8.5 Java

Ronald F. Boisvert and **Roldan Pozo** [51]

8.5.1 Introduction

The Java[52] programming language and environment have become extremely popular since their introduction in the mid 1990s. Indeed, some claim that the number of Java programmers will soon exceed those who use C/C++. Java was not designed for numerical computing, and few scientific applications exist in pure Java as of this writing. Nevertheless, Java has a number of features that make it quite desirable if one's goal is the development of highly reliable software. Among these are the portability provided by the Java Virtual Machine (JVM) and language features that promote safety. In addition, Java provides an object-oriented development framework, a network-aware environment, and access to a rich base of system utilities that make the development of complete applications quite convenient. Such features make it an attractive alternative to Fortran, C, or C++ for the development of scientific applications.

Our goal here is to provide some guidance to non-Java programmers as to whether Java might be appropriate for scientific applications. We will discuss some of Java's features which promote the safety, reliability, and portability, and then briefly address the biggest concern of most scientific users: performance. We conclude with a short discussion of some deficiencies of Java in the context of scientific computing.

8.5.2 Language Features

While Java's first applications were related to the development of embedded systems, the language itself is a general-purpose programming language [Gosling et al., 2000]. Its syntax is very similar to C/C++. It is object-oriented like C++, although it is a much simpler language. For example, in Java everything except a small set of primitive types is an object; conventional constructs like C `structs` have been eliminated.

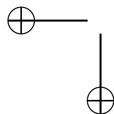
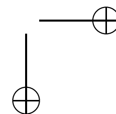
It is widely believed that the simplicity of the Java language leads to an ease in constructing software which, itself, leads to software with fewer errors. Unfortunately, there has yet to be a definitive study which establishes this. We are aware of only one published work on this topic [Phipps, 1999]. While it was a very limited experiment[53], the conclusion was that programming in C++ resulted in three times as many bugs as in a comparable Java program.

The features that Java eliminates from C/C++ that have been accused of leading to programmer error and unintended behavior at runtime include the following.

[51] Mathematical and Computational Sciences Division, National Institute of Standards and Technology (NIST), Mail Stop 8910, Gaithersburg, MD 20899, USA, email: `boisvert@nist.gov`. This section is a contribution of NIST, and is thus not subject to copyright in the USA.

[52] A number of trademarked products are identifi ed in this section. Java and Java HotSpot are trademarks of Sun Microsystems, Inc. Pentium and Itanium are trademarks of Intel. PowerPC is a trademark of IBM. Microsoft Windows is a trademark of Microsoft. Apple is a trademark of Apple Computer, Inc. The identifi cation of commercial products in this paper is done for informational purposes only. Such identifi cation does not represent recommendation or endorsement by NIST.

[53] A single experienced programmer implemented two systems of comparable size, each requiring three months of work, using a formal process to record information about work time, software defects, and bugs.

- *Address/pointer arithmetic*
  While there are pointer variables in Java one cannot perform arithmetic operations on them. One cannot determine the address of objects or primitive variables in Java. As a result, many types of vulnerabilities associated with undisciplined or malicious use of pointers and addresses are avoided.

- *malloc/free*
  In Java there is no explicit deallocation of memory. One creates an object with a `New` statement and the runtime system decides when the associated memory can be deallocated (i.e., when nothing is pointing to it). The Java runtime system uses a garbage collector to manage memory on behalf of the user. This eliminates the very common problem of memory leaks in C/C++ programs.

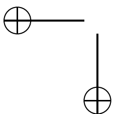
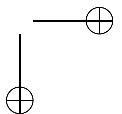
- *goto statements*
  It has long been known that goto statements are not necessary in programming languages, and most people agree that their undisciplined use leads to inscrutable code. Nevertheless, most languages still include goto statements, and their use persists. Java does not have a goto statement.

- *Multiple inheritance*
  The concept of inheritance is fundamental to object-oriented languages. This remains the case in Java. C++ allows classes to be descendants of two or more different parent classes. This multiple inheritance leads to very complex code dependencies that can result in unintended behavior. In Java, a class can have only one parent, greatly simplifying class hierarchies.

Java adds to the mix a number of features, only partially supported in Fortran/C/C++, that promote safety and reliability of software. Among these are the following.

- *Type checking*
  Arguments passed to procedures in Fortran 77 and Kernighan and Ritchie C are not checked to determine whether they match the types expected by the procedure. While this is used as a "feature" by some programmers, argument mismatches are the cause of much unintended behavior in numerical software. Java, like Fortran 90 and ANSII C/C++, is a strongly typed language. The Java runtime system is required to check variables passed as arguments to verify that they are of the correct type. This catches many programmer errors during software development, leading to more reliable code.

- *Run-time array bounds checks*
  Another common programmer error is indexing an array outside its boundaries. The Java specification requires that all index variables be verified for legality before they are used. This protects both the programmer and the system which is running the software.

- *Applet security model*
  Java introduces the notion of an *applet*, a special type of Java application that can be downloaded over the network to execute in the local environment. Of course, downloading executables poses significant security risks. Because of this, Java applets execute in a special restricted environment (a "sandbox"). Many system calls are prohibited, including those providing access to local files. These restrictions, along with the additional safety that comes from features such as array bounds checking, work to insure that applets can be safely executed.

- *Exception handling*
  Java, like C++, supplies a uniform method of handling all exceptional conditions, such as out-of-bounds array references or attempting to access an object through a null pointer. Programmers can write code to "catch" exceptions and handle them as they see fit. Programmers can also extend system-defined exceptions with those of their own. When error handling is a natural part of the language, programmers need not invent ad-hoc error handling facilities. Such ad-hoc facilities may conflict when multiple external packages are utilized in a given application. The use of Java exception handling avoids this.

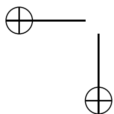
8.5.3 Portability in the Java Environment

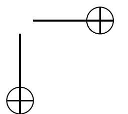
Portability is itself an aspect of reliability. One hopes to get similar behavior from an application regardless of the hardware on which it executes. The sensitivity of numerical computations to the details of floating-point arithmetic make cross-platform reliability a very difficult issue to deal with in scientific applications.

Java's high level of portability is a result of the Java Virtual Machine (JVM) [Lindholm and Yellin, 1999]. The JVM is a simple stack-oriented machine with a well-defined instruction set. Floating-point operations are based on the IEEE 754 standard [Overton, 2001]. In order to run a Java program one needs an emulator for the JVM; this is provided as a part of each Java implementation. Java programs are compiled into a sequence of instructions, called bytecodes, for the JVM. The compiled Java class files are then device-independent object files which can execute in any Java environment.

One of the goals of Java is to support programs that behave exactly the same regardless of what hardware/software environment in which they execute. The JVM is the key contributor to this. Another is the strict semantics of the Java language itself. More than other languages, Java regulates how compilers map language constructs to object code (bytecodes in this case). In many cases it is possible to get exact cross-platform reproducibility of results with Java to an extent impossible with other languages.

Another feature of Java which enhances portability of applications is the wealth of standard class libraries for various types of utility operations, or for particular types of applications. Portability of applications in other languages is impeded because many of these facilities are not universally available, or are provided in a different way on different platforms. In Java, libraries for developing graphical user interfaces (GUIs) are standardized, for example, allowing one to develop GUIs that are portable between Unix, Windows, and Apple platforms. Libraries for two-dimensional and three-dimensional graphics are available. Classes for manipulating network sockets are standardized. Threads are included as





an integral part of the language to support concurrent programming.

The portability provided by the JVM and standardized class libraries admits the possibility of dynamic network distribution of software components. Such a distribution system would greatly simplify the maintenance of scientific applications.

8.5.4 Performance Challenges

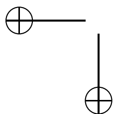
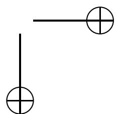
Java programs have a reputation of being highly inefficient. Indeed, many of the features that promote reliability, safety, and portability add overhead to Java's run-time processing. Early experience with Java certainly verified this impression for many. Java systems are steadily improving in their performance, however, and today there are Java systems that rival (and sometimes exceed) the performance of Fortran or C code. This is an issue which is critical in the selection of a scientific computing language. In this section we will touch on some of the main issues regarding the performance of Java.

One of the main characteristics of Java performance has been its high variability [Boisvert et al., 2001, Bull et al., 2001]. This is partly due to the fact that Java is a relatively young programming language and many routes to its optimizations have barely begun to be explored. Nevertheless, it is certain that the Java environment is much more complex that that of traditional programming languages like Fortran or C [Kazi et al., 2000]. For example, in Java one must cope with two levels of compilation. First, a Java compiler converts source code to Java byte codes. Then, perhaps using a different Java system on a computer of different manufacture, the bytecodes are "executed" by a Java Virtual Machine (JVM) simulator. Each step employs separate layers of abstraction, and hence provides further opportunities for inefficiencies to creep in.

The earliest JVMs simply interpreted bytecodes, leading to horribly inefficient execution. Today almost all JVMs employ some kind of *just-in-time* (JIT) compilation strategy, i.e., the bytecodes are compiled into native machine code as a runtime preprocessing step. Various strategies have been proposed for reducing the effect of the preprocessing phase for JIT compilers, the most notable being Sun's HotSpot JVM. HotSpot begins by interpreting bytecodes, while generating an execution profile. When it discovers portions of the code that are heavily executed, it compiles just those parts into machine code. Using this strategy, compilation overhead is expended only where it is likely to pay off. At the other end of the spectrum are Java native compilers, which translate Java directly into machine code. While such compilers have demonstrated fairly high levels of performance, they sacrifice the portability provided by Java bytecodes, one of the main reasons for using Java in the first place.

The issues that make obtaining high performance in Java a challenge for compiler writers include the following.

- *Array bounds checking.*
  Java mandates that all references to array elements outside of declared bounds generate a runtime exception. Array bounds checks can lead to very inefficient code, though the reported effects vary widely [Moreira et al., 2000, Riley et al., 2001]. A variety of optimizations exist to allow compilers to minimize the overhead of bound checking, and these are finding their way into Java environments. Such checks can, for example, be overlapped with other computation on modern processors with mul-

tiple functional units. Checks can be removed completely if the fact that no exceptions will occur can be deduced at compile time. Unfortunately, such theorem-prover approaches are sometimes too time-consuming for a dynamic compilation environment. Even if such deductions are impossible at compile time, code versioning can remove many checks. Here, code for a given loop both with and without checks is generated. At runtime a logic test is performed to determine whether an exception will occur. If no exception will occur, the code without checks is executed.

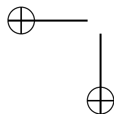
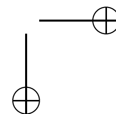
- *Strict exception semantics.*
  Java requires that exceptions are thrown precisely, i.e. at the point corresponding to the Java source code at which they occur. This requirement places severe restrictions on the compile-time optimization of Java code. For example, it prevents rearranging the order of execution of statements across points where exceptions might occur. This makes optimizing of array bounds checks a challenge.

- *Floating-point semantics.*
  Floating-point numbers in Java (`float` and `double` variables) adhere to the IEEE 754 standard for *single* and *double* precision numbers. Each floating-point operation in Java must, in general, produce numbers in this format. This presents a problem on some architectures (such as Intel's X86, the basis of the Pentium) which has registers based on the optional IEEE 754 *single extended* and *double extended* formats. The extended formats provide additional bits in both the fractional part and the exponent. Naive adherence to the requirements of Java on such machines led to the practice of storing the result of each floating-point operation to memory to properly round it down and to cause an exception if the exponent was out of range. As you might expect, such practices lead to very severe performance penalties for compute-intensive codes, as much as 10-fold in some cases.

  With the release of Java 1.2, a slight loosening of this requirement suggested by the [Java Grande Forum Numerics Working Group] of the [Java Grande Forum] was introduced which removed this performance penalty. Technically, in Java's default floating-point mode, anonymous variables, i.e., temporaries introduced by the compiler, may have extended exponents. Since Pentium processors have a control bit which automatically rounds the fractional part of results to single and double precision, only the extended exponent length is at issue. Allowing temporary variables to have larger exponent ranges permits them to stay in registers, removing a significant part of the performance penalty. At the same time, this loosens the implicit requirement that all executions of Java byte code produce the exact same result on all JVMs, though not by much[54]. Java programmers who need universal reproducibility may use the `strictfp` modifier on classes and methods to obtain the Java's original strict floating-point semantics.

- *Elementary functions.*
  A related issue is the definition of elementary functions, such as `sin` and `exp`. The original strict definition of these functions was operational, i.e., it required implementations to reproduce the results fdlibm, the freely available C libm software de-

[54]Operations which overflow on one processor might complete successfully on another.

veloped by Sun. This had several problems. First, while fdlibm was widely acknowledged as good software, it did not produce the correctly rounded version of the exact result. Thus, it required results be forever inaccurate, even in the face of improved algorithms. Second, it was largely ignored. This definition was subsequently replaced by one suggested by the Java Numerics Working Group, which allowed any result within one unit in the last place (ulp) of the correctly rounded result (0.5 ulp is optimal). Although this again relaxed Java's requirement for exact reproducibility, it admitted highly-efficient machine-specific implementations of the elementary functions. A new standard Java class library *StrictMath*, which does produce the same results on all platforms, is now available for uses who require exact reproducibility.

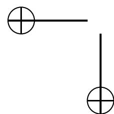
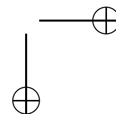
- *Floating multiply-add instructions.*
  Another roadblock to high levels of floating-point performance in Java is the implicit prohibition of the use of floating multiply-add (FMA) instructions. FMA's compute $\alpha x + y$ as a single instruction (i.e., with a single round). Java's semantics require two rounds, which is less accurate, but reproducible on all architectures. This can lead to as much as a 50 % performance penalty in the inner loops of linear algebra computations on processors, such as the PowerPC and the Itanium, on which FMAs are available. This issue has yet to be resolved for Java.

8.5.5 Performance Results

In spite of the challenges facing Java system designers, the performance of Java systems for scientific applications, as tracked in a variety of benchmarks and studies, has been steadily increasing [Bull et al., 2001, Scimark, Riley et al., 2001]. One benchmark useful for tracking Java peak performance is [Scimark]. Scimark includes five kernel benchmarks: fast Fourier transform (FFT), dense LU matrix factorization, Jacobi SOR matrix iteration, sparse matrix multiply, and Monte-Carlo quadrature. The benchmark comes in two versions, one whose problems fit into most caches, and one whose problems do not. A version of each component is also provided in C. The in-cache benchmark is distributed from the Scimark Web page as a Java applet; clicking on the URL will download the benchmark, run the applet, and allow one to upload the results back to a NIST database. Results are reported in megaflops; both individual components and an average of the five components are reported. The results are continuously updated on Scimark web page; as of this writing some 1500 results have been submitted. The range of reported results is impressive, from 0.03 Mflops (on a Winbook running Netscape in 1999) to 404 Mflops (on a 2.8 GHz Pentium IV running Linux 2.5.52 and an IBM 1.4.0 JVM in 2003).

One thing immediately evident from Scimark results is the steady improvement of performance over time due to better Java compilation systems and JVMs. Figure 8.1 illustrates this by showing increasing performance of the Scimark benchmark over time on a 333 MHz Sun Ultra 10 system using successive releases of a particular Java system.

Much variability of Java performance across platforms has also been reported, however. JVMs for PC-based systems have been seen to outperform their counterparts running on traditional Unix workstations and servers [Boisvert et al., 2001]. This may seem counterintuitive; however, it may simply reflect the fact that developers of Java for PC-based systems have been more aggressive in producing efficient Java implementations than their

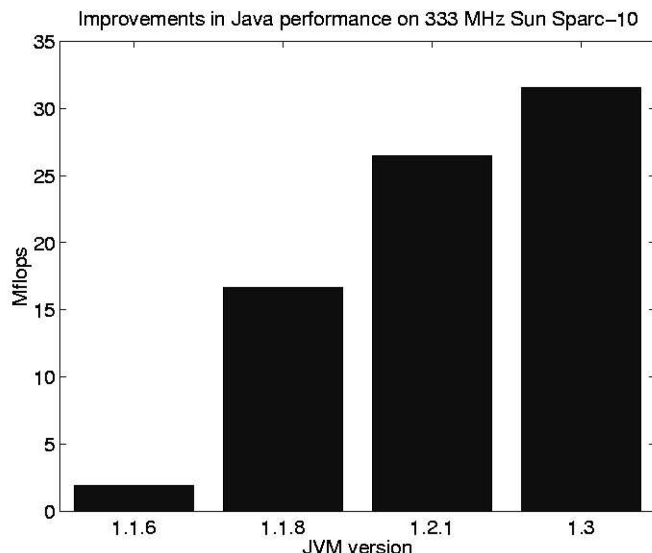


Figure 8.1. *Java Scimark performance on a 333 MHz Sun Ultra 10 system using successive releases of the JVM.*

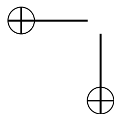
counterparts on traditional high-performance workstations. Since a variety of JVMs are available for most platforms, we also see variability in Java performance on a single processor.

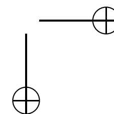
Recently we compared the performance of Scimark on two separate C/C++ environments and three separate Java environments on a 500 MHz Pentium III system running Microsoft Windows. The Java applications ran at 46, 55, and 58 Mflops, while the C applications ran at 40 and 42 Mflops. (High levels of optimization were enabled for the C compilers.) Remarkably, the applications written in Java outperformed those written in C on this platform. Figure 8.2 shows the performance of the Scimark component benchmarks on this platform for the best Java and C environment from these tests.

While the relative performance of C and Java shown here is far from universal, it does illustrate that much more attention has been directed in recent years to optimizing Java performance on PC platforms than to optimizing C performance. In many real scientific applications Java performance is typically 50 % of that of optimized Fortran or C [Riley et al., 2001].

8.5.6 Other Difficulties Encountered in Scientific Programming in Java

While the performance of Java is steadily improving, there are a number of additional difficulties that one encounters when attempting to use Java for large-scale scientific and engineering computations. Some of these are outlined below.





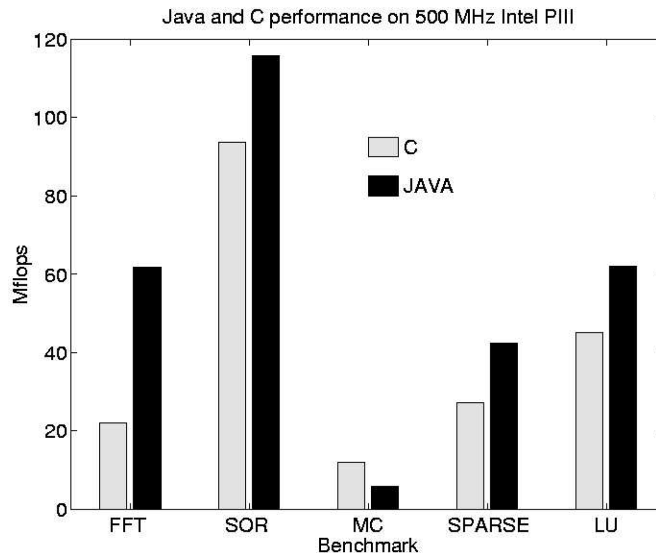


Figure 8.2. *Performance of Scimark component benchmarks in C and Java on a 500 MHz Pentium III system.*

- *Lack of complex data type.*
  There is no complex data type in Java as there is in Fortran. Since Java is an object-oriented language this should, in principle, present no problem. One simply defines a complex class and develops a set of methods which operate on objects in that class to implement the basic arithmetic operations. For Java, a straightforward approach like this leads to performance disaster [Wu et al., 1999]. Complex numbers implemented in this way are subject to substantial overhead, and paying this overhead for each arithmetic operation leads to extremely inefficient code. Also, each operation creates a temporary object, and the accumulation of huge numbers of small objects can lead to repeated calls to the garbage collector. Finally, code based upon such a class would be very unpleasant to write and to read. For example, a simple operation such as

```
a = b+c*d;
```

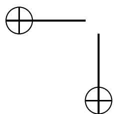
  would be expressed as either

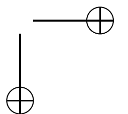
```
a.assign(Complex.sum(b,Complex.product(c,d))
```

  or

```
a.assign(b.plus(c.times(d)))
```

  Efficient manipulation of complex numbers requires that the programmer manage real and complex parts separately, a burden that is both inconvenient and error-prone.





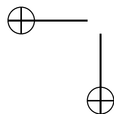
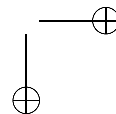
- *Difficulty in implementing alternate arithmetics.*
  The same difficulties as noted above for complex arithmetic exist in the implementation of any alternate arithmetics in Java, such as interval or multiprecision arithmetic. The problem stems from the fact that no lightweight aggregation facilities, like C `structs`, are available. In addition, no facilities for operator overloading are available to make arithmetic on alternate datatypes look natural. The absence of lightweight objects and operator overloading contribute significantly to the simplicity of the language, a feature that is central to Java's design philosophy, and hence it is unlikely that they will be added.

  A much more feasible approach is the development of language preprocessors admitting specialized datatypes of interest to the scientific computing community. If such preprocessors would emit pure Java, then all the advantages of Java would be preserved, while providing an environment much more conducive to the development of scientific applications. The feasibility of such preprocessors for complex arithmetic has been demonstrated [Günthner and Philippsen, 2000].

- *Lack of true multidimensional arrays.*
  The performance of scientific applications is often dependent upon linear algebra operations, such as the matrix multiplication, or the solution to linear systems, which lie at the core of the computation. As a result, highly optimized linear algebra libraries have been developed to provide the best performance possible for these fundamental operations. Effective management of the flow of data through the memory hierarchy is key to obtaining high performance on modern architectures. Doing this requires a careful coordination of data and how it is laid out in memory and the order of processing in a given algorithm. Fortran provides a complete specification of how multidimensional arrays are linearized for storage, thus providing the algorithm designer with a good indication of how such arrays may be laid out in memory.

  Java does not support multidimensional arrays. Instead, Java provides arrays of one-dimensional arrays, as in C. Rows of these simulated multidimensional arrays need not be contiguous in memory. Indeed, there is no guarantee in Java that adjacent elements of a one-dimensional array are contiguous! These facts make developing highly efficient and portable Java linear algebra kernels somewhat problematic. All kernels in the Scimark benchmark use native Java arrays, with fairly satisfactory performance, however. Nevertheless, there has been considerable interest in developing Java language extensions or specialized class libraries for efficient manipulation of multidimensional arrays [Moreira et al., 2001].

- *32-bit addressing*
  Indexes to Java arrays are limited to 32-bit integers, implying that one-dimensional arrays longer than 2,147,483,647 are not possible. While this is quite large, the size of routine numerical computations is also growing quite fast. While new 64-bit computer architectures support much larger address spaces, Java programs are unable to take advantage of them.

- *Lack of a legacy of mathematical software packages.*
  Since Java remains a relatively new language, it suffers from the lack of an existing base of available mathematical software components to aid in the development of applications. In addition, since the numerical analysis community remains wedded to Fortran, and to some extent C and C++, new packages that demonstrate state-of-the-art methods are not finding their way to Java. While it is possible to invoke components from other languages from the Java environment, this removes the advantages of portability provided by pure Java. Some Java class libraries for fundamental mathematical operations have begun to emerge, but progress has been slow. A list of known Java class libraries for numerical computing is maintained on the Java Numerics Web page at NIST [Java Grande Forum Numerics Working Group].

8.5.7 Summary

Java has a number of features which naturally lead to more reliable software. These, along with Java's universal availability, make it a strong candidate for a general-purpose scientific and engineering applications. However, these advantages come at a cost. While Java's raw performance can exceed that of optimized C in some cases, it still lags behind optimized C and Fortran on many platforms. Also, since it was not designed with scientific applications in mind, and there is not yet a strong legacy of scientific codes in Java, programmers face a number of inconveniences in practical code development. Nevertheless, Java may be a reasonable candidate, especially when portability and reproducibility are extremely important.