JAVA AND NUMERICAL COMPUTING

Java represents both a challenge and an opportunity to practitioners of numerical computing. This article analyzes the current state of Java in numerical computing and identifies some directions for the realization of its full potential.

For more information on this topic, see the special *Communications of the ACM* issue on "Java for HPC," vol. 44, no. 10, Oct. 2001.

Does Java have a role to play in the numerical computing world? We strongly believe so. Java has too much to offer to be ignored. First, it's 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 file type should behave the same on any computer with the appropriate Java compiler and Java virtual machine (JVM). Second, Java code is *safe* to the host computer. It can execute programs (more specifically, applets) in a sandbox environment that prevents them from doing any operation (such as writing to a file or opening a socket) that 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.[1] Third, Java implements a simple object-oriented model with features that facilitate the learning curve for newcomers (single inheritance and garbage collection, for example). But the most important feature Java offers 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 3D graphics to online transaction processing, are available in Java.

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

However, the wide-scale adoption of Java as a language for numerical computing presents difficulties. 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 features that hardcore numerical programmers have grown accustomed to, such as complex numbers and true multidimensional arrays. Finally, as with any language that caters to numerical application programmers, Java must pass the critical test: Its performance on float-

1521-9615/01/$10.00 © 2001 IEEE

RONALD F. BOISVERT
National Institute of Standards and Technology
JOSÉ MOREIRA
IBM T.J. Watson Research Center
MICHAEL PHILIPPSEN
University of Karlsruhe, Germany
ROLDAN POZO
National Institute of Standards and Technology

ing-point intensive code must be at least on par with the incumbents C and Fortran.

Java's performance

Numerical programmers confronted with the idea of using Java for their code commonly exclaim, "But Java is so slow!" Indeed, when the first JVMs appeared, they performed poorly by simply interpreting the bytecode in `.class` files.

Much has changed in the past few years; today nearly every JVM for traditional computing devices (that is, PCs, workstations, and servers) uses just-in-time compiler technology. JITs operate as part of the JVM, compiling Java bytecode into native machine code at runtime. Once the JVM generates the machine code, it executes it at raw machine speed. Modern JITs perform sophisticated optimizations, such as array bounds check elimination, method devirtualization, and stack allocation of objects. Driven by the enormous market for Java, vendors improve their JVMs and JITs continuously.

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

Together, these codes make up the SciMark benchmark,[2] a popular Java benchmark for scientific computing, whose components the Java Grande Benchmark Suite also incorporates.[3] SciMark was originally developed in Java, not translated from Fortran or C, so it represents a realistic view of how you would program computational kernels in that language. Furthermore, it is easy to use—anyone with a Java-enabled Web browser can run it with a few mouse button clicks.

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 has demonstrated scores of over 130 Mflops (the average for the five kernels). 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 correlates closely to the JVM's implementation technology rather than to the underlying hardware performance. This figure shows that JVMs for PCs typically outperform JVMs available for high-end workstations. To

The Java Grande Forum

The Java Grande Forum (www.javagrande.org) is a union of researchers, company representatives, and practitioners who are working to improve and extend the Java programming environment to enable efficient compute- or I/O-intensive applications—so-called *grande applications*. The forum's main goals include

- evaluating Java's applicability and the runtime 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, and
- creating demonstrations, benchmarks, prototype implementations, application programmer interfaces, and recommendations for improvements to make Java and its runtime environment useful for grande applications.

The Forum organizes regular public meetings, scientific conferences, workshops, minisymposia, and panels. The most important annual event is the ACM Java Grande Conference. You can find many of the Java Grande community's scientific contributions in back issues of *Concurrency: Practice & Experience* (vol. 10, nos. 11–13, 1998; vol. 9, nos. 6 and 11, 1999; vol. 12, nos. 6–8, 2000).

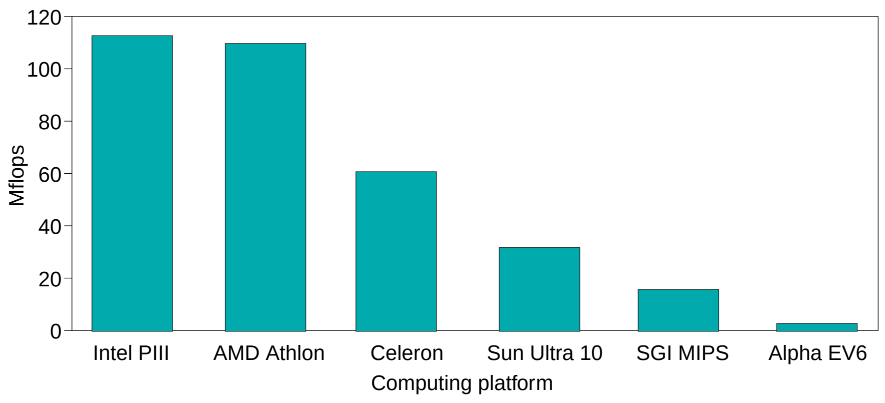

Figure 1. Java's performance for the SciMark benchmark varies greatly across computing platforms. Different JVM implementations—rather than underlying hardware architecture—mainly cause this difference.

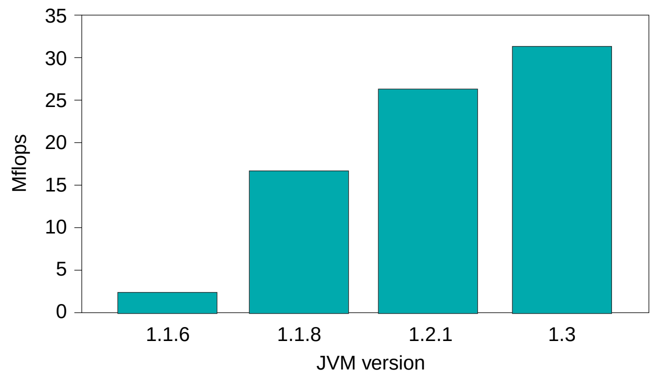

Figure 2. Evolution of Java performance for the SciMark benchmark on a single platform: a 333-MHz Sun Ultra 10.

demonstrate the continuous improvement in virtual machine technology, Figure 2 illustrates the performance of SciMark on progressively new Sun JVM versions on the same hardware platform.

Today we see Java codes performing 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. Both cases used full optimization (some SciMark kernels ran at about 50 percent of machine peak), but Java clearly outperforms C in this case. Figure 4 gives a look at the results for the component SciMark kernels for one C and one Java implementation. Although these results might be 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 should also point out that, for other platforms, the results are not as good. For example, a similar comparison on a Sun UltraSPARC shows that Java attains only 60 percent of the speed of 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 runs at about 50 percent of the performance of conventional compiled languages.

Researchers have already demonstrated the technology to apply advanced compiler optimizations in Java, including automatic loop transformation and parallelization. Table 1 illustrates the performance the Ninja compiler from IBM T.J. Watson Research Center achieved in a set of eight Java numerical benchmarks.[4] For each benchmark, the table shows the absolute performance achieved on a single-processor 200-MHz Power3 machine, that performance as a percentage of the equivalent Fortran code on the same machine, and the speedup obtained through automatic parallelization on four processors. For many benchmarks, Java's performance is better than 80 percent of equivalent Fortran code and for the most part achieves a reasonable speedup.

Language specifications' role

We do not attribute all the performance improvements during the last few years to enhancements in JVM and JIT technology. In some cases, the original Java language specification itself was detrimental to performance. Changes to the specification, which led to significant performance improvements, were made as a result of proposals put forward by the Java Grande Forum's Numerics Working Group. For example, from Java 1.2 forward, the Java specification allows floating-point computations to be performed with extended exponent range (until results are stored to memory). This is much more efficient on the x86 class of processors (such as the Pentium). The side effect is that exact reproducibility of results is no longer guaranteed in Java implementations when floating-point numbers are used—Java class files using floating point can produce slightly different results on the x86, the PowerPC, and the Sparc. The semantics of

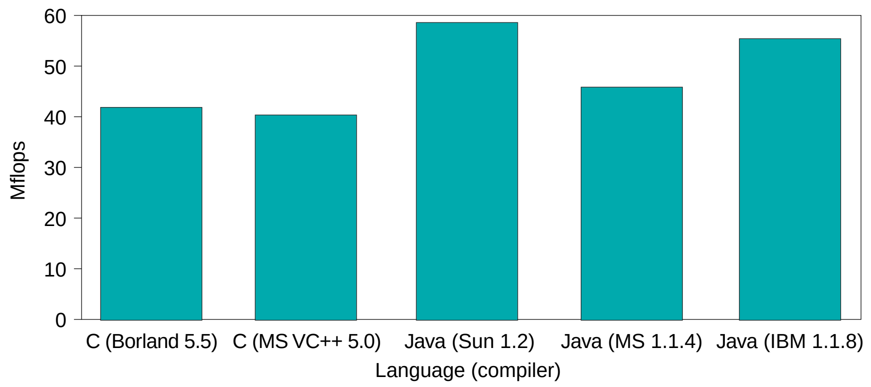

Figure 3. Surprisingly, Java outperforms some of the most common optimizing C compilers on Windows platforms, for the SciMark benchmark. These results are on a 500-MHz Intel Pentium III running Windows 98. Results for other platforms are not as good. Nevertheless, Java remains competitive with C and Fortran.

floating point are still much more strictly defined in Java than in other languages, however. Java programmers who still require strict reproducibility can use a new keyword, `strictfp`, on methods and classes to impose the original Java semantic.

In another recent development, Java 1.3 now lets vendors use different implementations of elementary functions (that is, `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. Methods in the new `java.lang.StrictMath` class can be used to enforce reproducibility of results. This class defines a specific implementation of the elementary functions that guarantees the same result on all platforms.

Are we there yet?

Problems with Java performance still remain, and we must tackle them as we have done before: with combined language specification changes and new JVM and compiler technologies. It is still possible to write Java code that is orders of magnitude 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 that current static compilers do. The representation of elementary numerical values (such as complex numbers) as full-fledged objects exerts a heavy toll on performance. We will discuss these challenges to Java performance and some proposed solutions later in more detail.

Although Java is not yet as efficient as optimized Fortran or C, its speed is better than its reputation suggests. Carefully written Java code can perform quite well (see the "Do's and Don'ts for Numerical Computing in Java" sidebar).[5,6] Still in its infancy, Java compiler and JIT technology is likely to continue to improve significantly in the near future. Taken with Java's other advantages, it could really become the best ever environment for numerical applications.

Numerical libraries in Java

An important consideration in selecting a programming environment is the availability of tools to facilitate developing applications. 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 the practitioners in the field commonly understand. Finally, a library's components constitute a specific group of operations that both expert programmers and smart compilers can highly optimize.

The straightforward mechanism to provide Java with libraries for numerical computing is the Java Native Interface. 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 legacy libraries for numerical computing. It provides, in particular, access to message-passing interface technology and linear algebra packages.[7]

The disadvantages of using native libraries with Java lie in five areas: safety, robustness, reproducibility, portability, and performance. Typically, Java systems cannot execute externally

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

Benchmark	Mflops	Percent of Fortran	Speedup on four 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

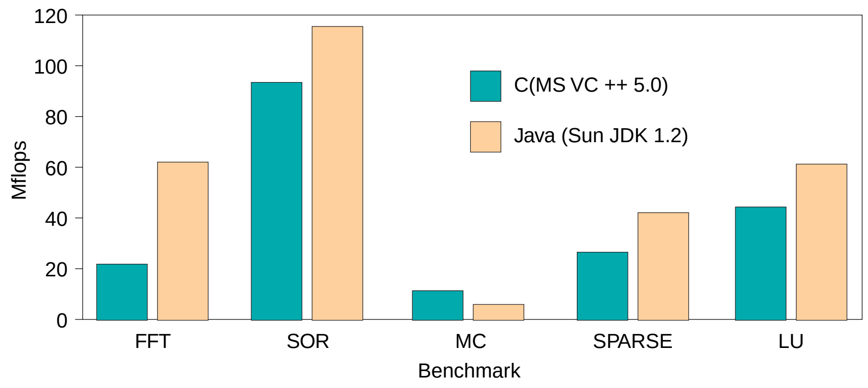

Figure 4. Comparison of SciMark component benchmarks on a 500-MHz Intel Pentium III running Windows 98. The figure compares a C implementation based on Microsoft Visual C++ 5.0 to a Java implementation using Sun's JDK 1.2.

Do's and Don'ts for Numerical Computing in Java

The following are suggestions as to which Java features to embrace and avoid if you wish to do efficient numerical computing in the language.

- **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; that is, turn `A[i][j][k]` into `Aij[k]`.
- **Do** employ the same optimizations for numeric computing as you would for C and C++; for example, pay attention to memory hierarchy.
- **Do** declare local (scalar) variables in innermost scope; that is, `for (int i=0;...)`.
- **Do** use += rather than + semantics for methods to reduce the number of temporaries created.
- **Don't** create and destroy many little objects in innermost loops; Java's garbage collector can slow things down. Instead, use your own pool of homogeneous objects (that is, 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.)

compiled native code in a controlled sandbox, and therefore this native code is not as safe to the host computer. Second, native code does not include all the runtime validity and consistency checks of Java bytecode and is therefore less robust. Third, native code, even for a standard library, is likely to have small differences from platform to platform, which might result in different outcomes in each one. Fourth, native code is not portable across machine architectures and operating systems. Finally, invoking a native method from Java incurs runtime overhead; if the operation's granularity is large, it can amortize the invocation's cost. 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 methods from Java, it makes sense to pursue numerical libraries development directly in Java. Research groups and commercial ventures have begun to develop many such libraries, principally in the area of numerical linear algebra, but more are needed. The lack of such easily available libraries is one reason for Java's slow adoption for numerical computing. You can find a fairly comprehensive listing of available class libraries for numerical computing on our Java numerics Web page (http://math.nist.gov/javanumerics).

Remaining difficulties

Despite the very impressive progress in Java performance during the last few years, some challenges still remain: overrestrictive floating-point semantics, inefficient support for complex numbers and alternative arithmetic systems, and lack of direct support for true multidimensional arrays.

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 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: $(a + b) + c$ might produce a different rounded result from $a + (b + c)$. Fortran compilers, in particular, routinely make use of the associative property of real numbers to optimize code. Java also forbids using fused multiply–add (FMA) operations. This operation computes the quantity $ax + y$ as a single floating-point operation. Many compute-intensive applications, particularly matrix computations, contain such 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 allow the use of FMAs and thus sacrifices up to 50 percent of performance on some platforms.

To make Java usable by programmers that require the fastest performance possible, it is necessary to further relax Java's restrictive floating-point semantics. 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 the programmer can only enable the fast mode by explicitly identifying classes and methods where he or she could use it.

Complex numbers and alternative arithmetic systems

Another indicator of a programming language's ability to support scientific and engineering computing is the ease and efficiency with which it can do computation with complex numbers. Other important alternative arithmetics of growing importance are interval arithmetic and multiprecision arithmetic. A good scientific computing language should have the flexibility to incorporate new arithmetics like these in a way that is both efficient and natural to use.

The most natural way to realize complex numbers in Java is in the form of a `Complex` class whose objects contain, for example, two `double` values. Programmers must then express complex-valued arithmetic 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. Moreover, although conceptually equivalent to real numbers implemented with primitive types (for example, `double`), `Complex` objects behave differently—the semantics of assignment (=) and equals (==) are different for objects. Finally, complex arithmetic is slower than Java's arithmetic on primitive types because it takes longer to create and manipulate objects. Objects also incur more storage overhead than primitive types. In addition, the program must create temporary objects for almost every method call. This leads to a glut of temporary objects that the garbage collector must deal with frequently. In contrast, Java directly allocates primitive types on the stack, leading to very efficient manipulation. Another disadvantage is that class-based complex numbers do not blend seamlessly with primitive types and their relationships. For example, an assignment of a `double` value to a `Complex` object will not cause an automatic typecast—although we would expect such a cast for a genuine primitive type `complex`.

A general solution to these problems would be the introduction of two additional features to the language: *operator overloading* and *lightweight objects*. Operator overloading is well known. It lets you define the meaning of `a + b` when `a` and `b` are arbitrary objects. Operator overloading is available in several other languages, such as C++, but practitioners have widely abused it, leading to very obtuse code. However, when dealing with alternative arithmetics, the mathematical semantics of the arithmetic operators remain the same, so operator overloading leads to naturally readable code. Java needs to overload the arithmetic operators, the comparison operators, and the assignment operator. Lightweight objects are final classes with value semantics. They can often be allocated on the stack and passed by copy.

An alternative approach to providing efficient support for complex numbers is to build into a JVM knowledge about the semantics of the `Complex` class, using a technique called *semantic expansion*.[8] Internally, this approach uses a complex value type in place of temporary objects and carries out the usual compiler optimizations for complex numbers (as in Fortran compilers). In particular, it replaces most complex arithmetic methods and constructor calls that prevail in the Java code, with direct stack 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. The *cj* compiler, developed at the University of Karlsruhe, compiles an extended version of Java (with primitive `complex` type) into conventional Java bytecode, which any regular JVM can execute.[9] It maps the primitive data type `complex` to a pair of `double` values. In this way, it avoids all object overhead.

All these approaches have their advantages and disadvantages. Introducing lightweight objects into the language is a fundamental change with far-reaching effects. Semantic expansion does not require any changes to the language or bytecode specification, but does require specializing the compiler for each new arithmetic that it needs to support 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 the best solution.

Multidimensional arrays

Numerical computing without efficient and convenient multidimensional arrays is unthinkable. Java offers multidimensional arrays only as arrays of one-dimensional arrays. This causes several optimization problems. One 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 runtime.

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

Multidimensional arrays could be provided in Java by an extra package plus semantic expansion, combined or not with a pre-processor. The approach without pre-processor support has delivered good performance but suffers from awkward `set` and `get` accessor methods instead of elegant [] notation.[4] The pre-processor approach requires a nontrivial syntactic extension because of the necessary interaction with regular one-dimensional Java arrays. Once again, further study is necessary to determine which solution, or combination of solutions, is most appropriate.

Many researches have demonstrated the technology to achieve very high performance in floating-point computations with Java. Its incorporation into commercially available JVMs is more an economic and market issue than a technical one. The combination of Java programming features, pervasiveness, and performance could make it the language of choice for numerical computing. Furthermore, all Java programmers can potentially benefit from the techniques developed for optimizing Java's numerical performance, not just those with "grande" applications. 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 Java's true potential for numerical computing.

Acknowledgments

Certain commercial products are mentioned in this article to document computational experiments. Mention of such products does not imply recommendation or endorsement of the National Institute of Standards and Technology.

References

1. I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Francisco, 1998.
2. R. Pozo and B. Miller, *SciMark: A Numerical Benchmark for Java and C/C++*, National Institute of Standards and Technology, Gaithersburg, Md., http://math.nist.gov/SciMark, (current 23 Jan 2000).
3. J.M. Bull et al., "A Benchmark Suite for High Performance Java," *Concurrency: Practice and Experience*, vol. 12, no. 6, May 2000, pp. 375–388.
4. J.E. Moreira et al., "Java Programming for High-Performance Numerical Computing," *IBM Systems J.*, vol. 39, no. 1, 2000, pp. 21–56.
5. R. Klemm, "Practical Guidelines for Boosting Java Server Performance," *ACM 1999 Java Grande Conf.*, ACM Press, New York, 1999, pp. 25–34.
6. M. Roulo, "Accelerate Your Java Apps!" *Java World*, Sept. 1998, www.javaworld.com/javaworld/jw-09-1998/jw-09-speed.html (current 29 Jan. 2001).
7. G. Judd et al., "Design Issues for Efficient Implementation of MPI in Java," *ACM 1999 Java Grande Conf.*, ACM Press, New York, 1999, pp. 58–65.
8. P. Wu et al., "Efficient Support for Complex Numbers in Java," *ACM 1999 Java Grande Conf.*, ACM Press, New York, 1999, pp. 109–118.
9. E. Günthner and M. Philippsen, "Complex Numbers for Java," *Concurrency: Practice and Experience*, vol. 12, no. 6, May 2000, pp. 477–491.

Ronald F. Boisvert leads the Mathematical and Computational Sciences Division of the Information Technology Laboratory at the National Institute of Standards and Technology. His research interests include mathematical software and information services that support computational science. He is editor-in-chief of *ACM Transactions on Mathematical Software*, vice-chair of the ACM Publications Board, chair of the IFIP Working Group 2.5 (Numerical Software), and co-chair of the Java Grande Forum's Numerics Working Group. He received a PhD in computer science from Purdue University. He is a member of the IEEE Computer Society, ACM, and SIAM. Contact him at boisvert@nist.gov.

José Moreira is a research staff member and manager, Blue Gene system software, at the IBM T.J. Watson Research Center. His research interests include languages, compilers, libraries, and runtime systems for the massively parallel Blue Gene system. He received his BS in physics and electrical engineering and an MS in electrical engineering from the University of Sao Paulo, Brazil, and his PhD in electrical engineering from the University of Illinois, Urbana-Champaign. He currently leads the expert group for standardization of multidimensional arrays in Java. Contact him at jmoreira@us.ibm.com.

Michael Philippsen is a senior researcher in the Computer Science Department at the University of Karlsruhe. He also manages the software engineering group of the Forschungszentrum Informatik in Karlsruhe. His primary research interests are parallel systems and software engineering. He received his diploma and PhD in computer science from the University of Karlsruhe. He is a member of the ACM, IEEE, and GI. Contact him at philipp@ira.uka.de.

Roldan Pozo leads the Mathematical Software Group at the National Institute of Standards and Technology, where he investigates issues in high-performance scientific computations, numerical linear algebra, and software environments and tools for parallel computing. Currently, he investigates portable high-performance computing with Java, is the co-chair of the Java Numerics Group, and serves as associate editor of *ACM Transactions on Mathematical Software*. He received his PhD in computer science from the University of Colorado. Contact him at pozo@nist.gov.