In Euro-PDS'98, 2nd European Parallel and Distributed Systems Conference, Vienna, Austria, July 1–3, 1998.

Is Java ready for computational science?

Michael Philippsen
Computer Science Dept., University of Karlsruhe, Germany
phlipp@ira.uka.de

Abstract

This paper provides quantitative and qualitative arguments that help to decide whether Java is ready for computational science. Current shortcomings of Java as well as appropriate countermeasures are discussed.

1 Introduction

In the computational science community, there are two large groups of people. Let us call them F and O. People of group F stick to their respective favorite version of Fortran. People of group O are convinced that object-oriented programs are easier to develop and to maintain and hence actively use C/C++ to solve some of their problems.

Members of group F prefer procedural Fortran programming because they either have a huge body of existing Fortran programs or they doubt that a language that is not called "Fortran" can ever beat Fortran's performance or its expressiveness.

Since assembly-style C/C++ code can achieve sufficient runtime performance, group O eventually got accredited. Members of group O have learned to use, to abuse, and to dislike C/C++ and often don't see a point of starting over with another language, especially since C/C++ can be customized by macros and templates.

Currently there is a core of people that consider to found a third group J using Java as a main language for computational science. Similar to the situation when group O was founded, it is doubted whether sufficient performance can be reached with the new language, whether its expressiveness is appropriate, etc. Whereas people of group O had only one line of defense, group J is bashed from two sides.

Bashing is not the purpose of this paper. Instead, this paper provides quantitative and qualitative arguments. Java's disadvantages are discussed, the likelihood and impact of hypothetic future fixes are estimated, trends are shown. On the other hand, several advantages of Java are mentioned.

For the remainder of this paper we take for granted that object-oriented programming is preferable from the software engineer's point of view. Object-oriented approaches result in better-designed code, in better maintainability, and in better chances of code re-usability. Given these assumptions, the open questions are discussed below. Section 2 discusses performance, section 3 focuses on the expressiveness, section 4 covers parallel and distributed programming, section 5 discusses problems related to Java's rules of arithmetic and section 6 collects other aspects.

2 Performance

We compare Java's performance to performance of Fortran90 and HPF in two benchmark experiments. The first benchmark is the Sieve of Eratosthenes for finding prime numbers. The Sieve is implemented in all three languages and is timed on a single workstation. The second benchmark targets large-scale geophysical algorithms, that have been studied in [8]. We cooperated with the Stanford Exploration Project [3] in implementing parallel versions of these algorithms in all three languages. Runtime were measured on an IBM SP/2 (distributed memory parallel architecture) and on a SGI Origin2000 (shared memory parallel architecture).

2.1 Sieve Performance

The Sieve was run on an IBM RS/6000 workstation. We used the Java Development Kit (JDK) 1.1.2, IBM's Fortran90 compiler, and Portland Group's High-Performance Fortran, version 2.2.

Java performance was measured in three contexts. We first timed the runtime of interpreted bytecode. Then we switched on the just-in-time compiler to speed up the interpretation. These are the two standard approaches that can be used with any JDK.

Every Java virtual machine (JVM) performs array boundary checks at runtime to ensure proper accesses. In case of an illegal access, the JVM raises a runtime exception coupled with a stack trace that points out the line number in the source code that has caused the bug. Array boundary checks have been introduced into Java because of applet security reasons. They are a valued instrument during debugging, but are often blamed for slowing down performance.

For RS/6000 workstations, IBM has an alpha version of a Java compiler, called High Performance Compiler for Java (HPJ) [7] that compiles Java code to native code. HPJ has an option to switch off array boundary checks, which we used in our measurements.

	Fortran		**Java**		
size ·E6	*F90*	*HPF*	*Java*	*Java JIT*	*HPJ w/o rt-checks*
0.1	20	20	1002	52	14
0.2	40	38	1888	107	30
0.5	440	323	4349	584	444
1.0	1080	807	8195	1393	1089
10.0	11790	9660	57162	20195	11690

Problem size ∈ [0.1E6, 10E6]; measurements in seconds.

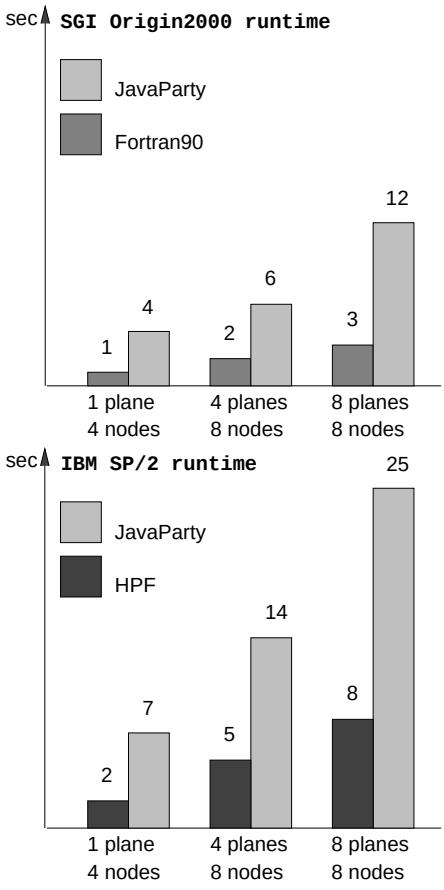
There are differences between the two Fortran versions: HPF seems to do better code optimization.

Interpreted Java (first Java column) is significantly slower than the Fortrans. For small problem sizes, the slowdown factor is outrageous, probably due to the overhead caused by starting the Java virtual machine. For a size of 10E6, the slowdown factor is down to about 6. The just-in-time compiler (second Java column) boosts Java performance significantly and achieves a slowdown factor of less than 2.5 compared to either Fortran. Hence, standard Java technology does not perform as badly as often claimed.

IBM's native Java compiler (third Java column) achieves almost the *same* performance as Fortran, although it is just an alpha version.

2.2 Application from Geophysics

We have studied large-scale geophysical algorithms, called Veltran velocity analysis and Kirchhoff migration to evaluate Java's efficiency. These are basic algorithms used in geophysics for the analysis of the earth's sublayers by means of sound wave reflection. Since it can take tera bytes of input data to cover a reasonable area, the performance of these algorithms is crucial. The geophysics and the details of the benchmarks can be found in [8].



We have implemented these algorithms in a Java environment, in HPF, and in Fortran90, and benchmarked the programs on up to 8 nodes of an SGI Origin2000 shared memory computer and on 8 nodes of an IBM SP/2 distributed memory parallel machine. On the SGI, we used SGI's standard Fortran90 compiler; on the IBM SP/2, version 2.2 of the Portland Group High Performance Fortran compiler was used. The Java implementation on both machines uses the JavaParty [9, 13] distributed runtime and communication system, which itself is written in and generates Java 1.1.5.

On the SGI, our JavaParty implementation is slower than the equivalent Fortran90 program by a factor of about 4. On the SP/2, JavaParty faces a slowdown by 3. In part, the slowdowns are due to Java's mandatory and implicit array boundary checking.

The JavaParty program automatically adapts both to the number of planes to be processed and to the number of nodes available. The Fortran programs did not have the same adaptability. Instead, we had to change some constants and recompile the Fortran code for each of the measurements. Without the manual changes and recompilation, the performance of the most general and slowest program would have shown up repeatedly. For example, to process one plane on four nodes with the general program, the Origin2000 needs 3 seconds and the SP/2 needs 8 seconds.

2.3 Performance Outlook

These benchmarks are not representative. However, it is possible to reason about the future trends of Java's performance.

We expect significant performance increases for Java in the near future because of two main reasons. First, we had to use JDK 1.1.x, because later releases are not yet available for our hardware platforms. Later versions (JDK 1.2, HotSpot) have increased performance (especially RMI performance, improved native thread support and just-in-time compilation) on Solaris and Wintel platforms and are likely to show the same effect in our environments. Second, compilers producing optimized native code like IBM's High Performance Java Compiler [7] are on the horizon. These compilers will approach Fortran performance because they can apply much more sophisticated optimization techniques than current just-in-time compilers. They will outperform C++ compilers in a few years. When comparing Java to C++ from the compiler writer's perspective, it is obvious that Java's control and data flow is much easier to analyze. Since there is no pointer arithmetic in Java, aliasing analysis is simpler and can produce more exact knowledge on control and data flow. Therefore, traditional optimization techniques, as for example code and object inlining, dispatch optimization, register allocation, etc., can be applied more often and presumably with more effect. As Budimlic and Kennedy stress in [2] the only serious difficulty is that Java's elaborate exception framework interferes with some optimization.[1]

Given that assumption and given the results of a recent study by Veldhuizen and Jernigan [16] who benchmarked Kuck and Associates' C++ compiler KAI-C++ and demonstrated that KAI-C++ can generate faster-than-Fortran code, it is reasonable to conclude, that Java might as well outperform Fortran.

[1] There are other areas in the current JDK that need improved performance. Most notably, thread and synchronization performance of the JVM and performance of RMI and serialization. These aspects are discussed in section 4.

But even if Java and/or C/C++ remain slower than Fortran on a runtime scale, economic costs must be considered. A comparative controlled experiment is needed, where scientific code is developed and maintained twice, both in procedural Fortran and with an object-oriented approach. The results of that study can help to put development cost in relation to relative runtime improvement. Unfortunately, we do not know of any such experiment.[2]

3 Expressiveness

3.1 Complex Numbers

Java supports efficient implementation of primitive types such as `boolean`, `char`, `int`, `long`, `float`, and `double`. Primitive types do not need more memory than necessary. They are neither inherited from the root of the class hierarchy nor are they accessed indirectly through a pointer (or pointers). However, for computational science it is unacceptable that Java does not offer complex numbers as primitive type.

With current Java, programmers can only fill that gap by defining a class `Complex` of their own (or by downloading such a class from the Web [17]). This approach has the following disadvantages.

• **Object creation cost.** Object creation in Java is expensive since each object needs some memory that must be initialized and registered for garbage collection, and an additional lock object must be created. Hence, for performance reasons numerous or frequent creation of tiny objects must be avoided, especially, since most of the costly features of objects are not needed in calculations with complex numbers.

• **Object access cost.** Whereas variables of primitive types can be referred to internally by a simple address in memory, object access requires a pointer indirection.[3]

• **Awkward arithmetic expressions.** The programmer must use methods to express arithmetic expressions, since infix operations are only defined for existing primitive types. For complicated arithmetic expressions the resulting code is harder to decipher than with infix operations.

• **Math library problems.** The JDK comprises a class `java.lang.Math`. This library contains methods for performing basic numeric operations such as min/max, the elementary exponential, logarithm, square root, and trigonometric functions. Most of the methods are offered several times, once for each relevant primitive type to be selected by static of dynamic dispatch. There are of course no versions for complex numbers in that library. But unfortunately, they cannot even be added by means of inheritance, since the class is declared `final`. Thus, if the programmer decides to use his own `Complex` class, he will find min and max methods for all primitive types in `java.lang.Math`, except for his own implementation of min and max in class `Complex`.

[2] We are interested to hear suggestions of experimental settings of manageable size that can be used for such a study.

[3] Early Java virtual machine implementations needed a double indirection to implement correct garbage collection. Performance has improved since then.

Adding complex numbers as primitive types to Java is neither costly for compiler writers nor does it cause compatibility problems with existing distributions. When complex numbers are added, corresponding infix operations are instantly available, the library problem will be trivial to fix.

The more general solution of adding a special type of `value` (i.e. pass-by-value, lightweight, or in-line) classes that allows the efficient implementation of primitive types has two disadvantages. First, they require more extensions of Java since general operator overloading is needed. Second, it might result in a redefinition of existing primitive types because of orthogonality.

It seems to be the only real problem that complex numbers are predominantly needed by the computational science community, which is rather small in size.

3.2 Generic classes

Java does not offer generic classes. This is especially painful when using container classes because of reduced performance and reduced type checking. Moreover, generic classes are a prerequisite for elegantly adding operator overloading to the language, as will be discussed in section 3.3.

In Java, container types must be based on class Object, which is the root of the class hierarchy. Whenever an object is added to the container, the knowledge about its type is lost. When the object is extracted from the container, the programmer must cast it down to its original type. The runtime check Java uses to guarantee the validity of this cast is time consuming.

If instead one could express generic container classes, that are specialized to a specific type upon instantiation, these runtime checks can be eliminated. Moreover, the static compile time type checking can prevent bugs that otherwise only show at runtime.

Adding generic classes to Java is non-trivial, since the new type system must be well-defined and contain the existing type system as special case. Another problem is that it is undesirable to change the bytecode or to alter Java's approach to separate compilation. Several solutions have been proposed, the Pizza type system is one of those [10], and there are rumors that generic classes are strong candidates for a future extension of Java. Compared to complex numbers, the demand for generic classes is not restricted to computational scientists.

In the meantime, Pizza's generic classes can be used. The Pizza system [10, 14] provides a source-to-source transformation of Java with generic classes into regular Java.

3.3 Operator Overloading

When complex numbers are introduced into Java, the corresponding infix operator will be introduced as well. This section discusses Java's lack of operator overloading for reference types.

For computational science there must be infix operators for arrays (and matrices) to cover demands from the Fortran group. If it is missing, development and maintenance are more difficult, since otherwise clear formulae must be distorted by long method names and awkward orderings of operands.

Although is it technically quite simple to add any form of operator overloading for reference types to a Java com-

piler, it is difficult to find a form that is suitable and that is not in conflict with Java's design goals. The key problem is that operator overloading can always be abused to write code that is difficult to understand and maintain. An appropriate form of operator overloading may guide programmers away from abuse.

Java's philosophy will not permit overloading of primitive type operators. No Java programmer will ever be able to redefine the meaning of a `+` between two `float` values. Similarly, array indexing or pointer dereferencing, and equality testing are unlikely to be candidates for overloading, although admittedly, there might be some rare situations where this type of overloading actually can improve code quality.

For reference types, an approach used in the Sather language [11, 12] seems reasonable. For an infix operator $\square$ in a class `A`, the programmer can provide a method that takes a `B` as an argument and returns a `C`. The signature looks like `C □-Op(B)`. Whenever the compiler sees an infix operator $\square$ between two reference types, e.g. `a` $\square$ `b`, this operator is taken to be syntacic sugar for a method invocation, like `a.□-Op(b)`. Hence, regular dispatch rules can be applied to operator overloading as well.

Two types of questions remain: First, does $\square$ stand for regular arithmetic operators (`+`, `-`, `*`, ...) and/or will it be possible to define methods for other (Unicode-) symbols,[4] e.g. $\cdot$, $\times$, $\div$, ...? Second, what is an appropriate name of the method `□-Op`? The name should be selected in a way to guide away programmers from abuse. It will probably be best, if the operator symbol appears textually in the method name.

In conclusion, operator overloading for complex numbers is for free if complex numbers are added as primitive types. Operator overloading for reference types is technically simple but needs to be carefully designed to reduce abuse.

3.4 Arrays

Java's arrays are not based on a consecutive memory layout that is visible to the user. Therefore, the mechanisms to access subsections of arrays differ from their counterparts in Fortran and C/C++.

The only way to extract a subsection of a one-dimensional array in Java is to use `arraycopy` and actually create a copy. The copy operation is time consuming although JVM implementations usually offer efficient native routines for that purpose. More significantly however, the necessity to actually copy array data might use up the available memory, especially in data intensive applications.

Since multi-dimensional arrays are implemented as arrays of array objects in Java, multi-dimensional arrays cannot be unrolled into one-dimensional arrays simply by casting (and vice versa). Instead, `arraycopy` must be used again. As in C/C++, it is easy to access a single dimension of an array in Java, but it is more difficult to access the other dimension. In C/C++, this problem can be solved by pointer arithmetic or by overloading the array access operators, both of which is not possible in Java.

Since it requires significant changes of the language, the JVM, and the garbage collector, it is unlikely, that either special syntax for multi-dimensional array will be added to Java or a special `new` operator will be added that requires contiguously stored multi-dimensional array data.

As a consequence, a modular design must be used to implement filtered accesses to multi-dimensional arrays in Java: array elements and the corresponding index transformations are hidden behind well-defined classes. Although such a design is preferable from the software engineer's point of view, C/C++ people claim that it results in poor performance and awkward accessor routines.

Due to Java's relative simplicity, an aggressive optimizer can effectively inline index computations and can successfully try to eliminate invariant expressions [1]. We expect that much of the performance loss encountered in C/C++ implementations that use the additional layer of abstraction, can be compensated in Java by a clever optimizer.

Array boundary checking for every single array access is expensive and—at the same time—one of the main reasons for the reliability of Java programs. Either compiler techniques will be used to prove that certain array boundary checks can be avoided, or JVM's will need an option to switch off these checks.

In conclusion, Java's array mechanisms coupled with some library support for multi-dimensional arrays will not prevent sufficient performance. However, due to the additional layer of abstraction, the code will look different from what it looks like in today's Fortran and C/C++ programs. The necessary compiler optimizations techniques are known and they can be used more often and more effectively than they can be in C/C++ code.

Although having Fortran90 style array section notation (`A[`*first*:*last*:*increment*`]`) would increase code readability, it is unlikely to be incorporated into Java. Since most users of Java are not interested in computational science, it appears unrealistic to hope for section notation that would require significant extensions of both the Java compiler and the bytecode.

3.5 Standard Libraries

Java is often accused of lacking standard libraries that are commonly used in computational science. However, if group J grows in size, standard libraries will become available.

In the meantime, the Java Native Interface (JNI) offers a platform independent way to call native code, i.e., to use existing implementations of standard libraries through wrapper classes. Although that approach is not acceptable in the long run, it is sufficient for flying a kite. The good news is that recently, such libraries have begun to appear, see for example [17].

4 Parallel and Distributed Computing

Java has standardized threads and synchronization in the language and offers portable libraries for distributed programming. This is an essential advantage over other languages. However, there are some performance problems to be solved.

• **Distribution.** Since there are MPI and PVM libraries for Java, Java can be used for low level distributed program-

[4] This is a suggestion of Guy Steele.

ming much like Fortran or C/C++. Similar to C/C++, Java also offers socket communication.

But in addition, the JDK includes a remote method invocation package (RMI) that can be used to distribute regular Java objects across the network. RMI thus is an object-oriented mechanism for distributed programming, providing a standard remote procedure call mechanism for objects, coupled with a remote garbage collector. Although similar libraries are available for C/C++, only RMI is fully portable.

The main problem is that the performance of RMI is weak and has some seemingly inexplicable drops in efficiency presumably because of internal buffering. The closely related serialization used for marshaling and unmarshaling of data structures is known to be slow as well. A better understanding, thorough benchmarking, and detailed documentation of the performance would help.

• **Threads.** Few people from the C/C++ group use threads. This is because thread programming usually results in non-portable programs, and because synchronization is difficult. Java's threads do not alleviate the difficulty of synchronization, but they are truly portable.

On the other hand, existing implementations of truly parallel threads on multi-processor shared memory machines currently do not perform well. There are performance problems with thread synchronization and scalable thread support in the JVM.

5 Rules of Arithmetic

Java has well-defined arithmetic operators. There seems to be nothing that is left open as an implementation decision. Hence, Java arithmetic is fully portable and computes the same results on any platform. Yet, there are differences between Java and both C/C++ and Fortran.

• **Integer Division.** One of problems with C/C++ has been solved in Java by definition. For the division of two integers it is not well defined in C/C++ whether this operation will round towards zero or towards -infinity, when either or both operands are negative. In Java, integer division rounds towards zero (exception for overflow conditions).

• **Floating Point Promotion.** Darcy and Kahan pointed out in [4] that there are cases where Java's arithmetic computes results that are different from Fortran's, although both languages are based on the IEEE 754 standard. The differences can only be noticed if `float` values are used and are caused by a loss of precision in temporary expressions.

Given a tree representation of an expression, Java analyzes the type information of the two operands for every binary operation. When at least one of them is a `double`, the other is widened to a `double` as well. If both operands of a binary operator are `float` values, no widening is used, i.e., Java carries out the operator with `float` precision. In contrast, Fortran instantly widens all operands if there is hardware support for `double` or `long double` precision.

• **Rounding.** The result of a binary floating point operation is a value, that in general cannot be represented in the IEEE floating point format exactly. Instead, the result must be rounded.

Although IEEE numerics allow the programmer to select from different rounding modes (zero, up, down, nearest), Java's numerics round to the nearest representable floating point number.

Several suggestions have been made to extend Java's rounding. The disadvantage of a global method to select a rounding mode, is that it might interfere with rounding modes used in fine tuned numeric libraries. The problem with newly introduced syntax for rounding is that it would cause another alteration of the language definition.

Another probable oversight in the definition of Java is that arithmetic exceptions are not mapped to Java exceptions, and all IEEE types of NaN values are collapsed into a single NaN recognized by Java.

Sun has recently announced [15] a proposal to change Java's floating point specification. Unfortunately, no details on that proposal are available at the time of writing.

6 Other Aspects

• **Graphical Interface.** Java provides a closely integrated GUI library that is both portable and easy to use. This is a significant improvement over Fortran, that does not support programming of GUIs.

In the C/C++ world, the situation is different but not better. Every window system comes with a platform specific library, that must be used to access the GUI. These libraries, for example X toolkit, Microsoft Foundation Classes, Borland OWL, ..., are non-portable and require skills to use them, skills that neither are nor should be standard for computational scientists.

Due to the close integration of a portable GUI library, appealing GUIs or even remote applet access to scientific applications will be implemented. It will become easier to share results with colleagues.

• **Programmer Force.** An increasing number of entry-level students is exposed to Java as their first language. Java is beginning to replace other languages in schools/colleges. And even Teenagers dive into Java because they want to create their own applets to spice up their home pages.

In a few years, students probably already know Java when they start to learn Fortran. Fortran will appear primitive in comparison. The situation will be similar for C/C++, where students will struggle with the complexity of C/C++ while trying to find the advantages over Java.

Sun has recently announced to cooperate with academia all over the world in launching courses that help practitioners from industry to gain Java knowledge. The courses will lead to a certificate called "Java Architect".

In a few years, it will be easier to recruit people to work in group J. Good Fortran programmers might become rare.

• **Availability.** Java is available on various platforms. Both SGI and IBM have implemented Java for their parallel machines. Other vendors will follow.

Java is inexpensive since the basic technology can be downloaded for free. Similar to the C/C++ world, where there are both commercial compilers and public domain tools, high performance compilers that do aggressive optimization, will no longer be for free. However, since the market for Java technology is much bigger than the market

for Fortran compilers, Java technology will be cheaper than Fortran.

7 Related Work

After this paper has been accepted for Euro-PDS, the Java Grande Forum [5] was founded that tackles some of the problems mentioned in this paper. The goal of Java Grande is to develop community consensus and recommendations for either changes to Java or establishment of standards for libraries and services. The hope of the Grande team effort is that a unified voice will result in the best-ever Java-enabled programming environment to support high-performance parallel and distributed computing and computational science applications.

James Gosling has thought about numerical computing in Java. Preliminary results can be found at [6]. Sun has recently announced [15] a proposal to change Java's floating point specification. Unfortunately, no details on that proposal are available at the time of writing.

8 Conclusion

Java will become a major language for computational science. This paper has tried to provide an objective evaluation of the pros and cons. Of course it is up to the reader to decide whether it is time to jump on the Java bandwagon right now or to remain watching for a while.

The paper mentioned several open issues. It suggested a competitive benchmark of memory consumption, and a controlled experiment to compare development time versus run time. Moreover, a Java compiler should experimentally be extended to handle complex numbers and corresponding infix operations. Furthermore, the implications of an additional interface Infix have to be studied in more detail.

Acknowledgments

I would like to thank Matthias Jacob for his implementation of the Sieve and the geophysical algorithms. The ICSI in Berkeley provided an opportunity to work in the Sather group and to gain an understanding of the demands of computational sciences.

References

[1] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[2] Z. Budimlic and K. Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, 1997.

[3] J. Clearbout and B. Biondi. Geophysics in object-oriented numerics (GOON): Informal conference. In *Stanford Exploration Project Report No. 93*. October 1996. http://sepwww.stanford.edu/sep.

[4] J.D. Darcy and W. Kahan. Borneo language. http://www.cs.berkeley.edu/~darcy/Borneo.

[5] Geoffrey Fox. The Java Grande forum. http://www.npac.syr.edu/projects/javaforcse/javagrande.

[6] James Gosling. The evolution of numerical computing in Java. http://java.sun.com/people/jag/FP.html.

[7] HPJ. http://www.alphaWorks.ibm.com.

[8] M. Jacob. Implementing Large-Scale Geophysical Algorithms with Java: A Feasibility Study. Master's thesis, Univ. of Karlsruhe, Dept. of Informatics, 1997.

[9] JavaParty. http://wwwipd.ira.uka.de/JavaParty.

[10] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. on Principles of Programming Languages* , pages 146–159, Paris, Fance, 1997.

[11] S.M. Omohundro. The Sather programming language. *Dr. Dobb's Journal*, 18(11):42–48, 1993.

[12] S.M. Omohundro and D. Stoutamire. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, 1996.

[13] M. Philippsen and M. Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[14] Pizza. http://www.cis.unisa.edu.au/~pizza.

[15] Sun proposes modification to Java's floating point spec. http://www.sun.com/smi/Press/sunflash/9803/sunflash.980324.17.html, March 1998.

[16] T.L. Veldhuizen and M. Ed Jernigan. Will C++ be faster than Fortran? In *Proc. of ISCOPE97, Intl. Conf. on Scientific Computing in Object-oriented Parallel Environments*, 1997.

[17] VisualNumerics. http://www.vni.com.