# Analysis and Development of Java Grande Benchmarks

J.A. Mathew, P.D. Coddington and K.A. Hawick

Advanced Computational Systems Cooperative Research Centre
Department of Computer Science, University of Adelaide
Adelaide, SA 5005, Australia
{jm,paulc,khawick}@cs.adelaide.edu.au

**Abstract**

Understanding the performance behaviour of Java Virtual Machines is important to Java systems developers and to applications developers. We review a range of Java benchmark programs, including those collected by the Java Grande Forum as useful performance indicators for large-scale scientific applications, or "Java Grande applications". We systematically analyse these benchmarks on a collection of compute platforms including Pentium II, UltraSPARC II, Silicon Graphics, iMac and Alpha, and operating systems including Windows NT, Solaris, and Linux, and consider differences in performance between benchmarks running on the Java Development Kit (JDK) versions 1.1.6 and 1.2. We also analyse some benchmark programs we have developed to augment existing collections. We observe some general trends in the performance of Java on various platforms and also conduct some comparisons between benchmarks written in Java and in traditional languages such as C and Fortran. We discuss performance variations across systems and the implications of Java systems performance for those developing scientific applications.

**Keywords:** Java, Java Grande, benchmarks

## 1 Introduction

The Java programming language is being widely adopted for many programming applications areas beyond the original intentions of its designers [8]. Applications developers are entering the Java coding arena, eager to reap the benefits of a widely used, portable language which has many available library packages and a plethora of viable platforms that are able to run Java applications. While this has created a wide availability of Java code for various purposes, it has also highlighted some of the performance inadequacies and performance anomalies of Java implementations. Applications programmers and systems developers accustomed to the performance behaviour and properties of conventional programming languages such as C and Fortran are still learning what to expect from the current generation of Java implementations and are learning a new set of coding styles and idioms geared towards achieving performance using Java.

In an ideal world compiler writers and system providers hope to be able to provide a programming environment where the application developer need not be aware of performance considerations, but inevitably in practice, this is a concern. Achieving performance with Java programs is still an elusive but much sought-after goal, particularly for computationally-intensive scientific applications.

The high-performance computing (HPC) research community is addressing the issue with a range of approaches, involving the parallel programming technology developed over the last two decades as well as compiler and system optimisation techniques applicable to the high-end superscalar processors and shared memory multi-processor platforms now commonly available. An obstacle to both application developers and Java system developers has been the lack of quantitative evaluation of the performance of Java systems given the inexperience with using the new language in HPC applications.

The Java Grande Forum [13] is addressing many of the issues that arise in using Java for the kinds of applications that have conventionally required HPC platforms. These include Java performance improvements, language features and design, and additional class libraries, that could all be of benefit to such "Java Grande applications". In particular, the questions of how to incorporate parallel message passing technology [12, 3] and data parallelism [11] into a Java environment are being addressed. Metacomputing software glue [10] that allows clusters of nodes that run Java Virtual Machines to participate in a distributed computation will also contribute to achieving reduced wall clock times for certain applications.

Work in these areas is important and will likely lead to significantly improved performance for those applications that can be parallelised or distributed across a number of cooperating compute nodes. However, these efforts are still in the early stages of development, and we believe there are still a number of performance issues to be addressed at the serial code level, that are vital to the single node performance that parallel or distributed programs are dependent upon. In this paper we therefore focus on a systematic investigation of the performance of single processor performance of Java applications. Our objective is to discuss the performance issues for the various idiomatic forms of expressing common application level operations in Java, both for application developers moving to Java from other languages and for Java systems developers to realise what features would be worth optimising.


JAVA'99 San Francisco California USA

To this end, we have developed some new benchmarks to augment the Java Grande Forum Benchmark Suite [6] with some additional low-level benchmarks and several scientific application kernels, including Java implementations of some of the NAS benchmarks [15] that are commonly used for benchmarking high-performance computers for scientific applications. We have also undertaken a systematic analysis of these new benchmarks, as well as the existing Java Grande benchmarks, on serial platforms including Pentium II (Linux, Solaris and NT); Celeron (NT); UltraSPARC (Solaris); Silicon Graphics (Irix); Alpha (Digital Unix); and iMac PowerPC (Linux).

In section 2 we review some of the Java benchmarks that are currently available and outline some of the benchmarking issues we believe to be relevant for Java Grande applications. In section 3 we focus on the Java Grande Forum Benchmark Suite and describe some additional benchmark codes we have developed to augment that collection. Results from the current Java Grande Forum benchmarks and our new benchmarks on selected platforms are presented in section 4, along with an analysis of these results and their implications for the performance of Java for scientific computations. We conclude with a discussion of future work on the analysis and development of Java Grande benchmarks in section 5.

## 2 Java Benchmarks

There are many benchmarks available for Java [7], however surprisingly few of them are useful for the kind of performance measurements and analysis that are relevant for the Java Grande Forum mission of making Java a better language for high-end scientific computing.

Most Java benchmarks are designed purely to compare the general performance of different Java Virtual Machines (JVMs) on different operating systems (or Web browsers) and platforms. Many of them are available in applet form, so they can be downloaded and run in a Web browser.

Almost all of the existing Java benchmarks provide only byte code, not Java source code, and very few provide implementations of the benchmark programs in standard languages such as C, Fortran, or C++. This makes it impossible to do a serious analysis of the performance of the code; a comparison with the performance of the same program written in standard languages used for scientific computing; a careful analysis of effects such as caching, memory management and garbage collection; and an analysis of how the Java language or compilers could be improved to increase the performance of the benchmark program. These are effectively "black-box" benchmarks, where the only real interest is in the number produced for comparison between JVMs.

Most of the available Java benchmarks deal with low-level, general-purpose routines such as basic arithmetic operations, object creation and memory allocation, method invocation, garbage collection, loops, array accesses, casting, exception handling, etc. The majority of these benchmarks are useful since they are operations that affect the performance of any Java program, however some focus on specific areas such as graphics toolkit and applet performance that are not so relevant to Java Grande applications. Most of the commonly-used benchmarks, such as JMark [16], CaffeineMark [17], and SPEC JVM98 [18], do not provide source code. Some low-level benchmarks for which source code is available include JavaWorld microbenchmarks [1], UCSD benchmarks [9], and some Java/C++ benchmarks [19].

There are some higher-level benchmarks available that implement applications or applications kernels, but most of these are targeted at general computing rather than scientific applications. For example, SPEC JVM98 applications include text compression, MPEG decoding, compilation speed, graphics, and database functions. Relatively few higher-level benchmarks are directly related to scientific computing. SciMark [14] is an interesting Java benchmark applet consisting of five common scientific application kernels: 1-D Fast Fourier Transform (FFT); differential equation solver using Jacobi successive over-relaxation (SOR); Monte Carlo integration; sparse matrix multiply; and dense matrix LU factorization. Unfortunately the source code is not available, and there are no equivalent C or Fortran versions of these routines for performance comparisons.

Java implementations of the BYTEmark benchmarks [2] include routines for sorting, prime number sieving, computing Fourier coefficients, and matrix factorization [19]. There is also a Java implementation of the classic Linpack linear algebra benchmark [5]. Source code, and comparable programs in C, C++ or Fortran, are available for all these benchmarks, so these are the most useful of the currently available Java benchmarks for analysing the performance of Java for scientific applications.

## 3 Java Grande Forum Benchmarks

The Java Grande Forum is collecting a set of benchmarks that are relevant for testing the performance of Java for scientific computations. The Java Grande Forum Benchmark Suite [6] is broken up into three different areas: low-level benchmarks that test general language features and operations; scientific and numerical application kernels; and full-scale science and engineering applications. The programs contained in version 1.0 of the benchmark suite are listed below.

The low-level benchmarks were developed specifically for this benchmark suite, and consist of the following operations:

- **Arith** executes basic arithmetic operations.
- **Assign** does variable assignment.
- **Cast** does casting between different primitive types.
- **Garbage** tests the performance of the garbage collector.
- **Math** executes all the mathematical functions in the `java.lang.Math` class.
- **Method** determines the cost of a method call.

There are some potential shortcomings with some of these benchmarks. The main problem is that with such low level operations, it is often difficult to avoid having operations that can be optimised out by an intelligent compiler. We believe this may be occuring with some of the more recent Java compilers for some low-level benchmarks such as the Assign benchmark and the Method benchmark, which makes calls empty methods which could easily be optimised out. We have developed an alternative benchmark for method calls that avoids this problem.

We were unable to get the Garbage benchmark to work under JDK 1.2 on most platforms. It is very difficult to adequately measure the performance of the garbage collector with a synthetic benchmark, and more work is probably required in this area. It is also difficult to isolate the overheads of garbage collection for a scientific application or an application kernel benchmark.

All of the low-level benchmarks run the tests within a `for` loop, with each loop performing 16 operations. It would be worthwhile to run a test that checks whether this amount of loop unrolling is large enough that the loop overhead is not significant for a given JVM. We have developed a test of this kind and found that 16 is adequate for all the systems we have tested.

The application kernels are obtained mostly from the BYTEmark and Linpack benchmarks, since these are virtually the only currently available Java benchmarks for scientific or mathematical computations that provide source code and comparable implementations in other languages. The kernels are:

- **Fourier** computes Fourier coefficients, which tests computation of floating point transcendental and trigonometric functions. There is little array activity, so this test should not be dependent on cache or memory architecture.
- **LUFact** is the Linpack LU factorization benchmark, which tests array access and basic floating point arithmetic speed.
- **Search** solves a game of connect-4 using an alpha-beta pruning search technique.
- **HeapSort** sorts an array of integers using a heap sort algorithm, and tests generic integer performance.
- **Crypt** performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes.

The large-scale application benchmarks are intended to be representative of Java Grande applications. Currently there is only one, a computational fluid dynamics application, however it is currently unavailable for download.

### 3.1 Some New Java Grande Benchmarks

The Web page for version 1.0 of the Java Grande Forum Benchmark Suite [6] lists a number of proposed benchmarks to be developed in the future. We have implemented several of these, both low-level and kernel benchmarks. We have also developed Java implementations of some other scientific and numerical kernels that should also be useful as Java Grande benchmarks, and work on others is in progress. The new benchmarks that have been completed and used in our studies are described below. They are available on the Web [4] in a package that is easy to run on any platform that supports a Java Virtual Machine (JVM).

The new low-level benchmarks are:

- **Object** measures the cost of creating various types of objects:
  1) the simplest Java object: `java.lang.Object`
  2) a simple user defined object with no contents and no constructor
  3) a simple user defined object with no contents but with a constructor
  4) a more complex object with primitive variables
  5) objects which instantiate other objects
  6) arrays of varying type and size
- **Exception** measures the speed at which exceptions can be thrown and caught. The benchmark includes three cases: throwing `java.lang.Exception` (the default exception); throwing a user defined exception (subclassing `java.lang.Exception`); and throwing a user defined exception that includes a message.
- **Thread** starts a number of threads, each of which increments a counter when run, and immediately yields. Hence the benchmark will provide a measure of the maximum speed at which threads can be switched. The benchmark tests the cases of having 2, 4, 8 and 16 threads.
- **Method** measures the overhead of a method call. In this alternative to the method testing benchmark of the previous section, we include the case of a method that increments a counter, and hence cannot be optimized out. We also measure the performance of synchronised methods, of non-synchronised methods with synchronised blocks, and of methods that return values.
- **Loop** measures the performance of `for` loops that iterate forwards and backwards, `for` loops with a synchronised block, and a `while` loop.
- **Generic** provides generic code segments whose performance may be of interest. There is a test that compares three approaches for incrementing a counter: incrementing a local variable inline; incrementing a class variable; and making a method call to increment the counter. Another test is based on the observation that in all of the current low-level Java Grande benchmarks, the tests are run within a `for` loop, with each loop performing 16 operations. In order to test if 16 is a reasonable value on all platforms (i.e. that the loop overhead is not significant), we have a test that performs integer addition, with either 1, 4, 16, 64 or 256 additions in the loop body.

The new kernel benchmarks are:

- **Sieve of Eratosthenes** is the standard algorithm for finding the prime numbers in a given interval, which has a lot of array indexing overheads.
- **Tower of Hanoi** solves a well-known sorting puzzle.
- **Fibonacci sequence** measures the cost of many recursive method calls, which are required to generate the sequence.
- **NAS EP** (embarrassingly parallel) benchmark generates pseudo-random numbers with a Gaussian probability distribution.
- **NAS IS** (integer sorting) benchmark does key sorting.
- **NAS FT** (Fast Fourier Transform) is a commonly-used scientific computation that requires floating point arithmetic and data shuffling.

The NAS Parallel Benchmarks [15] are a well-known suite of several application kernels that are indicative of some of the large-scale scientific and engineering computations done at NASA. They have mainly been used to benchmark the performance of parallel high-performance computers for these kinds of applications. It is quite an arduous task to directly port them to Java since they are written in Fortran and some of them have a substantial amount of code, so we have mostly taken C codes for the same tasks and converted them to fit the specifications of the NAS benchmarks as well as porting them to Java.

So far we have completed three of the benchmarks (EP, IS and FT), and we currently have partial implementations of the CG (Conjugate Gradient) and MG (Multigrid) codes. We also plan to implement parallel versions of these benchmarks, using the proposed Java bindings to MPI [12, 3].

## 4 Benchmark Results and Analysis

Here we present some benchmark results across a range of platforms and compilers for version 1.0 of the Java Grande Benchmark Suite and for our additional Java Grande benchmarks described in section 3.1. The complete set of results are available on our Web site [4], here we present a subset highlighting the more interesting results.

The benchmarks have been tested on a 333 MHz Pentium II processor running Windows NT; 266 MHz Pentium II processor running Solaris; 266 MHz Pentium II processor running Linux; 300 MHz Celeron processor running Windows NT; 300 MHz Sun UltraSPARC II processor running Solaris; 300 MHz Digital (DEC) Alpha 255 processor running Digital Unix; 180 MHz MIPS R5000 processor in a Silicon Graphics (SGI) O2 running Irix; and a 233 MHz PowerPC processor in an Apple iMac running Linux. We have used the release versions of Sun's JDK 1.2 for NT and Solaris, however JDK 1.2 is currently not available for some platforms including Irix and Linux, and is only in beta release for Digital Unix. We also benchmarked release versions of JDK 1.1 for all platforms, as well as alternative JDKs for Windows NT from IBM and Microsoft, and a beta release of the Kaffe JDK.

Selected results for low-level benchmarks from the Java Grande Benchmark Suite are shown in Table 1, with some results for our new low-level benchmarks presented in Table 2. Results for application kernels from the Java Grande Benchmark Suite are shown in Table 3, and results for our new application kernels are shown in Table 4.

Where equivalent native code (Fortran, C or C++) implementations of the Java benchmarks are available, we also provide results for these in order to compare the performance of Java and native code. For the native code we generally use the GNU `gcc` C compiler and `g77` Fortran compiler, since these are available for all the Unix platforms. For NT, we used Microsoft's Visual C++.

Timings were measured using the `currentTimeMillis` method in Java, which returns the current time in milliseconds. This is not guaranteed to have an accuracy of one millisecond, however the benchmarks all take at least a second so the results should be accurate enough for our purposes. We have repeated the benchmarks several times to check that they provide consistent results (generally within 1% between different runs), but we have not undertaken a detailed error analysis.

When comparing the results, you should be aware that the Java Grande Forum benchmarks have been defined in such a way that the results from the low-level benchmarks are in operations per second, so higher results are better, while the kernel benchmarks are in seconds to completion, so lower results are better.

Some implementation problems where faced in porting the arithmetic benchmark algorithms to C. We identified and corrected a few potential problems, but in some cases it appears that the benchmarks are too simplistic and the compilers are able to optimize out some of the operations, since in some cases the results for the C versions exceeded the peak performance of the processor. This is not apparent in the Java results, perhaps because the compilers are not as mature. One problem was that the floating point addition benchmark caused overflow. This caused the C problem to terminate and was fixed by resetting the values of the variables after 25 iterations of the loop, where each iteration of the loop includes 16 arithmetic operations. This problem is not apparent in Java as overflow creates a value of "Not a Number" (NaN) which is a valid operand for further calculations. It may be possible that the JVM can optimize subsequent operations based on the knowledge that if one of the operands is NaN then the resulting value is always NaN, so the Java benchmark should also be changed to avoid overflow.

Another problem we encountered was that the integer multiplication benchmark ended up with all values being zero after several operations. This was traced to the fact that Java does multiplication modulo $2^{32}$ (or modulo $2^{64}$ for `long` integers), and when any of the initial variable values is a power of two, the values would eventually converge to a large power of two, which would give zero after the modulus operation. Again, it is possible that a smart compiler could optimize multiplication by zero. The Java Grande arithmetic benchmark uses a simple route for floating point multiplication that essentially consists of alternatively multiplying a variable by pi and 1/pi, in order to avoid overflow. However there is potential for a compiler to optimize this out, so we modified the benchmark slightly to attempt to avoid this problem.

A number of general trends emerge from the benchmark results. The performance of the benchmarks is generally very good under Solaris for SPARC, while for the Pentium the results for the Sun JDK under NT tend to be slightly better than for Solaris. The performance under NT and Solaris is generally better than for Linux, Irix and Digital Unix. This trend seems to generalise across the benchmarks and machines we investigated. This is presumably just indicative of the amount of development effort that has gone into the JVMs for these platforms.

We ran benchmarks using the JDK for Linux on the PowerPC with a just-in-time (JIT) compiler from Metroworks, which provides quite impressive performance. Basic arithmetic operations (Table 1) were very competitive with other platforms, although other low-level tasks such as object creation and method calls (Table 2) were not as good, which probably contributed to the iMac performance on the kernels being weaker than might have been expected from the speed of the arithmetic operations.

In general, the JDK 1.2 provides performance improvements over JDK 1.1.6 in a number of areas. The improvements under Solaris, for both Pentium and SPARC, are in some cases quite substantial. Improvements under NT tend to be more modest, perhaps because in many cases it was already better optimized than Solaris. For basic arithmetic operations, the performance of JDK 1.2 versus 1.1.6 is similar in NT, better in Solaris for the Pentium, and roughly 10% worse for Solaris for the UltraSPARC.

In JDK 1.1 for Solaris, basic arithmetic operations took about the same time when using 32-bit `int` as for 64-bit `long` on 32-bit processors, but in JDK 1.2 the speed of using `int` is greatly increased. Under NT, there is little change since this problem did not occur under JDK 1.1. The Alpha is a 64-bit processor, and `int` and `long` have about the same performance, as expected. In all cases, using 32-bit `float` is only marginally faster than using 64-bit `double`. However some arithmetic operations and mathematical functions have become slightly slower on the various JVMs under JDK 1.2, particularly on UltraSPARC.

The performance of integer operations is similar across different JVMs on processors with similar clock speed, with only the Alpha and Linux JDKs being markedly slower. However the performance of basic floating point arithmetic, which underlies most Java Grande applications, varies greatly between similar processors, and can be much slower than the

| JVM | Add | | | | Mult | | | |
|---|---|---|---|---|---|---|---|---|
| | Int | Long | Float | Double | Int | Long | Float | Double |
| Celeron NT JDK 1.1.6 | 140.6 | 59.0 | 2.53 | 2.52 | 64.5 | 34.2 | 2.84 | 2.71 |
| Celeron NT JDK 1.2 | 140.7 | 53.7 | 2.52 | 2.62 | 64.7 | 33.5 | 2.84 | 2.78 |
| Pentium NT JDK 1.1.6 | 154.2 | 64.7 | 2.77 | 2.77 | 70.9 | 37.4 | 3.11 | 2.96 |
| Pentium NT JDK 1.2 | 154.3 | 59.0 | 2.77 | 2.87 | 70.9 | 36.8 | 3.12 | 3.04 |
| Pentium NT IBM JDK 1.1.7 | 148.1 | 10.6 | 2.27 | 2.09 | 13.6 | 8.8 | 2.31 | 2.07 |
| Pentium NT Microsoft JDK 1.1 | 132.3 | 73.5 | 2.96 | 2.55 | 59.1 | 19.5 | 3.04 | 2.58 |
| Pentium NT Visual C++ | 702.0 | — | 158.8 | 140.9 | 260.5 | — | 7.16 | 228.5 |
| Pentium Linux JDK 1.1.7 | 43.4 | 15.1 | 2.13 | 1.82 | 26.4 | 5.2 | 2.16 | 1.80 |
| Pentium Linux Kaffe 1.03beta | 72.8 | 29.2 | 2.35 | 1.65 | 39.2 | 6.2 | 2.28 | 1.66 |
| Pentium Linux gcc | 323.5 | — | 156.1 | 108.3 | 123.4 | — | 153.1 | 91.3 |
| Pentium Solaris JDK 1.1.6 | 43.1 | 42.1 | 2.44 | 2.13 | 29.6 | 6.3 | 2.49 | 2.11 |
| Pentium Solaris JDK 1.2 | 112.4 | 41.8 | 2.39 | 2.14 | 61.8 | 8.6 | 2.43 | 2.03 |
| Pentium Solaris Kaffe 1.03beta | 76.4 | 28.6 | 2.36 | 1.66 | 39.2 | 6.3 | 2.28 | 1.66 |
| Pentium Solaris gcc | 327.4 | — | 156.2 | 156.0 | 123.7 | — | 153.4 | 148.8 |
| UltraSPARC Solaris JDK 1.1.6 | 197.1 | 94.7 | 86.1 | 71.7 | 49.8 | 9.6 | 84.5 | 65.4 |
| UltraSPARC Solaris JDK 1.2 | 147.7 | 75.0 | 76.2 | 66.6 | 44.6 | 13.2 | 76.3 | 49.8 |
| UltraSPARC Solaris Kaffe 1.03beta | 67.8 | 42.7 | 35.6 | 25.0 | 8.7 | 1.8 | 35.8 | 25.7 |
| UltraSPARC Solaris gcc | 256.6 | — | 125.9 | 126.5 | 15.2 | — | 12.5 | 12.8 |
| Alpha Digital Unix JDK 1.1.6 | 31.2 | 27.4 | 0.10 | 0.07 | 10.6 | 8.9 | 0.09 | 0.07 |
| Alpha Digital Unix JDK 1.2beta | 31.1 | 25.3 | 0.12 | 0.06 | 10.6 | 8.9 | 0.08 | 0.06 |
| Alpha Digital Unix gcc | 312.7 | 296.3 | 78.8 | 77.1 | 24.0 | 25.4 | 99.1 | 95.0 |
| SGI O2 Irix JDK 1.1.5 | 75.6 | 62.1 | 25.2 | 19.7 | 22.0 | 12.9 | 25.9 | 0.11 |
| SGI O2 Irix gcc | 65.0 | 64.9 | 53.8 | 54.0 | 29.3 | 41.4 | 55.9 | 50.9 |
| PowerPC LinuxPPC JDK 1.1.6 | 196.3 | 37.7 | 58.3 | 58.3 | 113.0 | 6.6 | 38.9 | 33.3 |

Table 1: Selected results from the Arith low-level benchmark from version 1.0 of the Java Grande Forum Benchmark Suite, in millions of operations per second (so higher values are better). We also provide results for a C version of this benchmark. We have only provided C results for `long` integers on the 64-bit Alpha processor, which supports 64-bit `longs`.

performance of native C or Fortran code. For example, the 266 MHz Pentium II processor has a peak performance of 266 MFlops/sec, and it is possible to approach this value with synthetic benchmarks in C and using a highly optimised compiler. However on the Java benchmark code it achieves only 3 MFlops/sec. With native code, the Pentium provides much better floating point performance than the 233 MHz PowerPC processor in the iMac, however using Java under Solaris, NT or Linux on the Pentium gives results that are an order of magnitude slower than Linux on the iMac. The Irix floating point results are reasonably good, apart from multiplication of doubles, which surprisingly is 200 times slower than for floats! Floating point performance is best on the UltraSPARC, which attains values comparable to C code compiled with the generic gcc compiler. These results indicate that the quality of the JVM and the Java compiler can have much more of an effect on raw floating point performance than the processor speed.

The Alpha processor, which is comparable to the Pentium and UltraSPARC for C floating point benchmarks, is around 800 times slower for the Java floating point benchmarks! This may be due to an anomaly in the compiler, since it does not fare nearly so badly on the floating point intensive kernels such as Linpack (LUFact) and NAS FT. On some kernels the Alpha gives reasonable performance, but on most of the new kernels and low-level benchmarks it is very poor. There is little improvement between JDK 1.1.6 and the currently available JDK 1.2beta for the Alpha.

Object and array creation is much faster in JDK 1.2, improving by about 40% under NT and by an order of magnitude under Solaris. Creating a simple object is much faster in NT than in Solaris, but Solaris is faster at creating more complex objects and arrays.

Under NT and Solaris for SPARC, the empty method calls in the Java Grande Forum benchmark suite are hundreds of times faster than calling non-empty methods, which may be an indication that the compilers are optimising over them out. The overhead of method calls has been decreased by about a factor of two for Solaris on the UltraSPARC.

Java allows for synchronized methods that are thread-safe, and synchronized blocks of code within standard (unsynchronized) methods. We have tested these using a simple method that increments a variable, acting like a counter. A program with multiple threads that were all required to in-

| JVM | Object | | | Thread | | Method | |
|---|---|---|---|---|---|---|---|
| | Simple | Complex | Array | 2 | 16 | Synch | Unsynch |
| Celeron NT JDK 1.1.6 | 11.11 | 0.34 | 0.11 | 0.29 | 0.12 | 1.17 | 62.3 |
| Celeron NT JDK 1.2 | 14.23 | 0.44 | 0.09 | 0.27 | 0.12 | 2.29 | 62.0 |
| Pentium NT JDK 1.1.6 | 14.41 | 0.45 | 0.12 | 0.35 | 0.26 | 1.28 | 69.0 |
| Pentium NT JDK 1.2 | 19.78 | 0.61 | 0.11 | 0.33 | 0.25 | 2.53 | 69.0 |
| Pentium NT IBM JDK 1.1.7 | 1.31 | 0.84 | 0.18 | 0.31 | 0.28 | 3.31 | 5.5 |
| Pentium NT Microsoft JDK 1.1 | 1.45 | 0.67 | 0.37 | 0.23 | 0.23 | 4.97 | 27.8 |
| Pentium Linux JDK 1.1.7 | 0.59 | 0.29 | 0.08 | 0.19 | 0.13 | 1.21 | 3.9 |
| Pentium Linux Kaffe 1.03beta | 0.08 | 0.04 | 0.05 | 0.45 | 0.33 | 0.31 | 1.9 |
| Pentium Solaris JDK 1.1.6 | 0.47 | 0.22 | 0.11 | 0.27 | 0.04 | 0.52 | 19.2 |
| Pentium Solaris JDK 1.2 | 4.16 | 1.80 | 0.19 | 0.29 | 0.28 | 0.68 | 15.7 |
| Pentium Solaris Kaffe 1.03beta | 0.22 | 0.11 | 0.08 | 0.97 | 0.73 | 0.74 | 3.8 |
| UltraSPARC Solaris JDK 1.1.6 | 0.66 | 0.29 | 0.17 | 0.48 | 0.04 | 0.47 | 9.1 |
| UltraSPARC Solaris JDK 1.2 | 6.50 | 4.67 | 0.28 | 2.58 | 2.55 | 2.94 | 14.4 |
| UltraSPARC Solaris Kaffe 1.03beta | 0.17 | 0.09 | 0.09 | 0.31 | 0.26 | 0.49 | 3.1 |
| Alpha Digital Unix JDK 1.1.6 | 0.11 | 0.06 | 0.06 | 0.05 | 0.04 | 0.13 | 1.4 |
| Alpha Digital Unix JDK 1.2beta | 0.05 | 0.02 | 0.06 | 0.08 | 0.06 | 0.13 | 0.3 |
| SGI O2 Irix JDK 1.1.5 | 0.28 | 0.14 | 0.04 | 0.02 | 0.02 | 0.82 | 4.1 |
| PowerPC LinuxPPC JDK 1.1.6 | 0.34 | 0.16 | 0.07 | 0.19 | 0.09 | 0.35 | 4.9 |

Table 2: Selected results from our new low-level benchmarks, in millions of operations per second (so higher values are better). The Object benchmarks give results for creating a simple object, a more complex object, and an integer array of size 128. The Thread benchmarks give results for switching between 2 or 16 threads. The Method benchmarks give results for calling a method that increments a counter, using a standard unsynchronized method (Unsynch) and a standard method containing a synchronized block of code for the counter update (Synch).

crement the same counter should use synchronization. The results are shown in Table 2. Synchronization in JDK 1.2 is about 6 times faster on the UltraSPARC, and about 2 times faster for NT. Despite no improvement in Solaris on Pentium, this JVM remains an order of magnitude faster in synchronization than the others. Synchronization remains very expensive, with an overhead of 5-30 times in most cases. The IBM and Microsoft JDKs provide lower synchronization overhead than the Sun JDK under NT.

For JDK 1.1 under Solaris, performance was severely degraded as more threads were added, but this overhead has been virtually eliminated in JDK 1.2. Thread switching on the UltraSPARC is several times faster than on the Pentium. Under Solaris for both Pentium and UltraSPARC, JDK 1.2 allows threads to be run concurrently if multiple processors are available.

The performance of the different Java implementations varies widely across the various low-level and kernel benchmarks, with most of the JDKs performing very well on some benchmarks and quite poorly on others. Particularly for the low-level benchmarks, the best results for different operations are often from different JDKs, indicating that they all have room for improvement. In some cases the differences between JDKs can be an order of magnitude or more.

The beta version of the Kaffe JDK provides better performance for Linux than the standard Sun JDK, but performs relatively poorly under Solaris and NT. Object and array creation and thread switching are poor for all the Kaffe JDKs. Some of the Kaffe results seem incongruous and may be due to bugs in the beta code.

The IBM JDK for Windows NT has relatively poor performance in integer arithmetic, assignment, object creation, method calls and loops, but nevertheless does well on most of the kernels. There are some marked differences between the Sun, IBM, Microsoft and Kaffe results on the low-level benchmarks, but the performance on the arithmetic benchmarks and the kernels is much closer, generally within a factor of two and mostly within a few percent.

It is interesting to compare the results of benchmarks written in Java with equivalent C or Fortran code. In the case of the Fibonacci benchmark, where the main overhead is in the recursive method calls, the Java version under NT outperforms all the other platforms, even when they are running C code. For Tower of Hanoi benchmarks, the C code is only about twice as fast as the Java version, which is quite acceptable, and about as good as could be expected in most cases.

Java has substantial array indexing and array bounds checking overheads, which in some cases may be amortized by the amount of numerical computation required in some of the benchmarks. However for benchmarks such as prime number sieving and integer sorting, which involve minimal

| JVM | Fourier | Search | LUFact | HeapSort | Crypt |
|---|---|---|---|---|---|
| Celeron NT JDK 1.1.6 | 30.2 | 17.0 | 4.88 | 5.11 | 5.57 |
| Celeron NT JDK 1.2 | 28.7 | 15.1 | 4.78 | 4.91 | 5.73 |
| Pentium NT JDK 1.1.6 | 27.3 | 13.4 | 4.66 | 3.19 | 5.08 |
| Pentium NT JDK 1.2 | 26.0 | 12.6 | 4.64 | 3.23 | 5.24 |
| Pentium NT IBM JDK 1.1.7 | 22.6 | 10.0 | 4.93 | 3.27 | 4.78 |
| Pentium NT Microsoft JDK 1.1 | 33.4 | 12.1 | 4.90 | 3.41 | 10.18 |
| Pentium Linux JDK 1.1.7 | 71.7 | 37.9 | 15.11 | 11.89 | 15.90 |
| Pentium Linux Kaffe 1.03beta | 47.9 | — | 12.07 | 9.04 | 15.90 |
| Pentium Solaris JDK 1.1.6 | 65.1 | 23.6 | 10.47 | 5.56 | 16.00 |
| Pentium Solaris JDK 1.2 | 71.4 | 13.7 | 6.29 | 5.69 | 14.35 |
| Pentium Solaris Kaffe 1.03beta | 70.0 | — | 17.58 | 8.68 | 17.15 |
| UltraSPARC Solaris JDK 1.1.6 | 50.8 | 26.0 | 4.27 | 5.20 | 13.91 |
| UltraSPARC Solaris JDK 1.2 | 69.8 | 13.9 | 4.13 | 3.47 | 12.71 |
| UltraSPARC Solaris Kaffe 1.03beta | 63.6 | — | 93.10 | 9.33 | 3.37 |
| Alpha Digital Unix JDK 1.1.6 | 147.5 | 94.5 | 11.53 | 12.95 | 20.34 |
| Alpha Digital Unix JDK 1.2 | 515.7 | 157.9 | 12.09 | 13.29 | 20.30 |
| SGI O2 Irix JDK 1.1.5 | 87.7 | 59.5 | 18.50 | 22.09 | 18.54 |
| PowerPC LinuxPPC JDK 1.1 | 82.6 | 40.0 | 9.95 | 1.22 | 26.31 |

Table 3: Timings in seconds of kernels from version 1.0 of the Java Grande Forum Benchmark Suite (so lower values are better). The Search benchmark fails under Kaffe.

number crunching, the native code can be faster by an order of magnitude or more.

## 5 Conclusions and Future Work

Benchmarking can be a difficult and painstaking task in situations where many interacting effects occur, which is certainly the case for Java, with its interpreted virtual machine environment and built-in memory management. The problem is compounded by the multiplicity of Java compilers and JVMs which run on such a variety of hardware and operating systems environments, and there is clearly a lot of work required to analyze the mass of data obtained from all these benchmarks. Consequently we must be cautious in drawing firm conclusions from our data, but we believe there are some interesting general trends which emerge from the current Java implementations.

In general it does appear that while still much criticised, the performance of Java environments has improved significantly and in our opinion makes the use of Java for scientific work viable in many cases. There is still scope for considerable optimisation of some of the low level operations however, and JavaSoft and other JVM developers are to be congratulated but encouraged to improve performance in the problem areas identified by the benchmarks. The main areas where Java still has performance problems that could substantially impact Java Grande applications are in floating point arithmetic and array indexing.

While some of the classic coding considerations such as method and loop overheads do not appear appreciably worse in Java environments than in C or C++ compilers for example, there do still appear to be significant overheads in some areas such as array indexing and object creation. These observations may be important to applications developers who may therefore prefer to try to express certain applications code components in a more procedural style than Java might otherwise encourage. This is a particular concern for array style operations where Java's array of arrays imposes a significant overhead on operations.

Currently the platforms that appear to offer the best performance for Java Grande applications are Solaris on UltraSPARC and NT on Pentium, which is not too surprising since these are presumably the platforms that have attracted the most Java development effort. The recently available LinuxPPC for the the PowerPC processor used in Power Macintoshes and iMacs also performs quite creditably for some applications.

SGI MIPS and DEC Alpha processors both have a reputation for excellent number-crunching performance and are often used for scientific applications, however the Java environment and compilers for both these platforms are still rather immature, and their performance for many Java applications is very poor, particularly for the Alpha. Pentium processors running Linux have recently become very popular for scientific applications due to their excellent price to performance ratio, however the JDKs available for this platform are fairly immature and do not provide competitive performance for Java Grande applications.

The Java Grande benchmarks generally provide very useful information on the performance of Java for scientific applications on different platforms. However the current benchmark suite has some shortcomings, which we have gone some way toward addressing. Version 1.0 of the Java Grande benchmarks does not have native code implemen-

| JVM | Fibonacci | Sieve | Hanoi | NAS EP | NAS IS | NAS FT |
|---|---|---|---|---|---|---|
| Celeron NT JDK 1.1.6 | 8.6 | 1.38 | 5.3 | 66.9 | — | — |
| Celeron NT JDK 1.2 | 8.6 | 1.39 | 5.2 | 56.4 | — | — |
| Pentium NT JDK 1.1.6 | 7.8 | 1.26 | 4.8 | 60.6 | 38.6 | 17.8 |
| Pentium NT JDK 1.2 | 7.8 | 1.26 | 4.8 | 51.5 | 38.9 | 18.1 |
| Pentium NT IBM JDK 1.1.7 | 8.7 | 1.70 | 3.1 | 147.1 | 35.1 | 19.6 |
| Pentium NT Microsoft JDK 1.1 | 13.0 | 1.92 | 5.3 | 79.1 | 39.2 | 18.0 |
| Pentium NT Visual C++ | 8.9 | 0.85 | 2.4 | — | — | — |
| Pentium Linux JDK 1.1.7 | 51.2 | 7.99 | 23.7 | 185.5 | 178.4 | 82.0 |
| Pentium Linux Kaffe 1.03b | 68.2 | 6.35 | 23.2 | 217.4 | 325.1 | 64.4 |
| Pentium Linux gcc/g77 | 13.0 | 0.20 | 4.6 | 36.5 | 3.4 | — |
| Pentium Solaris JDK 1.1.6 | 15.4 | 4.25 | 8.5 | 142.4 | 57.6 | 30.1 |
| Pentium Solaris JDK 1.2 | 15.2 | 4.40 | 6.9 | 114.2 | 80.6 | 25.9 |
| Pentium Solaris Kaffe 1.03beta | 54.8 | 6.33 | 20.1 | 284.4 | 136.7 | 49.0 |
| Pentium Solaris gcc/g77 | 13.2 | 0.20 | 4.5 | 33.2 | 2.1 | — |
| UltraSPARC Solaris JDK 1.1.6 | 29.5 | 2.93 | 11.4 | 52.6 | 78.3 | 17.6 |
| UltraSPARC Solaris JDK 1.2 | 18.0 | 3.76 | 6.5 | 67.8 | 61.9 | 23.7 |
| UltraSPARC Solaris Kaffe 1.03beta | 79.5 | 8.92 | 6.8 | 119.2 | 182.7 | 41.3 |
| UltraSPARC Solaris gcc/g77 | 11.9 | 0.91 | 3.2 | 72.2 | 1.2 | — |
| Alpha Digital Unix JDK 1.1.6 | 161.9 | 7.70 | 47.1 | 202.4 | 148.8 | 47.5 |
| Alpha Digital Unix JDK 1.2beta | 694.9 | 7.32 | 188.5 | 851.8 | 318.6 | 51.8 |
| Alpha Digital Unix gcc/f77 | 28.5 | 0.58 | 14.1 | 30.8 | 2.7 | 6.8 |
| SGI O2 Irix JDK 1.1.5 | 62.6 | 12.91 | 33.0 | 168.3 | 187.3 | 51.2 |
| SGI O2 Irix gcc/g77 | 25.1 | 0.35 | 12.2 | 23.3 | 6.8 | 7.5 |
| PowerPC LinuxPPC JDK 1.1 | 41.6 | 3.09 | 23.4 | 176.8 | — | — |
| PowerPC LinuxPPC gcc/g77 | 14.5 | 1.2 | 5.7 | — | — | — |

Table 4: Timings in seconds of our new Java Grande benchmarks (so lower values are better). The NAS benchmarks use the S (small) problem size except for the IS benchmark where the W (workstation) problem size is used, since S is too small (takes less than a second on some platforms). The Fortran version of the NAS FT code does not compile under g77.

tations to allow a performance comparison with the Java versions, which we believe is extremely important. These should be simple to add for the existing kernels. The low-level benchmarks have potential problems in advanced compilers optimising out some operations. We have identified some of these problems, but others still remain. The benchmark suite also requires more scientific application kernels, and more low-level benchmarks to cover a range of features that are used in scientific applications. We have provided some of these in this work, but a wider variety of kernels and applications would be very useful.

Some of the new kernel benchmarks are fairly simplistic and not directly representative of scientific computations, but provide useful benchmarks of features that are commonly used in scientific computations, such as array indexing (Sieve of Eratosthenes), recursion (Fibonacci) and certain kinds of sorting (Tower of Hanoi). The other new kernels (the NAS benchmarks) are scientific computations that are more directly relevant to Java Grande applications. It may be worthwhile to break up the kernels in the benchmark suite into a set of scientific kernels such as the NAS benchmarks and matrix routines such as LU factorization, and a set of simpler and more general numerical kernels, which sit somewhere between the low-level benchmarks and the scientific kernels.

There are a number of areas that are important to Java Grande applications but are not addressed adequately (or at all) by current Java benchmarks. I/O performance is crucial for many large-scale scientific applications, however no current benchmark adequately tests Java object serialization and I/O performance over a range of data types and sizes. Many standard Java benchmarks are packaged as applets, and the applet "sandbox" model does not allow for testing I/O from local disk.

There is a dearth of benchmarks available for testing the performance of parallel or distributed Java applications or application kernels that use Java RMI, parallel threads, or the proposed Java MPI bindings. Parallel Java versions of the Linpack and NAS Parallel Benchmarks would be particularly interesting. There are also no large-scale scientific applications currently available as Java benchmarks. We aim to tackle some of these areas in future work.

**Acknowledgements**

This work was carried out under the Distributed High Performance Computing Infrastructure (DHPC-I) project and the Online Data Archives (OLDA) program of the Advanced Computational Systems (ACSys) Cooperative Research Centre (CRC), and supported by the Research Data Networks (RDN) CRC. ACSys and RDN are funded under the Australian Commonwealth Government's CRC program. We would like to thank Duncan Grove for assisting with benchmarking of some of the C and Fortran programs.

**References**

[1] Doug Bell, Make Java fast: Optimize, JavaWorld, April 1997, http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html.

[2] BYTE Magazine, BYTE Benchmarks, http://www.byte.com/bmark/bmark.htm.

[3] Bryan Carpenter *et al.*, MPI for Java - Position Document and Draft API Specification, Java Grande Forum Technical Report JGF-TR-03, November 1998, http://www.npac.syr.edu/projects/pcrc/reports/MPIposition/position.ps.

[4] Distributed and High-Performance Computing Group, University of Adelaide, Java Grande Benchmarks, http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/.

[5] Jack Dongarra and Reed Wade, Linpack Benchmark – Java version, http://www.netlib.org/benchmark/linpackjava/.

[6] Edinburgh Parallel Computing Centre, Java Grande Forum Benchmark Suite, http://www.epcc.ed.ac.uk/research/javagrande/benchmarking.html.

[7] Edinburgh Parallel Computing Centre, List of Java benchmarks, http://www.epcc.ed.ac.uk/research/javagrande/list.html.

[8] James Gosling, Bill Joy and Guy Steele, Java Language Specification, Addison Wesley, 1996.

[9] Bill Griswald and Paul Phillips, Bill and Paul's Excellent UCSD Benchmarks for Java, http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.

[10] K.A. Hawick *et al.*, DISCWorld: An Environment for Service-Based Metacomputing, to be published in J. of Future Generation Computing Systems. Also available as DHPC Technical Report DHPC-042, April 1998, http://www.dhpc.adelaide.edu.au/reports/042/abs-042.html.

[11] HPJava Project, http://www.npac.syr.edu/projects/pcrc/HPJava/.

[12] HPJava Project, mpiJava Home Page, http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html.

[13] Java Grande Forum, http://www.javagrande.org/.

[14] National Institue of Standards and Technology, Java SciMark Benchmark for Scientific Computing, http://math.nist.gov/scimark/.

[15] Numerical Aerospace Simulation Facility, NAS Parallel Benchmarks, http://science.nas.nasa.gov/Software/NPB/.

[16] PC Magazine, JMark 1.01, http://www8.zdnet.com/pcmag/pclabs/bench/benchjm.htm.

[17] Pendragon Software Corporation, CaffeineMark 3.0, http://www.pendragon-software.com/pendragon/cm3/index.html.

[18] Standard Performance Evaluation Corporation, SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98/.

[19] Gabriel Zachmann, Java/C++ Benchmark, http://www.igd.fhg.de/~zach/benchmarks/.