Java and Numerical Computing

Ronald F. Boisvert José Moreira Michael Philippsen Roldan Pozo

1 Introduction

We start this article by asking: Does Java have a role to play in the world of numerical computing? We strongly believe so. Java has too much to offer to be ignored. First of all, Java is portable at both the source and object format levels. The source format for Java is the text in a `.java` file. The object format is the bytecode in a `.class` file. Either type of file is expected to behave the same on any computer with the appropriate Java compiler and Java virtual machine. Second, Java code is *safe* to the host computer. Programs (more specifically, applets) can be executed in a sandbox environment that prevents them from doing any operation (such as writing to a file or opening a socket) which they are not authorized to do. The combination of portability and safety opens the way to a new scale of web-based *global computing*, in which an application can run distributed over the Internet [**?**]. Third, Java implements a simple object-oriented model with important features (*e.g.*, single inheritance, garbage collection) that facilitate the learning curve for newcomers. But the most important thing Java has to offer is its pervasiveness, in all aspects. Java runs on virtually every platform Universities all over the world are teaching Java to their students. Many specialized class libraries, from three-dimensional graphics to online transaction processing, are available in Java.

With such universal availability and support, it only makes sense to consider Java for the development of numerical applications. Indeed, a growing community of scientists and engineers developing new applications in Java has been slowly developing. A rallying point for this community has been the Java Grande Forum (`http://www.javagrande.org`) (see sidebar).

There are some difficulties, though, with the wide-scale adoption of Java as a language for numerical computing. Java, in its current state of specification and level of implementation, is probably quite adequate for some of the GUI, postprocessing, and coordination components of a large numerical application. It fails, however, to provide some of the features that hard-core numerical programmers have grown accustomed to, such as complex numbers and true multidimensional arrays. Finally, as any language that caters to programmers of numerical applications, Java has to pass the critical test: its performance on floating-point intensive code must be (at least) on par with the incumbents C and Fortran.

2 The Performance of Java

A common reaction of numerical programmers when first confronted with the thought of using Java for their code is "But Java is so slow!" Indeed, when the first Java virtual machines appeared, they worked by strictly interpreting the bytecode in `.class` files, and delivered very poor performance. Some people reported Java programs running up to 500 times slower than the equivalent C or Fortran codes.

Much has changed in the past few years. Today nearly every JVM for traditional computing devices (*i.e.*, PCs, workstations, and servers) uses just-in-time (JIT) compiler technology. JITs operate as part of the JVM, compiling Java bytecode into native machine code at runtime. Once the machine code is generated, it executes at raw machine speed. Modern JITs perform sophisticated optimizations, such as array bounds

checking elimination, method devirtualization, and stack allocation of objects. Driven by the enormous market for Java, vendors are motivated to continuously improve their JVMs and JITs.

2.1 Examples of Java performance

To help understand Java numerical performance, we took a sampling of common computational kernels found in scientific applications: Fast Fourier Transforms (FFT), successive over-relaxation (SOR) iterations, Monte-Carlo quadrature, sparse matrix multiplication, and dense matrix factorization (LU) for the solution of linear systems. Each kerenel typifies a different computational style with different memory access patterns and floating point manipulations.

Together, these codes make up the SciMark [3] benchmark, one of the more popular Java benchmarks for scientific computing, and whose components have also been incorporated into the Java Grande Benchmark Suite [**?**]. SciMark was originally developed in Java, not translated from Fortran or C, so it represents a realistic view of how one would program in that language. Furthermore, it is easy to use – anyone with a Java-enabled Web browser can run it by a few clicks of their mouse button.

At the benchmark's web site, `http://math.nist.gov/scimark`, we have collected SciMark scores for over 1,000 different Java/hardware/operating-system combinations, from laptops to high end workstations, representing a thorough sample of Java performance across the computational landscape. As of this writing, SciMark scores of over 130 Mflops (the average for the five kernels) have been demonstrated. Figure 1 shows the composite score (in Mflops) of this benchmark on six different architectures, and illustrates the wide range in performance over common platforms. The first observation is that Java performance is closely tied to the implementation technology of the JVM, rather than the underlying hardware performance. From this figure we see that PC platforms typically out-perform high end workstations. To demonstrate the continuous improvement in virtual machine technology, Figure 2 illustrates the performance of progressively new versions of the Sun JVM on the same hardware platform.

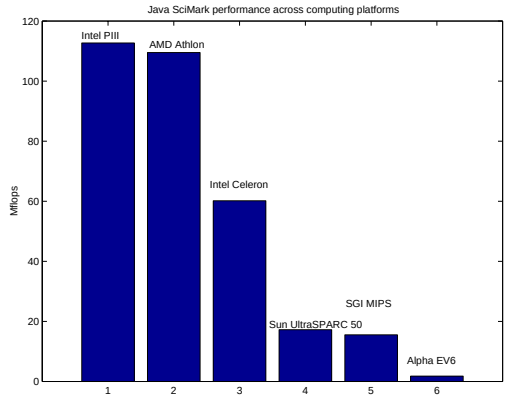

Figure 1: Performance of Java varies greatly across computing platforms. This difference is mainly due to different implementations of the JVM, rather than underlying hardware architecture.

Today, we are seeing Java codes perform competitively with optimized C and Fortran. Figure 3 compares three of the more commonly available Java environments (IBM, Sun, and Microsoft) against two of the most popular optimizing C compilers for the Windows PC: Microsoft and Borland. In both cases full optimization was used (some kernels were running at about 50% of machine peak) , but Java clearly outperforms compiled C. While this may seem surprising, remember that we are not comparing the two languages per se, but rather different *implementations* of compilers and execution environments, which differ from vendor to vendor. We

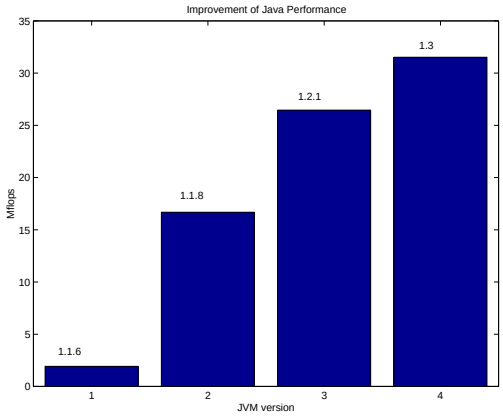

Figure 2: Evolution of Java performance on the same computational platform: a 333 MHz Sun Ultra 10.

should also point out that for other platforms, the results are not as good. For example, a similar comparison on a Sun UltraSPARC [1], shows that Java is only 60% the speed of compiled C. Nevertheless, it is safe to say in this case that Java performance is certainly competitive with C. One common rule of thumb is that for these types of numeric codes, Java currently runs at about 50% of the performance of conventional compiled languages.

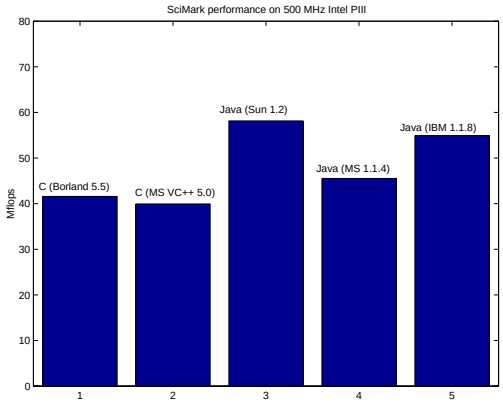

Figure 3: Surprisingly, Java outperforms some of the most common optimizing C compilers on Windows platforms. These results are on a 500 MHz Intel Pentium III, running Windows 98. Results for other platforms are not as good, but nevertheless Java can remain competitive with C and Fortran.

The technology to apply advanced compiler optimizations, including automatic loop transformation and parallelization, has already been demonstrated in a research environment. Table 1 illustrates the performance achieved by the Ninja compiler from IBM T. J. Watson [6] in a set of eight Java numerical benchmarks. For each benchmark, the table shows absolute performance achieved on a 200 MHz POWER3 machine, that performance as a percentage of the equivalent Fortran code in the same machine, and the speedup obtained through automatic parallelization on four processors. For many of the benchmarks the performance of Java

[1] Sun UltraSPARC 60, 360 MHz, running Sun OS 5.7, using cc (Workshop Compilers 5.0), with Sun Solaris VM (JDK 1.1.2.05a, native threads, sunwjit).

Table 1: A summary of Java performance with the Ninja compiler.

benchmark	Mflops	% of Fortran	speedup on 4 processors
MATMUL	340	84%	3.84
MICRODC	210	102%	3.05
LU	154	93%	2.27
CHOLESKY	167	97%	1.44
BSOM	175	81%	2.04
SHALLOW	156	83%	2.40
TOMCATV	75	40%	1.16
FFT	104	54%	2.40

is better than 80% of the performance of equivalent Fortran code, and in most of them a reasonable speedup is achieved by the compiler.

2.2 The role of language specifications

Not all the performance improvements during the last few years are attributed to enhancements in JVM and JIT technology. In some cases, the Java language specification itself was detrimental to performance, and the specification was changed. More specifically, Java 1.2 onwards allows for floating-point computations to be performed with extended exponent range, which is more efficient on *x86* class processors (e.g., the Pentium). The result is that Java class files utilizing floating-point can sometimes produce slightly different results on the *x86* than on the PowerPC or the SPARC, for example. Java programmers who still require strict reproducibility can use the new keyword `strictfp` on methods and classes to enforce it, but performance degradation may result.

In another recent development, Java 1.3 now allows vendors to use different implementation of elementary functions (*i.e.*, `sin`, `cos`, `exp`, and other functions in `java.lang.Math`), as long as they deliver results that differ by at most one bit in the last place from the correctly rounded exact result. Reproducibility of results can be enforced through the use of methods in the new `java.lang.StrictMath` class. This class defines a specific implementation of the elementary functions that guarantees the same result on all platforms.

2.3 Are we there yet?

Problems with Java performance still remain, and they will have to be tackled in the way they have been tackled before: with a combination of language specification changes and new JVM and compiler technologies. It is still possible to write Java code that is order of magnitudes slower than equivalent Fortran code. Because JITs operate at runtime, they have a limited time budget and cannot perform extensive analysis and transformations of the scale done by current static compilers. The representation of elementary numerical values (such as complex numbers) as full-fledged objects exerts a heavy toll on performance. These challenges to Java performance, and some proposed solutions, will be discussed below in more detail. For now, it is important to realize that the current performance of Java can be very competitive in some cases.

While Java is not yet as efficient as optimized Fortran or C, the speed of Java is better than its reputation suggests. Carefully written Java code can perform quite well [2, 4]. (See sidebar on do's and dont's for numerical computing in Java.) Java compiler and JIT technology is still in its infancy, and likely to continue

to improve significantly in the near future. Taken with the other advantages of Java, there is a real possibility for Java to become the best ever environment for numerical applications.

3 Numerical Libraries in Java

An important consideration in selecting a programming environment is the availability of tools to make the job of developing applications easier. Libraries are a particularly important example of programming tools. For one thing, standardized libraries serve as an extension of the programming language. They provide powerful application-specific primitives that tailor the language to a particular area and facilitate code development. Standardized libraries also define a notation for expressing domain-specific operations that are commonly understood by the practitioners in the field. Finally, the components of a library constitute a specific group of operations that can be highly optimized, both by expert programmers and by smart compilers.

The straightforward mechanism to provide Java with libraries for numerical computing is through the Java Native Interface (JNI). With JNI, Java programs can access native code libraries for the platform on which they execute. This approach makes available in Java a large body of tested and optimized libraries for numerical computing. It has been used, in particular, to provide Java with access to MPI and LAPACK [7].

The disadvantages of using native libraries with Java lie in five areas: safety, robustness, reproducibility, portability, and performance. First of all, native code cannot typically be executed in an environment as controlled as Java, and therefore it is not as safe to the host computer. Second, native code does not include all the run-time validity and consistency checks of Java bytecode and, therefore, is less robust. Third, native code, even for a standard library, is likely to have small differences from platform to platform, which may result in different outcomes in each one. Fourth, native code is not portable across machine architectures and operating systems. Finally, there is the performance issue. Invoking a native method from Java incurs on run-time overhead. If the granularity of the operation is large, then the cost of the invocation can be amortized. However, for simple operations the cost of going through JNI can completely dominate the execution time.

Because of all the problems associated with the use of native method from Java, it makes sense to pursue the development of numerical libraries written directly in Java. One of the first such libraries was the Java Numerical Library (JNL), which has been freely distributed by Visual Numerics since 1997. It provides basic facilities in linear algebra, special functions, and elementary statistics. In 1998, another proposal for a standardized linear algebra library called JAMA was developed by MathWorks and NIST. This library includes facilities for solving linear systems and least squares problems, computing standard matrix decompositions (LU, Cholesky, QR, SVD, eigenvalue), and for computing quantities such as norms, determinants, ranks, and inverses. OpsResearch.com has developed a substantial set of Java classes to aid in the development of science and engineering applications, with an emphasis on operations research. A fairly comprehensive listing of available class libraries for numerical computing can be found on the Java Numerics Web page (`http://math.nist.gov/javanumerics`).

Development of numerical libraries in Java present a particular challenge: the preservation of consistent results across multiple versions and multiple implementations of the same version. In many cases, the precise behavior of a library is defined by a reference implementation. Because of the precise floating-point and exception semantics of Java, this reduces the freedom of implementers. Any other implementation is forced to throw the same exceptions and produce the same results as the reference implementation. If the reference implementation uses an algorithm that produces suboptimal results, it becomes impossible to improve on it. That was the case with the original `Math` class in Java. One could not replace it by an implementation that produced more accurate results for the elementary functions, since they would be different than those produced by the reference implementation.

We want to emphasize that optimized libraries are an important performance tool, but they do not constitute a panacea for performance problems. Programming with libraries has its limitations, as discussed in the sidebar. Because of the intrinsic limitations of programming with libraries, one needs language and compiler support. Compilers need to optimize code constructed with language elements, potentially extended with standard library constructs.

4 Remaining Difficulties for Numerical Computing with Java

Despite the very impressive progress in Java performance during the last few years, some challenges still remain. We identify three major remaining difficulties for numerical computing with Java: (i) overrestrictive floating-point semantics, (ii) inefficient support for complex numbers and alternative arithmetic systems, and (iii) no direct support for true multidimensional arrays. We discuss each of these difficulties in detail.

4.1 Java floating-point semantics

Despite some relaxations in Java 1.2 and 1.3, reproducibility of floating-point results is still a feature very central to Java. As a consequence, Java currently forbids common optimizations, such as making use of the associativity property of mathematical operators, which does not hold in a strict sense in floating-point arithmetic: $(a+b)+c$ may produce a different rounded result than $a+(b+c)$. Fortran compilers, in particular, routinely make use of the associative property of real numbers to optimize code. Java also forbids the use of *fused multiply-add* (FMA) operations. This operation computes the quantity $ax+y$ as a single floating-point operation. Operations of this type are found in many compute-intensive applications, particularly in matrix operations. With this instruction, only a single rounding occurs for the two arithmetic operations, yielding a more accurate result in less time than would be required for two separate operations. Java's strict language definition does not permit use of FMAs and thus sacrifices up to 50% of performance on some platforms.

In order to make Java usable by programmers that require the fastest performance possible, it is necessary to further relax the above restrictions in Java. This can be accomplished by introducing a *fast* mode for execution of floating-point operations in Java. This mode would only be used in those classes and methods explicitly marked with a `fastfp` modifier. In this fast mode, FMAs and numerical properties such as associativity could be used by an optimizing compiler. We note that the default mode would continue to lead to more reproducible results (as today) and that the fast mode can only be enabled by the programmer by explicitly identifying classes and methods where it can be used.

4.2 Complex numbers and alternative arithmetic systems

Another indicator of the ability of a programming language to support serious scientific and engineering computing is the ease and efficiency in which computation with complex numbers can be done. Many applications, such as those in electromagnetic and acoustic modeling, for example, are best accomplished in the complex domain. Complex is just one example of an important alternate arithmetic. Others of growing importance are interval arithmetic and multiprecision arithmetic. A good scientific computing language would have the flexibility to incorporate new arithmetics like these in a way that is both efficient and natural to use.

In Java, complex numbers can only be realized in the form of a `Complex` class whose objects contain, for example, two `double` values. A skeleton of such class is shown in Figure 4. Complex-valued arithmetic must then be expressed by means of complicated method calls, as in the following code fragment, which computes $z = ax + y$, where $x = 5 + 2i$ and $y = 2 - 3i$.

```
Complex x = new Complex(5,2);
```

```
Complex y = new Complex(2,-3);
Complex z = a.times(x).plus(y);
```

This has several disadvantages. First, such arithmetic expressions are quite difficult to read, code, and maintain. In addition, although conceptually equivalent to real numbers implemented with primitive types (*e.g.* `double`), `Complex` objects behave differently. For example, the semantics of assignment (`=`) and equals (`==`) operators are different for objects. Finally, complex arithmetic is slower than Java's arithmetic on primitive types, since it takes longer to create and manipulate objects. Objects also incur more storage overhead than primitive types. In addition, temporary objects must be created for almost every method call. Since every arithmetic operation is a method call, this leads to a glut of temporary objects which must be frequently dealt with by the garbage collector. In contrast, Java primitive types are directly allocated on the stack, leading to very efficient manipulation. Another disadvantage is that class-based complex numbers do not seamlessly blend with primitive types and their relationships. For example, an assignment of a `double`-value to a `Complex`-object will not cause an automatic type cast – although such a cast would be expected for a genuine primitive type `complex`.

```
public final class Complex {

  private double re;
  private double im;

  public Complex(double r, double i) {
    re = r;
    im = i;
  }

  public Complex plus(Complex x) {
    return new Complex(this.re+x.re, this.im+x.im);
  }

  public Complex times(Complex x) {
    double r = this.re*x.re - this.im*x.im;
    double i = this.re*x.im + this.im*x.re;
    return new Complex(r,i);
  }
}
```

Figure 4: Skeleton of a `Complex` class for complex numbers.

A general solution to these problems would be afforded by the introduction of two additional features to the language: operator overloading and lightweight objects. Operator overloading is well known. It allows one to define, for example, the meaning of `a + b` when `a` and `b` are arbitrary objects. Operator overloading is available in several other languages, like C++, and has been widely abused, leading to very obtuse code. However, when dealing with alternative arithmetics, the mathematical semantics of the arithmetic operators remain the same, and hence it leads to naturally readable code. In Java, one would need to be able to overload the arithmetic operators, the comparison operators, and the assignment operator. Lightweight objects are defined by final classes with value semantics. Lightweight objects can often be allocated on the stack and passed by copy.

An alternative approach to providing efficient support for complex numbers is to built into a JVM knowledge about the semantics of the `Complex` class, using a technique called *semantic expansion* [5]. Internally, a complex value type is used in place of temporary objects in the code being compiled, and the usual compiler optimizations for complex numbers (as in Fortran compilers) are carried out. In particular, most of the arithmetic methods and constructor calls that prevail in the Java code using this class are replaced by stack and/or register operations.

Yet another approach to obtaining both efficiency and a convenient notation is to extend the Java language with a `complex` primitive type. We note that it is not necessary to extend the JVM accordingly. The *cj* compiler [1], developed at the University of Karlsruhe, has demonstrated compiling an extended version of Java (with primitive `complex` type) into conventional Java bytecode, which can be executed by any JVM. The primitive data type `complex` is mapped to a pair of `double` values. In this way, all object overhead is avoided.

All the alternatives we mention have their pros and cons. Introducing lightweight objects into the language is a fairly fundamental change with far-reaching effects. Semantic expansion does not require any changes to the language or bytecode specification, but requires specialization of the compiler for each new arithmetic system that needs to be supported efficiently. The same specialization also happens with the `complex` primitive type, but only at the level of Java to bytecode translation. Further study and experimentation is necessary to decide on the best solution.

4.3 Multidimensional arrays

In the same way as efficient and convenient complex arithmetic must be made available, numerical computing without efficient and convenient multidimensional arrays is unthinkable. Java offers multidimensional arrays only as arrays of one-dimensional arrays. This causes several problems for optimization. One problem is that several rows of a multidimensional array could be aliases to a shared one-dimensional array. Another problem is that the rows could have different lengths. Moreover, each access to a multidimensional array element requires multiple pointer indirections and multiple bound checks at run-time.

Effective application of optimizations that can bring Java performance on par with Fortran requires true rectangular multidimensional arrays, in which all rows have exactly the same length. Intra-array aliasing (aliasing of rows within an array) never occurs for true multidimensional arrays, and inter-array aliasing (aliasing between rows of different arrays) is easier to analyze and disambiguate than with arrays of one-dimensional arrays.

One approach to introduce true multidimensional arrays in Java is through a standard package. Semantic expansion can then be used by compilers to optimize code that uses the mutidimensional arrays in this package. This approach has been demonstrated to deliver good performance [6]. However, as with the class-based complex numbers, a multidimensional array class requires awkward `set` and `get` accessor methods instead of elegant `[]` notation. Operator overloading would again provide the general solution to this problem. (In this case, `[]` is considered an operator.)

Another approach is to extend the Java language to support true multidimensional arrays, allowing elegant and efficient access with a notation like `a[i,j]`. Such a syntactic extension is not trivial due to the necessary interaction with regular one-dimensional Java arrays. The implementation can be accomplished without changes to the JVM, as the Java compiler would translate the operations on multidimensional arrays to bytecode operations on one-dimensional arrays. Once again, further study is necessary to determine which solution, or combination of solutions, is most appropriate.

5 Conclusions

The technology to achieve very high performance in floating-point computations with Java has been demonstrated. Its incorporation into commercially available JVMs is more an economic and market issue than a technical issue. The combination of Java programming features, pervasiveness, and performance would make Java the language of choice for numerical computing. Furthermore, many of the techniques developed for optimizing numerical performance in Java are applicable other domains, and thus affect a larger group of users. We hope this article will encourage more numerical programmers to pursue developing their applications in Java. This, in turn, will motivate vendors to develop better execution environments, harnessing the true potential of Java for numerical computing.

Sidebar: The Java Grande Forum

The Java Grande Forum is a union of researchers, company representatives, and users who are working to improve and extend the Java programming environment, in order to enable efficient compute- or I/O-intensive applications, so called *grande applications*. The Forum was founded in March 1998, during the ACM/SIGPLAN Workshop on Java for Science and Engineering held at Stanford University. Geoffrey C. Fox of Florida State University and Sia Zadeh of Sun Microsystems played key roles in the initial organization of the Forum. Since then the Forum has organized regular public meetings, which are open to all interested parties, as are its web site (`www.javagrande.org`) and mailing list.

The main goals of the Java Grande Forum are the following:

- Evaluation of the applicability of the Java programming language and the run-time environment for grande applications.
- Bringing together the "Java Grande community" to develop consensus requirements and to act as a focal point for interactions with Sun Microsystems.
- Creation of demonstrations, benchmarks, prototype implementations, application programmer interfaces (APIs), and recommendations for improvements, in order to make Java and its run-time environment useful for grande applications.

The Forum organizes scientific conferences, workshops, minisymposia, and panels in order to present its work to interested parties. The most important annual event is the ACM Java Grande Conference. A large portion of the scientific contributions of the Java Grande community can be found in some issues of *Concurrency – Practice & Experience* (vol. 9, numbers 6 and 11, vol. 10, numbers 11-13, vol. 12, numbers 6-8).

Sidebar: Do's and Don't for Numerical Computing in Java

[In this sidebar we provide a list of simple suggestions on what Java features to embrace and to avoid if one wishes to do efficient numerical computing in Java. We also provide suggestions on what types of simple programmer optimizations are particularly effective. Examples are given below.]

- **do** use modern JVM implementations (the best are on PCs) that use JITs or other compiling technologies; avoid older JVMs that are interpreted.
- **do** alias multidimensional arrays in loops whenever possible, i.e. turn `A[i][j][k]` into `Aij[k]`.

- **do** employ the same optimizations for numeric computing as you would for C and C++, e.g. pay attention to memory hierarchy.
- **do** declare local (scalar) variables in inner-most scope, i.e. `for (int i=0; ,... )`
- **do** use `+=` , rather than `+` semantics for methods to reduce the number of temporaries created.
- **don't** create and destroy lots of little objects in inner-most loops: Java's garbage collector can slow things down. Instead, use your own pool of homogeneous objects (i.e. an array of these and manage the memory yourself.)
- **don't** use the `java.util.Vector` class for numerics: this is designed for a heterogeneous list of objects, not scalars. (You will need to cast each element when accessing it.)

Sidebar: Pitfalls of programming with libraries

The main performance disadvantage of programming of libraries is that of *composition*. We illustrate this concept with an example. Let x, y, and z be vectors of length n. We want to compute $z_i = ax_i + y_i$ for all elements i of the three vectors. If we have a library that provides two methods `foo` and `bar`, which perform scaling and addition of vectors, respectively, we can code the desired operation as a call to method `foo` followed by a call to method `bar`. (See Figure 5(a).) Alternatively, a library could provide a single method `foobar` that implements the complete operation. (See Figure 5(b).) Method `foobar` executes more efficiently than methods `foo` and `bar` in sequence, since it traverses memory fewer times.

```
void foo(double[] x, double a,
         double[] z) {
  int n = x.length;
  for (int i=0; i<n; i++)
    z[i] = a*x[i];
}

void bar(double[] x, double[] y,
         double[] z) {
  int n = x.length;
  for (int i=0; i<n; i++)
    z[i] = x[i] + y[i];
}

foo(x,a,z);
bar(z,y,z);
```

(a) Methods `foo` and `bar`

```
void foobar(double[] x, double a,
            double[] y, double[] z) {
  int n = x.length;
  for (int i=0; i<n; i++)
    z[i] = a*x[i] + y[i];
}

foobar(x,a,y,z);
```

(b) Method `foobar`

Figure 5: Two approaches to implementing $z = ax + y$.

The obvious temptation is to take an "everything but the kitchen sink" approach when designing a library, including every possible combination of operations in the library. This approach does not work for two reasons. First, nobody can, in general, possibly predict all combinations for individual operations. Second, there is a size (*i.e.*, cost) vs. functionality tradeoff with library design, and there is always a limit to the amount of functionality that can be provided.

References

[1] Edwin Günthner and Michael Philippsen. Complex numbers for Java. *Concurrency: Practice and Experience*, 12(6):477–491, May 2000.

[2] Reinhard Klemm. Practical guideline for boosting Java server performance. In *ACM 1999 Java Grande Conference*, pp. 25–34, San Francisco, June 12–14, 1999.

[3] R. Pozo, B. Miller. SciMark: a numerical benchmark for Java and C/C++. `http://math.nist.gov/SciMark`

[4] Mark Roulo. Accelerate your Java apps! *Java World*, September 1998.

[5] Peng Wu, Sam Midkiff, José Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*, pp. 109–118, San Francisco, June 12–14, 1999.

[6] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java Programming for High Performance Numerical Computing. IBM Systems Journal, vol. 39, no. 1, pp. 21-56, 2000.

[7] Glenn Judd, Mark Clement, Quinn Snell, and Vladimir Getov. Design issues for efficient implementation of MPI in Java. In *ACM 1999 Java Grande Conference*, pp. 58–65, San Francisco, June 12–14, 1999.

About the authors

Ronald F. Boisvert (boisvert@nist.gov) is Chief of the Mathematical and Computational Sciences Division at the National Institute of Standards and Technology in Gaithersburg, MD, USA, and co-chair of the Numerics Working Group of the Java Grande Forum.
José Moreira (jmoreira@us.ibm.com) is a Research Staff Member at the IBM Thomas J. Watson Research Center in Yorktown Heights, NY, USA.
Michael Philippsen (phlipp@ira.uka.de) is a member of the Fakultät für Informatik at the University of Karlsruhe in Germany.
Roldan Pozo (pozo@nist.gov) is Leader of the Mathematical Software Group at the US National Institute of Standards and Technology in Gaithersburg, MD, USA, and co-chair of the Numerics Working Group of the Java Grande Forum.