The MICROBE Benchmarking Toolkit for Java: a Component-Based Approach

Dawid Kurzyniec and Vaidy Sunderam

Department of Math and Computer Science, Emory University,
1784 North Decatur Road, Suite 100, Atlanta, GA 30322, USA
{dawidk, vss}@mathcs.emory.edu

Abstract. Java technology has recently been receiving increasing attention as a platform for high performance and large scale scientific computing. The MICROBE benchmarking toolkit is being developed to assist in the accurate evaluation of Java platforms. MICROBE is based on the tenet that benchmarking suites, in addition to furnishing benchmark codes, should provide flexibility in customizing algorithms, instruments, and data interpretation to facilitate more thorough evaluation of virtual environments' performance. The MICROBE architecture, projected usage scenarios, and preliminary experiences are presented in this paper.

1 Introduction

The number of areas in which Java technology [9] is adopted has been increasing continuously over last years. In particular, there is strong interest in using Java for so-called *Grande* applications, i.e. large scale scientific codes with substantial memory, network and computational performance requirements [2,20]. The Java Grande Forum [14] has been established to promote and augment Java technology for use in this area. Concerns about performance have traditionally accompanied Java from the very beginning of its evolution and continue to do so, especially in the context of high performance, distributed, and Grid computing. The performance of a Java application depends not only on the program code and the static compiler, as in the case of traditional languages, but is also highly influenced by the dynamic behavior of the Java Virtual Machine (VM). Although the advanced dynamic optimization techniques used in modern virtual machines often lead to application performance levels comparable to those using traditional languages, efficiency in Java is highly dependent on careful coding and following certain implementation strategies. In general, despite the ease and rapid deployment benefits that accrue, obtaining high performance in Java requires significant effort above and beyond the coding process. In this context, the existence and availability of reliable and thorough benchmarking codes, in addition to profiling tools, is of considerable importance to Java community.

Benchmarking (and especially microbenchmarking) of Java codes is much more complicated than benchmarking other languages due to the dynamic nature of Java Virtual Machines. Modern VMs are equipped with so-called Just-In-Time

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 709–719, 2002.

(JIT) compilers, which translate Java bytecode into optimized native code at run time. Because the optimization process can be time and memory consuming, state-of-the-art Virtual Machines [12,13] often defer compilation (resulting in so-called JIT warm-up time), and adjust the level of optimizations by observing program execution statistics that are collected in real-time. The objective of such VMs is to heuristically self-regulate compilation so that the "hot spots" of the program are strongly optimized while less frequently called methods are compiled without optimization or even not compiled at all. The common optimization techniques adopted include, among others, aggressive code inlining based on the analysis of data flow and the dependency of classes loaded into the VM.

Many benchmark suites try to cope with JIT warm-up issues simply by providing performance averages for increasing time intervals, but in general, the dynamic issues mentioned above can relate the VM performance not only to the time elapsed since the program was started and the number of times a given method was invoked but also to many other factors, like the dependency graph of classes loaded into the Virtual Machine or the amount of available memory. Dynamic garbage collection can also influence benchmark results if it is triggered at unexpected moments; further, background threads collecting run-time statistics can affect program execution in unpredictable ways. Therefore, results from a specific benchmark do not guarantee that equivalent results would be reported by another, similar benchmark – in fact, we observed differences up to an order of magnitude. Without considering dynamic issues and knowing exactly what the benchmark code does, such benchmarking results are of limited value. For benchmarks that do not publish source code, this problem is further exacerbated. Furthermore, different benchmarking approaches are required for server applications, where steady-state performance is in the focus, and client applications, where the VM startup overhead, the dynamic optimization overhead and memory footprint must also be taken into consideration. Also, in some applications (e.g. scientific codes) it is very important if the VM can compile and optimize a method that is executed only once (as the method can include a number of nested loops and have significant impact on overall performance) while in other applications that may be of no importance at all.

All the issues discussed above lead to the conclusion that it is virtually impossible to develop a single, generic and complete benchmarking suite for Java that is appropriate for all kinds of applications even if the area of interest was restricted only to large scale scientific codes. On the other hand, there is a definite interest in understanding Java performance in various, application dependent contexts. Rather than providing yet another standardized benchmarking suite, we propose to the Java community an open, extensible, component-based toolkit based upon separation of benchmarking algorithms, the instruments used for measurement, data transformation routines, and data presentation routines. As these components can be freely assembled and mixed, users can rapidly develop customized benchmarking codes they require. By demonstrating MICROBE version of Java Grande Forum benchmark suite, we show that the toolkit can also be a basis for development of complete, standardized benchmark suites.

2 Related work

A number of Java benchmarks have been developed in recent years. Some of these are microbenchmarks focusing on basic operations like arithmetic, method invocations, or object creation [10,1,11]. Others are computational kernels in both scientific [8,21] and commercial [5,22,19,4,7] domains. Typically, unlike microbenchmarks, these suites are proprietary and do not provide source code. There are also a few benchmarks approximating full-scale applications, and comparative benchmarks that implement the same application in Java and a traditional imperative language.

Perhaps the most comprehensive and important benchmarking framework for scientific applications is the Java Grande Forum Benchmark Suite [3,15], currently at version 2.0. This suite consists of three separate sections. Section 1 contains a set of microbenchmarks testing the overhead of basic arithmetic operations, variable assignment, primitive type casting, object creation, exception handling, loop execution, mathematical operations, method calls, and serialization. Section 2 includes numerical kernels, such as the computation of Fourier coefficients, LU factorization, heap sort, successive over-relaxation, encryption and decryption with the IDEA algorithm, FFT, and sparse matrix multiplication. Finally, Section 3 contains several full applications, including a solver for time-dependent Euler equations, a financial simulation using Monte Carlo methods, and a simple molecule dynamics modeler.

The interesting feature of JGF benchmarks is that they separate instrumentation and data collection from data presentation in a way that enables relatively easy addition of new benchmarks to the suite. Unfortunately, the microbenchmarking algorithms themselves are hard-coded thus very difficult to modify.

A few shortcomings of JGF benchmark suite were pointed out [17,7]. The suite was augmented with new benchmarks [6] including corrected arithmetic and method call microbenchmarks, extended microbenchmarks for object creation and exception handling, new microbenchmarks for thread handling, new kernels including sieve of Eratosthenes, tower of Hanoi, Fibonacci sequence and selected NAS [18] parallel benchmarks.

A common issue of Java benchmarks is that for sake of portability, they usually employ the `System.currentTimeMillis()` method as it is the only time measurement facility currently available in Java. The inaccuracy of this measure forces benchmarks to perform a large numbers of iterations, making them vulnerable to strong run time optimizations [17] and excluding more fine-grained tests, like analysis of JIT compilation process step by step or evaluation of interpreted code.

3 MICROBE Toolkit Architecture

In any benchmarking suite, it is possible to discriminate among several conceptual entities:

- **Operations** that are to be tested, like arithmetic operations, object creation, array sort, or FFT.
- **Algorithms** used to perform the benchmark: for microbenchmarking, there are usually different kind of loops, codes for calibration, etc.
- **Instruments** for quantity measurement: in Java, time is usually measured with `System.currentTimeMillis()` method, whereas memory footprint is measured with `Runtime.totalMemory()` method.
- **Data processors** transforming quantities collected by instruments into some interpretable output, like the number of operations performed per time unit, arithmetic mean and standard error computed for a series of measurements, etc.
- **Data reporters** used to present benchmark results to the user. They can show data in the console window, display graphs, generate tables or HTML output files.

In all the benchmarking suites that we reviewed, these entities are more or less tightly bounded together. Although in a few cases (notably, the JGF Benchmark Suite) it is relatively easy to use custom time measure or data reporting routine, it is still not possible to benchmark the same operation using different or modified algorithms, or using different data processing routines to reinterpret the collected data. The MICROBE toolkit is based on careful separation of all five coefficients leading to independence and orthogonality between them.

3.1 Benchlets and Yokes

In order to separate operations from algorithms, the MICROBE toolkit introduces the notion of *benchlets*. A benchlet is a small piece of code encapsulating only the operation to be tested and implementing the following simple Java interface:

```
interface Benchlet {
    void init(Object[] params);
    void execute();
}
```

The `execute()` method provides the operation to be tested, e.g. binary addition or object creation. Each benchlet has a no-argument constructor; the initialization may be performed in the `init()` method. Additionally, the benchlet can be attributed as *unrolled* or *calibratable* (by implementing appropriate interfaces). The *unrolled* benchlet performs more than one operation inside the `execute()` method. The *calibratable* benchlet designates some other benchlet to rule out benchmarking overheads related to the tested operation, like the empty loop cycles or necessary preliminary steps which would otherwise influence the results.

The algorithms in the MICROBE toolkit are represented by entities called *yokes*. A yoke is able to instantiate a benchlet and control its execution in a certain way, possibly performing some measurements. Instead of encapsulating

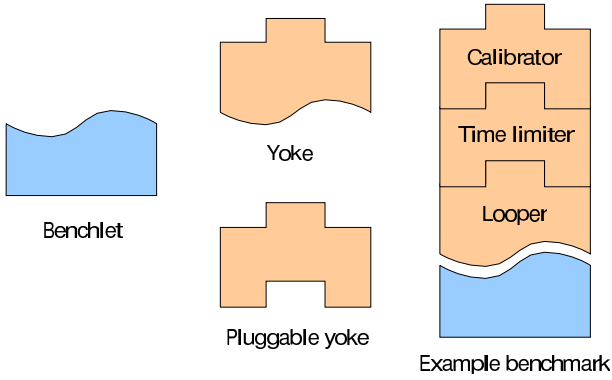

Fig. 1. Benchlets, yokes, and benchmarks

complete, sophisticated algorithms though, yokes focus only on elementary tasks, so the user needs to combine yokes to build more complex ones.

The simplest yoke provided in the toolkit is the *looper*, which repeats the `execute()` method of an instantiated benchlet within a loop for a specified number of iterations. Most other yokes are not self-sufficient but are rather designed to alter the behavior of other yokes, as denoted by the *pluggable* attribute. A few examples of pluggable yokes include: *calibrator* which adjusts results of measurements for some benchlet with that of its calibrating benchlet, *reseter* which provides benchlets with separate environments by loading them through independent class loaders and only after performing garbage collection, and *time limiter* which restricts the amount of time for which the associated yoke is allowed to run, if only that yoke has the *interruptible* attribute and responds to interrupt request. All standard yokes shipped with the MICROBE toolkit (including the looper yoke) are interruptible. Although this feature requires yokes to continuously monitor the value of a boolean flag, potentially introducing small overheads, it has also the positive side effect of improving the accuracy of calibration, as the calibrator loops (that often tend to be empty) become less vulnerable to dynamic optimization. The relationship between benchlets and yokes is illustrated in Fig. 1.

The modularity of the design enables easy customization of the algorithms formed by collaborating yokes: for instance, it is straightforward to enable or disable calibration, add time constraints, or compare the benchmark behavior as it is loaded through common or separate class loader – features difficult if not impossible to achieve using existing benchmarking suites.

To improve the accuracy and reliability of the results, it is often desirable to repeat a given benchmark several times and collect statistics. To address this need, the MICROBE toolkit provides the *repeater* yoke. This yoke invokes another selected yoke in a loop until the exit condition (specified by the user) is satisfied. During the loop, the repeater yoke can collect arbitrary data series

and compute statistics, including: weighted and unweighted arithmetic and geometric mean, median, standard deviation, standard error, relative error, and other standard statistical parameters. Further, a user-defined weight function may be applied at any time to the collected data. Two useful weight functions are predefined: one for sharp cutoff and second one for smooth exponential cutoff, which both permit recent results to bias the metric. Because the results of the statistics can be used in the evaluation of exit conditions, users can develop very specific and sophisticated algorithms that control the number of iterations of a benchmark. For instance, it is relatively easy to create the yoke that repeats a given operation until the result stabilizes at the certain level, e.g. at the third significant digit.

3.2 Data Producers, Consumers, and Filters

Data exchange in the MICROBE toolkit is based upon a variation of event listeners pattern. Yokes do not process or display collected data by themselves, but they may rather have *data outputs* (or *data producers*) to which other entities can attach. Many yokes contain also *data inputs*, or *data consumers*, which can be connected to appropriate data producers. For instance, the calibrator yoke provides inputs accepting the data to be calibrated, and appropriate outputs producing calibrated results. This approach leads to the extreme flexibility – e.g., the calibrator yoke does not have to assume that the measured quantity represents a particular parameter such as computation time or an amount of memory.

The MICROBE toolkit directly supports four kinds of data exchanged between producers and consumers: single *numbers*, *number arrays*, arbitrary *objects*, and *signals*, which do not carry additional information. On their way from one yoke to another, events and data can be processed through *filters* transforming them in numerous ways. The toolkit provides many basic transformation routines, like arithmetic operations on single numbers, *collectors* allowing gathering of several numeric events to be triggered together, array operations computing sum or product of the elements, and many more.

Of particular note are certain kinds of filters which convert signals into numerical values, because various instruments measuring different quantities fit into this category. For example, a *clock* in MICROBE toolkit is nothing but converter of query signal (analogous to pushing the stopper button) into the value expressed in seconds:

```
class Clock implements Transform.Signal2Number {
    public double transform() {
        return System.currentTimeMillis()/1000.0;
    }
}
```

Such a generalization allows the toolkit to be independent of the particular instruments, so users can replace them very easily according to the needs. Also,

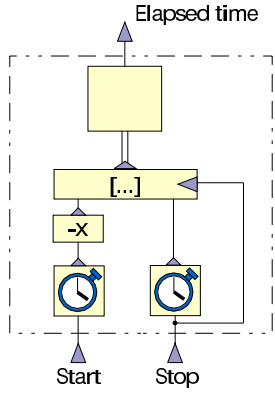

Fig. 2. Difference measurer used to compute time elapsed between two signals

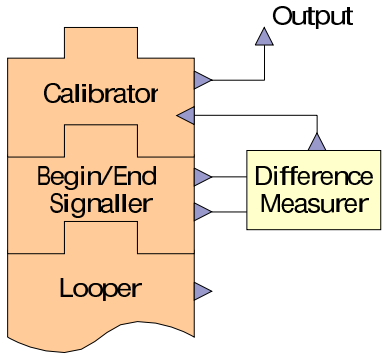

Fig. 3. Example benchmark

because the measurement is reduced just to invocation of simple Java method, new and customized instruments can be developed with little effort.

Apart from the (not very accurate) clock shown above, the MICROBE toolkit provides an additional notion of CPU time measurement based on native code, specifically the `clock()` function from the Standard C Library. Despite the use of native code, portability at the source code level is retained due to the use of Java Native Interface [16] and wide availability of the C library. Measures which are even more accurate but also more platform dependent (like those based upon internal CPU cycle counting) may be developed by users; those can be especially feasible for testing a short term behavior of the Java VM.

Just like yokes, filters can be grouped together to form more sophisticated ones. The Fig. 2 presents a *difference measurer* that computes the time elapsed between two signals using the *clock* described above as an internal instrument (as mentioned, the difference measurer is oblivious to the instrument it uses).

When the "start" signal appears at the input, the time is read and stored by the collector (represented in the Figure by the wide rectangle) after negation. When the "stop" signal is issued, the time is read again and also stored. Because the "stop" signal is connected to the trigger of the collector, these two aggregated values are triggered and their sum appears at the output.

Fig. 3 shows an application of the difference measurer in a example of a benchmark algorithm which calculates calibrated execution time for a benchlet invoked for some number of iterations. The looper, being the innermost yoke used, is wrapped into a *begin/end signaler* which signals the beginning and the end of the computation. These signals are attached to a difference measurer which computes elapsed time and passes it to the calibrator. The calibrated output constitutes the result.

4 Reporting Results

One of the important tasks of a benchmarking suite is to interpret collected data in certain way and report it to the user, e.g., displaying it in the console window, writing it to a text file or drawing a performance graph. Examples of data interpretation include computing a mean over a series of results, transforming time figures into a temporal performance (the number of operations per time unit), etc. The modular structure of the MICROBE toolkit allows virtually any data interpretation to be applied to the benchmark without changing its code. Fig. 4 shows how temporal performance can be computed for the example benchmark algorithm evaluated in previous Section.

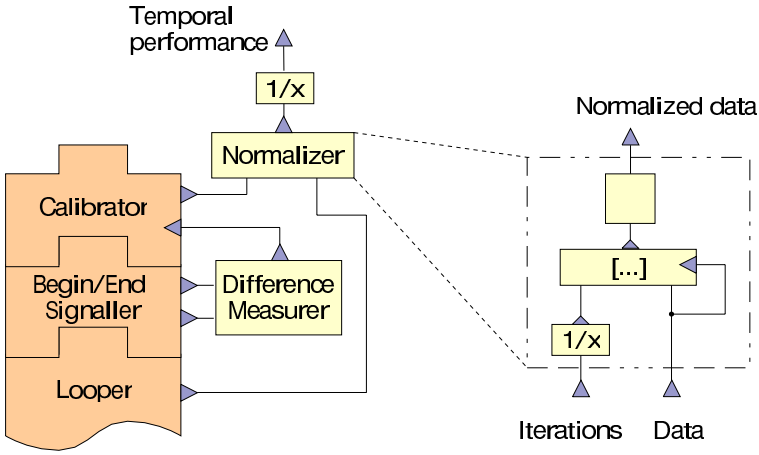

Fig. 4. Data interpreters applied to a benchmark algorithm

The flexible event listener mechanism can also support arbitrary data reporting method as users can decide exactly how (and which) data should be reported by developing proper data consumers and plugging them to appropriate producers. However, although such approach enables unrestricted customization of benchmark behavior, it also requires significant work. In contrast, there is often a need to quickly examine some benchmark code without developing reporting routines intended to be used only once – so, it is desirable that the yokes provide some default notion of data reporting out-of-the-box.

To satisfy this requirement, we added the optional *reporting* attribute to the yoke definition. The *reporting* yoke is the one capable of creating a hierarchically organized report from data collected during benchmark execution. That hierarchical data structure originates from analogous layout of yokes logically nested inside each other. The data report can be then passed to a *Reporter* object in order to exhibit it in some way. At this time we provide only one reporter class which simply displays the collected data in the console window. We are currently working on the improvements to this mechanism, specifically two other reporters: one generating HTML and another generating XML output files.

5 MICROBE and the Java Grande Forum Benchmark Suite

The Java Grande Forum Benchmark Suite is one of the most important benchmarking packages for Java related to large scale scientific computing. To show the appropriateness of MICROBE to become a basis of such benchmark suites, we translated JGF benchmarks into their MICROBE counterparts.

Section 1 of the JGF suite consists of a number of microbenchmarks. We have transformed all of these into benchlets that can be used within the MICROBE toolkit. The algorithm used by the JGF suite for microbenchmarking performs a sequence of successive loops with the number of iterations growing by a factor of two until either the limit of iterations or elapsed time has been reached. We provide an appropriate yoke (called JGFLooper) which implements the same algorithm, but is even more flexible as it fits the generic MICROBE model and can be combined with different yokes to facilitate more sophisticated testing patterns.

Section 2 of the JGF suite consists of various computational kernels. We have developed an appropriate adapter benchlet, which allows MICROBE yokes to execute and measure these kernels. As the adapter benchlet generates signals when approaching subsequent stages of the kernel execution, it is possible to develop some customized yokes especially suited to deal with JGF kernels, although the default ones are sufficient to perform measurements analogous to those of JGF suite.

In Fig. 5, we compare the results obtained by benchmarking the JGF suite (Section 1) and their MICROBE analogs for two different virtual machines. The results reported are geometric mean averages for each microbenchmark. The test platform was a Dell Dimension PC with a 450 MHz Pentium II processor and

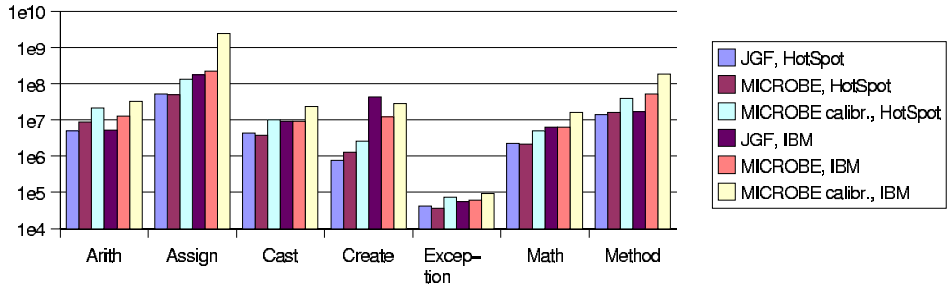

Fig. 5. Performance results [operations per second]

128 MB of memory running Mandrake Linux 7.2. The VMs were SUN HotSpot Client 1.3.0 and IBM 1.3.0 for Linux. As expected, the results for JGF and MICROBE are very similar in most of cases, except for the "Create" and "Method" microbenchmarks. That last anomaly is most likely a result of the simplicity of the methods invoked and the objects being created in JGF suite. These aspects, in conjunction with the benchmarking algorithms used, makes the benchmarks very sensitive to run-time optimizations and can introduce performance differences even for very similar codes. It is also worth noting that the calibration – absent in JGF suite but enabled by MICROBE – identified a weakness of the JGF "Assign" test, in which the IBM VM were able to optimize out some variable assignments, resulting in unrealistically high performance figure.

6 Conclusions and Future Work

In this paper, we have described the MICROBE toolkit which facilitates the rapid construction of Java benchmark codes. The toolkit addresses the need observed among the Java community to facilitate testing various aspects of dynamic Java Virtual Machine behavior, especially the issues of dynamic code optimization. We have evaluated the MICROBE version of the Java Grande Forum benchmarks, showing that MICROBE can be used to develop complete benchmark suites.

Currently, we are investigating the possibility of extending the concept of a benchlet to facilitate more sophisticated benchmarks, e.g. emulating the behavior of full blown applications as well as parallel codes. We also recognize the possibility of developing a graphical language which would permit the visual construction of benchlets and yokes in the manner outlined in this paper.

Scientific applications for Java are often integrated with components and libraries written in traditional languages. Such integration can be performed using the Java Native Interface (JNI) [16], which, however, may introduce significant overhead. To address this issue, we intend to establish a full size benchmarking suite for the JNI based on the MICROBE toolkit.

References

1. D. Bell. Make Java fast: Optimize! *JavaWorld*, 2(4), Apr. 1997. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html.
2. R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. In *ACM-1998 Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, February 1998. http://www.cs.ucsb.edu/conferences/java98/papers/jnt.ps.
3. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency, Practice and Experience*, 12:375–388, 2000. Available at http://www.epcc.ed.ac.uk/~markb/docs/javabenchcpe.ps.gz.
4. BYTE Magazine. BYTE benchmarks. http://www.byte.com/bmark/bmark.htm.
5. P. S. Corporation. CaffeineMark 3.0. http://www.pendragon-software.com/pendragon/cm3/.
6. Distributed and High-Performance Computing Group, University of Adelaide. Java Grande benchmarks. http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/.
7. O. P. Doederlein. The Java performance report. http://www.javalobby.org/fr/html/frm/javalobby/features/jpr/.
8. J. Dongarra, R. Wade, and P. McMahan. Linpack benchmark – Java version. http://www.netlib.org/benchmark/linpackjava/.
9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000. http://java.sun.com/docs/books/jls/index.html.
10. W. Griswold and P. Philips. Excellent UCSD benchmarks for Java. Available at http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.
11. J. Hardwick. Java microbenchmarks. http://www.cs.cmu.edu/~jch/java/benchmarks.html.
12. Java HotSpot technology. http://java.sun.com/products/hotspot.
13. Jalapeño project home page. http://www.research.ibm.com/jalapeno.
14. Java Grande Forum. http://www.javagrande.org.
15. Java Grande Forum. Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/.
16. Java Native Interface. http://java.sun.com/j2se/1.3/docs/guide/jni/index.html.
17. J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *ACM 1999 Java Grande Conference*, San Francisco, California, June 12-14 1999. Available at http://www.cs.ucsb.edu/conferences/java99/papers/41-mathew.ps.
18. NASA Numerical Aerospace Simulation. NAS parallel benchmarks. http://www.nas.nasa.gov/Software/NPB/.
19. PC Magazine. JMark 1.01. http://www8.zdnet.com/pcmag/pclabs/bench/benchjm.htm.
20. M. Philippsen. Is Java ready for computational science? In *Proceedings of the 2nd European Parallel and Distributed Systems Conference for Scientific Computing*, Vienna, July 1998. http://math.nist.gov/javanumerics/.
21. R. Pozo. SciMark benchmark for scientific computing. http://math.nist.gov/scimark2/.
22. Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. http://www.spec.org/osg/jvm98/.