A Note on the Use of Java in Scientific Computing

Juan Villacis

Computer Science Department, Indiana University

Bloomington, IN 47405, USA

jvillaci@cs.indiana.edu

Abstract

Java provides an easy-to-use language and platform for writing distributed, network-aware applications. For the scientific computing community, Java has the potential of extending the range and usefulness of scientific codes, as well as widening the codes current user base. However, certain features of Java may limit its usefulness as a platform in which to develop and deploy scientific codes. In this paper, we analyze where Java could be used within scientific computing, and examine several Java-based systems that aim to provide tools and runtimes suitable for programming scientific codes within distributed computing environments.

1 Introduction

One of the main benefits that Java brings to the scientific computing community is an easy-to-use, portable framework that facilitates distributed computing. Programmers can take advantage of this framework to convert otherwise stand-alone scientific codes into network-aware applications. However, speed, numerical precision and operator expressiveness, are important factors in most scientific codes, and a careful examination in these areas should be considered before diving into the programming. In the remainder of the paper, a discussion of these and related issues is presented from a programmer's perspective. An alternative programming paradigm that is not Java-specific, but which may take advantage of several key features of Java, is offered, along with several examples systems which illustrate the concept.

2 Overview of Issues

The kinds of codes used in scientific computing typically involve compute-intensive numerical calculations, as well as the management and distribution of large data sets. So a natural question to ask is: how does Java fit in this picture? At first glance, two extremes exist: use Java as a replacement for everything (equivalently, *program* everything in Java), or don't use Java at all. While the latter is not very interesting, the former does require some analysis. In this section, we explore several issues that identify Java's limitations as a programming environment and platform for scientific computing. A more comprehensive analysis, along with interesting debates on these issues, can be found in [11].

2.1 Platform Independence

By design, Java provides a platform-independent view of computing resources that programmers can access and rely on [1]. The Java Virtual Machine (JVM) and accompanying set of Java Developer Toolkit (JDK) libraries constitute the *platform* to which Java programmers code. This platform hides much of the underlying hardware, operating system, and even JVM/JDK implementation details through its use of a uniform application programming interface (API). A consequence of this is that programmers rely heavily on the JVM vendors to provide optimizations for key operations (e.g., floating point, vector arithmetic, etc.). Although just-in-time (JIT) compilation and related technologies promise to improve overall JVM performance, it is not clear whether specific optimizations unique to scientific codes will be available. Hence, optimal, or even good, performance relative to what might be obtained by programming "closer to the machine" may not be achievable within a Java-only environment.

2.2 Numerics

Java is not suitable for doing precise scientific computations due to its present lack of support for the IEEE floating point standard [16]. This means that numerical calculations which require a certain level of floating point accuracy and consistency cannot be guaranteed to possess such characteristics across all JVM implementations. This poses a fundamental problem for many codes.

2.3 Bandwidth

Java Remote Method Invocation (RMI) provides a framework that allows an object residing in one JVM to invoke methods on an object residing in another JVM. This facilitates distributed computing within Java, and makes it very easy to pass data between remote objects. However, RMI is not well-suited for high-bandwidth data passing due to its inefficient object serialization [3]. Furthermore, preliminary studies show that improvements in this area have not materialized under the most recent JVMs [5]. Thus, bandwidth scalability (and by extension, *problem size scalability*) are still unresolved issues in Java.

2.4 Memory Management

Java uses garbage-collection to automatically manage memory. Thus, the programmer has no direct control of *when*

memory resources are freed. Furthermore, since direct access to memory is prohibited (e.g., no pointer arithmetic), code which uses memory optimization techniques would need to be re-designed and re-written for Java. This can be problematic for codes which require large data sets, or whose algorithms depend on or assume manual memory management.

2.5 Legacy Code

A large percentage of scientific codes are written in Fortran and C, while a smaller percentage are written in C++ and other higher-level languages. It could be argued that these codes should be re-written in Java, in order to take advantage of the object-oriented paradigm and the Java platform features [6]. Unfortunately, direct support for complex types and an easy-to-use syntax for multi-dimensional arrays are not currently available as part of the core Java language/platform [16]. Furthermore, a significant amount of time and effort has been spent on getting existing scientific codes to work efficiently and reliably on specific platforms. Hence, there will be a reluctance to port existing codes to Java.

3 Middle Ground

Given the preceding points, Java seems ill-suited as a "replacement" language/platform for writing scientific codes. However, between the "all-or-nothing" Java extremes lies the possibility for using Java as the "glue" to connect existing scientific codes. That is, Java could be used for building software frameworks (or "glueware") that manage the *use* of scientific codes. Or, to put it another way, one need not write scientific codes *in* Java, rather, one can write Java *around* scientific codes.

3.1 Java as "Glueware"

The notion of building compositional tools to incorporate legacy codes into an underlying software framework is not new. In fact, there are quite a few non-Java software systems currently being used as the "glue" to connect scientific codes. In particular, scripting languages such as Perl, Python and Tcl, have recently been used in the construction of several scientific computing frameworks [19]. So, what makes Java better than these other "glue" programming systems? Perhaps no single feature gives Java a complete advantage over other systems (language preferences notwithstanding). But as an *integrated system*, Java does provide key mechanisms that facilitate its use as "glue". These include a uniform API (described below) for accessing non-Java codes, and built-in remote object interoperability facilities such as RMI [13] and IDL [15] that allow these codes to be accessible within a wider distributed computing environment.

3.2 Programming Approaches

Assuming Java is to be used as the "glueware" for scientific computing, two broad programmatic approaches become readily apparent. The first is to use a fine-grained, library-based approach that stays close to the Java platform. The second is to embrace language/platform heterogeneity and use a coarse-grained, component-based approach. Both approaches involve the use of *wrapper codes* which provide facilities for invoking functions and passing data between Java and (potentially) non-Java systems. A brief overview of these approaches is outlined below.

The library-based approach is a small step away from the Java-only extreme. This approach involves using the Java Native Interface (JNI) to make function calls between Java code and non-Java codes [12]. Depending on how far the programmer wishes to step into (or out of) the Java-only environment, the wrapper code could range from minimal (e.g., use Java only as a high-level front-end), to maximal (e.g., all code is in Java except for natively compiled computational kernels). This approach may work best for a restricted set of platforms for which little or no communication is required (e.g., stand-alone programs), or which can take advantage of the target platforms' special capabilities to enhance performance. In short, programmers can use this approach to retain the benefits of working within Java, while gaining some control over platform-specific optimizations.

An alternative is to move away from the Java-only extreme and adopt a language/platform neutral approach. In this scenario, scientific codes can be written in whichever language is convenient or appropriate to use, and for which wrapper code has been provided. The wrapper code provides access to a new kind of platform: the component framework. This framework defines a set of *interfaces* that describe the kinds of functions that may be invoked on the component. Like the previous approach, the incorporation of scientific codes into the framework may entail an initial lower-level programming phase (e.g., compiling and/or linking against a set of framework libraries to produce a run time executable). However, once the assimilation is complete, a different programming paradigm emerges: application level programming. Here, programmers, as well as end users, are able to assemble and compose scientific component codes to form larger applications based solely on a components public interface. This is one of the key ideas behind *component-based programming* [18].

4 Example Frameworks

A sampling of several Java-based frameworks that utilize the "glue" concept, and which specifically target scientific computing in a distributed computing context, are described below. A more comprehensive review of these and related systems can be found in [19]. These frameworks are examined in decreasing order of Java-dependency.

4.1 Symphony

Symphony is a client/server framework for specifying and transparently executing distributed (legacy) applications. It relies heavily on the JavaBeans [14] architecture to provide much of its functionality. Although programmers are required to write their scientific codes as "beans" (Java-centric components), certain portions may be JNI-wrapped. The server side is implemented as a Java daemon process, which performs the actual code execution. The client front end is derived from the Java BeanBox GUI, but does provide additional value with a set of simple, yet powerful, generic beans. Finally, beans communicate through Java itself (e.g., via Java RMI, Java sockets, etc); however, the underlying codes are not necessarily restricted to this. [17].

4.2 IceT

IceT is a system that facilitates the transport and dynamic execution of of native codes via Java [9]. Using the IceT

API, programmers can write JNI-wrapped codes optimized for specific platforms, and the IceT system handles the instantiation details at runtime via a "soft install" mechanism. The IceT framework automatically detects and sets up the proper environment for running the code on compatible machines within a distributed computing environment. By using IceT, the programmer is afforded some degree of runtime portability to his otherwise non-portable JNI-wrapped code.

4.3 WebFlow

WebFlow is a client/server system that enables high-performance commodity computing for dataflow applications [2]. The server side is implemented as mesh of Java servers, called the WebVM, which manage and coordinate distributed computations. Programmers can write their codes in whichever language is most suitable, and then use a Java-based API to to encapsulate their scientific codes as "modules" in the WebFlow system. The client side consists of a Java applet front end that allows users to visually compose modules into distributed applications.

4.4 VDCE

The Virtual Distributed Computing Environment (VDCE) provides a problem solving environment for performing parallel and distributed computing over wide-area networks [10]. The system is composed of two parts: an application editor and a runtime system. The Java-based application editor provides a set of libraries for developing VDCE applications, as well as a GUI for visually composing applications from a database of components. The runtime system provides a task scheduler for mapping components to a matching set of resources, a real-time task monitor for obtaining feedback on the status of components, and a socket-based point-to-point inter-component communication system. To use this system, programmers must write their codes using the VDCE support libraries, which allow components to interact with the application editor, as well as the runtime kernel services.

4.5 CAT

The Component Architecture Toolkit (CAT) facilitates component-based programming by providing programmers and end users with the following items: a conceptually simple "port-based" component model, a suite of developer tools for incorporating existing scientific codes into its framework, and a set of end user tools for locating, composing, building and running distributed component applications. The framework has been implemented in both Java (for the GUI composition workspace, and Java-based components), and in HPC++ [8] (for components written in Fortran, C, or C++). Using the CAT component model, programmers can target their codes to whichever framework implementation best suits their performance needs. Nexus [7] provides the multi-platform, high-performance communication layer that, with the use of NexusRMI [4], enables Java components to inter-operate with non-Java components. A complementary resource information subsystem (RIS) allows both programmers and end users to publish information about components (static or running) in a distributed directory service so that other RIS users may access them. A more detailed description can be found in [20].

5 Conclusions

Java provides a rich object model, extensive set of core libraries, and a framework that simplifies distributed computing. However, Java falls short of providing a viable programming and execution environment for scientific computing due to its unsatisfactory performance, poor scalability, inadequate numerics, and lack of core support for operations/types unique (and very useful) to scientific computing programmers. Nonetheless, it has been shown that Java is quite useful as the "glue" for programming *around* scientific codes, and that one can thereby gain some advantages of Java's distributed computing facilities to broaden the class of users who can make use of such codes.

6 Acknowledgments

References

[1] ARNOLD, K., AND GOSLING, J. *The Java Programming Language*. Addison Wesley, 1996.

[2] BHATIA, D., BURZEVSKI, V., CAMUSEVA, M., FOX, G., FURMANSKI, W., AND PREMCHANDRAN, G. *WebFlow - A Visual Programming Paradigm for Web/Java-based Coarse Grain Distributed Computing*. June 1997. http://osprey7.npac.syr.edu:1998/iwt98/products/webflow/.

[3] BREG, F., DIWAN, S., VILLACIS, J., BALASUBRAMANIAN, J., AKMAN, E., AND GANNON, D. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++ Distributed Components. In *Concurrency Parctice and Experience, Special Issue from the Fourth Java for Scientific Computing Workshop* (March 1998), John Wiley and Sons, Ltd., p. (to appear).

[4] BREG, F., AND GANNON, D. Compiler Support for an RMI Implementation using NexusJava. Tech. Rep. 500, Computer Science Dept., Indiana University, 1997.

[5] BREG, F., AND GANNON, D. A Customizable Implementation of RMI for High Performance Computing. In *International Workshop on Java for Parallel and Distributed Computing, Second Merged Symposium IPPS/SPDP 1999* (1999), Computer Science Dept., Indiana University, p. (to appear).

[6] BUDIMLIC, Z., KENNEDY, K., AND PIPER, J. The Cost of Being Object Oriented: A Preliminary Study. In *First UK Workshop: Java for High Performance Network Computing* (July 1998), Center for Research on Parallel Computation, Rice University, p. (to appear).

[7] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computing 37* (1996), 70–82.

[8] GANNON, D., BECKMAN, P., JOHNSON, E., AND GREEN, T. *Compilation Issues on Distributed Memory Systems*. Springer-Verlag, 1997, ch. 3 HPC++ and the HPC++Lib Toolkit.

[9] GRAY, P., SUNDERAM, V., AND GETOV, V. Aspects of Portability and Distributed Execution for JNI-Wrapped Code. In *First UK Workshop: Java for High Performance Network Computing* (July 1998), p. (to appear).

[10] HARIRI, S., TOPCUOGLU, H., FURMANSKI, W., KIM, D., KIM, Y., RA, I., BING, X., YE, B., AND VALENTE, J. *Problem Solving Environments*. IEEE Computer Society, 1998, ch. A Problem Solving Environment for Network Computing.

[11] JAVAGRANDE NUMERICS WORKING GROUP. Improving java for numerical computation, 1998. http://gams.nist.gov/javanumerics/jgfnwg-01.html.

[12] JAVASOFT. Java Native Interface Specification, 1997. http://java.sun.com/products/jdk/1.1/docs/guide/jni/index.html.

[13] JAVASOFT. Remote Method Invocation Specification, 1997. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html.

[14] JAVASOFT. Java beans, 1998. http://java.sun.com/beans/index.html.

[15] JAVASOFT. Java idl tutorial, 1998. http://java.sun.com/docs/books/tutorial/idl/index.html.

[16] KAHAN, W., AND DARCY, J. How JAVA's Floating-Point Hurts Everyone Everywhere, March 1998. http://www.cs.berkeley.edu/ wkahan/JAVAhurt.pdf.

[17] SHAH, A. Symphony: A Java-based Composition and Manipulation Framework for Distributed Legacy Resources. Tech. rep., MS Thesis, Department of Computer Science, Virginia Tech, March 1998 1998. http://actor.cs.vt.edu/ vanmetre/symphony/.

[18] SZYPERSKI, C. *Component Software: Beyond Object-oriented Programming*. Addison-Wesley, 1998.

[19] VILLACIS, J. The CAT, PhD Thesis, Department of Computer Science, Indiana University, (in progress) 1999. http://www.extreme.indiana.edu/cat.

[20] VILLACIS, J., GOVINDARAJU, M., STERN, D., WHITAKER, A., BREG, F., DEUSKAR, P., TEMKO, B., GANNON, D., AND BRAMLEY, R. CAT: A High Performance, Distributed Component Architecture Toolkit for the Grid, 1999. http://www.extreme.indiana.edu/cat/papers/hpdc99.ps.

17