1-2002

Java at Middle Age: Enabling Java for Computational Science

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Editor: Paul F. Dubois, paul@pfdubois.com

JAVA AT MIDDLE AGE: ENABLING JAVA FOR COMPUTATIONAL SCIENCE

By George K. Thiruvathukal

NOT LONG AFTER THE MOSAIC WEB BROWSER APPEARED AND NETSCAPE COMMUNICATIONS FORMED, SUN MICROSYSTEMS LAUNCHED JAVA IN A WAY THAT WILL BE FOREVER KNOWN AS A MARKETING SUCCESS STORY. ALL ACROSS THE US, there were many Java Day events. I still have my t-shirts as proof of having been there.

Java's early success owed much to the Netscape browser, which was designed to run Java applets directly (although I remain convinced that Java's unique approach would have earned it a place in language history independent of Netscape). You could store Java code on your server and download it to a browser; a Java Virtual Machine (the so-called JVM) could then load and execute it, in principle, anywhere. It is this ubiquity that has everyone still talking in terms of Java.

Java's early history explains its lure to this day. Java is alive and well, running on just about every computing platform, from handhelds to high-end servers such as multiprocessors. Implementations of Java for Windows, Macintosh (including OS X), and Linux have all reached sufficient maturity and are in widespread use. Performance and resource usage remain a problem in most Java implementations, but the language is improving all the time.

In this article, I examine the lure of Java for computational science, discuss the Java Grande effort to work with Sun, and identify areas for improvement. I won't rehash the stories that abound of bad Java performance but will instead focus on language and implementation issues that must be solved for Java to be taken seriously for computationally focused codes. As a motivational tool, I include a number of reflections on the C# language from Microsoft, which many claim is nothing more than Java with a new syntax. I excise a number of sections from the specification documentation to demonstrate that specific Java Grande Forum (JGF) recommendations are implemented in the current version of Microsoft's C#. What is particularly interesting (but not necessarily verifiable) is that language appearing in the Microsoft C# specification proper often appears to paraphrase similar points raised in the JGF documents available at www.javagrande.org.

Java takes the world by storm

In a nutshell, Java enjoys a great deal of popularity for several reasons:

- *Documentation*. It is one of the few languages (besides Python) that supports documenting comments, which makes documenting your code a snap.
- *Write once, run everywhere*. This is Sun's famous marketing jargon, and it's mostly true but not always with good results. This slogan emphasizes reproducible results, which is often at odds with achieving good performance.
- *Code mobility*. Long considered an important tenet of distributed systems, Java has a well-developed notion of code packaging (code shipping). You can migrate and execute a Java program's compiled representation anywhere Java runs. This is not possible under normal circumstances in other compiled languages—such as C or Fortran—especially between different architectures and operating systems.
- *Compiled*. Java is a compiled language. True, the JVM is a hybrid of interpreter and compiler, but with most current JVMs, performance-critical code is adaptively compiled and executed with native CPU instructions.
- *Memory management*. Unlike C, C++, and Fortran, where memory management issues are left entirely to the programmer, Java supports automatic garbage collection. It manages memory entirely at runtime, automatically freeing memory that is no longer needed. This determination of how and when memory is no longer needed is a long-standing research area in computer science, and it is still not perfect. Nevertheless, techniques exist for pooling memory to minimize unexpected interactions with the memory manager.
- *Graphical interfaces*. Java is one of the only environments where you can

develop an application using its Abstract Windowing Toolkit (the bows and arrows) or Swing (the heavy artillery) and be virtually guaranteed that the code will run on any significant platform. (I can attest to this being true for Windows, Macintosh OS X, Linux, and Solaris.)

- *Enterprise networking*. Java is completely enabled for the Internet. It can do everything from sockets programming to high-level remote procedure calls (Remote Method Invocation, Common Object Request Broker Architecture, and XML/RPC), and it just plain works.

Java is best described as a story in good (albeit conservative at times) language design and excellent integration through its class libraries. It has practically every capability needed in programming and, unlike C or Fortran, its source and executable forms of code can be distributed and executed with little or no modification to the source code, resulting in a nearly seamless environment for executing code.

About Java Grande

Against this backdrop, a number of us in the computational science community saw (and continue to see) Java's potential for scientific and technical computing. How many times have you written an application in C only to move it somewhere else just to make that one change to get it working due to a missing library function? Having worked on several Fortran and C applications myself, the potential was obvious: I could develop code on my personal computer or notebook and run it on a different machine simply by moving the binary code itself. To go one step further, I could develop, compile, and package the code (into a Java archive, a so-called .jar file) and then ship it to the target environment for execution *without* change.

The Java Grande concept began in 1998 as a result of the First International ACM Workshop on Java for Scientific Computing, which Geoffrey C. Fox founded. I served as the first (and only) secretary general for the JGF and was responsible for editing the first version of the Java Grande Report, which you could construe as a report card for Java and its support for high-end computing applications.

Members of the JGF, when appearing in public, are often asked, "How did you come up with the name grande?" The Spanish word *grande* is used in much the same way as the English word *grand*, to denote something that is either great or large. Our intention was to capture the essence of large-scale Java applications that take Java above and beyond its intended use (applets running on the desktop) to support the kinds of applications that many of us in the scientific community are developing in C and Fortran. The intention was not to replace any of these legitimate activities but to provide a viable alternative or supplement using a state-of-the-art programming language that provides a glimpse into the future.

A grande application is any application that requires extensive processing power (and thus wall-clock time) to execute. The Human Genome Project, astrophysics, fluid dynamics, aeronautical engineering, and other large-scale simulations are all application areas within the purview of Java Grande. Grande applications, in addition to having extensive processing power, also need extensive I/O support as well.

Java is not completely new

Although Java has several innovative aspects, emphasizing that Java and its approach are not completely new is important. In particular, let's focus on the byte-code concept, which is the machine code that Java compilers produce and that the JVM executes. Java is not the first language to attempt to use platform-independent byte codes. There are three documented success stories with this approach, including the Pascal environment (UC San Diego), Icon (University of Arizona), and Smalltalk (Xerox and Digitalk). Even one of my favorite programming languages, Python, used a byte-code approach for program execution.

There is a widespread misconception about Java that the byte-code concept implies inefficiency. This is because code tends to inhabit two worlds: "fast" and "slow," for lack of better terminology. To a large extent, this implication is only a reality because of today's JVM implementations and economic considerations. The JVM design itself could help create innovative coprocessors that would only run Java's byte-codes in lieu of a virtual machine. Obviously, economics don't favor bundling a coprocessor with every system, and it is unlikely that everyone would go out and purchase a computer system with a dedicated Java processor. I should point out for the sake of completeness

For Further Reading

Java Grande Report JGF-TR-1: Making Java Work for High-End Technical Computing, www.javagrande.org

ACM Java Grande Conference Series, www.acm.org/pubs/contents/proceedings/series/java

Java Numerics Effort, http://math.nist.gov/javanumerics

"Java and Numerical Computing," R.F. Boisvert et al., *Computing in Science & Eng.*, vol. 3, no. 2, 2001, pp. 18–24.

HotSpot link, http://java.sun.com/products/hotspot

C# link, http://msdn.microsoft.com/vstudio/nextgen/technology/Csharp_Language_Specification.doc

Café Dubois

Video tutorials

You will have noticed that some people can't read your fine manual. There are people who, if shown something, can do it much more easily than if they read about how to do it, so you feel that there is no substitute for visiting people and showing them. But with travel now even more miserable than it used to be, and the seats on airplanes now 20 percent smaller than the average person, this doesn't sound like a good way to spend your time.

Video sounds tempting, but it's harder than it seems. You have to have a good camera, decent lighting, and that guy named Joey from the AV department who knows how to run it. It doesn't show your computer screen very well unless you have even more special equipment or Joey gets smarter.

Solution: realize that the thing that doesn't have to appear on the screen is, well, ahem, you. You can make a tutorial by capturing your screen and recording your voice using a microphone. After you sync the two together, you can deliver by download, CD, or streaming server.

ScreenWatch (www.optx.com) is a neat product. To use it, you need a Windows 2000 machine to record on. If your software only runs on Unix, you can still record it by running X on the Windows machine. ScreenWatch is a commercial product, but the cost is modest—about $3,000 for a basic setup.

Here's how you can try it. Get an evaluation license for ScreenWatch, and then download RealProducer Basic from Real Networks (www.realnetworks.com). Your consumers will need RealPlayer (to which they install a component that can handle ScreenWatch files by downloading the component from the Real Web site). You'll need a basic microphone or better.

The RealProducer records your voice, and ScreenWatch records the screen while you run your software. ScreenWatch works by swapping in a virtual video driver that records your screen. It records a full frame and then only records changes for a period. By repeating this cycle, the file sizes it produces are kept modest. Therefore, the best situation is one in which the screen is relatively slowly varying. ScreenWatch does have a mode for high-rate capture, however. You can record voice while you record video or later while you are watching playback. I do both, because I find that speaking while demonstrating the software puts the right amount of space into it for a later voiceover.

Once you have the two files recorded, you create a file called a "smil" file that the RealPlayer recognizes. This is a text file written in an HTML-like syntax. The file tells the player what clips to play and at what point in their files to begin and end. By finding a point in each file to start from that represents the same point, you can sync voice to video. (ScreenWatch 4.0, which should be out by the time you read this, automates the process.) You can make a presentation out of a set of pieces from one or more recording sessions.

To deliver the tutorial, you now need a way to deliver the three files (video, audio, and smil) to your user. You can do this as a download, on a CD, or you can get a free RealServer Basic to stream video off your Web site if you only need to serve a couple of dozen simultaneous users. You can get pretty fancy; see the ScreenWatch demos to get an idea. You can buy an extra component to produce Windows Media versions, too.

The tow truck

Like most Americans, I have suffered a lot of grief and a lot of anger in the last month. (I'm writing this in mid-October.) One of the things I've thought about is a conversation I had with Dr. P when I was in my last year of graduate school. Dr. P knew something about freedom. He had escaped from an Iron Curtain country and earned a second PhD once he reached America because he couldn't prove he had received the first one.

Dr. P had an interesting grading system: each exam had five questions worth 20 points each. There was no partial credit ... ever. Dr. P explained that your answer was either complete and completely right, or it was wrong. The result was then translated into the standard 90–100: A, 80–90: B, and so on. Some of us tried to point out that you couldn't possibly get a 90, but Dr. P wasn't interested.

Having passed Dr. P's class by some miracle, I was teaching my first calculus class and realizing that the poor devils to whom I was giving a C were, in fact, totally incompetent. I worried about my moral responsibility: If I gave them a passing grade, and they became engineers and killed somebody, that would be my fault, right? Maybe Dr. P had a point. I asked Dr. P about my concern. He said that in his homeland there was a totally different attitude about bad engineering. If you built something in America and it fell down, well, they might sue your company. In his homeland, you would go to jail. He said it made you study harder.

So, underneath my horror on September 11th, as I watched those pictures of the plane hitting the World Trade Center, was some wonder, too. The guys who engineered that building surely did not get a C in my calculus class. It was simply amazing to me that the building hadn't fallen over immediately. Some good engineering saved a lot of lives.

Shortly thereafter, I needed to have my car towed. The truck that showed up was as beautiful as a cheetah. Like a cheetah, its form fitted its function so perfectly that a single man, without any substantial physical effort, was able to scoop up my car onto the truck's back and take it to the repair shop. The fee for this miracle was a mere $65. Overcome by recent events, tears came to my eyes, honestly. I said to the driver, "This truck makes me proud to be an American." He knew just what I meant. He smiled broadly and replied, "Yes it does, doesn't it. And it is so simple, just so simple."

Thanks, professors.

that Sun actually tried. It designed a Java chip and built some Java workstations, which (not surprisingly) were powered by a JavaOS and Java. It didn't go far.

The idea of building Java chips was not entirely a misguided one. There were real performance issues with respect to Java, and many of them could be resolved without building dedicated hardware. The Java Grande effort, aimed at the world of high-performance scientific and technical computing, was in reality an effort to make Java work better for computing in general. In short, Java's performance problems not only hurt scientific and technical computing codes, they hurt all codes. You don't need to look far to be convinced. Corel's early attempt to port its office suite to Java was a disaster due entirely to performance. (In fact, no one has since made a credible attempt to make general office-style applications work in Java.) Many of the performance problems of Swing (Java's GUI framework, which relies on bitmapped graphics) would improve were array performance issues addressed. Based on the needs of the many, it would appear Java Grande and its recommendations are for everyone.

Java + Java Grande = A tough sell

Java's state in 1998 was one of anything but performance. Implementations in many cases reported performance roughly equal to 1/100th the speed of C, especially for any code using arrays or floating point. The JVM operated to a large extent in interpreted mode. Then a number of just-in-time compilers emerged. A JIT can turn Java byte-code into native code on the fly, but it often imposes an additional runtime cost because it runs just about every time your program does. (JIT compilers have no memory and typically forget their results.) For long-running scientific codes, however, the JIT approach can work quite effectively because actual cost as a percentage of overall runtime is arguably negligible. For completeness, JIT compilers work incrementally. They can kick in when a module (class) is loaded for the first time and according to any criteria where compilation into native code would help performance.

The lack of compiled Java was definitely an initial turn-off for using Java in compute-bound applications. The good news is that compilers have improved. The HotSpot effort from Sun and the Ninja project at IBM have shown that it is possible to significantly improve performance by developing state-of-the-art compilation techniques (or in the case of HotSpot, putting many old tricks to use again).

However, compiler techniques alone do not lead to better performance. Language and virtual machine changes (often difficult to distinguish but an important distinction from Sun's point of view) are needed to support higher performance. In addition to performance considerations, many language changes could spruce up Java for those who grew up on other science-friendly languages such as Fortran, C, and C++. I will now address these changes.

Complex arithmetic

Complex numbers are used just about everywhere in mathematical problems in science and engineering applications. (I am not one to insult my audience. I believe everyone reading this magazine knows that complex numbers are of the form $a + bi$, where a and b are real and $i = \sqrt{-1}$.) One specific JGF recommendation was to introduce a complex data type that stands on equal footing with float, double, int, and other scalar types.

Adding a complex type to Java presents a couple of design opportunities. Either add support for it in the JVM instruction set or create a user-defined Complex class. Clearly, having a native type is the most desirable strategy and is the one science-friendly Fortran employs, but there is a significant problem to overcome to make this fully viable in Java. The JVM instruction set uses 8-bit operation codes (opcodes). If we were to add any new data type, as innocuous as it seems, this could easily exhaust the remaining opcodes (of which there are only a few—see the "Opcode Summary for Java" sidebar). Using extended instructions is possible, but the amount of work is the same to add the type thereafter.

Another possibility is to use Java's support for defining your own types. For those who are completely new to object-oriented programming, Java lets you define your own types using a language mechanism called a class. The following code shows a user-defined floating-point complex type, where the real and imaginary parts are single-precision floating-point numbers:

```
class FloatComplex {
  public float rpart, ipart;
  public Complex(float rpart,
      float ipart) {
    this.ipart = ipart;
    this.rpart = rpart;
  }

  // then the complex
      operations
  public add(FloatComplex
      another) {
    this.rpart = this.rpart +
      another.rpart;
    this.ipart = this.ipart +
      another.ipart;
  }
  public mpy(FloatComplex
      another) {
    float result_ipart,
      result_rpart;
    result_rpart = this.rpart
      * another.rpart –
      this.ipart * another.
      ipart;
      result_ipart = this.ipart
      * another.rpart –
      this.rpart * another.
      ipart;
  }
  // details omitted but
    easy to code
  public sub(FloatComplex
    another)
  public conjugate(FloatComplex
    another)
  public divide(FloatComplex
    another)
}
```

Classes are useful for creating your own types, but the storage overhead introduced to create class instances is much greater than the storage required for complex numbers in other languages. In Fortran, a complex is stored as a pair of floating-point (or double) values. When an operation is performed on complex numbers, the operation is always performed on the real and imaginary parts separately, entirely relying on the algebraic identities. Suppose you have complex numbers x and y, where $x = a + bi$, and $y = c + di$, and you want to multiply them. A Fortran compiler is smart enough to compute the real and imaginary parts merely by applying the algebra $(a + bi) \times (c + di) = ac + (bc + ad) \times i - bd = (ac - bd) + (bc + ad) \times i$.

Of course, a good compiler writer can easily write a Java language front end or a preprocessor (something no longer present in Java) that features a complex type (or macros for generating one) and then compile it into two floating-point storage locations. Generating code for the multiplication code is then a matter of generating it for the real and imaginary parts of the result separately. Roughly

```
complex x, y, result
result = x * y
```

is compiled as

```
result_real = real(x) * real(y)
  – imag(x) * imag(y)
result_imag = imag(x) * real(y)
  + real(x) * imag(y)
result = result_real +
  result_imag * i
```

where `real(var)` refers to the real floating-point part of the complex variable; `imag(var)`, the imaging floating point part.

Floating-point numbers

Floating-point numbers are a source of contention between Java Grande and the Java effort. On one hand, Java promises write once, run everywhere. On the other, Java Grande wants and needs the ability to exploit the floating-point capabilities present on conventional, commodity hardware, such as the Intel Pentium and Itanium chips. Java's write once, run everywhere proposition is not just good marketing, it is in many respects a business commitment that Sun has made with its partners that it must honor. Allowing floating-point results to differ on different architectures is akin to breaking a sacred covenant.

The Java Grande argument centers on a proposition that enforcing bitwise reproducibility and floating-point semantics is a daunting task. We can write floating-point computations that will produce different results on different processors, even with the current approach Java uses.

Let's examine this a bit more closely. Consider the Intel 80x86 processors. They operate naturally on 80-bit double extended floating-point values (so-called extended precision, which goes beyond double-precision arithmetic). Java implementations on the Intel processor must reduce any floating-point operation's precision to that of double (11 bits) or float (8 bits). Intel hardware can do this but not in a strictly correct sense. The x86 actually performs the computation in extended precision and then truncates the result to the number of required bits of precision. There are other problems as well, such as having to emulate just about everything that would happen in a lower-precision setting. The details are in the Java Grande Report and include issues such as narrowing the result's exponent, preserving all exceptions, and so on. The end result is that float and double require many twistings and contortions, resulting in a tenfold increase in execution time for most floating-point codes.

Within the JGF, many of us have long argued that the floating-point issue represents one of the most significant obstacles for Java to overcome as a scientific programming language. There is some good news to report on this front. The latest Java specification incorporates many of the JGF's ideas; Java implementations can now exploit the extended floating-point capabilities of processors, and those who want strictly reproducible results must now use a keyword qualifier strictfp (for strict floating point) to indicate that a particular context requires reproducible results. We're pleased that it will be optional to have reproducible results for floating point and not the default case.

Another desire in floating point (expressed in the Java Grande Report) is to have the ability to use an FMA instruction, which is available on many popular microprocessors. FMA stands for *fused multiply and add* and is considered to be as important to floating point as an increment in place instruction is for scalar integer data. This is not just something we want for recreational purposes. The FMA instruction lets floating point operate much faster, because it allows an extra load operation to be suppressed. Loading floating-point values, as opposed to loading integral values, comes at a much higher overhead, because the size of a floating-point value (in bits) is typically larger than the data bus width and thus requires more than one phase to be loaded.

Lightweight (value) objects

Classes represent the only current Java mechanism to extend the programming language with user-defined types. Consider again the complex class shown earlier. Java would let you create an instance of the class (an object) as

```
Complex c = new Complex
  (1.2, 3.4);
```

What exactly happens when you do this? The answer is somewhat involved but essentially,

1. An allocation request is issued to Java's memory manager. The storage required is the space to store the object's header and the class's variables.
2. Memory might or might not be available. If there is no suitable memory available, the memory might have to be compacted to free up space to perform the allocation.
3. A pointer is returned to store the new complex object.

Step 3 shows a key difference between Java and its predecessor, C++. In Java, storage must always be allocated on the heap, and heap allocation generally takes longer than allocation on the stack or in global storage. This led to another Java Grande recommendation: lightweight objects. A lightweight object differs from its heavyweight counterpart in its memory usage requirements and is primarily intended to define values that would never need to be extended using the concept of subclassing or inheritance. (The concept of an *integer* is such an example. It represents values and provides well-defined operations that never change. The concepts of complex numbers and, to a large extent, character strings, are similar.)

Value objects have been a part of the C++ language design from day one. They were explicitly removed in Java, primarily because having different kinds of objects leads to confusion at times. (Questions such as, "When do I use a value object versus a heap-allocated object?" *always* come up in the C++ community.) There are several implications but one thing is clear: value objects are faster and generally result in much more efficient code generation from a compiler's point of view. Value objects, much like other scalar data types (such as int and float) make it possible to use techniques such as inlining effectively, not to mention eliminating certain object-oriented overheads such as the v-table, which supports dynamic method dispatch. They also operate typically at approximately half the speed of subroutine or function calls in Fortran and C.

The notion of value objects has implications beyond the storage of simple values. It makes a significant difference when using collections of values such as arrays. In scientific codes, for example, arrays of complex numbers are frequently used, and we gain a significant storage and performance advantage by not allocating a heavyweight object for each complex number in the array.

Array layout and multidimensional array support

Array processing is another aspect of Java that detractors often point to as a weakness. There are two aspects of array processing in Java that are less than optimal from an implementation standpoint: layout and multidimensional array handling.

Let's first focus on the notion of layout. C and Fortran programmers have grown accustomed to the notion of contiguous layout for array structures. The notion of contiguity is expressed simply as storage where adjacent, or nearly adjacent, elements of the array are stored nearly adjacent to one another in actual memory. The same is not true of Java. It uses tree structures to store the array's elements, even in the case of one-dimensional arrays. Being a systems person first and a languages person second, I know from my earliest studies of operating systems the importance of locality. Java arrays, because they are all over the place, can exhibit quite poor locality, resulting in generally lower performance than equivalent code written in C or Fortran.

Multidimensional array processing is also absent from the current Java. You can define multidimensional arrays, but when it comes to the underlying representation and implementation, such arrays are maintained as one-dimensional arrays of one-dimensional arrays. The end result is extra work to find the element (an additional indexing operation per dimension).

Through the Java Community Process, there is an active proposal to change this and an apparent interest in doing so.

Generic Java: Still not numerics friendly

One of the features needed in Java is support for generic classes. The concept of genericity originally arose in Ada and Eiffel programming languages and was supported through the template mechanism in C++. A generic class is one that is defined in terms of other classes. Let's take a look at where these are needed.

Suppose you wanted to define a multidimensional array class in Java. You might go about it as

```
class Int_2D {
  private int [] data;
  private int d1, d2;

  public Int_2D(int d1,
      int d2) {
    data = new int [d1 * d2];
  }
  public int getElementAt
      (int i, int j) {
    return data[i * d2 + j];
  }
}

  public int setElementAt(int i,
      int j,int value)
      data[i * d2 + j] =
value;
}
```

Here's where it starts to get a bit messy. Suppose you wanted to define a multidimensional array class in Java that can be defined in terms of other types besides integer (int). Let's take a look at how C++ would have done this:

```
template <class T, const int
  dim1, const int dim2> {
private:
  T data[dim1][dim2];
public:
  T getElementAt
      (int i, int j) {
    ...
  }
}
```

C++ templates are much maligned (and rightly so) for sloppy implementations and an almost total lack of portability. However, there was a concern about making genericity work for scientific programming needs. These needs are very real, as exhibited by Linpack's design, which has a clever naming scheme for handling all the different methods and combinations of integer, floating-point, and double-precision cases. Of course, Linpack had to work with Fortran and, when written, even had to deal with complexities such as limited identifiers. Modern programming languages can do a great deal to help in the engineering of such codes, but Java does absolutely nothing to help in the engineering of such libraries except to allow the existing library to be wrapped using a framework called JNI. This is good for reuse but not necessarily a good use of the object paradigm. (As an aside, there are active projects ongoing with respect to scientific libraries and Java.)

Generic classes have great potential to change the landscape for scientific computing libraries, because generic classes can be written to anticipate changes that inevitably come about in computer architecture, such as higher-order floating-point types.

Java 1.4 and beyond implement generic classes, but early evidence supports that it won't be a numerics-friendly design. Generic classes in Java will only support template parameters that are class types. This means that building collection classes (lists, stacks, and arrays) of scalar data or fixed size will be infeasible. Those who really want such features will have to resort to ad hoc preprocessors that can process Java class templates and generate specific versions.

A look at C#: All vaporware or HPC-friendly?

C# addresses many issues covered in the Java Grande Report. It is almost as if they read our minds. I'll provide a quick tour of the C# features that closely relate to issues raised in our report.

Floating point

This perhaps best summarizes the progress on floating point, at least when observing C#. It comes from the C# specification, section 4.1.5:

> Floating-point operations may be performed with higher precision than the result type of the operation. For example, some hardware architectures support an"extended" or "long double" floating-point type with greater range and precision than the `double` type, and implicitly perform all floating-point operations using this higher precision type. Only at excessive cost in performance can such hardware architectures be made to perform floating-point operations with less precision, and rather than require an implementation to forfeit both performance and precision, C# allows a higher precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form `x * y / z`, where the multiplication produces a result that is outside the `double` range, but the subsequent division brings the temporary result back into the `double` range, the fact that the expression is evaluated in a higher range format may cause a finite result to be produced instead of an infinity.

In other words, see the Java Grande Report. We called specifically for this change, as I pointed out earlier. The C# specification's wording makes it abundantly clear that C# intends to provide the implementation with great latitude to give the best possible implementation of floating point. More importantly, we can expect to see proper working floating point in the reference (Microsoft) implementation.

Box–unbox concept

Java programmers frequently must use wrapper classes when working with numeric data in a collection. Collections are higher-level data structures than arrays, such as growable lists, mapping structures, and queues. Increasingly, computational scientists use data structures beyond basic arrays. I have worked with colleagues on some astrophysics applications that use tree-like data structures.

Java's Hashtable data structure is a so-called associative mapping structure. It can associate any object with any object (sometimes called a one-to-one association). Hashtables can maintain properties, which are used in computational science applications to configure an experiment:

```
Hashtable myParameters = new
  Hashtable();
myParameters.put("dataset_
  name", "xyz.dat");
myParameters.put("dimensions",
  new Integer(2));
myParameters.put("dim1", new
  Integer(100));
myParameters.put("dim2", new
  Integer(200));
```

This snippet indicates one of Java's problems: It goes a bit overboard when it comes to object orientation. To use the Hashtable, the *values* being placed in the table must be *objects* (meaning no scalar data can directly be used where an object is required). Here the call to create a new instance of the integer class results in an object that wraps the value 100. When we want the value of the `dim1` property, we must write code such as

```
int dim1 = ((Integer)myParameters.
  get("dim1")).intValue();
```

Even advanced programmers with strong computer science backgrounds find this kind of code to be extremely tedious to write.

C# has introduced the notion of boxing and unboxing, which other research programming languages have used for some time. In C#, the concept is worked out to a sophisticated degree. The details are related to the discussion of value objects. The concept of boxing is best understood as follows: When there is a need to store a value and an object is required, an implicit conversion will take place to put the value in an object box. Thus an object is created implicitly in this case. (C++ had the ability to approximate this concept with conversion operators; C# uses a more general mechanism that does not require any programmer intervention and is less error-prone.)

The example could then be written (just focusing on the lines where integer objects are explicitly constructed):

```
myParameters.put("dimensions", 2);
myParameters.put("dim1", 100);
myParameters.put("dim2", 200);
```

The unboxing concept is clever, but it does not perform the unboxing implicitly—a casting operation is still required. Even in this case, the amount of coding (and performance) is better and more comprehensible to boot:

```
int dim1 = (int) myParameters.
   get("dim1");
```

It is perfectly valid to claim that this is merely syntactic sugar, but the code bloat factor nevertheless decreases sig-

Opcode Summary for Java

Adding operation codes (opcodes) to Java is problematic. Many of the desired changes to Java as a language (Java Grande's wish list and others) require several opcodes to be added. Table A shows most of the current opcodes in the present Java Virtual Machine.

This table alone is not sufficient to understand the problem, which is due to opcode size (8 bits, or 256 are available—not all of which are actually available because they're reserved). There are well over 200 instructions in the current JVM instruction set.

I won't provide a detailed analysis here but will shed some light on what it takes to add a new type to the JVM. Most instructions are typed—for example, integer instructions for arithmetic and logical operations always begin with i (`iadd`, `isub`, `imul`, `iand`, `ior`). The same applies to most other scalar types. To add complex as a type requires a significant number of opcodes to be added. Although it is likely that adding complex alone would not exhaust the available opcodes, it is likely that there wouldn't be many left. There are other similar requests under review through the Java Community Process. It is unclear whether such requests can be accommodated without substantial JVM changes (and controversy).

There is support for extended opcodes, which brings me back to the comment about some standard opcodes being reserved. However, this represents a major language change; and (as mentioned in the main text) JVM changes tend to come slowly, if at all, except for major language revisions.

Table A. Current opcodes in the present Java Virtual Machine.

Category	Number of instructions	Examples
Arithmetic operations	24	`iadd, lsub, frem`
Logical operations	12	`iand, lor, ishl`
Numeric conversions	15	`int2short, f2l, d2I`
Pushing constants	20	`bipush, sipush, ldc, iconst_0, fconst_0`
Stack manipulation	9	`pop, pop2, dup, dup2`
Flow control instructions	28	`goto, ifne, ifge, if_null, jsr, ret`
Managing local variables	52	`astore, istore, aload, iload, aload_0`
Manipulating arrays	17	`aastore, bastore, aaload, baload`
Creating objects and arrays	4	`new, newarray, anewarray, multianewarray`
Object manipulation	6	`getfield, putfield, getstatic, putstatic`
Method call and return	10	`invokevirtual, invokestatic, areturn`
Miscellaneous	5	`throw, monitorenter, breakpoint, nop`

nificantly. Java's detractors in computational science often point to issues such as the wrapper classes as a deficiency in need of a remedy. One of the JGF recommendations, value classes, addresses the issue of wrapper classes, which are a special case.

Array layouts

C# supports both rectangular and sparse (jagged) array data structures (see the related sidebar). I have addressed the problem of Java array processing earlier in this article. In C#, rectangular arrays of dimension 2 and higher can be defined:

```
float [,] f2d = new float[d1,
  d2];
```

The jagged array syntax is patterned after Java's current array syntax and will not be discussed further. The semantics are the same. C# supports Java-style arrays for those who want them.

The semantics of C# array processing are much closer to Fortran's style than C or C++. I say this because there is no absolute requirement to access the first dimension before the second dimension in the case of a 2D array. There are two ways to subscript arrays in C#. For the rectangular array,

```
float f2d_00 = f2d[0, 0];
```

If the array is not rectangular, which is possible if the Java-style array allocation is used, we would write the access to the first element:

```
float f2d_00 = f2d[0][0];
```

There is a clear difference in these two accesses from a performance point of view. The second case requires two array references (and is the way Java currently works). The first case results in a single array reference with code to compute the offset from the base memory address associated with `f2d`.

Value objects (structs): A retro flashback

C# also supports value objects or so-called struct types. I call this a retro flashback because it's one of those rare cases where a feature that every object-oriented language designer thought was obsolete is back from the dead. I cite the C# specification again:

> Structs are similar to classes in that they represent data structures that can contain data members and function members. Unlike classes, structs are value types and

Aliased and Jagged Array Structures

Java is not particularly well suited to multidimensional array processing. In fact, multidimensional array processing in Java is reminiscent of the same concept implemented in C using pointers, which is fascinating considering that Java does not have a pointer concept—only a reference concept.

Java arrays are also particularly difficult to deal with at compile time because they are not declared statically. Every dimension of an array is allocated with the new operator and the argument to `new` can in fact be a variable (and not a constant), so implementing most of the traditional compiler optimizations used in Fortran and other language compilers is next to impossible.

There are two problems with Java arrays—aliasing and jaggedness—that the following code illustrates:

```
public class Jagged {

   public static void main(String args[]) {
      int i, j;
      int jaggedArray[][] = new int[2][];

      /* The following line creates an
           ALIASED array. */
      jaggedArray[0] = jaggedArray[1] =
           new int[10];
      for (i=0; i < jaggedArray[0].length;
              i++)
           jaggedArray[0][i] = i;

      System.out.println(“Printing the
           aliased array.”);
      for (i=0; i < jaggedArray.length; i++) {
        for (j=0; j < jaggedArray[i].length; j++)
           System.out.println(“[” + i + “][”
              + j + ”] = “ + jaggedArray[i][j]);
         System.out.println(“\n”);
      }

      /* The following line turns the array
           into a JAGGED array. */

      jaggedArray[1] = new int[5];
      for (i=0; i < jaggedArray[1].length;
           i++)
         jaggedArray[1][i] = i;

      System.out.println(“Printing the jagged
           array.”);
      for (i=0; i < jaggedArray.length; i++) {
         for (j=0; j < jaggedArray[i].length;
              j++)
            System.out.println(“[” + i + ”][” +
              j + ”] = “ + jaggedArray[i][j]);
         System.out.println(“\n”);
      }

   }
}
```

In this example, we see the construction of an aliased array and subsequently a jagged array—all by modifying the same array. Creating an aliased array is fairly straightforward. We first allocate the outer dimension and then initialize each of the rows to refer to another allocated array—the same array, in fact—of dimension 10. The output makes it clear that each row contains the same data.

The second comment marks the beginning of the jagged array structure's definition. In a jagged array, the number of elements in each row differs. We achieve this by taking the same array, allocating a smaller array, and then assigning the reference to the second row of the array `jaggedArray`. For each construction, output is generated to make it clear how Java can achieve these two unusual phenomena:

```
Printing the aliased array.
[0][0] = 0
[0][1] = 1
[0][2] = 2
[0][3] = 3
[0][4] = 4
[0][5] = 5
[0][6] = 6
[0][7] = 7
[0][8] = 8
[0][9] = 9

[1][0] = 0
[1][1] = 1
[1][2] = 2
[1][3] = 3
[1][4] = 4
[1][5] = 5
[1][6] = 6
[1][7] = 7
[1][8] = 8
[1][9] = 9

Printing the jagged array.
[0][0] = 0
[0][1] = 1
[0][2] = 2
[0][3] = 3
[0][4] = 4
[0][5] = 5
[0][6] = 6
[0][7] = 7
[0][8] = 8
[0][9] = 9

[1][0] = 0
[1][1] = 1
[1][2] = 2
[1][3] = 3
[1][4] = 4
```

do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

… the simple types provided by C#, such as *`int`*, *`double`*, and *`bool`*, are in fact all struct types. Just as these predefined types are structs, so it is possible to use structs and operator overloading to implement new "primitive" types in the C# language. Two examples of such types are given in at the end of this chapter…

What is of particular interest in this part of the specification is the keen sense of awareness the C# designers have when it comes to performance. The struct concept is not the same as C's struct mechanism. It's more like the class concept defined in C++ with a different keyword to show that it is not on the same standing as classes but not altogether removed from the class concept. The key difference: Structures are used for objects that are value-oriented only. They cannot be extended with subclassing as classes can be extended; they can actually be used in any situation where an object is required. (To understand why this is necessary, take a second look at the example of boxing and unboxing.)

C#'s structure concept is exactly what the Java Grande Report lists as a specific recommendation to Sun.

The lowdown on C#

C#, the language, has a long way to go before it will top my list of recommended languages. I remain a believer in Java. At the moment, C# is still a one-platform show. Ultimately, languages are not chosen merely for features but often just to have a choice—the same reason we choose to be a free society. At the same time, the C# design, which clearly drew inspiration from the Java design, appears to play effectively to Java's obvious weaknesses. It is important—urgent—for Sun Microsystems to take a closer look at what is going on in C# and realize that Microsoft has less of a history than Sun when it comes to high-performance computing.

Having worked with many similar languages from the early days of programming—such as Smalltalk—and having written compilers and interpreters for object-oriented languages, I worry for Java's future if performance issues don't come to the forefront soon. As I see other language designs emerge that have adopted the JGF recommendations almost verbatim, I wonder why JavaSoft has not embraced the proposals for which we worked painstakingly to frame with solid arguments and motivational examples.

There is no question that Java as a language design shows great sensitivity by not endorsing every random feature at the language level and by approaching JVM changes even more cautiously so as not to break existing codes. The language evolution is happening crawling—almost at the pace of the C++ effort, which lasted almost 10 years—and there is a great deal of concern about what this means for those who want to use Java for cutting-edge applications in performance-critical settings.

High-performance scientific and technical computing is no longer just a niche area. It is abundantly clear that the issues raised in the JGF Report are of importance in a general context—the context of general computing. The demands being placed on computers require performance at every level. The issues of array processing and value objects, for example, are important for efficient implementation of graphical interfaces and general-purpose network computing. Java does a good job in these areas but not a great one. If the JGF recommendations are adopted in full, there is no reason (except dollars) that your codes cannot run as fast as C, C++, or Fortran—if not faster.

George K. Thiruvathukal is a visiting associate professor of computer science at Loyola University in Chicago, Illinois. He also holds an adjunct professorship in the ECE department at Northwestern University. He teaches courses in computer science and engineering, including programming languages, operating systems, object-oriented design and programming, and distributed systems. He has coauthored two books: *High-Performance Java Platform Computing: Threads and Networking* (Prentice Hall, 2000) and *Web Programming in Python: Techniques for Integrating Linux, Apache, and MySQL* (Prentice Hall, 2001). Contact him at Loyola Univ., Dept. of Computer Science, 6525 Sheridan Rd., Chicago, IL 60626; gkt@toolsofcomputing.com.