Paper 107

©Civil-Comp Press, 2006.
Proceedings of the Fifth International Conference
on Engineering Computational Technology,
B.H.V. Topping, G. Montero and R. Montenegro,
(Editors), Civil-Comp Press, Stirlingshire, Scotland.

A Hybrid Concept for Numerical Applications in Java

M. Baitsch, N. Li and D. Hartmann
Ruhr-University of Bochum, Germany

Abstract

Java is widely recognized as a good object-oriented programming language. However, it is often considered as too slow for numerically intensive applications. This paper presents a hybrid software package that brings the computational efficiency of numerical Fortran libraries to Java. The package is organized in two layers. In the first layer, Java wrappers provide access to numerical libraries like BLAS, LAPACK or NAG. The second layer comprises vector and matrix classes as well as classes for common linear algebra tasks. Numerical experiments show that this strategy can substantially improve the performance of Java based numerical applications.

Keywords: Java, linear algebra, object-oriented, scientific computing, benchmarking

1 Introduction

This paper is about a linear algebra software package for numerical applications in Java. The software package emerged in the context of a Java-based finite element software for the structural analysis. Applications of this software include teaching as well as lifetime-oriented design optimization [2, 3]. The package is intended for small to medium sized problems involving up to 100,000 degrees of freedom.

Java as a modern object-oriented programming languages offers many features that make it an attractive choice also for numerical applications. In particular, Java programs are easily portable. Java has a clean and simple, purely object-oriented design and ships with a comprehensive class library. A strict compiler, the absence of pointers and extensive runtime error checking support the development of safe and reliable software. Finally, it can be stated that Java simplifies the development, testing and maintenance of large software systems. Contrary to a large, monolithic software system, an object-oriented, and well designed Java-based finite element application can easily be integrated into a system for structural optimization, an interactive application or an Internet-based solution.

The advantages of Java have also been realized in parts of the scientific computing community as documented by several ongoing research projects, e.g. [20, 19]. However, much of the software development for numerical applications nowadays is in Fortran, a language which first emerged in 1954. Will Fortran remain the dominant language in that field for the next 50 years? Why is the interest in Java with respect to numerical applications that low? Probably two reasons are most important (a comprehensive appraisal of Java for numerical computing is given in [5]). The first reason is computational efficiency. Evidently Java is slow compared to C or Fortran [6] and for numerical applications performance is critical. Although the performance gap is constantly shrinking because of improving just in time compilation technology [18], it can not be expected that Java will outperform highly optimized vendor tuned numerical code like the Intel Math Kernel Library MKL [15] or the AMD Core Math Library ACML [1]. The second reason is the availability of numerical libraries. Especially linear algebra is ubiquitous in scientific applications and, therefore, the availability of corresponding libraries which are complete, easy to use, fully documented and reliable is a prerequisite. Although a lot of work already has been undertaken [7, 4, 16], the situation for Java is still poor compared to the large number of libraries available in Fortran.

The concept presented in this contribution addresses both problems, efficiency and availability of libraries, using a hybrid concept that is based on existing numerical libraries. Most numerical applications spend large portions of the computing time in small regions of the program code. Basically, the hybrid strategy is to develop the overall software in Java and to delegate those numerically intensive tasks to libraries that are programmed in Fortran. For that, a two layer architecture is introduced:

- In the first layer, existing numerical libraries are linked to Java using the standard Java Native Interface (JNI). This provides access to a wide variety of high-performance numerical routines.

- The second layer is a thin object-oriented interface to the low-level computational routines. One package of the corresponding class library contains vector and matrix interfaces with various implementations for different storage schemes (packed, banded, profile, ...). Other packages contain solver classes for linear systems and eigenproblems.

This architecture provides ways and means to develop object-oriented numerical applications in Java that are driven by the numerical efficiency of procedural Fortran libraries. Numerical examples demonstrate that the hybrid approach suggested can improve the performance of Java based numerical applications significantly.

2 Fast Numerical Libraries for Java Applications

Numerical libraries for Java can be created using different strategies [4]. (i) The libraries can be coded directly in the Java. Naturally, this approach requires notable work for coding and testing. (ii) Another option is to use tools that automatically convert Fortran source code into Java [9]. (iii) Java wrappers can be applied to access existing numerical libraries. The wrapper code is automatically generated using a comparatively simple tool [11, 17]. The first two strategies result in code that is perfectly portable but suffers from Java's performance deficiencies. Therefore, we have chosen the third option that gives access to a large number of fast numerical libraries. Of course, programs based on that solution are not portable in a strict sense. However, programs are portable to any platform where the native libraries and the wrappers are available.

2.1 Java Access to Native Libraries

Subroutines in a library can be accessed from Java by using the Java Native Interface (JNI). JNI is a standard programming interface for writing Java native methods (and embedding the JavaTM virtual machine into native applications). Using JNI, binary compatibility of native method libraries across Java virtual machine implementations on a given platform can be guaranteed. Figure 1 shows how JNI acts as link between a Java application and native numerical libraries.

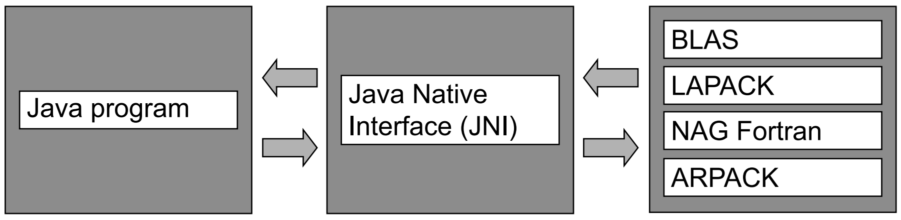


Figure 1: The Java Native Interface as link between Java and native numerical libraries

JNI basically requires a Java class that declares native methods and associated C or C++ code (for several reasons, we use C++) that contains the corresponding implementation. The individual components are explained in detail at hand of the Fortran function *DNRM2* from BLAS. In Fortran, the function signature is as follows

```
      DOUBLE PRECISION FUNCTION DNRM2 ( N, X, INCX )
*     .. Scalar Arguments ..
      INTEGER                         INCX, N
*     .. Array Arguments ..
      DOUBLE PRECISION                X( * )
      ...
```

where all parameters passed in the parameter list are unchanged on exit.

```
double DNRM2(const int& n, const double[] x, const int& incx);
...
```

(a) C++ header file blas.h containing function prototypes

```
public class JBlas {
    ...
    public static native double DNRM2(int n, double[] x, int incx);
    ...
}
```

(b) Java class JBlas containing native methods

```
#include <jni.h>    // JNI infrastructure
#include <blas.h>   // original header file
#include <JBlas.h>  // header generated by javah

...
JNIEXPORT jdouble JNICALL Java_JBlas_DNRM2(JNIEnv *env, jclass cls,
        jint n, jdoubleArray x, jint incx) {
    jdouble* xPtr = (jdouble*)env->GetPrimitiveArrayCritical(x, 0);
    jdouble tmp = DNRM2(n, xPtr, incx);
    env->ReleasePrimitiveArrayCritical((jarray)xPtr, x, 0);
    return tmp;
}
```

(c) C++ code that glues Fortran BLAS to Java

Figure 2: Accessing a subroutine from BLAS via JNI

Figure 2 shows all the ingredients that are necessary to access the *DNRM2* function from Java. In the following, the individual components are introduced.

A header file, that contains function prototypes is a prerequisite when a library function should be called from C++. However, the mapping of parameter data types from Fortran to C++ is not unique. Because Fortran uses an *everything is a pointer* semantic, a straightforward approach would be the signature

```
double DNRM2(int* n, double* x, int* incx);
```

which is actually the style of the headers shipping with Intel's MKL and AMD's ACML. However, this style does not take into account the different semantics (scalar, array, in, out, inout) of the parameters. The alternative we therefore adopted is shown in Figure 2(a): Here, parameters that are single numbers are passed by reference and the *const* keyword is used to indicate that the parameters remain unchanged within the function. This style is used in the header file that comes with the NAG Fortran library.

Next, a Java class containing methods that correspond to the library routines has to be provided (Figure 2(b)). Note that the *DNRM* method in the *JBlas* class is declared as native and does not have a method body indicated by the semicolon at the end of the declaration. For the present function parameters the mapping from C++ to Java is straightforward. Slightly more difficult are scalar parameters that have an in-out semantic. Because Java follows a strict call by value scheme for method parameters, special classes (*IntRef*, *DoubleRef*, ...) are used to emulate call by reference in Java. Using the *javah* tool that reads the compiled *JBlas* class, the header file *JBlas.h* is generated. This header contains C++ function prototypes corresponding to the native methods in *JBlas*.

Finally, the C++ program file contains the implementation of the native methods and acts as glue between Java and the actual library. The data types of function parameters depend on the parameters of the native Java method. Type mapping to C++ is defined by the JNI standard. Particularly, arrays are not passed as simple pointer (as one might expect) but as an array data structure. In order to access the array in a standard way, first it has to be extracted. Then the library function *DNRM2* is invoked. At the end, array parameters are released again.

2.2 Automatic Binding to Java

Creating the Java class and the C++ glue code for libraries that contain hundreds of subroutines (like the NAG library) is time-consuming and prone to errors. Furthermore, the required steps are schematic and thus can be automatized very well. For these reasons, the code generator *genjni* has been developed. The *genjni* tool reads a C or C++ header file containing the library function prototypes along with generates a corresponding Java class (Figure 2(b)) along with C++ glue code (Figure 2(c)).

The *genjni* tool is a Java program which constructs in a first step an object model of the function prototypes. In order to avoid the difficult task of directly parsing C or C++ code, the freely available gccxml tool [10] is used to generate a XML representation of the header file. Using a SAX parser, the object model is constructed based on the XML file. The second step is to generate the Java code (Figure 2(b)) and the C++ code (Figure 2(c)). For that, the objects that represent the return value and the individual function parameters know how to handle the corresponding data type.

Using *genjni*, the Fortran libraries BLAS (Basic Linear Algebra Subprograms), LAPACK (Linear Algebra PACKage), partially ARPACK (ARnoldi PACKage) and the NAG (Numerical Algorithms Group) library have been made available to Java applications. The corresponding libraries have been compiled on Windows and Linux.

3 Object-Oriented Layer

Using simply the native numerical libraries described above would lead to a Fortran programming style in Java. Although being efficient in terms of execution time, the advantages of the object-oriented paradigm were disappearing. Therefore, a class library, i.e. a thin object-oriented layer between the application and the libraries, has been developed. The primary design goal is to combine a convenient application programming interface (API) and numerical efficiency. The class library includes

- classes to store general matrices as well as classes for matrices with special properties like symmetry or a banded or sparse structure;
- support for basic linear algebra operations (addition, multiplication, etc.) on vectors and matrices;
- solvers for standard algebraic problems (linear systems and eigenproblems) based on dense and sparse matrices.

3.1 Matrices, Vectors and Linear Algebra

Before our package is presented, potential design concepts are to be discussed. In Java, there exist two fundamental approaches to the design of vector and matrix classes and linear algebra operations. An illustrative example is used to discuss the potential implications in both cases. Consider the simple but typical matrix-vector operation

$$\mathbf{y} \leftarrow 2\mathbf{A}^T\mathbf{x} + 3\mathbf{y} \tag{1}$$

where $\mathbf{A}$ is a matrix and $\mathbf{x}, \mathbf{y}$ are vectors.

Option A In this approach, vectors and matrices are objects that are capable of performing the required linear algebra operations. Corresponding classes would contain the necessary *add*, *multiply*, etc. methods. This approach seems to be quite natural for an object-oriented language and the packages JAMA (A Java Matrix Package [13]) or MTJ (Matrix Toolkits for Java [16]), just to mention two of them, are based on that design. However, it has severe drawbacks. Consider the Java code fragment that corresponds to operation (1):

```
y = a.transpose().times(x).times(2.0).add(y.times(3.0));
```

Here, first, the transpose of $\mathbf{A}$ is computed. Then, the returned matrix object is multiplied by the vector $\mathbf{x}$ and the resulting vector is scaled by 2, etc. Let's briefly discuss consequences: (i) The code is lengthy and hard to read. (ii) In that single line of code, one matrix object and four vector objects are constructed. This is a serious performance issue if the statement is executed often or if the objects are large. Of course, one could introduce more sophisticated methods that, for instance, add and assign in one operation. The problem with that is the resulting large number of methods necessary to cover all combinations of arguments (for instance, the *Matrix* interface in the MTJ package has nearly 60 methods).

Option B Here, vectors and matrices are data structures that expose only a minimal set of methods, basically to access the individual entries. The linear algebra operations are interpreted as algorithms operating on that data structures and are realized as static methods in a class that has an interface similar to BLAS and is called *BLAM* (Basic Linear Algebra Methods). Although this choice may remind one of an old-fashioned structured programming style, the corresponding code is surprisingly simple:

```
BLAM.multiply(2.0, BLAM.TRANSPOSE, a, x, 3.0, y);
```

The main advantages of this second approach are: (i) No objects are instantiated. (ii) The API of the matrix and vector classes is very easy; algorithms are collected in the *BLAM* class which has also a straightforward API. Compared to Fortran BLAS, the main advantage is that the matrix dimensions have not to be passed explicitly. One drawback compared to option A is that certain simple operations are more difficult to express and may require slightly more coding. However, we choose option B for its simplicity.

3.1.1 Matrix Classes

Because of the fundamental design decision taken, the matrix interfaces and classes are extremely lightweight. The corresponding class diagram is shown in Figure 3.

Operations to be implemented by matrix classes are defined in the *MatrixRO* and the *Matrix* interfaces. Loosely based on the immutable pattern [12], the *MatrixRO* interface defines solely read-only operations. This data type is used by methods to indicate that a matrix parameter object will not be modified. Furthermore, the immutable pattern allows classes to safely expose attributes to the outside world.

The *Matrix* interface extends *MatrixRO* with operations to modify matrix objects. This includes methods to set an entry, to add to an entry and to assign from another matrix object. This method allows sparse matrices to scan the argument matrix in order to use an efficient storage scheme. Based on these interfaces, transpose matrices, submatrices, etc. can be easily constructed using a delegate mechanism. The corresponding classes are not shown here. Several classes implement these interfaces whereas different storage formats are used. For instance, the *Array1D* class uses a one-dimensional array to store data either in Fortran like column-major ordering or in C style row-major format. The *Array2D* wraps a conventional two-dimensional Java array (array of arrays). Other classes use more specialized storage schemes like packed, banded, or skyline format.

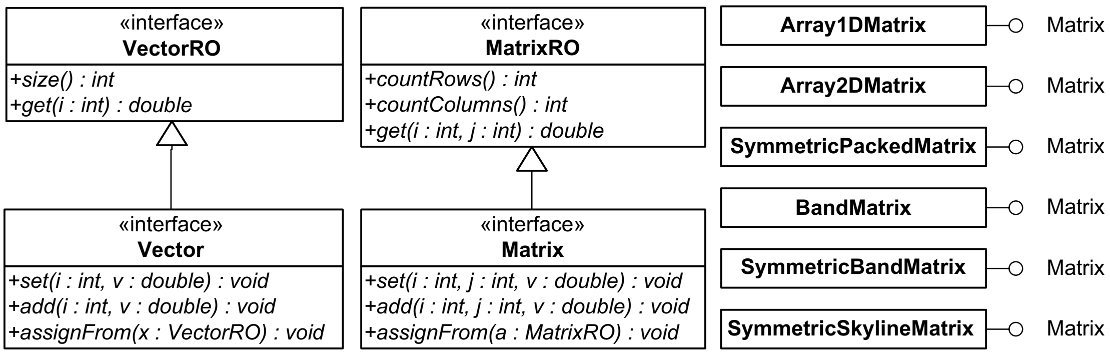
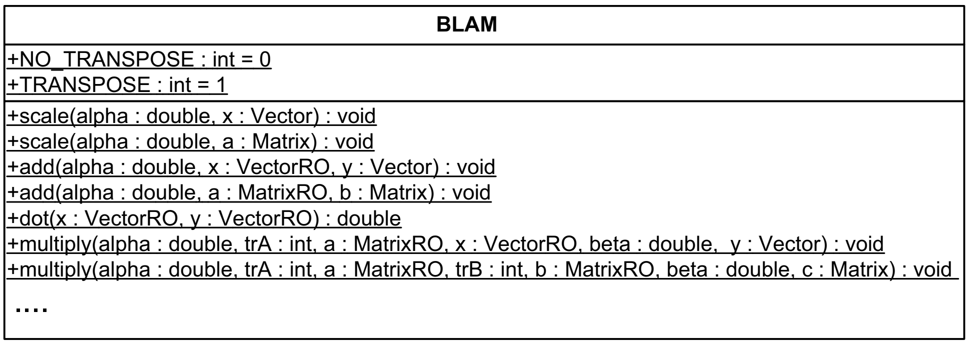


Figure 3: UML class diagram of matrix interfaces and classes



```
BLAM
+NO_TRANSPOSE : int = 0
+TRANSPOSE : int = 1
+scale(alpha : double, x : Vector) : void
+scale(alpha : double, a : Matrix) : void
+add(alpha : double, x : VectorRO, y : Vector) : void
+add(alpha : double, a : MatrixRO, b : Matrix) : void
+dot(x : VectorRO, y : VectorRO) : double
+multiply(alpha : double, trA : int, a : MatrixRO, x : VectorRO, beta : double,  y : Vector) : void
+multiply(alpha : double, trA : int, a : MatrixRO, trB : int, b : MatrixRO, beta : double, c : Matrix) : void
....
```

Figure 4: UML class diagram of the *BLAM* class

A	**x**	**y**	kernel
Array1DMatrix	*ArrayVector*	*ArrayVector*	DGEMV
SymmetricBandMatrix	*ArrayVector*	*ArrayVector*	DSBMV
SymmetricPackedMatrix	*ArrayVector*	*ArrayVector*	DSPMV
any *MatrixRO*	any *VectorRO*	any *Vector*	generic Java

Table 1: Computational kernels for matrix-vector product

3.1.2 Linear Algebra

Basic linear algebra operations are collected in the *BLAM* class that mimics to a large extent the interface of the Fortran BLAS library. But other than BLAS and related libraries that contain different subroutines for different matrix types, *BLAM* only has one single method for each operation (e.g. matrix-vector product). The UML class diagram in Figure 4 shows a subset of the *BLAM* methods.

Of special interest in the *BLAM* class (because computationally most intense) are the two different *multiply* methods that compute the products

$$\mathbf{y} \leftarrow \alpha\,\mathrm{op}(\mathbf{A})\,\mathbf{x} + \beta\,\mathbf{y} \quad \text{and} \quad \mathbf{C} \leftarrow \alpha\,\mathrm{op}(\mathbf{A})\,\mathrm{op}(\mathbf{B}) + \beta\,\mathbf{C}$$

where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are matrices, $\mathbf{x}, \mathbf{y}$ are vectors and $\mathrm{op}(\mathbf{A})$ is $\mathbf{A}$ or $\mathbf{A}^T$. For efficiency it is important to choose an algorithm from a native library that directly operates on the low level data structure of the matrix or vector object. As an example, Table 1 lists different computational kernels to compute a matrix-vector product. For various argument types, efficient subroutines from BLAS are applicable while for the general case, a generic Java implementation serves as fallback.

Because the methods in the *BLAM* class use the most basic data types, the task of choosing the most efficient computational routine for the actual parameters can not be handled by the compiler. Moreover, it is necessary to do this at runtime. For this purpose, an *AlgorithmManager* class is introduced. Matrix classes register specialized algorithms operating efficiently on their specific underlying data structure at the *AlgorithmManager* when loaded. Inside the *multiply* methods, firstly the *Algorithm-*

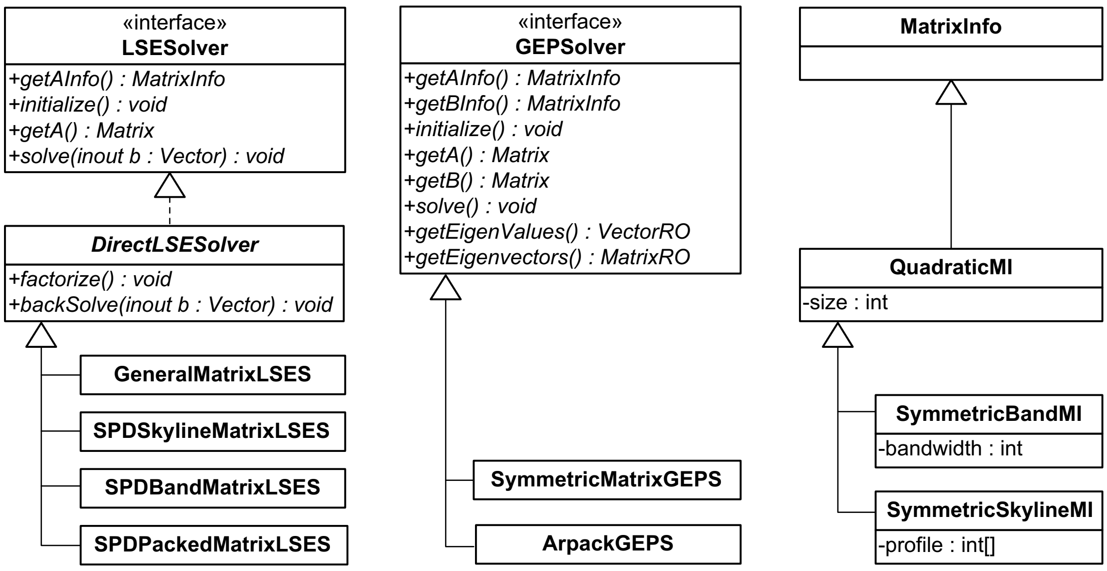


Figure 5: Solver classes for linear systems and eigenproblems

Manager is asked if a special algorithm for the actual parameters is available: If so, that algorithm is executed. Otherwise the generic Java code is used.

The design of the *BLAM* class allows for a programming style that is independent of the matrix/vector types actually being used. This level of generality simplifies the development of numerical software but, naturally, affects performance. Experiments have shown that the additional time needed to choose the algorithm is around 0.001 ms on most platforms. While such an overhead is negligible in most situations, it is a serious problem in the rare situation that a code spends hours multiplying say 5×5 matrices. In such situations, manual optimization is indispensable.

3.2 Solvers for Linear Systems and Eigenproblems

Solution algorithms for medium sized to large linear systems or eigenproblems have to use sparse storage schemes in order to be efficient. In a procedural software, the choice of a specific storage scheme potentially affects large portions of the overall program code. In this section, an object-oriented approach is presented that completely hides the storage scheme used by a specific solution algorithm.

Often solution algorithms for linear systems and eigenproblems are closely coupled to the storage format of the coefficient matrices. For instance, there exist variants of the Cholesky factorization for dense matrices as well as for banded or skyline matrices. The fundamental decision for the design of the solver classes for linear systems and eigenproblems is therefore to consider the coefficient matrices as part of the solver objects. Solvers create and initialize coefficient matrices internally. Clients can obtain a reference of type *Matrix* to that internal matrix object in order to provide the actual values. For the exchange of information about matrix properties, special *MatrixInfo* objects are employed. The corresponding classes are shown in Figure 5.

Currently, various direct solvers for linear systems based on NAG and LAPACK subroutines are included. For eigenproblems, there are solvers that are based either on LAPACK or ARPACK which uses an implicitly restarted Arnoldi method and is suitable for large problems.

4 Numerical Examples

The experiments are carried out on three levels in order to thoroughly evaluate the performance of the presented hybrid concept in comparison to pure Java code and to pure Fortran code. Of special interest is the performance penalty introduced by JNI and the object oriented layer that adds a further level of abstraction.

First of all, fundamental operations like matrix-matrix multiplication or matrix-vector multiplications are investigated. On the second level, linear systems are solved using the hybrid approach and pure Java programs. In many scientific computations, most time is spent on matrix multiplication and solving linear systems. Therefore, the results on these two levels can be adopted as a benchmark to generally indicate the efficiencies of different implementations. Finally, a full finite element analysis is carried out to complete the comparison. The analysis contains numerous essential operations of the first two levels.

The computing platform for the experiments is a Linux system with two 2.2GHz AMD Opteron processors. Sun's JDK version 1.5.0_06 is used; Java programs are run using the server virtual machine. Compilers for C and Fortran code are from the gnu compiler suite version 3.3.5. The BLAS and LAPACK implementations are from the AMD Math Core Library v. 3.0.0 and the NAG Fortran Library is Mark 21. Experiments on other platforms and/or with other libraries and compilers have shown similar results.

4.1 Matrix Multiplication

For the performance tests of essential basic linear algebra operations, the multiplication forms

$$\begin{aligned}\mathbf{C} &\leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}\\ \mathbf{y} &\leftarrow \alpha\mathbf{Ax} + \beta\mathbf{y}\end{aligned}$$

are used where **A**, **B**, **C** are matrices, **x**, **y** are vectors and α, β are scalars. Experiments are driven for the five different implementations of the operations collected in Table 2.

For each implementation, tests are executed for different sizes: Matrix sizes range from 1 to 250 in the matrix-matrix multiplication tests and from 1 to 500 in the matrix-vector multiplication tests. Experiments for larger matrix sizes do not result in significant additional information. Computing time for a specific task is a random variable. Therefore, the same calculation is repeated 100 times. The 10 maximum and the 10

program	implementation	description
Java	direct	operations directly coded in Java
	BLAS	subroutines from native BLAS library
	BLAM	native BLAS via BLAM (see section 3)
Fortran	direct	operations directly coded in Fortran
	BLAS	subroutines from BLAS library

Table 2: Different implementations

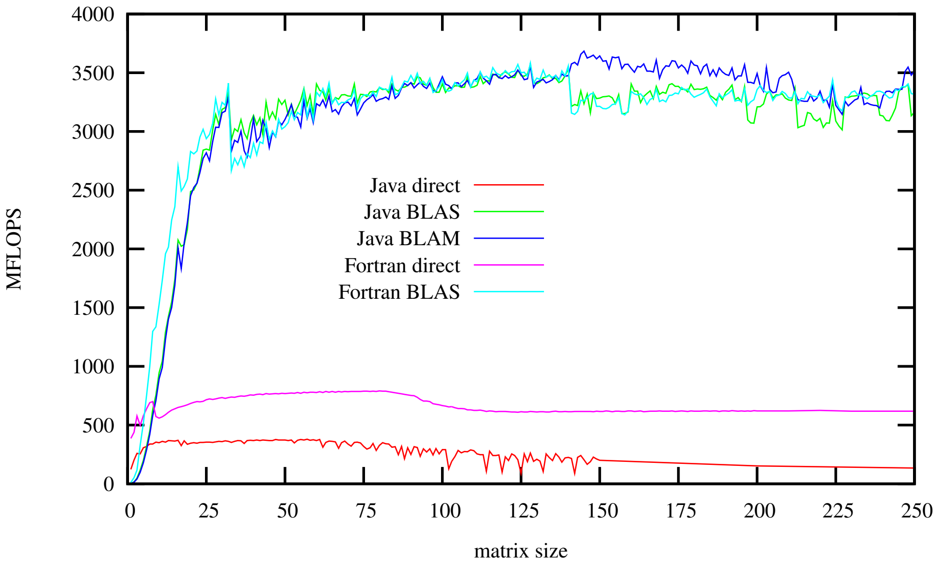


Figure 6: Matrix-matrix multiplication

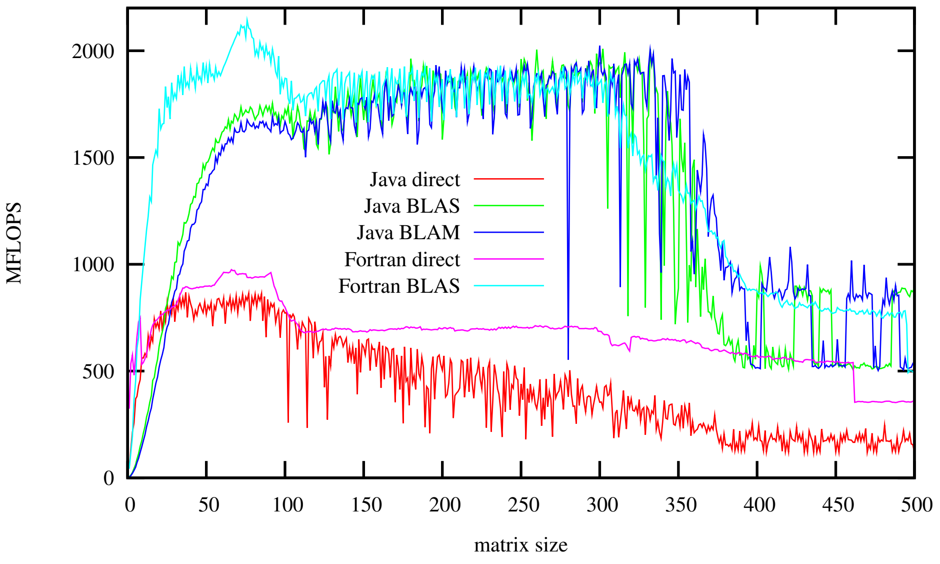


Figure 7: Matrix-vector multiplication

minimum computing times are discarded and the mean value of the remaining 80 values is adopted as final result [1].

The performance measure used to assess the different implementations is million floating-point operations per second (MFLOPS). For matrix-matrix multiplication, the number of floating point operations is $op = 2n^2(n+1)$; for matrix-vector multiplication it can be calculated by $op = 2n(n+1)$ where op is the number of floating point operations and n is the matrix size.

Figure 6 and Figure 7 show the comparison results of different implementations described in Table 2; Figure 6 contains the results for matrix-matrix multiplication and Figure 7 contains the results for matrix-vector multiplication.

First of all, it can be stated that just using Fortran does not a guarantee a fast program. Although the direct Fortran implementation is faster than the direct Java implementation, both approaches are significantly slower than any other solution (Java or Fortran) which is based on the vendor tuned BLAS library. Obviously, full performance can only be obtained by using numerical libraries that are carefully optimized for the actual platform.

Invoking a library function from Java using JNI is associated with some computational overhead (see Figure 2). This performance penalty can be assessed by comparing the curves for *Java BLAS* and *Fortran BLAS*. For matrix-matrix multiplication, there is a slight performance drawback for matrices of a size smaller than 25. In the matrix-vector case, which is computationally less intense, the performance impact of JNI is stronger. Up to a matrix size of about 75, there is a severe influence. For instance, if the matrix size is 25, roughly 50% of the performance is lost. Up to a matrix size of 100, the performance penalty is at maximum about 20%, above that size it becomes negligible.

In the *BLAM* class, the best algorithm for the actual parameter types is dynamically determined. By comparing the curves for *Java BLAS* and *Java BLAM*, it can be seen that this additional effort is not significant. Only for very small matrix sizes it may become necessary to hand-tune the code.

4.2 Solution of Linear Systems

In the second level, the numerical experiments are executed to solve the linear system. The performance of the hybrid approach is compared to the Java program.

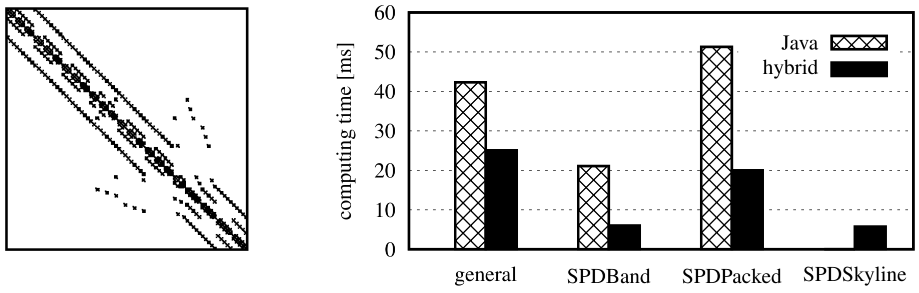
The test data of matrices of linear problems are taken from the matrix market repository [14]. The matrices used for the tests are listed in Table 3. All matrices are real symmetric positive definite.

In the numerical experiments, four types of solvers (see section 3) are applied: *GeneralMatrixLSESolver*, *SPDBandMatrixLSESolver*, *SPDPackedMatrixLSESolver* and *SPDSkylineMatrixLSESolver*. For the first three solvers, which are from LAPACK, two implementations are available: pure Java and hybrid Java/native code.

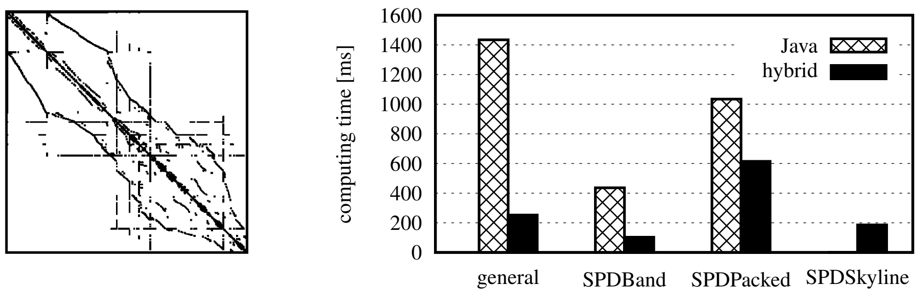
[1] Internally, a second loop is used to achieve a computing time measurable by the system clock

name	application	size	entries	bandwidth
NOS5	3 story building	468×468	2820	358
BCSSTK08	TV studio	1074×1074	7017	1082
S3RMT3M3	shell analysis	5357×5357	106526	10606

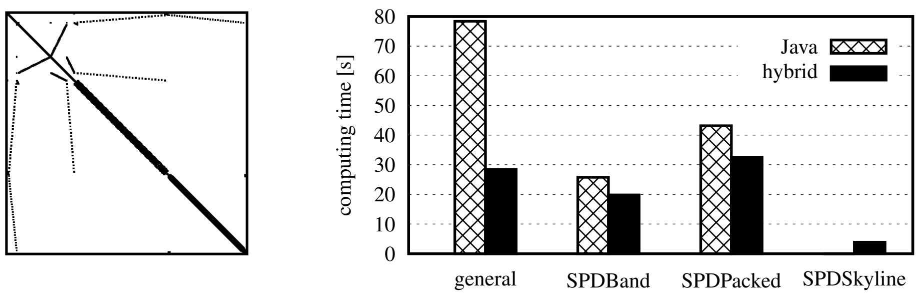
Table 3: Matrices for Experiments of Solution of Linear Systems.



(a) NOS5 matrix (468 x 468)



(b) BCSSTK08 matrix (1074 x 1074)



(c) S3RMT3M3 matrix (5357 x 5357)

Figure 8: Solution of linear systems experiments

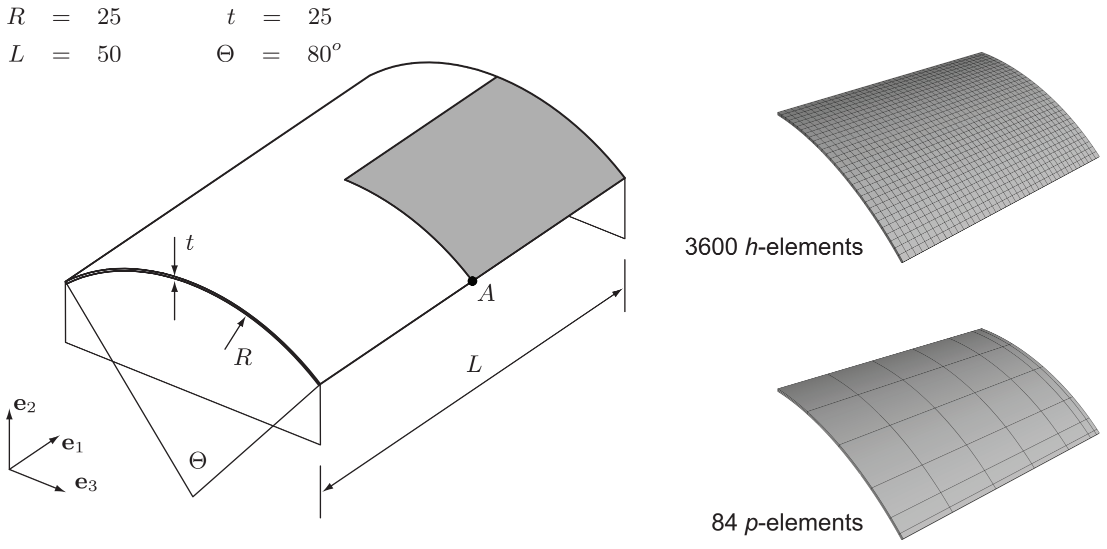


Figure 9: Scordelis-Lo Shell: System and Discretization Variants

Hereby, Java code is translated from the Fortran sources using a Fortran to Java converter. The *SPDSkylineMatrixLSESolver* is from the NAG library and is only available as native code.

All experiments are carried out as described in the previous section. Figure 8 demonstrates the experiment results. The left hand side illustrates the matrix structure, on the right hand side is the computing time for the different solvers. From an algorithmic point of view, the results show the efficiency of the underlying solution algorithms. Naturally, for sparse matrices algorithms, that exploit such a structure by using a banded or skyline storage scheme, are faster than algorithms which don't. From a programming language point of view, the experiments indicate that the hybrid concept yields performance improvements up to a factor of 5 compared to pure Java versions. Taking into account that a program may spend hours solving a linear system, this is a remarkable result.

4.3 Finite Element Analysis

This final numerical experiment is concerned with the overall performance of a Java-based finite element system [2]. Figure 9 shows the structure under investigation: It is the Scordelis-Lo shell which is a classical test problem for shell analysis. In this example, the structure is discretized with volume elements using two different approaches, the h-version and the p-version of FEM. (i) In the h-version, the model consists of 8-node brick elements whereas different meshes are investigated. Computations are carried out starting with 20×15 elements up to 115×86 elements. Figure 9 shows an intermediate mesh with 3600 elements. The finest mesh results in a stiffness matrix of size 59,800 and a bandwidth of 1757. (ii) In the p-version, the mesh consists of 84 high-order hexahedral elements that are arranged in two layers. Here, the number of elements remains unchanged for the different runs but the polynomial degree is raised from 1 to 7. For the element basis functions, an isotropic trunk space is employed (detailed information about the p-version of FEM for three-dimensional continua can be found e.g. in [8]). The finest mesh with $p = 7$ yields a stiffness matrix of size 16,996 having a bandwidth of 9247 which is significantly larger than for the h-version.

In order to assess the effect of the hybrid approach, two series of computations are carried out. In the first series, the hybrid approach described in this paper is used. For the second series, a purely Java based version of the object-oriented numerical library described in Section 3 is applied.

The performance results in Table 4 and in Figures 10 and 11 clearly show the effect of combining native code with Java. In the table, total computing time and the time for preprocessing, solving, and postprocesssing is listed. Also included is the relative amount of time for the individual tasks.

For the h-version of FEM, the overall computing time is reduced by a factor of 5. In this case, mostly all the time is spent solving the linear system. Figures 10(a) and 10(b) show that the time effort is nearly independent from the system size. Therefore, the performance gain caused by the native solver fully pays off. Additionally, the performance of the Java software is compared to a commercial product implemented in Fortran. For that, the finest h-version mesh has been analyzed using ANSYS. The computing time of ANSYS is 16 s compared to 22 s for the hybrid Java software. Taking into account that ANSYS is a mature product that has been optimized for years, this is a good result.

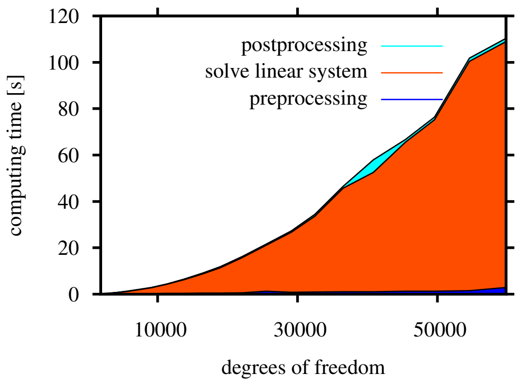
In the p-version of FEM, time is spent not only in solving the linear system but also in setting up the element matrices. Both tasks are significantly speeded up by native code: Preprocessing by a factor of 3.2 and solving the linear system by a factor of 2.6 which is also the overall speedup factor.

		preproc.	solve	postproc.	total
h-version	pure Java	2.9 (2.6%)	106.9 (96.9%)	0.5 (0.5%)	110.3
	hybrid	1.1 (5.1%)	20.4 (92.6%)	0.5 (2.4%)	22.0
	Ansys	–	–	–	16.0
p-version	pure Java	173.9 (32.0%)	345.6 (63.5%)	24.4 (4.5%)	543.9
	hybrid	54.4 (26.1%)	130.6 (62.5%)	24.0 (11.5%)	209.0

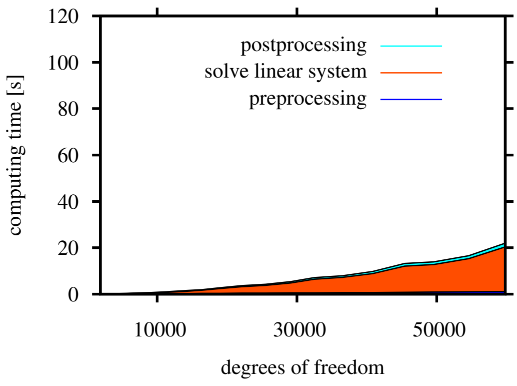
Table 4: Computing time in seconds for h-version and p-version, finest mesh each

5 Conclusion

The application examples show that the performance of Java applications can be significantly enhanced by delegating numerically intensive tasks to native libraries. The performance penalty implied by the overhead of calling a subroutine through JNI is

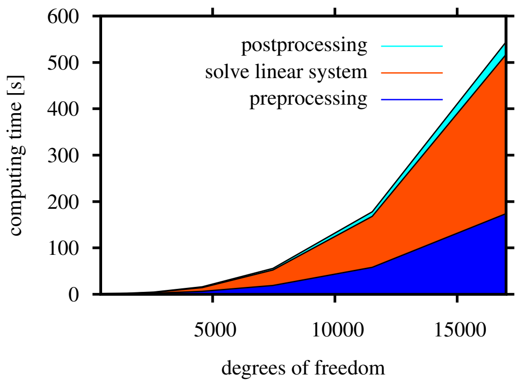


(a) pure Java

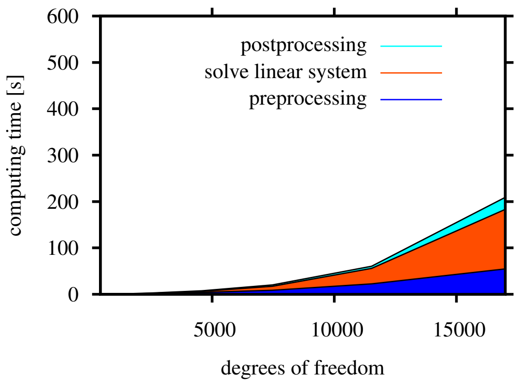


(b) hybrid Java/Fortran

Figure 10: Computing time for the h-version, meshsize (115×86)



(a) pure Java



(b) hybrid Java/Fortran

Figure 11: Computing time for the p-version, polynomial degree $p = 7$

negligibly small when the operations to be carried out have a certain complexity. For a typical finite element application this is mostly always the case.

The hybrid approach is not only advantageous in terms of performance but also gives Java programmers potential access to a large number of already existing high quality numerical libraries. For that, the developed code-generator *genjni* plays an essential role. By that, it is possible, to wrap libraries that contain a large number of subroutines without much effort, if suitable C or C++ headers are available.

The introduced object-oriented layer adds a level of abstraction to the wrapped native libraries that enables a modern, object-oriented programming style. In association with the native libraries, it can be used to create high performance numerical applications in Java.

References

[1] The AMD Core Math Library (AMD ACML).
`http://developer.amd.com/acml.aspx`.

[2] M. Baitsch and D. Hartmann. Object oriented finite element analysis for structural optimization using p-elements. In K. Beucke, B. Firmenich, D. Donath, R. Fruchter, and K. Roddis, editors, *X. ICCCBE-Digital conference proceedings*, Weimar, 2004.

[3] M. Baitsch and D. Hartmann. Towards lifetime optimization of hanger connection plates for steel arch bridges. In K.J. Bathe, editor, *Third MIT Conference on Computational Fluid and Solid Mechanics*, pages 1213–1217, Cambridge, 2005. Elsevier Science.

[4] R. F. Boisvert, J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. *Concurrency: Practice and Experience*, 10(11–13):1117–1129, 1998.

[5] R.F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Numerical computing in Java. *Computing in Science and Engineering*, 3(2):18–24, 2001.

[6] J. M. Bull, L. A. Smith, C. Ball, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. *Concurrency and Computation: Practice and Experience*, 15:417–430, 2003.

[7] H. Casanova, J. Dongarra, and D. M. Doolin. Java access to numerical libraries. *Concurrency: Practice and Experience*, 9(11):1279–1291, 1997.

[8] A. Düster. *High order finite elements for three-dimensional, thin-walled nonlinear continua*. Dissertation, Technische Universität München, 2001.

[9] G. Fox, X. Li, Z. Qiang, and W. Zhigang. A prototype Fortran-to-Java converter. *Concurrency: Practice and Experience*, 9(11):1047–1061, 1997.

[10] XML output for GCC (GCCXML).
`http://www.gccxml.org`.

[11] V. Getov, S. F. Hummel, and S. Mintchev. High-performance parallel programming in Java: exploiting native libraries. *Concurrency: Practice and Experience*, 10(11–13):863–872, 1998.

[12] M. Grand. *Patterns in Java*. Wiley, 2002.

[13] JAMA: A Java Matrix Package.
`http://math.nist.gov/javanumerics/jama`.

[14] Matrix market.
`http://math.nist.gov/MatrixMarket/`.

[15] The Intel Math Kernel Library (Intel MKL).
`http://www.intel.com/software/products/mkl/overview.htm`.

[16] Matrix Toolkits for Java (MTJ).
`http://rs.cipr.uib.no/mtj`.

[17] N. Muhtaroglu. Development and evaluation of a Java package for linear algebra based on native code. Master's thesis, Lehrstuhl für Ingenieurinformatik im Bauwesen, Ruhr-Universität Bochum, 2005.

[18] K. Reinholtz. Java will be faster than C++. *ACM Sigplan Notices*, 35(2):25–28,

2000.

[19] C.J. Riley, S. Chatterjee, and R. Biswas. High-performance Java codes for computational fluid dynamics. *Concurrency and Computation: Practice and Experience*, 15:395–415, 2003.

[20] W. B. VanderHeyden, E. D. Dendy, and N. T. Padial-Collins. CartaBlanca—a pure-Java, component-based systems simulation tool for coupled non-linear physics on unstructured grids. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 134–142. ACM Press, 2001.