JavaGrande – High Performance Computing with Java

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

Abstract. The JavaGrande Forum is a group of users, researchers, and interested parties from industry. The Forum members are either trying to use Java for resource-intensive applications or are trying to improve the Java platform, making it possible to create large-sized applications that run quickly and efficiently in Java.

In order to improve its floating point arithmetic, the Forum has suggested to add the keywords `strictfp` and `fastfp` to the Java programming language It has has worked on complex numbers, multidimensional arrays, fast object serializations, and a high-speed remote method invocation (RMI). Results about the new field of research that has been started by the JavaGrande Forum have been recognized both internationally and within Sun Microsystems.

1 JavaGrande Forum

Inspired by coffee house jargon, the buzzword *Grande* applications became commonplace.[1] Grande applications can be found in the fields of scientific, engineering or commercial computing and distinguish themselves, due to their complex data processing or through their complex input/output demands. Typical application classes include simulation and data analysis. In order to cope with the processing needs, a Grande application needs high performance computing, if necessary even parallelism or distributed computing.

The *JavaGrande Forum* [17] is a union of those users, researchers, and company representatives, who either try to create Grande applications with the help of the Java programming language, or those who try to improve and extend the Java programming environment, in order to enable efficient Grande applications with Java. The JavaGrande Forum was founded in March 1998, in a "Birds-of-a-Feather" session in Palo Alto. Since then the Forum organizes regular meetings, which are open to all interested parties, as are the web site [17] and the mailing list [18]. The scientific coordinator is Geoffrey C. Fox (Syracuse); and the main contact person within Sun Microsystems is Sia Zadeh.

[1] Grande, as found in the phrase "Rio Grande" is the prevalent designation for the term large or big in many languages. In American English, the term Grande has established itself as describing size within the coffee house scene.

1.1 Goals of the JavaGrande Forum

The most important goals of the JavaGrande Forum are the following:

- Evaluation of the applicability of the Java programming language and the run-time environment for Grande applications.
- Bringing together the "JavaGrande Community", compilation of an agreed-upon list of requirements and a focussing of interests for interacting with Sun Microsystems.
- Creation of prototypical implementations that have community-wide consensus, interfaces (APIs) and recommendations for improvements, in order to make Java and its run-time environment utilizable for Grande applications.

1.2 Members of the JavaGrande Forum

The participants in the JavaGrande Forum are primarily American companies, research institutions, and laboratories. More recently, JavaGrande activities can be seen in Europe, as well [19].

In addition to Sun Microsystems, IBM and Intel, as well as Least Square, MathWorks, NAG, MPI Software Technologies and Visual Numerics are participants. Cooperation with hardware dealers is crucial, especially in reference to questions dealing with high-speed numerical computing. Academic circles are represented by a number of universities, such as Chicago, Syracuse, Berkeley, Houston, Karlsruhe, Tennessee, Chapel Hill, Edinburgh, Westminster and Santa Barbara, to name a few. The American Institute for Standardization (NIST), Sandia Labs and ICASE participate, as well.

The members are organized into two working groups. Section 3 presents the results of the numerical computing working group. Section 4 dedicates itself to the working group parallelism and distribution.

Former results and activities organized be the JavaGrande Forum have been well-received by Sun Microsystems. Gosling, Lindhom, Joy and Steele have studied the work of the JavaGrande Forum. Impressive public relations work was done by panelist Bill Joy, who praised the work of the JavaGrande Forum in front of an audience of almost 21,000 at the JavaOne 1999 Conference.

1.3 Scientific Contributions of the JavaGrande Forum

The Forum organizes scientific conferences and workshops or is represented on panels; see Table 1. The most important annual event is the ACM Java Grande Conference. A large portion of the scientific contributions of the "JavaGrande Community" can be found in the conference journals (Table 1) and in some issues of *Concurrency – Practice & Experience* [9–12]. In addition, the JavaGrande Forum publishes working reports at regular intervals [34, 35].

The scientific work is important for bringing the "JavaGrande Community" together and for creating cohesiveness. This make it possible to achieve consensus about ideas and to focus interests, thus making it easier to achieve the goals.

Table 1. Events organized by the JavaGrande Forum

Workshop, Syracuse, December 1996, [9]
PLDI Workshop, Las Vegas, June 1997, [10]
Java Grande Conference, Palo Alto, February 1998, [11]
EuroPar Workshop, Southampton, September 1998, [8]
Supercomputing, Exhibit and Panel, November 1998, [34]
IEEE Frontiers 1999 Conference, Annapolis, February 1999
HPCN, Amsterdam, April 1999, [33]
IPPS/SPDP, San Juan, April 1999, [7]
SIAM Meeting, Atlanta, May 1999
IFIP Working Group 2.5 Meeting, May 1999
Mannheim Supercomputing Conference, June 1999
ACM Java Grande Conference, San Francisco, June 1999, [12]
JavaOne, Exhibit and Panel, San Francisco, June 1999, [35]
ICS'99 Workshop, Rhodos, June 1999
Supercomputing, Exhibit and Panel, September 1999
ISCOPE, JavaGrande Day, December 1999, [32]
IPPS, Cancun, May 2000
ACM Java Grande Conference, San Francisco, June 2000

2 Why Java for high-performance computing?

In addition to the usual reasons for using Java that also apply to "normal" applications, such as portability, the existence of development environments, the (alleged) productivity-enhancing language design (with automatic garbage collection and thread support), and the existence of an extensive standard library; there are other important arguments for Grande applications.

The JavaGrande Forum designates Java as an universal language, which enables the mastering of the entire spectrum of tasks necessary for dealing with Grande applications. Java offers a standardized infrastructure for distributed applications, even in heterogeneous environments. Since the graphical interface is well-integrated (although there are a few portability problems) visualized can often be implemented as well. In addition, Java can also be used as glue code, in order to couple (interconnect) existing high-performance applications, to link computations that have been realized in other programming languages to one another, and as a universal in-between layer to function between computations and I/O. Another aspect that is no less important for the scientific and engineering fields, is that Java is being taught and learned – in contrast to Fortran for which a serious shortage of new recruits is expected.

2.1 Run-time performance and memory consumption

A commonplace and persistent rumor is that Java programs run very slowly. The first available version of Java (1.0 beta) interpreted ByteCode and was, in fact, extremely slow. Since then, much has changed: almost no Java program

is executed on a purely interpretive basis anymore. Instead, so-called Just-In-Time-Compilers of different coinage are being used to improve the execution times. The following comparisons give a good impression of the current state.

1. Comparative Experiment. Prechelt [30] conducted an experiment at the University of Karlsruhe, in which the same task was solved by 38 different people who were supposed to provide a reliable implementation. The experiment was not designed as a speed contest. As a result 24 Java programs, 11 C++ programs, and 5 C implementations were submitted. Amongst the 38 people taking part, all were computer science students at the post-graduate level with an average of 8 years and 100 KLOC of programming experience. The group included excellent to relatively poor programmers.

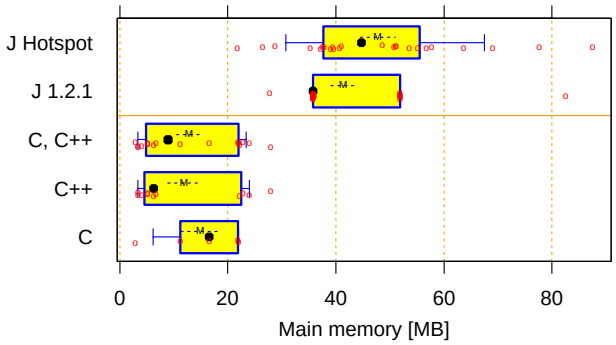

Fig. 1. Box plots of the main memory requirements of the programs; measured on Solaris 7 including data structures, program code, libraries, process administration and JVM. The individual measurements are symbolized by small circles. The M shows the mean, the black dot the median. Inside of a box, the innermost 50% of the measuremnts can be found, the H-line contains the innermost 80% of the measurements.

On average, Java (Version 1.2.1) needs four to five times as much main memory as C/C++, see Figure 1. For all languages, the interpersonal differences between the experiment participants were quite radical. The observed run-times varied from a matter of seconds to 30–40 minutes, see Figure 2. However, the medians of Java and C++ were similar. The variability within a language was much higher than the difference between the languages. The fastest five Java programs were five times faster than the median of the C++ programs; although it must be said that these programs were three times slower than the five fastest C++ programs. In this experiment, the C programs were clearly the fastest, a fact that could be explained by the fact that only few (and presumably the best) programmers selected C.

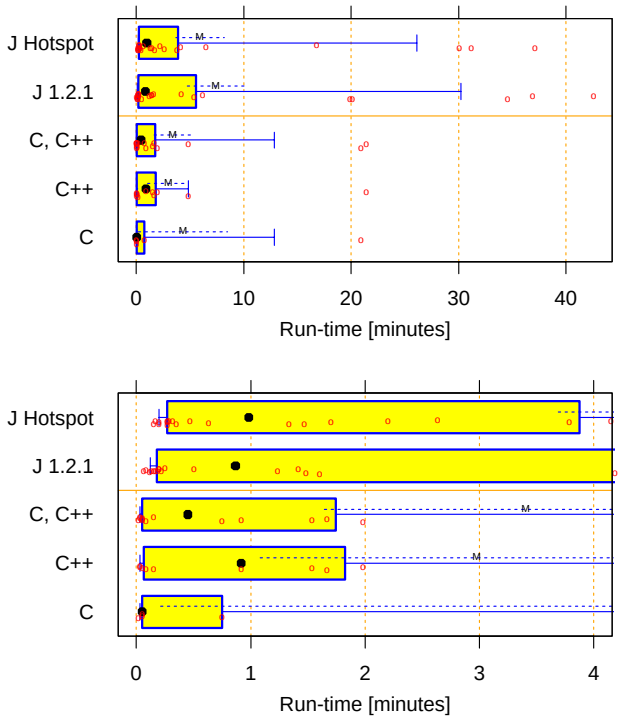

Fig. 2. Run-times of the different programs measured on Solaris 7. Enlarged in the second graph. The legend is the same as that of the previous figure.

In conclusion, it can be said that C++ showed almost no advantage over Java in terms of run-time. The ability differences of the programmers were greater than the differences in the programming languages. C++ still requires less main memory than Java.

2. Benchmark for Scientific Applications. Pozo and Miller combined five medium-sized numerical kernels in SciMark 2.0 [29] (compex valued fast Fourier transformation, Gauss-Seidel-relaxation, Monte-Carlo-integration of e^{-x^2}, multiplication of sparsely populated matrixes, and the LU-factorization of dense matrixes with pivoting). For each of these kernels, a Java and a C version are available. In addition, there is a version that fits in the cache, as well as one that does not fit.

The antiquated JVM 1.1.5 of the Netscape-Browser reaches approximately 0,7 MFlops on an Intel Celeron 366 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.

As a whole, the speed of Java is better than its reputation suggests and further optimizations are expected. In addition, there are papers that either describe how to optimize a Java application, or that discuss which rules to follow, so that a fast Java application is created from the outset, e.g. [24, 31]. Given these facts we feel that it is by no means out-of-place to seriously consider Java as a programming language for the central components of Grande applications. The results of both working groups of the JavaGrande Forum, which will be discussed below, provide support for this idea.

3 Numerical Computing Working Group

3.1 Goals of the Working Group

The Numerical Computing Working Group, that is supported by the IFIP Working Group 2.5 (International Federation for Information Processing), set its goal as the evaluation of the applicability of Java for numerical computing. Building upon that, the group aims to provide recommendations reached on a consensual basis and to implement them (at least prototypically), in order to eradicate deficiencies of the Java programming language and its run-time environment. The following sections discuss individual problems and results.

3.2 Improvement of the Floating Point Arithmetic

For scientific computing, it is important to reach acceptable speeds and precision on most types of processors. For Grande applications, it is additionally important to achieve a very high level of floating point performance on *some* processors. Numericians have learned to deal with architecture-specific Rounding errors since the beginning of floating point arithmetic, so that an exact reproducibility of bits is *seldom* of great importance – in contrast to Java's design goal of exact reproducibility of results. A too imprecise computation in the mathematical library (trigonometric functions, for example) is unacceptable, as is observable in many Java implementations up to this point in time.

Floating Point Performance. In order to achieve exact reproducibility, Java forbids common optimizations, such as making use to the associativity of operators, since these optimizations might cause differently rounded results. Further prohibitory measures affect processor features.

1. Prohibition to use the 80 Bit "double extended format". Processors of the x86-family have 80 bit wide registers that use the double extended format as defined in the IEEE Standard 754. To implement Java's bitwise reproducibility, every (temporary) result needs to be rounded to a more imprecise number

format. Even if the precision control bit is set to enforce a rounding in the registers after each step, still 15 bits instead of the standard 11 bits are used to represent the exponents. In the x86-family the exponent can only be rounded slowly by transferring it from the register to the main memory and re-loading it. Many JVM implementations ignore proper rounding for exactly this reason. In addition, the two-phase rounding (immediately after the operation and during the interaction with the main memory) leads in many cases to deviations from the specification on the order of 10^{-324}. A correction of this type of mistake is extremely time-intensive and also is not undertaken in most implementations. On the basis of both of these problems on Intel processors, Java's floating point arithmetic is either wrong or two to ten times slower than possible.

2. Prohibition to use for "FMA – Fused Multiply Add" machine commands. Processors of the PowerPC-family offer a machine command, that multiplies two floating point numbers and adds a further number to them. In addition of being faster that two single operations, only one rounding is needed. Java's language definition prohibits the use of this machine command and thus sacrifices 55% of the potential performance in experiments, see Figure 3. Without all optimizations, only 3,8 MFlop will be achieved. If all of the common optimizations are carried out (including the elimination of redundant boundary checks), then IBM's native Java compiler achieves 62% of the performance of an equivalent Fortran program (83,4 MFlop). If the FMA machine command could also be used – which the Fortran compiler does routinely – almost 97% of the Fortran performance could be achieved.

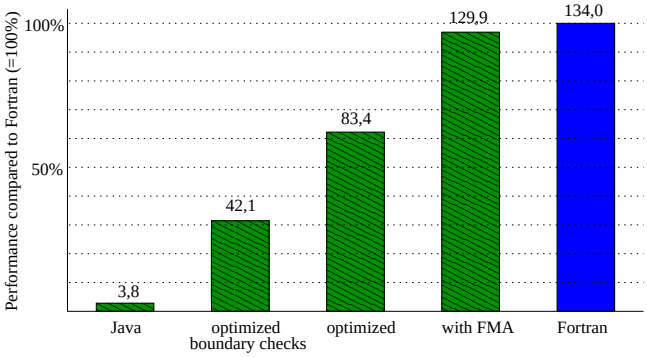

Fig. 3. Cholesky Factoring on a 67 MHz Power-2-Processor. Information given in MFlops and relative to optimized Fortran.

Keyword `strictfp` in Java 2. As suggested by the the JavaGrande Forum, the keyword `strictfp` was included in the version 1.2 of the Java programming

language. Only when this attribute is used on classes or methods, will the bitwise reproducibility of results be guaranteed. In the standard case without the keyword, Java can use the 15 bit long exponents for anonymous `double` variables to avoid the expensive memory copy and re-loading operations. In the standard case without the keyword, different numerical results may occur not only between processors with and without extended exponent lengths, but also between different JVM implementations on one platform. The reason for this is that the use of the extended exponents for anonymous variables is optional.

Reproducibility of `Math`-Functions In addition to the strict prohibitions that limit floating point performance, Java's floating point arithmetic suffers from yet another problem regarding precision. According to the Java specification (version 1.1.x), the `java.lang.Math` library must be implemented by porting the `fdlibm` library.[2] Instead of this, most manufacturers use the faster machine commands offered by their hardware that often return slightly incorrect results, though. In order to face up to this development, Sun offers two wrapper methods for the `fdlibm` functions realized in C, which should make it simpler to achieve `fdlibm` results on all platforms. The first complete implementation of `fdlibm` in Java is due to John Brophy of Visual Numerics, making it possible to forswear the wrapper methods in the future [1].

An ideal math library would deliver foating point numbers, that at the most deviate 0,5 ulp (unit in the last place) from the actual result,[3] this means they have been as well-rounded off as possible. The `fdlibm` library itself only delivers a precision of 1 ulp, for which reason Sun decided to change the specification in Java 1.3, so that the results of the math library would be allowed a fault tolerance level of 1 ulp. Abraham Ziv, IBM Haifa, created a math library (in ANSI C), that rounds correctly, i.e., with a fault level 0,5 ulp at the most [37].

Recommendation of the keyword `fastfp`. The JavaGrande Forum is currently working on a JSR (Java Specification Request [21]) that recommends integrating a further modifier, `fastfp`, to the Java programming language. In classes and methods, that are labeled with this modifier, the use of the FMA machine command will be allowed. In addition, the math library should be extended with an additional FMA method, whose use forces the FMA command to execute. Currently, the sensibility of allowing associativity optimizations via this modifier is being examined.

3.3 Efficient complex Arithmetic

A requirement for the use of Java for scientific computing is the efficient and comfortable support of complex numbers. With Java's expressiveness however,

[2] 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.

[3] For numbers between 2^k and 2^{k+1}, 1 ulp = 2^{k-52}.

complex numbers can only be realized in the form of a `Complex` class, whose objects contain two `double` values. Complex valued arithmetic must then be expressed by means of complicated method calls, as in the following code fragment.

```
Complex a = new Complex(5,2);
Complex b = a.plus(a);
```

This has three disadvantages: first, arithmetic expressions are difficult to read without operator overloading. Secondly, complex arithmetic is slower than Java's arithmetic on primitive types, since it takes longer to create an object and more memory space is necessary than for a variable of a primitive type. In addition, temporary helper objects need to be created for almost every method call. In contrast, primitive arithmetics can use the stack to pass results. IBM analyzed the performance of class-based complex arithmetic. Using a Jacobi-relaxation, an implementation based on a class `Complex` achieved only 2% of the implementation that used two `double` instead. Thirdly, 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 necessary for a class-based solution, where a literal would be sufficient to represent a constant value.

Since scientific computing only makes up an 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 best way of introducing complex numbers in Java. In addition, it is not known, if (and if so, when) Java will be extended by general operator overloading and value classes. And since even after such an extension value classes can still not be seamlessly integrated into the system of existing primitive types, the JavaGrande Forum regards that the following twin-track way to be more sensible (see Figure 4).

Class `java.lang.Complex`. The JavaGrande Forum has defined and prototypically implemented 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. This class will be submitted to Sun in the form of a JSR.

IBM has built the semantics of this `Complex` class permanently into its native Java compiler; internally a complex type is used and the usual optimizations are carried out on it [36]. In particular, most of the method and constructor calls that prevail in the Java code using this class are replaced by stack operations. Hence, at least for the platforms that are supported, class `Complex` is supported efficiently.

Primitive Data Type `complex`. As a further JSR, the JavaGrande Forum suggests a primitive type `complex` with corresponding infix operations. In order

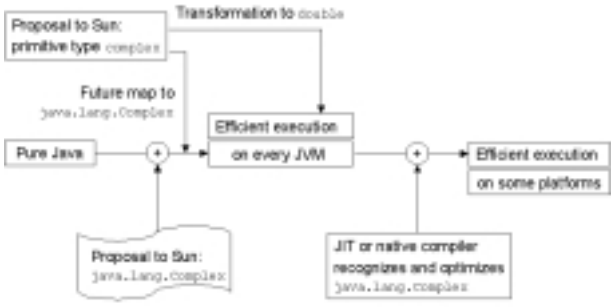

Fig. 4. Introduction of complex numbers and arithmetics

that both the ByteCode format and the existing JVM implementation do not have to undergo changes, the language extension are transformed back to normal Java in a pre-processor step. Currently it is being discussed whether it is necessary to introduce a further primitive data type `imaginary` in conjunction with the primitive data type `complex`, as it has been done in C99 [23, 2].

Figure 4 shows two alternative transformations. First, the primitive data type `complex` is mapped to a pair of `double` values. Secondly, it is mapped to the `Complex` class to make use of the above mentioned compiler support. The compiler *cj* developed at the University of Karlsruhe both formally describes and prototypically realizes the transformation of the primitive type `complex` [13, 14].

3.4 Efficient multi-dimensional Arrays

In the same way as efficient and comfortable complex arithmetic must be made available, numerical computing without efficient (i.e., optimizable) and comfortable multi-dimensional arrays is unthinkable.

Java offers multi-dimensional arrays only as arrays of one-dimensional arrays. This causes several problems for optimization. Especially since index vectors cannot be mapped to memory locations. One problem is that several rows of such a multi-dimensional array could be aliases to a shared one-dimensional array. Another problem is that the rows could have different lengths. Moreover, each field access to a multi-dimensional array not only requires a pointer indirection but also causes several 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 JavaGrande Forum recommends a class for multi-dimensional arrays (once again in the form of a JSR), which would be mapped to one-dimensional Java arrays with a specific roll-out scheme. The compiler can then use common optimizations; algorithms can make use of the roll-out scheme to improve their cache locality.

As with the class-based complex numbers, this multi-dimensional array class requires awkward access methods instead of elegant `[]`-notation. For this reason, a twin track solution is appropriate in this case, as well. On the one hand, IBM will build permanent support for multi-dimensional arrays into their native Java compiler. At the same time, an extended array access syntax will be set up, allowing elegant access to multi-dimensional arrays. This syntactical extension, which is quite difficult due to the necessary interaction with regular one-dimensional Java arrays, and the corresponding pre-processor will be recommended to Sun in JSR form.

3.5 More fundamental Issues

Lightweight classes and operator overloading are the general solutions for both complex numbers, as well as for multi-dimensional arrays. Lightweight classes have value semantics, their instantiated variables cannot be changed after object creation. Thus, the problem of equality semantics does not come up at all. In addition, lightweight objects can often be allocated on the stack and by passed by copy. If the programmer could overload basic operators like `+`, `-`, `*`, `/`, `[]`, a pre-processor would no longer be necessary.

Why does the JavaGrande Forum not try to introduce lightweight classes and operator overloading? The answer is quite pragmatic. The JavaGrande Forum hopes the above-mentioned JSRs are light enough to withstand the formal process of language alterations. The community of normal Java users remains almost completely unaffected and quite possibly will not notice the changes at all. Very few of today's Java users are even aware of the existence, much more the importance, of the `strictfp` keywords. The smaller the number of the people affected, the less-damaging the endorsement of JSR will be.

Value classes and operator overloading demand a greater change to the language as a whole and (supposedly) impact the ByteCode format and thus, the JVM. For this reason and due to the almost religious character of operator overloading, the outcome of such efforts remains open to speculation, while the above recommendations have better prospects.

4 Parallelism and Distribution Working Group

4.1 Goal of the Working Group

The Parallelism and Distribution Working Group of the JavaGrande 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 band widths 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 [20], 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.[4] Figure 5 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.

UKA Serialization. The UKA serialization [15] 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 JavaGrande 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.

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

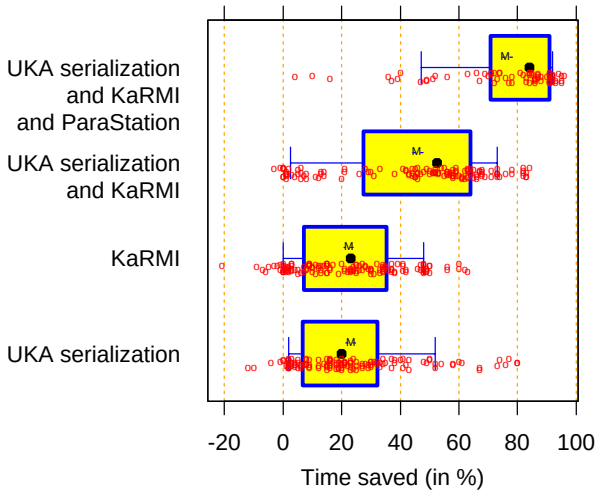

Fig. 5. The bottom three box plots each show 2·64 measured results for diverse benchmarks (64 points represent Ethernet on PC, 64 stand for FastEthernet on Alpha). The first (bottom-most) box plot shows the run-time improvement that was achieved with regular RMI and the UKA serialization. The second 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).

- 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.
- Sun remains steadfast about layering for reasons of clearer object-oriented design, is though, 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 [26, 28] 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. Based upon the work of the Java-Grande Forum, Suns plans to make this possible (still being outlined) for 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 tis 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 [27], the official RMI sees no alternative garbage collector as being necessary, in contrast to KaRMI.

4.3 Message Passing in Java

While Java's mechanisms for parallel and distributed programming are based on the client/server paradigm, MPI is a symmetrical, message-based parallel calculation model. In order to ensure that the MPI-based solutions are not lost during the transition to Java, a sub-group of the JavaGrande Forum is working on providing an MPI binding for Java. The availability of such an interface would make MPI-based Grande applications possible. Members of the JavaGrande Forum have made the following recommendations for that reason:

- mpiJava: a collection of wrapper classes that reach back to the C++ binding of MPI through the native Java interface (JNI) [4, 16].
- JavaMPI: automatically created JNI wrappers for a program with a C binding to MPI [25].
- MPIJ: an implementation of MPI interfaces in Java, that is based on the C++ binding and demonstrates relatively good performance results [22].

At the moment, the sub-group is working on the unification of previous prototypes [3]. One of the major issues that has arisen, is how the mechanisms of Java can be made useful in association with MPI. Under investigation is whether the types used in MPI could be extended with Java objects, which could then be sent in serialized form [5]. In addition, it is being studied if and how Java's thread model can be utilized to extend the process-based approach of MPI.

4.4 Benchmarks

The JavaGrande 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) [6]. 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 JavaGrande Forum are the keywords `strictfp` and `fastfp` for improved floating point arithmetic, work in the field of complex numbers, 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 "JavaGrande 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 JavaGrande Forum used collected presentation documents from Geoffrey Fox, Dennis Gannon, Roldan Pozo and Bernhard Haumacher as a source of information. Lutz Prechelt made Figures 1 and 2 available. A word of thanks to Sun Microsystems, especially to Sia Zadeh, for financial and other support.

References

1. John Brophy. Implementation of `java.lang.complex`. http://www.vni.com/corner/garage/grande/.
2. C9x proposal. http://anubis.dkuug.dk/jtc1/sc22/wg14/ and ftp://ftp.dmk.com/DMK/sc22wg14/c9x/complex/.
3. B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. MPI for Java: Draft API specification. Technical Report JGF-TR-003, JavaGande Forum, November 1998.
4. 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.
5. 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.
6. Java Grande Benchmarks. http://www.epcc.ed.ac.uk/javagrande
7. J. Rolim et al., editor. *Parallel and Distributed Processing.* Number 1586 in Lecture Notes in Computer Science. Springer Verlag, 1999.
8. *Proc. Workshop on Java for High Performance Network Computing at EuroPar'98.* Southampton, September 2–3, 1998.
9. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 9(6). John Wiley & Sons, June 1997.

10. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 9(11). John Wiley & Sons, November 1997.
11. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, volume 10(11–13). John Wiley & Sons, September–November 1998.
12. Geoffrey C. Fox, editor. *Concurrency: Practice and Experience*, to appear. John Wiley & Sons, 2000.
13. Edwin Günthner and Michael Philippsen. Komplexe Zahlen für Java. In *JIT'99, Java-Informations-Tage*, pages 253–266, Düsseldorf, September 20–21, 1999. Springer Verlag.
14. Edwin Günthner and Michael Philippsen. Complex numbers for Java. *Concurrency: Practice and Experience*, to appear, 2000.
15. 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.
16. http://www.npac.syr.edu/projects/pcrc/HPJava/.
17. Java Grande Forum. http://www.javagrande.org.
18. Java Grande Forum, mailinglist. All Members: javagrandeforum@npac.syr.edu, Subscribe: gcf@syracuse.edu.
19. Java Grande Forum, Europe. http://www.irisa.fr/EuroTools/Sigs/Java.html.
20. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
21. The Java community process manual. http://java.sun.com/aboutJava/communityprocess/java_community_process.html.
22. 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.
23. 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.
24. Reinhard Klemm. Practical guideline for boosting Java server performance. In *ACM 1999 Java Grande Conference*, pages 25–34, San Francisco, June 12–14, 1999.
25. S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In M. Bubak, J. Dongarra, and J. Wańiewski, editors, *Recent Advances in PVM and MPI*, Lecture Notes in Computer Science, pages 135–142. Springer Verlag, 1997.
26. Christian Nester, Michael Philippsen, and Bernhard Haumacher. Ein effizientes RMI für Java. In *JIT'99, Java-Informations-Tage*, pages 135–148, Düsseldorf, September 20–21, 1999. Springer Verlag.
27. Michael Philippsen. Cooperating distributed garbage collectors for clusters and beyond. *Concurrency: Practice and Experience*, to appear, 2000.
28. Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, to appear, 2000.
29. Roldan Pozo and Bruce Miller. SciMark 2.0. http://math.nist.gov/scimark/.
30. Lutz Prechelt. Comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM*, 42(10):109–112, October 1999.
31. Mark Roulo. Accelerate your Java apps! *Java World*, September 1998.

32. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, (17):410–423, 1991.