A Benchmark Suite for High Performance Java

J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty and R. A. Davey
EPCC, James Clerk Maxwell Building, The King's Buildings,
The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ,
Scotland, U.K.
email: epcc-javagrande@epcc.ed.ac.uk

Abstract

Increasing interest is being shown in the use of Java for large scale or *Grande* applications. This new use of Java places specific demands on the Java execution environments that could be tested and compared using a standard benchmark suite. We describe the design and implementation of such a suite, paying particular attention to Java-specific issues. Sample results are presented for a number of implementations of the Java Virtual Machine (JVM).

1 Introduction

With the increasing ubiquity of Java comes a growing range of uses for the language that fall well outside its original design specifications. The use of Java for large scale applications with large memory, network or computational requirements, so called *Grande* applications, represent a clear example of this trend. Despite concerns about performance and numerical definitions an increasing number of users are taking seriously the possibility of using Java for Grande codes.

The Java Grande Forum (JGF) is a community initiative led by Sun and the Northeast Parallel Architectures Center (NPAC) which aims to address these issues and in so doing promote the use of Java in this area. This paper describes work carried out by EPCC in the University of Edinburgh on behalf of the JGF to initiate a benchmark suite aimed at testing aspects of Java execution environments, (JVMs, Java compilers, Java hardware etc.) pertinent to Grande Applications. The work involves constructing a framework for the benchmarks, designing an instrumentation class to ensure standard presentation of results, and seeding the suite with existing and original benchmark codes.

The aim of this work is ultimately to arrive at a standard benchmark suite which can be used to:

- Demonstrate the use of Java for Grande applications. Show that real large scale codes can be written and provide the opportunity for performance comparison against other languages.
- Provide metrics for comparing Java execution environments thus allowing Grande users to make informed decisions about which environments are most suitable for their needs.
- Expose those features of the execution environments critical to Grande Applications and in doing so encourage the development of the environments in appropriate directions.

A standard approach, ensuring that metrics and nomenclature are consistent, is important in order to facilitate meaningful comparisons in the Java Grande community. The authors are keen to invite contributions from the community to add to the benchmark suite and comments on the approach taken.

The remainder of this paper is structured as follows: Section 2 gives a brief survey of related work. Sections 3 and 4 outline the methodology we adopted in designing this suite and describe the instrumentation API. Sections 5 and 6 give the current status of the serial part of the suite, and some results which illustrate the existing suite in action. Section 7 outlines directions for future work, concentrating on the parallel part of the suite, and invites participation in this effort, and Section 8 provides some conclusions.

2 Related work

A considerable number of benchmarks and performance tests for Java have been devised. Some of these consist of small applets with relatively light computational load, designed mainly for testing JVMs embedded in browsers—these are of little relevance to Grande applications. Of more interest are a number of benchmarks [2, 7, 8, 9, 16] which focus on determining the performance of basic operations such as arithmetic, method calls, object creation and variable accesses. These are useful for highlighting differences between Java environments, but give little useful information about the likely performance of large application codes. Other sets of benchmarks, from both academic [5, 14, 15, 19] and commercial [13, 17, 20] sources, consist primarily of computational kernels, both numeric and non-numeric. This type of benchmark is more reflective of application performance, though many of the kernels in these benchmarks are on the small side, both in terms of execution time and memory requirements. Finally there are some benchmarks [3, 10, 18] which consist of a single, near full-scale, application. These are useful in that they can be representative of real codes, but it is virtually impossible to say why performance differs from one environment to another, only that it does.

Few benchmark codes attempt inter-language comparison. In those that do, (for example [16, 19]) the second language is usually C++, and the intention is principally to compare the object oriented features. It is worth noting a feature peculiar to Java benchmarking, which is that it is possible to distribute the benchmark without revealing the source code. This may be convenient, but if adopted, makes it impossible for the user community to know exactly what is being tested.

3 Methodology

In this Section we discuss the principal issues affecting the design of a benchmark suite for Java Grande applications, and describe how we have addressed these issues.

For a benchmark suite to be successful, we believe it should be:

- **Representative**: The nature of the computation in the benchmark suite should reflect the types of computation which might be expected in Java Grande applications. This implies that the benchmarks should stress Java environments in terms of CPU load, memory requirements, and I/O, network and memory bandwidths.
- **Interpretable**: As far as possible, the suite as a whole should not merely report the performance of a Java environment, but also lend some insight into why a particular level of performance was achieved.
- **Robust**: The performance of suite should not be sensitive to factors which are of little interest (for example, the size of cache memory, or the effectiveness of dead code elimination).
- **Portable**: The benchmark suite should run on as wide a variety of Java environments as possible.
- **Standardised**: The elements of the benchmark should have a common structure and a common 'look and feel'. Performance metrics should have the same meaning across the benchmark suite.
- **Transparent**: It should be clear to anyone running the suite exactly what is being tested.

We observe that the first two of these aims (representativeness and interpretability) tend to conflict. To be representative, we would like the contents of the benchmark to be as much like real applications as possible, but the more complex the code, the harder it is to interpret the observed performance. Rather than attempt to meet both these objectives at once, we provide three types of benchmark, reflecting the classification of existing benchmarks used in Section 2: low-level operations (which we refer to as Section I of the suite), simple kernels (Section II) and applications (Section III). This structure is employed for both the serial and parallel parts of the suite.

The low-level operation benchmarks have been designed to test the performance of the low-level operations which will ultimately determine the performance of real applications running under the Java environment. Examples include arithmetic and maths library operations, serialization, method calls and casting in the serial part and ping-pong, barriers and global reductions in the parallel part. The kernel benchmarks are chosen to be short codes, each containing a type of computation likely to be found in Grande applications, such as FFTs, LU Factorisation, matrix multiplication, searching and sorting. The application benchmarks are intended to be representative of Grande applications, suitably modified for inclusion in the benchmark suite by removing any I/O and graphical components. By providing these different types of benchmark, we hope to observe the behaviour of the most complex applications and interpret that behaviour through the behaviour of the simpler codes. We also choose the kernels and applications from a range of disciplines, which are not all traditional scientific areas.

To make our suite robust, we avoid dependence on particular data sizes by offering a range of data sizes for each benchmark in Sections II and III. We also take care to defeat possible compiler optimisation of strictly unnecessary code. For Sections II and III this is achieved by validating the results of each benchmark, and outputting any incorrect results. For Section I, even more care is required as the operations performed are rather simple. We note that some common tricks, used to fool compilers into thinking that results are actually required, may fail in interpreted systems where optimisations can be performed at run time. Another potential difficulty, particularly relevant to Section I benchmarks, is that very simple codes may fail to trigger run-time code compilation. Typically each JVM uses some heuristics, based on run-time statistics, to determine when to use its just-in-time (JIT) compiler and switch from interpreted byte-code to a compiled version. Microbenchmarks may therefore reflect the interpreted, rather than the compiled, performance of the JVM. This is sometimes referred to as the JIT warm-up problem. This effect may, however also impact on larger codes. Some Grande applications may spend a significant amount of time in methods which are called only a few times. Failure to trigger the JIT in these cases may have a significantly detrimental effect.

For maximum portability, as well as ensuring adherence to standards, we have taken the decision to have no graphical component in the benchmark suite. While applets provide a convenient interface for running benchmarks on workstations and PCs, this is not true for typical supercomputers where interactive access may not be possible. Thus we restrict ourselves to simple file I/O.

For standardisation we have created a JGFInstrumentor class to be used in all benchmark programs. This is described in detail in Section 4.

Transparency is achieved by distributing the source code for all the benchmarks. This removes any ambiguity in the question of what is being tested: we do not consider it acceptable to distribute benchmarks in Java byte code form.

4 Instrumentation

4.1 Performance metrics

We present performance metrics for the benchmarks in three forms: execution time, temporal performance and relative performance. The execution time is simply the wall clock time required to execute the portion of the benchmark code which comprises the 'interesting' computation—initialisation, validation and I/O are excluded from the time measured. For portability reasons, we chose to use the `System.currentTimeMillis` method from the `java.lang` package. Millisecond resolution is less than ideal for measuring benchmark performance, so care must be taken that the run-time of all benchmarks is sufficiently long that clock resolution is not significant.

Temporal performance (see [6]) is defined in units of operations per second, where the operation is chosen to be the most appropriate for each individual benchmark. For example, we might choose floating point operations for a linear algebra benchmark, but this would be inappropriate for, say, a Fourier analysis benchmark which relies heavily on transcendental functions. For some benchmarks, where the choice of most appropriate unit is not obvious, we allow more than one operation unit to be defined.

Relative performance is the ration of temporal performance to that obtained for a reference system, that is a chosen JVM/operating system/hardware combination. The merit of this metric is that it can be used to compute the average performance over a groups of benchmark. Note that the most appropriate average is the geometric mean of the relative performances on each benchmark.

For the low-level benchmarks (Section I) we do not report execution times. This allows us to adjust the number of operations performed at run-time to give a suitable execution time, which is guaranteed to be much larger than the clock resolution. This overcomes the difficulty that there can be one or two orders of magnitude difference in performance on these benchmarks between different Java environments.

4.2 Design of instrumentation classes

Creating an instrumentation class raises some interesting issues in object-oriented design. Our objective is to be able to take an existing code and to both instrument it, and force it to conform with a common benchmark structure, with as few changes as possible.

A natural approach would be to create an abstract benchmark class which would be sub-classed by an existing class in the benchmark's hierarchy: access to instrumentation would be via the benchmark class. However, since Java does not support multiple inheritance, this is not possible. Other options include:

- Inserting the benchmark class at some point in the existing hierarchy.
- Creating an instance of the benchmark class at some point in the existing hierarchy.
- Accessing benchmark methods as class methods.

The last option was chosen because minimal changes are required to existing code: the benchmark methods can be referred to from anywhere within existing code by a global name. However, we would like, for instance, to be able to access multiple instances of a timer object. This can achieved by filling a hash-table with timer objects. Each timer object can be given a global name through a unique string.

We can force compliance to common structure to some extent by sub-classing the lowest level of the main hierarchy in the benchmark, and implementing a defined `interface`, which includes a 'run' method. We can then create a separate `main` class which creates an instance of this sub-class and calls its 'run' method. It is then straightforward to create a `main` which, for example, runs all the benchmarks of a given size in a given Section.

4.3 The JGF Benchmark API

Figure 1 describes the API for the benchmark class. `addTimer` creates a new timer and assigns a name to it. The optional second argument assigns a name to the performance units to be counted by the timer. `startTimer` and `stopTimer` turn the named timer on and off. The effect of repeating this process is to accumulate the total time for which the timer was switched on. `addOpsToTimer` adds a number of operations to the timer: multiple calls are cumulative. `readTimer` returns the currently stored time. `resetTimer` resets both the time and operation count to zero. `printTimer` prints both time and performance for the named timer; `printperfTimer` prints just the performance. `storeData` and `retrieveData` allow storage and retrieval of arbitrary objects without, for example, the need for them to be passed through argument lists. This may be useful, for example, for passing iteration count data between methods without altering existing code. `printHeader` prints a standard header line, depending on the benchmark Section and data size passed to it.

Figure 2 illustrates the use of an interface to standardise the form of the benchmark. The interface for Section II is shown here; that for Section III is similar, while that for Section I is somewhat simpler.

To produce a conforming benchmark, a new class is created which `extends` the lowest class of the main hierarchy in the existing code and `implements` this interface. The `JGFrun` method should call `JGFsetsize` to set the data size, `JGFinitialise` to perform any initialisation, `JGFkernel` to run the main (timed) part of the benchmark, `JGFvalidate` to test the results for correctness, and finally `JGFtidyup` to permit garbage collection of any large objects or arrays. Calls to `JGFInstrumentor` class methods can be made either from any of these methods, or from any methods in the existing code, as appropriate.

5 Current Status

Currently the parallel codes are under development. The following serial codes are available in the release version 2.0.

5.1 Section I: Low Level Operations

Arith Measures the performance of arithmetic operations (add, multiply and divide) on the primitive data types int, long, float and double. Performance units are additions, multiplications or divisions per second.

Assign Measures the cost of assigning to different types of variable. The variables may be scalars or array elements, and may be local variables, instance variables or class variables. In the cases of instance and class variables, they may belong to the same class or to a different one. Performance units are assignments per second.

Cast Tests the performance of casting between different primitive types. The types tested are int to float and back, int to double and back, long to float and back, long to double and back. Performance units are casts per second. Note that other pairs of types could also be tested (e.g. byte to int and back), but these are too amenable to compiler optimisation to give meaningful results.

```
public class JGFInstrumentor{
// No constructor
// Class methods
    public static synchronized void addTimer(String name);
    public static synchronized void addTimer(String name, String opname);
    public static synchronized void startTimer(String name);
    public static synchronized void stopTimer(String name);
    public static synchronized void addOpsToTimer(String name, double count);
    public static synchronized double readTimer(String name);
    public static synchronized void resetTimer(String name);
    public static synchronized void printTimer(String name);
    public static synchronized void printperfTimer(String name);
    public static synchronized void storeData(String name, Object obj);
    public static synchronized void retrieveData(String name, Object obj);
    public static synchronized void printHeader(int section, int size);
}
```

Figure 1: API for the JGFInstrumentor class

```
public interface JGFSection2 {
     public void JGFsetsize(int size);
     public void JGFinitialise();
     public void JGFkernel();
     public void JGFvalidate();
     public void JGFtidyup();
     public void JGFrun(int size);
}
```

Figure 2: Interface definition for Section II

Create This benchmark tests the performance of creating objects and arrays. Arrays are created for ints, longs, floats and objects, and of different sizes. Complex and simple objects are created, with and without constructors. Performance units are arrays or objects per second.

Exception Measures the cost of creating, throwing and catching exceptions, both in the current method and further down the call tree. Performance units are exceptions per second.

Loop This benchmark measures loop overheads, for a simple for loop, a reverse for loop and a while loop. Performance units are iterations per second.

Serial Tests the performance of serialization, both writing and reading of objects to and from a file. The types of objects tested are arrays, vectors, linked lists and binary trees. Results are reported in bytes per second.

Math Measures the performance of all the methods in the `java.lang.Math class`. Performance units are operations per second. Note that for a few of the methods (e.g. exp, log, inverse trig functions) the cost also includes the cost of an arithmetic operation (add or multiply). This was necessary to produce a stable iteration which will not overflow and cannot be optimised away. However, it is likely the the cost of these additional operations is insignificant: if necessary the performance can be corrected by using the relevant result from the Arith benchmark.

Method Determines the cost of a method call. The methods can be instance, final instance or class methods, and may be called from an instance of the same class, or a different one. Performance units are calls per second. Note that final instance and class methods can be statically linked and are thus amenable to inlining. An infeasible high performance figure for these tests generally indicates that the compiler has successfully inlined these methods.

5.2 Section II: Kernels

Series Computes the first N Fourier coefficients of the function $f(x) = (x+1)^x$ on the interval 0,2. Performance units are coefficients per second. This benchmark heavily exercises transcendental and trigonometric functions.

LUFact Solves an $N \times N$ linear system using LU factorisation followed by a triangular solve. This is a Java version of the well known Linpack benchmark [4]. Performance units are Mflops per second. Memory and floating point intensive.

HeapSort Sorts an array of N integers using a heap sort algorithm. Performance is reported in units of items per second. Memory and integer intensive.

SOR The SOR benchmark performs 100 iterations of successive over-relaxation on an $N \times N$ grid. The performance reported is in iterations per second. Array access intensive.

Crypt Performs IDEA (International Data Encryption Algorithm [11]) encryption and decryption on an array of N bytes. Performance units are bytes per second. Bit/byte operation intensive.

FFT This performs a one-dimensional forward transform of N complex numbers. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions.

Sparse This uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references. An $N \times N$ sparse matrix is multiplied by a dense vector 200 times.

5.3 Section III: Applications

Euler Solves the time-dependent Euler equations for flow in a channel with a "bump" on one of the walls. A structured, irregular, $N \times 4N$ mesh is employed, and the solution method is a finite volume scheme using a fourth order Runge-Kutta method with both second and fourth order damping. The solution is iterated for 200 timesteps. Performance is reported in units of timesteps per second.

MonteCarlo A financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates N sample time series with the same mean and fluctuation as a series of historical data. Performance is measured in samples per second.

MolDyn A simple N-body code modelling the behaviour of N argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The solution is advanced for 100 timesteps. Performance is reported in units of timesteps per second.

Search Solves a game of connect-4 on a 6 x 7 board using a alpha-beta pruned search technique. The problem size is determined by the initial position from which the game in analysed. The number of positions evaluated, N, is recorded, and the performance reported in units of positions per second. Memory and integer intensive.

RayTracer This benchmark measures the performance of a 3D ray tracer. The scene rendered contains 64 spheres, and is rendered at a resolution of $N \times N$ pixels. The performance is measured in pixels per second.

6 Results

The benchmark suite has been run on a number of different execution environments on two different hardware platforms. The following JVMs have been tested on a 200MHz Pentium Pro with 256 Mb of RAM running Windows NT:

- Sun JDK Version 1.2.1_02 (production version)
- Sun JDK Version 1.2.1 (reference version) + Hotspot version 1.0
- IBM Win32 JDK Version 1.1.7
- Microsoft SDK Version 3.2

The following JVMs have also been tested on a 250MHz Sun Ultra Enterprise 3000 with 1Gb of RAM running Solaris 2.6:

- Sun JDK Version 1.2.1_02 (production version)
- Sun JDK Version (reference version) + Hotspot version 1.0

6.1 Programming language comparison

The benchmark suite has been developed to allow the performance of various execution environments on different hardware platforms to be tested. Also of interest is language comparisons, comparing the performance of Java versus other programming languages such as Fortran, C and C++. Currently, the LUFact and MolDyn benchmarks, allow programming language comparisons with Fortran77 and C. It is intended, however, that the parallel part of the suite will contain versions of well-known Fortran and C parallel benchmarks, thus facilitating further inter-language comparisons.

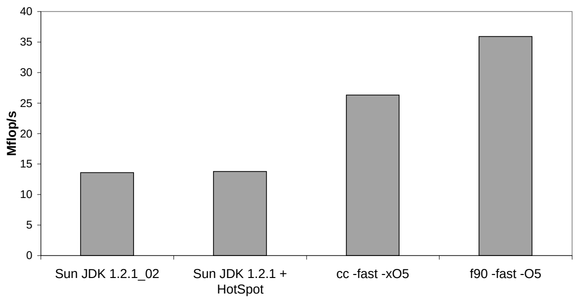
Measurements have been taken for the Linpack Benchmark (on a 1000×1000 problem size) and the Molecular Dynamics benchmark (2048 particles), using Java (Sun JDK 1.2.1_02 production version, and Sun JDK 1.2.1 reference version + Hotspot 1.0), Fortran and C on a 250MHz Sun Ultra Enterprise 3000 with 1Gb of RAM and the results are shown in Figure 3. For the Linpack code, both JVMs give performance which is approximately half that of C and one third that of Fortran. It should be noted that the LU factorisation code used in all cases was *not* optimised for cache reuse, hence the percentage of peak performance obtained by these codes is lower than would be expected for a well-tuned LU factorisation.

For the MolDyn code, the Hotspot JVM is about twice as fast as the production version, giving approximately two-thirds of the performance of C and nearly 90% the performance of Fortran. The relatively poor performance of Fortran on this code may be attributable to the data layout—both the C and Java implementations store all the fields for one particle in one data structure, whereas the Fortran implementation uses a separate array for storing each field for all the particles.

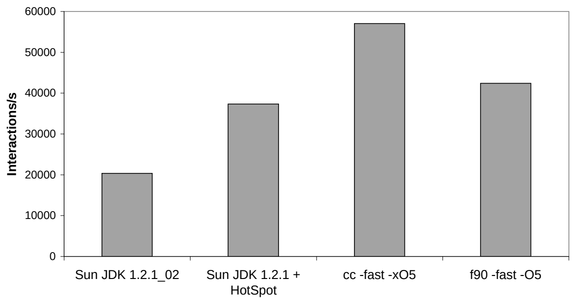
6.2 JVM comparison

Figure 4 shows the averaged relative performance for each of the tested systems on the three sections of the benchmark suite. For Sections II and III the medium data size was used. The reference implementation (relative performance = 1.0) was the Sun JDK 1.2.1_02 production version on a 250MHz Sun Ultra Enterprise 3000 running Solaris 2.6.

Perhaps the most striking feature of these results is that on both platforms, the Sun production JDK gives much higher performance on Section I that the other JVMs. The likely cause of this is JIT warmup effects. For some benchmarks, all the computation takes place within a single method call and it is known, for example, that the Hotspot JIT is not invoked for the first two calls to any method. The



(a)



(b)

Figure 3: Language comparisons for (a) the LUFact (Linpack) benchmark and (b) the MolDyn benchmark.

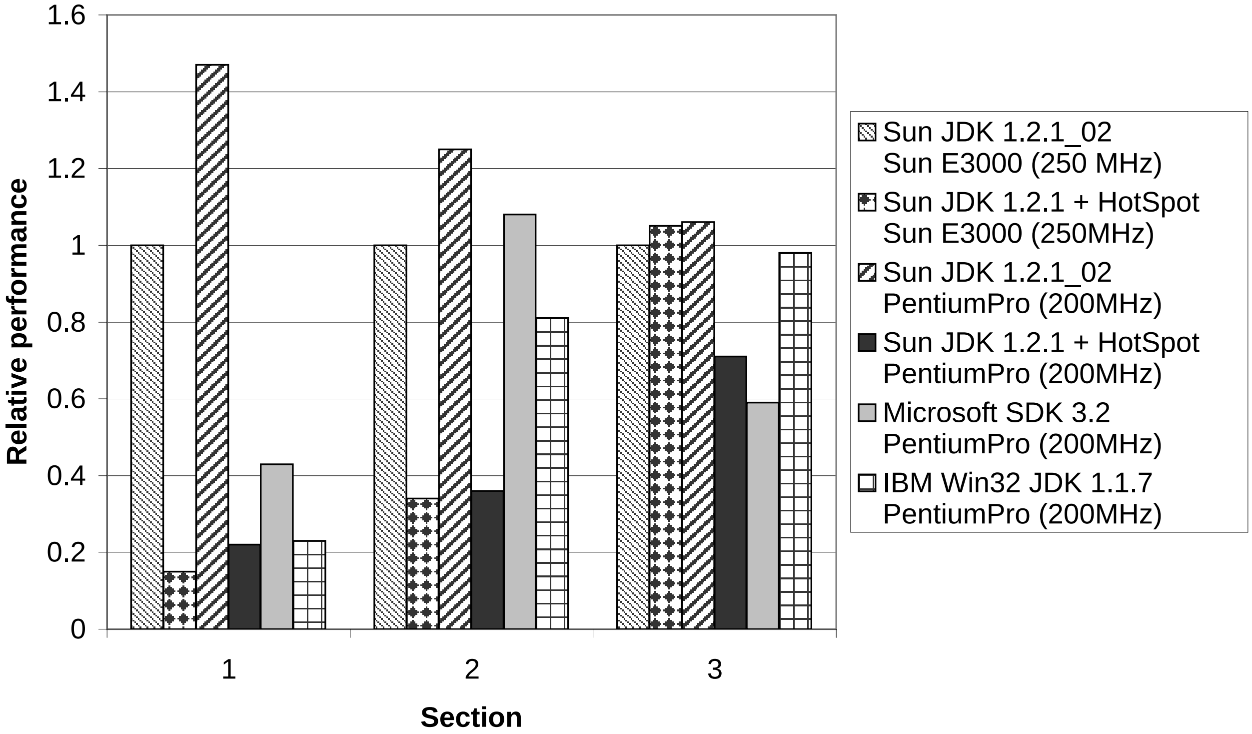


Figure 4: Relative performance of JDKs on Section 1, 2 and 3

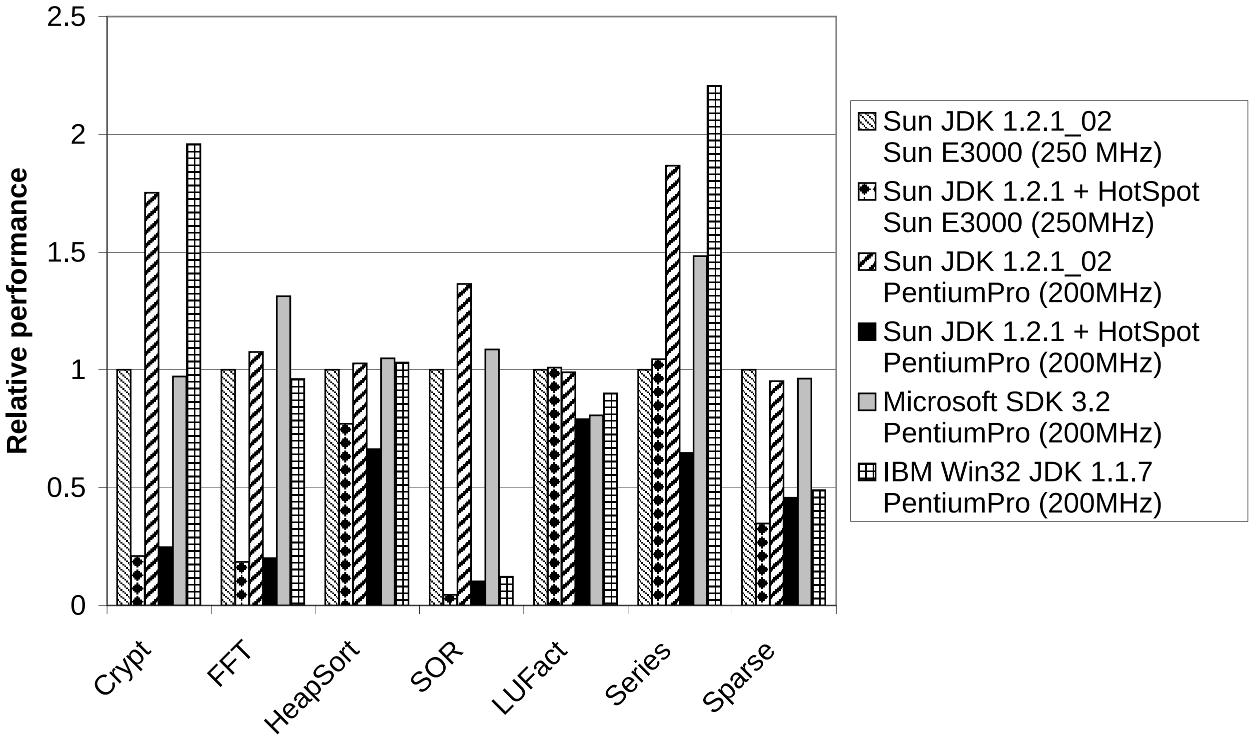


Figure 5: Relative performance of JDKs on Section 2 benchmarks, medium data size

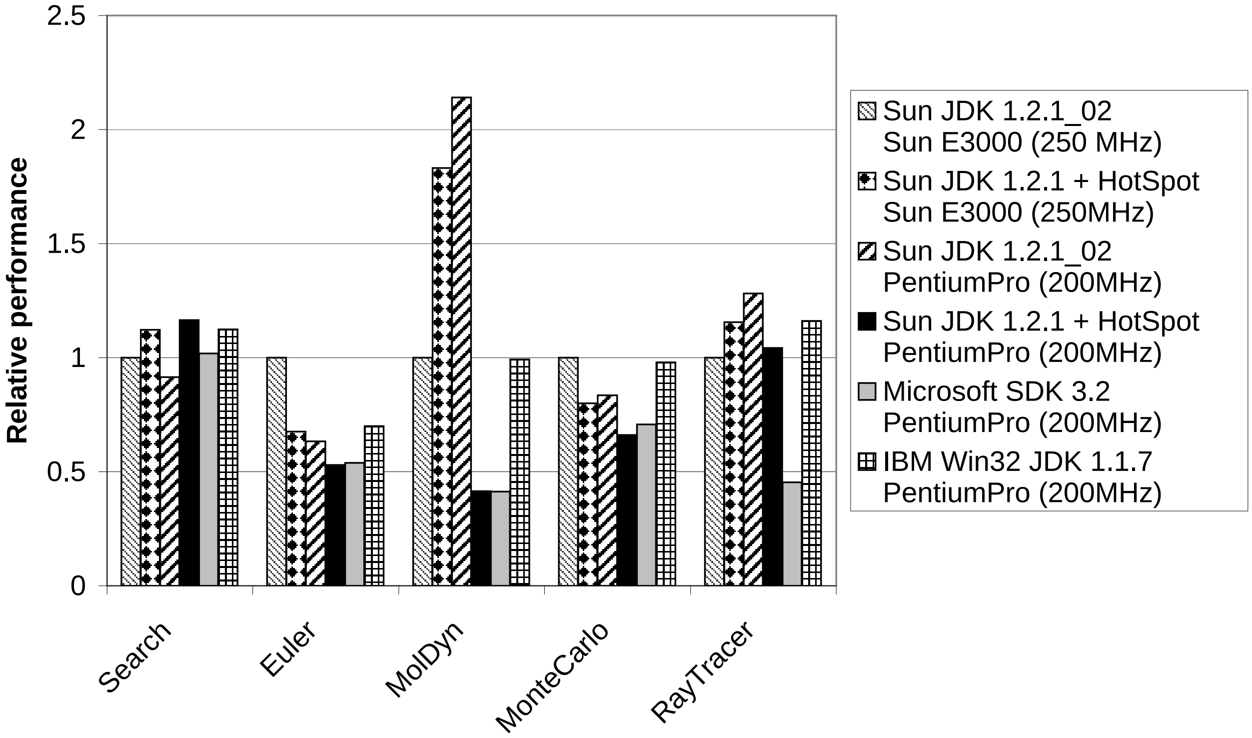


Figure 6: Relative performance of JDKs on Section 3 benchmarks, medium data size

Microsoft and IBM JDKs may also be suffering a similar problem. Additional efforts need to be made to make these benchmarks more robust, though it is difficult to anticipate all possible heuristics for JIT invocation.

For the Section II benchmarks the relative performance for each benchmark is shown in Figure 5. JIT warmup problems are evident for Hotspot on certain benchmarks (notably Crypt, FFT and SOR) and contribute to a low overall performance for this Section. For the other JVMs there is some variability on individual benchmarks, but the averaged scores are surprisingly close. No single JVM shows consistently better or worse performance.

This latter observation also holds true for Section III benchmarks (Figure 6). For example, the Windows NT version of Sun JDK with Hotspot gives the best performance on Search and the worst on Euler and MolDyn. Perhaps surprisingly, the variability in performance is very low. The MolDyn benchmark shows the greatest differences between best and worst performance (a factor of around 5.5), the others considerably less.

7 Future Work

All the benchmarks presented in this paper are serial and only utilise a single processor. However, one of our major aims is to increase the scope of the suite to include benchmarks that measure how efficiently Java codes and environments can utilise multi-processor architectures. Parallel machines are obvious target platforms for our work since they have the potential to supply the large amounts of CPU power and memory required by Grande applications. Understanding Java performance on a single processor is a necessary precursor to this work, and having developed the serial benchmark we will now start to include parallel benchmarks into the suite.

There are issues to be considered when developing any parallel benchmark, even when targeting well established languages such as C or Fortran. In the past, portability was a major issue, but with the almost universal adoption of the MPI standard this is no longer a problem. However, MPI provides many possible methods for even the most simple task of a single point-to-point communication and the performance of the different modes can vary dramatically on different platforms. A generic parallel benchmark using MPI must either attempt to measure as many different modes as possible (with the aim of allowing the programmer to choose the most efficient method) or simply time what is considered to be the most commonly used method (in an attempt to predict the performance of "typical" applications). This problem has lead to different disciplines developing their own parallel benchmarks designed to test the communication methods and patterns most commonly used in specific applications areas. For examples see OCCOMM [1] (ocean modelling) and SSB [12] (social sciences).

Developing a useful parallel Java benchmark is even more problematic since it is not yet clear which of the many parallelisation models to target. A parallel Java code might use built-in Java methods such as threads or RMI, Java library routines such as a Java implementation of MPI, or a Java interface to a platform-specific MPI implementation written in C. For the built-in methods, parallel performance may depend critically on the details of the JVM implementation, for example how effectively threads are scheduled across multiple processors.

Consideration of these issues has lead us to decide on the following strategy:

- Low-level benchmarks will be written to test the performance of the fundamental operations of the various parallel models, e.g. the overhead of creating a new thread, or the latency and bandwidth of message-passing using MPI.
- Kernel benchmarks will be written that implement one or more common communications patterns using a variety of parallel models.
- We will collect a set of genuine parallel applications. Each application is likely to be only available for a single parallel model. However, we hope that we will have applications using a wide variety of models. All parallel applications will also be available in serial form.

The current suite, instrumentation classes, and a more comprehensive set of results, are available at `http://www.epcc.ed.ac.uk/javagrande/`. We would strongly welcome use of, and comments on, this material from developers both of Grande applications and of Grande environments.

8 Conclusions

We have presented a methodology for, and initial implementation of, a suite of benchmarks for Java Grande applications. We have set out criteria which we believe such a suite should meet, and have demonstrated how we have met these criteria. We have discussed methods of benchmark instrumentation, and have presented the instrumentation API. Sample results show that the suite gives some useful and meaningful insight into the performance of Java environments. Finally, we have discussed the future of the benchmark suite in terms of parallel benchmarks.

Acknowledgements

We wish to thank the following who have contributed benchmarks to the suite: Jesudas Mathew and Paul Coddington of the University of Adelaide, Roldan Pozo of NIST, Florian Doyan, Wilfried Klauser and Denis Caromel of INRIA, Hon Yau, formerly of EPCC, Gabriel Zachmann of the Fraunhofer Institute for Computer Graphics, Reed Wade of the University of Tennessee at Knoxville, John Tromp of CWI, Amsterdam and David Oh, formerly of MIT's Computational Aerospace Sciences Laboratory.

This work was funded by the UK High Performance Computing Initiative, EPSRC grant GR/K42653.

References

[1] Ashworth, M. (1996) *OCCOMM Benchmarking Guide. Version 1.2*, Daresbury Laboratory Technical Report, available from `http://www.dl.ac.uk/TCSC/CompEng/OCCOMM/uguide-1.2.ps`

[2] Bell, D. (1997) *Make Java fast: Optimize!*, JavaWorld, vol. 2, no. 4, April 1997, `http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html`

[3] Caromel, D., F. Doyon, W. Klauser and J. Vayssiere, *A distributed raytracer for benchmarking Java RMI and Serialization*, `http://www.inria.fr/sloop/C3D/`

[4] Dongarra, J. J. (1998) *Performance of Various Computers Using Standard Linear Equations Software (Linpack Benchmark Report)*, University of Tennessee Computer Science Technical Report, CS-89-85.

[5] Dongarra, J. J. and R. Wade, *Linpack benchmark: Java version*, `http://www.netlib.org/benchmark/linpackjava/`

[6] Hockney, R. W. (1992) *A Framework for Benchmark Performance Analysis*, Supercomputer, vol. 48, no. IX(2), pp. 9-22.

[7] Getov, V.S. *The ParkBench single-processor low-level benchmarks in Java*, available from `http://perun.hscs.wmin.ac.uk/CSPE/software.html`

[8] Griswold, W. and P. Phillips, *UCSD Benchmarks for Java*, `http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html`

[9] Hardwick, J. *Java Microbenchmarks*, `http://www.cs.cmu.edu/~jch/java/benchmarks.html`

[10] Jacob, M., M. Philippsen and M. Karrenbach, (1998) *Large-scale parallel geophysical algorithms in Java: a feasibility study*, Concurrency: Practice and Experience, vol. 10, nos. 11–13, pp. 1143-1154.

[11] Lai, X., J. L. Massey and S. Murphy (1992) *Markov ciphers and differential cryptanalysis*, in Advances in Cryptology—Eurocrypt '91, pp. 17–38, Springer-Verlag.

[12] Openshaw S., and J. Schmidt (1997) *A Social Science Benchmark (SSB/1) Code for Serial, Vector and Parallel Supercomputers*, Geographical and Environment Modelling, vol. 1, no. 1, pp. 65–82.

[13] Pendragon Software Corp. *Caffeine Mark 3.0*, `http://www.pendragon-software.com/pendragon/cm3/`

[14] Pozo, R. *Java SciMark benchmark for scientific computing*, `http://math.nist.gov/scimark`

[15] Richter, H. *BenchBeans: A Benchmark for Java Applets*, `http://user.cs.tu-berlin.de/~mondraig/english/benchbeans.html`

[16] Roulo, M. (1998) *Accelerate your Java apps!* JavaWorld, vol. 3, no. 9, Sept. 1998, available from `http://www.javaworld.com/javaworld/jw-09-1998/jw-09-speed.html`

[17] SPEC, *SPEC JVM98 Benchmarks*, `http://www.spec.org/osg/jvm98/`

[18] Trom, J. *The Fhourstones 2.0 Benchmark*, `http://www.cwi.nl/~tromp/c4/fhour.html`

[19] Zachmann, G. *Java/C++ Benchmark*, `http://www.igd.fhg.de/~zach/benchmarks/java-vs-c++.html`

[20] ZDNet, *JMark 2.0*, `http://www.zdnet.com/zdbop/jmark/jmark.html`