The cost of being object-oriented: A preliminary study

Zoran Budimlić*, Ken Kennedy and Jeff Piper

Center for Research on Parallel Computation, Rice University, CRPC – MS 41, 6100 Main Street, Houston, TX 77005-1892, USA
E-mail: zoran@rice.edu

Since the introduction of the Java programming language, there has been widespread interest in the use Java for the high performance scientific computing. One major impediment to such use is the performance penalty paid relative to Fortran. To support our research on overcoming this penalty through compiler technology, we have developed a benchmark suite, called OwlPack, which is based on the popular LINPACK library. Although there are existing implementations of LINPACK in Java, most of these are produced by direct translation from Fortran. As such they do not reflect the style of programming that a good object-oriented programmer would use in Java. Our goal is to investigate how to make object-oriented scientific programming practical. Therefore we developed two object-oriented versions of LINPACK in Java, a true polymorphic version and a “Lite” version designed for higher performance. We used these libraries to perform a detailed performance analysis using several leading Java compilers and virtual machines, comparing the performance of the object-oriented versions of the benchmark with a version produced by direct translation from Fortran. As such they do not reflect the style of programming that a good object-oriented programmer would use in Java. Our goal is to investigate how to make object-oriented scientific programming practical. Therefore we developed two object-oriented versions of LINPACK in Java, a true polymorphic version and a “Lite” version designed for higher performance. We used these libraries to perform a detailed performance analysis using several leading Java compilers and virtual machines, comparing the performance of the object-oriented versions of the benchmark with a version produced by direct translation from Fortran. Although Java implementations have been made great strides, they still fall short on programs that use the full power of Java’s object-oriented features. Our ultimate goal is to drive research on compiler technology that will reward, rather than penalize good object-oriented programming practice.

1. Introduction

Since the introduction of the Java programming language, there has been widespread interest in the use of Java for high performance scientific computing. The principal impediment to such use is poor performance, relative to the similar programs written in Fortran and C. There are three main reasons why Java programs do not achieve high performance by comparison with Fortran and C:

- Java compilers and execution environments are not yet on par with the traditional optimizing compilers. Although there has been a significant advancement in this area lately [2,7,9], especially with the run-time compilation and optimization techniques, Java systems still have to improve to be able to compete with the traditional languages.
- The non object-oriented features of Java add significant overhead. Bytecode portability requires that major optimizations be delayed until run time. Garbage collection, synchronization, and the exception mechanism all require additional overhead for their implementation. Java security measures force the virtual machine implementation to examine the code for security holes before execution. All these requirements, important as they are, reduce the performance of Java programs at run time.
- Java is an object-oriented language, and as such it encourages programmers to use object-oriented style when writing scientific programs. It is far more natural for the programmers to think about matrices, vectors and complex numbers as objects and pass them around and use encapsulation and code reuse when performing operations on them, than to perform all the operations directly on the Fortran-style arrays. Later in the paper, we show that Java compilers (both static and JIT) are not yet up to the task of effectively optimizing away the overhead resulting from using the object-oriented style.

In spite of the cost, much of the value of using Java is lost if the programmer does not freely use the advanced features of the language, particularly the support for object-oriented program development. The right solution is to build compiler systems that minimize the penalties for fully utilizing the features of the language. However, effective research on Java compiler systems must be driven by experimental methods.
Without good benchmarks on which to conduct these experiments, it will be difficult to validate the compiler strategies proposed by researchers.

The majority of benchmarks available for evaluation of the cost of using Java in high performance scientific computing are either microbenchmarks or benchmarks obtained by direct translation from Fortran (either automatic [4] semi-automatic or manual [11]). Neither of these two closely resemble the programs that Java programmers would prefer to write. The need for a benchmark that would closely reflect the ‘real world’ scientific computation in Java is clear. Unfortunately, although there have been some reports of scientific applications implemented in Java that could be easily converted to serve as benchmarks, we suspect that many of these have been translated to Java without a corresponding conversion to true object-oriented programming style. A good example is the Java version of the LINPACK Benchmark, which strongly resembles the Fortran version.

To address this issue and to help foster more research on Java compilation, we have designed and implemented in Java an object-oriented version of the LINPACK linear algebra library. We call this library OwlPack (Objects Within Linear algebra PACKage). We used OwlPack to perform a detailed analysis of the performance of Java programs written in different programming styles. Specifically, we compared the performance of the object-oriented version of the benchmark with a version written in a style closer to Fortran. We then analyzed the cost of the additional overhead incurred when object-oriented design is used in high-performance computing.

The principal contributions of this research are as follows:

- We have constructed an object-oriented Java implementation of the LINPACK library that is available to the public (http://www.cs.rice.edu/~zoran/OwlPack.zip) for evaluating the performance of Java compilers and run-time systems.
- We have used this package to evaluate the cost that is associated with object-oriented design of scientific programs using the best Java compilers and VMs available today.
- We have analyzed the results of these experiments to illuminate important issues that must be addressed by Java compiler and run-time systems research if we are to improve the performance of scientific programs written in Java.

The rest of this paper is organized in the following way: in Section 2 we describe the design of our implementation of the OwlPack, present the class hierarchy and explain some design decisions. In Section 3 we present the experimental results we obtained by executing several of our routines on various VMs and using different static compilers, along with the same experiments performed on a Fortran-style version of LINPACK library. In Section 4 we provide some insights and guidance for compiler designers, based on the results reported in Section 3. In Section 5 we describe some of the work related to the topic of this paper and future research plans.

2. OwlPack design

The creation of a general framework for classes in OwlPack is motivated by the primary goal of employing the most natural hierarchy possible while achieving the highest level of abstraction. The central design decision was how to store the elements of a matrix. Using two-dimensional array of primitive types, like double and float, permits direct manipulation of the data and does not force translation during parameter passing to the library, since the users would be likely to have their data in this form. However, this approach suffers from the disadvantage that the class hierarchy would have to be replicated for each primitive type.

On the other hand, a polymorphic solution could be achieved by creating a new abstract class of numbers to which any primitive type can be translated. This approach allows for a single implementation for all primitive types and can be extended to any type for which the required operations are defined. Therefore, it is easier to program, easier to maintain, and more general than the specializing the code to each primitive type. In that sense, it uses the full power of the object-oriented features of the language.

As an experiment, we decided to explore both styles. We created two different matrix abstractions, Matrix and NMatrix, that represent the different storage strategies while performing the same numeric computations. The Matrix class is the superclass of all classes that are specialized based on primitive types and gives rise to a replicated class hierarchy parametrized by primitive element type. We call the resulting style of programming “lite” object-oriented style (Lite OO) style.

The classes that derive from NMatrix all use the abstract number class, LNumber. Thus NMatrix is fully polymorphic and can be extended to new number types
easily. We will refer to this style of programming as object-oriented (OO) style.

FMatrix, DMatrix, and CMatrix all extend Matrix and contain the BLAS (Basic Linear Algebra Subroutines) for the corresponding data type (F = float, D = double, C = complex). The CMatrix class is further specialized for double and single precision complex numbers. In our current implementation, the complex numbers used in CMatrix are objects that are an extension of our LNumber class, although consistency with the efficiency goals of the Lite OO style would dictate that CMatrix use pairs of two-dimensional arrays of primitive types – one array for the real part of the complex numbers and one array for the imaginary parts. We plan to make this change in a future version. All classes whose names begin with an N are extensions of NMatrix and use LNumbers.

For all classes, the next differentiation is based on the storage type, so DMatrix is extended by DBanded, DPacked, DTDiag, and DFull (and similarly for FMatrix, CMatrix and NMatrix). By separating the matrices by storage form, the classes further down the hierarchy can reuse code because the internal variables are accessed the same way.

To attain a high degree of generality, Matrix and NMatrix contain the size information, an internal pivot array and the base interface for all of the methods available for matrices. These method are, in general, overridden by the extended classes. However, the base functions throw an exception whenever the base method is not overridden because it does not always make sense to provide every method for every extension. For example, with banded matrices it does not make sense to implement the inverse operation, because the inverse would require more storage than the allocated for the original Matrix. In the interest of abstracting these functions, our solution is to throw an exception whenever an invalid function call is made. This brings most of the interesting methods to the Matrix level where storage form and data type are not important.

In order to raise solve() and determ() to the highest level, we created a Vector class that is barely more than an array of the primitive type being used in the corre-
for(int i=0; i<cols; i++) {
    if(pivot[i] != i+1) Det[0].negate();
    Det[0].multTo(Mat[i][i]);
    if(Det[0].equal(0)){
        return Det;
    }else{
        while(!Det[0].abs().greaterOrEqual(1)) {
            Det[0].multT(10);
            Det[1].subTo(1);
        }
        while((Det[0].abs()).greaterOrEqual(10)) {
            Det[0].divTo(10);
            Det[1].pplus();
        }
    }
}
return Det;

Fig. 3. Object-oriented style determinant computation.

for(int i=0; i<cols; i++) {
    if(pivot[i] != i+1) Det[0] = -Det[0];
    Det[0] *= this.Mat[i][i];
    if(Det[0] == 0){
        return (new DVector(Det));
    }else{
        while(Math.abs(Det[0]) < 1) {
            Det[0] *= 10;
            Det[1] -= 1;
        }
        while(Math.abs(Det[0]) >= 10) {
            Det[0] /= 10;
            Det[1]++;}
    }
}
return (new DVector(Det));

Fig. 4. "Lite" OO style determinant computation.

The Fortran-style code, while similar in appearance to the second style above, follows Fortran conventions of passing in the Matrix and breaking out of loops. This function also gets a reference to det as an argument and hence there is no need to return it.

DBanded, DFull, and DTDiag all extend DMMatrix and are at the highest level of non-abstract Matrices. These classes contain the Gaussian (LU) decomposition routines since they are the most general routines that can be used by any Matrix with the corresponding storage form. If a Matrix is declared as a sub-class of one of these types, it can be factored and solved with
for (i=0; i < n; i++) {
    if(ipvt[i] != i) det[0] = -det[0];
    det[0] *= a[i][i];
    if (det[0] == 0.0) break;
    while (Math.abs(det[0]) < 1.0) {
        det[0] *= Math.pow(10, ten);
        det[1] --;
    }
    while(Math.abs(det[0]) >= ten) {
        det[0] /= Math.pow(10, ten);
        det[1]++;
    }
}

Fig. 5. Fortran style determinant computation.

Gaussian elimination by calling its super class’s factor
and solve routines. Dpacked is an abstract class which
does not contain the Gaussian routines.

DFull also contains the singular value decomposition,
qr-decomposition, and Cholesky’s updating rou-
tines. The code for Cholesky decomposition is also in
DFull so that both symmetric subclasses can use it, but
it is impossible to call it from DFull.

In designing NMatrix, we encountered some diffi-
culties when dealing with LNumbers, because these
two classes are completely separate in their internal
representation and LNumber is a fully self-supporting
class. There were several instances when a new LNum-
ber was needed inside a NMatrix, but because LNumber
is an abstract class, it was impossible to create a new
instance. To solve this problem, we made LNumber
implement Clonable() and added the methods set-
Zero() and setOne() which would set the value of the
LNumber to zero or one, respectively. An example
of the intended implementation can be found in the
method NFull.determ():

Det[0] = Mat[0][0].Clone();
Det[1] = Mat[0][0].Clone();
Det[0].setOne();
Det[1].setZero();

Another consideration when implementing LNumber
was the way in which methods should be called.
For example, LNumber could have add defined to take
LFloat, LDouble, LCFloat, and LCDouble and throw
an exception if the method is called and not overloaded
for the type, much the way Number.getValue() is
implemented in the standard Java library. Every class
would overload the method that took its own type and
perform the operation quickly because it knows pre-
cisely what the passed object is. While slightly faster,
this method was not considered appropriate for the full
object oriented version, since it would introduce into
LNumber a knowledge of all of the classes that extend
it. Instead, by having the method take an LNumber and
cast it, only one version of the method is written, and
the exception is still thrown if an incompatible LNumber
is passed.

3. Experimental results

We compared our implementation of the LINPACK
library with the partial Java version from FPL Statistics
Group [11], obtained by the straight-line transforma-
tion from the Fortran source code. We will refer to
this version of the code as “Fortran style”. We have
only included the timings for the routines that have
been implemented in the FPL version – factorization
and solving the positive definite matrix, LU and QR
decomposition and solving of the full matrix, inverse
determinant computation and singular value de-
composition. These routines were only available for
double precision floating point numbers, so we com-
pared the running times of their equivalents in our im-
plementation: DPoFull and DFull classes handle these
functions in our “Lite” OO version, while NPoFull and
NFull classes instantiated with LDouble numbers han-
dle them in our OO version. A short description of
these routines is given below:

- dpofa factors a $300 \times 300$ random generated posi-
tive definite matrix
- dposl solves the equation $A \cdot x = B$, where $A$
is the matrix factored by dpofa and $B$ is random
generated vector of 300 numbers
- dpodi computes the determinant and the inverse
of the $300 \times 300$ positive definite matrix
- dgefa performs an LU factorization of a $200 \times
200$ random generated full matrix
- dgesl solves $A \cdot x = B$, where $A$ is factored
full matrix, $B$ is a vector size 200
- dgesl computes the determinant and the inverse
of a full $200 \times 200$ matrix
- dqrdc performs QR decomposition with pivot-
ing on a $300 \times 300$ random full matrix
- dqrsl solves $A \cdot x = B$, where $A$ is QR decom-
posed matrix, $B$ is a vector
- dsvdc performs the singular value decomposi-
tion on a random $100 \times 100$ matrix

The tests on Solaris were performed on the Sun Ul-
tra 5, with 64MB of memory, running under Solaris
2.6, with the jdk 1.1.5 from JavaSoft for the interpreter
Table 1

<table>
<thead>
<tr>
<th>Function</th>
<th>Sparc Ultra 5</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>jdk 1.1.5 interpreter</td>
<td>jdk 1.2 JIT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>“Lite” OO</td>
<td>OO style</td>
<td>“Lite” OO</td>
<td>OO style</td>
<td></td>
</tr>
<tr>
<td>dpofa</td>
<td>4.669</td>
<td>5.548</td>
<td>115.155</td>
<td>1.353</td>
<td>1.584</td>
</tr>
<tr>
<td>dposl</td>
<td>8.820</td>
<td>10.362</td>
<td>127.392</td>
<td>2.582</td>
<td>2.831</td>
</tr>
<tr>
<td>dpoi</td>
<td>12.736</td>
<td>14.315</td>
<td>297.346</td>
<td>2.614</td>
<td>3.462</td>
</tr>
<tr>
<td>dgefa</td>
<td>3.226</td>
<td>3.511</td>
<td>81.981</td>
<td>0.698</td>
<td>0.838</td>
</tr>
<tr>
<td>dgesl</td>
<td>4.079</td>
<td>4.640</td>
<td>35.868</td>
<td>0.865</td>
<td>0.967</td>
</tr>
<tr>
<td>dgedi</td>
<td>6.234</td>
<td>6.971</td>
<td>161.868</td>
<td>1.428</td>
<td>1.677</td>
</tr>
<tr>
<td>dqrdc</td>
<td>21.197</td>
<td>25.139</td>
<td>55.378</td>
<td>6.921</td>
<td>8.118</td>
</tr>
<tr>
<td>dqrsl</td>
<td>14.504</td>
<td>15.757</td>
<td>162.044</td>
<td>3.953</td>
<td>3.903</td>
</tr>
<tr>
<td>dsvdc</td>
<td>9.008</td>
<td>15.439</td>
<td>226.495</td>
<td>1.456</td>
<td>3.043</td>
</tr>
</tbody>
</table>

Table 2

<table>
<thead>
<tr>
<th>Function</th>
<th>Pentium Pro</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>jdk 1.1.6 interpreter</td>
<td>Symantec JIT 3.00.029</td>
<td>Microsoft VM 4.79.2405</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>“Lite” OO</td>
<td>OO style</td>
<td>“Lite” OO</td>
<td>OO style</td>
<td>“Lite” OO</td>
</tr>
<tr>
<td>dpofa</td>
<td>5.528</td>
<td>6.309</td>
<td>242.138</td>
<td>0.591</td>
<td>0.861</td>
</tr>
<tr>
<td>dpoi</td>
<td>13.019</td>
<td>13.860</td>
<td>384.954</td>
<td>1.252</td>
<td>2.183</td>
</tr>
<tr>
<td>dgefa</td>
<td>3.986</td>
<td>4.186</td>
<td>135.414</td>
<td>0.421</td>
<td>0.521</td>
</tr>
<tr>
<td>dgesl</td>
<td>5.398</td>
<td>5.859</td>
<td>45.716</td>
<td>0.571</td>
<td>0.701</td>
</tr>
<tr>
<td>dgedi</td>
<td>7.210</td>
<td>8.032</td>
<td>238.172</td>
<td>0.842</td>
<td>1.041</td>
</tr>
<tr>
<td>dqrdc</td>
<td>24.185</td>
<td>28.521</td>
<td>830.104</td>
<td>3.925</td>
<td>4.466</td>
</tr>
<tr>
<td>dqrsl</td>
<td>18.757</td>
<td>19.969</td>
<td>264.510</td>
<td>2.083</td>
<td>2.264</td>
</tr>
<tr>
<td>dsvdc</td>
<td>11.106</td>
<td>20.639</td>
<td>450.548</td>
<td>1.713</td>
<td>2.594</td>
</tr>
</tbody>
</table>

For reference, we performed the same measurements of the Fortran version of the LINPACK, using the Fortran 90 native compiler for Solaris and the Digital PowerStation F90 compiler for the Windows NT machine. In general, the execution times for Fortran style Java LINPACK on the jdk 1.2 Production Release and on the Microsoft VM were within the factor of four of native Fortran code.

4. Analysis of the results

The data presented in the previous section provides some interesting and somewhat surprising results. The substantial performance hit associated with the object-oriented design was expected, although the magnitude of this cost exceeded our expectations.

In general, the Lite OO version of OwlPack performed slightly slower than the Fortran-style version, with differences ranging from 7% faster for dposl on Symantec JIT, to around 2.7 times slower for dsvdc on Microsoft VM. The main reason for this performance degradation is extra indirection in the innermost
loops of our routines. The Fortran-style library has all the matrices passed by reference to the targeted routines, where the argument of the routine contained the direct reference to the matrices involved in the computation. The “Lite” OO library treats matrices as objects, with the reference to the actual array that contains the matrix data stored as an instance variable. The most natural way to reference the matrix data is through the instance variable, i.e., Mat[i][j], but this generates an extra field access to obtain Mat from this reference, followed by the usual array access instructions. In most, but not in all cases, the JIT compilers were able to optimize this extra reference away. This problem could be easily eliminated even in a static compiler with some form of loop invariant code motion.

The most interesting timings were the ones recorded for the OO version of the OwlPack. The object oriented version was substantially slower in all of our tests, ranging from around 4 times slower than the Fortran-style version for dqrsl and dgesl on jdk 1.2 on Solaris, to more than 250 times slower for dqrdc on the Symantec JIT. The average slowdown was from a factor of 19 for the Solaris JIT to the factor of 140 for the Symantec JIT.

Although the Lite OO version of OwlPack has an enormous performance advantage over the OO version on current Java implementations, the polymorphic object-oriented style results in a code that is much smaller in size – a factor of 3 for OwlPack, expected to grow to factor of 4 when the intended implementation of CFMatrix and CDMatrix is complete. In addition it is easier to maintain because algorithmic changes need only be made once. Ideally, future compiler technologies will be able to automatically transform the OO style OwlPack code into something that approaches the Lite OO version in performance.

The main reasons for the poor performance of the OO version are fairly obvious:

- Every number that is a part of a computation is allocated on the heap as a separate object, requiring additional overhead for instantiation and garbage collection.
- Numbers that are elements of a matrix are scattered over the heap, effectively eliminating the cache performance benefits of spatial locality in standard matrices used in the Fortran and OO Lite versions.
- All operations on numbers are done through method calls to the corresponding objects, incurring additional overhead for the method invocation and the dynamic dispatch required to determine which method is being invoked (since all numbers are abstracted in the LNumber class).
- There is a much greater memory requirement for the OO style, as every number takes up memory associated with object representation in addition to memory for the data itself.
- The presence of objects and method calls prevents some forms of local compiler optimization, such as common subexpression elimination, that are possible in the Fortran and Lite OO versions.

It should be possible to eliminate most of these performance penalties through advanced compiler optimizations. A form of specialization or procedure cloning [5,6,8] could convert the NMatrix class hierarchy into four specialized hierarchies based on the four types of the numbers that could be used as elements of the matrix (LDouble, LFloat, LCDouble, LCFloat). This optimization alone would not bring significant performance gains, but coupled with unboxing or object inlining [3,9,14] that would convert the internal representation from arrays of LDouble for example, to arrays of double and inline all of the operations on numbers, it would generate a class structure very similar to the Matrix hierarchy in OwlPack. The real challenge is to improve on these techniques so they can be applied in Java environment without sacrificing the portability and security of the Java VM.

5. Related and future work

Dongarra et al. [10] are currently working on automatic translation from Fortran to Java and applying it to LAPACK library. While this is an attractive way to quickly transform existing high performance Fortran codes into Java and obtain reasonable efficiency (as demonstrated in this paper), the resulting code would not reflect the best object-oriented programming style. If the goal is to conduct research on how to ameliorate the cost of using the powerful object-oriented features of Java, code produced by this process would not make an ideal benchmark.

Pendragon Software Corporation [16] provides one of the most frequently cited benchmarks for Java. However, the benchmark is not designed to reflect the high performance scientific computation. Furthermore, the product is a microbenchmark, which has led to some controversy about the practice of matching patterns in the code.

This paper demonstrates that there is a need for further research on compiler technology for object-
oriented languages like Java. The JIT compilers of today are not sophisticated enough to eliminate the overhead incurred by polymorphic object-oriented implementations. Static compilers must overcome special problems even before they can apply traditional optimizations because the exception mechanism interferes with control flow analysis and makes it difficult to use the straightforward implementation of some code motion optimizations [2].

Whole [7] and almost whole [3] program optimization techniques have to be further developed for Java environment. Because these strategies sacrifice some of the portability and security of the language, they might not be acceptable to some users.

Dean et al. [7,8], Agesen [1] and others [2,3,5,9] have been working on compiler technologies that would utilize whole program optimization and type analysis to deliver procedural code from object-oriented programs and eliminate the overhead imposed by the object-oriented design. It has been shown that the use of object inlining [2,3,9] and specialization and customization techniques [5,8] in compilation of scientific Java programs can lead to major improvements in running time. However, these techniques have yet to be validated on a large scale, real world applications and in a more restrictive Java bytecode environment. One goal of the benchmark suite we have produced is to make such evaluations possible.

6. Conclusions

We have designed and implemented object-oriented versions of the BLAS and LINPACK linear algebra library in Java. The implementation has been done in two flavors: a fully polymorphic, object-oriented version and a more performance-oriented “Lite” version. Using several leading Java implementations, we compared the performance of the object-oriented versions of the benchmark with a version using Fortran style Java generated by a translator and analyzed the cost of additional overhead that object-oriented design brings to the high-performance computing. We observe that there is a substantial cost associated with a full object-oriented design in scientific programs, and that the current compilers and VMs have a room for substantial improvements in this area.

We should emphasize that the intention of this project was not to design a linear algebra library that scientific programmers should use in their Java programs – if that were the goal we would probably use the translation from Fortran or the Lite style. Instead, our goal was to expose the performance penalties that are associated with object-oriented design and programming style, and to provide compiler researchers with benchmarks that more closely reflect how numerical applications would be implemented using the full power of an object-oriented language. Our long-term goal is to drive our own research (and that of others) on the Java compiler technology needed to eliminate the overhead of the object-oriented approach. If this research is successful, it will reward, rather than penalize, good programming practice.

Ideally, scientific programmers should be able to design their systems in an object-oriented style, without attention to the performance issues. One goal of Java compiler research should be to help achieve that ideal.

References


