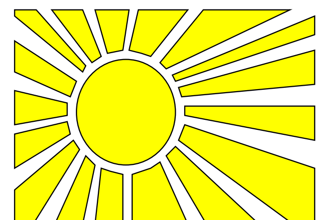
EPCC-SS99-06

Developing a parallel benchmarking suite for Java Grande applications

Alexandros Karatzoglou

Abstract

A *Grande* application is one which requires large amounts of processing, I/O, network bandwidth, or memory. *Grande* applications include computational science and engineering codes, as well as large scale database applications, business and financial models. Increasing interest is being shown in the use of Java for *Grande* applications. The purpose of developing a parallel benchmarking suite is to provide ways of measuring and comparing alternative Java parallel execution environments in ways which are important to *Grande* applications.



1 Introduction

1.1 Grande Applications

A *Grande Application* is defined [3] as an application of large-scale nature, potentially requiring large amounts of processing power, network bandwidth, I/O, and memory. *Grande* applications often rely on parallel execution environments to provide the necessary computational power. Several categories, can be used to describe *Grande Applications*

- High Performance Network Computing
- Scientific and Engineering Computations
- Distributed Modelling and Simulation (as in DoD DMSO activities)
- Parallel and Distributed Computing
- Data Intensive Computing
- Communication and Computing Intensive Commercial and Academic Applications
- Computational Grids (e.g., Globus and Legion)

Examples of *Grande Application* are:

- **Commercial:** Data-mining, Financial Modelling, Oil Reservoir Simulation, Seismic Data Processing, Vehicle and Aircraft Simulation
- **Government:** Nuclear Stockpile Stewardship, Climate and Weather prediction, Satellite Image Processing, Forces Modelling
- **Academic:** Fundamental Physics (particles, relativity, cosmology), Biochemistry, Environmental Engineering, Earthquake Prediction
- **Cryptography:** The recent DES-56 challenge presents an interesting Grande Application.

1.2 Java in Large Scale Applications

With the increasing popularity of Java comes a growing range of uses for the language that fall well outside its original design specifications. Increasing interest is being shown in the use of Java for *Grande* applications.

Java has the greatest potential to deliver an attractive productive programming environment spanning the very broad range of tasks needed by the Grande programmer. Java promises a number of breakthroughs that have eluded most technologies thus far. Specifically, Java has the potential to be written once and run anywhere. This means, from a consumer standpoint, that a Java program can be run on virtually any conceivable computer available on the market. While this could be argued for C, C++, and FORTRAN, true portability has not been achieved in these languages, save by expert-level programmers.

Java also offers a series of advanced programming features like ,object orientation, built in support for multi-threading and network programming and easy of use. All these features make the language suitable for use in *Grande* applications. Despite concerns about performance and numerical definitions an increasing number of users are using Java for large scale codes.

1.3 The Benchmark Suite

The aim of this work is to develop a standard parallel benchmarking suite which can be used to :

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

1.4 Related work

A considerable number of serial benchmarks and performance tests for Java are available. 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 in parallel execution environments. Of more interest are the benchmarks [1] 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 in parallel environments.

Other sets of benchmarks, from both academic and commercial 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 which 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.

Few benchmark codes attempt inter-language comparison. In those that do, the second language is usually C++, and the intention is principally to compare the object oriented features.

2 The Benchmarking suite

2.1 Characteristics

For a benchmark suite to be successful, 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 stretch Java environments in terms of CPU load, memory requirements, 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 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.

2.2 The Benchmark Sections

The Benchmarking Suite consists of three types of benchmarks, reflecting the classification of existing benchmarks : low level parallel operations (Section I), simple kernels (Section II) and applications (Section III).

- **Section I :** The low-level parallel operation benchmarks have been designed to test the performance of the low level parallel operations which will ultimately determine the performance of real applications running under the Java parallel environment. Examples include barrier operations, Join operations, calling synchronised methods and starting threads .
- **Section II :** The kernel benchmarks are chosen to be short codes, executed in parallel, (multi-threaded) each containing a type of computation likely to be found in Grande applications, for example: LU Factorisation, matrix multiplication.
- **Section III :** The application benchmarks are intended to be representative of Grande applications, decomposed and suitably modified for inclusion in the benchmark suite by removing any I/O and graphical components.

The suite is robust,because it avoids dependences on particular data sizes by offering a range of data sizes for each benchmark in Sections II and III. It also takes 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. 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.

For maximum portability, as well as ensuring adherence to standards, there are no graphical components 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 the suite is restricted to simple file I/O.

For standardisation a JGFBenchMark class is used in all benchmark programs. Transparency is achieved by distributing the source code for all the benchmarks. This removes any ambiguity in the question of what is being tested.

2.3 Performance Metrics

The performance metrics for the benchmarks are represented in two forms: execution time and temporal performance [1]. 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, the System.currentTimeMillis method from the Java.lang package is used. 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 is defined in units of operations per second, where the operation is chosen to be the most appropriate for each individual benchmark. For example, choosing floating point operations are appropriate linear algebra benchmark, but this would be inappropriate for a Fourier analysis benchmark which relies heavily on calls to transcendental functions. For some benchmarks, where the choice of most appropriate unit is not obvious, there are more than one operation units. For the low-level benchmarks (Section I) only temporal performance is reported. For other benchmarks (Sections II and III) both execution time and temporal performance are reported.

2.4 The JGF Benchmark API

The API developed by the **EPCC** for the benchmark class is described below [1]:

```
public class JGFBenchMark{

//No constructor

//Class methods

public static syncronized void addTimer(String Name);

public static syncronized void addTimer(String Name,String
opname);

public static syncronized void add Timer(String Name);

public static syncronized void stopTimer(String Name);

public static syncronized void addOpsToTimer(String Name, douple
Count);

public static syncronized double readTimer(String Name);

public static syncronized void resetTimer(String Name);

public static syncronized void printTimer(String Name);

public static syncronized void printperfTimer(String Name);

public static syncronized void storeData(String Name, Object
obj);

public static syncronized void retrieveData(String Name, Object
obj);
```

```
public static syncronized void printHeader(int section, int
size);

}
```

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 is 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.

The use of an interface to standardise the form of the benchmark is shown below.

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

The interface for the Section II benchmarks is shown here. The interface 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 JGFBenchmark class methods can be made either from these methods, or from methods in the existing code, as appropriate.

3 The Benchmarks

3.1 Java Threads

The decomposition of the Benchmarks code was done by the use of Java native *threads* [2] [4]. The individual *threads* cary out equal part of the computation and are equally distributed between processors on shared memory machines by the operating system. In writing the parallel benchmarking suite the following approach has been taken: The code to be executed by each thread is separated into a class which implements a Runnable interface. The run() method of the class is were the execution of the threads code starts. Other methods required are included in the class. A typical example of such a class follows.

```
  class BenchRunner implements Runnable {
```

```
int id;

    public BenchRunner(int id){
this.id=id;
        }

    public void run() {
for(int i=L;i<H;i++)
calculate();
        }

   public double calculate(){
        }

 }
```

In order to create a separate thread, an instance of the BenchRunner class is needed. The created object is then passed to constructor of the Thread class. For example:

```
    public class JGFBench {
      public static void main {
   Runnable rn = new BenchRunner(1);
   Thread th = new Thread(rn);
   th.start;
      }
    }
```

The new *thread* object's start() method is called to begin execution of the new *thread*. The id and other data needed by the *thread* are passed to it. Parameters (arrays, variables, objects, etc) passed to the thread are passed by reference and treated as shared data among the *threads*. Public static class variables are also considered as shared data and used in some benchmarks as reduction variables. The threads own class and method variables are private data of the *thread*.

3.2 Section I Benchmarks

The Section I benchmarks measure the performance of *low level parallel operations*. They are simple parallel operations repeated long enough to be appropriately timed.

Thread Measures the time spend to start a thread. Performance measures are in threads per second. In this benchmark we start a number of threads measure the time spend and divide the total number of threads started by the time spend.

Join Measures the time spend to Join (stop) the threads. Performance measures are in Join operations per second. In this benchmark threads are started and joined a number of times, only the time spend to Join the threads is actually measured.

Barrier Measure the performance of a barrier operation. Performance measures are in Barrier

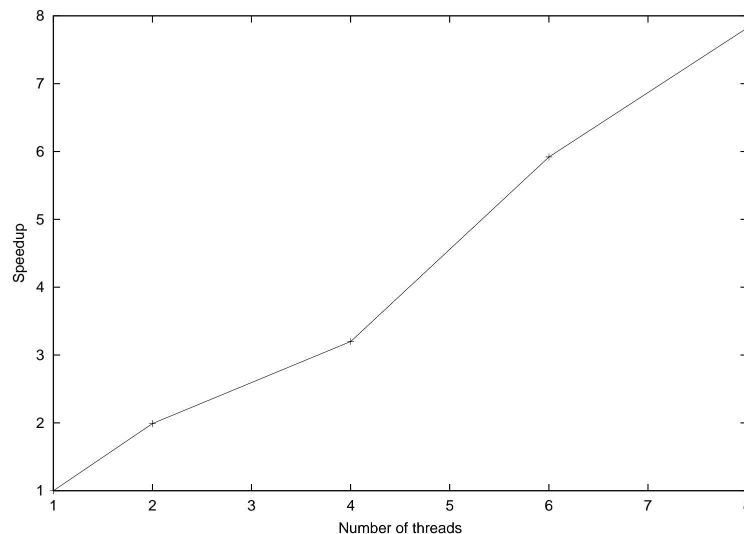
operations per second. In this benchmark the threads call repeatedly a barrier and the time spend by the threads is measured.

Synchronised A number of calls to a synchronised method from several threads is compared to the same total number of calls from one thread to a non-synchronised method. Performance measures are in calls per second.

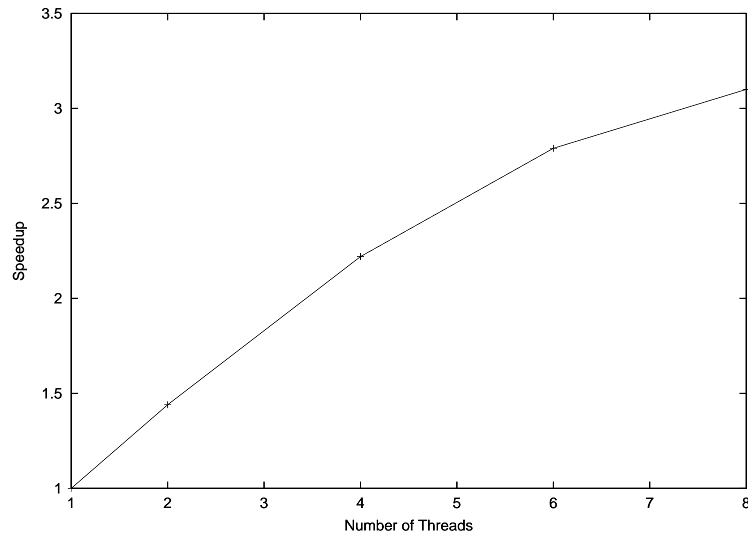
3.3 Section II

The benchmarks in section II are kernel benchmarks each containing a computation likely to be found in Grande Applications. The decomposition of these benchmarks which originally where serial (JavaGrande Benchmarking Suite v2.0) is mainly based in spiting the main loop of the program equally among the threads and using barriers and synchronised statements wherever needed. The speedup graphs are produced by running the small size (A) benchmarks on an 8 CPU SUN E3500 machine using the 1.1.6 SUN SDK.

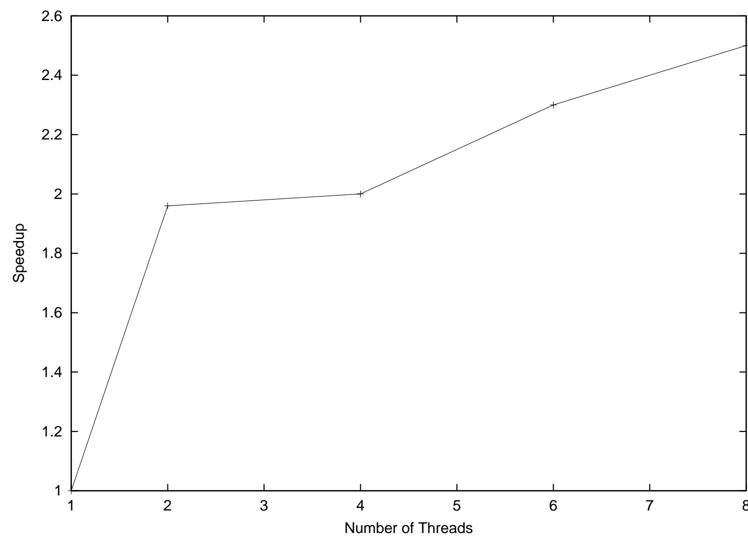
Series Series computes the first N Fourier coefficients of the function $f(x) = (x + 1)^x$ on the interval (0,2). Performance units are coefficients per second. This benchmark heavily uses transcendental and trigonometric functions. A graph showing the speedup of the parallel version of the code is illustrated below.



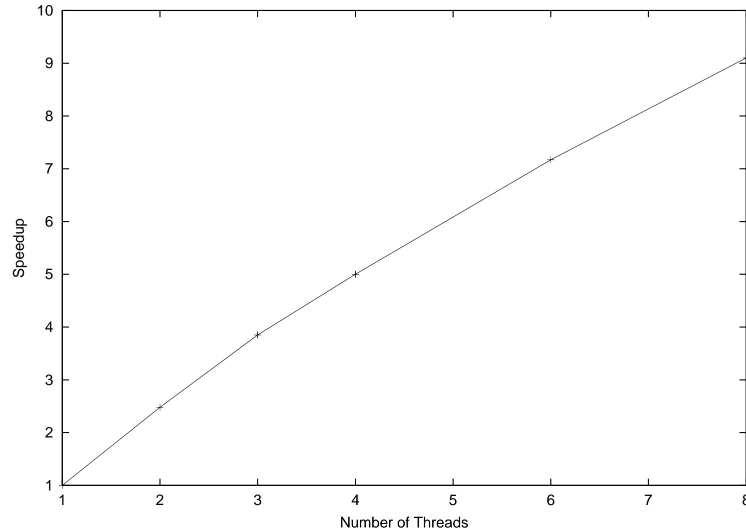
LU Factorisation LU Factorisation solves an NxN linear system using LU factorisation followed by a triangular solve. This is a Java version of the well known Linpack benchmark. Performance units are Mflops per second. This benchmark is memory and floating point intensive. This benchmark was only partially decomposed. The Gaussian elimination is part of the computation performed and is decomposed, the rest of the computation is inherently serial. The speedup graph is shown below.



Crypt Crypt Performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes. Performance units are bytes per second. Bit/byte operation intensive. The speedup of the code is shown below. The code scales bad probably because of memory conflicts.



SOR The SOR benchmark performs 100 iterations of successive over-relaxation on a NxN grid. The performance reported is in iterations per second.The speedup of the code is shown below. The code is scaling super-linear probably because of cache effects.

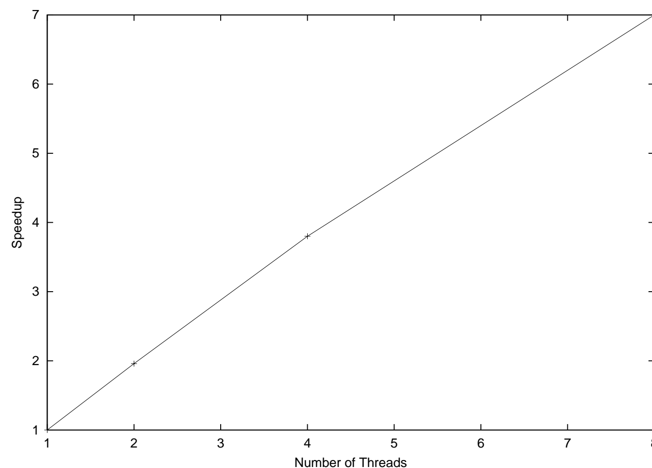


Sparse This benchmark uses an unstructured sparse matrix stored in compressed-row format with a prescribed structure. This kernel exercises indirection addressing and non-regular memory references. A N x N sparse matrix is used for 200 iterations. The speedup of this code is about 1.2 on all number of processors probably heavy memory access..

3.4 Section III

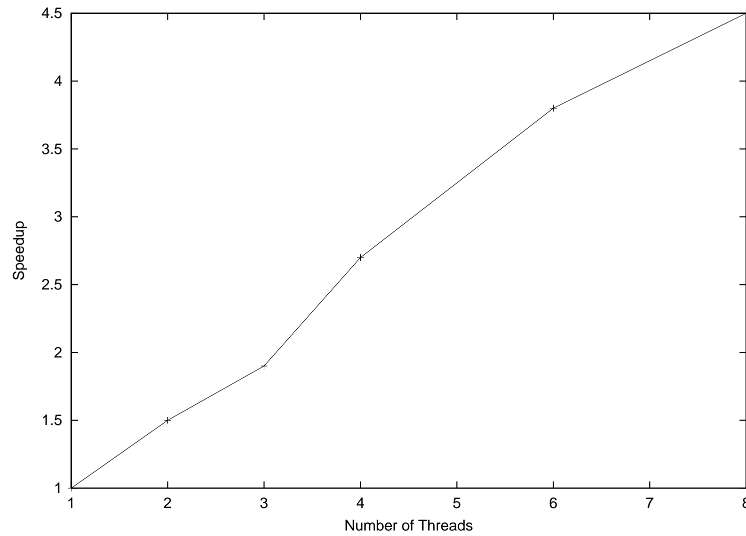
The section III benchmarks are application benchmarks intended to be representative of Grande applications, suitably modified for inclusion in the benchmark suite by removing any I/O and graphical components. The same approach as in the section II benchmarks has been taken into decomposing the existing code.

Monte Carlo This benchmarks is 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. The speedup of the code is shown below.



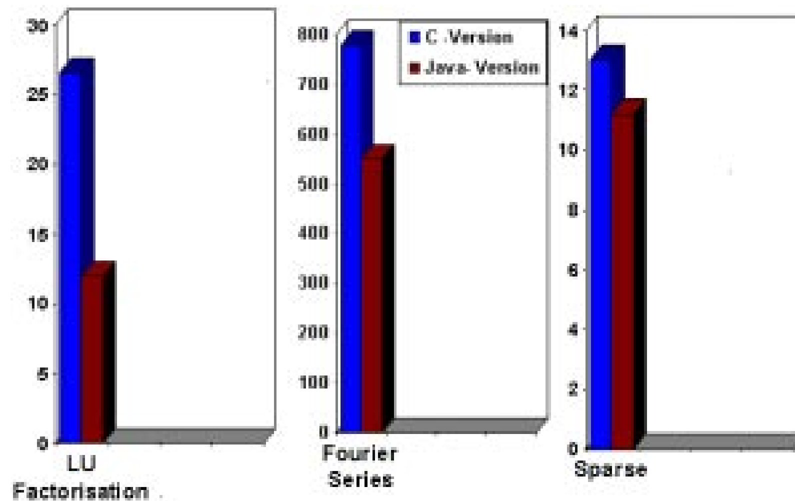
Ray Tracer This benchmark measures the performance of a 3D ray tracer. The scene rendered

contains 64 spheres, and is rendered at a resolution of NxN pixels. The performance is measured in pixels per second.



4 Comparing Java with C

Part of this project was also the translation of some Java benchmarks into C in order to compare the performance of the two languages. The Series and the Sparse benchmark was translated into C. The LU Factorisation (Linpack) already existed in C. The results are shown in figure X. The Java code was run on the JDK 1.1.6 using optimisation (-O) and the C code was compiled with the native Solaris compiler (cc) using optimisation -xO4.



5 Conclusions

The creation of the parallel benchmarking suite demonstrates the use of Java in Grande applications. The Java language proved to be easy to use in writing parallel code. The performance

comparison between C and Java shows that the Java has still potential for improvement. In particular the floating point performance of the language needs to be improved.

References

[1] J.M. Bull, L A. Smith, M. D. Westhead, D.S. Henty, and R. A. Davey. A Methodology for Benchmarking Java Grande Applications. EPCC, June 1999.

[2] Scott Oaks and Henry Wong. Java Threads. O'Reilly, January 1997.

[3] Java Grande Forum Panel. Making Java Work for High end Computing. www, November 1998.

[4] SUN. http://java.sun.com. www, August 1999.

Alexandros Karatzoglou is studying Physics at the Aristotle University of Thessaloniki Greece. Currently he is working on his diploma thesis on Neural Networks at the Katholieke Universiteit Leuven in Belgium.

Supervisors: Lorna Smith, Douglas Smith