MULTIPARADIGM IN JAVA FOR G COMPU

The result is a promising programming approach for enabling, controlling, and coordinating resource sharing in computational Grids.

The computational science community has long been at the forefront of advanced computing, due to its need to solve problems requiring resources beyond those provided by the most powerful computers of the day. Examples of such high-end applications range from financial modeling and vehicle design simulation to computational genetics and weather forecasting. Over the years, these considerations have led computational scientists to be aggressive and innovative adopters of vector computers, parallel systems, clusters, and other novel computing technologies.

More recently, the widespread availability of high-speed networks and the growing awareness of the new problem-solving modalities made possible when these networks are used to couple geographically distributed resources have stimulated interest in so-called Grid computing [6]. The term "the Grid" refers to an emerging network-based computing infrastructure providing 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 [7]. A number of ambitious projects are today applying Grid computing concepts to challenging problems like the distributed analysis of experimental physics data, community

COMMUNICATIONS

RID

TING

❖ Vladimir Getov,
Gregor von Laszewski,
Michael Philippsen, and
Ian Foster

access to earthquake engineering facilities, and the creation of "science portals," or thin clients providing remote access to the information sources and simulation systems supporting a particular scientific discipline.

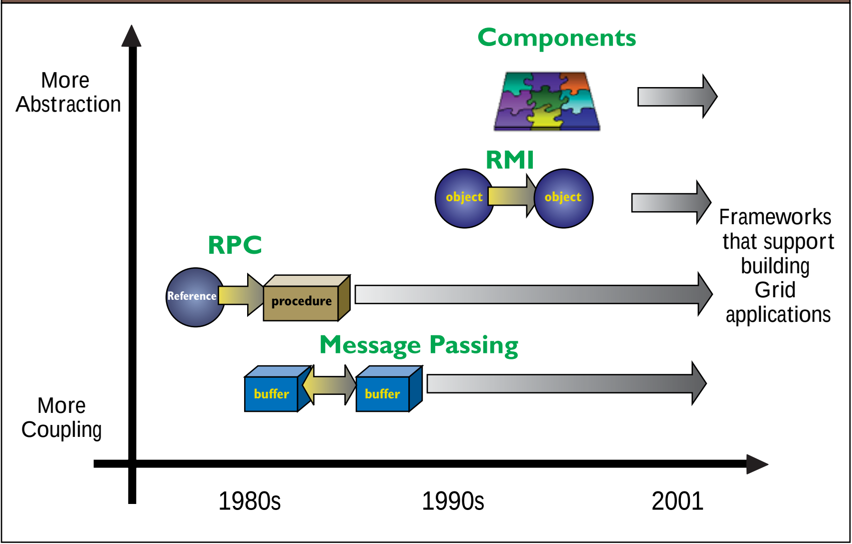
Underpinning both parallel and Grid computing is the common need for coordination and communication mechanisms allowing multiple resources to be applied in a concerted fashion to these complex problems. Scientific and engineering applications have for the most part addressed this requirement in an ad hoc and low-level fashion, using specialized message-passing libraries within parallel computers and communication mechanisms among networked computers.

While low-level approaches have let users achieve their application performance goals, an unfortunate consequence is that the computational science community has not benefited to any great extent from the advances in software engineering that have occurred in industry over the past 10 years. In particular, the Java programming environment, which seems ideal for multiparadigm communications, is hardly exploited at all. Java's platform-independent "bytecode" can be executed securely on many platforms, making the language an attractive basis for portable Grid computing. In addition, Java's performance on sequential codes, a prerequisite for developing such "Grande" applications, has increased substantially over the past few years [4] (see the sidebar "Java Grande"). Inspired originally by coffeecup jargon, the buzzword Grande is now also commonplace for distinguishing this emerging type of high-end applications when written in Java. Java also provides a sophisticated graphical user interface framework, as well as a paradigm for invoking methods on remote objects. These features are of particular interest for steering scientific instruments from a distance (see the sidebar "10 Reasons to Use Java in Grid Computing").

The rapid development of Java technology now makes it possible to support, in a single OO framework, the various communication and coordination structures in scientific applications. Here, we outline how this integrated approach can be achieved, reviewing in the process

CAREN ROSENBLATT

Figure 1. Multiple communication frameworks help program the diverse infrastructure in Grids. Remote procedure calls, message passing, remote method invocation, and component frameworks are the technologies of choice for building Grid applications.



the state-of-the-art in communication paradigms within Java. We also present recent evaluation results indicating this integrated approach can be achieved without compromising performance.

Communication requirements. Communication and coordination within scientific and engineering applications combine stringent performance requirements (especially in Grid contexts) with highly heterogeneous and dynamic computational environments. A number of communication frameworks have been introduced over the years as a result of the development of computer networks and distributed and parallel computing systems (see Figure 1). The pioneering framework—the remote procedure call (RPC)—has been around for at least 20 years. The message-passing paradigm arrived along with distributed memory parallel machines. More recently, other frameworks have been developed based on the RPC concepts: remote method invocation (RMI) and component frameworks. Three communication and coordination programming approaches have emerged, each of which can be expressed effectively in 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. Efficiency is the main concern, and message-passing libraries, such as the Message Passing Interface (MPI) [8], have become the technology of choice.

Remote method invocation. 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 such issues as asynchrony, error handling, and ease of argument passing become more prominent. In this context, such technologies as CORBA and Java's RMI have benefits.

Component frameworks. When constructing programs from separately developed components, the ability to compose and discover the properties of components is critical. Component technologies, including JavaBeans, and their associated development tools become very attractive, as do proposed high-performance component frameworks [2].

Message Passing

Java includes several built-in mechanisms allowing the exploitation of the parallelism inherent in a given program. Threads and concurrency constructs are well suited for shared-memory computers, but not for large-scale distributed-memory machines. For distributed applications, Java provides sockets and the RMI mechanism. For the parallel computing world, the explicit use of sockets is often too low-level, while RMI is oriented too much toward client/server-type systems and does not specifically support the symmetric model adopted by many parallel applications. Obviously, there is a gap within Java's set of programming models, especially for parallel programming support on clusters of tightly coupled processing resources. A solution inevitably builds on the message-passing communication framework, one of the most popular parallel programming paradigms since the 1980s.

The architecture of a message-passing system can generally follow one of two approaches: implicit or explicit. Solutions taking the implicit approach usually provide the programmer a single shared-memory system image, hiding the message passing at a lower level of the system hierarchy. Thus, a software developer works within an environment often called the distributed shared memory programming model. Translating the implicit solution to Java leads to development of cluster-aware Java virtual machines (JVMs) providing fully transparent and truly parallel multithreaded programming environments [1]. This approach preserves

Execution time (in seconds) for the Integer Sort kernel from the NAS Parallel Benchmarks on the IBM SP2. Use of JVM and MPJ here is approximately two times slower than the same code written in C and using MPI. When using HPCJ and MPJ, the difference disappears, and Java and M PJ perform as well as C and MPI for this experiment. This result confirms that the extra overhead introduced by MPJ is negligible, compared with MPI.

	Number of Processors			
	2	4	8	16
JVM + MPJ	48.04	24.72	12.78	6.94
HPCJ + MPJ	23.27	13.47	6.65	3.49
C + MPI	24.52	12.66	6.13	3.28

full compatibility with the standard Java bytecode format. However, these advantages result from adopting a complex nonstandard JVM that introduces additional overhead to the Java runtime system. This extra complexity makes it difficult for such JVMs to keep up with the continuous improvements and performance optimizations of the standard technology.

Java Grande

The notion of a Grande application is familiar to researchers, but the term itself is relatively new. Such applications might require any combination of high-end processing, communication, I/O, and storage resources to solve one or more large-scale problems. For the past four years, Java—as both a language and a platform for solving this type of problem—has been the focus of the international Java Grande Forum (www.javagrande.org), whose main goals are to:

- Evaluate and improve the applicability of the Java environment for Grande applications;
- Unite the Java Grande community to develop consensus requirements and act as a focus for interactions with the larger Java community;
- Create prototype implementations, benchmarks, API specifications, and recommendations for improvements to make the Java environment useful for Grande applications.

The Forum organizes open public meetings, the annual ACM Java Grande Conference, workshops, symposia, and panels. A large portion of the Forum's scientific work has been published in *Concurrency: Practice & Experience*, and, since 1999, the conference proceedings have been published by ACM Press. C

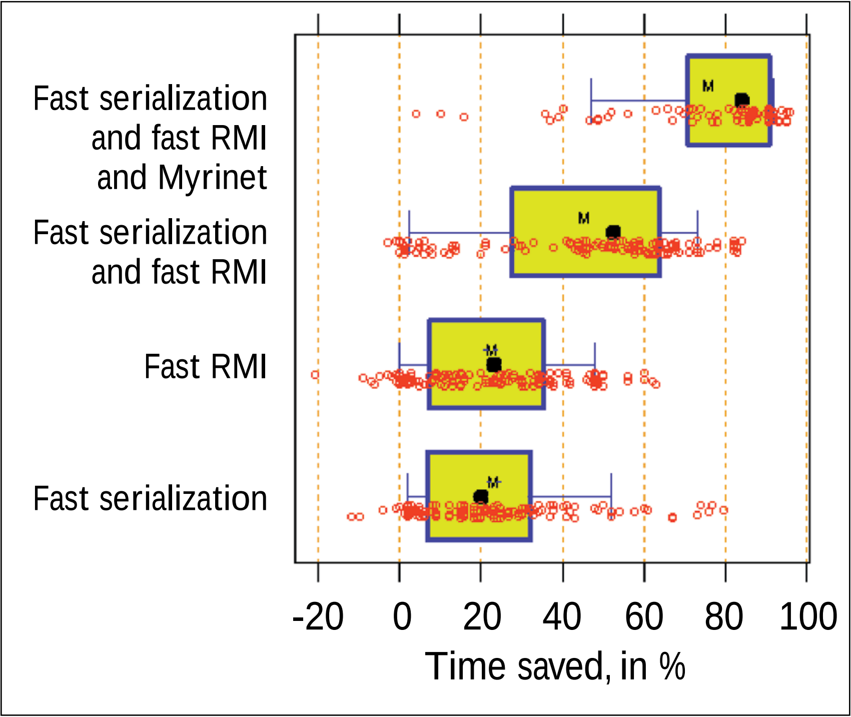
Unlike sockets and RMI, explicit message passing supports symmetric communications directly, including point-to-point and collective operations, such as broadcast, gather, all-to-all, and others, as defined by the MPI standard. Programming with MPI is relatively straightforward because it supports the single program multiple data (SPMD) model of parallel computing, wherein a group of processes cooperate by executing identical program images on local data values.

With the evident success of Java as a programming language, and its inevitable use in connection with parallel, distributed, and Grid computing, the absence of a well-designed explicit message-passing interface for Java would lead to divergent, nonportable practices. Indeed, in 1998, the message-passing working group of the Java Grande Forum was formed to agree on a common MPI-like application programming interface (API) for message passing in Java (MPJ) [5]. An immediate goal was to provide an ad hoc specification for portable message-passing programming in Java that would also serve as a basis for conversion between programs written in C, C++, Fortran, and Java.

MPJ can be implemented in one of two ways: as a wrapper to existing native MPI libraries or as a pure Java implementation. The former provides a quick solution, usually with negligible runtime overhead introduced by the wrapper software. However, using native code breaks the Java security model and does not allow work with applets (advantages of the pure Java approach). Unfortunately, a direct MPJ implementation in Java is usually much slower than wrapper software for existing MPI libraries. One solution to this problem is to employ more sophisticated design approaches. For instance, the use of native conversion into linear byte representation, often called "marshaling," and advanced compilation technologies for Java can make the two design options comparable in terms of performance. Our experiments have used the statically optimizing IBM High-Performance Compiler for Java (HPCJ), which generates native code for the RS6000 architecture, to evaluate the performance of MPJ on an IBM SP2 distributed-memory parallel machine. The results show that when using such a compiler, the MPJ communication component is as fast as the native message-passing library (see the table).

Closely modeled on the MPI-1 standard, the existing MPJ specification should be regarded as a first phase in a broader program aimed at defining a more

Figure 2. The bottom three benchmark plots each show 2 X 64 measured results; 64 points represent measurements on PCs connected by Ethernet; 64 stand for DEC Alphas connected by FastEthernet. The first (bottommost) plot is the runtime improvement achieved with regular RMI and fast serialization. The second plot is the improvement fast RMI achieves when used with Java's regular serialization. The third plot is the combined effect. The topmost plot (64 measured results) demonstrates what happens when the Alphas are connected by Myrinet, in addition to the fast serialization and fast RMI. In all plots, the small circles are individual data points; the fat dot is the median; the box indicates the 25% and 75% quantiles; the whiskers indicate the 10% and 90% quantiles; and the M and dashed lines indicate the mean plus/minus one standard error of the mean.



and the code being called, or the callee. The callee uses a proxy object to decode, or unmarshal, the stream of bytes and then perform the actual invocation. The results travel in the other direction, from callee to caller.

Although RMI inherits this basic design, it has distinguishing features beyond the original RPC. In addition, RMI is no longer meant to bridge OO and procedural languages or to bridge languages with different kinds of elementary types and structures. The main advantages of RMI are that it is truly object-oriented, supports all Java data types, and is garbage collected. Since most practitioners agree that, for sequential code, garbage collection saves programmer time, it is likely the same is true for distributed code as well. These features also allow the caller and the callee to be developed separately, as long as they agree on interfaces. As a result, software development and maintenance of distributed systems becomes much easier.

To illustrate these advantages, consider the remote invocation of a method `add(Atom name)`. The OO nature of RMI allows the caller to pass objects of any subclass of `Atom` to the callee. The object is encoded into a machine-independent byte representation (Java calls it "object serialization") that also includes information on the class implementation. More precisely, if the callee does not know the concrete class implementation of `name`, it can load the class implementation dynamically. When the caller invokes an instance method on `name`, say, `name.bond`, the `bond` code of the particular subclass of `Atom` is executed on the side of the callee. Thus, one of the main advantages of OO programming—reuse of existing code with refined subclasses—can also be exploited for distributed code development.

These novel features come at a cost in terms of runtime overhead. With the regular implementation of RMI on top of Ethernet, a remote invocation takes milliseconds; concrete execution times depend on the number and the types of arguments. About a third of the time is needed for the RMI itself, a third for the

Java-centric high-performance message-passing environment. We can expect future work to consider more high-level communication abstractions and, perhaps, layering on other standard transports, as well as on Java-compliant middleware. A primary goal should be to offer MPI-like services to Java programs in an upward-compatible fashion. Middleware developed at this level should allow a choice of emphasis—performance or generality—while always supporting portability.

Fast Remote Method Invocation

Remote invocation is an established programming concept behind both the original RFC [3] and Java's RMI. To implement a remote invocation, the procedure identifier and its arguments are encoded (marshaled) in a wire format understood by both the caller serialization of the arguments, and another third for the data transfer (TCP/IP-Ethernet). While such latency might be acceptable for coarse-grain applications with limited communication needs, it is too slow for high-performance applications running on low-latency networks, such as a closely connected cluster of workstations.

Several projects are under way to improve the performance of RMI, including Manta [9] and JavaParty [10]. In addition to improving the implementation of regular RMI by, say, removing layering overhead, they employ a number of novel optimization ideas:

Precompiling marshaling routines. The goal is to save the runtime overhead for generating these routines via dynamic type inspection.

Employing an optimized wire protocol. For type encoding, detailed type descriptions are needed only if the objects are stored into persistent storage. For communication purposes alone, a short type identifier may be sufficient.

Caching, or replicating, objects. The related techniques help avoid retransmission if the object's instance variables do not change between calls.

Minimizing memory copy operations. When efficiency is important, there should be as few memory copies as possible in either direction between the object and the communication hardware.

Minimizing thread switching overhead. Because Java is inherently multithreaded, traditional RPC optimizations are generally insufficient for minimizing runtime. In addition, optimized RMI implementations in Java cannot be as aggressive as native approaches because the JVM concept does not allow direct access to raw data and hides the way threads are handled internally.

Using an efficient communication subsystem. JavaParty's RMI is implemented on a Myrinet-based library (ParaStation.ira.uka.de) employing user-level communication, hence avoiding costly kernel operations.

The JavaParty project has optimized both RMI and the object serialization in pure Java. Remote invocations can be completed within 80 microseconds on a cluster of DEC Alpha computers connected by Myrinet (see Figure 2). With similar optimization ideas, the Manta group compiles to native code and uses a runtime system written in C, making it less portable compared to JavaParty (see Kielmann et al.'s "Enabling Java for High-Performance Computing" in this issue). Nevertheless, both projects report similar remote invocation latencies of 40 microseconds on clusters of Pentium machines.

10 Reasons to Use Java in Grid Computing

Language. The Java programming language includes features beneficial for large-scale software engineering projects, including object-orientation, single inheritance, garbage collection, and unified data formats. Since threads and concurrency-control mechanisms are part of the language, parallelism can be expressed directly at the user level.

Class libraries. Java provides a variety of additional class libraries, including functions essential for Grid computing, such as the ability to perform secure socket communication and message passing.

Components. A component architecture is provided through JavaBeans and Enterprise JavaBeans to enable component-based program development.

Deployment. Java's bytecode allows for easy deployment of the software through Web browsers and automatic installation facilities.

Portability. Besides the unified data format, Java's bytecode guarantees full portability as represented by the concept "write once, run anywhere."

Maintenance. Java contains an integrated documentation facility. Components written as JavaBeans can be integrated within commercially available integrated development environments.

Performance. Recent research results prove the performance of many Java applications can come very close to that of their C and Fortran counterparts.

Gadgets. Java-based smart cards, PDAs, and smart devices will expand the working environment for scientists.

Industry. Scientific projects are sometimes required to evaluate the longevity of a technology before it can be used. Strong vendor support helps make Java a technology of current and future consideration.

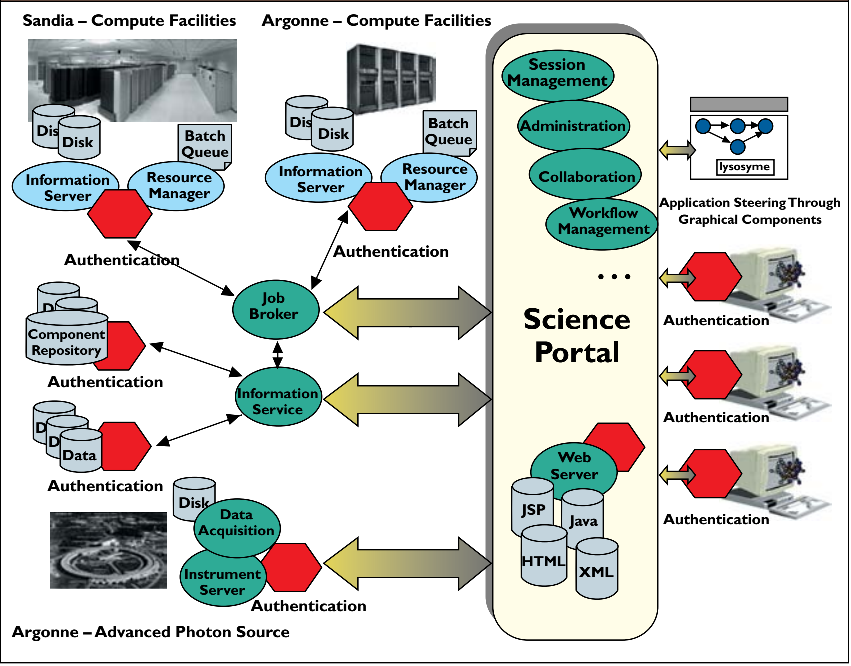
Education. Universities all over the world are teaching Java to their students. C

Adaptive Grid Computing

Besides making these communication paradigms available and efficient in Java, further advances are needed to realize the full potential of emerging Grids in which users deal with heterogeneous systems, diverse programming paradigms, and the needs of multiple user communities. Adaptive services are needed for security, resource management, data access, instrumentation, policy, and accounting for applications, users, and resource providers.

Java eases this software engineering problem. Because of its OO nature, ability to develop reusable

Figure 3. With the help of the Java Commodity Grid Kit (available from Argonne National Laboratory, www.globus.org/cog), we constructed a domain-specific science portal from reusable components. The components access Grid services for authentication, job submission, and information management. Workflow management and other advanced services must be developed to provide an integrated problem-solving environment for scientists.



software components, and integrated packaging mechanism, Java offers support for all phases of the life cycle of a software engineering project, including problem analysis and design and program development, deployment, instantiation, and maintenance.

Java's reusable software-component architecture, called JavaBeans, allows users to write self-contained, reusable software units (see the sidebar "Components and JavaBeans"). Using commercially available visual application builder tools, software components can be composed into applets, applications, servlets, and composite components. Components can be moved, queried, and visually integrated with other components, enabling a new level of convenient computer-aided-software-engineering-based programming within the Grid environment.

Component repositories, or containers, allow a number of engineers to work collectively on similar tasks and share the results with the community of scientific users and engineers. Moreover, the Java framework includes a rich set of predefined Java application protocol interfaces, libraries, and components supporting access to databases and directories, network programming, sophisticated interfaces to XML, and more.

We've been evaluating the feasibility of using these advanced Java features for Grid programming as part of our development of several application-specific Grid portals. A portal defines a commonly known access point to the application reachable via a Web browser. Many portal projects use the Java Commodity Grid, or CoG, Kit [11], allowing access to services provided by the Globus Toolkit (www.globus.org) in a way familiar to Java programmers. Thus, the Java CoG Kit is not a simple one-to-one mapping of the Globus API into Java; instead it uses features of the Java language not available in the original C implementation. For example, it includes both the OO programming model and the Java event model.

Another important Java advantage is a graphical user interface for integrating graphical components into Grid-based applications. Our experience with collaborators from various scientific disciplines, including structural biology and climatology, has shown that development of graphical components hiding the complexity of the Grid lets the scientist concentrate on the science, instead of on the Grid's inherent complexity [12] (see Figure 3).

Besides simplifying program development, Java eases development and installation of the client software accessesing the Grid. While trivial for a Java software engineer to install client libraries of the Java CoG Kit on a computer, installation of client software written in other programming languages or frameworks, including C and C++, is much more involved due to differences in compilers and operating systems. Another advantage of using the bytecode-compiled archives is they can also be installed on any operating system supporting Java, including Windows. Using the Java framework allows development of drag-and-drop components enabling information exchange between the desktop and the running Grid application during a program instantiation. Thus, it is possible to integrate Grid services seamlessly into the Windows and the Unix desktops.

Using a commodity technology like Java as the basis for future Grid-based program development represents yet another advantage. The committed support for Java by major vendors in e-commerce allows scientists to exploit a greater range of computer technology—from supercomputers to state-of-the-art commodity devices like cell phones, PDAs, and Java-enabled sensors—all within a Grid-based problem-solving environment.

Conclusion

Advanced applications like those in science and engineering can require multiple communication abstractions, ranging from message passing to remote method invocation and component frameworks. We've sought to show how a mixture of existing Java constructs and innovative implementation techniques allow Grid-based software engineers and Java programmers to use these communication abstractions efficiently within a single integrated Java framework. The result is a programming approach that appears particularly advantageous for dynamic and heterogeneous Grid environments. **C**

Components and JavaBeans

A software component is a unit of composition. Its design is restricted by a contract requiring a specific set of interfaces. Software components can be reused, interchanged, and deployed independently and are subject to composition into bigger systems by third parties. For instance, builder tools can extract design information, determine a component's capabilities, and reveal them conveniently to the software engineer to encourage reuse.

JavaBeans. A Bean is a reusable software component that can be visually manipulated through builder tools (see java.sun.com/products/javabeans/). JavaBeans include:

Dynamic type inspection (often called "introspection" in Java terminology). Allows other programs to analyze at runtime how the defined Bean works.

Customization. Allows users to alter Bean appearance and behavior.

Events. Allows Beans to fire events and provide (through dynamic type inspection) information to builder tools concerning both the events they can fire and the events they can handle.

Properties. Allows Beans to be manipulated and customized programatically.

Persistence. Allows the state of a customized Bean to be saved and restored.

Beans are especially useful for computational scientists and application experts reusing components designed for Grid computing. **C**

References

1. Aridor, Y., Factor, M., Taperman, A., Eilam, T., and Shuster, A. A high-performance cluster JVM presenting a pure single system image. In *Proceedings of the ACM Java Grande Conference* (San Francisco, June 3–4). ACM Press, New York, 2000, 168–176.
2. Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B. Toward a common component architecture for high performance scientific computing. In *Proceedings of the 8th IEEE International Symposium on High-Performance Distributed Computing* (Redondo Beach, CA, Aug. 3–6). IEEE Press, 1999, 115–124.
3. Birrell, A. and Nelson, B. Implementing remote procedure calls. *ACM Transact. Comput. Syst. 2,* 1 (Feb. 1984), 39–59.
4. Boisvert, R., Moreira, J., Philippsen, M., and Pozo, R. Java and numerical computing. *IEEE Comput. Sci. Engin. 3,* 2 (Mar./Apr. 2001), 22–28.
5. Carpenter, B., Getov, V., Judd, G., Skjellum, A., and Fox, G. MPJ: MPI-like message passing for Java. *Concurrency: Pract. Exper. 12,* 11 (Sept. 2000), 1,019–1,038.
6. Foster, I. and Kesselman, C., Eds. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, Orlando, FL, 1999.
7. Foster, I., Kesselman, C., and Tuecke, S. The anatomy of the Grid: Enabling scalable virtual organizations. *Int. J. Supercomput. Applic. 15,* 3 (Fall 2001); see www.globus.org/research/papers.html.
8. Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, MA, 1994.
9. Maassen, J., van Nieuwport, R., Veldema, R., Bal, H., and Plaat, A. An efficient implementation of Java's remote method invocation. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Atlanta, May 4–6, 1999), 173–182.
10. Philippsen, M., Haumacher, B., and Nester, C. More efficient serialization and RMI for Java. *Concurrency: Pract. Exper. 12,* 7 (May 2000), 495–518; see wwwipd.ira.uka.de/JavaParty/.
11. von Laszewski, G., Foster, I., Gawor, J., and Lane, P. A Java commodity Grid kit. *Concurrency and Comput: Pract. Exper. 13,* 8–9 (July 2001); see www.globus.org/cog/documentation/papers/.
12. von Laszewski, G., Westbrook, M., Barnes, C., Foster, I., and Westbrook, E. Using computational Grid capabilities to enhance the capability of an X-ray source for structural biology. *Cluster Comput. 3,* 3 (third quarter 2000), 187–199.

Vladimir Getov (V.S.Getov@wmin.ac.uk) is a professor of distributed and high-performance computing in the School of Computer Science, University of Westminster in London, U.K., and a visiting scientist in the Computer and Computational Sciences Division of Los Alamos National Laboratory, New Mexico.
Gregor von Laszewski (gregor@mcs.anl.gov) is an assistant scientist in the Mathematics and Computer Science Division of Argonne National Laboratory and a fellow in the Computation Institute of the University of Chicago, IL.
Michael Philippsen (michael@philippsen.com) is a senior researcher in the Computer Science Department of the University of Karlsruhe, Germany.
Ian Foster (foster@mcs.anl.gov) is a senior scientist and associate director in the Mathematics and Computer Science Division of Argonne National Laboratory and a professor of computer science in the University of Chicago, IL.

Research conducted by Vladimir Getov is supported in part by the Higher Education Funding Council for England (U.K.) and by Los Alamos National Laboratory (U.S.) under the Exchange Visitors program. The research conducted by Gregor von Laszewski and Ian Foster is supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by DARPA under contract N66001-96-C-8523; by the National Science Foundation; and by the NASA Information Power Grid project.