Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 3 (2011) 474–478

Procedia Computer Science

www.elsevier.com/locate/procedia

WCIT-2010

Evaluating Java performance for linear algebra numerical computations

Bogdan Oancea[a] *, Ion Gh. Rosca[b], Tudorel Andrei[b], Andreea Iluzia Iacob[b]

[a]*Nicolae Titulescu" University, 040051, Bucharest, Romania,*
[b]*The Bucharest Academy of Economic Studies, 010374, Bucharest, Romania*

Abstract

In this paper we evaluate the performance of a Java library in the context of designing numerical solutions for linear algebra problems. Nowadays there are a number of libraries like LAPACK or ATLAS available in C or FORTRAN for solving such problems. Java is a relatively new object-oriented language and is almost universally recognized as a very good programming language for writing portable programs. Despite these advantages, it is a common believe that Java still lags behind C/C++ or Fortran performance especially for computational intensive numerical applications. We developed a Java library for matrix computations and show that using a set of optimization techniques Java can achieve performance comparable with other libraries developed in C or Fortran.

Selection and/or peer-review under responsibility of the Guest Editor.

Keywords: Java; numerical computing; linear algebra.

1. Introduction

Linear systems of equations can be found in almost every type of scientific and engineering numerical computing applications. Therefore, we will pay a special attention to develop high performance algorithms and software packages to solve linear algebra problems and especially to linear systems of equations. There are two methods for solving such a system: direct methods, based on matrix factorization, and iterative methods. In this paper we will focus on the direct methods and we will implement these algorithms using the Java programming language. Measuring the performance of our matrix Java library we show that Java can obtain results comparable to those of C or FORTRAN libraries.

Java has become an important platform for software development and deployment. Since 1995, the year of its creation, Java grew exponentially in many directions: from e-banking to mobile devices, from data mining to Web applications, from econometric modeling to fundamental physics or image processing. Nowadays Java is the foundation for deploying servlets, applets, Web services, large commercial, academic or governmental applications. Therefore we evaluate the performance of a Java library in the context of designing numerical solutions for linear algebra problems.

* Bogdan Oancea. Tel.: +40-723745035; fax: +40-213308606.
E-mail address: oanceab@ie.ase.ro

1877-0509
doi:10.1016/j.procs.2010.12.080

2. Problems regarding Java as a language for numerical computing

Java has a great potential to become an excellent environment for developing large-scale applications with numerical intensive kernels. We can list several features that make Java an attractive programming environment for such applications: Java supports object-oriented model, Java has a built-in support for graphics and networking, Java programs are portable – "write once run everywhere", Java will be pervasive in the environment surrounding and driving numeric intensive applications (GUI, client-server code etc);

Despite these advantages, Java still has some unresolved issues especially concerning numerical computing. Among the critical issues that make Java programs performance to lag behind C or Fortran programs we can mention:

- *Multidimensional arrays:* these are the most common data structures in numerical computing, especially in linear algebra but currently Java doesn't support multidimensional arrays. Most of the numerical computation programs work with matrices, which are two-dimensional arrays. Java represents a two-dimensional array with vectors of vectors: one-dimensional arrays whose elements themselves are one-dimensional arrays. According to Moreira et. al. [8], [9] this approach presents two main difficulties to optimizing compilers: *shape volatility* and *aliasing disambiguation*. Another disadvantage of these kinds of arrays arises from the run-time checks;
- *complex arithmetic*: complex numbers are essential in many areas of scientific computing and they must be supported in Java;
- *the absence of templates, like to ones in C++.* Although, starting with Java 5, there is some support for using *generics*, this is not what the *template* mechanism provides to C++. Especially, if we want to implement a Matrix class, we have to write separate implementations for single precision, double precision or integer matrices.
- *operator overloading:* Java doesn't support operator overloading like C++. This makes the Java code not so easy to read like the equivalent C++ code;
- *use of floating point hardware:* applications performance can benefit from the exploitation of distinctive features of hardware. Such examples include the fused add-and-multiply instruction in some microprocessor architectures.

3. Recent improvements of the JVM performance

Although Java is an interpreted language and one can say that is slow for numerical computing, recent advances in Java compiler and Java virtual machine greatly improves the performance of Java programs. The Java HotSpot™ virtual machine implementation which is Sun Microsystems, Inc.'s virtual machine for the Java platform has known as series of considerable improvements regarding the performance. Currently there are two implementations of the Java virtual machine:

- Java HotSpot Client VM, which is tuned for reducing application start-up time and memory requirements.
- Java HotSpot Server VM, which is tuned for execution speed.

The Server virtual machine incorporates an advanced adaptive compiler that supports many of the optimizations performed by modern C++ compilers, and some optimizations that cannot be done by traditional compilers like aggressive inlining across virtual method invocations.

The Java HotSpot virtual machine implements a new and improved garbage collector and leading-edge techniques for both uncontended and contended synchronization operations which improves synchronization performance by a large factor.

The Java HotSpot virtual machine solves the performance issues by using adaptive optimization technology. Adaptive optimization uses the following programs property: almost all programs spend the majority of their time executing only a small part of their code. The Java HotSpot virtual machine starts the program using an interpreter, and analyzes the code as it runs to detect the "hot spots" of the program. Then it uses a number of native-code optimization techniques on these hot spots. The virtual machine is continuously monitoring the hot spots as the program is executing, so that it can adapt the optimization as program continues to run. The optimization techniques of the HotSpot complier includes all the classic optimizations like dead code elimination, common subexpression elimination, loop unrolling, constant propagation, and global code motion.

Sun Microsystems [12] indicates that the main compiler optimizations are: deep inlining and inlining of potentially virtual calls, fast instanceof/checkcast, range check elimination, loop unrolling, feedback-directed optimizations - the Java virtual machine profiles the program execution in the interpreter before compiling the Java

bytecode to optimized machine code and the profiling information is used later by the compiler to more aggressively and optimistically optimize the code in certain situations.

4. The Java linear algebra library

There are two alternatives to use a linear algebra library in JAVA: use an existing library, possibly developed in C, C++, FORTRAN through native calls or to implement the linear algebra package entirely in Java. Both approaches have advantages and disadvantages. The first one has the advantage of a little programming effort and a relatively high performance of the native libraries. The main disadvantage is lack of code portability and the absence of any guarantee of results reproducibility because of the differences in the native library implementations. For the second choice, the disadvantage would be a great programming effort to develop an entirely library in Java and the main advantage is the portability of the library – "write once run everywhere".

We have chosen to develop an entirely new Java linear algebra library. Until now only little effort has been made to develop such libraries: Jama (Boisver, [3]), Jampack (Stewart, [11]), Colt (Hoschek, [7]).

Many numerical need efficient subroutines to work with vectors/matrices and to perform basic linear algebra operations on them. There are many software packages in the area of matrix computations like BLAS (Dongarra, [4],[5]) and LAPACK (Anderson et al., [2]), but these packages are written in FORTRAN or C; Java still needs such a package.

We developed JLA – a Java package which implements a subset of the basic linear algebra subroutines frequently encountered in scientific applications. In order to achieve a good performance, matrices are stored using a C-like row-major layout because this layout enables cache reusing. Before matrices are transferred to computation routines, they are transformed from to one-dimensional arrays by a conversion method.

All JLA subroutines works with the one-dimensional form of the matrices but the result can be converted back to the Java arrays of arrays. The names of our routines were kept the same as the in the well known BLAS and LAPACK libraries. Table 1 shows a list of the currently implemented routines in JLA.

Table 1. Linear algebra routines implemented in JLA package

BLAS routines implemented in JLA	LAPACK routines implemented in JLA
Dasum, Daxpy, Dcopy, Ddot, Dscal, Dger, Dnrm2, Dswap, Dsyrk Dtrsm, Dgemv, Dgemm	Dgetf2, Dgetrf, Dgetrs, Dgesv, Dpotf2, Dpotrf

In the implementation of the linear algebra routines we used a series of techniques that improves the performance (Hammond, [6]):

Loop Unrolling. This method replaces the body of a loop with several copies, adjusting the loop control code such that the body of the new loop executes the exactly the same instructions as the initial loop but with a smaller proportion of execution spent on evaluating the control instruction. In the JLA library we used an unrolling factor of 5. This is a tradeoff between the improvement in the overall performance and the maintainability of the code. Also, a large unfolding factor may have the opposite effect by making the code too large to fit into cache lines in the processor.

Loop Unfolding. Loop unfolding removes a number of the first iterations of a loop and places them before the main body. Unfolding has an important advantage: it allows the earlier iterations of the loop to execute without requiring the processor to follow jump instructions back to the beginning of the loop, improving the ability of the code to be pipelined. Both loop unrolling and loop unfolding can increase the size of the code, but in our case this increase was insignificantly.

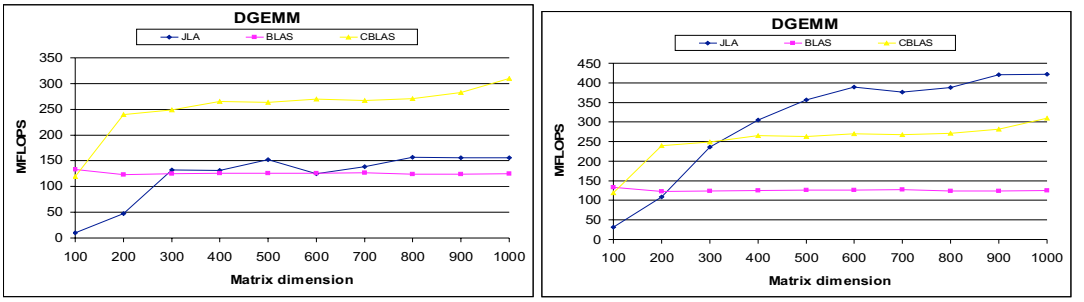
Loop Invariant Code Motion. Loop invariant code motion consist in moving out of the loop the code within the loop that does not change on each iteration of the loop, i.e. the code whose execution is independent of the loop variable. Loop invariant code motion has the effect of a faster execution of the loops because the redundant code is eliminated from being executed multiple times saving processor resources.

Data Flow Optimizations: common subexpression elimination, constant folding and propagation;

Other Optimizations: dead code elimination, inline expansion, strength reduction, factoring out of invariants.

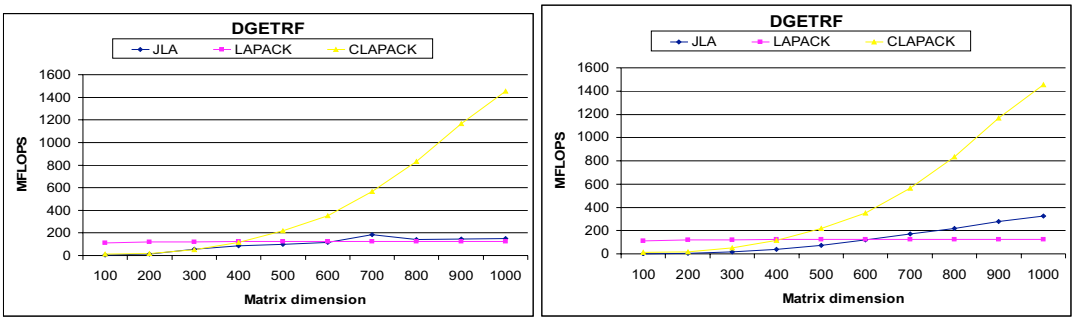
We run a series of experiments to evaluate the performance of the JLA library. For these experiments we used a machine with AMD Sempron 3000+ processor and 1GB of main memory. We developed our library using JDK 6 update 16 under the Linux operating system. We compared the performance of the Java JLA library with the BLAS and LAPACK performance (the FORTRAN versions) and with their C language versions CBLAS and CLAPACK. The BLAS and LAPACK libraries were the reference implementations downloaded from www.netlib.org. The experiments used only the double precision matrices. We also compared the performance of the JLA library run by the Client JVM and the Server JVM. The results are presented in figures 1 and 2.

Figure 1 shows the matrix multiplication performance for the JLA routine run with Client (a) and Server JVM (b). As one can see, the MFLOP rate for the Java code on Client JVM is superior with about 15-20% to the MFLOP rate of the FORTRAN BLAS routine but inferior to the CBLAS routine, achieving about 65% of the C routine performance. Surprisingly, with the Server JVM, the Java code performs better even than the CBLAS for matrix dimensions greater than 300. It outperforms the C routine being 2-3 times faster. Figure 2 shows the LU matrix factorization performance for the JLA routine run with Client (a) and Server JVM (b). The MFLOP rate for LU factorization is comparable or even greater than the LAPACK routine, but is much lesser than the CLAPCK routine especially for large matrices. These results are very encouraging, showing that Java is no longer a slow environment for numerical computing. Using the state-of-art HotSpot JVM and different optimization techniques, the Java code implemented in our JLA library obtains even better results than the equivalent FORTRAN or C code.



a) Client JVM b) Server JVM

Figure 1. Matrix multiplication performance



a) Client JVM b) Server JVM

Figure 2. Matrix LU factorization performance.

5. Conclusions

We have developed a Java library that implements linear algebra operations that appear in the solutions of many scientific or engineering applications without the use of any native library. This approach has the advantage of portability, a general characteristic of any Java program. We also achieved a high performance from our library. Experiments showed that for Client JVM, the JLA matrix multiplication runs with 15-20% faster than the BLAS (FORTRAN version) and the for Server JVM, the JLA outperforms the BLAS routine, being 2-3 times faster than the equivalent BLAS routine for matrices with dimensions greater than 200. Comparing with the CBLAS version, the JLA run by the Client JVM achieves about 65-70% of the performance of the C routine and the Server JVM is about 60% faster than the C routine. The results for the LU factorization show that the Java routine achieves a performance comparable with the one of the equivalent FORTRAN routine but this performance is lesser than the C version of the LU factorization for both Client and Server JVM.

These results show that the recent enhancements made to the Sun JVM turned Java into a very fast platform suitable for numerical computation.

Acknowledgements

The work related to this paper was supported by the CNCSIS–UEFISCSU projects PNII – IDEI ID_1814 and ID_1793.

References

1. Almasi G., Gustafson, F.G., Moreira, J.E., Design and Evaluation of a Linear Alebgra Package for Java, In Proceedings of the ACM 2000 Java Grande, (2000), pp. 150-159 .
2. Anderson, E., Bai, Z., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S. , McKenney, A. , Ostrouchov, S., Sorensen, D., *LAPACK Users's Guide*, SIAM, Philadelphia, 1992.
3. Boisvert, R.F., Dongarra, J., Pozo, R., Remington, K.A., Stweart, G.W., Developing numerical libraries in Java, ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.
4. Dongarra, J., Du Croz, J., Hammarling, S., and Duff, I., A set of level 3 basic linear Algebra subprograms, ACM Trans. Math. Soft., 16,(1), (1990), pp. 1-17.
5. Dongarra, J., Du Croz, J., Hammarling, S., and Hanson, R., An extended set of FORTRAN basic linear algebra subprograms, ACM Trans. Math. Soft., 14, (1), (1988), pp. 1-17.
6. Hammond, S., and Lacey, D., Loop Transformations in the Ahead-of-Time Optimization of Java Bytecode, Proceedings of the 15th International Conference on Compiler Construction, Springer Verlag, 2006.
7. Hoschek, V., Colt: Open Source Libraries for High Performance Scientific and Technical Computing in Java, http://dsd.lbl.gov/~hoschek/colt.
8. Moreira, J.E., Midkiff, S.P., Gupta, M., A Standard Java Array Package for Technical Computing, IBM Research Report RC 21369, 1998.
9. Moreira, J.E., Midkiff, S.P., Gupta, M., From flop to Megaflop: Java for technical Computing, Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, 1999.
10. Moreira, J.E., Midkiff, S.P., Gupta, M., Artigas, P.V., Snir, M., and Lawrence, R.D., Java programming for high performance numerical computing, IBM Systems Journal, 39(1), (2000), pp. 21–56.
11. G.W. Stewart, JAMPACK-A Java Package for Matrix Computations. ⟨ftp://math.nist.gov/Jampack/Jampack/AboutJampack.html⟩.
12. Sun Microsystems, The Java HotSpot Performance Engine Architecture, WhitePaper, 2004.