Java Grande – High Performance Computing with Java

Michael Philippsen, Ronald F. Boisvert, Valdimir S. Getov, Roldan Pozo, José Moreira, Dennis Gannon, and Geoffrey C. Fox

Abstract. “Grande” applications are those with demanding CPU and I/O requirements. They originate in many disciplines, such as astrophysics, materials science, weather prediction financial modeling, and data mining. Java has many features of interest to developers of such applications. At the same time, there are currently many barriers to the effective use of Java in this way. The Java Grande Forum is a group of researchers and software developers from industry, academia, and government with an interest in the use of the Java programming language and environment for grande applications. The Forum seeks to increase awareness of issues important to this community, and to work towards their solution. In this article we describe the workings of the Java Grande Forum and the major issues that it has brought to the forefront. We outline approaches to the solutions of these problems, and describe efforts to standardize them within the larger Java community. Among the issues addressed are: floating-point performance, multidimensional arrays, complex arithmetic, fast object serializations, and high-speed remote method invocation (RMI).

1 Introduction

The Java language and environment provide a number of well appreciated features for software developers. Among these are

- clean object-oriented approach
- support for memory management, threads, exceptions
- network, Web awareness
- intrinsic portability
- large collection of standard class libraries, including GUI components
- growing user base

The promise of portability is a particularly compelling one. Java programs are compiled into byte codes for the Java Virtual Machine (JVM). This machine is emulated in order to execute Java programs. JVMs are now available on nearly every computer platform, in Web browsers, and in many other devices. Thus, compiled Java class files are highly transportable. Since the semantics of a single-threaded piece of Java code are defined precisely and deterministicaly, the results of execution on any conforming JVM should be the same.

Other features of Java provide conveniences which lead to fewer errors and more productive programmers. Among these is automated memory management.

Java programmers do not need to explicitly deallocate blocks of memory. Instead, Java maintains a garbage collector which automatically recovers unused storage. The safety of Java is enhanced by the absence of arbitrary pointers, and by the requirement that all array bounds be checked before access.

Because of such features, Java is now widely used both in commercial software development, in research, and as an educational tool in universities. Indeed, computer science departments everywhere seem to be switching to Java as the main language they teach their students. This, as much as everything else, will assure Java's place in software development for some time to come.

While Java has made great inroads in a variety of areas, it is far from the language of choice for "grande" applications, i.e., those with the most demanding CPU and I/O requirements. (The term "grande", like many things associated with Java, is inspired by coffee house jargon, where grande means large.) They originate in many disciplines, such as astrophysics, materials science, weather prediction, financial modeling, and data mining. However, system requirements for modern grande applications go far beyond mere compute cycles. Communication with distributed components in a heterogeneous environment must be maintained. Graphical user interfaces must be developed. High levels of portability must be maintained to insulate the application from changes in the underlying hardware and software platform. Java is the first single environment to provide all of these features.

Nevertheless, when Java is found in grande contexts today it is typically being used as glue, interconnecting existing high-performance applications, linking computations realized in other languages to one another, and acting as a layer between computations and the user. This a a perfectly reasonable use of Java today. However, the benefits of portability afforded by Java cannot be realized unless the entire application is run in a Java environment.

Why isn't Java commonly used for the compute- or I/O-intensive core of grande applications? The main reason is undoubtedly performance. In its early days JVMs were strictly interpreters, resulting in very poor performance. In the science and engineering community Java has not shaken this early perception.

Today nearly every JVM for traditional computing devices uses just-in-time (JIT) compiler technology. JITs operate as part of the JVM; they compile Java class files into native code at runtime, thus providing a much higher level of performance than interpreted code. Since JITs operate at runtime, they cannot expend a lot of time performing extensive analysis of class file code looking for optimizations. Nevertheless, JITs have been improving steadily, and JIT technologies such as Sun's Hotspot [], which only compiles those portions of the code it deems necessary to improve performance, promise even greater strides forward.

Still, some of Java's features, while kind to programmers, can be performance reducers, and thus sore points for those developing grande applications. Things like overactive garbage collection and unoptimized array bounds checking can take a significant toll on performance.

How good is the performance that one can get out of Java today? Since Java is available on so many platforms, and multiple JVMs are available on each of these, this is somewhat difficult to assess. Certainly, the variance in observed performance for a given application remains great. In order to track the progress of Java performance for numeric-intensive applications, Roldan Pozo and Bruce Miller of NIST have developed a Web-based benchmark called SciMark [33]. SciMark combines five medium-sized numerical kernels: complex-valued fast Fourier transformation, successive over-relaxation (SOR) iteration, Monte-Carlo quadrature of e^{-x^2}, multiplication of sparsely populated matrices, and the LU factorization of dense matrices with pivoting. These are representative of many scientific and engineering computations. For each of these kernels, a Java and a C version are available. In addition, there is a small memory version, appropriate for studying cache-contained kernels, as well as one that exhibits out-of-cache behavior typical of large memory applications.

The small problem Java version of the benchmark is available as an applet at http://math.nist.gov/scimark/. This can be downloaded into a browser and run to assess the local environment. Composite results of the benchmark are reported in megaflops (Mflops); results of the individual components are also available. The applet supports upload of the benchmark data to NIST, and the SciMark Web site maintains an archive of all uploaded results. These provide a snapshot of typical Java performance. For example, the antiquated JVM 1.1.5 of the Netscape browser reaches approximately 0.7 Mflops on an Intel Celeron 366 processor under Linux and is, as such, 135 times slower than a C implementation on that platform. Java 1.1.8 is a huge improvement: using the same processor (running on OS/2) about 76 MFlops have currently been achieved — only 35% less than with C.

Table 1. Results of the SciMark benchmark on a Dell Pentium III 733 MHz system running Windows NT 4.0 and Netscape VM 1.1.5.

Components	**Mflops**
FFT	41
SOR relaxation	227
Monte Carlo quadrature	13
Sparse matrix multiplication	89
LU factorization	131
Average	100

As of this writing (June 2000), the fastest in-browser SciMark results posted are over 100 Mflops. A typical one of these is for a Dell Pentium III 733 MHz system running Windows NT 4.0 and Netscape VM 1.1.5. The individual results reported for this system are as given in Table 1. However, Java performance remains highly variable. The 20 most recent results reported, which come from sys-

tems based on processors ranging from a 166 MHz AMD to a 700 MHz Celeron, show SciMark composite ratings of from 1 Mflops to 89 Mflops.

The Ninja group at the IBM Thomas J. Watson Research Center uses a set of eight benchmarks to evaluate the effect of their Java optimizations. The bencmarks include dense matrix operations (matrix multiply, LU factorization, Cholesky factorization), a neural network kernel, a PDE solver through Jacobi relaxation, a shallow water simulation, an FFT computation, and a version of the SPECfp benchmark TOMCATV. They report that, when a commercially available environemnt (IBM DK 1.1.6) is used on a 200 MHz POWER3 platform, Java performance ranges from 2 to 50% Fortran performance on the same benchmarks. However, when a set of optimizations are applied to the Java code, its performance improves to between 40 and 100% of Fortran. See Table 2 for details.

Table 2. A summary of Java performance with the Ninja compiler.

benchmark	IBM DK 1.1.6		Ninja	
	Mflops	% of Fortran	Mflops	% of Fortran
MATMUL	7	1.7%	340	84%
MICRODC	53	26%	210	102%
LU	45	27%	154	93%
CHOLESKY	5	2.8%	167	97%
BSOM	47	22%	175	81%
SHALLOW	45	24%	156	83%
TOMCATV	50	27%	75	40%
FFT	101	53%	104	54%

The same group at IBM T.J. Watson has also reported good performance results with a data mining application in Java [30]. Executing on a 4-processor RS/6000 model F50, the computational part of the application achieves 109 Mflops for a single threaded Java execution, as compared to 120 Mflops for single threaded Fortran. Moreover, when parallelism is exploited, the multithreaded Java version achieves up to 340 Mflops with four threads. The portability and convenience of parallel programming in Java is a major boost for those in the field of high-performance computing.

Additional Java benchmarking efforts are described in Section 4.4.

In summary, while Java is not yet as efficient as optimized Fortran or C, the speed of Java is better than its reputation suggests. Carefully written Java code can perform quite well [27, 34], and Java compiler and JIT technology is still in its infancy. Taken with the other advantages of Java, there is a real possibility for Java to become the best ever environment for grande applications. In the remainder of this article we explore this idea further in the context of the work of the Java Grande Forum.

2 The Java Grande Forum

The *Java Grande Forum* [21] is a union of researchers, company representatives, and users who are working to improve and extend the Java programming environment, in order to enable efficient grande applications. The Forum was founded in March 1998, during the ACM/SIGPLAN Workshop on Java for Science and Engineering held at Stanford University. Geoffrey C. Fox of Florida State University and Sia Zadeh of Sun Microsystems played key roles in the initial organization of the Forum. Since then the Forum has organized regular public meetings, which are open to all interested parties, as are its web site [21] and mailing list [22].

The main goals of the Java Grande Forum are the following:

- Evaluation of the applicability of the Java programming language and the run-time environment for grande applications.
- Bringing together the "Java Grande community" to develop consensus requirements and to act as a focal point for interactions with Sun Microsystems.
- Creation of demonstrations, benchmarks, prototype implementations, application programmer interfaces (APIs), and recommendations for improvements, in order to make Java and its run-time environment utilizable for grande applications.

The participants in the Java Grande Forum primarily represent American and European companies, research institutions, and laboratories (see Table 2. Cooperation with hardware and software vendors is crucial, especially in reference to questions dealing with high-speed numerical computing.

The scientific work of the Forum is important for establishing a cohesive community of researchers and users of Java for grande applications. This makes it possible to focus interests and to achieve consensus, thus making it easier to achieve goals.

The Forum organizes scientific conferences, workshops, minisymposia, and panels in order to present its work to interested parties; see Table 4. The most important annual event is the ACM Java Grande Conference. A large portion of the scientific contributions of the Java Grande community can be found in the conference proceedings (Table 4) and in some issues of *Concurrency – Practice & Experience* [14–17]. In addition, the Java Grande Forum publishes working reports at regular intervals [37, 38].

The members are organized into two working groups. The Numerics Working Group is co-chaired by Ronald Boisvert and Roldan Pozo of NIST. The Concurrency and Applications Working Group is co-chaired by Dennis Gannon of Indiana University and Denis Caromel of INRIA. The next two sections describe the technical work of these groups in some detail.

The reports developed and events organized by the Java Grande Forum have been well-received by key Java developers within Sun Microsystems, such as Tim Lindholm, Bill Joy, James Gosling, John Gage, and Guy Steele. Impressive

Table 3. Some participants in the Java Grande Forum

Companies
IBM
Intel
Least Squares Software
The MathWorks
MPI Software Technologies
NAG
Sun Microsystems
Unidata
Visual Numerics
Waterloo Maple
Universities
Florida State University
Karlsruhe University
Loyola University of Chicago
Syracuse University
University of California at Berkeley
University of California at Santa Barbara
University of Edinburgh
University of Houston
University of Maryland
University of North Carolina at Chapel Hill
University of Tennessee at Knoxville
Westminster University
Government Supported Laboratories
INRIA, Institute National de Recherche en Informatique et en Automatique, France
Institute for Computer Applications in Science and Engineering (ICASE)
Pittsburgh Supercomputer Center
Sandia National Laboratories
U.S. National Institute of Standards and Technology (NIST)

public relations work was done by Bill Joy, who praised the work of the Java Grande Forum in front of an audience of about 21,000 at the 1999 JavaOne Conference.

Table 4. Some events organized by the Java Grande Forum

Workshop on Java for High Performance Computing, Syracuse University, December 1996, [14]
Workshop at Principles and Practice of Parallel Prog., Las Vegas, June 1997, [15]
Panel at SC'97, San Jose, November 1997
ACM SIGPLAN Workshop on Java for High Performance Network Computing, Stanford University, February 1998, [16]
Workshop at EuroPar, Southampton, September 1998, [13]
Exhibit and Panel at SC'98, Orlando, November 1998, [37]
Panel at IEEE Frontiers of Massively Parallel Computing, Annapolis, February 1999
Workshop at HPCN/Europe, Amsterdam, April 1999, [36]
Workshop at IPPS/SPDP, San Juan, April 1999, [12]
Minisyposium at SIAM Annual Meeting, Atlanta, May 1999
Java Grande Forum Meeting, Palo Alto, May 1999
Mannheim Supercomputing Conference, June 1999
ACM Java Grande Conference, San Francisco, June 1999, [17]
Exhibit and BOF, JavaOne, San Francisco, June 1999, [38]
Workshop at ACM ICS'99, Rhodes, Greece, June 1999
Java Grande Forum Meeting, Palo Alto, August 1999
Exhibit and Panel at SC'99, Portland, Oregon, September 1999
Java Grande Day at ISCOPE99, December 1999, [35]
Workshop at HPCN/Europe 2000, Amsterdam, May 2000
Workshop at ICS2000, Santa Fe, New Mexico, May 2000
Workshop at IPDPS, Cancun, May 2000
ACM Java Grande Conference, San Francisco, June 2000
SIAM National Meeting (Minisymposium), Puerto Rico, July 2000
Seminar on High Performance Computing in Java, Dagstuhl, Germany, August 2000

3 Numerics Working Group

3.1 Goals of the Working Group

The Numerics Working Group set the evaluation of the applicability of Java for numerical computing as its initial goal. Building on that effort, the group developed consensus on basic requirements for numerical computing in Java. These have led to a series of particular recommendations to eradicate a variety of deficiencies in the Java language and its runtime environment. Some of these have been adopted by Sun. Related to these efforts, group participants have implemented a variety of prototype class libraries and language processors to demonstrate the feasibility and utility of their recommendations.

Among the areas of critical concern for the Working Group have been floating-point performance, complex arithmetic, efficiency of multidimensional array operations, as well as the development of prototype class libraries for common mathematical operations. The following sections discuss individual problems and results. Futher details can be found at the Java Numerics web site [23].

3.2 Improvement of Floating Point Arithmetic

One of the most important principles in the design of Java has been the portability of its class files. This has led to a very precise definition of the semantics of floating-point arithmetic in Java and its JVM. This precision in specification is unlike that in any language for scientific computing. While this can lead to exactly reproducible results, it ties the hands of compiler optimizers, making it very difficult, and often impossible, to produce the most highly optimized code on a given undelying hardware platform. Those who work on the most computationally intensive problems — grande applications — have learned to deal with architecture-specific differences in the details of floating-point arithmetic. They are happy to live with some ambiguity in the quest for the highest performance on a given processor. However, these are just a portion of those who do numerical computing. The majority of users need a high level of predictability (though not necessarily absolute), and good overall performance (though not necessarily optimal). At the other extreme, some users do, in fact, need absolute predictability. Someone writing software for realtime control of surgical instruments might want to know precisely how every operation is to be preformed in order to prevent unexpected cancellations, overflows, etc. If Java is to be *the* language of scientific computing it must satisfy all three types of users.

The original Java specification [] uses the IEEE 754 arithmetic standard as its basis. Since most general-purpose computing systems now follow this standard, this was a very rational choice to make. However, IEEE 754 itself is not a narrow specification; it admits a number of optional features, some of which have become central design features of particular microprocessors. Originally, Java mandated use of only the most basic subset of IEEE 754. This led to an inevitable conflict between advocates of reproducibility on the one hand, and performance on the other. A related issue is the specification of the results computed by the elementary functions (like sin and exp). These too were precisely laid out in the Java specification, and this also led to a disconnect between the specification and practice. The Numerics Working Group has made efforts, some successful, to try and bridge the gap between these competing interests.

Floating Point Performance. In order to achieve exact reproducibility, Java forbids common optimizations, such as making use of the associativity property of mathematical operators, which does not hold in a strict sense in floating-point arithmetic, $(a+b)+c$ may produce a different rounded result than $a+(b+c)$. Further prohibitions affect the use of particular processor features.

1. Prohibition on use of 80 Bit "double extended format". Processors of the x86-family (e.g. the Intel Pentium) have 80-bit wide registers that use the double extended format as defined in the IEEE 754 Standard. x86 double extended numbers have a 15 bit exponent and a 65 bit mantissa, while regular IEEE doubles precision numbers have an 11 bit exponent and a 53 bit mantissa. The original Java specification required the exclusive use of regular IEEE doubles to represent the Java primitive type `double`. Thus, on the x86, every intermediate result of a mathematical operation must be rounded to a more imprecise number format. Even if the x86's precision control bit is set to enforce this rounding in the registers after each step, a 15 bit exponent is still produced. This means that overflow and underflow may not occur at the proper point. Thus, to conform to Java's strict semantics, each intermediate result is typically stored to memory (where IEEE double format is used) and reloaded to the register to continue the computation. This is an extremely time consuming operation, leading to a two to ten-fold performance hit in numeric-intensive applications. Many JVM implementations ignored proper rounding for exactly this reason. Thus, on Intel processors, Java's floating-point arithmetic was either wrong or impossibly slow.

2. Prohibition to use for Fused Multiply Add (FMA) machine instructions. Processors of the PowerPC-family offer a machine instruction that computes the quantity $ax + y$ as a single operation. This is the fused multiply-add, or FMA. Operations of this type are found in many compute-intensive applications; they are at the innermost loop of most matrix operations, for example. Use of this facility can lead to superscalar performance on these architectures, a highly prized feature for grande applications. An added benefit of the use of such instructions is that only a single rounding occurs for the two arithmetic operations, yielding a more accurate result in less time than would be required for two separate operations.

Because of the single rounding, Java's strict language definition does not permit use of FMAs and thus sacrifices 50% of the peak performance on some platforms. Figure 1 shows experiments in running a Java Cholesky factorization code on a Power-2 processor at IBM's T.J. Watson Research Center using an experimental static Java compiler, i.e., one that produces an executable by staticaly translating from Java bytecode to machine code. Using the strict Java semantics, and without any optimizations, only 3.8 Mflops is achieved. If common optimizations that are legal in the context of Java are carried out (including the elimination of proveably redundant array boundary checks), then 62% of the performance of an equivalent optimized Fortran program results (83.4 MFlop). Finally, if FMAs are used – which the Fortran compiler does routinely – nearly 97% of the Fortran performance is achieved.

Improvements in Java Floating-Point Semantics. In the spring of 1999, Sun introduced changes in the semantics of floating-point in Java (version 1.2). These changes, which were part of a set of proposals made by the Java Grande Forum, significantly improve the performance of conforming Java platforms for

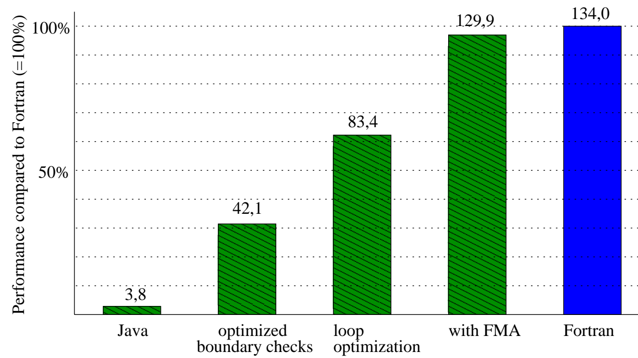

Fig. 1. Cholesky Factoring on a 67 MHz Power-2-Processor. Information given in MFlops and relative to optimized Fortran.

the x86 architecture. The main change was, in a sense, very small: to admit the use of 15-bit exponents for anonymous `double` variables, i.e., for intermediate results. (A similar change was made in the semantics of `float`). This eliminates the need for store/reload after each operation on the x86, thus allowing the processor to run at full speed. Note that this constitutes an exception to Java's strict reproducibility — results on the x86 can be different from those produced on the SPARC, for example. Since use of 15-bit exponents is optional in the Java specification, these differences could conceivably occur in two different JVMs executing on the same processor. Fortunately, these differences will be rare in practice. They only occur when a computation would overflow or underflow with an 11-bit exponent, but not with a 15-bit exponent.

The principle of exact reproducibility has not been completely thrown to the wind, however. A new keyword, `strictfp` was also added to the language, again at the recommendation of the Numerics Working Group. When `strictfp` is affixed to classes or methods, the original Java semantics must be respected in the execution of the corresponding (static) code. In this way, users who need exact reproducibility for floating-point computations can still obtain it, but the majority of users can opt for a slight relaxation of this requirement to achieve greatly improved performance. The latter is now the default in Java.

Reproducibility of `Math`-Functions In its quest for exact reproducibility, the original Java specification also attempted to precisely define the behavior of its elementary mathematical functions such as sin and exp. These are found in the package `java.lang.Math`. According to the specification, this library must be implemented by porting the `fdlibm` library.[1] However, an actual port of this library was never done, and hence most JVMs use the local `libm`, or the faster

[1] The `fdlibm` library is the free math library distributed by Sun. `Fdlibm` is considerably more stable, to a greater degree correct, and much more easily portable than the `libm` libraries available on most platforms.

hardware implementations of these functions. The result is that, in practice, Java programs using the elementary functions produce different results when run on different JVMs.

Another difficulty with an operational specification of this type is that it prohibits the use of more accurate methods should they be developed. (The `fdlibm` algorithms are quite accurate, but they do not produce the correctly rounded version of the exact result in all cases.)

Again, the Java Grande Forum's Numerics Working Group developed a recommendation for improvement, which was adopted in Java 1.3, unveiled in the spring of 2000. The specification for `java.lang.Math` states that the functions must return a result which is within one unit in the last place (ulp)[2] of the correctly rounded result. In addition, the results produced by the functions should be monotone where the exact functions are monotone. `Fdlibm`, as well as existing hardware implementations, has these properties, but its use is not mandated. This relaxes the exact reproducibility requirement, but admits implementations which are faster or more accurate. For example, Abraham Ziv of IBM Haifa has created a math library (in ANSI C), that is guaranteed to produce correctly rounded results in IEEE arithmetic (i.e., errors less than 0.5 ulp) [40].

Proponents of exact reproducibility need not fear, however. Java 1.3 also introduces an alternate math library `java.lang.StrictMath`, which must adhere to the original Java specification, thus providing a means to obtain the same result on all platforms. Java Numerics Working Group member John Brophy of Visual Numerics has developed a full implementation of `fdlibm` completely in Java [4]. It is expected that this code will be provided with future standard distributions of Java, thus making it likely that Java implementations will faithfully adhere to the specification.

Further improvements in floating-point performance. The creation of two separate floating-point modes in Java satisfies the needs of those who want strict reproducibility and those who are willing to relax this slightly in hopes of gaining much better performance. Unfortunately, this may not be enough for those programmers, typically those with grande applications, who require the fastest performance possible, and are willing to relax floating-point semantics much further in order to achieve it. To begin to satisfy the needs of this latter group, the Numerics Working Group is in the process of developing a specification for a third floating-point mode. This mode would be designated by a new `fastfp` keyword for classes and methods. In this new mode, FMAs would be allowed in particular well-defined circumstances. The sensibility of also allowing the use of the associative rule to rearrange the order of computations is also being examined.

[2] For `doubles` between 2^k and 2^{k+1}, 1 ulp $= 2^{k-52}$.

3.3 Efficient Complex Arithmetic

Another indicator of the ability of a programming language to support serious scientific and engineering computing is the ease and efficiency in which computation with complex numbers can be done. Many applications, such as those in electromagnetic and acoustic modeling, for example, are best accomplished in the complex domain. Complex is just one example of an important alternate arithmetic. Others of growing importance are interval arithmetic and multiprecision arithmetic. A good scientific computing language would have the flexibility to incorporate new arithmetics like these in a way that is both efficient and natural to use.

In Java, complex numbers can only be realized in the form of a `Complex` class whose objects contain, for example, two `double` values. Complex-valued arithmetic must then be expressed by means of complicated method calls, as in the following code fragment, which says $y = ax + b$, where $a = 5 + 2i$ and $b = 2 - 3i$.

```
Complex a = new Complex(5,2);
Complex b = new Complex(2,-3);
Complex y = a.times(x).plus(b);
```

This has several disadvantages. First, such arithmetic expressions are quite difficult to read, and hence are error-prone to code and maintain. Second, complex arithmetic is slower than Java's arithmetic on primitive types, since it takes longer to create and manipulate objects. Objects also incur more storage overhead than primitive types. In addition, temporary objects must be created for almost every method call. Since every arithmetic operation is a method call, this leads to a glut of temporary objects which must be frequently dealt with by the garbage collector. In contrast, Java primitive types are directly allocated on the stack, leading to very efficient manipulation.

To illustrate the inefficiencies of complex classes, researchers at IBM recently analyzed the performance of class-based complex arithmetic in Java using Jacobi relaxation as a computational kernel. This kernel is typical of iterative methods for large sparse linear systems on grids. An implementation based on a class `Complex` achieved only 1% of an equivalent Fortran code [39]. The main culprit of Java's dismal performance is the voracious rate of temporary object creation and destruction. This can be visualized in Figure 2. Figure 2(a) is a plot of memory utilization over time for a version of the code that uses the primitive `double` data type. Figure 2(b) is the same plot for a version using the `Complex` class. We note that, in the second case, the memory is exhausted several times during execution, forcing a garbage collection operation.

Finally, class-based complex numbers invariably cannot be fully integrated in the system of primitive types. They are not integrated into the type relationships that exist between primitive types, so that for example, the assignment of a primitive `double` value to a `Complex` object does not result in any automatic type cast. Equality tests between complex objects refer to object identities rather than to value equality. In addition to this, an explicit constructor call is

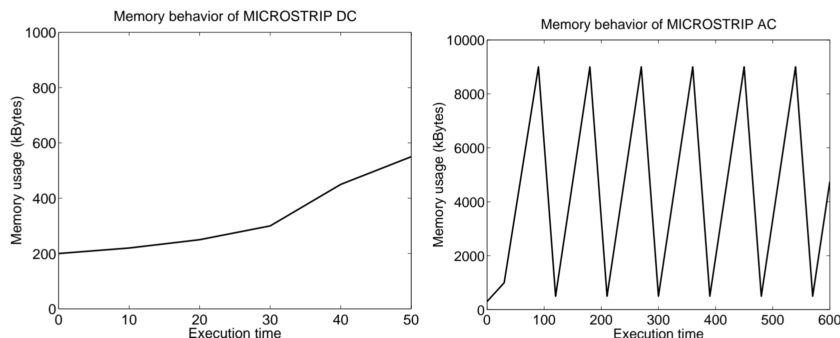

Fig. 2. Profile of memory utilization using (a) `double` and (b) `Complex` class.

necessary for a class-based solution, where a literal would be sufficient to represent a constant value. Thus, methods which manipulate complex numbers have essential differences from their real number counterparts. This places a heavy burden on library developers who would like to use automated tools to create `float`, `double`, and complex versions of the same algorithm.

These same difficulties arise in the implementation of any other arithmetic system in Java, such as intervals and multiprecision. A general solution to these problems would be afforded by the introduction of two additional features to the language: operator overloading and lightweight objects.

Operator overloading is well known. It allows one to define, for example, the meaning of `a + b` when `a` and `b` are arbitrary objects. Operator overloading is available in several other languages, like C++, and has been widely abused, leading to very obtuse code. However, when dealing with alternative arithmetics, the mathematical semantics of the arithmetic operators remain the same, and hence it leads to naturally readable code. In Java, one would need to be able to overload the arithmetic operators, the comparison operators, and the assignment operator.

Lightweight objects are defined by final classes with value semantics. Their instantiated variables cannot be changed after object creation. Lightweight objects can often be allocated on the stack and passed by copy. Their methods could be inlined, thus leading to efficient computations in JVMs which are serious about performance.

Unfortunately, it is not known when, or if, operator overloading or lightweight objects will be integrated into Java. Since scientific computing only makes up a small portion of total Java use, it is improbable that the Java Virtual Machine (JVM) or the bytecode format will be extended to include a new primitive type `complex`, although this would probably be the most straightforward way of introducing complex numbers into Java. As a result, the Java Grande Forum regards the following twin-track strategy to be more sensible (see Figure 3).

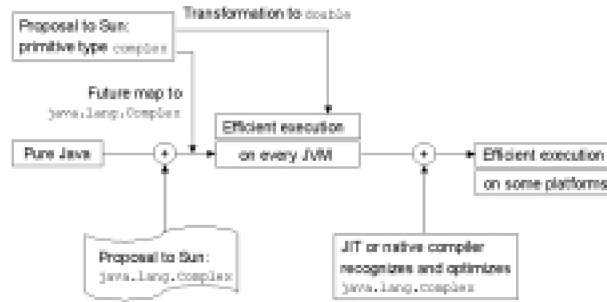

Fig. 3. Introduction of complex numbers and arithmetics

Class `java.lang.Complex`. The Numerics Working Group has defined and prototyped a class `java.lang.Complex` that is similar in style to Java's other numerical classes. In addition, method-based complex arithmetic operations are provided. The Group plans to submit this class to Sun in the form of a Java Specification Request (JSR) for possible incorporation into the collection of standard classes.

Researchers at the IBM T.J. Watson Research Center have built the semantics of this `Complex` class permanently into their experimental native Java compiler using an approach they call *semantic expansion* [39]. Internally, a complex value type is used in place of temporary objects in the code being compiled, and the usual compiler optimizations for complex numbers (as in Fortran compilers) are carried out. In particular, most of the arithmetic methods and constructor calls that prevail in the Java code using this class are replaced by stack and/or register operations. This provides an alternate way to efficiently support a complex class, but it requires special attention by compiler developers. Because of this it is unlikely that such support will be widespread.

Primitive Data Type `complex`. The Numerics Working Group is also planning to submit a related JSR calling for a primitive type `complex`, with corresponding infix operations, in the standard edition of Java. In order to avoid the need for changes to the bytecode format and existing JVM implementations, the language extensions are mapped back to normal Java in a pre-processor step. An additional `imaginary` data type for purely imaginary numbers as has been included in C99 [26, 6] is also being considered.

Figure 3 illustrates two alternative pre-processor transformations. In the first case, the primitive data type `complex` is mapped to a pair of `double` values. In this way, all object overhead is avoided. In the second alternative, complex operations are mapped to the previously described `Complex` class. In this case, compiler-supported semantic expansion would be needed to achieve the necessary efficiency. The compiler *cj*, developed at the University of Karlsruhe, is a prototype implementation based upon a formal description of this process [18].

3.4 Efficient Multidimensional Arrays

In the same way as efficient and convenient complex arithmetic must be made available, numerical computing without efficient (i.e., optimizable) and convenient multidimensional arrays is unthinkable. Java offers multidimensional arrays only as arrays of one-dimensional arrays. This causes several problems for optimization. One problem is that several rows of a multidimensional array could be aliases to a shared one-dimensional array. Another problem is that the rows could have different lengths. Moreover, each access to a multidimensional array element requires multiple pointer indirections and multiple bound checks at run-time. By means of dataflow analysis and code cloning, plus guards, the optimizer can only reduce the amount of boundary checks. The optimizer can seldom avoid all run-time checks.

For this reason the Java Grande Forum has recommended a standard class for multidimensional arrays (once again in the form of a JSR). The `Multiarray` package implements true rectangular multidimensional arrays, in which all rows have exactly the same length. Intra-array aliasing (aliasing of rows within an array) never occurs, and inter-array aliasing (aliasing between rows of different arrays) is easier to analyze and disambiguate than with arrays of one-diemensional arrays. The rectangularity and aliasing properties of the `Multiarray` package enable a number of compiler optimizations to be applied.

As with the class-based complex numbers, a multidimensional array class requires awkward `set` and `get` accessor methods instead of elegant `[]` notation. Also, method-based access to multidimensional array elements would add much overhead, diminishing many of the advantages of having a standard class. Lightweight objects and operator overloading would again provide the general solution to this problem. (In this case, `[]` is considered an operator.)

For this reason, a twin track solution is appropriate in this case, as well. IBM has built permanent support for multidimensional arrays, based upon semantic expansion, into their experimental static Java compiler, and report excellent results from automatic loop transformations and exploiting parallelism [1, 29, 30].

At the same time, an extended multidimensional array access syntax will be proposed, allowing elegant access to multidimensional arrays, e.g. with notation like `a[i,j]`. Such a syntactic extension is quite tricky due to the necessary interaction with regular one-dimensional Java arrays. Once defined, such syntax can also be handled by a pre-processor, translating it either to operations on one-dimensional arrays, or to calls on the standard `Multiarray` class.

3.5 Strategy

Why doesn't the Java Grande Forum try to introduce lightweight classes and operator overloading into Java? The answer is quite pragmatic. The Java Grande Forum hopes the above mentioned JSRs are light enough to withstand the formal process of language alterations. The majority of Java users will remain almost completely unaffected by the proposed changes, and quite possibly will not notice

the changes at all. Very few of today's Java users are even aware of the existence, much less the importance, of the `strictfp` keyword, for example. The smaller the number of the people affected, the more likely the endorsement of the JSR.

Value classes and operator overloading demand a greater change to the language as a whole. The former may impact the bytecode format and thus the JVM. For this reason, and due to the almost religious character of arguments about operator overloading, the outcome of such efforts would remain open to speculation. The recommendations made above seem to have better prospects.

3.6 Additional Standard Class Libraries for Mathematics

In addition to the proposed standard classes for complex arithmetic and multidimensional arrays, members of the Numerics Working Group have developed a number of additional prototype classes for core numerical computing. These include the following.

- The JAMA package for basic matrix algebra developed by The MathWorks and NIST [3].
- The MPJAVA package for multiprecision arithmetic developed at the University of North Carolina [10]. This is based upon Bailey's well-known MPFUN Fortran package [2].
- The Sfun package of higher mathematical functions developed by Visual Numerics [5].

In addition, packages for interval arithmetic and fast Fourier transforms have been discussed. If these gain sufficient community support, then they too will be submitted for approval in the formal standardization process for Java.

4 Concurrency and Applications Working Group

4.1 Goal of the Working Group

The Concurrency and Applications Working Group of the Java Grande Forum evaluates the applicability of Java for parallel and distributed computing. Actions based on consensus are formulated and carried out, in order to get rid of inadequacies in the programming language or the run-time system. The results that have been achieved will be presented in the following sections. Further work in the field of parallel programming environments and "Computing Portals" have not yet been consolidated and will not be covered in this article.

4.2 Faster Remote Method Invocation

Good latency times and high bandwidths are essential for distributed and parallel programs. However, the remote method invocation (RMI) of common Java distributions is too slow for high performance applications, since RMI was developed for wide area networks, builds upon the slow object serialization, and does

not support any high speed networks. With regular Java, a remote method invocation takes milliseconds – concrete times depend on the number and the types of arguments. A third of that time is needed for the RMI itself, a third for the serialization of the arguments (their transformation into a machine-independent byte representation), and a third for the data transfer (TCP/IP-Ethernet).

In order to achieve a fast remote method invocation, work must be done at all levels. This means that one needs a fast RMI implementation, a fast serialization, and the possibility of using communication hardware that does not employ TCP/IP protocols.

Within the framework of the JavaParty Project at the University of Karlsruhe [24], all three of these requirements were attacked to create the fastest (pure) Java implementation of a remote method invocation. On a cluster of DEC-Alpha computers connected by Myrinet, called ParaStation, currently a remote method invocation takes about 80 μs although it is completely implemented in Java.[3] Figure 4 shows, that for benchmark programs 96% of the time can be saved, if the UKA serialization, the high-speed RMI (KaRMI), and the faster communication hardware is used. The central ideas of the optimization will be highlighted in the next sections.

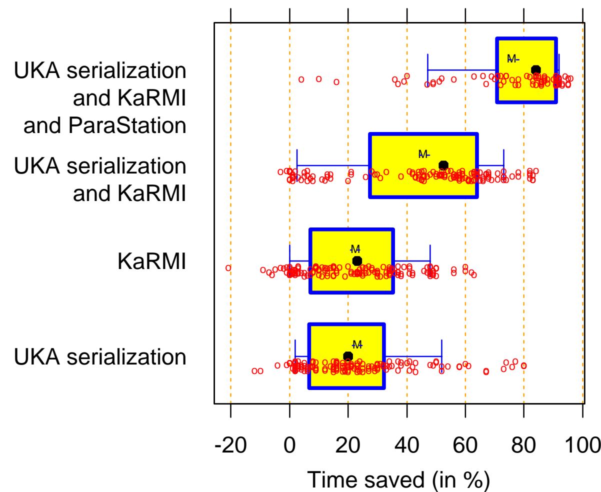

Fig. 4. The bottom three box plots each show 2·64 measured results for diverse benchmarks (64 points represent Ethernet on PC, 64 stand for FastEthernet on Alpha). The first (bottom-most) box plot shows the run-time improvement that was achieved with regular RMI and the UKA serialization. The second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

[3] Of course, the connection of the card driver was not realized in Java.

UKA Serialization. The UKA serialization [19] can be used instead of the official serialization (and as a supplement to it) and saves 76%–96% of the serialization time. It is based on the following main points:

- Explicit serialization routines ("marshalling routines") are faster than those used by classical RMI that automatically derive a byte representation with the help of type introspection.
- A good deal of the costs of the serialization are needed for the time-consuming encoding of the type information that is necessary for persistent object storage. For the purposes of communication, especially in work station clusters with common file systems, a reduced form of the type encoding is sufficient and faster. The Java Grande Forum has convinced Sun Microsystems to make the method of type encoding plugable in one of the next versions.
- Copied objects need to be transferred again for each call in RMI. RMI does not differentiate between type encoding and useful data, meaning that the type information is transferred redundantly.
- Sun has announced (without concretely naming a version) it will pick up on the idea of a separate reset of type information and user data.
- The official serialization uses several layers of streams that all possess their own buffers. This causes frequent copying operations and results in unacceptable performance. The UKA serialization only needs one buffer, which the byte representation can be directly written in.
- Although Sun remains steadfast about layering for reasons of clearer object-oriented design, they are at least improving the implementation of the layers.

KaRMI. A substitute implementation of RMI, called KaRMI, was also created at the University of Karlsruhe. KaRMI [32] can be used instead of the official RMI and gets rid of the following deficiencies, as well as some others found in official RMI:

- KaRMI supports non-TCP/IP networks. Suns plans to add support in the official RMI-Version as well.
- KaRMI possesses clearer layering, which will make it easier to employ other protocol semantics (i.e. Multicast) and other network hardware (i.e. Myrinet-Cards).
- In RMI, objects can be connected to fixed port numbers. Therefore, a certain detail of the network layer is passed to the application. Since this is in conflict with the guidelines of modular design, KaRMI only supports use of explicit port numbers when the underlying network offers them.
- The distributed garbage collection of the official RMI was created for wide area networks. Although there are optimized garbage collectors for tightly coupled clusters and for other platforms [31], the official RMI sees no alternative garbage collector as being necessary, in contrast to KaRMI.

4.3 Message Passing in Java

The Java language has several built-in mechanisms which allow the parallelism inherent in scientific programs to be exploited. Threads and concurrency constructs are well-suited to shared memory computers, but not large-scale distributed memory machines. Although sockets and the RMI interface allow the development of big network applications, they have been designed and optimized for client-server programming, whereas the parallel computing world is mainly concerned with a more symmetric model, where communications occur in groups of interacting peers. Therefore, codes based on sockets and RMI would naturally underperform platform-specific implementations of standard communication libraries based on the successful Message Passing Interface (MPI) standard [42]. By contrast with sockets and RMI, 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.

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 would lead to divergent, non-portable practices. The Message-Passing Working Group of the Java Grande Forum was formed in the Fall of 1998 as a response to the appearance of the various APIs for message-passing. Some of these early "proof-of-concept" implementations [20, 28, 25] have been available since 1997 with successful ports on clusters of workstations running Linux, Solaris, Windows NT, Irix, AIX, HP-UX, and MacOS, as well as on parallel platforms such as the IBM SP-2 and SP-3, Sun E4000, SGI Origin-2000, Fujitsu AP3000, Hitachi SR2201 and others. An immediate goal was to discuss and agree on a common API for MPI-like libraries for Message Passing in Java (MPJ) [7].

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 MPJ API specification follows this model, lifting the structure of its class hierarchy directly from the C++ binding. The purpose of this phase of the effort is to provide an immediate, ad hoc standardization for common message passing programs in Java, as well as to provide a basis for conversion between C, C++, Fortran, and Java.

In this paper we present performance analysis and comparisons of evaluation results for both Java and C/Fortran on three different message-passing parallel platforms – a shared memory multi-processor (Sun E4000), a Linux cluster, and a distributed memory computer (IBM SP-2). The NAS parallel Embarrassingly Parallel (EP) and the Integer Sort (IS) benchmarks were used in our performance evaluation. The IS routine evaluates integer operations and bi-directional communications (the sorted keys are exchanged between nodes), while the EP kernel tests floating point operations performance but requires minimal communications.

The JVM and the Java compiler used on the IBM SP-2 machine were part of the JDK for AIX. The execution environment consisted of IBM's Parallel Operating Environment (POE), which supports the loading and execution of parallel processes across the nodes of the IBM SP-2. The machine is built of thin nodes with POWER2 Super Chip (P2SC) processors and 256 Mbytes of memory on each processor. The communication subsystem of the SP-2 features a high-performance switch which was used throughout the experiments. The NAS EP and IS benchmarks were also run on a 200 MHz dual Pentium Pro processor cluster running Linux Red Hat 6.0 on a 10baseT Ethernet. The same experiments were performed on a 14x336 MHz Ultra Sparc II processor Sun E4000 running Solaris 2.6. The LAM MPI library was used on the Linux cluster, whilst both the SP-2 and the E4000 provided native MPI libraries for message passing.

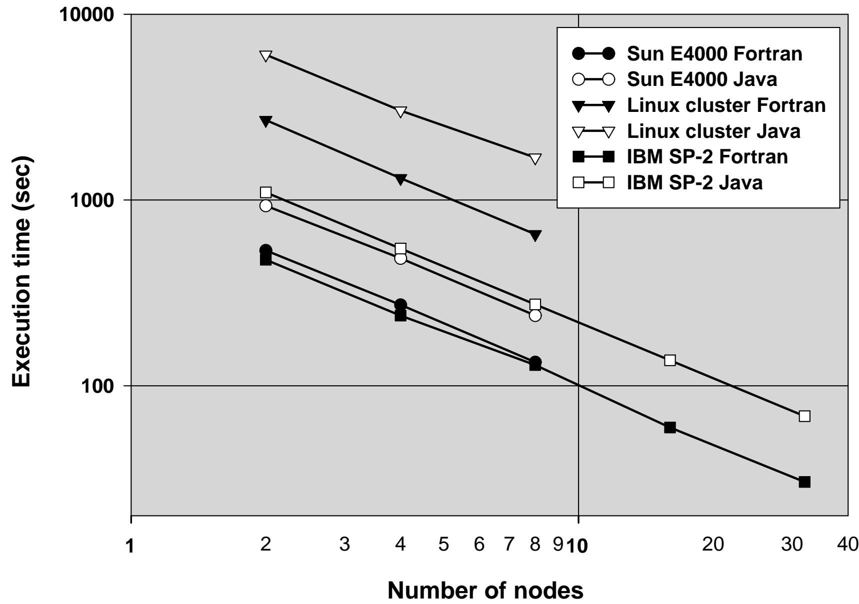

Fig. 5. Execution times for the NPB EP kernel (class B) on the IBM SP-2

A native code compiler for Java can be used instead of the JVM in order to overcome the above problem. Fortunately, rapid progress is being made in this area by developing optimizing Java compilers, such as the IBM High-Performance Compiler for Java (HPCJ), which generates native codes for the RS6000 architecture [?]. It works in the same manner as compilers for C, C++, Fortran, etc. and unlike JIT compilers, the static compilation occurs only once, before execution time. Thus, traditional resource-intensive optimisations can be applied in order to improve the performance of the generated native executable

code. In our experiments, we have used a version of HPCJ, which generates native code for the RS/6000 architecture. The input of HPCJ is usually a bytecode file, but the compiler will also accept Java source as input. In the latter case it invokes the JDK source-to-bytecode compiler to produce the bytecode file first. This file is then processed by a translator which passes an intermediate language representation to the common back-end from the family of compilers for the RS/6000 architecture. The back-end outputs standard object code which is then linked with other object modules and the previously bound legacy libraries to produce native executable code. Further experiments to evaluate the performance of the environment based on HPCJ have been carried out with the IS kernel on an IBM SP-2 machine. The results obtained are shown in Figure 6.

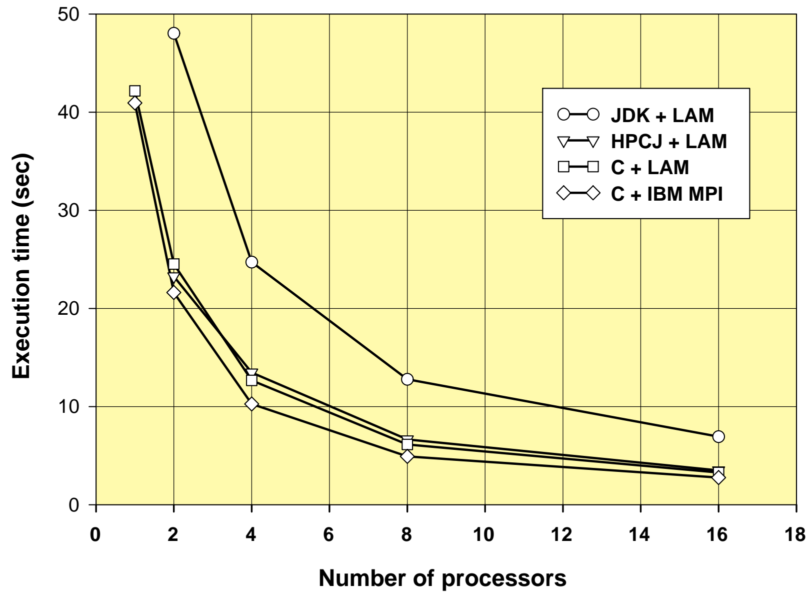

Fig. 6. Execution time for the NPB IS kernel (class A) on the IBM SP-2

Back in 1994, MPI-1 was originally designed with relatively static platforms in mind. To better support computing in volatile Internet environments, modern message passing designs for Java will have to support (at least) features such as dynamic spawning of process groups and parallel client/server interfaces as introduced in the MPI-2 specification. In addition, a natural framework for dynamically discovering new compute resources and establishing connections between running programs already exists in Sun's Jini project [41], and one line of investigation is into MPJ implementations operating in the Jini framework.

Closely modelled as it is on the MPI standards, the existing MPJ specification should be regarded as a first phase in a broader program to define a more Java-centric high performance message-passing environment. In future a detach-

ment from legacy implementations involving Java on top of native methods will be emphasized. We should consider the possibility of layering the messaging middleware over standard transports and other Java-compliant middleware (like CORBA). Of course, a primary goal in the above mentioned, both current and future work, should be the aim to offer MPI-like services to Java programs in an upward compatible fashion. The purposes are twofold: performance and portability.

4.4 Benchmarks

The Java Grande Forum has begun a benchmark initiative. The intentions are to make convincing arguments for Grande applications and to uncover the weaknesses in the Java platform. The responsibility for this initiative is being carried by the EPCC (Edinburgh) [11]. Currently a stable collection of non-parallel benchmarks exists in three categories:

- Basic operations are being timed (such as arithmetic expressions, object generation, method calls, loop bodies, etc.)
- Computational kernels: similar to the example of SciMark, numerical kernels are being observed. The IDEA-encryption algorithm is also in the collection.
- Applications: The collection is made up of an Alpha-Beta search with pruning, a Computational-Fluid-Dynamics application, a Monte-Carlo-simulation, and a 3D ray-tracing.

Thread benchmarks for all three categories are being worked on. For these purposes, the basic operations are being timed (create, join, barrier, synchronized methods); some of the applications (Monte Carlo and Ray-tracer) are being implemented in parallel. In addition, for quantitative language comparisons it is intended to provide equivalent implementations in C/C++.

5 Conclusion

Contributions of the Java Grande Forum are the keywords `strictfp` and `fastfp` for improved floating point arithmetic, work in the field of complex numbers, the multidimensional array package, the high-speed serialization, the fast RMI, and finally the benchmark initiatives.

Due to the cooperation with Sun Microsystems, due to the creation of a new branch of research, and due to the focussing of interests of the “Java Grande Community”, the future holds the hope that the requirements of high performance computing will be made a reality in Java.

6 Acknowledgements

This compilation of the activities and results of the Java Grande Forum used collected presentation documents from Geoffrey Fox, Dennis Gannon, Roldan Pozo and Bernhard Haumacher as a source of information. A word of thanks to Sun Microsystems, especially to Sia Zadeh, for financial and other support.

References

1. P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Automatic Loop Transformations and Parallelization for Java. Proceedings of *International Conference on Supercomputing*, Santa Fe, NM, May 2000.
2. David H. Bailey. A Fortran-90 Based Multiprecision System. *ACM Trans. Math. Softw.*, volume 21(4):379–387, December 1995. http://www.nersc.gov/ dhb/mpdist/mpdist.html
3. Ronald Boisvert, Joe Hicklin, Bruce Miller, Cleve Moler, Roldan Pozo, Karin Remington, and Peter Webb. JAMA: A Java Matrix Package. http://math.nist.gov/javanumerics/jama/.
4. John Brophy. Class `Jmath`. http://www.vni.com/corner/garage/grande/.
5. John Brophy. Class `Sfun`. http://www.vni.com/corner/garage/grande/.
6. C9x proposal. http://anubis.dkuug.dk/jtc1/sc22/wg14/ and ftp://ftp.dmk.com/DMK/sc22wg14/c9x/complex/.
7. B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12, 2000 (to appear).
8. Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with "HPJava". *Concurrency: Practice and Experience*, 9(6):579–619, June 1997.
9. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface for MPI. In *ACM 1999 Java Grande Conference*, pages 66–71, San Francisco, June 12–14, 1999.
10. Siddhartha Chatterjee. MPJAVA: A Multiple Precision Floating Point Computation Package in Java. http://www.cs.unc.edu/Research/HARPOON/mpjava/.
11. Java Grande Benchmarks. http://www.epcc.ed.ac.uk/javagrande
12. J. Rolim et al., editor. *Parallel and Distributed Processing*. Number 1586 in Lecture Notes in Computer Science. Springer Verlag, 1999.
13. *Proc. Workshop on Java for High Performance Network Computing at EuroPar'98*. Southampton, September 2–3, 1998.
14. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 9(6). John Wiley & Sons, June 1997.
15. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 9(11). John Wiley & Sons, November 1997.
16. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 10(11–13). John Wiley & Sons, September–November 1998.
17. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, to appear. John Wiley & Sons, 2000.
18. Edwin Günthner and Michael Philippsen. Complex numbers for Java. *Concurrency: Practice and Experience*, to appear, 2000.
19. Bernhard Haumacher and Michael Philippsen. More efficient object serialization. In *Parallel and Distributed Processing*, number 1586 in Lecture Notes in Computer Science, Puerto Rico, April 12, 1999. Springer Verlag.
20. http://www.npac.syr.edu/projects/pcrc/HPJava/.
21. Java Grande Forum. http://www.javagrande.org.
22. Java Grande Forum, mailinglist. All Members: javagrandeforum@npac.syr.edu, Subscribe: gcf@syracuse.edu.
23. Java Numerics web site. http://math.nist.gov/javanumerics/.

24. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
25. Glenn Judd, Mark Clement, Quinn Snell, and Vladimir Getov. Design issues for efficient implementation of MPI in Java. In *ACM 1999 Java Grande Conference*, pages 58–65, San Francisco, June 12–14, 1999.
26. William Kahan and J. W. Thomas. Augmenting a programming language with complex arithmetics. Technical Report No. 91/667, University of California at Berkeley, Department of Computer Science, December 1991.
27. Reinhard Klemm. Practical guideline for boosting Java server performance. In *ACM 1999 Java Grande Conference*, pages 25–34, San Francisco, June 12–14, 1999.
28. S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In M. Bubak, J. Dongarra, and J. Waśniewski, editors, *Recent Advances in PVM and MPI*, Lecture Notes in Computer Science, pages 135–142. Springer Verlag, 1997.
29. J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java Programming for High Performance Numerical Computing. IBM Systems Journal, vol. 39, no. 1, pp. 21-56, 2000.
30. J. E. Moreira, S. P. Midkiff, M. Gupta, and R. D. Lawrence. High Performance Computing with the Array Package for Java: A Case Study using Data Mining. In *Proceedings of SC '99*, Portland, OR, November 1999.
31. Michael Philippsen. Cooperating distributed garbage collectors for clusters and beyond. *Concurrency: Practice and Experience*, to appear, 2000.
32. Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, to appear, 2000.
33. Roldan Pozo and Bruce Miller. SciMark 2.0. http://math.nist.gov/scimark/.
34. Mark Roulo. Accelerate your Java apps! *Java World*, September 1998.
35. R. R. Oldehoeft S. Matsuoka and M. Tholburn, editors. *Proc. ISCOPE'99, 3rd International Symposium on Computing in Object-Oriented Parallel Environments.* Number 1732 in Lecture Notes in Computer Science. Springer Verlag, 1999.
36. P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors. *Proc. 7th Intl. Conf. on High Performance Computing and Networking, HPCN Europe 1999.* Number 1593 in Lecture Notes in Computer Science. Springer Verlag, 1999.
37. George K. Thiruvathukal, Fabian Breg, Ronald Boisvert, Joseph Darcy, Geoffrey C. Fox, Dennis Gannon, Siamak Hassanzadeh, Jose Moreira, Michael Philippsen, Roldan Pozo, and Marc Snir (editors). Java Grande Forum Report: Making Java work for high-end computing. In *Supercomputing'98: International Conference on High Performance Computing and Communications*, Orlando, Florida, November 7–13, 1998.
38. George K. Thiruvathukal, Fabian Breg, Ronald Boisvert, Joseph Darcy, Geoffrey C. Fox, Dennis Gannon, Siamak Hassanzadeh, Jose Moreira, Michael Philippsen, Roldan Pozo, and Marc Snir (editors). Iterim Java Grande Forum Report. In *ACM Java Grande Conference'99*, San Francisco, June 14–17, 1999.
39. Peng Wu, Sam Midkiff, José Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*, pages 109–118, San Francisco, June 12–14, 1999.

40. Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, (17):410–423, 1991.
41. Ken Arnold, Bryan O'Sullivan, Robert Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison Wesley, 1999.
42. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.