A Benchmark Suite for High Performance Java

J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty and R. A. Davey
Edinburgh Parallel Computing Centre, 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 goals. 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) that aims to address these issues and in so doing promote the use of Java in this area. This paper describes work carried out by Edinburgh Parallel Computing Centre (EPCC) 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 that 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 that 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 micro-benchmarks [2, 7, 8, 9, 16] that 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] that consist of a single, near full-scale, application. These are useful in that they are 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 difficult for the user community to know exactly what is being tested without resorting to use of a Java de-compiler. Furthermore, distributing byte code prevents the Java platform benefiting from analysis and optimisation at the source code to byte code compilation stage.

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 that 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 the suite should not be sensitive to factors that 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 platforms 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 benchmark types, 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 that will ultimately determine the performance of real Java applications. 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 terminal 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 chose the kernels and applications from a range of disciplines, that 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. Micro-benchmarks 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 that 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 to ensure 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 that 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 that 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 ratio 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. Note that for both temporal and relative performance, higher numbers indicate better performance.

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, that 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 that 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 static 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 a common structure to some extent by sub-classing the lowest level of the main hierarchy in the benchmark, and implementing a defined `interface`, that includes a 'run' method. We can then create a separate `main` class that creates an instance of this sub-class and calls its 'run' method. It is then straightforward to create a `main` that, 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 that `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 Version 2.0 release, that together with the 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.

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 variables. 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

```
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

different one. Performance units are assignments per second.

Cast Measures 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.

Create Measures 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 Measures loop overheads, for a simple for loop, a reverse for loop and a while loop. Performance units are iterations per second.

Serial Measures 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. Performance units are 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 that 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 Measures the performance of method calls. 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. 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 units are items per second. Memory and integer intensive.

SOR Computes 100 iterations of successive over-relaxation on an $N \times N$ grid. Performance units are 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 Computes a one-dimensional Fourier transform of N complex numbers. Performance units are samples per second. Exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions.

Sparse Multiplies an $N \times N$ sparse matrix stored in compressed-row format by a dense vector 200 times. Performance units are iterations per second. Exercises indirect addressing and non-regular memory references.

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 units are 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 units are 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 units are timesteps per second.

Search Solves a game of connect-4 on a 6 x 7 board using an 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. Performance units are positions per second.

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. Performance units are 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:

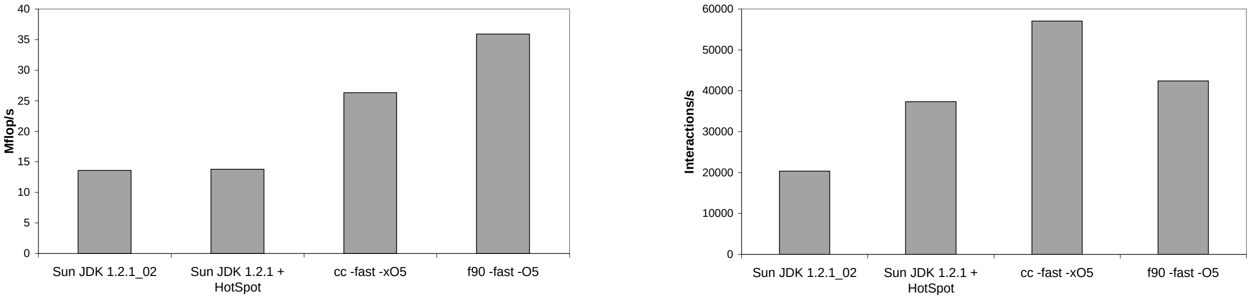
- Sun JDK Version 1.2.1_02 (production version)
- Sun JDK Version (reference version) + Hotspot version 1.0

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 are language comparisons, that is, 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. However, we intend the parallel part of the suite to contain versions of well-known Fortran and C parallel benchmarks, thus facilitating further inter-language comparisons. It is worth noting that the Fortran and C versions of these benchmarks do not attempt to implement any of the additional run-time checking operations (such as array bounds checking) implicit in the Java version.

Measurements have been taken for the LUFact Benchmark (on a 1000 × 1000 problem size) and the MolDyn 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. The results are shown in Figure 3. For the LUFact code, both JVMs give performance that 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.



(a) (b)

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

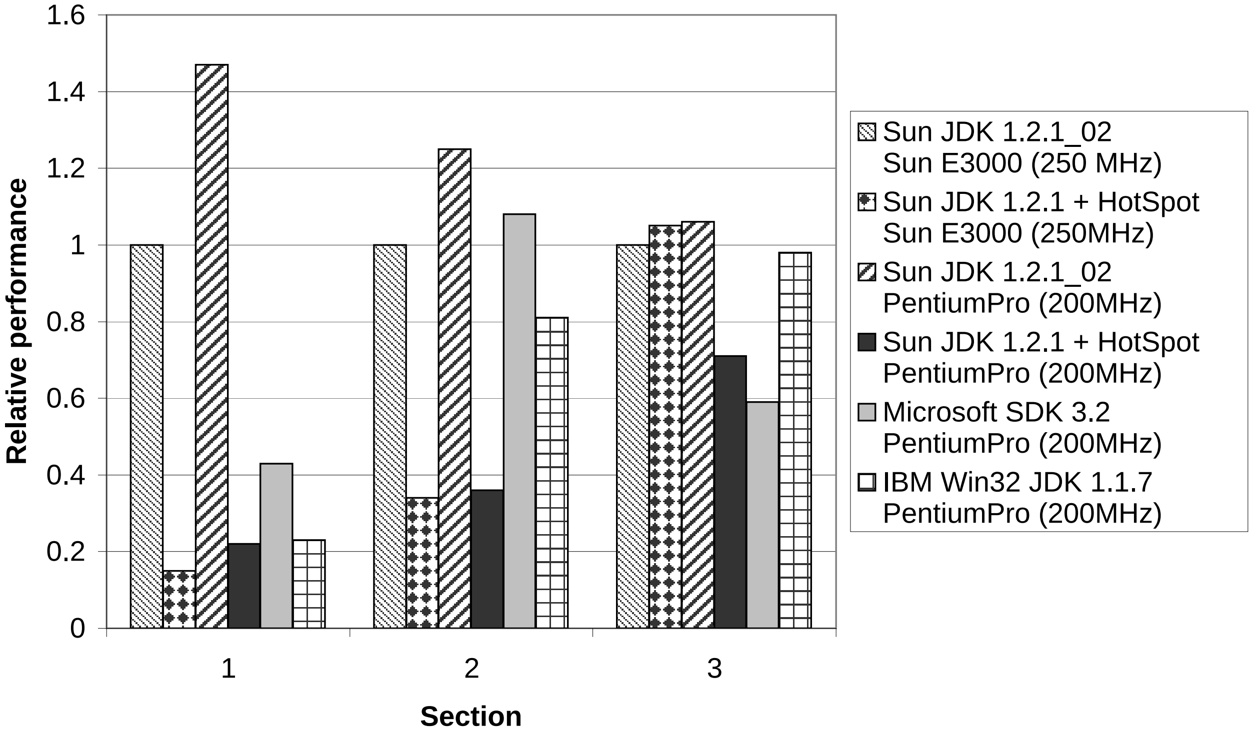


Figure 4: Relative performance of JDKs on Section 1, 2 and 3