Interim Java Grande Forum Report

ACM Java Grande Conference 1999
June 12-14, 1999

JavaOne Conference
June 14-17, 1999

San Francisco, California

Java Grande Forum Technical Report JGF-TR-4
(Original Report JGF-TR-1)

http://www.javagrande.org/reports.htm

Java Grande Forum

Overview of Document

This document represents an update to the Java Grande Report, which was presented to Sun Microsystems at a special panel session held at SC 98 in Orlando during November 1998. The purpose of the Java Grande Report was to 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 Java Grande Forum is pleased to report in this update that significant progress has been made in discussions with Sun, particularly in addressing the numerics issues.

Section 1: Recent Activity of the Java Grande Forum

The highly successful Java Grande panel at SC 99 has resulted in a number of activities being coordinated throughout the world. JGF participants have organized workshops and tutorials at several international conferences, and a new European funded initiative is under way. This section will present the list of known events and a discussion of the details of this initiative.

Section 2: About the Java Grande Forum and Grande Applications

This section re-introduces 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 3: Concurrency and Applications Working Group Status

The Concurrency/Applications Working Group of the Java Grande Forum addresses the suitability of Java for concurrent (parallel and distributed) applications. In its initial assessment of Java, the working group has focused on several areas where improvements and further investigation are both needed: (a) Remote Method Invocation, (b) Object Serialization, and (c) Benchmarking JVM Performance. The working group also identified several areas for continued investigation: (a) seamless (desktop) access to high-performance computers and (b) the availability of MPI-like message passing facilities for programming networks of computers or parallel machines more naturally.

Section 4: Numerics Working Group Update

The Numerics Working Group 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.

1 Recent Activity of the Java Grande Forum

The highly successful Java Grande panel at SC 99 has resulted in a number of activities being coordinated throughout the world. JGF participants have organized workshops and tutorials at several international conferences, and a new European funded initiative is under way.

Recent Conferences

JGF participants have organized successful workshops and the first ACM conference on Java in both the United States and Europe. These conference have been widely attended and have featured cutting-edge technical papers and tutorials on how to apply Java to computational science.

International Conference on Supercomputing, Workshop on Java, Rhodes, Greece, 20 June 1999

- http://pylos.csrd.uiuc.edu/ics99/

ACM Java Grande 1999, 12-14 June 1999, San Francisco, California

- http://www.cs.ucsb.edu/conferences/java99/

JavaOne, 14-17 June 1999, San Francisco, California

- http://www.javasoft.com/javaone/

HPCN-Europe, Workshop on Java in High-Performance Computing, 12-14 April 1999, Amsterdam, Netherlands

- http://perun.hscs.wmin.ac.uk/JHPC/

First UK Workshop on Java for High-Performance Network Computing, Euro-Par 98, 2-3 September, Southampton, UK

- http://www.cs.cf.ac.uk/hpjworkshop/

Mannheim Supercomputing Conference, 14 June 1999

- http://www.supercomp.de/

IPPS/SPDP 1999 International Workshop on Java for Parallel and Distributed Computing, 12-14 April 1999.

- http://www.ippsxx.org

European Java Grande Forum

The objective of the forum is to bring together an interdisciplinary group of researchers from academia and industry to promote the use of Java as the preferred language for high performance computing. The forum will complement and collaborate with the highly successful US JavaGrande and ensure that Europe plays a significant part in advancing Java for HPC. This European forum has two main working groups: Metacomputing and Interoperability, and HPC Applications and Tools.

The European JavaGrande Forum is currently sponsored by the European funded networking project EuroTools. A separate funding proposal is in preparation and will be submitted to the latest European funding proposal.

EU JavaGrande -- http://www.irisa.fr/EuroTools/Sigs/JavaGrande.top.html

EuroTools -- http://www.irisa.fr/EuroTools/

2 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:** Data mining, 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 gkt@cs.depaul.edu 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.

3 Concurrency and Applications Working Group Status/Update

The Concurrency and Applications Working Group (hereafter ACG) has focused on the following issues:

- Critical JDK issues
- Benchmarks
- Message passing
- Desktop Access to Remote Resources (now called the Computing Portal Working Group)
- Other parallel and distributed computing issues

After the public dissemination of the Java Grande report at Supercomputing in Orlando, progress has been made in most of these areas.

Critical JDK issues

Performance of Object Serialization

In the Orlando report, the performance problems of object serialization have been discussed thoroughly. Object serialization is the basic mechanisms that is used by Java's RMI (Remote Method Invocation) to convert Java objects into a byte stream representation and vice versa. Hence, the performance of object serialization is crucial for Grande applications that need distributed parallel machines.

The JavaParty group of the University of Karlsruhe, Germany, has designed and implemented a drop-in replacement for the JDK-serialization, called UKA-serialization, that can be used in situations where persistence is not an issue. The UKA-serialization is written in Java and reduces the serialization overhead by 81% to 97%.

Most of the individual show-stoppers mentioned in the Orlando report have been solved in that implementation. For example, the UKA-serialization uses explicit marshaling instead of reflection, it uses a slim type encoding, it adds an additional type of reset-operation that avoids costly retransmission of type information, and it implements a more efficient buffer handling.

More information on the UKA-serialization can be found at http://wwwipd.ira.uka.de/JavaParty/ and in Bernhard Haumacher and Michael Philippsen. More Efficient Object Serialization. In Parallel and Distributed Processing, LNCS 1586, pp. 718-732, International Workshop on Java for Parallel and Distributed Computing, San Juan, Puerto Rico, April 12, 1999.

The JavaGrande forum has discussed the central ideas of the UKA-serialization with Sun's RMI group in Burlington. It seems likely that some of the ideas will be incorporated in future releases of the JDK. In particular, it is likely that there will be a way to switch to a slim type encoding.

Performance of RMI

In the Orlando report the performance problems and other difficulties of RMI have been discussed in depth. Although Grande applications that require parallel computers need an efficient use of non-TCP/IP communication hardware, the current RMI does not provide any support. Moreover, since the current RMI has been designed and implemented with rather slow internet-connections in mind, the design includes many mechanisms that deal with instable or slow networks.

For Grande applications, access of high-performance networks is essential. Moreover, in closely connected cluster settings, a lot of RMI's overhead can be traded for speed.

The JavaParty group of the University of Karlsruhe, Germany, has designed and implemented a drop-in replacement for the RMI, called KaRMI, that can be used in situations where persistence and TCP/IP-addressing mechanisms are not an issue. KaRMI is completely written in Java. In combination with the UKA-serialization, KaRMI saves up to 71% of the time needed for a remote method invocation between two PCs connected by Ethernet. Between two Dec Alphas connected by a Myrinet-based ParaStation network, a remote method invocation can be executed within 80 mirco seconds.

In addition to being as compatible as allowed by the requirement of support for non-TCP/IP networks, KaRMI comes with simple and documented interfaces. Several transport technologies can be mixed, an optimized distributed garbage collector can be plugged in which handles combinations of different communication technologies.

More information on the KaRMI can be found at http://wwwipd.ira.uka.de/JavaParty/ and in Christian Nester, Michael Philippsen, and Bernhard Haumacher A More Efficient RMI. In Proc. ACM 1999 Java Grande Conference, San Francisco, June 12-13, 1999.

The JavaGrande forum has discussed the central ideas of KaRMI with Sun's RMI group in Burlington. It is unclear whether any improvements will make it into the official RMI implementation.

The following issues are still open:

- Still needed are mechanisms for a closer interaction with Java's thread scheduler, especially if the communication hardware operates in user level.
- The RMI source code is still unavailable.
- The internal layers of RMI are still undocumented.
- It would be interesting to see, whether the improved RMI could used by IBM's San Francisco project. This project, unfortunately, uses internal and undocumented classes of the RMI implementation.
- Class Compatibility Problem, see Orlando report
- Dynamic Class Loading from Remote Hosts, see Orlando report

Other papers that contain information relevant to Java Grande are:

Ronald Veldema, Rob van Nieuwport, Jason Maassen, Henri E. Bal, and Aske Plaat. Efficient Remote Method Invocation, Technical Report IR-450, Vrije Universiteit, Amsterdam, The Netherlands, September 1998.

Chi-Chao Chang and Thorsten von Eicken. Interfacing Java with the Virtual Interface Architecture. In Proc. ACM 1999 Java Grande Conference, San Francisco, June 12-13, 1999.

G. K. Thiruvathukal, L.S. Thomas, A.T. Korczynski, Reflective Remote Method Invocation. In Proc. ACM 1999 Java Grande Conference, San Francisco, June 12-13, 1999.

Benchmarks

EPCC at the University of Edinburgh has been coordinating the construction of a Java Grande Forum Benchmarking suite. The purpose of the suite is to provide ways of measuring and comparing alternative Java execution environments in ways which are important to Grande applications. In order to provide standard measurements and reporting we wrote an instrumentation class which is used throughout the suite and we plan to propose this class for registration as a Java standard.

The serial suite is now on version 2.0 and almost complete and we have added code to calculate a normalized score for each section and to present the results in a Web page. The benchmarking group has achieved a 1 gigaflop performance using parallel Java Threads on benchmark codes.

The contents of the suite are as follows:

Section 1: Low level operations - measuring the performance of low level operations such as arithmetic and math library operations, garbage collection, method calls and casting.

Section 2: Kernels - short codes which carry out specific operations frequently used in grande applications.

Section 3: Large scale applications - real grande codes, possibly less useful for comparative performance studies but worthwhile to demonstrate the potential of Java for tackling real problems.

Section 1: Low Level Operations Arith: execution of arithmetic operations Assign: variable assignment Cast: casting Create: creating objects and arrays Loop: Loop overheads Math: execution of maths library operations Method: method invocation Serial: Serialisation Exception Exception handling

Section 2: Kernels Series: Fourier coefficient analysis LUFact: LU Factorisation SOR: Successive over-relaxation HeapSort: Integer sorting Crypt: IDEA encryption FFT: FFT Sparse: Sparse Matrix multiplication

Section 3: Large Scale Applications Search: Alpha-beta pruned search Euler: Computational Fluid Dynamics MD: Molecular Dynamics simulation MC: Monte Carlo simulation Ray Tracer: 3D Ray Tracer

The next phase of the Benchmarking work will be to develop complementary parallel benchmarks. Initially we aim to concentrate on Threads and MPI tests and we have a series of benchmarks planned which are based around the NAS codes and parallelisations of the existing applications. Serial versions of these codes will be available for comparison purposes.

The new release of the benchmark suite along with results is available from:

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

These pages contain detailed results for a number of systems and a league table of scores. We would like to invite members of the community to download and run the benchmarks on their platforms and contribute the results to this growing database. Anyone with results, codes or comments to contribute can contact us at epcc-javagrande@epcc.ed.ac.uk.

The JavaParty group at the University of Karlsruhe, Germany, has put together an RMI benchmark. The collection is available from http://wwwipd.ira.uka.de/JavaParty

The Numerics Working Group has put together the Scimark 2 benchmark. See Section 4.2 for details.

We would like to thank the following people for their contributions to the benchmark suite: Jesudas Mathew, Paul Coddington, Roldan Pozo, Gabriel Zachmann, David Oh, Reed Wade, John Tromp, Florian Doyon, Wilfried Klauser, Hon Yau.

Message Passing

Introduction

The Message-Passing group is a community effort within the frame of the JavaGrande Forum activities. The aim of the group is to provide an environment for discussion and collaboration in the research and development of portable message-passing frameworks and corresponding APIs for the Java programming language with focus on JavaGrande applications. The group intends to forward any draft standard proposals (e.g. Java-MPI) whenever they are ready to the MPI forum for the official standardization procedure. Important current objectives of the group are

- the development of an MPI-like API for Java;
- reserach into O-O Java-centric message-passing environments;
- performance evaluation and comparisons between different frameworks and implementations.

MPI-like API for Java

A mailing list has been created for discussions in this area with around 50 subscribers at present. java-mpi@npac.syr.edu is an open e-mail reflector for people interested in developing or using Java bindings to the Message Passing Interface standard MPI. Subscribing to the e-mail list is as simple as sending the command

“subscribe java-mpi”

in the body of an email addressed to Majordomo@npac.syr.edu.

The topics for discussion include:

- Issues relating to agreement on a common Java API for MPI (a draft proposal can be found in the appendix of the “MPI for Java” position document at “www.java-grande.org/reports.htm”).
- Issues about implementing MPI functionality in Java. Implementations can be 100% Java, or Java wrappers to native MPI implementations.
- User’s experiences developing parallel application codes in Java, using an MPI binding. Experiences using Java-MPI in Grande applications as well as a vehicle for teaching message-passing parallel programming are very relevant.
- Parallel benchmark codes written in Java, using an MPI binding and results of running such benchmarks.

Object-oriented Java-centric message-passing environments

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, drawing on experience from distributed computing real-time activities.

Performance evaluation and comparisons

There is a growing number of parallel benchmark codes written in Java, using an MPI-like binding. This effort involves using and further developing these benchmarks for validation, evaluation, and comparison of MPI-like environments for Java. We also focus on both methodology and evaluation guidelines in order to ensure reproducibility, sound interpretation, and comparative analysis of performance results. The codes currently available include:

- A test suite for MPI-like bindings for Java which covers all 128 MPI-1 functions. This task was completed at NPAC.
- Low-level message-passing benchmarks, which we have used to evaluate our existing prototype bindings on different platforms. Some of the participating organizations in this work are Brigham Young University, Emory University, NPAC, University of Portsmouth, University of Westminster, and others.
- Draft Java and MPI-like versions of the NAS Parallel Benchmarks IS, EP, and FT have been developed as a result of collaboration between San Diego Supercomputer Center and University of Westminster. NASA-Ames have recently allocated resources to join and lead the completion of a full set of NPB kernels in Java-MPI.

All message passing benchmarks in Java are publicly available from the web-site of the Message Passing Group.

Computing Portal Working Group (formerly Datorr)

This Computing Portal Working Group (http://www-fp.mcs.anl.gov/~gregor/datorr) is a joint activity of the JavaGrande Forum and the newly formed Gridforum (http://www.gridforum.org/) project, which links the major high performance computing activities building the infrastructure for computational grids. This working group was originally called Datorr with this acronym denoting Desktop Access to Remote Resources. The new terminology more clearly describes the mission of this group and relates it to the exploding field of enterprise information portals (as described by Merrill Lynch in http://www.sagemaker.com/company/lynch.htm). Further computing portal research activities have been launched by both NASA's Information Power Grid (http://science.nas.nasa.gov/IPG) and the NSF partnership in computational infrastructure centered at Illinois (The NCSA Alliance http://www.ncsa.uiuc.edu/alliance/). We will leverage these and other projects in our further work. We also hope that the more descriptive name will encourage broad involvement in our activities and welcome new participants.

The previous mission and activities of the Datorr working group will continue unchanged with the new name. Since our last report at the end of 1998, we held a good meeting on February 15th and 16th, 1999 at Sandia National Laboratory. Judy Beringer of Sandia National, and Gregor von Laszewski of Argonne National Laboratory organized the meeting. The technical preparation for the meeting was steered by Geoffrey C. Fox, Dennis Gannon, Piyush Mehotra, and Gregor von Laszewski. The meeting had over 25 participants with in particular contributions from the Gateway, Hotpage, Legion, NetSolve, Ninf, Ninja, and Unicore groups. The meeting broke up into two working groups discussing roughly the user/service and service backend resource interfaces respectively. In the latter case we identified the importance of the use of XML in describing computing related resources. We targeted a demonstration of the use of this technology with a simple web based interface to computing resources by SC99 in November 99. The first group continued the definition of the user view and task object started at our first meeting. As a follow-up, the Globus team has been very active in the investigation of the use of XML as one of the core technologies for interchanging information between different components of the Globus and the portal architecture. They have demonstrated that XML can be used as a language to describe components, jobs and tasks. Furthermore, they showed that the information expressed through the Grid Metacomputing Directory Service can be exposed as objects defined in XML.

In defining the proper institutional implementation of the Computing Portal working group, we are pleased that the voluntary contributions by the Globus project allowing Gregor von Laszewski, Argonne National Laboratory, to work partially in this effort will be continued and enhanced with other NCSA Alliance resources. We discussed the architecture defined by the Datorr group at the Alliance meeting in Chicago in May and found widespread approval. Current Alliance plans for developing application specific computing portals was substantially built on the work of Datorr and we expect continued close collaboration. We are in the process of forming subgroups to work on selected common portal architecture requirements and implementations.

Gregor von Laszewski will continue to serve as secretary for the working group and will organize future workshops. These Computing Portal workshops will be coordinated with the Gridforum and the JavaGrande Forum in order to leverage one another's efforts. The working group will register the domain name www.computingportal.org. During the month of July, the current Datorr web pages (http://www-fp.mcs.anl.gov/~gregor/datorr) will be transferred to the new web site, which will be hosted at Argonne National Laboratory. The mailing list datorr@mcs.anl.gov will temporarily continue to exist until the move to the new domain name is completed. Once completed, a new set of mailing lists will be created. We intend to schedule the next Workshop alongside Grid forum and JavaGrande meetings.

4 Numerics Working Group Update

Contents

1 Introduction

If Java[TM] 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 (JGF) 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.

For detailed information about the work of the Java Grande Forum, see its Web pages at http://www.javagrande.org/. The work of the JGNWG is described in the Java Numerics Web pages maintained at NIST, http://math.nist.gov/javanumerics/.

This report constitutes an update on JGNWG activities since the JGF's first detailed report, *Making Java Work for High-End Computing*, was issued in November 1998. This report is available in PDF form at http://www.javagrande.org/sc98/sc98grande.pdf. The sections relating to the JGNWG can also be viewed in HTML form at http://math.nist.gov/javanumerics/jgfnwg-01.html. We assume that the reader is familiar with the contents of these reports.

2 Recent Accomplishments

2.1 Floating-point Semantics in Java 2

The report of the Java Grande Forum was influential in preventing the adoption of the Proposal for Extension of Java Floating-Point Semantics (Revision 1) released by Sun in May 1998. Instead, for Java 1.2 Sun adopted the key ideas of the Java Grande counterproposal. The principal intent of these proposals was to improve the performance of Java floating-point on processors like the Intel Pentium (i.e. those with the x86 architecture), whose registers operate using IEEE 754's 80-bit double-extended format. The original Java floating-point semantics lead to a 2- to 10-fold performance slowdown when implemented on such processors.

The JGNWG counterproposal suggested the adoption of two floating-point modes (`strictfp` and default). (The `widefp` keyword of the original Sun proposal is eliminated.) `strictfp` mode, which applies to classes or methods with the `strictfp` keyword, corresponds to the original (Java 1) floating-point semantics. The goal in this case is to provide bit-for-bit reproducibility of Java class files across JVMs. Default mode relaxes these requirements slightly. In default mode, anonymous doubles may be represented using 15-bit exponents (rather than standard 11-bit exponents). The x86 has a control bit which causes the fractional part of floating-point results to be truncated to 52 bits (double precision) after each operation without any performance degradation. However, the floating-point exponent remains at 15 bits. By admitting 15-bit exponents, artificial store/load to memory to adjust the exponent can be avoided. This minor change also leads to floating-point results that differ only rarely among processors. Difference in observed results between the x86 and SPARC processors, for example, would occur only in cases where the SPARC would overflow and the x86 would not. A similar design is used for floats.

The JGNWG requirement that the use of extended exponents be used consistently (i.e. either for all anonymous variables or for none) was not adopted for Java 2. While Sun believes that the improved predictability of such a requirement would be beneficial, it would require adding new typed pop/swap/dup JVM instructions to manipulate such numbers on the stack. As such, it was deemed too significant a change to be considered at this time.

Similarly, JGNWG proposals to admit use of floating-multiply-accumulate (FMA) instructions and the associative law in optimizing expressions were not adopted. It was felt that more study to understand their effects more clearly before they are considered. The JGNWG has been considering how to frame a proposal for use of FMAs (see Section 3.1).

2.2 Endorsement by IFIP Working Group 2.5

The work of the JGNWG has been endorsed by the International Federation for Information Processing Working Group 2.5 (WG2.5). WG2.5, whose charter is to improve the quality of numerical computation by promoting the development and availability of sound numerical software, reports to the IFIP Technical Committee on Programming Languages (TC2). At its May 1999 meeting at Purdue University, WG2.5 passed the following resolution:

> *WG2.5 supports the efforts of the Numerics Working Group (NWG) of the Java Grande Forum to represent the interests of the numerical software community in the evolution of the Java language and its environment. WG2.5 intends to trace NWG activities and continue to provide input to the recommendations of the NWG.*

2.3 Working Group Meetings

The last open working meeting of the JGNWG was held in August 1998 in Palo Alto, CA. In March 1999 a group of JGNWG members visited Sun in Menlo Park, CA to present the report of the JGNWG to key Java developers. Representing the working group were Ron Boisvert (NIST), John Brophy (Visual Numerics), Shah Datardina (NAG), Jack Dongarra (University of Tennessee at Knoxville and Oak Ridge National Labs), Geoffrey Fox (Syracuse University), Siamak Hassanzadeh (Sun), Cleve Moler (The Mathworks), Jose Moreira (IBM), and Roldan Pozo (NIST). Representing Sun were James Gosling, Tim Lindholm, Joshua Bloch, Henry Sowizral, Thomas Arkwright, David Hough, and Greg Tarsey.

At the March meeting, the JGNWG expressed its appreciation to Sun for taking the time to exchange information on progress and plans for improving Java's usability for numerical computations common to most Grande applications. Tim Lindholm said that the JGNWG was providing valuable input to Sun and that its work had already had a significant effect on Java. Sun realizes that although scientific and engineering computation is not Java's primary market, it represents an important constituency that they would like to accommodate. Sun is happy to cooperate with the Java Grande Forum and will seek their advice on matters relating to Java Numerics. They appreciate the Forum's willingness to find compromise solutions that preserve Java's overall design and philosophy. (See the note from Tim Lindholm in the Appendix to this document.) At this meeting, Sun indicated that it would reassign a staff member to spend full time working on Java Numerics issues with the Java inner circle.

A half-day open meeting of the JGNWG is scheduled in conjunction with the ACM 1999 Java Grande Conference to be held in San Francisco on June 12-14.

2.4 Other Events

The work of the JGNWG was described at the following events.

Java Grande Panel, *SC'98*, Orlando, FL, November 13, 1998 (R. Boisvert, C. Moler, M. Snir)
International Workshop on Java for Parallel and Distributed Computing, *IPPS/SPDP* 1999, San Juan, Puerto Rico (G. Fox)
IEEE Frontier's 1999 Conference, Annapolis, MD, February 25, 1999 (G. Fox, M. Snir, R. Pozo)
SIAM National Meeting, Atlanta, GA, May 14, 1999 (R. Pozo, C. Moler, T. Bryan, J. Brophy)
IFIP Working Group 2.5 Meeting, Purdue University, May 20, 1999 (R. Boisvert)

3 Status of Proposals

3.1 Use of Floating Multiply-Accumulate (FMA)

Sun appears willing to consider allowing use of the hardware fused-multiply-accumulate (FMA), instruction under carefully controlled circumstances. The FMA operation, y ß y + a*x, often appears in the inner loops of dense matrix computations. As a result, significant speedups can occur when hardware FMAs are used. Processors with FMAs include the Power PC and the Merced. IBM has analyzed the impact of using the FMA operation in several Java benchmarks. For matrix-multiply (real numbers), performance can be improved from 100.9 Mflops to 209.8 Mflops through the use of FMAs. For BSOM (a neural-network data mining kernel) the improvement is from 94.5 Mflops to 120.5 Mflops. For Cholesky factorization, the improvement is from 83.4 to 129.9 Mflops. These FMAs were all obtained from explicit `a*b+c` operations. The Fortran numbers for MATMUL (matrix multiply), BSOM, and Cholesky are 244.6, 28.0, and 134.0, respectively. (All experiments were performed on an RS/6000 model 590.)

This topic was discussed extensively at the March meeting in Menlo Park. Marc Snir of IBM has developed a proposal to modify the semantics of Java floating-point in order to admit FMAs. It is included as Appendix 2.

Another way to provide access to FMAs is to provide explicit methods that represent them. Two separate FMA methods could be added to Java: `java.lang.Math.fma()` and `java.lang.StrictMath.fma()`. `fma(y,a,x)` would return `a*x+y` in each case. Its behavior would be different depending on the package.

StrictMath **Forced FMA.** Requires use of extended precision FMA, even if it requires simulation in software. That is: `y, a, x` are each converted to IEEE extended format (double or float as appropriate), the computation occurs in extended format, with the result rounded to double or float as appropriate.

Math **Opportunistic FMA.** The FMA should be used if it can be done fast, e.g. as a single hardware instruction, otherwise the expression `a*x+y` should be computed using the usual Java floating-point semantics.

These methods should not be incorporated in place of the proposed extensions above, however. Rather, they should be included to provide an additional method of fine control of FMA operations.

3.2 Math Library Specification

The original specification for `java.lang.Math` says, roughly, to use the algorithms of fdlibm (Sun's freely distributable libm, available on netlib) translated straightforwardly to Java. It is not clear whom, if anyone, does this. In practice, Java programs that reference elementary functions are not producing the same results on all Java platforms. For the Java 2 platform, Sun is distributing Java wrappers for fdlibm in C for `java.lang.Math`. This should encourage more uniform results from the Java elementary functions.

The JGNWG has had extensive discussions regarding the proper specification of the elementary functions in Java. Cleve Moler has studied the various the issues; his report is included as Appendix 3.

Two recent contributions point to new possibilities. John Brophy of Visual Numerics announced the release of Jmath, a complete translation of Sun's fdlibm in Java. This is posted at http://www.vni.com/corner/garage/grande/. The use of such a library would insure reproducible results.

Abraham Ziv and colleagues at the IBM Haifa Labs recently released a suite of correctly rounded math functions for IEEE arithmetic in C[1]. (Only binaries are released, for Solaris, NT and AIX. See http://www.alphaWorks.ibm.com/tech/mathlibrary4java/.) They have recommended its use for Java. Providing correctly rounded results in all cases is, of course, the ideal specification for the elementary functions. Of high concern to Sun is the speed of the library and its footprint (size of the distributable binary). Users will not tolerate a significant degradation of speed, and Sun does not want to significantly increase the size of the Java core. Ziv claims that the Haifa codes are usually significantly faster than fdlibm. The size of the binaries that would be included with Java are hard to judge. The compressed AIX binary is 185Kb. The codes include large tables, but their sizes can be adjusted. (The source form of the atan table is 250Kb, for example, although it is shared with atan2.) Further analysis is needed to assess the size in a Java distribution. In addition, versions for other platforms would have to be developed. An alternative would be to provide a version of the Haifa library coded in Java.

The JGNWG also sees merit in relaxing the strict specification for the elementary functions in default mode to admit use of hardware functions such as `FSIN` and `FSQRT` on the Pentium.

1. See A. Ziv, Fast evaluation of elementary mathematical functions with correctly rounded last bit, *ACM Transactions on Mathematical Software* 17 (1991), pp. 410-423. http://www.acm.org/pubs/citations/journals/toms/1991-17-3/p410-ziv/

The following are possible specifications for the Java elementary functions.

`strictfp` mode: produce the correctly rounded result.

Pros This is the ideal definition.

Cons Pure Java code for this does not yet exist. Existing algorithms may be too large for inclusion in the Java core.

`strictfp` mode: mandate use of VNI's native Java fdlibm.

Pros This code is available now. It is reasonably fast and has a small footprint. It produces results that are usually within one unit in the last place of the correctly rounded result.

Cons An operational definition like this would mandate incorrect results. Incorporating future improvements in the algorithms yielding better results would change the behavior of existing Java class files.

default mode: specify largest acceptable relative error in result for each function

Pros This allows flexibility of implementation. It allows (guarded) use of specialized hardware instructions for square root, sine, cosine, etc., yielding greatly improved performance. It allows improvements in algorithms producing more accurate results to be trivially accommodated

Cons Verifying conformance to a tight bound (e.g., 1 ulp) is quite difficult.

3.3 Operator Overloading and Lightweight Objects

Sun clearly sees the need for these extensions for the numerics community. The performance and usability of core math capabilities such as complex arithmetic, interval arithmetic, multidimensional arrays, and others hinge on these capabilities. IBM has done experiments on supporting lightweight objects. Representing complex numbers as full-fledged objects results in a performance of 1.1 Mflops for matrix-multiply and 1.7 Mflops for a CFD kernel (two-dimensional convolution). Using lightweight objects and extensive compiler optimization raises those numbers to 89.5 Mflops (MATMUL) and 60.5 Mflops (CFD). Fortran numbers are 136.4 and 66.5 Mflops, respectively. (All experiments performed on an RS/6000 model 590.)

Sun has little experience in the community process for making changes to the Java language itself. As a result, Gosling suggested that Sun should take the lead in developing proposals for operator overloading and lightweight classes. Sun agreed to reassign a staff member to work with Gosling and colleagues to develop such a proposal. The JGF would help in the proposal process, providing justification of the need, providing comments, etc. It was suggested that finding allies in other user communities might help.

Some worries were expressed about the reaction of "purists" to a proposal for operator overloading. Ideally, one would like to see mechanisms that led to only reasonable usage. (Examples: use descriptive names for overloaded methods; require implementation of an arithmetic interface.) It does not seem possible to stop the truly perverse, however.

Lightweight objects are more problematical. There are many ways to provide such a facility, and these would have to be fleshed out. Lightweight objects can be first introduced in the Java language with little or no change to the VM. However, extensive changes to the VM may be necessary to actually deliver performance benefits. (A major problem is how to return lightweight objects from a method.)

The timing for such proposals was discussed. It is unlikely that major language changes will be on the table for the next few releases, so this is viewed as a longer-term project. It was also agreed that the proposals for operator overloading and lightweight classes should be unbundled in order to maximize the chances for success with "the process".

4 Related Activities

4.1 Relationship with vecmath

`javax.vecmath` is a package that provides a variety of low level mathematical utilities of use in computer graphics. Examples include multiplication of 2x2, 3x3 and 4x4 matrices. Also included in a class `Gmatrix` that implements operations on general nxn matrices. Methods for computing LU, Cholesky and singular value decompositions are included.

Some preliminary testing by JGNWG members indicates that `vecmath` may not be completely debugged. On computing the singular value decomposition (SVD) of a 3x3 matrix of zeros `vecmath` returned left and right singular vectors of all `NaN`s. The matrix of singular values was unchanged from its value on input. In another test, the SVD of the rank 2 matrix with rows (1 2 3), (4 5 6), (7 8 9) was computed. The computed singular vectors were correct, but the matrix of singular values was unchanged from its input state, and the rank was reported incorrectly as 3.

In discussions with Henry Sowizral of Sun it was agreed that the numerical analysis community should participate more actively in the development of APIs for linear algebra. One option would be for `Gmatrix` to be deprecated in `vecmath` and such matrix operations be provided instead in a separate class for numerical linear algebra which could be shepherded by the JGNWG. There will be a "call for experts" soon to consider extensions to `vecmath`. The JGNWG agreed to provide representatives to this team to help work out details for future development of this package.

4.2 SciMark 2 Benchmark

NIST recently released Version 2 of its SciMark benchmark for Java. SciMark is a composite Java benchmark measuring the performance of numerical kernels occurring in scientific and engineering applications. It consists of five kernels that typify computations commonly found in numeric codes: FFT, Gauss-Seidel relaxation, sparse matrix-multiply, Monte Carlo integration, and dense LU factorization.

These kernels are chosen to provide an indication of how well the underlying JVM/JITs perform on applications utilizing these types of algorithms. The problem sizes are purposely chosen to be small in order to isolate the effects of memory hierarchy and focus on internal JVM/JIT and CPU issues. Results in SciMark 2 are recorded in megaflops for ease of interpretation (although this makes them incompatible with SciMark 1.) A larger version of the benchmark (SciMark 2.0 LARGE) addresses performance of the memory subsystem with out-of-cache problem sizes. The source of the SciMark benchmark is available, and implementations in C and Fortran will be made available to admit comparisons with these language processors.

The benchmark may be downloaded as an applet and run in any Java environment. A table of submitted benchmark data is maintained on the SciMark Web page at http://math.nist.gov/scimark.

4.3 Java Preprocessor Admitting Complex Type

The JavaParty group at the University of Karlsruhe, Germany, has developed a preprocessor that adds a new basic type complex to Java and maps the resulting code back to pure Java.

Compared to code that uses complex objects and method invocations to express arithmetic operations the new basic type increases readability and it is also executed faster. On average, the versions of our benchmark programs that use the basic type outperform the class-based versions by a factor of 2 up to 21 (depending on the JVM used).

More information can be found at http://wwwipd.ira.uka.de/JavaParty/.

Appendix 1

Subject: Feedback for the Numerics Working Group
Date: Fri, 12 Mar 1999 15:59:40 -0800 (PST)
From: Tim Lindholm <Timothy.Lindholm@Eng.Sun.COM>
To: boisvert@cam.nist.gov

Hi Ron,

The following gives my perspective on the Java Grande Numerics Working Group and its relationship and relevance to Sun in Sun's role as the steward of Java development.

As you know, I'm a Distinguished Engineer at Sun, one of the members of the original Java project, the author of The Java Virtual Machine Specification, and currently one of the architects of the Java 2 platform. I work closely with the other architects of the Java technologies, such as James Gosling, and while I can't speak for them in detail I can say that the opinions I express below are not out of line with theirs.

During the development of the Java 2 platform version 1.2 (formerly known as JDK 1.2), I was handed responsiblity for creating licensee and industry consensus around changes to Java's primitive floating-point arithmetic targeted at improving performance on Intel CPUs. This was an extremely difficult task because it required careful attention to the balance between many factors, and was being done in an extremely charged political environment. We who were responsible did not have a broad background in numerics, and had not been successful finding help within Sun. We understood that there was high risk: Java had taken a rather different approach to floating point than many other languages, and the wrong decision in 1.2 could throw away many of the advantages of that approach.

Our best attempts led to a public proposal that we considered a bad compromise and were not happy with, but were resigned to. At this point the Numerics Working Group wrote a counterproposal to Sun's public proposal that gave new technical insight on how to balance the demands of performance on Intel without throwing away the important part of reproducibility. The counterproposal was both very sensitive to the spirit of Java and satisfactory as a solution for the performance problem. When we saw the new proposal we revived efforts to reach a better answer.

We were subsequently aided by email and phone calls with a number of members on the Numerics Working Group (you, Roldan, and Cleve). Joe Darcy, one of the authors of the counterproposal, helped us to understand the proposal generally, and specifically to evaluate the effects of modifications we required of the counterproposal. All of you helped us understand how the Numerics Working Group represented a number of rather different fields with different perspectives on numerics. This helped us gain confidence that the new solution reflected a consensus that would satisfy a broad range of Java users.

We are sure that we ended up with a better answer for 1.2, and arrived at it through more complete consideration of real issues, because of the efforts of the Numerics Working Group. We have internally resolved to consult with the Group on future numeric issues, and to do so earlier in the process. Our attendance at the 3/11 Numerics Working Group meeting at Sun and all the work we accomplished there is evidence of this resolution. We think that the Group continues to show great sensitivity to the needs of the Java platform and the difficulty of introducing change while preserving compatibility and robustness. We look forward to continuing this relationship into the future.

```
-- Tim
```

Appendix 2: Proposal for Use of FMAs in Java (Draft)

Marc Snir
IBM T.J. Watson Research Center
snir@us.ibm.com

The current JVM 1.2 spec supports two double formats: IEEE format with 53 bit mantissa, 12 bit exponent; and a default format, that has exactly the same mantissa, but may have a larger exponent. Internal values (not visible to the user) can be held in the default format, and converted to the IEEE format when assigned to program variables. If a method is declared strictfp, then the use of large exponents is disallowed. A similar design is used for floats. This design enables Intel to take (partial) advantage of its 80 bit floating point registers, using the mode where only 53 bit mantissas are used in floating point operations. It does not allow Power PC to take advantage of the fused multiply add, and does not allow the use of full sized mantissas in the 80 bit Intel floating point registers.

The JGNWG has discussed several proposals to remediate this problem. To be adopted, a proposal should

a) require limited implementation effort, and negligible implementation effort on platforms that are not affected by current Java floating point restrictions. ,

b) introduce only limited platform dependence (similar to the platform dependence due to the default/strictfp design),

c) not worsen accuracy, and

d) improve performance of codes without required significant, platform-dependent, code rewrite.

The proposal outlined below seems to achieve these four goals. An "extended" floating point mode is added, where larger mantissas are allowed, in the same circumstances where larger exponents are allowed in JVM 1.2: internal values can be stored with more digits of precision than required by IEEE format, but need be converted back to IEEE format when assigned to program variables. This solution would allow (i) the use of fused multiply adds, where the intermediary output from the multiplication is not rounded, (ii) the promotion of floats to doubles, on machines where double arithmetic is faster, and (iii) the use of full 80 bit precision on Intel machines – at least for internal values.

We can have two versions of this design. V1: The use of the "extended" format would be allowed only for methods (and classes) that were explicitly tagged with the keyword `extendedfp` in the source. The byte file will contain an `extendedfp` flag. Otherwise, the rules of JVM 1.2 apply. We then have two floating-point modes: `strictfp` and `extendedfp` (in addition to the default mode). V2: "extended" format can be used in default mode – no new keyword.

Note that the user can always prohibit the use of extended format by explicitly naming each internal value (replace `x=a*b+c` with `x1=a*b; x=x1+c`), by using `strictfp`, or, if V1 is adopted, just by avoiding the use of the extendedfp attribute. Note, too, that the implementation effort is minimal, for implementations that do not take advantage of the extended format: the Java compilers need generate a new flag, but this flag may be ignored by Java interpreters and JITs.

A downside of this design is that it is unlikely that Sun will add new keywords and new floating-point modes. Thus, either (i) we agree upfront what `extendedfp` entails (does it allow usage of associativity, as proposed with `associativefp`?), or (ii) we agree that `extendedfp` may add new relaxations, over time, or (iii) we agree that none will be added. (i) is unlikely, due to short time.

APPENDIX 3: The Java Elementary Functions

Cleve Moler
The MathWorks
moler@mathworks.com

An important design goal of Java has been machine independence. A single program should run on a wide range of computers, and get the same result everywhere. But, as Java matures and becomes widely used for a broad range of applications, the difficulty of achieving this goal in practice is becoming apparent.

The machine independence goal conflicts with execution speed. Keeping intermediate quantities in extended precision floating point registers increases speed, but may produce different results and lead to different exceptions on different machines.

Tradeoffs between machine independence and speed become particularly important in the design of a library for the elementary math functions, square root, exponential, logarithm, power, and a half dozen trig functions.

These fundamental tradeoff considerations are confounded in at least one currently available Java math library by serious design flaws. The exponential, sine and cosine functions in Microsoft's math library are so inaccurate for large arguments that the library is unsuitable for general-purpose use.

FDLIBM

The documentation of `Java.lang.Math` in the Java platform 1.2 Beta 4 API specification says:

> *To help ensure portability of Java programs, the definitions of many of the numeric functions in this package require that they produce the same results as certain published algorithms. These algorithms are available from the well-known network library netlib as the package "Freely Distributable Math Library" (FDLIBM). These algorithms, which are written in the C programming language, are then to be understood as executed with all floating-point operations following the rules of Java floating-point arithmetic.*

However, very few people are paying any attention to this specification. There are two widely used Java environments for Microsoft operating systems on Intel PCs, one from Microsoft and one from Sun. Neither one of them uses currently FDLIM; both use the "libm" from Visual C++. As a consequence, Java numerical results on the PC may not be the same as those obtained on a Sun SPARC, and, worse yet, may be inaccurate or completely wrong.

FDLIBM deserves to be better known and more widely used. It is a very valuable asset. Authored by K. C. Ng, David Hough and possibly others at Sun, it is essentially a publicly available version of Sun's elementary function library. See http://www.netlib.org/fdlibm.

(At the MathWorks, we use FDLIBM as the underlying library for MATLAB on all platforms. We did this primarily because we got tired of working around the bugs, poor algorithms and inconsistent handling of exceptional cases that we found in the libm's provided by other hardware and operating systems vendors.)

FDLIBM has been translated to Java by John Brophy of Visual Numerics in Houston. This is useful in situations where the native code generated by the C version of FDLIBM is inappropriate. Brophy's source code is publicly available, http://www.vni.com/corner/garage/grande/index.html.

An Idealized Model

Let *F(x)* be any of the elementary math functions under consideration. How can we assess the accuracy of a routine that computes *F(x)*? One possibility is to regard any floating point input argument, *x*, as a precisely known real number, call upon some oracle to evaluate *F(x)* exactly, and then round the result to the nearest floating point number. There is only one rounding error in this entire theoretical computation. This is what we mean by "the correctly rounded result" or "getting the right answer".

This ultimate accuracy may not be justified by itself, but the implied consequences are very important. If a particular routine is known to fit this model, then it will always get the same results on different machines. If a particular function *F(x)* has monotonicity properties, such as *x <= y* implies *exp(x) <= exp(y)*, or range properties, such as *-1 <= cos(x) <= 1*, then the computed function will inherit these properties.

Several years ago, a widely used Unix math library had *cos(0) > 1*. The value was an accurate approximation to the exact value; it differed from 1 by only one bit in the last place. But it violated a fundamental property of the cosine function. This kind of violation is not possible when computations produce correctly rounded results.

It is often argued that the input x is not known exactly, so there is no reason to evaluate *F(x)* exactly. This same argument could be (and actually has been) used to justify sloppy rounding with basic arithmetic operations like additional and multiplication. Again, the goal is not the accuracy itself, but the desirable mathematical consequences that correct rounding brings.

Whether or not this idealized model is justified in any particular context can be discussed in that context. Whether or not such a model can be achieved with reasonable cost in time and program size is a key question in the current discussion.

Ulps

Ulps stands for "units in the last place". This is a strong measure of error in which the error in y is expressed relative to the spacing of the floating-point numbers near y. For all double precision numbers between 1 and 2, an ulp is 2^{-52}. For numbers between 2^{k} and 2^{k+1}, an ulp is 2^{k-52}.

The distance from any real number to the nearest floating-point number is less than or equal to the half the spacing between the nearby numbers. So the correctly rounded results in our idealized model have errors <= 0.5 ulps. Documentation for various elementary function libraries, including FDLIBM, often says that the error is less than one ulp. That's good, but not perfect. Half an ulp is the ultimate goal.

Square Root

The documentation for FDLIBM itself says:

```
 *   ------------------------------------------
 *   |  Use the hardware sqrt if you have one |
 *   ------------------------------------------
```

It may be true that the FDLIM software square root and various hardware square roots all produce the same results. But, if that's not the case, what should Java use? For square root, the choice is clear: the hardware square root should be producing the correctly rounded answer, so use it. But, for other functions, the resolution is not so clear.

What is pi?

Let pi be the transcendental mathematical quantity usually denoted by a Greek letter and let PI be `Java.lang.Math.PI`, the floating point number closest to pi. What is the correct value for `Java.lang.Math.sin(PI)`? The answer is not zero, because PI is not equal to pi. Here are two answers, one obtained from the `FSIN` instruction on an Intel Pentium and the other obtained from FDLIBM:

```
1.224606353822377e-16
1.224646799147353e-16
```

These two values agree to only five significant figures. If the FDLIBM result is regarded as the correct answer, then the error in the `FSIN` result is 1.64e+11 ulps, or 164 gigaulps. It can be argued that both values are so small that either one is an acceptable result, but the mere fact that more than one answer is possible violates both the machine independence objective and the idealized model.

The first stage in the computation of sin(x) is argument reduction. If abs(x) is greater than pi/2, then an integer multiple of some approximation to pi is subtracted from x to bring it into the desired range. Different hardware and software manufacturers use different approximations to pi in this argument reduction. The `FSIN` instruction on Intel's Pentium has a 66-bit approximation to pi available in silicon for this purpose. Before the Pentium, earlier Intel 80x87 chips used less accurate approximations to pi. Documentation for new Cyrix and AMD chips indicate they may use more than 66 bits.

FDLIBM has a 1144-bit approximation to pi. This allows argument reduction equivalent to an infinitely precise pi for any input value less than 2^{1024}, the largest possible double precision floating point number. (See the papers by K. C. Ng available from http://www.validgh.com/.)

So, for sin(PI), the FDLIBM sin function gives the "correct" result required by our idealized model, while the hardware instructions on various chips give different, less correct, results. On the Pentium, the Java Development Kits from both Microsoft and Sun appear to use the less accurate `FSIN` instruction.

Using these different values for pi for the argument reduction in sin(), cos() and tan() is the most serious deterrent to machine independent results. When measured if terms of relative error, or ulps, the discrepancies can be huge.

Moreover, it is going to be difficult to get everyone to agree on one particular value for pi. The infinitely precise scheme in FDLIBM has the best overall mathematical properties, but its advantages may not be strong enough to convince PC uses to abandon their builtin functions.

Computing sin(x) for large x

Because the Pentium `FSIN` instruction only "knows" 66 bits of pi, it can not do accurate reduction of large arguments. When invoked with an argument larger than 2^{63}, the instruction sets an error bit in the floating-point status word and returns the argument unchanged. Systems, like Java, that use the `FSIN` instruction without checking the error bit, find that for large x, sin(x) is computed really fast, but is equal to x. This is unacceptable for general purpose technical computing.

Serious flaw in exp(x)

Like sin(x), computation of exp(x) begins with an argument reduction. The role of pi is played by ln2, the natural logarithm of 2. For any x, let n = round(x/ln2). Then

$$\exp(x) = 2^n * \exp(x - n*\ln 2).$$

The reduced argument is in a range where exp() can be approximated quickly and accurately by a modest number of terms in a power series. The final scaling by 2^n is accomplished by simply adding n to the floating-point exponent.

The integer n has been chosen so that severe subtractive cancellation occurs in the computation of the reduced argument, x - n*ln2. If the reduced argument is to have full relative precision, then ln2 must be represented to, and the computation must be carried out in, higher precision. It appears that the exp function in Microsoft's libm, which is used by Visual C and by both Microsoft's and Sun's JDKs, fails to do this. For example,

```
java.lang.Math.exp(100) = 2.688117141816161e+043
                                          ^^
```

whereas the exact answer is

```
exp(100) = 2.6881171418161135448... * 10^43
                              ^^^^^
```

The error is more than 52 ulps.

More extensive experiments show that as x increases, the maximum error in exp(x) doubles whenever x/ln2 cross a power of 2. This is strong evidence that the argument reduction is not being done carefully.

Table Maker's Dilemma

Is a perfectly rounded elementary math library possible and practical? The difficulty is that nobody knows how much computation is required to achieve perfect rounding in all cases. If *F(x)* is a transcendental function, then, except for special arguments like *x = 0* and *x = 1*, the value of *F(x)* is irrational. So, it can't be a floating-point number, and it can't fall exactly halfway between two floating-point numbers. But it can fall arbitrarily close to halfway between two numbers. A potentially very large number of extra digits is required to decide which way to round. This is the Table Maker's Dilemma.

A good example involving the exponential function has been found by Vincent Lefevre and Jean-Michel Muller of the Ecole Normale Superieure de Lyon (http://www.ens-lyon.fr/~jmmuller/Intro-to-TMD.htm.) Let x be the floating point number obtained by entering

x = 0.8375224553405740

The exact value of x is 7543731635572464 x 2^{-53}, or, in hexadecimal, x = 1.ACCFBE46B4EF0 x 2^{-1}. The decimal representation of exp(x) begins with

exp(x) = 2.3106351774748008...

The reason why we are interested in this number is revealed by the hexadecimal expansion of exp(x)

exp(x) = 1.27C2E4BC1EE70 7FFFFFFFFFFFF EB0C2DA33BA8F ... x 2^1

The first blank is at the point where we have to decide how to round. Should we round down to a floating-point number that ends in EE70, or up to EE71? The binary expansion after the rounding point has a zero, followed by 54 ones, then another zero. We need to go into the third double precision number before we can make the correct decision. If we decide to accept

$$\exp(x) = 1.27C2E4BC1EE70 \times 2^1$$

the error is 0.4999999999999999818 ulps. If, instead, we were to round up to EE71 the error would be 0.5000000000000000182 ulps. We have found our correctly rounded result, but it wasn't easy to get it.

Lefevre and Muller have done an exhaustive search to establish that this is the worst case for the exponential function with arguments between 1/2 and 1. But no one knows how much extra precision is needed to do perfect rounding for all elementary functions and all double precision floating point arguments.

Library Development

Two research groups, one at Sun in Menlo Park and one at IBM in Haifa, are currently working on elementary function libraries that deliver correctly rounded results. As far as we know, their efforts have not yet been reviewed by others. We do not know how the execution time compares with machine instructions or FDLIBM. And, the libraries may use fairly large tables, so their memory requirements may be significant.

Proving that such a library is perfect by exhaustive search is out of the question. A rigorous proof will have to involve careful mathematical analysis of the underlying algorithms and code.

Conclusions

Here are my recommendations for the vendors of Java libraries and the maintainers of the Java specifications.

Immediately:

> Fix the libraries on the PC so that exp(x) uses accurate argument reduction and sin(x), cos(x) and tan(x) can be computed for large x. Check to see if there are any other serious flaws.

The Java Spec:

> Augment the reference to NETLIB with a link to Brophy's work. In strict mode, continue to require the FDLIBM results. In nonstrict mode, allow different values of pi and any results with less than one ulp error.

A Future Java Spec:

> Monitor the development of the perfectly rounded libraries. If they become viable, independently reviewed, have publicly available source code and live up to their claims, then change the spec to require perfectly rounded results in strict mode.

5 Participants

Chairs

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

- Mark Baker, European JavaGrande Initiative
- 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, DePaul University, JHPC Laboratory, Secretary General
- Martin Westhead, EPCC, University of Edinburgh, UK, Benchmarking Group

Numerics Working Group

The following individuals contributed to the development of the original 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, DePaul University, JHPC Laboratory, 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 the original 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
- Gregor von Laszewski, Argonne National Laboratory
- 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, DePaul University, JHPC Laboratory, 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, 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