—	466.24	638.42	233.87
2000 × 2000	—	—	699.98	955.35	364.29
2250 × 2250	—	—	994.94	1363.77	—
2500 × 2500	—	—	1379.51	1875.49	—
2750 × 2750	—	—	—	2500.32	—
3000 × 3000	—	—	—	—	—

Table 1: Performance of different JVM implementations used. All times are in seconds.

Microsystems for bigger problems. If time is not an issue, Transvirtual Kaffe 1.0.5 seems to be an option.

3.2 Experiment 2: Comparing *LAPACKJ*, *jlpack*, and *Jama*

In order to compare the performance of *LAPACKJ*, with respect to that of *jlpack*, and *Jama*, we used them to solve randomly generated linear algebraic systems having the same traits as in the previous experiment. Table 2 collects the results, where the times are in seconds.

Matrix size	LAPACKJ		jlpack		Jama	
	SUN JDK 1.2.2	IBM JDK 1.1.8	SUN JDK 1.2.2	IBM JDK 1.1.8	SUN JDK 1.2.2	IBM JDK 1.1.8
250 × 250	0.14	0.14	1.41	0.83	0.68	0.36
500 × 500	1.15	1.14	10.78	5.49	4.57	2.45
750 × 750	3.98	3.96	35.61	17.98	15.25	8.18
1000 × 1000	9.53	9.52	85.49	40.96	36.86	19.24
1250 × 1250	19.68	19.49	170.25	82.31	70.64	38.06
1500 × 1500	35.49	35.18	293.17	147.83	122.26	—
1750 × 1750	60.02	59.95	466.24	233.87	205.34	—
2000 × 2000	96.41	96.37	699.98	364.29	—	—
2250 × 2250	138.90	—	994.94	—	—	—
2500 × 2500	203.19	—	1379.51	—	—	—
2750 × 2750	—	—	—	—	—	—

Table 2: Performance of *LAPACKJ*, *jlpack*, and *Jama*. All times are in seconds.

These results show that *jlpack* has the poorest performance, a fact that agrees with our

expectations, since *jlpack* is directly obtained from *LAPACK* by using *f2j*⁵, a Fortran 77-to-Java compiler. Memory-wise, *Jama* does not seem to be very efficient, which limits users to deal with smaller linear algebra problems than *LAPACKJ* or *jlpack*. *LAPACKJ* exhibits the best performance, both in time and space. The difference between the results of *LAPACKJ* for JDK 1.1.8 from IBM, and JDK 1.2.2 from SUN Microsystems, is negligible. This also agrees with our expectations, because most of the computation is performed by native code.

4 Related Work

Jama [3] is a basic linear algebra package totally implemented in Java. It provides classes for constructing and manipulating real, dense matrices. However, *Jama* does not have classes to work with complex matrices, as does *LAPACKJ*.

Since *jlpack* [4] is obtained from *LAPACK* by automatic translation using *f2j*, a Fortran 77-to-Java compiler, the resulting code is not totally object-oriented and its interface is not natural, since it includes parameters that are not necessary in Java. For example, methods that have unidimensional arrays as parameters also need their length to be explicitly passed. *LAPACKJ*'s methods do not need this parameter, simply because they can use method `length` to compute it.

*Jampack*⁶ is a Java linear algebra package in development. It currently includes classes for complex numbers only. Developed by Pete Stewart of the University of Maryland and NIST (National Institute for Standards and Technology), *Jampack* exists as preliminary version that is being released to sense whether there is sufficient interest to continue its development.

*JNT*⁷ (Java Numerical Toolkit) is an early design of a numerical toolkit for Java from the NIST. As the web page indicates, “*This software is under constant development and is not yet supported.*” It is also being put out to get feedback from the numerics community.

Java is gaining acceptance as a language to tackle parallel and distributed applications. Message-passing packages with interfaces similar to PVM [10] (Parallel Virtual Machine), and MPI [11] (Message Passing Interface) have already been developed [12, 13]. However, there has been little interest to develop a parallel linear algebra package in Java. Well-known parallel linear algebra libraries exist for other languages, mainly for C and Fortran, such as *ScaLAPACK* [14] (Scalable Linear Algebra PACKage) and *PLAPACK* [15] (Parallel Linear Algebra PACKage).

5 Conclusions and Future Work

In this paper we have presented *LAPACKJ*, a Java interface to *BLAS* and *LAPACK*, two well-known high performance linear algebra libraries written in Fortran 77. *LAPACKJ* is object-

⁵<http://www.cs.utk.edu/f2j>

⁶<http://math.nist.gov/javanumerics>

⁷<http://math.nist.gov/jnt>

oriented, and includes advanced methods for various tasks, such as the resolution of a system of linear equations with several right-hand-side vectors using an LU or Cholesky factorization, the numerical solution of least-square problems by using LQ or QR decomposition, and the computation of eigenvalues/eigenvectors, using matrices with different types of entries, including complex numbers.

We have developed randomly generated experiments which suggest that carefully designed class hierarchies in Java can be used to present a clean interface to already existing libraries without compromising efficiency. This is a clear indication that JNI-based approaches look like an attractive alternative to develop efficient distributed linear algebra packages by reusing already existing standard code.

Our approach to reusability should exhibit better quality properties when compared to approaches based on automatic, and manual translation, for the latter are error-prone. Sources of errors in our approach are well localized, i.e. the wrappers, and therefore easier to manage.

Portability is always an issue to be discussed when using JNI. Even though our approach is mainly aimed at reusing legacy code, porting it to different platforms does not seem to pose a big challenge. In our implementation, JNI classes interact with C wrappers, which in turn, interoperate with the Fortran 77 native implementation. The interaction JNI-C-Fortran does not need to be changed when going from one platform to another, provided that the same C and Fortran compilers are available on both platforms, and interoperability is possible between them. For instance, this is the case of GNU-based compilers, very well-known and amply used by the computer science community.

We are currently working on the problem of using JNI to develop a coordination substrate on which existing Fortran libraries can be embedded, and on the problem of building tools that automatically produce the required middleware (JNI-based or otherwise) by direct inspection of the Fortran code to be reused.

References

- [1] José Moreina. Closing the performance gap between Java and Fortran in technical computing. Java for High Performance Computing Workshop, Europar 98, 1998.
- [2] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.
- [3] Jama: A Java Linear Algebra Package.
<http://math.nist.gov/javanumerics/jama>.

- [4] jlapack: An Automatic Translation of LAPACK.
<http://www.cs.utk.edu/f2j/download.html>.
- [5] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, pages 1–17, 1990.
- [6] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, pages 1–17, 1988.
- [7] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High Performance Computers. University of Tennessee, CS-90-105, 1990.
- [8] E. Anderson and J. Dongarra. Performance of LAPACK: A Portable Library of Numerical Linear Algebra Routines. University of Tennessee, CS-92-156, 1992.
- [9] S. Liang. *The JavaTM Native Interface: Programmer's Guide and Specification*. The JavaTM Series: From the SourceTM. Addison-Wesley, 1999.
- [10] A. Geist, A. Beguelin, J. Dongarra, and W. Jiang. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.
- [11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, Knoxville, Tennessee, June 1995.
- [12] A. Ferrari. JPVM: Network Parallel Computing in Java. ACM 1998 Workshop on Java for High-Performance Network Computing. <http://www.cs.ucsb.edu/conferences/java98/papers/jpvm.ps>.
- [13] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li. mpiJava: A Java MPI Interface. First UK Workshop on Java for High Performance Network Computing, 1998.
- [14] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. *IEEE Computer Society Press*, pages 120–127, 1992.
- [15] P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, and Y. Wu. PLAPACK: Parallel Linear Algebra Package. *in Proceedings of SIAM Parallel Processing Conference*, 1997.