Multi-Paradigm Communications in Java for Grid Computing

Vladimir Getov[1,2], Gregor von Laszewski[3], Michael Philippsen[4], Ian Foster[3]

[1]School of Computer Science, University of Westminster, London, UK
[2] Computer & Computational Sciences Division, Los Alamos National Laboratory, USA
[3] Math & Computer Science Division, Argonne National Laboratory, USA
[4] Computer Science Department, University of Karlsruhe, Germany
Correspondence to: vgetov@lanl.gov

The scientific community's need to solve increasingly challenging computational problems currently demands a more diverse range of high-end resources then ever before. Despite continuous and exponential improvements in clock rate, communication speed, chip density, and storage size, very often multiple resources are necessary to provide solutions to state-of-the-art problems. Examples of such high-end applications range from financial modeling and vehicle simulation to computational genetics and weather forecasting. Therefore, high performance computing practitioners have been using distributed and parallel systems for years. This professional community has become a major customer segment for research and development of advanced communication frameworks because such systems have to provide coordination and synchronization among participating resources.

Typically, the code that is responsible for the communication between multiple resources is implemented in different programming languages. As good engineering practice, software layers are used to implement communication paradigms that hide these differences. Traditionally, message-passing libraries have been used for applications that can exploit symmetric multi-processing, while remote procedure calls have been introduced in the early eighties for applications that need a client/server structure (Figure 1). For most scientists, both approaches in general have been limited by the computing facilities available within a single computer center. However, current high-end applications require resources that are much more diverse not only in terms of locations and distances but also in terms of functionality. A large number of geographically distributed resources for processing, communication, and storing of information across computing centers is required. The integration of such resources to a working environment is referred to as the computational Grid [6].

The distribution of applications over the Grid requires an additional framework that hides differences of the underlying distribution of resources—that is, that bridges between different operating systems, programming paradigms, and heterogeneous networks, but also is opaque and robust for the programmer to use. Many Grid applications also would benefit from three additional features: the use of message-passing libraries, fast remote method invocation (RMI), and component frameworks (see Figure 2). For example, structural biology studies using x-rays from advanced synchrotron radiation sources require enormous data rates and compute power, easily reaching the Gbit/s and the Tflop/s ranges, respectively [11]. Since the computational demand is so large, and since the community has significant experience with the message-passing paradigm and with remote

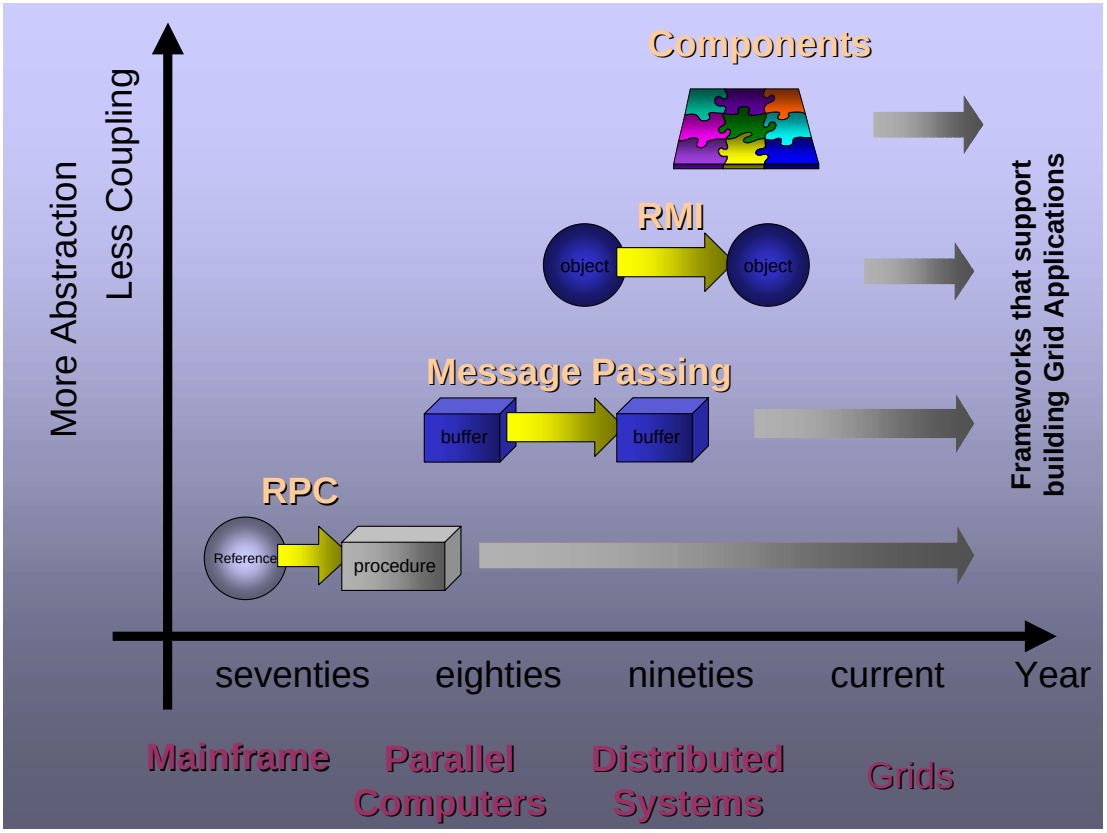

Figure 1: *Multiple communication frameworks are necessary to support programming the diverse infrastructure that Grids comprise.*

procedure calls, a *single, integrated environment* that provides such communication features for the emerging Grid applications is highly desirable. A component framework is also necessary to ease the software engineering problem of integration.

Java seems ideal for this multi-paradigm communications environment. Java's platform-independent bytecode can be executed securely on many platforms, making Java (in principle) an excellent basis for portable high-end applications that need the Grid. Inspired originally by coffee house jargon, the buzzword *Grande* has become commonplace in order to distinguish this new type of application when written in Java.[1] In addition, Java's performance on sequential codes, which is a strong prerequisite for the development of Grande applications, has increased substantially over the past years [3]. Furthermore, Java provides a sophisticated graphical user interface framework, as well as a paradigm to access state variables through remote objects; these features are of particular interest for remote steering of scientific instruments.

The rest of this article demonstrates the state-of-the-art in Java's integrative support of communication paradigms and the use of component frameworks. We present ways to use of message-passing libraries in a Java setting. We show, that Java's RMI does allow for high-performance applications on clusters, even if the standard implementation of RMI is slow. And we illustrate a Grid framework that allows Java applications to be efficiently distributed over the computational Grid.

Message Passing

The Java language has several built-in mechanisms that allow the parallelism inherent in a given program to be exploited. Threads and concurrency constructs are well suited for shared memory computers, but not large-scale distributed memory machines. For distributed applications, Java provides sockets and an RMI mechanism [12]. For the parallel computing world, often the former is too low-level and the latter is oriented too much towards client/server type applications and does not specifically support the symmetric model adopted by many parallel applications. Obviously, there is a gap within the set of programming models provided by Java, especially for parallel programming support on clusters of tightly coupled processing resources. A solution to this problem inevitably builds around the message-passing communication framework, which has been one of the most popular parallel programming paradigms since the 80s.

In general, the architecture of a message-passing system can follow two approaches – implicit and explicit. Solutions that adopt the implicit approach usually provide the end user with a single shared-memory system image, hiding the message-passing at a lower level of the system hierarchy. Thus, a programmer works within an environment often referred to as the virtual shared-memory programming model. Translating this approach to Java leads to the development of a cluster-aware Java virtual machine (JVM) that provides fully transparent and truly parallel multithreading programming environment [1]. This approach allows for substantial performance improvements using Java on clusters within the multithreading programming model. It also preserves full compatibility with the standard Java bytecode format. The price to be paid for these advantages, however, is a nonstandard JVM that introduces extra complexity and overheads to the Java run-time system.

[1] "Grande" is the prevalent designation for the term "large" or "big" in several languages. In the US (and the South-West in particular), the term "grande" has established itself as describing size within the coffee house scene.

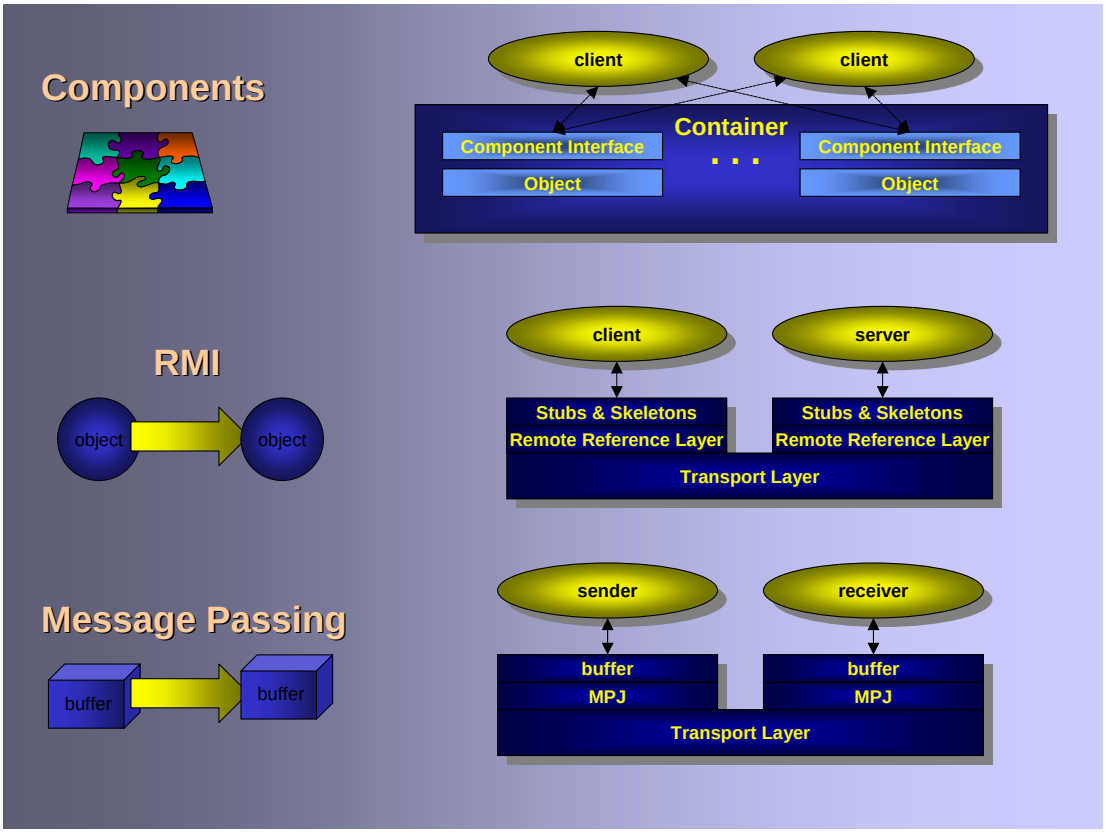

Figure 2: *A simplistic comparison between message-passing and RMI shows that they are conceptually related. The component framework allows clients to reuse components that are stored in a container and published through properties defined in an interface.*

By contrast with sockets and RMI, explicit message-passing directly supports symmetric communications including both point-to-point and collective operations such as broadcast, gather, all-to-all, and others, as defined by the Message Passing Interface (MPI) standard [5]. Programming with MPI is easy 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, non-portable practices. Indeed, the message-passing working group of the Java Grande forum (see sidebar in [3]) was formed in the Fall of 1998 as a response to the appearance of various explicit message-passing interfaces for Java. Some of these early "proof-of-concept" implementations have been available since 1997 with successful ports on clusters of workstations running Linux, Solaris, Windows NT, Irix, AIX, HP-UX, and MacOS, as well as, on parallel platforms such as the IBM SP-2 and SP-3, Sun E4000, SGI Origin-2000, Fujitsu AP3000, Hitachi SR2201 and others. An immediate goal is to discuss and agree on a common MPI-like application programming interface (API) for Message Passing in Java (MPJ) [4]. The purpose of the current phase of the effort is to provide an immediate, ad hoc standardization for common message-passing programs in Java, as well as, to provide a basis for conversion between C, C++, Fortran, and Java.

MPJ can be implemented in two different ways: as a wrapper to existing native MPI libraries or as a pure Java implementation. The first approach provides a quick solution, usually with only negligible time overhead introduced by the wrapper software. The use of native code, however, breaks the Java security model and does not allow work with applets—clear advantages for the pure Java approach. A direct MPJ implementation in Java is much slower, but the emplyment of more sophisticated design such as the use of native marshalling and advanced compilation technologies for Java can improve significantly the efficiency and make the two approaches comparable in terms of performance. For example, we have used the statically optimizing IBM High-Performance Compiler for Java (HPCJ), which generates native codes for the RS6000 architecture [3], to evaluate the performance of MPJ on an IBM SP-2 machine. The results show that when using such a compiler, the MPJ communication component is as fast as the native message-passing library (see Figure 3).

Closely modeled as it is on the MPI standards, the existing MPJ specification should be regarded as a first phase in a broader program to define a more Java-centric high performance message-passing environment. In the future, a detachment from legacy implementations involving Java on top of native methods will be emphasized. We should consider the possibility of layering the message-passing middleware over other standard transports and Java-compliant middleware (like CORBA). In a sense, the middleware developed at this level should offer a choice of emphasis between performance or generality, while always supporting portability. One can also note the opportunity to study and implement aspects of real-time and fault-aware message-passing programs in Java, which can be particularly useful in a Grid environment. Of course, a primary goal in the above mentioned, both current and future work, should be the aim to offer MPI-like services to Java programs in an upward compatible fashion. The purposes are twofold: performance and portability.

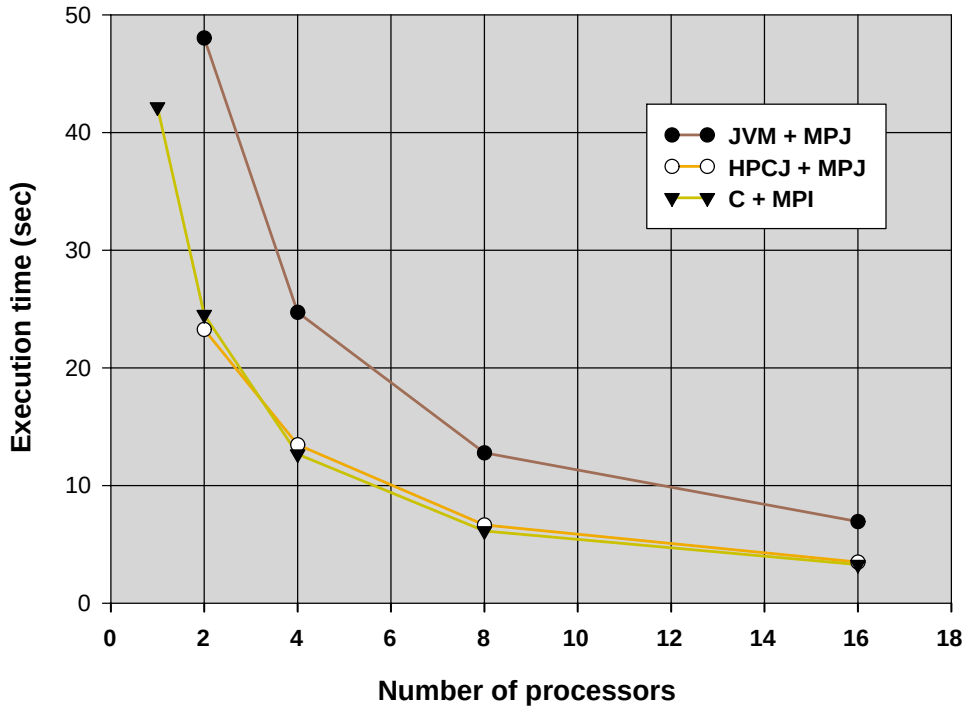

Figure 3: *Execution time for the Integer Sort kernel from the NAS Parallel Benchmarks on the IBM SP-2. The use of JVM and MPJ in this particular case is approximately 2.5 times slower than the same code written in C and using MPI. When using HPCJ and MPJ, however, this difference disappears with Java and MPJ performing as good as C and MPI for this experiment. This confirms that the extra overhead introduced by MPJ in comparison with MPI is negligible.*

Fast Remote Method Invocation

Remote invocations are a well-known concept. Remote procedure calls (RPC) have at least been around since 1981[9]. Today, CORBA uses RPCs to glue together code that is written in different languages. To implement a remote invocation, the method identifier and its arguments are encoded (marshalled) in a wire format that is understood both by the caller and the callee. The callee uses a proxy object to decode (unmarshall) that stream of bytes and to then perform the actual invocation. The results travel the other way round.

Although RMI inherits this basic design in general, it has distinguishing features that reach beyond the basic RPC. RMI is no longer defined by the common denominator of different programming languages. It is not meant to bridge between object-oriented versus procedural languages or to bridge between languages with different kinds of elementary types and structures. With RMI, a program running in one Java virtual machine (JVM) can invoke methods of other objects residing in different JVMs. The main advantages of RMI are that it is truly object-oriented, that it supports all the data types of a Java program, and that it is garbage collected. This allows for a clear separation between caller and callee. Development and maintenance of distributed systems becomes easier.

To understand the advantages, let us consider a remote invocation of a method `foo(Bar bar)`. In contrast to earlier RPCs, RMI is truly object-oriented and allows the caller to pass objects of any subclass of `Bar` to the callee. The object is encoded and shipped in a way so that the callee gets hold of the object (instead of a reference to an object). This encoding – Java uses the term "object serialization" – includes information on the class implementation, i.e., if the callee does not know the concrete class implementation of `bar` it can dynamically load it. Thus, not only the values of an object are shipped, but also the whole implementation is shipped. When the caller invokes an instance method on `bar`, say `bar.foobar`, the `foobar` code of the particular subclass of `Bar` will be executed at the side of the callee. One of the main advantages of object-oriented programming, the re-use of existing code with refined subclasses can thus be exploited for distributed code as well. Caller and callee can be developed separately as long as they agree on interfaces. From the software engineer's point of view, another advantage of RMI is the distributed garbage collection. Most practitioners agree that for sequential code, garbage collection helps save programmer time. The same is true for distributed code as well.

The novel features of RMI come at a cost. With a regular implementation of RMI on top of Ethernet, a remote method invocation takes milliseconds – concrete times depend on the number and the types of arguments. About a third of that time is needed for the RMI itself, a third for the serialization of the arguments (their transformation into a machine-independent byte representation), and a third for the data transfer (TCP/IP-Ethernet). While this kind of latency might be acceptable for coarse grained applications with little communication needs, it is too slow for high performance applications that run on low-latency networks, for example on a closely connected cluster of workstations.

Several projects are underway to improve the performance of RMI, for example the Manta project [7] and the JavaParty project (`http://wwwipd.ira.uka.de/JavaParty/`)[10]. In addition to simply improve the implementation for better speed, the main optimization ideas are first, to use pre-compiled marshalling routines instead of the ones that are generated at run-time by means of dynamic type introspection. A second idea is, to use only as little bandwidth for type encoding as necessary for the application. Only if objects are stored into persistent storage it is necessary to

save detailed type description. For communication purposes alone, a short type identifiers might be sufficient. Third, objects can be cashed to avoid re-transmission if their instance variables did not change between calls. Fourth, several layers have been removed in the RMI implementation to reduce loss at the interfaces between layers.

The JavaParty group reports that their optimized implementation of the RMI (called KaRMI) and of the underlying object serialization (called UKA serialization) achieves remote invocations within 80 μs on a cluster of DEC-Alpha computers connected by Myrinet. Figure 4 shows, that for benchmark programs 96% of the time can be saved, if the UKA serialization, the high-speed RMI (KaRMI), and the faster communication hardware is used. The Manta group reports even faster remote invocations, however their environment is no longer in pure Java but a native code compiled through C.

A Framework for Adaptive Grid Computing

So far we have discussed how well known communication paradigms can be made available and efficient in Java. Nevertheless, additional advances are needed to realize the full potential of emerging computational Grids [6]. The programming of such Grids raises a significant software engineering problem. We must deal with heterogeneous systems, diverse programming paradigms, and the needs of multiple users. Adaptive services must be developed that provide security, resource management, data access, instrumentation, policy, and accounting for applications, users, and resource providers.

Java provides some significant help to ease this software engineering problem. Due to its object-oriented nature, the ability to develop reusable software components, and the integrated packaging mechanism Java offers support for all phases of the lifecycle of a software engineering project from problem analysis and design, program development, program deployment, program instantiation, and maintenance.

Java's reusable software component architecture known as *JavaBeans* allows users to write self-contained, reusable software units. With commercially available visual application builder tools, software components can be composed into applets, applications, servlets, and composite components. Components can be moved, queried, or visually integrated with other components, enabling a new level of convenient Computer Aided Software Engineering (CASE) based programming in the Grid.

Component repositories or containers[8] (Figure 2) provide the chance to collectively work on similar tasks and to share the results with the community. Additionally, the Java framework includes a rich set of predefined Java APIs, libraries, and components including the access to databases and directories, network programming, and sophisticated interfaces to XML to list only a few.

We have evaluated the feasibility of using these advanced features of Java 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 that can be reached via a Web browser. All of these portal projects use a common toolkit, called the "Java Commodity Grid Toolkit" (Java CoG). This toolkit allows one to access many services of the Globus toolkit (http://www.globus.org) in a way familiar to Java programmers. Thus, Java CoG is not a simple one-to-one mapping of the Globus API into Java; it makes use of features of the Java language that are not available in the original C implementation. These features, for example, include the use of the object-oriented pro-

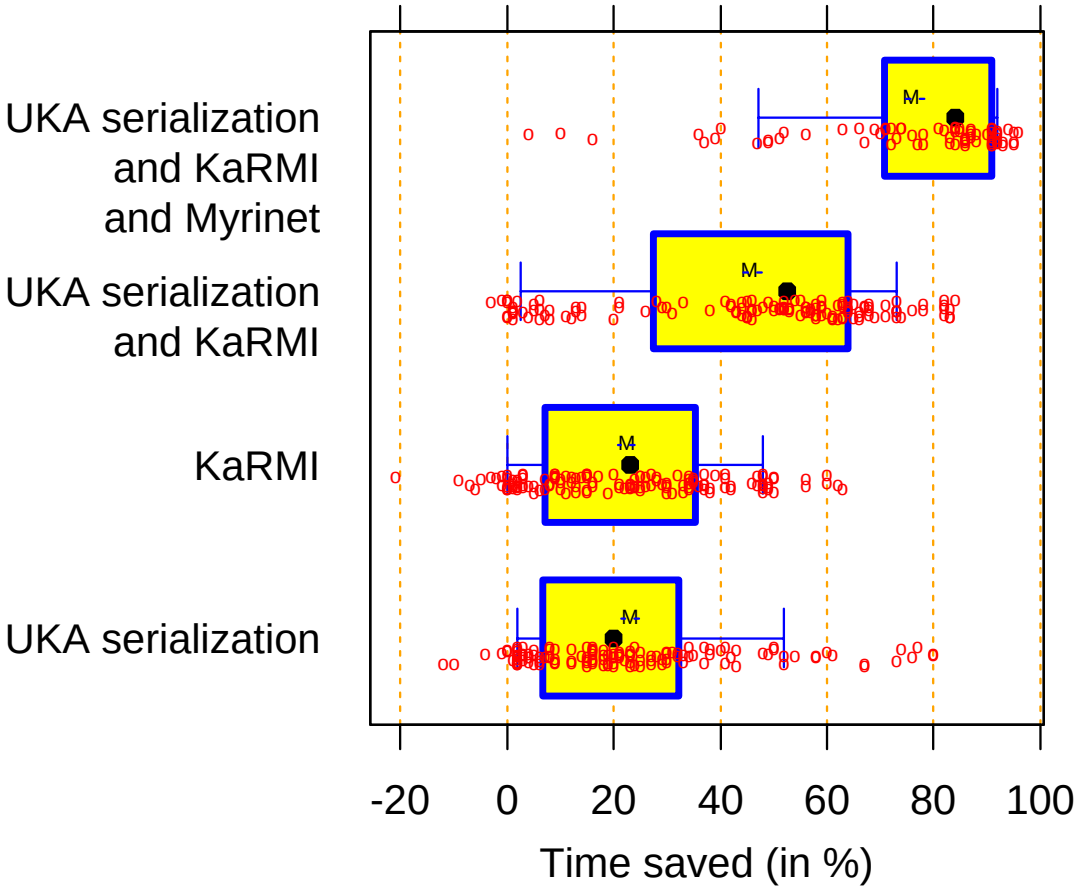

Figure 4: *The bottom three box plots each show 2·64 measured results for diverse benchmarks (64 points represent measurements on PCs connected by Ethernet, 64 stand for Dec-Alphas connected by FastEthernet). The first (bottom-most) box plot shows the run-time improvement that was achieved with regular RMI and the UKA serialization. The second box plot shows the improvement that KaRMI achieves when used with Java's regular serialization. The third box plot shows the combined effect. The top line demonstrates what happens, if Myrinet cards are used to connect the Alphas in addition to the UKA serialization and KaRMI (64 measured results)*

gramming model and the event model. Another important advantage of Java is the availability of a graphical user interface for integrating graphical components into Grid-based applications. Our extensive experience with collaborators from various scientific disciplines showed that the development of graphical components, hiding the complexity of the Grid, lets the scientist concentrate on the science, instead of dealing with the complex environment of a Grid.

Besides the advantages of Java during the program development, using Java eases the development and installation of client software that accesses the Grid. While it is trivial to install client libraries of the Java CoG Kit on a computer, the installation of client software written in other programming languages or frameworks such as C and C++ is much more involved, because of differences in compilers and operating systems. One of the biggest advantages in using the bytecode compiled archives is that they can also be installed on any operating system that supports Java, including the Windows operating system. Using the Java framework allows development of Drag&Drop components that enable 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 or the Unix desktop.

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

Conclusion

A variety of communication frameworks, such as message-passing and RMI are necessary to cover the range of communication needs for state-of-the-art applications. Abstractions provided as part of component frameworks are essential to build integrated grid applications. Adding the advantageous properties of the already existing Java framework leads us to the conclusion that Java can be helpful for the development of many Grid applications.

In addition, a natural framework for dynamically discovering new compute resources and establishing connections between running programs already exists in the Jini project [2], and one promising line for future research and development is into an integrated communication frameworks for grid computing based on message-passing, RMI, components, and Jini.

References

[1] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Shuster. A High Performance Cluster JVM Presenting a Pure Single System Image. In *ACM Java Grande Conference*, pages 168–176, San Francisco, June 3–4, 2000.

[2] K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification.* Addison Wesley, 1999.

[3] R. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and Numerical Computing. *CACM (submitted to the same special issue).*

[4] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12, 2000.

[5] J. Dongarra, S. Otto, M. Snir, and D. Walker. A Message Passing Standard for MPP and Workstations. *CACM*, 39(7):84–90, July 1996.

[6] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann, 1998.

[7] J. Maassen, R. van Nieuwport, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Proc. of the 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP*, pages 173–182, Atlanta, GA, May 1999.

[8] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly and Associates, March 2000.

[9] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, Pittsburg, PA, May 1981.

[10] M. Philippsen, B. Haumacher, and C. Nester. More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[11] G. v. Laszewski, M. Westbrook, I. Foster, E. Westbrook, and C. Barnes. Using Computational Grid Capabilities to Enhance the Ability of an X-Ray Source for Structural Biology. *IEEE Cluster Computing*, 3(3):187–199, 2000.

[12] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, July–September 1998.

Sidebar: Ten Reasons to Use Java in Grid Computing

Often the question arises, Why use Java for Grid computing? We summarize here the principal answers to this question.

1. The Language: Java as a programming language offers some features that are beneficial for large scale software engineering projects such as packages, object-oriented, single inheritance, garbage collection, and unified data formats. Since threads and concurrency control mechanisms are part of the language a possibility exists to directly express parallelism on the lowest user level in Java.

2. The Class Library: Java provides a wide variety of additional class libraries including essential functions, such as the availability to perform socket communication and access SSL, as needed for grid computation.

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

4. **The Deployment:** Java's bytecode allows for easy deployment of the software through web browsers and automatic installation facilities.

5. **The Portability:** Besides the unified data format Java's bytecode guaranties portability known under the flashy term "write-once-run-anywhere."

6. **The Maintenance:** Java contains an integrated documentation facility. Components that are written as JavaBeans can be integrated within commercially available IDEs (Interface Development Environments).

7. **The Performance:** It has been proven by well respected vendors that the performance of many Java applications can currently come close to that of C or FORTRAN.

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

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

10. **The Education:** Universities all over the World are teaching Java to their students.

Sidebar: Java as an Integration Tool for Building Grid Applications

In Figure 5 we show that Java provides many desired features for building Grid portals. This includes the access to distributed frameworks such as message passing or remote objects through RMI and CORBA. Java is supported within the infrastructure necessary to build portals. Web-, directory-, and database-servers can be accessed through Java class libraries. The development of portals is supported by the availability of Interface Development Tools that may already use XML for representing components. Java can be run on a variety of operating systems.

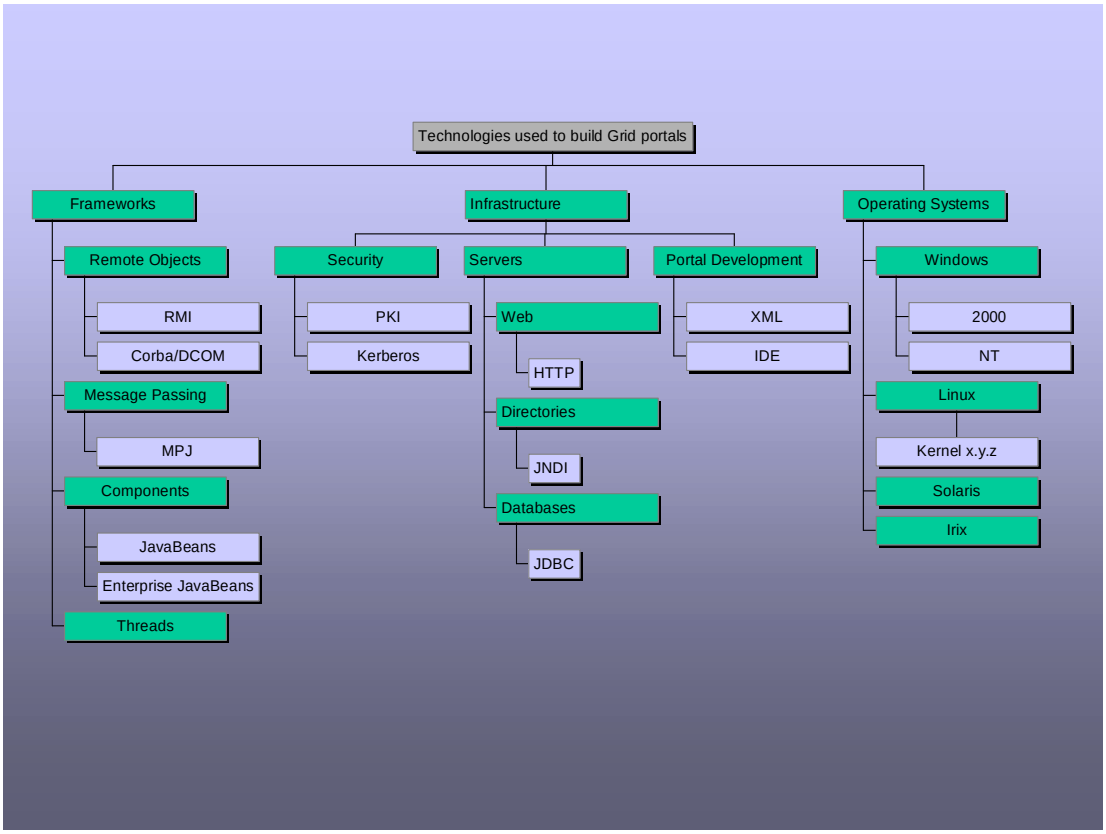

Figure 5: *Java allows to integrate many technologies that are used for portal development.*