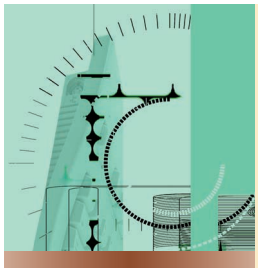
Java in High-Performance Environments

Bahador Ghahramani and Mark A. Pauley
University of Nebraska at Omaha

Although originally intended for use in digitally controlled consumer devices, Java became prominent because of features that made it useful for Internet programming. In recent years, however, it has gained growing acceptance as a general-purpose language. The increasing number of trained Java programmers now enjoy access to many resources, including a wide selection of class libraries.

Java programs are executed by a Java virtual machine (JVM), which interprets intermediate compiled bytecode that is nominally platform independent. Although early versions of Java interpreted unoptimized bytecode in a relatively unsophisticated manner, recent developments including static analysis, just-in-time compilation, JVM optimization, and instruction-level optimizations have improved execution efficiency. Consequently, Java is now competitive with C and C++ for some applications and on some platforms.

Despite Java's increasing popularity, there is a lingering perception that deficiencies in the language make it unsuitable for high-performance computing. Here we address some of those deficiencies and discuss the suitability of using Java in a distributed environment.



Java offers efficient interprocessor communication and sequential execution for high-performance computing applications.

CAN JAVA DELIVER?

For several reasons, Java typically has been viewed as inappropriate for high-performance computing:

- Java does not directly provide truly rectangular multidimensional arrays; rather, it simulates multidimensional arrays using arrays of arrays. Although this enables creation of very complicated array structures, Java must determine an array's shape during execution, which is an expensive runtime operation.
- Through its exception model, Java checks all array accesses for dereferencing via null-pointer and out-of-bounds indices, which also contributes to runtime overhead.
- Java generally prohibits code reordering, which prevents most optimizations of numerical codes.
- Java requires representing nonprimitive data types as full-fledged objects. In contrast, Fortran and C++ both use low-cost data structures allocated on the stack or in registers to support other arithmetic systems, including complex numbers.

Optimized libraries are important tools for enhancing performance in numerical applications. One way to address this in Java is to make existing native libraries, written in other languages such as C or Fortran, available through the Java Native Interface. Another option is to develop new libraries written entirely in Java.

To overcome the absence of rectangular multidimensional arrays in Java, IBM's Numerically Intensive Java project (www.alphaworks.ibm.com/tech/ninja) recently developed an Array package that provides multidimensional arrays of various types and ranks. In this package, arrays have an "immutable rectangular and dense shape" that simplifies testing for aliases and facilitates the optimization of runtime checks.

The runtime overhead of a method invocation for each element access is unacceptable for high-performance computing. To avoid this problem, a compiler can use *semantic expansion* to look for specific methods and substitute efficient code for the call. JVMs that recognize the Array package can use it to achieve high performance.

With the Array package, a compiler can perform simple transformations that eliminate the problems that Java's runtime checks and exception model cause. The compiler performs versioning of loop nests to create exception-free regions so that it can then apply code-reordering optimization. Because these safe regions cannot cause an exception, the code omits runtime checks.

The Array package also includes a complex number class. The compiler uses semantic expansion to convert calls to these methods into code that directly manipulates complex number values instead of objects. The compiler transforms the complex numbers into values for efficiency.

JAVA IN A DISTRIBUTED ENVIRONMENT

Java's efficient compiling and runtime systems make it well suited for parallel computing. Researchers are implementing two approaches to achieve this goal. Hyperion (www.irisa.fr/paris/pages-perso/Gabriel-Antoniu/dsm-pm2) compiles multithreaded bytecode for execution on a *distributed virtual machine* so that threads on different nodes can share objects. Alternatively, the Manta compiler (www.cs.vu.nl/manta/) statically compiles Java source code to executable programs to indicate which objects multiple threads will share.

Java is also effective in network-based grid computing. An emerging infrastructure aims to provide security, resource access, information, and other services that enable the controlled and coordinated sharing of resources among virtual organizations formed dynamically by individuals and institutions with common interests.

Underlying both parallel and grid computing is the need for coordination and communication mechanisms to apply multiple resources in a concerted fashion to a complex problem. Programmers have thus far had to rely on ad hoc and low-level solutions such as specialized message-passing libraries within parallel computers and communication mechanisms among networked computers.

Scientific and engineering applications combine tight performance constraints with dynamic computational requirements. To meet these needs, three communication and coordination programming approaches—remote method invocation (RMI), message passing, and component frameworks—have emerged, and each can be expressed effectively in Java.

Fast RMI

When components of a single program are distributed over less tightly coupled elements or when collective operations are rare, communication structures may be less predictable, and issues such as asynchrony, error handling, and ease-of-argument passing become more prominent.

Java's efficient compiling and runtime systems make it well suited for parallel computing.

As an object-oriented language, Java's main communication concept is method invocation. Inside a single JVM, concurrent control threads can communicate through synchronized method invocations. On a multiprocessor system with shared memory, Java's method invocation allows for limited parallelism by mapping threads for different physical processors. For distributed memory systems, Java's RMI transfers the method invocation, along with its parameters and results, across a network to and from the serving object onto a remote JVM.

To implement a remote invocation, Java encodes or *marshals* the procedure identifier and its arguments in a wire format that both the caller and the code being called understand. The callee uses a proxy object to decode or *unmarshal* the stream of bytes and then performs the actual invocation. The results travel in the other direction, from callee to caller.

Although RMI is truly object-oriented, supports all Java data types, and offers garbage collection, a regular implementation is costly in runtime overhead. Several projects aim to improve RMI performance using novel techniques such as precompiling marshaling routines, employing an optimized wire protocol, caching or replicating objects, minimizing memory copy operations and thread switching overhead, and using an efficient communication subsystem. The JavaParty project (www.ipd.uka.de/JavaParty/) has optimized both RMI and the object serialization in pure Java.

Message passing

Within parallel computers and clusters, communication structures and timings are often highly predictable for both senders and receivers and may involve multiparty or collective operations. Java includes several built-in mechanisms to exploit the inherent parallelism in many programs.

Threads and concurrency constructs are well suited for shared-memory computers but not for large-scale distributed-memory machines. Java provides sockets and the RMI mechanism for distributed applications. However, the explicit use of sockets is often too low-level in parallel computing, while RMI is often oriented toward client-server-type systems and does not specifically support the symmetric model adopted by many parallel applications.

A potential solution builds on the message-passing communication framework. Unlike sockets and RMI, explicit message passing directly supports symmetric communications—point-to-point as well as collective operations such as broadcast and gather all-to-all—as defined by the message-passing interface (MPI) standard.

The Java Grande Forum (www.javagrande.org/) was formed to agree on a common MPI-like API for message passing in Java. Unfortunately, direct MPJ implementation is usually slow. One solution to this problem is to employ more sophisticated design approaches such as the use of native conversion into linear byte representation as well as advanced compilation technologies.

Component frameworks

Developers can use high-performance component frameworks and compo-

nent technologies such as JavaBeans, Java's reusable software-component architecture, to compose and discover the properties of separately developed components and then use them to construct programs. Java uses commercially available visual application builder tools to compose software components into applets, applications, servlets, and composite components. Java can move, query, and visually integrate these components to provide a convenient new level of computer-aided software-engineering-based programming within the grid environment.

Realizing the full potential of emerging grids in which users deal with differing systems, diverse programming forms, and the needs of multiple-user communities requires further advances. Applications, users, and resource providers need adaptive services for security, resource management, data access, instrumentation, policy, and accounting.

Java eases this software engineering problem. Because of its object-oriented nature, ability to develop reusable software components, and integrated packaging mechanism, Java supports all phases of a software engineering project's life cycle, from problem analysis and design to program development, deployment, instantiation, and maintenance. ■

Bahador Ghahramani *is an associate professor in the College of Information Science & Technology, University of Nebraska at Omaha. Contact him at bghahramani@mail.unomaha.edu.*

Mark A. Pauley *is a Senior Research Fellow at the College of Information Science & Technology, University of Nebraska at Omaha. Contact him at mpauley@mail.unomaha.edu.*

Editor: Richard G. Mathieu, Dept. of Decision Sciences and MIS, St. Louis University, St. Louis, MO 63108; mathieur@slu.edu