Java Grande Forum Report: Making Java Work for High-End Computing

Java Grande Forum Panel
SC 1998
Orlando, Florida
13 November 1998

Java Grande Forum

Overview of Document

This document describes the Java Grande Forum and includes its initial deliverables.These are reports that convey a succinct set of recommendations from this forum to Sun Microsystems and other purveyors of Java™ technology that will enable Grande Applications to be developed with the Java programming language. The document consists of three major sections:

***Section 1: About the Java Grande Forum and Grande Applications*.** This section motivates the need for Grande Applications and provides a general overview of the Java Grande Forum, its activities, and the process. Grande Applications are of interest to an ever-growing community with applications to science, design, finance, and numerous applications where performance counts. This section intended for anyone who wants an overview of the Java Grande Forum and the remainder of the document but is not technical in nature.

Section 2: Numerics Working Group Recommendations. This section addresses the suitability of Java for numerical computing. In its initial assessment of Java, the working group has focused on five critical areas where improvements to the Java language are needed: (a) floating-point arithmetic, (b) complex arithmetic, (c) multidimensional arrays, (d) lightweight classes, and (e) operator overloading.

Section 3: Concurrency/Applications Working Group Recommendations. This section addresses the suitability of Java for high-end problem-solving applications by focusing on its concurrent and distributed features. The two critical areas where the working group agreed improvements to the Java language are needed include Remote Method Invocation and Object Serialization. The working group also identified several areas for continued investigation (community activities and research): (a) benchmarking JVM performance, (b) providing seamless (desktop) access to high-performance networked computing environments, and (c) providing support for MPI (Message Passing Interface) or MPI-like message passing facilities.

See “Summary of Working Group Recommendations” on page 6 for a detailed summary of each working group’s activities and recommendations.

1 About the Java Grande Forum and Grande Applications

The notion of a Grande Application is familiar to many researchers in academia and industry but the term itself is new. In short, a GA is any application, scientific or industrial, that requires a large number of computing resources, such as those found on the Internet, to solve one or more problems. Examples of Grande Applications are presented in this report as well as a discussion of why we believe Java technology has the greatest potential to support the development of Grande Applications.

The forum is motivated by the notion that Java could be the best possible Grande Application development environment and the extensive use of Java could greatly help the large scale computing and communication fields. However this opportunity can only be realized if important changes are made to Java in its libraries, language and perhaps Virtual Machine. The major goal of the forum is to clearly articulate the current problems with Java for Grande Applications and detail the requirements, analysis and suggestions for specific changes. It will also promote and energize widespread community activities investigating the use of Java for Grande Applications.

The forum is open and operates with a mix of small working groups and public dissemination and request for comments on its recommendations

The recommendations of the forum are intended primarily for those developing Java Grande base resources such as libraries and those directly influencing the direction of the Java language proper. (Presently, this implies Sun Microsystems or any standards body that may be formed.)

Mission and Goals

Java has potential to be a better environment for *Grande Application Development* than any previous languages such as Fortran and C++. The goal of the Java Grande Forum (hereafter, JGF) is to develop community consensus and recommendations for either changes to Java or establishment of standards (frameworks) for *Grande* libraries and services. These language changes or frameworks are designed to realize the *best ever* Grande programming environment.

The Java Grande Forum does not intend to be a standards body for the Java language per se. Rather, JGF intends to act in an advisory capacity to ensure those working on Grande Applications have a unified voice to address Java language design and implementation issues and communicate this input directly to Sun or a prospective Java standards group.

Grande Applications

This section addresses the questions of immediate interest: What is a Grande Application? What is an example of a Grande Application? Why are Grande Applications important? After this, we will discuss the relevance of Java.

Grande Applications are suddenly everybody's interest. The explosive growth of the number of computers connected to the Internet has led many researchers and practitioners alike to consider the possibility of harnessing the combined power of these computers and the network connecting them to solve more interesting problems. In the past, only a handful of computational scientists were interested in such an idea, working on the so-called grand challenge problems, which required much more computational and I/O power than found on the typical personal computer. Specialized computing resources, called parallel computers, seemingly were the only computers capable of solving such problems in a cost-effective manner.

The advent of the more powerful personal computers, faster networks, widespread connectivity, etc. has made it possible to solve such problems even more economically, simply by using one's own computer, the Internet, and other computers.

With this background, a **Grande Application** is therefore defined as an application of large-scale nature, potentially requiring any combination of computers, networks, I/O, and memory. Examples are:

- **Commercial:** Datamining, Financial Modeling, Oil Reservoir Simulation, Seismic Data Processing, Vehicle and Aircraft Simulation
- **Government:** Nuclear Stockpile Stewardship, Climate and Weather, Satellite Image Processing, Forces Modeling,
- **Academic:** Fundamental Physics (particles, relativity, cosmology), Biochemistry, Environmental Engineering, Earthquake Prediction
- **Cryptography:** The recent DES-56 challenge presents an interesting Grande Application. See http://www.rsa.com.

You can also note several categorizations, which can be used to describe Grande Applications

- High Performance Network Computing
- Scientific and Engineering Computations
- Distributed Modeling and Simulation (as in DoD DMSO activities)
- Parallel and Distributed Computing
- Data Intensive Computing
- Communication and Computing Intensive Commercial and Academic Applications
- Computational Grids (e.g., Globus and Legion)

Java for Grande Applications

A question that naturally arises is:

Why should one use Java in Grande Applications?

The Java Grande Forum believes that, more than any other language technology introduced thus far, Java has the greatest potential to deliver an attractive productive programming environment spanning the very broad range of tasks needed by the Grande programmer. Java offers from a combination of its design features and the ready availability of excellent Java instructional material and development tools. The Java language is not perfect; however, it promises a number of breakthroughs that have eluded most technologies thus far. Specifically, Java has the potential to be written once and run anywhere. This means, from a consumer standpoint, that a Java program can be run on virtually any conceivable computer available on the market. While this could be argued for C, C++, and FORTRAN, true portability has not been achieved in these languages, save by *expert-level* programmers.

While JGF is specifically focused on the use of Java to develop Grande Applications, the forum is not concerned with the elimination of other useful frameworks and languages. On the contrary, JGF intends to promote the establishment of standards and frameworks to allow Java to use other industry and research services, such as Globus and Legion. These services already provide many facilities for taking advantage of heterogeneous resources for high-performance computing applications, despite having been implemented in languages other than Java.

Java Grande Forum Process and Membership

The forum has convened for a total of three working meetings with a core group of active participants. The output of these meetings will be a series of reports (this being the first in a series), which are reviewed in public forums and transmitted appropriately within the cognizant bodies within the Java and computational fields. The forum is open to any qualified member of academia, industry or government who is willing to play an active role in support of our mission.

For more information on the forum itself and to provide comments, please visit http://www.javagrande.org or direct e-mail to George K. Thiruvathukal, Forum Secretary, at george.k.thiruvathukal@acm.org or Geoffrey C. Fox, Academic Coordinator, at gcf@npac.syr.edu

There are two relevant mailing lists to which one may subscribe by sending one of us e-mail indicating an interest in joining either or both of the lists:

- javagrandeforum@npac.syr.edu is a mailing list for coordinating communications and meetings among active members of the Java Grande Forum. This list is used to coordinate actual work and is not intended for those who are generally interested in the JGF.
- java-for-cse@npac.syr.edu is an open-ended interest for those keeping informed of Java Grande activities and the use of Java in computer/computational science and engineering applications. This list is intended primarily for those who will not participate actively in the Java Grande Forum but wish to keep informed.

Meeting Summaries

First Java Grande Forum Meeting at Java ‘98

The inaugural meeting of the Java Grande Forum was held at Java ‘98 on February 28 and March 1, 1998, in Palo Alto, California. At this time an informal process was defined and two discussion groups were formed to address issues of numerics and concurrency issues that need to be addressed in Java. It was agreed to hold a follow-up meeting to allow the two working groups to explore issues in greater detail and to formalize a charter for the Java Grande Forum.

Second Grande Forum Meeting

The Second Java Grande Forum meeting was held May 9-10, 1998, in Palo Alto, California. It was sponsored by Sun Microsystems (Siamak Hassanzadeh) and coordinated by Geoffrey Fox and George K. Thiruvathukal as secretary.

The meeting started with technology updates from Sun (their Hotspot optimizing compiler and the Java Native code Interface JNI) and IBM (Marc Snir on the performance of Java in scientific computing). Then we pursued the classic mix of parallel and plenary sessions using two working groups.

- **Numerics and Libraries** led by Roldan Pozo and Ron Boisvert (both of NIST)

- **Applications and Concurrency** led by Dennis Gannon (Indiana) and Denis Caromel (INRIA Nice Sophia Antipolis)

Both groups made good progress and their preliminary reports were made available by early June. After appropriate review of our suggestions by key scientific computing communities, we expect to submit a set of near term action items to JavaSoft. Our proposal to JavaSoft will also discuss the Java VM and RMI enhancements needed for scaling Java to large-scale concurrent and distributed applications.

We divided our action items into three categories

1. Proposals to JavaSoft as discussed above. These were further divided into either essential or desirable.
2. Community activities to produce infrastructure and standards
3. Community research which will clarify the value of new activities of type 1) and 2)

Action items of type 2) include standard interfaces and reference implementations for Java libraries of math functions, matrix algebra, signal processing etc. We also proposed a Java Grande Application benchmark suite with kernels and more substantial applications. There was significant discussion of the importance of a "Java Framework for computing" -- a set of interfaces to support seamless computing or the ability to run a given job on any one of many different computers with a single client interface. A typical community research activity is the study of the scaling of the Java Virtual Machine to large applications or understanding the trade-offs between Java thread and distributed VM forms of parallelism.

Third Java Grande Forum Meeting

The Third Java Grande Forum meeting was held August 6-7, 1998, in Palo Alto. It was sponsored by Sun Microsystems (Siamak Hassanzadeh) and coordinated by Geoffrey Fox with George K. Thiruvathukal as secretary. This meeting had over 30 participants from academia, industry and government. This was intended to be our last meeting prior to SC 98, where the Java Grande Forum has a scheduled 3-hour panel session to include public presentations and debate as well as presentation of this report.

The meeting had interesting plenary presentations on a variety of topics. Jini (http://java.sun.com/products/jini) offers a general approach to distributed resource registration and discovery and seemed applicable to both hardware and software Grande components. Note that this base technology has a Linda-like distributed computing environment Java Spaces (http://java.sun.com/products/javaspaces) built (conceptually if not in practice) on top of it. Henry Sowizral described the extensive (but focussed) Java matrix capability in Java3D graphics framework. This was contrasted to the scientific matrix package Jama (http://math.nist.gov/javanumerics/jama/) developed by NIST and MathWorks, which was announced at our meeting in Cleve Moler's presentation. There was a lively presentation from Professor William Kahan (UC Berkeley) and Joseph D. Darcy on "How Java's Floating-Point Hurts Everyone Everywhere" (http://www.npac.syr.edu/javagrande/JAVAhurt.pdf) and why Sun's proposed floating point changes were flawed.

Tim Wilkinson described his company (http://www.transvirtual.com/) with its open Java VM Kaffe available freely. He also discussed optimization issues and noted that obviously he was not content to meet C++ performance but aimed at raw C and Fortran levels. Marvin Solomon from the Wisconsin Condor group described their distributed computing system and how it can both use Java as a development tool and support Java Grande Applications.

We continued to have two major working groups with a crosscutting interest in benchmarks. In this respect note the new Java benchmark collection from NIST at http://math.nist.gov/scimark/.

The numerics working group reviewed the interim report and affirmed their basic positions on the issues of complex, efficient classes, operator overloading, and multidimensional arrays.

The second working group carefully reviewed issues and decided that their request to Sun should only address the issue of RMI performance where capabilities to add fast transport layers are needed. Areas such as the scaling and performance of the JavaVM needed further study. We also discussed "Seamless computing" and started a working group to study systems such as UNICORE, WebSubmit, Condor, Globus and Legion to extract the features of a "Java framework for Grande Computing". We also agreed to discuss the MPI Java binding while the collection of a set of "application" benchmarks was agreed.

Desktop Access to Remote Resources Meeting

The *First Workshop on Desktop Access to Remote Resources* was held October 8-9, 1998, at Argonne National Laboratory in conjunction with the Java Grande Forum Working Group on Concurrency/Applications. This meeting was spun off as a result of the seamless computing discussion that arose during the third meeting of the Java Grande Forum.

The term *remote resource* is defined to include:

- compute resources (supercomputers, networked supercomputers and metacomputers, networks of workstations, and workstations)
- data resources (databases, directories, data services)
- compute services (specialized problem solving and visualization environments)

This first meeting concentrated on issues related to compute resources. The goal of this meeting was to

- collect the most recent information about the status of current projects accessing remote resources
- derive a strategy on how remote access to resources can be supported
- bring the research community in this area together to initiate dialog and collaboration

In the first part of the meeting short project presentations, including Condor, Globus, UNICORE, Websubmit, Arcade, Webflow, Tango, DisCom2, Akenti, IGP, and others, provided an overview to initiate discussions and dialogue. These discussions were continued during working group sessions in the second half of the meeting. Three working groups were defined according to the following topics

- Interface: Defining the interface to desktop access, i.e. the metacomputing API.
- User requirements: Analyzing the user requirements to the desktop access.
- Security: Enabling a secure desktop access to remote resources.

The working groups tried to identify issues related to the design of an architecture, which makes a desktop access to remote resources possible. This included identifying a list of services to enable seamless desktop access to occur. The working groups identified four types of user interfaces, which are related to the *users* accessing a metacomputing *grid*.

- The *End User* who wishes to submit a single job, multiple instances of a job, access a collaboration or invoke a Grid based problem solving environment.
- The *System Administrator* who is responsible for the installation of grid tools and the addition of new resources into the metacomputing system.
- The *Developer of Applications* that takes advantage of advanced grid services.
- The *Owners* of the metacomputing resources that comprise the Grid.

The interface-working group identified the need for fundamental abstractions like tasks and jobs, resources, events, file names and object handles. It was determined that services such as resource services, accounting, notification and event services, transaction services, logging services, keep-alive services, collaboration services, and execution services have to be defined.

The user requirements group focused on identifying services that are needed by the users. The user requirements were ultimately included into the services identified by the interface group. A separation between the graphical user interfaces and the services to support the desktop access to remote resources were found to be important. Besides the development of component building tools such as Arcade, Gecco, and Webflow, the participants viewed the development of shell commands as very important.

The third group focused primarily on issues related to a *secure desktop access to remote resources*. The security group had a short second meeting two weeks after the workshop to discuss issues related to secure data exchange, access control, authentication, as well as the need for simple administration.

During the workshop, it was decided to produce a report to be distributed at SC 98 and available as a technical report from Argonne National Laboratory. The report contains, an executive summary, a summary of the working group results, as well as, a collection of one-page descriptions of related projects. We expect the report to evolve over time, so that a distribution via WWW seems appropriate.

In order to facilitate the exchange of ideas, a mailing list (datorr@mcs.anl.gov) has been established as well as a web site (http://www-fp.mcs.anl.gov/~gregor/datorr). The web page will contain future announcements of this group, and is intended as a collection of resources including the on-line proceedings of the workshop.

Two follow up meetings are already scheduled. The first meeting takes place as *Birds of the Feather Meeting at SC 98*, while a second meeting on the 28 and 29 of January 1999 will take place either at Argonne National Laboratory or at Sandia National Laboratory.

Summary of Working Group Recommendations

Numerics Working Group

The goal of the Numerics Working Group of the Java Grande Forum is to assess the suitability of Java for numerical computing, and to work towards community consensus on actions which can be taken to improve the language and its environment for applications with significant requirements for floating-point computation. In its initial assessment of Java, the working group has focused on five critical areas where improvements to the Java language are needed: (a) floating-point arithmetic, (b) complex arithmetic, (c) multidimensional arrays, (d) lightweight classes, and (e) operator overloading. Lightweight classes and operator overloading provide key components to proposed improvements for complex arithmetic and multidimensional arrays; in addition, they admit natural and efficient implementation of many alternate arithmetic systems. The working group's proposal for changes to Java's floating-point semantics balances the need for efficiency for high performance applications with the need for predictability which is a hallmark of Java's original design. This is in contrast to Sun's Proposal for the Extension of Java Floating-Point Semantics, issued in August, which completely sacrifices predictability, about which the working group issued a critical response to Sun. SciMark, a benchmark for numerical computing was developed and released to provide a measure of progress for Java compilers and runtime systems. Finally, the Numerics Working Group is working to catalyze efforts to develop standard APIs for numerical computing in Java. Detailed initial proposals for APIs for complex arithmetic, array operations, linear algebra and special functions are now being evaluated. Other APIs for interval arithmetic, Fourier transforms and multiple precision arithmetic are planned. Information about Numerics Working Group activities is maintained at the JavaNumerics web page at http://www.math.nist.gov/javanumerics.

Concurrency/Applications Working Group

The goal of the Concurrency and Applications Working Group of the Java Grande Forum is to assess the suitability of Java for parallel and distributed computing and so-called problem-solving environments (applications) and to work towards community consensus on actions which can be taken to improve the language and environment. The focus has been on the Java-specific frameworks designed to support concurrent and high-performance computing methodology. Although there is an emphasis on getting the best performance, these issues are of concern not only to scientists and technical types but enterprise computing as well. The working group was able to establish consensus on many issues pertaining to Java's support for remote procedure calls (Remote Method Invocation or RMI). A key area where improvements are needed is in Object Serialization, which is used in RMI but is very much a separate entity in the Java space. The key recommendations include (a) using a slim technique for encoding type information, (b) providing more efficient serialization of float and double data types, (c) enhancing the reflection mechanism to allow fewer calls to be made to obtain information about a class, and (d) general implementation issues. A second key area where improvements are needed is in Remote Method Invocation itself (issues not overlapping with Object Serialization). The key recommendations include (a) improved management of socket (or other) connections, (b) improved resource management (focus on TCP ports and thread usage), and (c) improved support for custom transports (SocketFactory alone will not do). A number of other miscellaneous suggestions for improvement were presented. There was general consensus that these miscellaneous issues are important to developing Grande Applications; however, further investigation is needed to work out the details. A number of other issues that do not necessarily reflect problem areas for Java (RMI and Object Serialization) but are generally regarded as useful to developing Grande Applications were also considered as candidates for *community activities*, including (a) benchmarking JVM performance, (b) seamless (desktop) access to high-performance computers and (c) the availability of MPI-like message passing facilities for programming networks of computers or parallel machines more naturally. Information about the Java Grande Concurrency and Applications Working Group activities can be found at http://www.javagrande.org and following the direct link to Working Groups.

2 Numerics Working Group Recommendations

If Java™ is to become the environment of choice for high-performance scientific applications, then it must provide performance comparable to what is achieved in currently used programming languages (C or Fortran). In addition, it must have language features and core libraries that enable the convenient expression of mathematical algorithms. The goal of the Numerics Working Group (JGNWG) of the Java Grande Forum is to assess the suitability of Java for numerical computation, and to work towards community consensus on actions which can be taken to overcome deficiencies of the language and its run-time environment.

The proposals we put forth operate under a number of constraints.

- Relatively small, but visible, changes to Java and JVM can be made. Upward compatibility should be maintained, as well as a worthwhile amount of backward compatibility.
- The proposals should support good execution speed on widely available microprocessors. However, while improved performance is important, predictability of results should be maintained. JGNWG proposals provide different levels of performance/predictability trade-off.

In this section, we present initial findings of the working group. These were developed in working sessions of the Java Grande Forum meetings held in May and August of 1998. Various versions of this report were made available for comment from Forum participants. In most cases there was overall agreement on major issues. In some cases disagreement remains on details of implementation; these are identified explicitly in this section. We expect that the overall report is interim in nature, and that remaining issues will be resolved after further discussion. We welcome input from the community at large. Please send questions or comments to javagrandeforum@npac.syr.edu. Further information on the activities of the Java Grande Forum can be found at http://www.javagrande.org. Information developed by the Numerics Working Group can be found at http://math.nist.gov/javanumerics/.

Critical Java Language and Java Virtual Machine Issues

We begin by outlining critical requirements for numerical computing that are not currently served well by Java's design and implementation. Unless these requirements are met it is unlikely that Java will have much impact on the numerical computation community. This would be detrimental to the entire Java enterprise by slowing the dissemination of high quality components for solving commonly occurring mathematical and statistical problems.

	Issue	**Requirement**
1	**Complex arithmetic**	Complex numbers are essential in the analysis and solution of mathematical problems in many areas of science and engineering. Thus, it is essential that the use of complex numbers be as convenient and efficient as the use of `float` and `double` numbers.
2	**Lightweight classes**	Implementation of alternative arithmetic systems, such as complex, interval, and multiple precision requires the support of new objects with value semantics. Compilers should be able to inline methods that operate on such objects and avoid the overheads of additional dereferencing. In particular, lightweight classes are critical for the implementation of complex arithmetic as described in issue 1.

3	**Operator overloading**	Usable implementation of complex arithmetic, as well as other alternative arithmetics such as interval and multiprecision, requires that code be as readable as those based only on `float` and `double`.
4	**Use of floating-point hardware**	The high efficiency necessary for large-scale numerical applications requires aggressive exploitation of the unique facilities of floating-point hardware. At the other extreme, some computations require very high predictability, even at the expense of performance. The majority of users lie somewhere in between: they want reasonably fast floating-point performance but do not want to be surprised when computed results unpredictably misbehave. Each of these constituencies must be addressed.
5	**Multidimensional arrays**	Multidimensional arrays are the most common data structure in scientific computing. Thus, operations on multidimensional arrays of elementary numerical types must be easily optimized. In addition, the layout of such arrays must be known to the algorithm developer in order to process array data in the most efficient way.

Elaborations of each underlying issue, along with proposed solutions are presented in the following section. In suggesting solutions, the working group has been careful to balance the needs of the numerical community with those of Java's wider audience. Although the proposed solutions require some additions to the current Java and JVM design, we have tried to avoid change, relying on compiler technology whenever feasible. This minimizes the changes that affect all Java platforms, and enables implementors to optimize for high numerical performance only in those environments where such an effort is warranted.

Discussion of Critical Issues

Issue 1: Complex arithmetic

Using complex numbers conveniently means that expressions on complex numbers must look and behave like expressions on `float` or `double` values. This is critical for code understanding and reuse. Efficient complex arithmetic operations are only a few times slower than their real counterparts; ideally, the speed of a complex computation should be limited by the speed of the underlying floating point arithmetic and not the speed of memory allocation or object copying.

Providing a straightforward complex class using existing Java object mechanisms fails to provide an acceptable solution.

- The object overhead of complex methods makes them unacceptably inefficient.
- The semantics of complex objects are different than those of `float` and `double`. For example, the = and == operators manipulate references rather than values. Such differences lead to many errors.
- Use of method calls for elementary arithmetic operations leads to inscrutable code, which is very tedious to write and debug. Users would simply stay away.

The second and third items also mean that code reuse is severely limited. In the LAPACK project, much of complex code is identical to its real counterparts and this greatly eased the generation and maintenance of the library. Such economies are much more difficult to obtain if the syntax and semantics of complex is significantly different than that of the primitive types.

An alternative that solves both the convenience and speed issues would be to add `complex` as a new primitive type in Java. However, this approach also has a number of drawbacks.

While `complex` could be added naturally to the language, adding support for a `complex` type in the JVM is more problematic. Either the JVM would have to directly support `complex`, or `complex` expressions and variables in a Java program would have to be mapped into existing JVM instructions in a predetermined way. Fundamental changes to the JVM should be avoided if the existing functionality is sufficient to implement a given feature. However, translating the `complex` type into a pair of `double` values (a real and an imaginary component) in the JVM presents a number of difficulties.

- *How are* `complex` *numbers passed into a method?*
  Since the desired behavior of `complex` is analogous to a primitive type, `complex` numbers should be passed by value. One way to accomplish that is to represent each `complex` parameter as a pair of `double` parameters. Unfortunately, this approach circumvents Java method type checking at the JVM level; it would not be possible in the bytecode to distinguish between a method that took a `complex` argument and a method with the same name that took a pair of `double` arguments. (It would be possible to mangle the names of methods with `complex` parameters, but then the mangled name might conflict with a different Java method.)
- *How are* `complex` *numbers returned from a method?*
  The JVM cannot directly return a pair of `doubles` from a method. The method could return a two-element `double` array or an object with two `double` fields. However, these approaches could introduce additional memory allocation and copying overhead.

Even if `complex` numbers can be accommodated with some variation of the above approach, this would only solve the problem for `complex`. Many other numeric types such as decimal, interval, and arbitrary precision have similar requirements. Thus, instead of merely providing a specialized solution for `complex`, a better approach is to make Java extensible enough to add new numeric types that can be operated on conveniently and efficiently. To meet this goal, Java needs two capabilities: lightweight classes and operator overloading.

Issue 2: Lightweight classes

Lightweight classes allow the creation of a new kind of Java object. The goal of lightweight classes is to allow the runtime use of what appears to be a C `struct`, that is:

- Lightweight objects have "value" semantics; the = operator performs a deep copy (instead of changing a pointer) and the == operator performs a deep comparison (instead of comparing pointer values).
- A variable of a lightweight class is never `null`; such variables always have a value.
- No dynamic dispatch overhead is needed for the methods of a lightweight class; these classes are always `final`. (This also removes the necessity to store a dispatch pointer for lightweight objects.)

At the Java level, a new class modifier can be introduced to indicate a class is a lightweight class. At the JVM level there are several implementation options:

1. Introduce what amounts to an explicit `struct` at the JVM level (a very large change), or
2. Translate lightweight classes in Java classes to normal JVM classes with the Java to JVM compiler enforcing restrictions that hide the fact that lightweight classes are implemented as regular Java classes. The back-end compiler should heavily optimize lightweight classes for speed and space (e.g. using *escape analysis* to allow stack allocation instead of heap allocation, see **Storage issues**).

Since requiring large JVM changes reduces the likelihood of acceptance, and due to its greater backward compatibility, Java Grande recommends the second approach to implementing lightweight classes.

Implementation implications. Complex numbers can be implemented as a lightweight class. By implementing `complex` as a lightweight class, type checking is preserved at both the Java and JVM level. Lightweight classes reuse Java's existing facility to create new types; the compiler is responsible for the tedious job of disguising what appears (to extant JVMs) to be a normal Java object as a lightweight object. To the programmer, lightweight classes appear to be outside of the Java class hierarchy rooted at `Object`. However, lightweight classes can be implemented in a way backward compatible with existing virtual machines. Additional class attributes could be included in a `class` file to indicate to new VMs that optimizations tailored to lightweight classes can be used. When compiling programs using lightweight classes, the Java compiler is responsible for enforcing the following restrictions:

- A lightweight object cannot be observed to have a `null` value. This implies the following.

  A lightweight object cannot be assigned `null` or compared to `null`. All such expressions are caught as compile-time errors.

  A compiler-generated non-overridable default constructor is used to initialize lightweight objects. The default constructor initializes the fields of the lightweight object to the default value for that type (zero of numeric types, `null` for reference types). The compiler inserts calls to the default constructor before any code that can access the object. These default constructors are not dependent on any external program state. For local variables, each call of the default constructor for a lightweight class is wrapped with a `catch` block that catches `OutOfMemoryError` and rethrows the exception as `StackOverflowError`. Although user code may try to recover from an `OutOfMemoryError` an attempt to recover from a `StackOverflowError` is unlikely. Failure to allocate memory for a local lightweight object variable corresponds to running out of stack space.
- A lightweight class cannot define a `finalizer` method. In Java, `finalizer` methods are run when an object is garbage collected. Lightweight objects are intended to have semantics similar to the semantics of primitive types. Therefore, lightweight classes do not need `finalizer` methods.
- A user-defined lightweight class constructor must not explicitly invoke `super`. In a constructor, calling `super` invokes the constructor of the superclass. Since lightweight objects are defined to be outside of the `Object` class hierarchy, it is not meaningful for a lightweight class constructor to call `super`.
- Lightweight objects cannot be cast to `Object` or any other reference type. Other types cannot be cast to the type of a lightweight class. Casts between primitive types construct a new value whereas casts between reference types reinterpret a pointer; no new value is constructed. However, user-defined conversions between lightweight classes are other types are permissible.
- Lightweight classes cannot implement interfaces.
- It is a compile-time error to apply the `instanceof` operator to an expression having the type of a lightweight class and it is a compile-time error to use `instanceof` to test for the type of a lightweight class.
- The JVM creates `Class` objects to represent lightweight classes.
- Lightweight classes can overload the assignment (=) and equality (==) operators (see Operator Overloading).

Virtual machines are encouraged to inline the methods of lightweight classes where appropriate.

To behave like primitive types, lightweight classes should be passed by value, that is, when given as an argument to a method or when returned from a method, a lightweight object is copied. C++'s copy constructor performs this task. However, references were added to C++ to avoid the overhead of copying small objects like `complex` [17]. Objects in Java are already passed by reference. Therefore, for performance reasons it may be acceptable (but somewhat contradictory semantically) to pass lightweight objects by reference.

Storage issues. Since lightweight classes are `final` and since references to lightweight objects are not `null`, there is no need to store a dispatch pointer at runtime.

Heap allocation is potentially more expensive than stack allocation. Additionally, stack-allocated objects may be cheaper to garbage collect than heap allocated ones. Therefore, it is preferable to allocate lightweight objects on the stack instead of the heap. Replacing heap allocation with stack allocation is an oft-discussed optimization. An object can be allocated on the stack if it can be determined that the object cannot be accessed outside of the lifetime of the scope that allocated it. *Escape analysis* [15] makes this determination. Recent work suggests escape analysis may be useful in Java [6][2]. The related problem of compile-time garbage collection is addressed by *region inference* [19] and its extensions [1].

Issue 3: Operator overloading

Operator overloading is necessary to allow user-defined numeric types, such as `complex`, to be used reasonably. Without it, many numeric codes would be extremely difficult to develop, understand and maintain. For example, codes using complex arithmetic class would look very different than similar code using real arithmetic, burdening library developers and users alike. A simple statement such as

```
a = b+c*d;
```

might be expressed as

```
a.assign(Complex.sum(b, Complex.product(c,d))
```

or

```
a.assign(b.plus(c.times(d)))
```

Faced with coding like this, a large portion of the scientific computing community would choose to avoid Java as being too unfriendly.

At a minimum, a useful subset of the existing operators must be overloadable. It is useful, but not a requirement of the working group, to allow novel operators to be overloaded as well. (Allowing novel operators to be overloaded does not have to introduce the language complications and programmer difficulties found in ML and Haskell, see [3].)

What operators can be overloaded. The arithmetic, comparison, assignment, and subscripting operators can be overloaded. Neither the `instanceof`, `new`, field access, nor method call operators can be overloaded. The && and `||` operators cannot be overloaded because they have different expression evaluation rules than all other operators.

If normal classes are also allowed to use operator overloading, it may be convenient to have Pascal's `:=` as a supplemental assignment operator. Java's = can be used to indicate the current semantics of moving a pointer while `:=` can be used for a deep assignment (deep copy, by convention the current semantics of a class' `clone` method). If `:=` can be overloaded, it can be used for a copy operator appropriate for a given class. For example, even when performing a deep copy, an arbitrary precision arithmetic class may want to use a copy-on-write policy for the bits of the number to avoiding copying the (possibly large) data structure unnecessarily.

If `:=` is introduced for classes, `:=` should designate normal assignment on the existing primitive types. That way code using primitive types can more easily be converted to using a user-defined type instead. For example, if roundoff problems on `double` numbers were suspected of causing loss of accuracy problems, it would be convenient to replace `double` with a floating-point type with more precision to see if the roundoff problems abate. With sufficiently powerful operator overloading, potentially only the variable declarations would need to be changed.

How to name methods corresponding to overloaded operators. The strings making up operators, "+", "+=", etc., are outside of the set of strings Java allows to form an *Identifier*. Therefore, to add operator overloading to Java, some technique must be used to allow operator methods to be declared. Either the text of the operator can be included in the method declaration (as with C++'s `operator+` syntax for an addition operator) or there can be an implicit mapping from textual names to operators (as with Sather's [16] mapping of "`plus`" to +).

If the latter implicit mapping is used, it is important to have a separate name for each target operator. This avoids the Sather problem where `a < b` is defined as `!(a >= b)`. Sather's scheme is problematical for IEEE style numbers since `(NaN < b)` and `(NaN >= b)` are both false. To overload operators such as +=, the corresponding name should be `plusAssign` instead of `plusEquals`. This names the operator according to what it does instead of what characters constitute the text of operator. In general, allowing each operator to be separately overloaded provides the flexibility to model mathematical entities other than traditional fields.

How to resolve overloaded methods. For normal Java classes, operator overloading can easily be mapped into method dispatch. For example, the compiler can translate `a + b` into

```
a.plus(b)
```

or

```
a.op+(b)
```

depending on how operator methods are named.

However, this level of support is not sufficient for more general uses of operator overloading. For example, this technique does not work if the type of a is a primitive type like `float` or `double` since these types do not have corresponding classes. Clearly, from an orthogonality and usability perspective, the programmer wants to be able to write `double + complex` as well as `complex + double`. Therefore, in addition to letting operators be instance methods (methods dispatched from an object), operators must also be `static` methods (methods not dispatched from an object). However, using `static` operator methods for regular Java classes presents name resolution problems.

In regular Java code, if a `static` method is used outside the defining class the class name (and possibly the package name) must be prefixed to the method call. For example, instead of writing

```
sin(x)
```

even if an `import` statement is used the Java programmer must write

```
Math.sin(x)
```

to allow the Java compiler to properly determine the location of the `sin` method. Using class name prefixes for operators would ruin the readability of operator overloading. However, since lightweight classes are `final`, the problems of using `static` methods can be circumvented.

Since lightweight classes are `final`, all method calls can be resolved at compile time; there need not be any dynamic dispatch at runtime. Therefore, even if used outside of the defining class, `static` operators on lightweight objects do not need to be prefixed with the class name; the compiler can use a rule of looking for operator methods defined in the lightweight class of the left hand operand.

Additionally, using `static` operators allows for better software engineering. If `static` operator methods can be located from either the class of the left-hand operand or the class of the right hand operand, new lightweight classes can be made to interact with existing lightweight classes. Otherwise, for symmetric treatment of `a + b` the classes of a and b must be written concurrently.

Issue 4 : Use of floating-point hardware

Recently, Sun released for public comment a *Proposal for Extension of Java™ Floating Point Semantics, Revision 1* [18] (abbreviated in this document as PEJFPS). PEJFPS is primarily targeted at improving Java's floating-point performance on the x86 line of processors. (No explicit license is granted (yet) to use the fused mac (multiply-accumulate) instruction, which would benefit users of the PowerPC, among other architectures.)

Assiduously implementing Java's current strict floating point semantics on the x86 using previously published techniques is very expensive, potentially more than an order of magnitude slower than slightly different semantics [7]. A less expensive technique developed recently [8] will be discussed later. PEJFPS grants partial access to the `double extended` floating point format found on the x86 in order to overcome the speed problem. However, the reckless license to use or not to use `double extended` granted by PEJFPS destroys Java's predictability (see recent submissions to the numeric-interest mailing list, http://www.validgh.com/java/).

Java has been billed as providing "write once, run anywhere" program development. For both theoretical and practical reasons, Java programs are not nearly so portable nor reproducible as programmers would naively expect. However, by exercising discipline (using single threaded code, using default `finalize` methods), it is far easier to produce a Java program whose behavior can be predicted than to produce an analogous C or C++ program with the same property. Dropping Java's predictability would be a significant loss. Therefore, the JGNWG recommends that PEJFPS not be incorporated into Java. Instead, JGNWG presents a counter-proposal that works within similar constraints as PEJFPS but maintains the predictability of the language and addresses additional numeric programming needs omitted from PEJFPS.

What is the problem on the x86? x86 processors most naturally operate on 80-bit `double extended` floating-point values. A precision control word can be set to make the processor round to single or double precision. However, even when rounding to a reduced precision, the floating point registers still use the full 15 exponent bits of the `double extended` format (instead of the 11 exponent bits for true `double` and 8 bits for true `float`). A store to memory is required to narrow the exponent as well. Since the register is not changed by the store, for further computation the stored value must be loaded back from memory. This memory traffic degrades Java's floating-point performance on the x86. Moreover, this technique suffers from a small discrepancy between operating on true `double` values and `double` values with increased exponent range. Values that would be subnormal `doubles` are not subnormal in `double` with extended exponent range. When such a number is stored to true `double`, it can be rounded twice, leading to a difference in the last bit, about 10^{-324}. Published techniques to remove this remaining minor discrepancy can lead to an order of magnitude slowdown, so Java VMs on the x86 generally set the precision control to `double` precision and allow double rounding on underflow, at variance with Java's specification [7].

The 10-fold potential performance degradation for exact floating point conformance on the x86 is largely a hypothetical concern since in practice VMs on the x86 use the store-reload technique. PEJFPS aims to eliminate the smaller 2-fold to 4-fold penalty from store-reload. PEJFPS would remove this speed penalty by allowing the x86's `double extended` registers to be used at full precision and range. However, PEJFPS would put too few constraints on when, where, and whether extended precision is used, leading to unpredictability.

There are two issues for exact reproducibility stemming from the x86's wider exponent range: maintaining the proper overflow threshold and preserving the proper gradual underflow behavior. The store-reload technique solves the former problem but not the latter. Since additions and subtractions resulting in subnormal values are exact, the underflow threshold is properly preserved. Using the store-reload technique, double rounding on underflow can only occur for multiplication and division.

Recently, a refinement of the store-reload technique that eliminates the double rounding problem has been developed [8]. To avoid double rounding during multiplication, the new technique scales down one of the operands by $2^{(E_{max}\,\texttt{double extended} - E_{max}\,\texttt{double})}$ where E_{max} is the largest exponent for a given floating point format. After this scaling, all operations that would result in subnormals in true `double` also result in subnormals in `double` with extended exponent range. This result is then rescaled back up by the same quantity; normal results are unaffected and subnormals are properly rounded once. A store of the product after being scaled enforces the overflow threshold.

The procedure for division is analogous; the dividend can be scaled down or the divisor can be scaled up. In both cases, the resulting quotient is rescaled up to the proper magnitude.

This new technique has many advantages over previous approaches to making the x86 exactly round to true `double`:

- The new technique is only marginally more expensive than the currently used store-reload method. Therefore, exact emulation of true `double` only entails a factor of 2 to 4 slowdown instead of a factor of 10.
- No special testing is needed to handle ±`0.0`, infinities, and NaN.
- Since the scalings up and down are exact, the proper IEEE sticky flags are set.
- Also due to the exact scalings, the technique works under dynamic rounding modes.

The JGNWG strongly encourages JVM writers for the x86 to adopt this new technique.

What capabilities are needed? Different numeric applications have different needs. Some, like certain implementations of higher precision arithmetic using standard floating point formats, depend on strict floating-point semantics and could easily break if "optimized." Other calculations, such as dot product and matrix multiply, are relatively insensitive to aggressive optimization; meaningful answers result even when blocking and other answer-changing optimizations are applied. The vendor-supplied BLAS are heavily optimized for a given architecture; vendors would not spend considerable resources creating optimized BLAS, sometimes included hand-coded assembly, if there were not demand for such faster programs. The needs of the typical Java programmer fall somewhere between these extremes; there is a desire for floating point computation that is not unnecessarily slow, but the programmer doesn't want to be surprised when his computed results misbehave unpredictably.

Since Java is aimed at a wide audience, the JGNWG proposal changes Java's default floating point semantics to allow somewhat better performance on the x86 and PowerPC. However, for most programs on most inputs the change in numerical results will not be noticed. Like PEJFPS, the JGNWG proposal adds a "strict floating-point" declaration to indicate current Java semantics must be used. JGNWG also includes a second declaration to allow optimizers to rearrange certain floating-point operations as if they were associative. Associativity enables many useful optimizations, including aggressive code scheduling and blocking.

PEJFPS would mingle increased speed and increased precision. In PEJFPS "`widefp`", which allows use of float extended and double extended formats, presumably yields code that runs faster and may incidentally use increased precision and range. Although it my have been intended only for the sake of fast register spilling, PEJFPS would allow almost arbitrary truncation of results from extended to base formats. In any case, the programmer is faced with an unpredictable program, leading to the resurrection of bugs from earlier systems, like the Sun III (see the recent comp.compilers thread "inlining + optimization = nuisance bugs" for a contemporary example). JGNWG's proposal does not mix speed and precision, rather, as a concession to the x86, JGNWG allows extended exponent range in some circumstances.

Some architectures, such as the PowerPC, include a fused mac instruction that multiplies two numbers exactly and then adds a third number, with a single rounding error at the end. Machines with this instruction run faster when it is used. Current Java semantics prohibit fused macs from being used.

There are three degrees of fused mac usage to support:

1. do not use fused macs at all,
2. use fused macs if they are fast (i.e. if there is hardware support), and
3. used fused mac even if it requires (slow) software simulation.

Fused mac must be forbidden in some sensitive calculations. For example, using fused mac recklessly can also lead to inappropriately negative discriminants in quadratic formula calculations [12]. Using a fused mac if available would give more accurate and faster dot products and matrix multiplies. Some algorithms require a fused mac to work properly. Mandating a fused mac is necessary to simulate a fused mac capable machine on one that isn't.

Java Grande Counter Proposal. The JGNWG proposal is based upon an analysis that finds very few of the diverse floating-point semantics allowed by PEJFPS to be both useful and necessary. On the other hand, each of the few semantics of the JGNWG proposal is necessary because dropping one would either

- hobble the performance of a commercially important family of computers, or
- make multiplying large matrices unnecessarily slow.

The semantic contexts proposed by the JGNWG are as follows.

1. Java's present semantics. All floating-point values are true `float` and true `double` values.
2. The present Java semantics except that some subexpressions that would have over/underflow in option 1 remain representable; and if underflow is signaled then some underflowed values may be rounded twice instead of once, differing from option 1 by around 10^{-324}. These semantics are used by default on the x86 to ameliorate some of the performance implications of exactly implementing Java's present floating point semantics on that line of processors. (This can be accomplished by allowing the use of 15-bit exponents in the representation of `double` values on the JVM operand stack.)
3. Permission to use fused mac (multiply-accumulate) where available. This can be used with either of the above.
4. Permission to use associativity with any of the above granted by the code's author.

There are three kinds of floating point semantics in JGNWG, strict, default, and "associative." The following table illustrates what effect these have on current processors.

	Architecture		
Modifier	*x86*	*PowerPC*	*SPARC*
`strictfp`	Java 1.0 (no double rounding on underflow)	Java 1.0 (no fused mac)	Java 1.0
default (no modifier)	larger exponent range allowed	fused mac can be used	Java 1.0
`associativefp`	many optimizations allowed on all platforms		

Strict semantics are indicated by the new `strictfp` class and method modifier. Strict semantics are the current Java floating point semantics; fused mac is not permitted in strict code (unless the compiler can prove the same answer results). On the x86, using stores to restrict the exponent range can readily give exactly Java-conforming results for the `float` format. Using the new technique described above, exactly Java-conforming results for the `double` format can also be implemented at a tolerable cost.

Like PEJFPS, JGNWG proposes to modify the default Java floating point semantics, i.e. those used in the absence of a `strictfp` or `associativefp` declaration.

The `strictfp` and `associativefp` declarations are mutually exclusive. In general, default semantics allow increased exponent range of anonymous expressions (on the x86) or use of fused mac (on the PowerPC). In default mode on the x86, anonymous `float` and `double` values created during expression evaluation are allowed to use the larger exponent range of `double extended`. If the extended exponent range is used in code with default semantics, it must be consistently used for all anonymous values (this implies that anonymous values spilled to memory must be spilled as 80 bit quantities to preserve the exponent values). All explicit stores must be respected and only true `double` and true `float` can be stored into programmer-declared variables.

System properties indicate whether fused mac and extended exponent range are used by a particular VM. It is always permissible to implement the default semantics as the "strict" Java 1.0 semantics. Therefore, on a SPARC there is no difference between the two kinds of floating point semantics.

It is possible that a program intended for strict semantics will fail under the non-strict JGNWG default semantics. To ease detection of such cases, JVMs should provide a runtime flag to force default semantics to be implemented as strict.

Finally, JGNWG also includes an `associativefp` declaration to allow optimizers to rearrange floating-point operations as if they were associative if the operations would be associative in the absence of over/underflow and roundoff. Associativity is an important property for optimization and parallelization of numerical codes that may change the numerical outcome of a computation.

On machines with fused mac instructions, chained multiply and add/subtract operations in the source code can be fused at runtime in default mode. Expressions are fused preferring fusing a product that is the right hand argument to an addition over the left hand argument; for example, the expression

```
a*b + c*d
```

is treated as `fmac(a, b, c*d)` and not `fmac(c, d, a*b)`. Such fusing operations must occur if fused mac is in use; this prevents programmer surprises. Otherwise, optimizers could potentially fuse unexpected expressions or prevent an expression from being fused (e.g., common subexpression elimination reusing a product that prevents a fused mac). Thearguments to fused mac must be evaluated in left to right order.

Programmers can require fused mac to be used by an explicit method call to a new method `Math.fmac`.

When fused mac is being used, the programmer can explicitly store each intermediate result to locally implement strict semantics.

Unresolved issues regarding additional optimizations. There is agreement within the working group that, at a minimum, allowing the use of associativity should be permitted as described above. Disagreement remains as to whether compilers should be allowed to employ additional types of optimizations that can, in some cases, cause incorrect results. Under current Java semantics, for example, common transformations such as `0*x` to `0` (wrong if `x` is NaN) or `x*4-x*3` to `x` (requires distributivity) are disallowed.

Some have argued strongly in favor of allowing wide latitude for optimizations, provided that the software developer and user concur on their use. From their point of view, as long as both parties have the option to disable potentially harmful optimizations, then compiler writers should not be barred from providing them. Gosling has proposed an `idealizedNumerics` [9] mode that would allow any transformation that would preserve results on the real number field, for example. Others argue that the predictability of the Java language is its key feature and that users are not served well if potentially harmful optimizations are admitted.

A related issue is whether the strict exception model of Java should be relaxed in `associativefp` mode to give additional freedom to the optimizer. `NullPointerException` and `IndexOutOfBoundsExceptions` would still be thrown and caught by the same handlers but the values in variables might not be in the same state as in an unoptimized version of the code. Others have argued that the strict exception model should be preserved so that programming styles that rely on exceptions would behave correctly. An example is the following somewhat unusual code for computing the dot product of two vectors:

```
double s = 0;
try {
  for (int i=0;true;i++) {
    s += a[i]*b[i];
  }
} catch (ArrayIndexOutOfBounds bounds) {
}
```

Finally, some issues regarding the interplay between `associativefp` and the fused multiply-add must be resolved.

These issues will require further discussion, and public comment is encouraged.

Code Examples for the x86. The dot product loop

```
static double dot(double a[], double b[])
{
  double s;

  for(i = 0; i &LT; a.length; i++)
    s += a[i] * b[i];

  return s;
}
```

can be compiled differently under strict and default semantics. In the examples that follow, loop tests and other integer calculations are omitted and are assumed to overlap with the floating-point calculations. Under strict Java 1.0 semantics on the x86, one compilation of the dot product loop is:

```
// x86 code for strict dot product loop
// rounding precision set to double
// assume scaling factors are in the register stack
push scaling factor
load a[i] and multiply with scaling factor
load b[i] and multiply with a[i] scaled down
multiply to rescale product
store product to restrict exponent
```

```
reload back restricted product
add product to s
store s to restrict exponent
load back restricted s
increment i
loop
```

As shown above, the product (`a[i] * b[i]`) and the sum (`s + a[i] * b[i]`) both need to be stored to memory and loaded back in. As shown below, under default semantics, only the sum needs to be written out to memory, halving the excess memory traffic and removing two multiplications.

```
// x86 code for default dot product loop
// rounding precision set to double
load a[i]
load b[i] and multiply with a[i]
// product does not need to be stored/reloaded and scaled
add product to s
store s to restrict exponent
reload s
increment i
loop
```

A larger number of anonymous values in an expression results in a greater reduction in the number of excess stores. A VM might also be able to use trap handlers to achieve faster average execution. For example, trapping on overflow could remove a load from the loop above, yielding

```
// x86 code for default dot product loop
// rounding precision set to double
// VM is using an overflow trap handler to remove a load
load a[i]
load b[i] and multiply with a[i]
add product to s
store s to restrict exponent    // dummy store
                                // reload if store overflows
// reload of s elided
increment i
loop
```

This trap handler is used by the compiler and not visible to the applications programmer. The functionality of this trap handler is simple; the trap handler just has to reload the stored value.

On the x86, if an expression (and all its subexpressions) neither overflows nor underflows for a given input, executing the default compilation of the expression will give the same result as the executing the strict compilation. As when using fused mac, explicitly storing each intermediate result can be used to implement strict semantics in a limited area. In default mode, both `float` and `double` expressions can use the extended exponent range. Method arguments and return values from methods must be strict `double` or `float` values to prevent programmers from being ambushed by greater exponent range they neither intended nor anticipated.

Cost of strictness. Using the new scaling technique, a matrix multiply with strict semantics can be a little more than twice as slow as a matrix multiply with default semantics. In both the loops below, an overflow trap handler is used to remove excess loads in the common case. The sum variables are already in the stack. For better instruction scheduling, two elements of the matrix product are calculated simultaneously:

```
// x86 code for fast matrix multiply using default semantics
// rounding precision set to double
// VM is using an overflow trap handler
// the loop has approximately an 8 or 9 cycle latency on a Pentium
load b[i]
dup b[i]
load a_k[i] and multiply with b[i]
swap top two stack elements
load a_k+1[i] and multiply with b[i]
swap top two stack elements
add with pop a_k[i] * b[i] to sum_k
add with pop a_k+1[i] * b[i] to sum_k+1
loop
```

The strict code is similar:

```
// x86 code for fast matrix multiply using strict semantics
// rounding precision set to double
// VM is using an overflow trap handler
// the loop has approximately a 19 cycle latency on a Pentium
// assume scaling constants are already in the register stack
put scaling constant on the top of the stack
load b[i] and multiply with scaling factor
dup b[i]^scaled down
load a_k[i] and multiply with b[i]^scaled down
swap top two stack elements
load a_k+1[i] and multiply with b[i]^scaled down
swap top two stack elements
rescale a_k[i] * b[i]^scaled down
swap
rescale a_k+1[i] * b[i]^scaled down
dummy store of a_k+1[i] * b[i]
swap
dummy store of a_k[i] * b[i]
add with pop a_k[i] * b[i] to sum_k
add with pop a_k+1[i] * b[i] to sum_k+1
store sum_k
swap
store sum_k+1
swap
loop
```

Scoping. Whatever new floating-point semantics are expressible in Java need to be expressible in the JVM too. PEJFPS uses spare bits in a method descriptor to indicate which kind of floating-point semantics a methods has; JGNWG can use the same approach. This provides method-level control of floating point semantics. This would be the coarsest level acceptable to the JGNWG.

It may also be convenient to have a finer granularity block-level control. While this is not critical to the JGNWG proposal, it should be considered. Such a declaration is easily added to Java, but it is not immediate apparent how to encode such information in the JVM. Java compilers can include extra attributes in a `class` file ([11] §4.7.1). These attributes can be used to support things such as improved debugging facilities. JVMs are required to ignore unknown attributes. Therefore, JGNWG could represent the different floating-point semantics of different portions of code using a table emitted as an extra `class` file attribute. Strict semantics is always a permissible policy under the JGNWG proposal; so, in this respect a JGNWG-style class file would be backward compatible with existing VMs.

Discussion. The JGNWG proposal allows improved hardware utilization over standard Java while preserving program predictability. Programmers can test system properties to determine the VM's behavior.

A reasonable question is that if Java Grande is opposed to PEJFPS due to its unpredictability, why does Java Grande's proposal also allow some unpredictability by default? JGNWG permits much less unpredictability than PEJFPS and JGNWG has fewer differences between strict and default semantics. For example, in JGNWG a floating-point feature must be used consistently; fused mac or extended exponent range cannot be used on one call to a method and not used on the next (something allowed by PEJFPS). On the x86, between allowing extended exponent range and allowing extra precision, allowing extended exponent range results in many fewer visible differences between strict code and default code.

The differences arising from extended exponent range on the x86 are visible only if the calculation would over/underflow on a machine like the SPARC. Over/underflow is comparatively rare in practice; therefore the Java Grande differences would be observed at most rarely. PEJFPS allows extended *precision* for intermediate results. Differences stemming from extended precision would almost always be visible. For example,

```
// implicit widefp under PEJFPS
// default semantics under Java Grande
static foo() {
   double one = 1.0;
   double three = 3.0;

   double a;
   double b[] = new double[1];

   a = one/three;
   b[0] = a;
}

                          // Results under different proposals
                          // JGNWG             PEJFPS
if(a == b[0]){...}        // always true       true or false?

if(a == (one/three)){...} // always true       true or false?
```

If `(one/three)` is calculated to extended precision and a is treated as an extended precision value, then a `== b[0]` will be false under PEJFPS since arrays are always stored in the base format (32 bit `float` or 64 bit `double`). If a is stored as `double` precision, a `== (one/three)` will be false if `(one/three)` is calculated to `double extended` precision. The Java Grande proposal would always return true for these cases. In short, on the x86 the cases where the JGNWG proposal allows differences between default and strict semantics are where overflow or underflow would occur; the cases where PEJFPS allows differences between default and strict semantics are (approximately) where an operation is inexact, as most are.

Additional Floating-point Types. The JGNWG proposal thus far does not provide any access to the `double extended` format found on the x86. Consistent access to this format is important to allow good hardware utilization and to ease writing numerical programs; having access to several more bits of precision than the input data often allows simpler (and sometimes faster) algorithms to be used. To access `double extended`, JGNWG proposes that Java include a third primitive floating point type called "`indigenous`." The `indigenous` floating-point type corresponds to the widest IEEE 754 floating point format that directly executes on the underlying processor. On the x86, `indigenous` corresponds to the `double extended` format; on most other processors, `indigenous` corresponds to the `double` format. (The `indigenous` type must be at least as wide as `double`.) For a particular VM, class variables indicate whether `indigenous` is implemented as `double` or `double extended`.

The `float` and `double` floating point types have hardware support. Adding a `double extended` type would require costly simulation on most architectures other than the x86. Having an `indigenous` type preserves the performance predictability of a program and keeps a close correspondence between floating point types in a language and floating point formats on a processor.

Implementing `indigenous` at the JVM level can be done either by adding new JVM instructions or (as an interim measure) using the operator overloading and lightweight classes described earlier.

Additional Floating-point Expression Evaluation Policies. To better utilize certain processors and to lessen the impact of rounding errors, it is useful for the programmer to conveniently be able to evaluate a floating-point expression in a wider format. For example, a programmer may want to evaluate an expression consisting of `float` variables in `double` precision. Pre-ANSI C used this floating-point expression evaluation policy exclusively.

JGNWG adds a new declaration, `anonymous` *FloatingPointType*, to control the expression evaluation policy:

1. `anonymous double` gives the original C expression evaluation; all `float` values are promoted to `double`.
2. `anonymous indigenous` promotes `float` and `double` values to `indigenous`. Using `anonymous indigenous` makes best use of the x86's double extended registers.
3. `anonymous float` specifies to use the existing Java expression evaluation policy.

No JVM changes are required to support `anonymous double`.

Issue 5 : Multidimensional arrays

Our goal is to provide Java with the functionality and performance associated with Fortran arrays.

Multidimensional arrays are *n*-dimensional rectangular collections of elements. An array is characterized by its *rank* (number of dimensions or axes), its elemental data *type* (all elements of an array are of the same type), and its *shape* (the extents along its axes).

Elements of an array are identified by their indices along each axis. Let a *k*-dimensional array *A* of elemental type *T* have extent n_j along its *j*-th axis, $i = 0,...,k-1$. Then, a valid index i_j along the *j*-th axis must be greater than or equal to zero and less than n_j. An attempt to reference an element $A[i_0,i_1,...,i_{k-1}]$ with any invalid index i_j causes an `ArrayIndexOutOfBoundsException` to be thrown.

Rank, type, and shape are immutable properties of an array. Immutable rank and type are important properties for effective code optimization using techniques developed for Fortran and C. Immutable shape is an important property for the optimization of run-time bounds checking according to recent techniques developed for Java [13][14].

We can understand the differences between multidimensional arrays and Java arrays of arrays through an example. Consider the following declaration and definition of a Java array of arrays.

```
double[][] A = new double[m][n];
```

At a first glance, this seems equivalent to a two-dimensional m×n array of doubles. However, nothing prevents this array from becoming jagged at a later point in the code:

```
for (int i=0; i<A.length; i++) {
   A[i] = new double[i+1];
}
```

(The array could also have been created jagged in the first place.) Also, two or more rows of the array A can be aliased to the same data:

```
for (int i=0; i<A.length-1; i+=2) {
   A[i] = A[i+1];
}
```

In general, given a reference of type `double[][]` the compiler has no information about shape and aliasing.

While arrays of arrays are important data structures for their flexibility, this flexibility comes at a performance cost. The potential aliasing between rows of an array of arrays forces compilers to generate additional stores and loads. The potential shape changing in arrays of arrays complicates bounds checking optimization, by requiring array privatization [13][14]. True rectangular, multidimensional arrays solve both these problems.

Proposal. We propose the development of standard Java classes that implement multidimensional rectangular arrays. These classes can be included as a subpackage in `java.lang.Math` or in their own package `java.lang.Array`. Standardizing the classes as part of Java is important for maximum compiler optimization. (In particular, it enables *semantic inlining* techniques [20].)

The rank and type of an array are defined by its class. That is, for each rank and type there is a different class. (This is necessary for traditional compiler optimizations, since Java does not support templates.) Supported types must include all of Java primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`), one or more complex types (at least the `Complex` class in this proposal), and `Object`. Supported ranks must include 0- (scalar), 1-, 2-, 3-, and possible 4- to 7-dimensional arrays. (Rank 7 is the current standard limit for Fortran.)

The shape of an array is specified at its creation, and it is immutable. An example might be

```
doubleArray2D A = new doubleArray2D(m,n);
```

which creates an m×n two-dimensional array of doubles. (Note that A itself can be changed to refer to another array, unless it is declared with the `final` modifier.) The shape parameters m and n must be `int` expressions. The value of any array shape parameter must be a nonnegative `int`. (That is, it must be greater than or equal to 0 and less than or equal to 2147483647.) If any of the shape parameters is negative, the constructor must throw a `NegativeArraySizeException` exception. If an array of the specified shape cannot be created because of memory limitations, then an `OutOfMemoryError` must be thrown.

The array classes should support the concept of *regular array sections*. A regular array section corresponds to a subarray of another array, which we call the *master array*. Each element of an array section corresponds to a unique element of its master array. Referencing one element of an array section (for reading or writing) has the effect of referencing the corresponding element of the master array. Regular array sections have the same type as, and rank less than or equal to, their master arrays. Regular array sections behave exactly like regular arrays for all operations, including sectioning. (In fact, there are no separate classes for array sections.)

A regular array section is defined by specifying a subset of the elements of its master array. For each axis of the master array, the specification can be (i) a single `int` index, or (ii) a *regular range* of `int` indices. A regular range is an arithmetic progression defined by a first element, a last element, and a stride. The rank of an array section is equal to the number of regular ranges in its definition. The shape is defined by the number of elements in those ranges. Note that indices for an axis of an array section must be greater than or equal to 0 and less than the extent of that axis. Array sections might be referenced as follows, for example.

```
doubleArray3D A = new doubleArray3D(m,n,p);
doubleArray2D B = A.section(new Range(0,m-1),new Range(0,n-1),k);
doubleArray1D C = B.section(new Range(0,m-1),j);
```

The first creates an $m \times n \times p$ three-dimensional array `A`. It then extracts an $m \times n$ two-dimensional section `B` corresponding to the *k*-th plane of `A`. It also extracts the *j*-th column of `B` into `C`, which also corresponds to the *j*-th column of the *k*-th plane of `A`.

The array classes should also support the concept of *irregular array sections*, although in a more limited fashion. An irregular array section is defined by specifying, for at least one of the axis of a master array, a generic set of indices. Operations on an irregular array section are limited to extracting and setting the values of its elements. (These correspond to the gather and scatter operations typical in vector codes.) For example, these might be specified as follows.

```
doubleArray2D A = new doubleArray2D(20,n);
int idx[] = { 1, 3, 5, 7, 11, 13, 17, 19 };
Index index = new Index(idx);

A.set(new Range(0,19),new Range(0,n-1),0.0);
A.set(Index,new Range(0,n-1),1.0);
```

This sets the value of the rows of `A` with prime numbers indices to `1` and the value of all other rows to `0`.

The array classes provide methods that implement Fortran-like functionality for arrays. In particular, the following operations must be provided:

- Get and set the values of an array element, array regular section, or array irregular section.
- Operations to query the rank and shape of an array.
- Operations to reshape and transpose an array.
- Elemental conversion functions (e.g., the equivalent of Fortran `REAL` and `AIMAG`, that convert complex arrays into `double` arrays).
- Elemental transcendental functions.
- Elemental relational functions (<, >, <=, >=, ==, !=).
- Array reduction functions (sum, minval, etc.).
- Array construction functions (merge, pack, spread, and unpack).
- Array reshape function.
- Array manipulation functions (shift, spread).
- Array location functions (maxloc, minloc).
- Array scatter and gather operations.
- Operations that correspond to array expressions (addition, multiplication, etc.)

Finally, it must be possible to cast Java arrays into multidimensional array objects and vice-versa.

Discussion. The array classes can be implemented with no changes in Java or JVM. However, it is essential that the get and set methods be implemented as efficiently as array indexing operations are in Fortran or in C. We expect that inlining will be used for this purpose, and that garbage collectors will treat rectangular arrays efficiently. Multidimensional arrays are extremely common in numerical computing, and hence we expect that efficient multidimensional array classes will be heavily used.

The inclusion of standard array classes in `java.lang.Math` or `java.lang.Array` does not require any change to the Java language. However, the use of explicit method invocation to effect all array operations will significantly decrease the readability of Java code and incur the wrath of users. The introduction of a simple notation for multidimensional arrays which maps to the standard array classes would make the use of such arrays much more natural. A multi-index notation, like `a[i,j]` to refer to such array elements would be ideal. This would allow statements like

```
a.set(i,j,b.get(i,j)+s*c.get(k,l));
```

to be more naturally expressed as

```
a[i,j] = b[i,j] + s*c[k,l];
```

The front-end compiler could disambiguate the expression according to the type of a. This requires changes in the Java language or fancier operator overloading mechanisms.

Additional facilities that would be very helpful, but are not strictly necessary are the following.

- Operator overloading applied to array arithmetic; e.g. $A = B + C$.
- Facilitating indexing operations by explicitly triplet notation, e.g. `a[f:l:s]` referring to the section of the one-dimensional array a from element `f` to element `l` in steps of `s`. This requires new syntax, or fancy overloading of the indexing.

Unresolved issues. While most working group members see benefit from the provision of multidimensional array classes, disagreement remains regarding a number of critical implementation issues.

The first issue regards the internal implementation of the array class. Library developers want the simplicity of Fortran arrays. They would like to see the requirement that multidimensional arrays be implemented using a one-dimensional native Java array with elements filled in, say, row-major order. The simplicity of such a scheme would benefit both optimizers and library developers, leading to better overall performance of codes. Library developers, for example, would know exactly how to traverse the array to maximize locality of reference. Some developers even want the ability to extract the internal 1D array, manipulate it, and then put it back.

Others feel that it is a mistake to expose the internal storage scheme. Compiler developers want the flexibility to decide on the layout based on their own analysis. They argue, for example, that packing everything in a 1D array would artificially limit the size of multidimensional arrays based on indexing considerations. One proposal would provide an additional array attribute called the *preferred access mode*. This could have values of (i) row major, for C-style coding, (ii) column major, for Fortran-style coding, and (iii) block major, for block-oriented recursive algorithms. The access mode would provide a hint to the compiler regarding the most likely access order for the array. The default would be row-major. Compilers could ignore the access mode attribute.

Library developers counter that it is unreasonable to expect them to provide optimized routines for each type of access mode (and for all combinations in case of array-array operations), and that the alternative of casting to and from differing access modes adds unacceptable overhead.

Another issue that must be considered further is the semantics of array sections. Array sections can be either aliases to the master array or copies extracted from the master array. In either case, the effect on the master array of writes to the section, and vice versa, must be carefully spelled out.

Finally, many array operations will result in an array of output. The best way of providing the output array is not clear. Providing a new array object as output is sometimes convenient, but may be inefficient. Providing a separate parameter to contain the output may be better.

Additional Concerns

The following additional problems were addressed by the Working Group.

1. **Alternative definition of the `java.lang.Math` library of transcendental functions.** The current operational definition is imprecise and suboptimal. (The functions are defined in terms of compatibility with a particular implementation, C's fdlibm source, interpreted using Java's strict semantics). Alternative definitions are

   precise rounding -- result is as if computed in infinite precision arithmetic, then rounded;

   within fixed bound of precise result; or

   improved operational definition.

   The first definition is very desirable if it can be achieved with acceptable performance overhead. The second weakens bitwise reproducibility. Note that current Java implementations are not in strict adherence to this aspect of the Java standard: most JVMs use their native C math library.

   As a compromise, we propose that fdlibm be translated to Java and that this particular implementation be mandated when Java's strict semantics are being enforced. Otherwise, a looser, implementation-dependent alternative which conform to the requirements of C9X (as fdlibm does) should be allowed.

2. **Access to additional IEEE floating-point features.** The high reliability required in certain sensitive floating-point computations requires the ability to manipulate IEEE floating-point flags. The sticky flags can also be used to create significantly faster robust linear algebra algorithms [5]. The Working Group proposes that standard methods to sense, save, clear and raise all IEEE floating-point flags be included in Java.

   Similarly, reliability concerns, as well as the ability to efficiently implement interval arithmetic, requires the ability to set rounding modes for floating-point operations. It is sufficient to provide methods to set (and get) the global rounding mode to accomplish these goals.

   In order for such features to be used reliably, compilers and JVMs must respect the semantics of the special methods used to implement these operations. In particular, the floating-point state must be saved and restored across thread context switches, and compiler and JVM optimizations must be modestly limited.

3. **Implementation of additional elementary functions and predicates.** The functions and predicates recommended in the IEEE 754 standards document, as well as others commonly available in C, should be provided in `java.lang.Math`. Equivalents of two of the ten IEEE 754 recommended functions are already available in the Java API (`IsInfinite` and `isNaN`). The following six should also be added.

`copySign(x,y)`	returns `x` with the sign of `y`.
`scalb(y,n)`	returns $y * 2^n$ for integers `n` without explicitly computing 2^n.

`nextAfter(x,y)`	returns the next representable neighbor of `x` in the direction towards `y`.
`unordered(x,y)`	Returns true is one of its arguments is unordered with respect to the other. This occurs when at least one is a NaN.
`fpClass(x)`	Returns an integer that indicates which of the nine "kinds" of IEEE floating-point numbers `x` is.
`logb(x)`	Returns the unbiased exponent of `x`.

The description of the functions given above is quite terse and ignores some subtleties for extreme arguments. (The same is true of the IEEE 754 document itself.) For a detailed discussion of how these functions can be implemented in Java, see [4].

In addition, several elementary functions which are provided in C should also be included in the Java API, the following, for example.

`hypot(x,y)`	returns `sqrt(`$x^2 + y^2$`)` without overflow whenever the result does not overflow.

Note that it is important that functions for `float` and `double` be put in the same class (e.g., the `Math` class). Currently, separate `isNaN` and `isInfinite` methods are found in the `Float` and `Double` classes. Because of this, the type of the argument has to be part of the function call, e.g. `Float.logb(f)` and `Double.logb(f)`. If both were in the same class, then the function names could be overloaded, allowing for easier maintenance of codes that are provided in both `float` and `double` versions.

4. **Extensions to support multiple NaN values.** This seems to be already in the making.

Development of Core Classes and Interfaces for Numerical Computing

The Numerics working group has agreed to begin the development of a variety of core numerical classes and interfaces to support the development of substantial Java applications in the sciences and engineering. The main purpose of this work is to standardize the interfaces to common mathematical operations. A reference implementation will be developed in each case. The purpose of the implementation will be to clearly document the class and its methods. Although we expect these to be reasonably efficient, we expect that highly tuned implementations or those relying on native methods will be developed by others. Also, the simple methods, such as get or set, will not provide reasonable performance unless they are inlined, because the method invocation overhead will be amortized over very few machine instructions. Unless otherwise specified, we will initially only define classes based on `doubles`, since computations with Java `floats` are less useful in numerical computing.

The classes identified for first consideration are the following.

- **Complex**

  This implements a complex data type for Java. It includes methods for complex arithmetic, assignment, as well as the elementary functions. A strawman proposal has already been developed and released for comment.

  Current working proposal: http://www.vni.com/corner/garage/grande/index.html

Contacts: John Brophy, Visual Numerics
Marc Snir, IBM

- **Multidimensional arrays**

  This implements one, two and three-dimensional arrays for Java as described above. A strawman proposal has already been developed and released for comment.

  Current working proposal: http://math.nist.gov/javanumerics/#proposals

  Contacts: Jose Moreira, IBM
  Marc Snir, IBM
  Roldan Pozo, NIST

- **Linear algebra**

  This implements matrices (in the linear algebraic sense) and operations on matrices such as the computation of norms, standard decompositions, the solution of linear systems, and eigenvalue problems. A strawman proposal has already been developed and released for comment.

  Current working proposal: http://math.nist.gov/javanumerics/jama

  Contacts: Cleve Moler, The MathWorks
  Roldan Pozo, NIST
  Ron Boisvert, NIST

- **Basic Linear Algebra Subroutines (BLAS)**

  These implement elementary operations on vectors and matrices of use to developers of linear algebra software (rather than to average users). This work will be done in conjunction with the BLAS Technical Forum. Some working notes on this effort can be found at http://math.nist.gov/javanumerics/blas.html

  Contacts: Roldan Pozo, NIST
  Steve Hague, NAG
  Keith Seymour, University of Tennessee

- **Higher Mathematical Functions**

  This includes functions such as the hyperbolics, erf, gamma, Bessel functions, etc. A strawman proposal has already been developed and released for comment.

  Current Working Proposal: http://www.vni.com/corner/garage/grande/index.html

  Contacts: John Brophy, Visual Numerics
  Ron Boisvert, NIST

- **Fourier Transforms**

  This includes not only a general complex transform, but specialized real, sine and cosine transforms.

Contacts: Lennart Johnsson, University of Houston

- **Interval Arithmetic**

  This implements an interval real data type for Java. It includes methods for interval arithmetic, assignment, as well as elementary functions. An API is actively under development.

  Contacts: Dmitri Chiriaev, Sun

- **Multiprecision Arithmetic**

  This implements a multiprecision real data type for Java. It includes methods for arithmetic, assignment, as well as elementary functions.

  Contacts: Sid Chatterjee, University of North Carolina

The working group will review these proposals and open them up for public comment. It will also set standards for testing and documentation for numeric classes. It will work with Sun and others to have such classes widely distributed.

3 Concurrency and Applications Working Group Recommendations

Preface

The primary concern of the Java Grande Forum (hereafter JGF) is to ensure that the Java language, libraries and virtual machine can become the implementation vehicle of choice for future scientific and engineering applications. The first step in meeting this goal is to implement the complex and numerics proposals described in the report of the Numerics Working Group. Accomplishing this task provides the essential language semantics needed to write high quality scientific software. However, more will be required of the Java class libraries and runtime environment if we wish to capitalize on these language changes. The Java Grande Forum Applications & Concurrency Working Group (hereafter ACG) focuses on these issues.

It is possible that many of the needed improvements will be driven by commercial sector efforts to build server side enterprise applications. Indeed, the requirements of technical computing overlap with those of large enterprise applications in many ways. For example, both technical and enterprise computing applications can be very large and they will stress the memory management of the VM. The demand for very high throughput on network and I/O services is similar for both. Many of the features of the Enterprise Bean model will be of great importance to technical computing.

But there are also areas where technical computing is significantly different from Enterprise applications. For example, the performance of fine-grained concurrency (or parallelism) is substantially more critical in technical computing where a single computation may require 10,000 threads that synchronize in frequent, regular patterns. These computations would need to run on desktops as well as very large, shared memory multiprocessors. In technical applications, the same data may be accessed again and again, while in enterprise computing there is a great emphasis on transactions involving different data each time. Consequently, memory locality optimization may be more important for Grande Applications than it is elsewhere in the Java world. Some technical applications will require the ability to link together multiple VMs concurrently executing on a dedicated cluster of processors which communicate through special high performance switches. On such a system, specialized, ultra low latency versions of the RMI protocol would be necessary. (In such an environment, an interface to shared memory, via RMI or the VM, would also be desirable.)

It is important to observe that there are problems which can be described as technical computing today which will become part of the enterprise applications of the future. For example, image analysis and computer vision are closely tied to application of data mining. The processing and control of data from arrays of sensors has important applications in manufacturing and medicine. The large scale simulation of non-linear mathematical systems is already finding its way into financial and marketing models. It is also the case that many technical computing applications do impact our day-to-day lives, such as aircraft simulation (the recent design of the Boeing 777) and weather forecasting. At least in the case of aircraft design, the industry has a valuation in the billions of dollars, which means it is far from being merely niche area being of limited interest.

Organization

This section of the Java Grande Report pertains to Concurrency/Applications. It is organized as follows:

- critical JDK issues
  highest priority issues, mostly related to Remote Method Invocation
- benchmarks
- seamless computing
- other parallel and distributed computing issues

In this report, we present preliminary findings of the working group. We welcome a continuing discussion of these issues. Please send questions or comments to javagrandeforum@npac.syr.edu.

Critical JDK Issues

Sequential VM performance is of utmost importance to develop Grande Applications. Since there are many groups working on this issue, the ACG simply provides some additional kernel benchmarks illustrating performance aspects in areas that are particularly important for Grande Applications.

In addition to sequential VM performance, Grande Applications require high performance for parallel and distributed computing. Although some more research is needed on other paradigms that might be better suited for parallelism in Java, this report will focus on RMI (Java's remote method invocation mechanism), since there is wide-spread agreement on both the general usefulness and the deficiencies of RMI.

In general, RMI provides the capability of allowing objects running in different JVMs to communicate (more specifically, to invoke methods on each other). Current RMI is specifically designed for peer-to-peer client/server-applications that communicate over TCP based networks. For high performance scientific applications, however, some of RMI's design goals and implementation decisions are inappropriate and cause serious performance limitations. This is especially troublesome on platforms targeted by the Java Grande community, i.e., closely connected environments, e.g. clusters of workstations and Distributed Memory Processors (DMPs) that are based on user-space interconnects like Myrinet, ParaStation, or SP2. On these platforms, a remote method invocation may not take longer than a few tens microseconds.

The choice of a client/server model may prove too limited for many applications in scientific computing, which usually take advantage of collective communication as found in the Message Passing Interface (MPI); however, the goal of this document is not to propose such sweeping changes to RMI. We rather choose to focus on how to make RMI, a client/server design, suitable for Grande Applications with no changes to the core model itself. It is our hope that a better RMI design and implementation will stimulate community activities to support better communication models that are well-suited to solving community problems.

Performance of Object Serialization

Requirement. Fast remote method invocations with low latency and high bandwidth are essential, especially in areas of science and engineering, where fine-grained parallelism is exploited. Since object serialization and parameter marshaling are the mechanisms used for passing parameters to remote calls and the associated cost(s) amount to a significant portion of the cost of remote method invocation, serialization should be as fast as possible.

In an ideal solution, the exact byte representation of an object would be sent over the network and turned back into an object at the recipient's side without any unnecessary buffering and copying.

The ACG understands that the JDK's object serialization is used for several purposes, e.g. for long term storage of persistent objects and for dynamic class loading on remote hosts via http-servers. It is obvious that some of these special purpose uses require properties that are either costly to compute at runtime (latency) or that are verbose in their wire representation (bandwidth).

However, since some of these features are not used in Grande Applications, there is room for improvement. The following subsections identify particular aspects of the current implementation of serialization that result in bad performance. The problems are described, and some solutions are suggested. Where possible, some benchmark results demonstrate the quantitative effects of the proposed solution.

Experiment. Experiments at Amsterdam [31] indicate that easily up to 30% of the run time of a remote method invocation are spent in the serialization, most of which can be avoided by compile time serialization.

Slim Encoding of Type Information

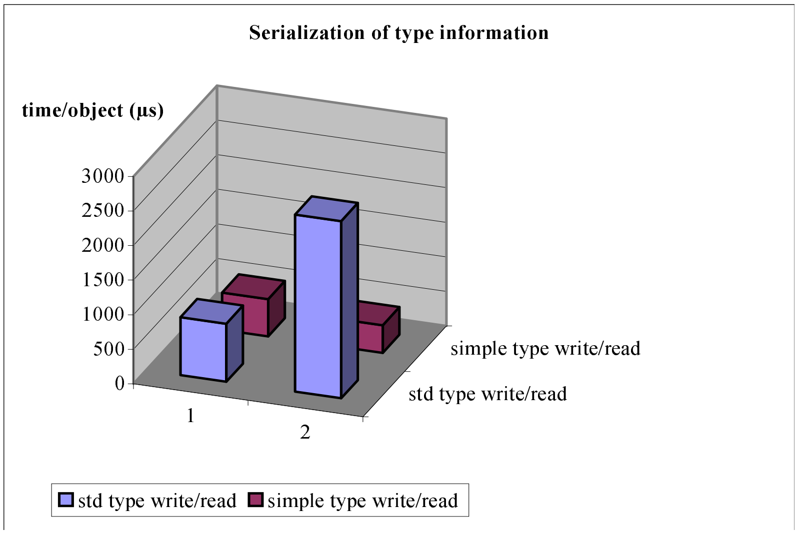
Problem. For every type of object that is serialized, the current implementation prepends a complete description of the type, i.e., all fields of the type are described verbosely. For a single serialization connection, every type is marshaled only once. Subsequent objects of the same type use a reference number to refer to that type description. Type description is not only useful when objects are stored persistently and when the recipient does not have access to the byte code representation of the type, but it also guarantees the correct handling of inheritance.

When RMI uses the serialization for marshaling of method parameters, a new serialization connection is opened for every single method invocation. (More specifically, the `reset` method is called on the serialization stream.) Hence, type information is marshaled over and over again, thus consuming both latency and bandwidth. The current implementation cannot keep the connection open, because the serialization would otherwise refrain from re-sending arguments with modified instance variables. (Note, that the whole structure of objects reachable from argument objects is serialized; one of the objects deeply burried in that graph might have changed.)

Approach. For Grande Applications it can be assumed that all JVMs that collaborate on a parallel application use the same file system, i.e., can load all classes from a common CLASSPATH. Furthermore, it is safe to assume that the life time of an object is within the boundaries of the overall runtime of the application. Hence, there is no need to completely encode and decode the type information in the byte stream and to transmit that information over the network, unless the type information is needed to resolve inheritance issues. Many scientific applications are pretty straightforward and do not exploit deep class hierarchies, so that type information often need not be present in the bytestream.

Experiment. At Karlsruhe University, a slim type encoding has been implemented prototypically [28]. It has improved the performance of serialization significantly by avoiding the latency of complicated type encoding and decoding mechanisms. Moreover, some bandwidth can be saved due to the slimmer format. Figure 1 shows the runtime of standard serialization in the first/blue row and the runtime of the improved serialization with slim type encoding in the second/red row. The effect is much more prominent on the side of the reader (right two bars, 2) than on the side of the writer (left two bars, 1).

FIGURE 1. **Serialization with Slim Type Encoding**



Solution. The ACG sees three options to avoid costly encoding of type information.

- Some performance improvements would be reached if the object serialization would offer two types of `reset` methods. Similar to the current `reset` method, one method would reset the cached information on both types and objects. The other method would reset only the information on objects that have already been sent.
- An additional `ProtocolVersion` is added to the implementation of serialization. Both the `rmiregistry` and the RMI user programs must select the new protocol version. For this purpose, the `RMISocketFactory` will need a new method `setProtocolVersion(int)` that RMI user programs must call. In addition, the `rmiregistry` may need a new command line option (protocol number), although this is not so important, since the registry is not likely to become a bottleneck.
- The more general solution is similar to the socket factory approach where the user can provide the name of the class that implements his own implementation of object serialization. Both the `rmiregistry` and the RMI user programs must select the new protocol version. For that purpose, the `rmiregistry` will need a new command line option (class name). Similar to `RMISocketFactory`, an `RMISerializationFactory` is added to the API that RMI user programs must initialize. Experiments conducted at the Indiana University have shown that this approach is indeed feasible and can lead to speedup.

The ACG favors one of the latter two approaches. The first approach would help a little, but the complete type information would still be sent, although less frequently. The other approaches allow more control over the process, thus allowing for more optimizations.

Handling of Floats and Doubles

Problem. Since in scientific applications, floats and arrays of floats are used frequently (the same holds for doubles), it is absolutely essential, that these data types are packed and unpacked efficiently. Nevertheless, the current serialization does not handle these primitive types efficiently.

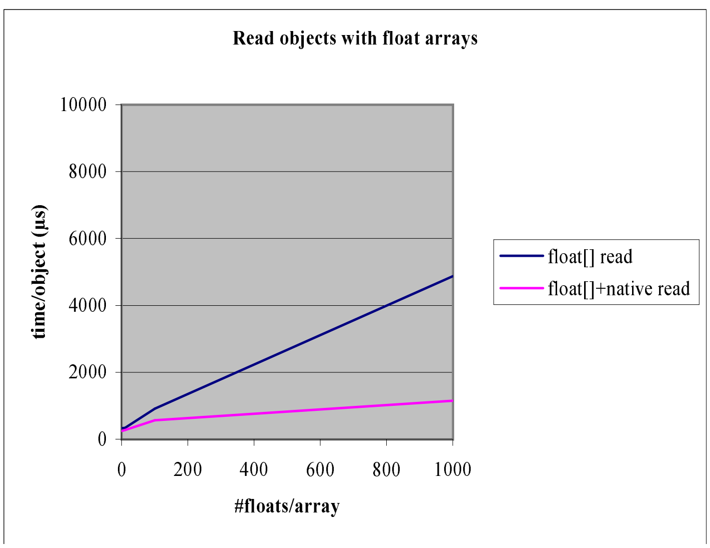
The conversion of these primitive data types into their equivalent byte representation is (on most machines) a matter of a type cast. However, in the current implementation, the type cast is implemented in the JNI and hence requires various time consuming operations for check pointing and state recovery upon JNI entry and JNI exit. Moreover, the serialization of float arrays (and double arrays) currently invokes the above mentioned JNI routine *for every single array element*.

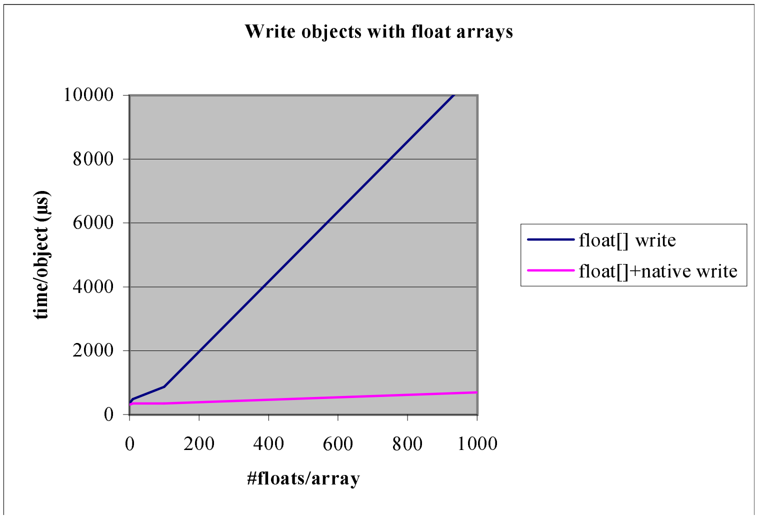
Approach.

- The costly JNI-entry/exit can be avoided by teaching the JITs to recognize and inline the conversion of single floats or doubles into byte arrays. The conversion routine could even be included into the JVM.
- For arrays, it is much faster to enter the JNI only once for the whole array (or at least for the section that still fits in the available communication packet).
- If the JNI code must be retained, at a minimum we believe a single JNI call should be made to convert multiple scalars. See section on Reflection Enhancements.

Experiment. A prototype at Karlsruhe University (see Figure 2, [28]) stresses the effect of this approach.

FIGURE 2. **Serialization of Float Arrays**





Solution. It is absolutely essential that

- floats and doubles are turned into their corresponding wire representation as efficiently as possible and that
- arrays of floats and doubles are serialized efficiently.

And since there are easy ways to do it that do not require any change of an API, this should be done in the next release of the JDK.

Reflection Enhancements

Problem. A byte array is used as communication packet. During serialization, every single instance variable is copied into/from that byte array individually. Most of this copying is done at the Java level using reflection. (As mentioned above, only for floats and doubles the JNI is asked to return the appropriate byte representation.)

Part of the reflection overhead can be relieved by means of special compiler support. The RMI implementation developed at Indiana University uses a compiler to traverse over all fields of a serializable object, writing all appropriate fields to the buffer. This compiler adds public traversal methods to the Java source code. In principle this task could be performed by any standard Java compiler.

In contrast to zero-copy protocols that are used (or attempted to be used) in other messaging protocols, where data is copied directly from user data structures to the network interface board, at least two complete copy operations (one at the sender side and one at the recipient side) are needed in every pure Java implementation. (Unfortunately, current implementations do much more copying that this.) Although the ACG regrets it, it seems very unlikely that future JVM implementations will closely interact with the communication mechanisms to allow for zero-copy protocols. This seems to be one of the price tags caused by Java's portability that can only be avoided by compile time serialization, see [31].

Approach. Obviously, there should be as few copy operations as possible. Those that remain should be performed as fast as possible.

Solution. Better performance can be achieved in two ways

- The object stream opens up its communication buffer for public access so that the object itself can use a `writeExternal` routine to write into that buffer immediately, i.e., without reflection.

  Since the ACG does not believe that the communication buffer will ever be made public, the following proposal seems to be more promising.

- The reflection mechanism should be enhanced so that it can copy all instance variables into a buffer at once with a single method call. For example, class `Class` could be extended to return an object of a new class `ClassInfo`:

```
ClassInfo getClassInfo(Field[] fields);
```

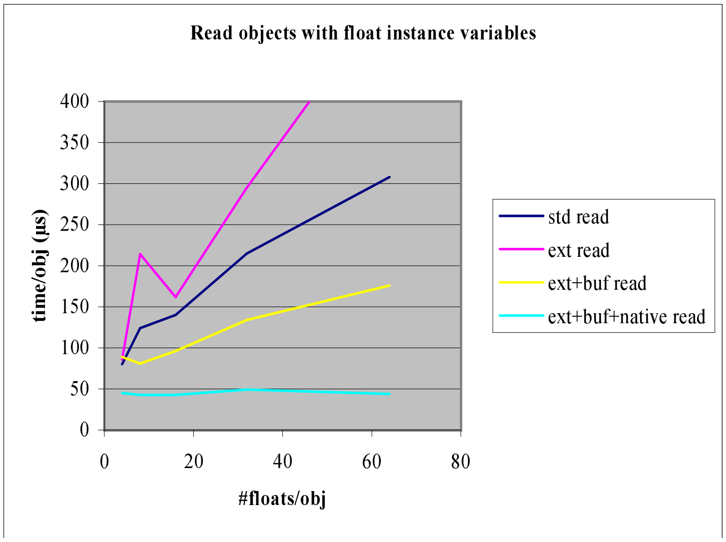
The object of type `ClassInfo` then provides two routines that do the copying to/from the communication buffer.

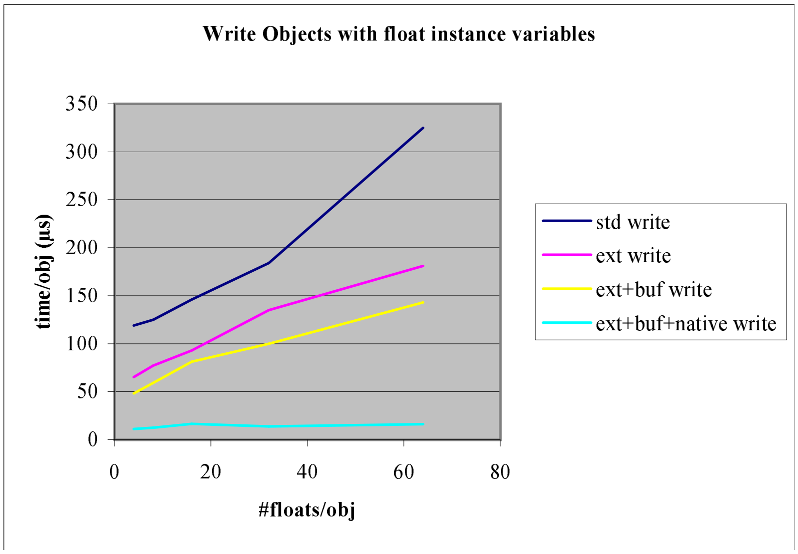
```
int toByteArray(Object obj, int objectoffset,

byte[] buffer, int bufferoffset);

int fromByteArray(Object obj, int objectoffset,

byte[] buffer, int bufferoffset);
```

The first routine copies the bytes that represent all the instance variables into the communication buffer (ideally the network interface board), starting at the given buffer offset. The first `objectoffset` bytes are left out. The routine returns the number of bytes that have actually been copied. Hence, if the communication buffer is too small to hold all bytes, the routine must be called again, with appropriately modified offsets.

Experiment. A serialization with better buffering and less copying has been implemented at Karlsruhe University [28]. The prototype is based on an enhanced reflection mechanism, that can deal with all instance variables of an object at once. (This has been hacked into native code that is called through the JNI.) The performance numbers are shown in Figure 3. In the figures, "std" refers to the standard serialization, "ext" indicates the fact that the object provides a `writeExternal` routine, "buf" incorporates more efficient internal buffering, and "native" uses the hack that is supposed to be replaced by an enhance reflection.

FIGURE 3. **Serialization of Several Instance Variables**





The ACG strongly recommends to allow for more efficient handling of the communication buffer, since performance of serialization can be increased by a factor of more than 10.

Implementation Improvements

Problem. In addition to the suggestions for improved implementation of object serialization, object serialization should take about the same time for writing and reading. Otherwise, the pipeline between sender and receiver shows an imbalance resulting in wasted time.

Experiment. In the current implementation, it is much faster to produce the byte array representation of an object than it is to recreate the object. The severity of this problem, that is probably caused by the overhead of object creation in Java, can be seen in Figure 1, Figure 2, and Figure 3 above.

Approach. The ACG cannot provide any specific suggestions here.

Performance of RMI

Requirement. Fast remote method invocations with low latency and high bandwidth are essential, especially in areas of science and engineering, where fine-grained parallelism is exploited. Apart from object serialization, there are other aspects of RMI that should be improved.

Experiment. Experiments at Amsterdam [31] indicate that easily up to 30% of the run time of a remote method invocation is spent in the RMI implementation. The other 30% is spent in the network. By compiling Java code to native, a latency of a few tens microseconds for a remote method invocation can be achieved on Myrinet. About the same performance is needed for jit compiled pure RMI code on custom interconnection hardware.

Improved Connection Management

Problem. The current RMI implementation closes and re-opens connections to other hosts too frequently. For object serialization streams, that causes the re-transmission of type information, as has been discussed above. Re-creation of socket connections is a costly operating system job.

Approach. Whereas RMI's approach seems to be useful in case of unstable wide area networks, for closely connected JVMs this approach is far too pessimistic. Socket connections should be left open for the duration of the user program. At least, it might make more sense to keep a working set and use a least recently used algorithm to kick connections out of the working set.

Solution. To achieve this, the user should have an option to switch RMI into that mode. Again, this requires a command line option for the `rmiregistry` and another method in the `RMISocketFactory` class.

Careful Resource Consumption

Problem. In the current RMI implementation, every single remote object uses a port of its own. In addition, a thread is started monitoring that port. Every remote method invocation causes the creation of a new socket and a new thread. In addition, a watch dog thread monitors the state of a connection. Since object creation is known to be quite slow in current JVMs and since thread creation is even slower, this approach is not amenable for high performance comptuing, especially for problems where fine-grained parallelism is used.

Approach. Instead of creating new objects and threads almost on every RMI activity, the internal layers of RMI should reuse sockets. Moreover, worker threads from a thread pool should repeatedly service incoming requests. The underlying asumption is that the network is quite stable and communication failure will cause failure of the whole application.

Problem. Operating systems typically allow one thread to monitor several socket connections by means of a `select` statement. There is currently no way for a Java thread to do the same. Instead, one Java thread is needed for every single open socket connection. This is not only a significant overhead for the JVM's thread scheduler but it prevents efficient implementations of socket based custom transports layers for RMI.

Solution. The JPACWG strongly recommends that JDK's sockets should provide a `select` statement. The RMI transport implementation should then make use of it.

Custom Transport

Requirement. Fast remote method invocations with low latency and high bandwidth are essential, especially in areas of science and engineering, where fine-grained parallelism is exploited. It is absolutely essential that non-Ethernet special purpose high-end communication hardware can be used.

Problem. It is almost impossible to use specialized, high performance network protocols through RMI. Although SCI, ATM AAL5, Myrinet, ParaStation, and Active Messages are all used in technical applications, there is no straightforward way for Java applications to make use of them.

- RMI is tied to TCP. The current RMI implementation is tightly connected to TCP sockets. Whenever UnicastRemoteObject and UnicastServerRef are used, a TCP transport is selected implicitly.

  The transport cannot be replaced individually, since the implementation does not properly isolate the transport layer. More specifically, the UnicastRemoteObject source code has lots of type casts that explicity use TCP sockets and their methods. Hence, to get rid of the TCP transport, a completely new type of server needs to be implemented by hand as well.

  Based on the current implementation and documentation of RMI, this work is comparable to re-inventing most of RMI's functionality. Although deeply burried in the internal layers of the RMI implementation, the transport seems to be plugged in very flexibly, there is no way that flexibility can be exploited without proper documentation.

  In [29], experiments are described with an RMI implementation built on top of Nexus. Because of the close integration of the individual parts in RMI the implementation described here contains compilers and a runtime system that were built from scratch, which shows that it is difficult to make each part separately customizable.

- Non-Socket-Services. Most available high speed protocols do not operate on the socket level, e.g. FM and Nexus. Although RMI offers a socket factory where user defined socket classes can be plugged in, this is unsufficient because it would require a costly socket protocol to be implemented on top of the high speed base functionality.

  There should be a way to make use of non-socket protocols in RMI. The following aspects need to be considered:

  For sockets (and TCP/IP) most of the necessary protocol initialization is done by the operating system. This explains that the current implementation does not offer appropriate interfaces to plug in specific protocol initialization code as needed by other protocols, especially by Nexus. It might be possible to use the static class initializer of a abstract transport protocol class for that purpose but that has to be determined.

  Another related problem that is caused by the fact that the current approach is based on sockets is the lack of an interface to provide addressing information to interact with other transport protocols. IP and DNS names are not the only meaningful forms. For example, on the SP2 or a Cray T3E, a combination of IP# and node number. (e.g., quad.mcs.anl.gov/25, for node 25 at quad.mcs.anl.gov) is used.

- User-Level-Services. Some high speed protocols offer socket like functionality in the user level, i.e., the operating system is left out for performance reasons.

  Since these protocols offer socket functionality, it seems to be easy to wrap them in a Java socket class that is then plugged into the RMI socket factory. Unfortunately, that approach does not work properly.

  The problem is that `SocketImpl` returns file descriptors that are later on used by the JVM's thread scheduler. It issues calls on read/write/select methods which are only useful on the level of the operating system. For example, a select might be executed that waits for the arrival of a communication packet in the kernel although that packet has already arrived at the user level.

  Hack around. The only reasonable way to go seems to be to load a dynamic library that replaces the standard operating system calls with those that can handle the high end communication hardware. This replacement is done unnoticed by the JVM. Such an approach has been successfully tested at the University of Karlsruhe on ParaStation hardware, it required an undesirable source code modification of the `rmiregistry` implementation (to load the dynamic library before `main`).

The JPACWG strongly recommends that RMI will be extended to allow for high-speed communication hardware to be used. Since the source code of the lower layers of the RMI implementation are not public, no specific suggestions can be made in this report. The minimal requirement is that the RMI group provides documentation on how to plug in alternative transport implementations.

Other Suggestions

These additional suggestions are not at the same level of criticality as those presented in the previous section (See “Critical JDK Issues” above). Nonetheless, they have been addressed repeatedly in working group discussions and will continue to be explored as part of ongoing working group meetings.

Source Code Availability

Problem. The RMI implementation needs class files from several packages. A lot of the code is contained in `sun.rmi.*`. For this code, the Java source is not released. In particular, there is no source code for the various versions of JDK 1.2.

There are people who are willing to work on that code to improve it's runtime efficiency. Because of the missing source code (and the fact that de-compilers do not recreate the comments) the missing source code these people are prevented from doing work the ACG is interested in.

Solution. The ACG suggests that Sun will include the source code of current RMI implementations either in the standard JDK distribution or will create a process so that interested research groups can get access to it before a release gets final.

Class Compatibility

Problem. The class compatibility test is too stringent. In the case where one is not using NFS (more common than the case where one is using it), it appears one can take the same code (unmodified), compile it with the same version of JDK (resulting in the same class files), and cause an exception, because the classes are deemed incompatible.

Solution. There may be no easy fix for this. Compatibility should probably be defined in terms of the version of Java being used as it is now; however, two classes with the same class and instance variables should be compatible, provided they are compiled with the same Java compiler.

Experiment. None known. Related to this problem is section 4.3 for which an experimental prototype has been constructed.

Dynamic Class Loading from Remote Hosts

Problem. The RMI implementation includes a network class loader (for which very little documentation exists on its purpose or usage). The basic idea and purpose of a network class loader is to load classes from the network instead of a local file system. The essence of how to build a network class loader is shown in the Class Libraries Reference by Lee and Chan (accomplished by extending the ClassLoader class in Java).

The notion of loading classes from the network is already known to be useful in the case of applets, where the code lives on the server in a code directory (often called the code "base") and can be downloaded to a web browser and loaded, subject to being verified by the class verifier. This is well known not to be without problems, because different browsers may not support the same Java (see Class Compatibility above); however, in principle it makes the life of a client easy.

In the case where a browser is not being used, it is still useful to consider the notion of a network class loader. In scientific and technical computing involving a large number of nodes (as found in massively parallel machines and clusters), the use of a network class loader can facilitate the deployment of code on a large number of nodes where there is not a shared file system (e.g., NFS or AFS). As it currently stands, when file systems are not shared, the programmer is forced to copy the class files to all nodes in a network where file systems are not shared. This can be quite cumbersome, although systems such as Unix provide remote commands to make the task easier. For Windows users, there is presently inadequate support for remote copying of files and remote execution, meaning that Windows file sharing must be used (still leaving one without adequate remote execution facilities). When heterogeneous systems are involved, the problem of deployment is even more complex, where everyone involved needs to know system administration.

Experiment. Thiruvathukal, Thomas, and Korczynski of Loyola University and jhpc.org have shown how to extend the class loader for a version of RMI called RRMI [29], where the client can start very thin and most, if not all, classes are loaded from the network.

Benchmarks

The purpose of developing a suite of benchmark tests is to provide ways of measuring and comparing alternative Java execution environments. In constructing this benchmark the aim is to provide a series of tests which challenge the execution environments in ways which are important to Grande Applications. The benchmark has been divided into three sections:

- Low level operations - measuring the performance of low level operations such as serialization, RMI, garbage collection.
- Kernels - short codes which carry out specific operations frequently used in Grande Applications.
- Large scale applications - real grande codes, less useful for comparative performance studies but worthwhile to demonstrate the potential of Java for tackling real problems.

ACG feels that it is important wherever possible that the benchmarks should be open source to allow users to understand exactly what the benchmark is testing.

EPCC in the University of Edinburgh is coordinating this effort, at time of writing, and are in the process of collecting existing benchmark tests to form a coherent suite. The aim of this collation is to have a package that can be downloaded and run as a whole, with consistent output format and a consistent definition of terms. This suite can then be added to over time as more and better tests are developed. The work being done at Edinburgh is ongoing and details can be found at:

http://www.epcc.ed.ac.uk/research/javagrande/

Comments suggestions and contributions from the forum and the community at large are invited and any contributions would be warmly welcomed (please email: epcc-javagrande@epcc.ed.ac.uk).

The following sections elaborate on the proposed structure for the benchmark including examples of the tests that could be included. In most cases these tests are either available or have been promised.

It is anticipated that up to three different sizes of dataset should be available for section 2 (and maybe even two for section 3).

The user also should be able to run:

- any benchmark individually, no matter what section
- all of section 1
- all of section 2 with a specific application size
- all of section 3 with a specific application size

It seems unlikely that users would wish to run the entire benchmark suite in one go, but this facility could be added for completeness.

For timed benchmarks, the benchmarks will report a raw time (in seconds) and a performance measure in X/second, where the operation X is specific to the given benchmark (e.g. flops, method calls, iterations, etc.). Non-timing benchmarks will typically report a single figure (e.g. maximum size of object), but units should be consistent (all memory sizes in bytes, for example).

Low Level Operations

- loop overhead
- access variables and arrays
- method invocation
- execution of arithmetic operations
- casting
- object and array creation/instantiation
- exception handling
- memory management and garbage collection (Piyush Mehrorta)
- thread creating / switching
- threads - synchronization, scalability (Dennis Gannon)
- I/O scalability (Dan Reed?)
- RMI/serialization (Michael Philippsen/Bernhard Haumacher)

Kernels

- FFT
- Numerical integration
- SOR
- LU Factorisation
- Sparse Matrix multiply
- Video/audio (de)compression
- Searching
- Sorting

Large Scale Applications

- Parallel Geophysics Operations in Java(University of Karlsruhe)
- Monte Carlo simulation (NCSA/EPCC)

- Discrete Event Simulation (INRIA/EPCC)
- Image Analysis, Radio Astronomy (NCSA)
- Gravitational N-Body simulations (Indiana)
- Computational Fluid Dynamics (Syracuse)

Message Passing Interface

Introduction and background

A basic prerequisite for parallel programming is a good message passing API. Java comes with various ready-made packages for communication, notably an easy-to-use interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. Interesting as these interfaces are, it is questionable whether parallel programmers will find them especially convenient. Sockets and remote procedure calls have been around for about as long as parallel computing has been fashionable, and neither of them has been popular in that field.Both communication models are optimized for client-server programming, whereas the parallel computing world is mainly concerned with “symmetric” communication, occurring in groups of interacting peers.

This symmetric model of communication is captured in the successful Message Passing Interface (MPI) standard, established a few years ago [25]. MPI directly 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. Reliable point-to-point communication is provided through a shared, group-wide communicator, instead of socket pairs. MPI allows numerous blocking, non-blocking, buffered or synchronous communication modes. It also provides a library of true collective operations (broadcast is the most trivial example). An extended standard, MPI-2 [26], allows for dynamic process creation and access to memory in remote processes.

The MPI standard document thus far has provided language-independent specification as well as language-specific (C and Fortran) bindings. While the MPI-2 release of the standard added a C++ binding, no Java binding has been offered or is planned by the MPI Forum. With the evident success of Java as a programming language, and its inevitable use in connection with parallel as well as distributed computing, the absence of a well-designed language-specific binding for message-passing with Java will lead to divergent, non-portable practices. Therefore, a standard specification is urgently needed to enable the development of portable Java Grande Applications using MPI.

Current status

There are several known efforts towards the design of early MPI interfaces for Java with three fully functional but different Java-MPI implementations - mpiJava, JavaMPI, MPIJ, and ActiveJava (formerly JavaNOW).

The design of mpiJava is based on the use of native methods to build a wrapper to existing MPI library (MPICH). A comparable approach has been followed in the development of JavaMPI, but the JavaMPI wrappers were automatically generated by a special-purpose code generator. A large subset of MPI is implemented in pure Java within the DOGMA system for Java-based parallel programming.

MPI Software Technology, Inc. have also announced that there is a commercial effort under way to develop a message-passing framework and parallel support environment for Java called JMPI [24]. Some of these “proof-of-concept'' implementations have been available for more than a year with successful ports on clusters of workstations running Solaris or Windows NT, as well as IBM SP2, SGI Origin-2000, Fujitsu AP3000, and Hitachi SR2201 parallel platforms. ActiveJava [30] is a framework that has been prototyped by the Java and High-Performance Computing research group that is based on Linda and Actors that supports an MPI “personality” (focused primarily on collective communication/computation operators) via a class library that utilizes the kernel of the framework.

The mpiJava wrapper. The mpiJava API is modelled as closely as practical on the C++ binding defined in the MPI 2.0 standard but currently supports only the MPI 1.1 subset [22]. A number of changes to argument lists are forced by the restriction that arguments cannot be passed by reference in Java. In general outputs of mpiJava methods come through the result value of the function. In many cases MPI functions return more than one value. This is dealt with in mpiJava in various ways. Sometimes an MPI function initializes some elements in an array and also returns a count of the number of elements modified. In Java we typically return an array result, omitting the count. The count can be obtained subsequently from the length member of the array. Sometimes an MPI function initializes an object conditionally and returns a separate flag to say if the operation succeeded. In Java we return an object handle which is null if the operation fails. Occasionally an extra field is added to an existing MPI class to hold extra results - for example the Status class has an extra field, index, initialized by functions like Waitany. Rarely none of these methods work and we resort to defining auxilliary classes to hold multiple results from a particular function. In another change to C++, we often omit array size arguments, because they can be picked up within the wrapper by reading the length member of the array argument.

As a result of these changes mpiJava argument lists are often more concise than the corresponding C or C++ argument lists.

Normally in mpiJava, MPI destructors are called by the Java finalize method for the class. This is invoked automatically by the Java garbage collector. For most classes, therefore, no binding of the MPI_class_FREE function appears in the Java API. Exceptions are Comm and Request, which do have explicit Free members. In those cases the MPI operation could have observable side-effects (beyond simply freeing resources), so their execution is left under direct control of the programmer.

Automatic generation of MPI wrappers. In principle, the binding of existing MPI library to Java using JNI amounts to either dynamically linking the library to the Java virtual machine, or linking the library to the object code produced by a stand-alone Java compiler. Complications stem from the fact that Java data formats are in general different from those of C. Java implementations will have to use JNI which allows C functions to access Java data and perform format conversion if necessary. Such an interface is fairly convenient for writing new C code to be called from Java, but is not adequate for linking existing native code.

Clearly an additional interface layer must be written in order to bind a legacy library to Java. A large library like MPI has over a hundred exported functions, therefore it is preferable to automate the creation of the additional interface layer. The Java-to-C interface generator (JCI) [4] takes as input a header file containing the C function prototypes of the native library. It outputs a number of files comprising the additional interface: a file of C stub-functions; files of Java class and native method declarations; shell scripts for doing the compilation and linking. The JCI tool generates a C stub-function and a Java native method declaration for each exported function of the MPI library.

Every C stub-function takes arguments whose types correspond directly to those of the Java native method, and converts the arguments into the form expected by the C library function.

As the JavaMPI bindings have been generated automatically from the C prototypes of MPI functions, they are very close to the C binding. However, there is nothing to prevent from parting with the C-style binding and adopting a Java-style object-oriented approach by grouping MPI functions into a hierarchy of classes.

Pure Java implementation of MPI. MPIJ is a completely Java-based implementation of MPI which runs as part of the Distributed Object Group Metacomputing Architecture (DOGMA) system. Being closely based on the C++ binding, MPIJ implements a large subset of MPI functionality including point-to-piont communication (all modes), intracommunicator operations, groups, user-defined reduction operations. Noteable capabilities that are not yet implemented include process topologies, caching, intercommunicators, and user-defined datatypes (these are arguably needed for legacy code only).

MPIJ communication uses native marshaling of primitive Java types. This technique allows MPIJ to achieve communication speeds comparable to, and frequently exceeding that, of native MPI implementations. (Java communication speed would be greatly increased if native marshaling were a core Java function.)

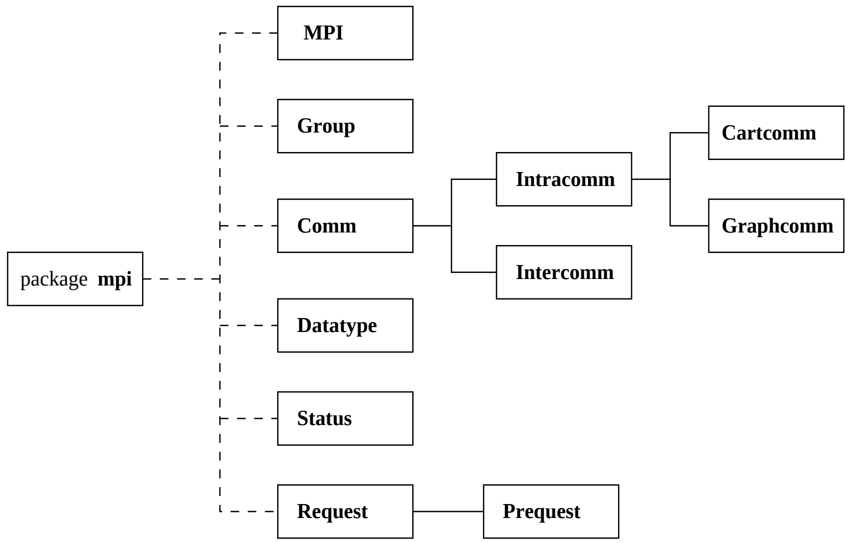
Current MPIJ work involves porting native marshalling to platforms other than Win32, investigation of standard libraries (e.g. BLAS) for improved performance, and porting to an improved version of DOGMA.

Proposed joint efforts

Java-MPI - API Specification. The MPI standard is explicitly object-based. The C and Fortran bindings rely on “opaque objects'' that can be manipulated only by acquiring object handles from constructor functions, and passing the handles to suitable functions in the library. The C++ binding specified in the MPI-2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The Java-MPI API specification follows this model, lifting the structure of its class hierarchy directly from the C++ binding.

The immediate infrastructure to be provided builds literally on the MPI-1 infrastructure offered by the MPI Forum, together with language bindings motivated by the MPI-2 forum's C++ bindings. The purpose of that 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++, Fortran77, and Java. Eventually, support for aspects of MPI-2 belong under this category as well, particularly dynamic process management, but not necessarily all of MPI-2, given its spartan implementation in the non-Java space.

FIGURE 4. Principal classes of Java-MPI



The major classes of Java-MPI are illustrated in Figure 1. The class MPI only has static members. It acts as a module containing global services, such as initialization of MPI, and many global constants including the default communicator COMM_WORLD. The most important class in the package is the communicator class Comm. All communication functions in Java-MPI are members of Comm or its subclasses. As usual in MPI, a communicator stands for a “collective object'' logically shared by a group of processors. The processes communicate, typically by addressing messages to their peers through the common communicator. Another class that is important for the Java-MPI specification is the Datatype class. This describes the type of the elements in the message buffers passed to send, receive, and all other communication functions.

The complete Draft Java-MPI API Specification is available as a separate document [23].

Advanced Message Passing Infrastructure for Java. The present effort's purpose is to offer a first principles study of how to present MPI-like services to Java programs, in an upward compatible fashion. The purposes are twofold: performance and portability. For performance, we seek to take advantage of what has been learned since MPI-1 and MPI-2 were finalized, or which were ignored in MPI standardization for various reasons. The study will, for instance, draw on the body of knowledge just recently completed within the MPI/RT Forum, which strives to enhance both real-time and performance of message passing programs. From MPI/RT, we will at least glean design hints concerning channel abstractions, and the more direct use of object-oriented design for message passing than was done in MPI-1 or MPI-2 (despite existence of C++ bindings). Additionally, a fundamental look at data marshalling and unmarshalling in the Java context will be undertaken, and preference for Java-natural mechanisms and policies will be attempted. Along the lines of portability, a detachment from legacy implementations of Java over existing native methods will be emphasized, while also considering the possibility of layering the messaging middleware over standard transports and other Java-compliant middleware (such as CORBA). In a sense, the middleware developed at this level should offer a choice of a performance or generality emphasis, while always supporting portability. A policy/opportunity to support aspects of real-time and fault detection/fault-aware programs will be studied and standardized insofar as possible, again drawing on the concepts learned in the MPI/RT activity, and also drawing on experience from distributed computing real-time activities. The validity of this type of messaging middleware in the embedded and real-time Java application spaces will also be considered.

Further Actions. The ACG currently consideres to release a Request for Proposals for the Java-MPI efforts to cover:

- Standard Java-MPI API Specification
- Java-MPI wrapper publicly available on the Web
- Intelligent generator of wrappers to legacy MPI libraries
- Pure Java MPI implementation
- Test suite
- Java-MPI Benchmarks

Other Parallel Issues

The role parallel computation plays in high performance technical computing cannot be under estimated. There are several ways to building a Java parallel computing environment. The following approaches are controversial and require community research.

- Grande parallelism design patterns. One approach is to take the experience of the last ten years of parallel programming and build a set of Grande parallelism design patterns that can be cast as a set of interfaces and base classes that simplify the task of writing parallel Grande Applications. This API can then be hosted on either a set of concurrently executing JVMs or in an environment where large numbers of native threads are well supported. Such an API may be as simple as defining a truly object oriented version of MPI, or it may define a new category of distributed object aggregates and collective operations.
- Grande Beans. A second approach that may be more consistent with current Java directions would be to design a Grande Bean specification that extends the basic Bean model to one appropriate for technical applications. This would follow what has been done with Enterprise Beans for transaction oriented business applications. The Enterprise Beans model has allowed CORBA based resources to be woven into a unified component model. Grande beans can build upon this to incorporate high end, parallel computational modules and visualization and VR tools into a grid of resources controlled by the JVM on the users desktop system.
- Object model: value semantics in procedure calls, remote reference management, problems for large objects [API], clumsy use of inheritance.
- Threads, costly synchronization, making use of LWP structures, etc.

4 References

Numerics References

[1] Alexander Aiken and Manuel Fahndrich and Raph Levien, Better static memory management: improving region-based analysis of higher-order languages, in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[2] Bruno Blanchet, Escape Analysis: Correctness Proof, Implementation and Experimental Results, in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 19-21, 1998.

[3] Joseph D. Darcy, *Borneo 1.0: Adding IEEE 754 floating point support to Java*, M.S. thesis, University of California, Berkeley, May 1998, http://www.cs.berkeley.edu/~darcy/Borneo.

[4] Joseph D. Darcy, Writing Robust IEEE Recommended Functions in "100% Pure Java"™, University of California, Berkeley, 1998, http://www.cs.berkeley.edu/~darcy/Research/ieeerecd.ps.gz.

[5] James W. Demmel and Xiaoye Li, Faster Numerical Algorithms via Exception Handling, IEEE Transactions on Computers, vol. 42, no. 8, August 1994, pp. 983-992.

[6] Alain Deutsch, On the Complexity of Escape Analysis, in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 15-17, 1997.

[7] Roger A. Golliver, *First-implementation artifacts in Java*™.

[8] Roger A. Golliver, Personal communication, August 1998.

[9] James Gosling, *The Evolution of Numerical Computing in Java*, http://java.sun.com/people/jag/FP.html

[10] James Gosling, Bill Joy, and Guy Steele, *The Java™ Language Specification*, Addison-Wesley, 1996. See http://java.sun.com/docs/books/jls/.

[11] Tim Lindholm and Frank Yellin, *The Java™ Virtual Machine Specification*, Addison-Wesley, 1997. See http://java.sun.com/docs/books/vmspec/.

[12] William Kahan, *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*, http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps.

[13] Sam P. Midkiff, Jose E. Moreira, and Marc Snir, *Optimizing bounds checking in Java programs.* IBM Systems Journal, vol. 37 (1998), no. 3, pp. 409-453.

[14] Jose E. Moreira, Sam P. Midkiff, and Manish Gupta, *From flop to megaflops: Java for technical computing.* Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC'98. IBM Research Report 21166, 1998.

[15] Y. G. Park and B. Goldberg, Escape Analysis on Lists, in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, July 1992.

[16] David Stoutamire and Stephen Omohundro, *Sather 1.1*, August 18, 1996, http://www.icsi.berkeley.edu/~sather/Documentation/Specification/Sather-1.1/.

[17] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley Publishing Company, 1994.

[18] Sun Microsystems, *Proposal for Extension of Java Floating Point in JDK 1.2*, http://java.sun.com/feedback/fp.html.

[19] Mads Tofte and Jean-Pierre Talpin, Region-Based Memory Management, in *Information and Computation*, vol. 132, no. 2, pp. 109-176, February 1997.

[20] Peng Wu, Sam P. Midkiff, Jose E. Moreira, and Manish Gupta. Improving Java performance through semantic inlining. IBM Research Report 21313, 1998. *Submitted for publication*.

Concurrency References

[21] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon: RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. Concurrency: Practice and Experience, volume 10, 1998, to appear.

[22] B. Carpenter, G. Fox, G. Zhang, and X. Li. A draft Java binding for MPI. http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html, November, 1997

[23] B. Carpenter, G. Fox, V. Getov, G. Judd, and A. Skjellum. Java-MPI - API Specification (Draft Position Document). Java Grande TR-03-98, November 1998.

[24] G. Crawford III, Y. Dandass, and A. Skjellum. The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users. http://www.mpi-softtech.com/publications/JMPI_121797.html, MSTI, December 1997.

[25] Message Passing Interface Forum. MPI: A message-passing interface standard. International Journal of Supercomputer Applications, 8(3/4), 1994.

[26] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, July 1997.

[27] S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In: M. Bubak, J. Dongarra, J. Waniewski (Eds.), ‘‘Recent Advances in PVM and MPI”, Lecture Notes in Computer Science, 1332, 135-142, Springer Verlag, 1997.

[28] Michael Phillippsen and others, http://www.ira.uka.de/. The benchmarks have been performed on a 300 Mhz UltraSparc II with JDK 1.2beta3, JIT enabled. JDK 1.2 (production release) shows numbers that are similar in ratio.

[29] G. K. Thiruvathukal, L.S. Thomas, and A. T. Korczynski: Reflective Remote Method Invocation. Concurrency: Practice and Experience, volume 10, 1998, to appear.

[30] G. K. Thiruvathukal, The Active Java Framework (and JavaNOW Project), Euro-Par ‘98, First European Workshop on Java for High Performance Network Computing. http://www.jhpc.org.

[31] Ronald Veldema, Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, and Aske Plaat: Efficient Remote Method Invocation. Technical Report IR-450, Dept. of Computer Science, Vrjie Universiteit Amsterdam, The Netherlands, September 1998.

[32] Jim Waldo: Remote Procedure Calls and Java Remote Method Invocation. IEEE Concurrency. Pages 5-7, September 1998.

5 Participants

Chairs

The following individuals have acted as coordinators of the Java Grande Forum and its working groups.

- Ronald Boisvert, NIST, Numerics Working Group Co-chair
- Denis Caromel, INRIA Nice Sophia Antipolis, Concurrency Working Group Co-chair
- Geoffrey C. Fox, Syracuse University and NPAC, Academic Coordinator
- Dennis Gannon, Indiana University, Concurrency Working Group Co-chair
- Siamak Hassanzadeh, Sun Microsystems, Industry Coordinator
- Roldan Pozo, NIST, Numerics Working Group Co-Chair
- George K. Thiruvathukal, Loyola University at Chicago, Secretary General

Numerics Working Group

The following individuals contributed to the development of this document at the Java Grande Forum meetings on May 9-10 and August 6-7 in Palo Alto, California.

- Ronald Boisvert, NIST, **Co-chair**
- John Brophy, Visual Numerics
- Sid Chatterjee, University of North Carolina
- Dmitri Chiriaev, Sun
- Jerome Coonen
- Joseph D. Darcy
- David S. Dixon, Mantos Consulting
- Geoffrey Fox, University of Syracuse
- Steve Hague, NAG
- Siamak Hassanzadeh, Sun
- Lennart Johnsson, University of Houston
- William Kahan, University of California, Berkeley
- Cleve Moler, The MathWorks
- Jose Moreira, IBM
- Roldan Pozo, NIST, **Co-chair**
- Kees van Reevwijk, Technical University, Delft
- Keith Seymour, University of Tennessee
- Nik Shaylor, Sun
- Marc Snir, IBM
- George K. Thiruvathukal, Loyola University, Chicago
- Anne Trefethen, NAG

The working group also acknowledges helpful discussions with and input from the following individuals.

- Cliff Click, Sun
- Susan Flynn-Hummel, IBM
- Roger Golliver, Intel
- James Gosling, Sun
- Eric Grosse, Bell Labs
- Bill Joy, Sun
- Tim Lindholm, Sun
- Sam Midkiff, IBM
- Bruce Miller, NIST
- Karin Remington, NIST
- Henry Sowizral, Sun
- Pete Stewart, University of Maryland and NIST

Participants in the Applications & Concurrency Working Group

The following individuals contributed to the development of this document at the Java Grande Forum meetings on May 9-10 and August 6-7 in Palo Alto, California.

- Fabian Breg, Indiana University
- Denis Caromel, INRIA, France, **Co-chair**
- George Crawford, MPI Software Technology
- Geoffrey Fox, NPAC, Syracuse University
- Dennis Gannon, Indiana, **Co-chair**
- Vladimir Getov, University of Westminster, UK
- Piyush Mehrotra, ICASE
- Michael Philippsen, University of Karlsruhe, Germany
- Omer Rana, University of Wales, Cardiff, UK
- Tony Skjellum. MPI Software Technology
- Henry Sowizral, Sun
- George K. Thiruvathukal, Loyola University, Chicago
- Julien J. P. Vayssiere, INRIA Nice Sophia Antipolis
- Martin Westhead, EPCC, University of Edinburgh, UK

The following additional individuals also contributed comments which helped in the development of this document.

- Henri Bal, Vrjie Universiteit Amsterdam, The Netherlands
- Bernhard Haumacher, University of Karlsruhe, Germany
- Mary Pietrowicz, NCSA

Editor in Chief

- George K. Thiruvathukal, Loyola University, Java Grande Forum Secretary General

Associate Editors

- Fabian Breg, Indiana University
- Ronald Boisvert, NIST, Numerics Working Group Co-chair
- Joseph Darcy
- Geoffrey C. Fox, NPAC, Syracuse University, JGF Academic Coordinator
- Dennis Gannon, Indiana University, Concurrency Working Group Co-chair
- Siamak Hassanzadeh, Sun Microsystems, Industry Coordinator
- Jose Moreira, IBM Thomas J. Watson Research Center
- Michael Philippsen, University of Karlsruhe, Germany
- Roldan Pozo, NIST, Numerics Working Group Co-chair
- Marc Snir, IBM Thomas J. Watson Research Center